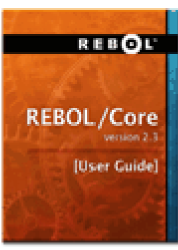


REBOL/Core Users Guide

For REBOL Version 2.3

Copyright 2005 REBOL Technologies
Коментарии/замечания к переводу pochinok@bk.ru

1	**Введение**	Введение в REBOL/Core, информация о руководстве, информация о технической поддержке и куда отправлять комментарии.
2	**Подготовка**	Установка, запуск и завершение сеансов REBOL. Использование консоли REBOL. Получение помощи от REBOL. Сообщения об ошибках. Как обновить REBOL.
3	**Краткий обзор**	Краткий обзор, в котором описаны значения, слова, блоки, переменные, оценка, функции, пути, объекты, сценарии, файлы и сеть.
4	**Выражения**	Как оцениваются блоки, значения и слова. Условные, повторяющиеся и выборочные выражения. Остановка оценки. Восстановление после ошибки.
5	**Скрипты**	Заголовки скрипта. Аргументы командной строки для скриптов. Загрузка, сохранение и комментирование скриптов. Руководство по хорошему стилю написания сценариев.
6	**Серии**	Серии - основа REBOL. Описание функций серий и типов данных. Создание и копирование серий. Итерация серии. Поиск и сортировка серий. Серии как наборы данных.
7	**Серии блоков**	Особенности серии блоков. Блоки блоков. Пути для вложенных блоков. Массивы. Составные блоки.
8	**Серии строк**	Специальные строковые функции и преобразование значений в строки.
9	**Функции**	Оценка функций и аргументов. Определение функций. Вложенные, безымянные и условные функции. Атрибуты функции. Объем переменных. Отражающие свойства. Онлайн-справка по функциям. Просмотр исходного кода функции.
10	**Объекты**	Создание и клонирование объектов. Доступ к объектам. Ссылаясь на себя. Инкапсуляция. Отражающие свойства.
11	**Математика**	Скалярные типы данных. Порядок оценки. Стандартные функции и операторы. Преобразование типов. Сравнение. Функции журнала, триггера и логики.
12	**Файлы**	Имена и пути файлов. Чтение и запись файлов. Преобразование строк и блоки строк. Доступ к каталогам и функции файлов каталога.
13	**Сетевые протоколы**	О сети REBOL. Начальная настройка. DNS, Whois, Finger Daytime, HTTP, SMTP, POP, FTP, NNTP, CGI, TCP и UDP.
14	**Порты**	О портах ввода/вывода. Открытие, чтение, запись, закрытие портов. Обновление и ожидание портов. Другие режимы порта. Права доступа к файлам. Каталог портов.
15	**Парсинг**	Разбиение строк. Грамматические правила. Пропуск ввода. Типы соответствия. Рекурсия и оценка.
A1	**Значения**	Обзор типов данных в Rebol.
A2	**Ошибки**	Сообщения об ошибках в Rebol, категории ошибок, перехват ошибок, объекты ошибок, ошибки генерируемые пользователем.
A3	**Консоль**	Командная строка. История ввода и повторный ввод. Индикатор занятости. Расширенные консольные операции.
C1	**Изменения**	Дополнения к этому документу для последующих версий 2.3.0-2.5.X

Copyright REBOL Technologies. All Rights Reserved.

Глава 1. Введение

Содержание:

1. О REBOL
2. О этом руководстве
  2.1 Рекомендации для новых программистов
  2.2 Рекомендации для опытных программистов
3. Условные обозначения
4. Техническая поддержка
  4.1 Новости и информация для разработчиков
  4.2 Обсуждения и форумы
  4.3 Отчеты об ошибках и запросы на улучшения
  4.4 Библиотека сценариев REBOL.org
  4.5 Новые Альфа- и Бета-версии
5. Комментарии приветствуются

1. О REBOL

Несколько фактов о REBOL:

- **REBOL** означает язык объектов на основе относительных выражений (Rebol -- акроним: **Relative Expression-Based Object Language**).
- **REBOL произносится, как "реб-ел"**, почти как рабби.
- **REBOL - это язык обмена сообщениями.** Его основная цель - предоставить лучший доступ к распределенным вычислениям и коммуникациям.
- **REBOL был разработан Карлом Сассенратом**, архитектором операционной системы, ответственным за Amiga OS, первую в мире многозадачную операционную систему для персональных компьютеров.
- **REBOL - это больше, чем просто язык программирования.** Это также язык для представления данных и метаданных. Он предоставляет единый метод вычисления, хранения и обмена информацией.
- **Код и данные REBOL охватывают более 40 системных платформ.** Сценарий, написанный для Windows, одинаково хорошо работает на Linux, UNIX и многих других платформах ... без каких-либо изменений.
- **REBOL представляет концепцию диалектирования** - маленьких, эффективных, предметно-ориентированных "под" языков для кода, данных и метаданных.
- **REBOL намеренно поддерживается маленьким размером** несмотря на наличия множества функций, типов данных, встроенной помощи, множественным интернет-протоколам, сжатием, системой ошибок, консолью отладки, шифрованием и многому другому.
- **Программы на REBOL легко писать**. Все, что вам нужно, это текстовый редактор. Программа может быть размером в одну строку или большим, целым приложением, состоящим из нескольких десятков/сотен/тысяч файлов.
- **REBOL/Core служит основой для всей технологии REBOL.**Хотя этот язык разработан, чтобы быть простым и продуктивным для новичков, он откроет много нового и для профессионалов.

Графическая версия REBOL, называется REBOL/View и основана на REBOL/Core. Она так же может быть загружена с официального сайта REBOL .

2. Об этом руководстве

В этом руководстве представлена основная информация, необходимая для использования REBOL/Core. Предполагается, что читатель уже знаком с общей терминологией и концепциями программирования и операционной системы..

2.1 Рекомендации для новых программистов

Если вы новичок в программировании, REBOL - отличный способ начать.

REBOL повсеместно использует несколько общих концепций. Например, концепция *серии* используется во всем, от структур данных до блоков кода. Как только вы изучите концепции и методы построения серий, их можно будет применять в ваших программах. Вам следует хорошо изучить эти концепции. Это окупится позже. Главы в этом руководстве пользователя упорядочены, чтобы помочь вам начать работу.

Если у вас возникли проблемы с использованием REBOL, не расстраивайтесь. Есть много людей, которые могут вам помочь. Список рассылки REBOL (см. Раздел поддержки ниже) отслеживают сотни людей, которым нравится помогать новичкам начать работу. Не стесняйтесь посещать этот форум по любой причине.

2.2 Рекомендации для опытных программистов

Если вы уже знакомы с другими языками программирования, такими как C, C ++, Java, Pascal, Python, PERL, BASIC и т.д., мы должны вас предупредить: **REBOL - это совсем другое**.

REBOL был разработан не только для того, чтобы отличаться, но, скорее, для того, чтобы дать программистам большую выразительность. Программисты, освоившие REBOL, считают, что лучше всего забыть то, что вы уже знаете о других языках. Это потому, что вы не создаете программы REBOL таким же образом. Конечно, вы можете создавать программы REBOL, похожие на C, но если вы это сделаете, вы потеряете множество преимуществ, которые предлагает REBOL.

В технических определениях, REBOL -- высоко рефлективный, функциональный, символический язык с описываемой областью правил. Если вы не знаете, что это означает -- не стоит беспокоиться. REBOL вобрал в себя всё самое продвинутое из компьютерной науки, но для его использования вовсе не обязательно быть учёным от компьютерной отрасли..

Как опытный программист, у вас может возникнуть соблазн пропустить большинство глав этого руководства. По большей части это нормально. Однако такие концепции, как *серии*, имеют решающее значение для понимания REBOL. Если вы не потратите время на освоение таких концепций, вам будет сложно по-настоящему свободно говорить на языке REBOL..

3. Условные обозначения

В следующей таблице описаны типографские обозначения, используемые в этом руководстве.

Вещь	Конвенция	Пример
Слова, которые являются частью языка REBOL (как имена функций, специальные переменные, системные объекты).	Жирный, зелёный, моноширинный	**Добавить при изменении**
Слова и значения, не относящиеся к REBOL, такие как имена файлов, имена каталогов, имена программ и имена переменных.	Зелёный, моноширинный	myfile window-color
Примеры кода.	Прямоугольник, жирный, моноширинный	**do %feedback.r**
Результаты выводимый в консоли REBOL.	Прямоугольник, синий, моноширинный	true

4. Техническая поддержка

Чтобы задать общие вопросы или оставить отзыв о продукции REBOL или нашем веб-сайте, посетите нашу страницу отзывов. Обычно мы отвечали на сообщения в течение 24–48 часов. Не забудьте указать свой адрес электронной почты, если хотите получить ответ.

4.1 Новости и информация для разработчиков

Сеть разработчиков REBOL предоставляет последние технические новости, информацию, документацию, обсуждения, выпуски и многое другое.

На этом сайте также находится блог Карла REBOL , информативный архив идей, идей и запросов изобретателя и основателя REBOL Карла Сассенрата.

Список рассылки REBOL REBOL Talk Forum Это независимый интернет-форум для открытых обсуждений REBOL. REBOL Группа Google Эта новая группа обсуждения в Интернете и по электронной почте недавно была запущена в группах Google. Это все еще находится на экспериментальной стадии использования. 4.3 О Клиенты, разработчики и пользователи REBOL теперь могут напрямую искать информацию, связанную с известными проблемами, сообщать о новых проблемах или запрашивать улучшения, используя нашу базу данных RAMBO . 4.4 Библиотека скриптов REBOL.org Сайт www.REBOL.org . 4.5 Новые альфа- и бета-версии Мы регулярно публикуем промежуточные сборки для наших продуктов. Эта услуга предназначена только для клиентов и опытных разработчиков. Эти страницы содержат необработанные сборки, а не пакеты распространения. 5. Комментарии приветствуются Чтобы помочь нам с будущими выпусками этой документации, мы хотели бы знать, какие исправления или уточнения вы найдете полезными. Отправляйте их на страницу обратной связи на нашем сайте. Пожалуйста, укажите название, версию и главу руководства.

4.2 Обсуждения и форумы

- **REBOL Mailing List**
  Список рассылки REBOL - это форум для вопросов и ответов по всем темам, связанным с REBOL. Вы можете просмотреть предыдущие сообщения в архиве списка рассылки на REBOL.org.
- **REBOL Talk Forum**
  Это независимый интернет-форум для открытых обсуждений.
- **REBOL Google Group**
  Эта новая группа обсуждения в Интернете и по электронной почте недавно была запущена в группах Google. Это все еще находится на экспериментальной стадии использования.
- **Другие обсуждения**
  REBOL Technologies также поддерживает различные частные дискуссионные группы, использующие нашу технологию IOS и систему обмена сообщениями AltME.

4.3 Отчёты об ошибках и запросы на улучшение

Клиенты, разработчики и пользователи REBOL теперь могут напрямую искать информацию, связанную с известными проблемами, сообщать о новых проблемах или запрашивать улучшения, используя нашу базу данных **RAMBO**.

4.4 Библиотека скриптов REBOL.org

Сайт **www.REBOL.org** - это сайт, управляемый сообществом, который предоставляет обширную библиотеку примеров REBOL. Он также включает в себя множество руководств, а также архив сообщений списка рассылки REBOL.

4.5 New Alpha and Beta Releases

Мы регулярно публикуем промежуточные сборки для наших продуктов. Эта услуга предназначена только для клиентов и опытных разработчиков. Эти страницы содержат необработанные сборки, а не пакеты распространения.

5. Комментарии приветствуются

Что бы помочь нам с будущими выпусками этой документации, мы хотели бы знать, какие исправления или уточнения вы найдете полезными. Отправляйте их на страницу обратной связи на нашем сайте. Пожалуйста, укажите название, версию и главу руководства.

С замечаниями о переводе пишите на pochinok@bk.ru

Глава 2 - Подготовка

Содержание:

1. Установка REBOL
    - 1.1 Файлы дистрибутива
    - 1.2 Переменная домашнего окружения
    - 1.3 Настройка сети
    - 1.4 Настройки прокси и брандмауэра
    - 1.5 Лицензионное соглашение
2. Запуск REBOL
    - 2.1 из значка
    - 2.2 из оболочки
    - 2.3 из другого приложения
    - 2.4 Проблемы безопасности
    - 2.5 Аргументы программы
    - 2.6 Файл сценария (скрипты)
    - 2.7 Указание параметров
    - 2.8 Перенаправление файлов
    - 2.9 Аргументы сценария
    - 2.10 Файлы запуска
3. Выход из REBOL
4. Использование консоли
    - 4.1 Многострочный ввод
    - 4.2 Прерывание сценария
    - 4.3 Вызов истории
    - 4.4 Завершение слова
    - 4.5 Индикатор занятости
    - 4.6 Сетевые подключения
    - 4.7 Виртуальный терминал
5. Получение справки
    - 5.1 Интерактивная справка
    - 5.2 Просмотр исходного кода
    - 5.3 Загрузка документов
    - 5.4 Библиотека сценариев
    - 5.5 Список рассылки пользователей
    - 5.6 Связь с нами
6. Ошибки
    - 6.1 Сообщения об ошибках
    - 6.2 Перенаправление ошибок
7. Обновление

1. Установка REBOL

Установка REBOL занимает всего несколько секунд и очень проста, незаметна и не мешает работе.

Для REBOL/Core единственная процедура установки - это распаковать файлы дистрибутива и сохранить их в любом каталоге вашей системы.

Для других продуктов REBOL при установке может потребоваться предоставить дополнительную информацию, например, где хранить связанные файлы. См. Примечания к выпуску, включенные в файлы распространения.

1.1 Файлы дистрибутива

Дистрибутив REBOL/Core включает следующие файлы:

rebol (.exe)	Исполняемая программа, запускающая консоль REBOL.
rebol.r	Файл начальной загрузки системы. (Не требуется для запуска REBOL.)
setup.html	Информация о настройке и установке.
changes.html	Изменения в последних выпусках.
license.txt	лицензионное соглашение REBOL.

Другие файлы могут быть предоставлены в зависимости от продукта и версии REBOL.

1.2 Переменная домашнего окружения

Хотя это не обязательно, но если ваша операционная система устанавливает переменную среды HOME, REBOL будет использовать ее для поиска файлов запуска. Во многих операционных системах, таких как UNIX или Linux, переменная HOME уже может быть установлена по умолчанию. (Так что самому ставить не нужно).

1.3 Настройка сети

При первом запуске REBOL запрашивает информацию о сети. Эта информация не является обязательной. Для некоторых протоколов, например электронной почты или FTP, требуется адрес электронной почты или имя почтового сервера. Кроме того, если вы находитесь за брандмауэром или используете прокси-сервер, вам необходимо предоставить конкретную информацию для доступа в Интернет.

Чтобы настроить вашу сеть:

1. Введите свой адрес электронной почты. Например, name@example.com.
2. Введите имя вашего почтового сервера. Например: mail.example.com. Используйте имя почтового сервера, который вы обычно используете. Если вы не уверены в имени сервера, обратитесь к своему сетевому администратору или поставщику услуг Интернета, чтобы узнать имя вашего SMTP-сервера (электронной почты).
3. Укажите, используете ли вы прокси-сервер. Если вы напрямую подключены к Интернету через модем или Ethernet, введите N (нет). Если вы используете прокси или брандмауэр, предоставьте необходимую информацию, как описано в разделе «Параметры прокси и брандмауэра» ниже.

После ответа на вопросы запуска REBOL создает файл `user.r` и помещает в него ваши сетевые настройки. Вы можете изменить эти настройки в любое время, отредактировав этот файл.

1.4 Настройки прокси и брандмауэра

Часто организации используют брандмауэр или прокси-сервер для защиты доступа в Интернет и из него. Прежде чем REBOL сможет получить доступ к Интернету через эти системы, вам необходимо предоставить некоторую дополнительную информацию.

Чтобы предоставить информацию о прокси-сервере:

1. Когда REBOL спросит, используете ли вы прокси-сервер, ответьте утвердительно, набрав Y (да).
2. Введите имя вашего прокси-хоста. Это компьютер или брандмауэр в вашей сети, выступающий в качестве прокси.
3. Введите номер порта, используемый прокси-хостом для запросов прокси. Обычно это порт 1080, но он может отличаться. Если вы не знаете номер порта, проверьте настройки своего веб-браузера или обратитесь к администратору сети.

REBOL по умолчанию использует прокси-протокол SOCKS. Вы можете указать другой тип прокси, отредактировав файл `user.r` предоставив функции **`set-net`** соответствующий типу используемого прокси.

Поддерживаются следующие настройки:

```
socks   - использовать последнюю версию SOCKS (5)
socks5  - использовать socks5 прокси
socks4  - использовать socks4 прокси
generic - использовать общий прокси CERN
none    - не использовать прокси
```

Эти настройки предоставляются как шестой аргумент функции**set-net** содержащийся в файле `user.r`. Дополнительные сведения об изменении параметров прокси в файле `user.r` смотрите в главе «Сетевые протоколы» .

1.5 Лицензионное соглашение

Лицензионное соглашение с конечным пользователем REBOL, с которым вы согласились при загрузке или установке REBOL, можно просмотреть в любое время с консоли REBOL, введя `license` в командной строке REBOL.

2. Запуск REBOL

REBOL работает на большом количестве систем. REBOL запускается так же, как и другие приложения в вашей системе. В зависимости от конкретной операционной системы REBOL может быть запущен из одного или нескольких из следующих приложений: значка, командной оболочки или других приложений.

2.1 Из иконки

REBOL можно запустить, дважды щелкнув значок программы REBOL, или по ассациированому файлу сценария с расширением `.r`.

Если вы дважды щелкните значок программы, REBOL загрузится, отобразит консоль и отобразит приглашение.

Если вы хотите запустить REBOL со скриптом, вы можете сделать это следующими способами:

- Перетащите скрипт на значок программы
- Ассоциируйте файл со программой REBOL
- Создайте ярлык или псевдоним или ссылку с информацией о программе и сценарии.

См. Дополнительную информацию в руководстве по вашей операционной системе.

2.2 Из командной строки

Из командной строки оболочки перейдите в каталог, содержащий файл `rebol.exe` (или `rebol` на не-Windows системах), и введите rebol или ./rebol.

В некоторых операционных системах, таких как UNIX, вы можете создавать команды оболочки псевдонимы, которые могут запускать REBOL с набором аргументов и файлов. Кроме того, UNIX позволяет создавать сценарии оболочки, включающие путь и файл интерпретатора, например:

```
!#/path/to/rebol
```

в первой строке файла сценария. Когда вы вводите имя файла сценария в командной строке, UNIX запускает REBOL для выполнения сценария.

2.3 Из другого приложения

Для написания и отладки сценариев REBOL удобно настроить ваш любимый текстовый редактор для запуска REBOL и передать ему файл сценария, который вы редактируете. Каждый текстовый редактор делает это по-своему.
Например, в редакторе Premia Codewright вы можете использовать параметры компилятора языка для настройки REBOL. Укажите программу REBOL, а не компилятор. Затем вы можете нажать одну клавишу, которая сохраняет сценарий и оценивает его.

2.4 Проблемы безопасности

По умолчанию установлена защита, предотвращающая изменение сценариями любых ваших файлов или каталогов.

2.4.1 Безопасность порта

Функция **secure** (защита/безопасность) обеспечивает гибкость в настройке и управлении функциями безопасности REBOL. Текущие настройки безопасности возвращаются в результате вызова функции **secure**.

В настройках безопасности используется диалект REBOL, то есть язык внутри языка. Нормальный диалект состоит из блока парных значений. Первое значение в паре определяет, что "защищается":

file	определяет файловую безопасность.
net	определяет сетевую безопасность.

Имя файла или путь к каталогу позволяет указать уровни безопасности для конкретного файла или каталога.

Второе значение в паре определяет уровень безопасности. Это может быть слово "уровня безопасности" или "блок" слов. Слова уровня безопасности:

allow	разрешить доступ без ограничений.
ask	спросить разрешения, если потребуется какой-либо доступ.
throw	выдать ошибку, если потребуется какой-либо ограниченный доступ.
quit	закройть этот сеанс REBOL, если потребуется какой-либо ограниченный доступ.

Например, чтобы разрешить весь доступ к сети, но закрыть доступ к любому файлу:

```
secure [
    net allow ;разрешить любой сетевой доступ
    file quit ;любая попытка доступа к файлам завершит REBOL
]
```

Если вместо слова уровня безопасности используется блок, он может содержать пары уровней безопасности и типов доступа. Это позволяет указать более высокий уровень детализации требуемой безопасности. Разрешённые типы доступа:

read	контролирует доступ для чтения.
write	контролирует доступ на запись, удаление и переименование.
all	контролирует весь доступ.

Пары обрабатываются в том порядке, в котором они появляются, при этом более поздние пары изменяют эффект более ранних пар. Это позволяет установить один тип доступа без явной настройки всех остальных.

Например:

```
secure [
    net allow
    file [
        ask all
        allow read
    ]
]
```

Вышеуказанное устанавливает уровень безопасности, чтобы запрашивать(**ask**) все (all) операции, кроме чтения, которое должно быть разрешено. Этот метод также можно использовать для отдельных файлов и каталогов. Например:

```
secure [
    net allow
    file quit
    %source/ [ask read]
]
```

спрашивает, если сделана попытка прочитать каталог `%source` . В ином случае, он использует по умолчанию (**quit**).

Есть особый случай, когда функция **secure** принимает аргумент из одного слова, который должен быть одним из уровней доступа безопасности. В этом случае уровень безопасности для всей сети и доступа к файлам устанавливается на этот уровень.

```
secure quit
```

Функция **secure** также может принять значение **none**, что сделает доступ без каких - либо ограничений (так же, как **allow**).

```
secure none
```

Уровень безопасности теперь следующий:

```
secure [
    net allow
    file [
        ask all
        allow read
    ]
]
```

Если уровень безопасности не устанавливался, то "по умолчанию", в случаи необходимости выдаётся запрос **ask**. Текущие настройки безопасности `не будут` изменены, если при обработке блока аргументов произошла ошибка.

2.4.2 Предыдущие настройки безопасности

Функция **secure** теперь возвращает предыдущие настройки безопасности до того, как были сделаны новые настройки. Это блок с настройками глобальной сети и файла, за которыми следуют настройки файла или каталога. Слово **query** может быть использовано для получения настроек без изменения их.

```
current-security: secure query
```

Вы можете сохранить текущий уровень безопасности, запросив текущие настройки, изменив их, а затем использовать функцию **secure** для установки новых значений.

Понижение уровня безопасности вызывает запрос на изменение настроек безопасности. Исключение составляют случаи, когда сеанс REBOL работает в тихом режиме, который вместо этого завершает сеанс REBOL. При повышении уровня безопасности запрос не создается. Обратите внимание, что запрос безопасности теперь включает возможность разрешить весь доступ для оставшейся части обработки скриптов.

Запуск REBOL из оболочки аргумент **-s** эквивалентен:

```
secure allow
```

а аргумент **+s** эквивалентен:

```
secure quit
```

Теперь вы можете следовать аргументу **--secure** с одним из уровней безопасности доступа как для сети, так и для доступа к файлам:

```
rebol --secure throw
```

2.5 Аргументы программы

Существует ряд аргументов, которые можно указать в командной строке оболочки, в пакетном сценарии или в свойствах значка. Чтобы просмотреть аргументы и параметры, доступные для любой версии языка REBOL, введите `usage` в командной строке консоли.

```
Использование командной строки:

    REBOL <параметры> <сценарий> <аргументы>

Все поля являются необязательными. Поддерживаемые параметры:

    --cgi (-c)        Проверить ввод CGI
    --do expr         Выполнить выражение
    --help (-?)       Показать эту информацию об использовании
    --nowindow (-w)   Не открывать окно
    --noinstall (-i)  Не устанавливать (View)
    --quiet (-q)      Не печатать баннеры
    --reinstall (+i)  Принудительно устанавливать (View)
    --script file     Явно указывать сценарий
```

```
    --secure level   Установить уровень безопасности:
                     (allow ask throw quit)
    --trace (-t)     Включить режим трассировки
    --uninstall (-u) Деинсталировать/удалить REBOL (View)

Другие параметры командной строки:

    +q               Принудительно выключить quiet (-q) (View)
    -s               Нет контроля безопасности
    +s               Максимальный режим безопасности
    -- args          Предоставлять аргументы без сценария

Примеры:

    REBOL script.r
    REBOL script.r 10:30 test@domain.dom
    REBOL script.r --do "verbose: true"
    REBOL --cgi -s
    REBOL --cgi --secure throw --script cgi.r "debug: true"
    REBOL --secure none
```

Опять же, формат командной строки:

```
REBOL параметры сценарий аргументы
```

Где:

параметры	одна или более опции программы
сценарий	имя файла сценария, который вы хотите запустить. Если имя файла содержит пробелы, его следует полностью взять в кавычки.
аргументы	аргументы, переданные скрипту в виде строки. К этим аргументам можно получить доступ из сценария.

Все приведенные выше аргументы являются необязательными, и любая их комбинация разрешена.

Значки быстрого доступа

В некоторых операционных системах, таких как Windows или AmigaOS, вы можете создавать значки, в которых указать любой из вышеперечисленных параметров как часть имени исполняемого файла. Используя эту технику, вы можете создавать значки, которые напрямую выполняют сценарии REBOL с требуемыми параметрами.

2.6 Файл сценария

Обычно вы запускаете REBOL с именем файла сценария, который вы хотите, чтобы он выполнил. Допускается указание только одного файла со сценарием. Например:

```
REBOL script.r
```

Если имя файла содержит пробелы, его необходимо заключить в двойные кавычки.

2.7 Указание параметров

Опции программы обозначаются знаком плюс (+) или минус (-) перед символом или двойным тире (-) перед словом. Это стандартная практика для указания параметров программы в большинстве операционных систем.

Вот несколько примеров использования опций.

Чтобы запустить сценарий с параметром, например параметром `-s` , который выполняет сценарий с отключенной безопасностью, введите:

```
REBOL -s script.r
```

Чтобы получить информацию о параметрах REBOL, введите:

```
REBOL -?
REBOL --help
```

Чтобы запустить REBOL без открытия нового окна (это делается, когда вам нужно перенаправить вывод в файл или сервер), введите:

```
REBOL -w
REBOL --nowindow
```

Чтобы предотвратить распечатку информации о запуске, которая полезна, если вы перенаправляете вывод в файл или на сервер, введите:

```
REBOL -q
REBOL --quiet
```

Чтобы Выполнить выражение REBOL из командной строки, введите:

```
REBOL --do "print 1 + 2"
REBOL --do "verbose: true" script.r
```

Можно выполнить удаленный скрипт, расположенный на сервере, с помощью такой строки:

```
REBOL --do "do http://www.rebol.com/speed.r"
```

Чтобы изменить уровень безопасности REBOL, при выполнении скрипта введите:

```
REBOL -s script.r
REBOL --secure none script.r
```

Чтобы использовать сценарии REBOL для CGI (дополнительную информацию см. В разделе CGI - Common Gateway Interface Section в главе о сетевых протоколах), введите:

```
REBOL -c cgi-script.r
REBOL --cgi
```

Также можно указывать несколько буквенных аргументов, при этом аргументы НЕ разделяются пробелами.

```
REBOL -cs cgi-script.r
REBOL --cgi --secure none cgi-script.r
```

Приведенный выше пример работает в режиме CGI с отключенной безопасностью. Сокращенный метод требуется для различных веб-серверов, которые ограничивают количество аргументов, разрешенных в командной строке (например, сервер Apache в Linux).

2.8 Файловое перенаправление

В большинстве систем можно перенаправить стандартный ввод и вывод из файлов и в файлы. Например:

```
rebol -w script.r > output-file
```

перенаправляет вывод информации из интерпретатора в файл. Так же, как

```
rebol -w script.r < input-file
```

перенаправляет ввод данных из файла.

При перенаправлении файлового ввода-вывода ...

Используйте параметр `-w`, чтобы предотвратить открытие окна консоли REBOL, поскольку оно мешает стандартному перенаправлению ввода и вывода.

2.9 Аргументы скрипта

Все, что указано в командной строке после имени файла сценария, передаётся сценарию в качестве аргумента. Это позволяет писать сценарии, принимающие аргументы прямо из командной строки. Например, если вы запустите REBOL со строки:

```
REBOL script.r 10:30 test@domain.dom
```

Есть два способа получить аргументы командной строки. Первый метод возвращает аргументы в виде блока значений REBOL:

```
probe system/options/args
["10:30" "test@domain.dom"]
```

Второй метод возвращает аргументы в виде строки:

```
probe system/script/args
"10:30 test@domain.dom"
```

Примечание, зависящее от версии:

В более ранние версии REBOL возвращали блок значений для поля script/args (аналогично options/args). Рекомендуется убедиться, что ваш сценарий получает правильный тип данных args, как показано выше.

2.10 Файлы запуска

При запуске REBOL пытается загрузить загрузочные файлы `rebol.r` и `user.r`. Эти файлы необязательны, но при обнаружении их можно использовать для настройки сети, определения общих функций и инициализации данных, используемых скриптами.

`Rebol.r` - файл сценария содержит специальные функции или расширения REBOL, которые входят в стандартный комплект поставки. Рекомендуется не редактировать этот файл, так как он перезаписывается с каждой новой версией REBOL.

`User.r` файл сценария содержит пользовательские настройки. Вы можете отредактировать этот файл и добавить любые необходимые определения или данные.

В многопользовательских системах может быть отдельный `user.r` для каждого пользователя. Пока файл `user.r` не является частью дистрибутива, он автоматически создаётся, если не существует.

При запуске REBOL ищет `rebol.r` и `user.r` файлы сначала в домашнем каталоге, а если их там нет, то в текущем каталоге.

Чтобы установить HOME каталог, вы можете установить переменную среды в соответствующем сценарии входа или запуска для вашей системы. Обратите внимание, что некоторые системы, такие как UNIX или Linux, уже это сделали, поэтому в этом нет необходимости.

Для примера в Windows NT для установки HOME вам нужно добавить:

```
set HOME=C:\REBOL
```

в окружении операционной системы, необходимо:

1. Выбрать "Панель управления" настройками в Windows - меню "Пуск",
2. Двойной клик по иконке "Система", затем выбрать "Переменные среды".
3. Имя переменной "HOME", значение C:\REBOL (или другой путь, по которому расположен REBOL).

В системах Unix вы можете установить путь к REBOL, добавив строку, подобную следующей, в свой сценарий оболочки входа или профиль:

```
set HOME=/usr/bin/rebol
```

Для некоторых версий REBOL путь хранится в файле `.rebol`, который находится в вашем домашнем каталоге.

3. Выход из REBOL

Чтобы выйти из REBOL в любое время, выберите **Quit** в меню **Console File** или набрав **quit** или **q** в командной строке.

Вы также можете выйти из программы из скрипта:

```
if now/time > 12:00 [quit]
```

Консоль REBOL может также выйти, если во время запуска произошла ошибка.

Exit не тоже самое, что Quit

Не используйте слово **exit** для выхода из REBOL. Это слово используется для выхода из функций, и при вводе с консоли будет возвращена ошибка.

4. Использование консоли

Всякий раз, когда вы запускаете REBOL/Core, он открывает консоль для отображения приглашения к вооду данных и ожидает ввода. Если вы указали программе в аргументе имя файла сценария, то этот сценарий будет запущен, и вы увидите результат его работы.

Если вы не указывали файл сценария, консоль предложит вам ввести данные. Приглашение к вводу выглядит так:

```
>>
```

Если вы вводите выражение в строке ввода, оно выполнится, и все возвращённые значения отображаются после индикатора результата:

```
==
```

Например:

```
>> 100 + 20
== 120
>> now - 7-Dec-1944
== 20341
```

Изменение вида приглашения

Символы подсказки можно изменить, смотрите Приложение к консоли для получения дополнительной информации.

Консоль также становится активной, если сценарий обнаруживает ошибку или если сценарий вызывает функцию остановки.

4.1 Многострочный ввод

Если вы начинаете блок в командной строке и не заканчиваете его, блок расширяется до следующей строки. На это указывает приглашение, которое начинается со скобки и сопровождается отступом. Строка будет иметь отступ в четыре пробела для каждого открытого блока. Например:

```
loop 10 [
[    print "Пример"
[    if odd? random 10 [
[        print "тут"
[        ]
[    ]
```

Это также верно для многострочных строк, заключённых в фигурные скобки.

```
Print {Это длинная строка,
{    которая состоит не из одной,
{    а из нескольких строк.}
```

Скобки и фигурные скобки внутри строк в кавычках игнорируются. Вы можете выйти из режима ввода в любое время, нажав клавишу ESC.

4.2 Прерывание сценария

Сценарий можно прервать, нажав клавишу ESC, которая немедленно вернётся в командную строку.

Во время некоторых типов операционной системы или сетевой активности может наблюдаться задержка ответа на прерывание ESC.

4.3 Вызов истории

Каждая строка, введённая в REBOL, сохраняется для последующего использования. Клавиши со стрелками вверх и вниз используются для прокрутки списка ранее введённых строк. Например, однократное нажатие стрелки вверх вызывает предыдущую строку ввода.

Строки истории можно записать в файл, сохранив блок истории. Смотрите приложение к консоли для получения дополнительной информации.

4.4 Завершение слова

Чтобы ускорить набор длинных слов и имен файлов, в консоли REBOL есть функция автозавершения слов и имен файлов. После ввода нескольких букв слова нажмите клавишу табуляции. Если буквы однозначно идентифицируют слово, отображается остальная часть слова. Например, набрав:

```
>> sq
```

и нажав клавишу Tab на экране появится:

```
>> square-root
```

Если буквы не идентифицируют слово однозначно, вы можете снова нажать клавишу Tab, чтобы получить список вариантов. Например, набрав:

```
>> so
```

и нажав клавишу Tab дважды, получим :

```
>> sort source
so
```

и вы можете ввести остальную часть слова, чтобы оно было уникальным, и повторно нажать Tab для завершения его.

Завершение работает для всех слов, включая слова, определённые пользователем. Это также работает для файлов, которым предшествует знак процента.

```
>> print read %r
```

Нажатие на Tab может привести к:

```
>> print read %rebol.r
```

в зависимости, от содержимого вашего текущего директория.

4.5 Индикатор занятости

Когда REBOL ожидает завершения сетевой операции, появляется индикатор занятости, указывающий на то, что что-то происходит. Вы можете изменить индикатор на свой собственный. Смотрите Приложение к консоли для получения дополнительной информации.

4.6 Сетевые подключения

При инициировании сетевых подключений на консоли появляется сообщение. Например, набрав:

```
>> read http://www.rebol.com
connecting to: www.rebol.com
```

Если необходимо, вы можете отключить этот вывод, установив флаг молчания. Смотрите приложение к консоли для получения дополнительной информации.

4.7 Виртуальный терминал

Консоль предоставляет возможность виртуального терминала, который позволяет выполнять такие операции, как перемещение курсора, адресация курсора, редактирование строки, очистка экрана, ввод управляющих клавиш и запрос положения курсора.

Виртуальный терминал использует стандартные последовательности символов ANSI. Это позволяет вам писать независимые от платформы терминальные программы, такие как текстовые редакторы, почтовые клиенты или эмуляторы telnet.

Смотрите приложение к консоли для получения дополнительной информации.

5. Получение помощи

Существует несколько источников информации, для получения помощи: интерактивная справка , встроенный в REBOL, в исходной функции, документы на веб - сайте REBOL , в библиотеке сценариев REBOL , в списке рассылки REBOL и отправки обратной связи с REBOL .

Существует несколько источников информации: встроенная функция **source**, документы на сайте REBOL, библиотека скриптов, список рассылок, и форма обратной связи.

5.1 Встроенная справка

Функция **help** используется для быстрого получения справки по командам REBOL.

Введите **help** или **?** в командной строке консоли, чтобы просмотреть сводную справку:

```
Функция справки предоставляет простой способ получить
информацию о словах и значениях. Чтобы использовать его,
укажите в качестве аргумента слово или значение:

    help insert
    help find

Чтобы просмотреть все слова, соответствующие шаблону:

    help "path"
    help to-

Чтобы просмотреть все слова указанного типа данных:

    help native!
    help datatype!

Также есть автозавершение слов из командной строки.
Введите несколько символов и нажмите клавишу TAB,
чтобы завершить слово. Если ничего не происходит, значит
найдено более одного слова. Для однозначной идентификации
слова необходимо достаточное количество символов.
Другие полезные функции:

    about - общая информация
    usage - аргументы командной строки
    license - условия пользовательской лицензии
    source func - исходный код функции
    upgrade - обновить REBOL

Для получения дополнительной информации смотрите руководство пользователя.
```

Если вы указываете функциональное слово в качестве аргумента **help**, то печатается справочная информация, о этой функции. Например, если вы наберете:

```
help insert
```

то вы увидете:

```
ИСПОЛЬЗОВАНИЕ:
     INSERT series value /part range /only /dup count

ОПИСАНИЕ:
     Вставляет значение в серию и возвращает серию после вставки.
     INSERT - это значение действия

АРГУМЕНТЫ:
     series -- Серия в точке для вставки (Тип: битовый набор порта)
     value -- Значение для вставки (Тип: любой)

УТОЧНЕНИЯ:
     /part -- Ограничивается заданной длиной или положением.
         range -- (Тип: номер серии портов)
     /only --  Вставляет серию как серию.
     /dup --  Дублирует вставку указанное количество раз.
         count - (Тип: число)
```

Функция **help** также находит слова, содержащие указанную строку. Например, чтобы найти все слова, содержащие `path`, введите:

```
? "path"
```

И в результате мы получим:

```
Найдены следующие слова:
     clean-path     (функция)
     lit-path!      (тип_данных)
     lit-path?      (действие)
     path           (файл)
     path!          (тип_данных)
     path-thru      (функция)
     path?          (действие)
     set-path!      (тип_данных)
     set-path?      (действие)
     split-path     (функция)
     to-lit-path    (функция)
     to-path        (функция)
     to-set-path    (функция)
```

Вы также можете искать все глобально определённые слова с заданным типом данных. Например, чтобы перечислить все слова, которые являются функциями! типы данных, тип:

```
? function!
```

и увидим следующее:

```
Найдены следующие слова:
     ?               (функция)
     ??              (функция)
     about           (функция)
     alert           (функция)
     alter           (функция)
     append          (функция)
     array           (функция)
     ask             (функция)
...
```

Чтобы получить список всех типов данных REBOL, введите::

```
? datatype!

Найдены следующие слова:
     action!         (тип_данных)
     any-block!      (тип_данных)
     any-function!   (тип_данных)
     any-string!     (тип_данных)
     any-type!       (тип_данных)
     any-word!       (тип_данных)
     binary!         (тип_данных)
     bitset!         (тип_данных)
     block!          (тип_данных)
     char!           (тип_данных)
     datatype!       (тип_данных)
     date!           (тип_данных)
     decimal!        (тип_данных)
     email!          (тип_данных)
...
```

Нет справки по объектам

Функция **help** не предоставляет полезной информации об объектах системы, например:

```
help system/options/home
system/options/home - это путь.
```

Не пытайтесь делать:

```
help system
```

поскольку это может занять продолжительное время и произвести более мегабайта текста.

5.2 Просмотр исходного кода

Опытные пользователи могут узнать больше о конкретных функциях REBOL, изучив их исходный код. Функция **source** отображает код для любой REBOL функции. Если вы наберёте:

```
source join
```

Будет показан исходный код функции `join`:

```
join: func [
    "Concatenates values."
    value "Base value"
    rest "Value or block of values"
][
    value: either series? value [copy value] [form value]
    repend value rest
]
```

Функции, реализованные в машинном коде, не могут быть отображенны.

5.3 Скачать документы

Посетите сайт REBOL http://www.rebol.com/ для получения списка текущей документации.

В дополнение к этому руководству существует *словарь REBOL*, который охватывает все стандартные слова, доступные в REBOL. Если в справке консоли или в этом руководстве недостаточно информации о слове REBOL, поищите в словаре подробное описание.

Словарь обновляется с каждой версией REBOL и доступен по адресу http://www.REBOL.com/docs/dictionary.html.

5.4 Библиотека скриптов

Веб-сайт REBOL содержит библиотеку с многочисленными полезными отлаженными сценариями, которые охватывают множество тем. Библиотека разделена на категории, чтобы упростить поиск сценария, специфичного для данной функции. Вы также можете искать в библиотеке скрипты, содержащие определённое слово.

Библиотеку сценариев можно найти по адресу http://www.REBOL.com/library/library.html.

5.5 Список рассылки

Вы также можете получить помощь от сообщества пользователей REBOL, присоединившись к списку обсуждений REBOL по электронной почте. Чтобы зарегистрироваться, отправьте электронное письмо на адрес rebol-request@rebol.com, указав в строке темы слово «subscribe». Например:

```
send rebol-request@rebol.com "subscribe"
```

Убедитесь, что ваш верный адрес электронной почты был корректно задан с помощью `set-net`.

5.6 Как с нами связаться

Мы хотим узнать, что вы думаете; свяжитесь с нами, чтобы:

- Сообщайте о сбоях или проблемах
- Расскажите, как вы используете REBOL
- Напишите предложения
- Запросите дополнительную информацию о наших продуктах.

Вы можете связаться с нами через страницу обратной связи на нашем веб-сайте.

6. Ошибки

6.1 Сообщения об ошибках

В REBOL есть несколько типов ошибок. При возникновении ошибки отображается сообщение о том, что это была за ошибка и примерно где она произошла. Например, если вы наберёте:

```
abc
** Ошибка сценария: abc не имеет значения.
** Где: abc
```

Тип ошибки обозначается несколькими первыми словами сообщения. В приведённом выше примере ошибка является `ошибкой сценария` . Ошибки сценария являются наиболее распространёнными и возникают, когда вы неправильно используете функцию языка или с неправильными аргументами. Другие типы ошибок описаны в разделе "Типы ошибок".

Тип ошибки	Описание
`Syntax errors` Синтаксические ошибки	Происходит, когда сценарий содержит недопустимое значение или отсутствует заголовок, цитата, скобка или кавычка.
`Math errors` Математические ошибки	Возникает при делении на ноль или при математическом переполнении или потере значимости.
`Access errors` Ошибки доступа	Происходит, когда нет доступа к файлу, каталогу или сетевой операции или ограничены права доступа.
`Throw errors` Ошибка прерывания	Происходит при неправильном использовании команды break, exit или throw.
`User errors` Ошибки пользователей	Определяется скриптом пользователя.
`Internal errors` Внутренние ошибки	Возвращается при возникновении проблемы в системе REBOL. Если вы столкнулись с одним из этих типов ошибок, сообщите об этом через форму обратной связи.

Сценарий может улавливать и обрабатывать большинство типов ошибок. Смотрите описание функции trying в разделе «Пробные блоки».

Приложение «Ошибки» также содержит полезную информацию об ошибках.

6.2 Перенаправление ошибок

При обнаружении ошибок в не интерактивных сеансах, например, при работе в режиме CGI (-c или --cgi) или в безоконном режиме (-w или --nowindow), сеанс автоматически завершается.

Если сценарий завершается во время работы в не интерактивном режиме, вы можете использовать перенаправление оболочки для вывода ошибки в файл:

```
REBOL -cs my_script.r >> my_script.log
```

Эта запись позволит добавить вывод к файлу в большинстве операционных систем.

7. Обновление

2.3.0.3.1 : система печати / версия Обновить : Windows 95/98 / NT iX86 REBOL / core 2.3.0.3.1 или же:
При запуске интерпретатора отображается баннер, указывающий версию программы. Номера версий имеют формат:

```
версия.редакция.обнавление.платформа.вариант
```

Например, номер версии:

```
2.3.0.3.1
```

указывает, что вы используете версию 2, редакцию 3, обновление 0 для Windows 95/98 / NT (номер платформы REBOL 3.1). Полный список всех номеров платформ доступен на http://www.rebol.com . Этот HTML-файл также содержит скрытую базу данных REBOL, которую можно использовать для определения платформы.

Вы можете получить номер версии из приглашения REBOL с помощью:

```
print system/version
```

REBOL Technologies поддерживает только последнюю версию REBOL. Вы можете убедиться, что у вас установлена последняя версия, и автоматически обновить ее, если она устарела. Для этого убедитесь, что вы подключены к Интернету, а затем из REBOL введите:

```
upgrade
```

RREBOL возвращает одно из следующих сообщений о вашей версии:

```
Эта копия Windows 95/98/NT iX86 REBOL/core 2.3.0.3.1 в настоящее время актуальна.
```

или же:

```
Эта копия Windows 95/98 / NT iX86 REBOL / core 2.1.2.3.1 устарела. Текущая версия:
2.3.0.3.1.
Скачать текущую версию?
```

Для обновления до последней версии введите Y (да). В противном случае введите N (нет)..

Глава 3 - Краткий обзор

Содержание:

1. Обзор
2. Значения
  2.1 Числа
  2.2 Время
  2.3 Даты
  2.4 Деньги
  2.5 Кортежи
  2.6 Строки
  2.7 Теги
  2.8 Адреса электронной почты
  2.9 Ссылки (URL)
  2.10 Имена файлов
  2.11 Пары
  2.12 Выпуски
  2.13 Двоичные
3. Слова
4. Блоки
5. Переменные
6. Оценка
7. Функции
8. Пути
9. Объекты
10. Сценарии
11. Файлы
12. Сеть
  12.1 HTTP
  12.2 FTP
  12.3 SMTP
  12.4 POP
  12.5 NNTP
  12.6 Daytime
  12.7 Whois
  12.8 Finger
  12.9 DNS
  12.10 TCP

1. Обзор

Эта глава предоставляет быстрый способ познакомиться с языком REBOL. На примерах в этой главе представлены основные концепции и структура языка, иллюстрирующие все, от значений данных до выполнения сетевых операций.

2. Значения

Сценарий написан с последовательностью ***значения***. Существует множество значений, и многие из них вам знакомы из повседневного опыта.

По возможности, REBOL также позволяет использовать международные форматы для значений, таких как десятичные числа, деньги, время и дата.

2.1 Цифры

Числа записываются в виде целых чисел, десятичных знаков или экспоненциальной записи. Например:

```
1234 -432 3.1415 1.23E12
```

А также можно написать в европейском формате:

```
123,4 0,01 1,2E12
```

2.2 Время

Время записывается в часах и минутах с необязательными секундами, каждое из которых разделено двоеточиями. Например:

```
12:34 20:05:32 0:25.345 0:25,345
```

Секунды могут включать дробную долю секунды. Время также может включать AM и PM без промежуточных пробелов:

```
12:35PM 9:15AM
```

2.3 Даты

Даты записываются в международном формате: день-месяц-год или год-месяц-день. Дата также может включать время и часовой пояс. Название или аббревиатура месяца могут использоваться, чтобы сделать его формат более узнаваемым в США. Например:

```
20-Apr-1998 20/Apr/1998 (как в  США)
20-4-1998 1998-4-20     (международное)
1980-4-20/12:32         (дата со временем)
1998-3-20/8:32-8:00     (с часовым поясом)
```

2.4 Деньги

Деньги записываются как необязательный международный трехбуквенный символ валюты, за которым следует числовая сумма. Например:

```
$12.34  USD$12.34  CAD$123.45  DEM$1234,56
```

2.5 Кортежи

Кортежи используются для номеров версий, значений цвета RGB и сетевых адресов. Они записываются как целые числа от 0 до 255 и разделены точками. Например:

```
2.3.0.3.1  255.255.0  199.4.80.7
```

Требуются как минимум две точки (иначе число будет интерпретировано как десятичное значение, а не как кортеж). Пример:

```
2.3.0   ; кортеж
2.3.    ; кортеж
2.3     ; десятичное
```

2.6 Строки

Строки записываются в однострочном или многострочном формате. Строки однострочного формата заключаются в кавычки. Строки многострочного формата заключаются в фигурные скобки. Строки, содержащие кавычки, табуляторы или разрывы строк, должны быть заключены в фигурные скобки в многострочном формате. Например:

```
"Вот однострочная строка"

{Вот многострочная строка,
содержит "строку" в кавычках.}
```

Специальные символы (escape-символы) внутри строк обозначаются символом вставки (^). Смотрите Раздел, посвященный строкам, в главе «Значения» для получения таблицы управляющих последовательностей.

2.7 Теги

Теги полезны для языков разметки, таких как XML и HTML. Теги заключаются в угловые скобки. Например:

```
<title> </body>

<font size="2" color="blue">
```

2.8 Адреса электронной почты

Адреса электронной почты пишутся прямо в REBOL. Они должны включать знак "@". Например:

```
info@rebol.com

pres-bill@oval.whitehouse.gov
```

2.9 Ссылки (URL)

REBOL принимает большинство типов URL-адресов в Интернете. Они начинаются с имени схемы (например, HTTP), за которым следует путь. Например:

```
http://www.rebol.com

ftp://ftp.rebol.com/sendmail.r

ftp://freda:grid@da.site.dom/dir/files/

mailto:info@rebol.com
```

2.10 Имена файлов

Перед именами файлов стоит знак процента, чтобы отличать их от других слов. Например:

```
%data.txt

%images/photo.jpg

%../scripts/*.r
```

2.11 Пары

Пары используются для обозначения пространственных координат, например положения на дисплее. Они используются для обозначения позиций и размеров. Координаты разделены знаком `x` . Например:

```
100x50

1024x800

-50x200
```

2.12 Выпуски

Выпуск связаны с идентификационными номерами, такими как номера телефонов, номера моделей, номера кредитных карт. Например:

```
#707-467-8000

#0000-1234-5678-9999

#MFG-932-741-A
```

2.13 Двоичные

Двоичные значения - это байтовые строки любой длины. Их можно закодировать напрямую в шестнадцатеричном формате или в формате base-64. Например:

```
#{42652061205245424F4C}

64#{UkVCT0wgUm9ja3Mh}
```

3. Слова

Слова - это символы, используемые REBOL. Слово может быть или не быть переменной, в зависимости от того, как оно используется. Слова также используются непосредственно как символы.

```
show next image

Install all files here

Country State City Street Zipcode

on off true false one none
```

REBOL не имеет ключевых слов; нет никаких ограничений на то, какие слова используются или как они используются. Например, вы можете определить свою собственную функцию с именем `print` и использовать ее вместо предопределенной функции для печати.
Слова не чувствительны к регистру и могут включать дефисы и некоторые другие специальные символы, такие как:

```
+ - `  * ! ~ & ? |
```

Следующие примеры иллюстрируют допустимые слова:

```
number?  time?  date!
image-files  l'image
++ -- == +-
***** *new-line*
left&right left|right
```

Конец слова обозначается пробелом, разрывом строки или одним из следующих символов:

```
[ ] ( ) { } " : ; /
```

В словах нельзя использовать следующие символы:

```
@ # $ % ^ ,
```

4. Блоки

REBOL состоит из группировки значений и слов в ***блоки***. Блоки используются для кода, списков, массивов, таблиц, каталогов, ассоциаций и других последовательностей.

Блок - это тип *серии*, который представляет собой набор *значений*, организованных *в определённом порядке*.

Блок заключён в квадратные скобки []. Внутри блока значения и слова могут быть организованы в любом порядке и могут охватывать любое количество строк. Следующие примеры иллюстрируют допустимые формы блоков:

```
[белый красный зеленый синий желтый оранжевый черный]

["Спилберг" "Назад в будущее" 1:56:20 MCA]

[
    Ted     ted@gw2.dom   #213-555-1010
    Bill    billg@ms.dom  #315-555-1234
    Steve   jobs@apl.dom  #408-555-4321
]

[
    "Elton John"  6894  0:55:68
    "Celine Dion" 68861 0:61:35
    "Pink Floyd"  46001 0:50:12
]
```

Блоки используются как для кода, так и для данных, как показано в следующих примерах:

```
loop 10 [print "hello"]

if time > 10:30 [send jim news]

sites: [
    http://www.rebol.com [save %reb.html data]
    http://www.cnn.com   [print data]
    ftp://www.amiga.com  [send cs@org.foo data]
]

foreach [site action] sites [
    data: read site
    do action
]
```

Сам файл сценария также является блоком. Блок подразумевается, хотя и не включает скобки. Например, если следующие строки были помещены в файл сценария:

```
красный
зелёный
синий
жёлтый
```

Когда файл загружен, это будет блок, содержащий слова красный, зелёный, синий и жёлтый. Это эквивалентно написанию:

```
[красный зелёный синий жёлтый]
```

5. Переменные

Слова могут использоваться как переменные, относящиеся к значениям. Чтобы определить слово как переменную, после слова поставьте двоеточие (:), а затем значение, на которое ссылается эта переменная, как показано в следующих примерах :

```
возраст: 22
обеденное_время: 12:32
день_рождения: 20-Mar-1997
друзья: ["Женька" "Пашка" "Светка"]
```

Переменная может относиться к любому типу значения, включая функции (см. Функции) и объекты (см. Объекты).

Переменная относится к определенному значению только в определенном контексте, таком как блок, функция или вся программа. Вне этого контекста переменная может ссылаться на какое-то другое значение или вообще не иметь значения. Контекст переменной может охватывать всю программу или ограничиваться конкретным блоком, функцией или объектом. В других языках контекст переменной часто называют областью действия переменной.

6. Оценка

Блоки выполняются для вычисления их результатов. Когда блок выполняется, то получаются значения его переменных. В следующих примерах оцениваются переменные `возраст` , `обеденное_время` , `день_рождения` и `друзья`, которые были определены в предыдущем разделе:

```
print возраст
22
if now/time > обеденное_время [print обеденное_время]
12:32
print third друзья
Светка
```

Блок можно выполнить несколько раз с помощью цикла, как показано в следующих примерах:

```
loop 10 [prin "*"]  ;("prin" это не опечатка, см. руководство)
**********
loop 20 [
    wait 8:00
    send friend@rebol.com read http://www.cnn.com
]

repeat count 3 [print ["Счёт:" count]]
Счёт: 1
Счёт: 2
Счёт: 3
```

Оценка блока возвращает результат. В следующих примерах 5 и PM являются результатами оценки каждого блока:

```
print do [2 + 3]
5
print either now/time < 12:00 ["AM"]["PM"]
PM
```

В REBOL нет специальных правил приоритета операторов для оценки блоков. Значения и слова блока всегда оцениваются от первого до последнего, как показано в следующем примере:

```
print 2 + 3 * 10
50
```

Можно использовать скобки для управления порядком вычисления, как показано в следующих примерах:

```
2 + (3 * 10)
32
(length? "boat") + 2
6
```

Вы также можете оценить блок и вернуть каждый результат, который был вычислен в нем. Это цель функции **reduce**:

```
reduce [1 + 2  3 + 4  5 + 6]
3 7 11
```

7. Функции

Функция - это блок с переменными, которым при каждом запуске блока присваиваются новые значения. Эти переменные называются аргументами функции.

В следующем примере слово sum устанавливается для ссылки на функцию, которая принимает два аргумента, a и b:

```
sum: func [a b] [a + b]
```

В приведенном выше примере **func** используется для определения новой функции. Первый блок функции описывает аргументы функции. Второй блок - это блок кода, который выполняется при вызове функции. В этом примере второй блок складывает два значения и возвращает результат.

В следующем примере показано одно использование функции sum, которая была определена в предыдущем примере:

```
print sum 2 3
5
```

Некоторым функциям нужны локальные переменные, а также аргументы. Чтобы определить этот тип функции, используйте **function** вместо **func**, как показано в следующем примере:

```
average: function [series] [total] [
    total: 0
    foreach value series [total: total + value]
    total / (length? series)
]

print average [37 1 42 108]
47
```

В приведённом выше примере слово `series` является аргументом, а слово `total` - локальной переменной, используемой функцией для расчётов.

Блок аргументов функции может содержать строки, описывающие назначение функции и её аргумент, как показано в следующем примере:

```
average: function [
    "Возвращает среднее числовое значение"
    series "числа для расчёта среднего"
] [total] [
    total: 0
    foreach value series [total: total + value]
    total / (length? series)
]
```

Эти описательные строки сохраняются вместе с функцией, и их можно просмотреть, запросив помощь по функции, как показано ниже:

```
help average
ИСПОЛЬЗОВАНИЕ:
    AVERAGE series
ОПИСАНИЕ:
    Возвращает среднее числовое значение
    AVERAGE is a function value.
АРГУМЕНТЫ:
    series -- числа для усреднения (Тип: любой)
```

8. Пути

Если вы используете файлы и URL-адреса, то вы уже знакомы с концепцией путей. Путь предоставляет набор значений, которые используются для перехода от одной точки к другой. В случае файла путь определяет маршрут через набор каталогов к местоположению файла. В REBOL значения в пути называются ***refinements***(уточнения).

Косая черта (/) используется для разделения слов и значений в пути, как показано в следующих примерах пути к файлу и пути URL:

```
%source/images/globe.jpg

http://www.rebol.com/examples/simple.r
```

Пути также можно использовать для выбора значений из блоков, выбора символов из строк, доступа к переменным в объектах и уточнения работы функции, как показано в следующих примерах:

```
USA/CA/Ukiah/size  (выбор блока)
names/12           (позиция в строке)
account/balance    (функция объекта)
match/any          (опция функции)
```

Функция `print` в следующем примере показывает простоту использования пути для доступа к мини-базе данных, созданной из нескольких блоков:

```
towns: [
    Hopland [
        phone #555-1234
        web   http://www.hopland.ca.gov
    ]

        Ukiah [
        phone #555-4321
        web   http://www.ukiah.com
        email info@ukiah.com
    ]
]

print towns/ukiah/web
http://www.ukiah.com
```

9. Объекты

Объект - это набор переменных, которые имеют определённые значения в контексте. Объекты используются для управления структурами данных и более сложным поведением. В следующем примере показано, как банковский счёт настраивается как объект для определения его атрибутов и функций:

```
аккаунт: make object! [
  имя: "Джеймс"
  баланс: $100
  ss-номер: #1234-XX-4321
  депозит:  func [amount] [баланс: баланс + amount]
  снятие: func [amount] [баланс: баланс - amount]
]
```

В приведённом выше примере слова `имя`, `баланс`, `ss-номер`, `депозит` и `снятие` являются локальными переменными объекта `аккаунт`. `Депозит` и `снятие` это функции, которые определены в пределах объекта. Доступ к переменным аккаунта можно получить по пути, как показано в следующем примере:

```
print аккаунт/баланс
$100.00
аккаунт/депозит $300

print ["Баланс " аккаунт/имя " составляет " аккаунт/баланс]
Баланс Джеймс составляет $400.00
```

В следующем примере показано, как создать другой аккаунт с новым балансом, но при этом все остальные значения остаются прежними:

```
другой-аккаунт: make аккаунт [
    баланс: $2000
]
```

Обратите внимание, что новый объект называется `другой-аккаунт`, и он разделяет значения исходного аккаунта, но предоставляет новую сумму баланса.
Так же легко вы можете создать новый объект, который расширяет старый объект, добавляя новые переменные для названия банка и даты последней активности:

```
другой-аккаунт: make аккаунт [
    банк: "ООО Банк"
    дата-посещения: 20-Jun-2000
]

print другой-аккаунт/баланс
$2000.00
print другой-аккаунт/банк
ООО Банк
print другой-аккаунт/дата-посещения
20-Jun-2000
```

10. Скрипты

Сценарий - это файл, содержащий блок, который можно загрузить и оценить. Блок может содержать код или данные и обычно содержит ряд подблоков.

Сценариям требуется заголовок для определения наличия кода. Заголовок может включать имя сценария, дату и другую информацию. В следующем примере сценария первый блок содержит информацию заголовка:

```
REBOL [
    Title: "Детектор изменения веб-страницы"
    File:  %webcheck.r
    Author: "Reburu"
    Date:  20-May-1999
    Purpose: {
        Определить, изменилась ли веб-страница с момента
        последней проверки, и если да, то отправить новую страницу
        по электронной почте.
    }
    Category: [web email file net 2]
]
page: read http://www.rebol.com
page-sum: checksum page

if any [
    not exists? %page-sum.r
    page-sum <> (load %page-sum.r)
][
    print ["Страница изменена" now]
    save %page-sum.r page-sum
    send luke@rebol.com page
]
```

11. Файлы

В REBOL к файлам легко получить доступ. В следующей таблице описаны некоторые способы доступа к файлам.

Вы можете прочитать текстовый файл с помощью:

```
data: read %plan.txt
```

Вы можете отобразить текстовый файл с помощью:

```
print read %plan.txt
```

Чтобы написать текстовый файл:

```
write %plan.txt data
```

Например, вы можете записать текущее время с помощью:

```
write %plan.txt now
```

Вы также можете легко добавить в конец файла:

```
write/append %plan data
```

Двоичные файлы можно читать и записывать с помощью:

```
data: read/binary %image.jpg
write/binary %new.jpg data
```

Чтобы загрузить файл как блок или значение REBOL:

```
data: load %data.r
```

Сохранить блок или значение в файл так же просто:

```
save %data.r data
```

Чтобы запустить файл как сценарий (для этого нужен заголовок):

```
do %script.r
```

Вы можете прочитать каталог файлов с помощью:

```
dir: read %images/
```

а затем вы можете отобразить имена файлов с помощью:

```
foreach file dir [print file]
```

Чтобы создать новый каталог:

```
make-dir %newdir/
```

Чтобы узнать текущий путь к каталогу:

```
print what-dir
```

Если вам нужно удалить файл:

```
delete %oldfile.txt
```

Вы также можете переименовать файл с помощью:

```
rename %old.txt %new.txt
```

Чтобы получить информацию о файле:

```
print size? %file.txt
print modified? %file.txt
print dir? %image
```

12. Сеть

В REBOL встроено несколько Интернет-протоколов. Эти протоколы просты в использовании и требуют очень мало знаний о работе в сети.

12.1 HTTP

В следующем примере показано, как использовать протокол HTTP для чтения веб-страницы:

```
page: read http://www.rebol.com
```

В следующем примере изображение загружается с веб-страницы и записывается в локальный файл:

```
image: read/binary http://www.page.dom/image.jpg

write/binary %image.jpg image
```

12.2 FTP

Следующее читает и записывает файлы на сервер, используя протокол передачи файлов (FTP):

```
file: read ftp://ftp.rebol.com/test.txt

write ftp://user:pass@site.dom/test.txt file
```

В следующем примере получается список каталогов с FTP:

```
print read ftp://ftp.rebol.com/pub
```

12.3 SMTP

В следующем примере отправляется письмо на электронную почту, с помощью простого протокола передачи почты (SMTP):

```
send luke@rebol.com "Используй силу."
```

В следующем примере текст файла отправляется по электронной почте:

```
send luke@rebol.com read %plan.txt
```

12.4 POP

В следующем примере выполняется выборка электронной почты по протоколу почтового отделения (POP) и печать всех текущих сообщений. Сообщения остаются на сервере:

```
foreach message read pop://user:pass@mail.dom [
    print message
]
```

12.5 NNTP

В следующем примере извлекаются новости по сетевому протоколу передачи новостей (NNTP), читаются все новости в определённой группе новостей:

```
messages: read nntp://news.server.dom/comp.lang.rebol
```

В следующем примере считывается и печатается список всех групп новостей:

```
news-groups: read nntp://news.some-isp.net

foreach group news-groups [print group]
```

12.6 Daytime

Следующий пример получает текущее время с сервера:

```
print read daytime://everest.cclabs.missouri.edu
```

12.7 Whois

В следующем примере выясняется, кто отвечает за домен, используя протокол whois:

```
print read whois://rebol@rs.internic.net
```

12.8 Finger

Следующий пример получает информацию о пользователе с помощью протокола finger:

```
print read finger://username@host.dom
```

12.9 DNS

В следующем примере определяется интернет-адрес из доменного имени и доменное имя из адреса:

```
print read dns://www.rebol.com
print read dns://8.8.8.8
```

12.10 TCP

В REBOL также возможны прямые соединения с TCP/IP. Следующий пример представляет собой простой, но полезный сервер, который ожидает подключения к порту, а затем выполняет все, что было отправлено:

```
server-port: open/lines tcp://:9999

forever [
    connection-port: first server-port
    until [
        wait connection-port
        error? try [do first connection-port]
    ]
    close connection-port
]
```

Глава 4 – Выражения

Содержание:

1. Обзор
2. Блоки
3. Значения
    - 3.1 Прямые и косвенные значения
    - 3.2 Типы данных значений
4. Оценка выражений
    - 4.1 Оценка входных данных с консоли
    - 4.2 Оценка простых значений
    - 4.3 Оценка блоков
    - 4.4 Редукционные блоки
    - 4.5 Оценка скриптов
    - 4.6 Оценка строк
    - 4.7 Ошибки оценки
5. Слова
    - 5.1 Имена Слов
    - 5.2 Использование слов
    - 5.3 Установка слов
    - 5.4 Получение слов
    - 5.5 Буквенные слова
    - 5.6 Неустановленные слова
    - 5.7 Защита слов
6. Условная оценка
    - 6.1 Условные блоки
    - 6.2 Любые и все
    - 6.3 Условные циклы
    - 6.4 Распространённые ошибки
7. Повторная оценка
    - 7.1 Loop
    - 7.2 Repeat
    - 7.3 For
    - 7.4 Foreach
    - 7.5 Forall and Forskip
    - 7.6 Forever
    - 7.7 Break
8. Выборочная оценка
    - 8.1 Select
    - 8.2 Switch
9. Остановка оценки
10. Trying Blocks

1. Обзор

Основная цель REBOL - установить стандартный метод связи, охватывающий все компьютерные системы. REBOL предоставляет простые, прямые средства выражения любого вида информации с оптимальной гибкостью и минимальным синтаксисом. Например, рассмотрите следующую строку:

```
Sell 100 shares of "Acme" at $47.97 per share
```

(Продать 100 акций "Акме" по цене 47,97 доллара за акцию.)

Строка очень похожа на английский, поэтому ее легко составить, если вы ее отправляете, и легко понять, если вы ее получаете. Однако эта строка на самом деле является допустимым выражением в REBOL, поэтому ваш компьютер также может ее понять и действовать в соответствии с ней. (Обратите внимание, что строка является «диалектом» REBOL. Ее можно напрямую оценить. Подробнее об этой концепции ниже.)

REBOL обеспечивает общий язык между вами и вашим компьютером. Кроме того, если ваш компьютер отправляет это выражение на компьютер вашего биржевого брокера, на котором также работает REBOL, компьютер вашего биржевого брокера может понять это выражение и действовать в соответствии с ним. Итак, REBOL обеспечивает общий язык между компьютерами. И эта линия может быть отправлена в миллионы других компьютерных систем, которые также могут с ней работать.

Следующая строка - еще один пример выражения REBOL:

```
Reschedule exam for 2-January-1999 at 10:30
```

(Перенести экзамен на 2 января 1999 г. в 10:30)

Выражение, показанное в приведённом выше примере (написанное на другом диалекте), могло быть получено вашим врачом, набравшим его, или, возможно, оно возникло из приложения, которое запускал ваш врач. Это неважно. Важно то, что выражение можно использовать независимо от типа компьютера, портативного устройства, киоска или телевизионной консоли, которую вы используете.

Значения данных (числа, строки, цены, даты и время) во всех выражениях, показанных в предыдущих примерах, являются стандартизованными допустимыми форматами REBOL. Однако слова зависят от конкретного контекста интерпретации, чтобы передать их значение. Такие слова, как `sell` (продавать) , `at` (в) и `read` (читать), имеют разное значение в разных контекстах. Слова являются относительными выражениями - их значение зависит от контекста.

Выражения могут обрабатываться одним из двух способов: ***напрямую*** интерпретатором REBOL или ***косвенно*** скриптом REBOL. Выражение, обрабатываемое косвенно, называется ***диалектом***. Предыдущие примеры являются диалектами, поэтому они обрабатываются сценарием. Следующий пример не является диалектом и обрабатывается непосредственно интерпретатором REBOL:

```
send master@rebol.com read http://www.rebol.com
```

В этом примере слова **`send`** и **`read`** - это функции, которые обрабатываются интерпретатором REBOL.

REBOL делает различие в том, что **информация интерпретируется прямо или косвенно**. Различие не в том, является ли информация кодом или данными, а в том, как она обрабатывается. В REBOL код часто обрабатывается как данные, а данные часто обрабатываются как код, поэтому традиционное разделение между кодом и данными размывается. То, как обрабатывается информация, определяет, является ли это кодом или данными.

2. Блоки

Выражения REBOL основаны на этой концепции: вы объединяете ***значения*** и ***слова*** в ***блоки***.

В скриптах блок обычно заключён в квадратные скобки [ ]. Все, что заключено в квадратные скобки, является частью блока. Содержимое блока может занимать любое количество строк, а его формат полностью произвольный.

В следующих примерах показаны различные способы форматирования содержимого блока:

```
[белый красный зеленый синий желтый оранжевый черный]

["Спилберг" "Назад в будущее" 1:56:20 MCA]

[
    "Билл" billg@ms.dom # 315-555-1234
    "Стив" jobs@apl.dom # 408-555-4321
    "Тед" ted@gw2.dom # 213-555-1010
]

места: [
    http://www.rebol.com [save %reb.html data]
    http://www.cnn.com   [print data]
    ftp://www.amiga.com  [send cs@org.foo data]
]
```

Некоторые блоки не требуют квадратных скобок, потому что они подразумеваются. Например, в сценарии REBOL весь сценарий не заключён в скобки, однако содержимое сценария представляет собой блок. Подразумеваются квадратные скобки ***внешнего блока*** скрипта. То же самое верно для выражений, вводимых в командной строке, или для сообщений REBOL, отправляемых между компьютерами - каждое из них является подразумеваемым блоком.

Ещё один важный аспект блоков - это то, что они несут дополнительную информацию. Блоки ***группируют*** набор значений в определённом ***порядке***. То есть блок может использоваться как набор данных, а также как последовательность. Это будет описано более подробно в главе серии.

3. Значения

REBOL предоставляет встроенный набор ***значений***, которые могут быть выражены и обменены между всеми системами. Значения являются основными элементами для составления всех выражений REBOL.

3.1 Прямые и косвенные значения

Ценности могут быть выражены ***прямо*** или ***косвенно***.

Непосредственно выраженное значение ***известно***, как оно написано лексически или буквально. Например, число 10 или время 10:30 являются непосредственно выраженными значениями.

Косвенно выраженное значение ***неизвестно*** до тех пор, пока оно не будет выполнено. Значения `none`, `true` и `false` требуют слов для их представления. Эти значения выражаются косвенно, потому что они должны быть вычислены, чтобы их значения были известны. Это также верно для других значений, таких как списки, хэши, функции, объекты.

3.2 Типы данных значений

Каждое значение REBOL относится к определённому ***типу данных***. Тип данных значения определяет:

1. Диапазон возможных значений для типа данных. Например, тип данных логики может быть только истинным или ложным.
2. Операции, которые можно выполнять. Например, вы можете сложить два целых числа, но не можете сложить два логических значения.
3. Способ хранения значений в памяти. Некоторые типы данных могут храниться напрямую (например, числа), а другие - косвенно (например, строки).

По соглашению после слов типа REBOL стоит восклицательный знак (!), Чтобы выделить их. Например:

```
integer!
char!
word!
string!
```

Слова типа данных - это просто слова

Слова, используемые для типов данных, такие же, как и любые другие слова в REBOL. В этом нет ничего волшебного! используется для их представления.

См. Приложение «Значения» для описания всех типов данных REBOL.

4. Оценка выражений

Чтобы ***оценить*** выражение, нужно вычислить его значение. REBOL работает, оценивая серию выражений, составляющих сценарий, и затем возвращает результат. Оценка также называется запуском, обработкой или выполнением сценария.

Оценка проводится по блокам. Блоки можно набирать на консоли или загружать из файла сценария. В любом случае процесс оценки одинаков.

4.1 Оценка ввода с консоли

Любое выражение, которое может быть вычислено в сценарии, также может быть вычислено из приглашения REBOL, предоставляя простые средства проверки отдельных выражений в сценарии.

Например, если вы введёте следующее выражение в командной строке консоли:

```
>> 1 + 2
```

выражение оценивается и возвращается следующий результат:

```
== 3
```

О примерах кода ...

В приведенном выше примере показано приглашение консоли (>>) и индикатор результата (==), чтобы дать вам представление о том, как они отображаются в консоли. Для следующих примеров строки подсказки и результата не показаны. Однако вы можете предположить, что эти примеры можно ввести в консоль, чтобы проверить их результаты.

4.2 Оценка простых значений

Поскольку значение прямо выраженных значений известно, они просто возвращают свои значения. Например, если вы наберёте следующую строку:

```
10:30
```

возвращается значение 10:30. Это поведение всех прямо выраженных значений. Оно включает:

```
integer    1234
decimal    12.34
string     "REBOL world!"
time       13:47:02
date       30-June-1957
tuple      199.4.80.1
money      $12.49
pair       100x200
char       #"A"
binary     #{ab82408b}
email      info@rebol.com
issue      #707-467-8000
tag        <IMG SRC="xray.jpg">
file       %xray.jpg
url        http://www.rebol.com/
block      [milk bread butter]
```

4.3 Оценка блоков

Обычно блоки ***не*** оцениваются. Например, набрав следующий блок:

```
[1 + 2]
```

возвращает тот же блок:

```
[1 + 2]
```

Блок не оценивается; это просто воспринимается как данные.

Чтобы оценить блок, используйте функцию do, как показано в следующем примере:

```
do [1 + 2]
3
```

Функция do возвращает результат оценки. В предыдущем примере возвращается число 3.

Если блок содержит несколько выражений, возвращается только результат последнего выражения:

```
do [
    1 + 2
    3 + 4
]
7
```

В этом примере вычисляются оба выражения, но возвращается только результат выражения 3 + 4.

Существует ряд функций, таких как **if**, **loop**, **while**, и **foreach**, которые оценивают блок как часть своей функции. Эти функции подробно обсуждаются позже в этой главе, но вот несколько примеров:

```
if time > 12:30 [print "после полудня"]
после полудня
loop 4 [print "зацикливание"]
зацикливание
зацикливание
зацикливание
зацикливание
```

Это важно помнить: блоки обрабатываются как данные до тех пор, пока они не будут явно оценены функцией. Только функция может вызвать их оценку.

4.4 Редукционные блоки

Когда вы оцениваете блок с помощью **do**, в результате возвращается только значение его последнего выражения. Однако бывают случаи, когда вы хотите, чтобы возвращались значения всех выражений в блоке. Чтобы вернуть результаты всех выражений в блоке, используйте функцию **reduce**. В следующем примере **reduce** используется для возврата результатов обоих выражений в блоке:

```
reduce [
    1 + 2
    3 + 4
]
[3 7]
```

В приведённом выше примере блок был ***редуцирован*** до результатов оценки. Функция **reduce** возвращает результаты в блоке.

Функция **reduce** важна, поскольку она позволяет вам создавать блоки выражений, которые оцениваются и передаются другим функциям. **Reduce** оценивает каждое выражение в блоке и помещает результат этого выражения в новый блок. Этот новый блок возвращается в результате **"редуцирования"**.

Некоторые функции, такие как **print**, используют **reduce** как часть своей операции, как показано в следующем примере:

```
print [1 + 2  3 + 4]

3 7
```

В **rejoin** , **reform** и **remold** функции также используют **редуцирования** как часть своей работы, как показано в следующих примерах:

```
print rejoin [1 + 2  3 + 4]
37
print reform [1 + 2  3 + 4]
3 7
print remold [1 + 2  3 + 4]
[3 7]
```

В rejoin, reform, и remold функции основаны на join , form и mold функциях, но сперва выполняется reduce.

4.5 Оценка скриптов

Функция do может быть использована для оценки целых сценариев. Обычно do оценивает блок, как показано в следующем примере:

```
do [print "Привет!"]
Привет!
```

Но когда do оценивает имя файла вместо блока, файл будет загружен в интерпретатор как блок, а затем оценён, как показано в следующем примере:

```
do %script.r
```

Требуется заголовок REBOL

Для оценки файла сценария он должен включать допустимый заголовок REBOL, который описан в главе "Сценарии". Заголовок указывает, что файл содержит сценарий, а не просто случайный текст.

4.6 Оценка строк

функция do ожет быть использована для вычисления выражений , которые находятся в пределах текстовых строк. Например, следующее выражение:

```
do "1 + 2"
3
```

возвращает результат 3. Сначала строка преобразуется в блок, затем блок оценивается.

Оценка строк может быть удобной время от времени, но это следует делать только при необходимости. Например, чтобы создать процессор строки консоли REBOL, введите следующее выражение:

```
forever [probe do ask "=> "]
```

Приведённое выше выражение предложит вам ввести => и дождаться, пока вы наберёте строку текста. Затем текст будет оценён, и его результат будет напечатан. (Конечно, на самом деле это не так просто, потому что сценарий мог вызвать ошибку.)

Если в этом нет необходимости, оценка строк обычно не является хорошей практикой. Оценка строк менее эффективна, чем оценка блоков, а контекст слов в строке неизвестен. Например, следующее выражение:

```
do form ["1" "+" "2"]
```

намного менее эффективен, чем ввод:

```
do [1 + 2]
```

Блоки REBOL могут быть созданы так же легко, как и строки, а блоки лучше подходят для выражений, которые необходимо вычислить.

4.7 Ошибки оценки

Во время оценки ошибки могут возникать по разным причинам. Например, если вы разделите число на ноль, оценка остановится и отобразится ошибка.

```
100 / 0
** Математическая ошибка: попытка разделить на ноль.
** Где: 100 / 0
```

Распространенная ошибка - использование слова до того, как оно было определено:

```
size + 10
** Ошибка скрипта: size не имеет значения.
** Где: size + 10
```

Другая распространённая ошибка заключается в том, что функции в выражении не предоставляются правильные значения:

```
10 + [size]
** Ошибка скрипта: невозможно использовать добавление значения в блок!.
** Где: 10 + [size]
```

Иногда ошибки не так очевидны, и вам нужно будет поэкспериментировать, чтобы определить, что конкретно вызывает ошибку.

5. Слова

Выражения строятся из ценностей и слов. Слова используются для обозначения смысла. Слово может представлять идею или конкретную ценность.

В предыдущих примерах этой главы в выражениях использовалось несколько слов без объяснения причин. Например, слово **do**, **reduce** и **try** используются, но не объясняются.

Слова оцениваются несколько иначе, чем прямо выраженные ценности. Когда слово оценивается, его значение ищется, оценивается и возвращается в качестве результата. Например, если вы наберёте следующее слово:

```
Zero

0
```

возвращается значение 0. Слово **zero** заранее определено как «ноль». При поиске слова обнаруживается и возвращается ноль.

Когда ищутся такие слова, как **do** и **print**, их значения оказываются функциями, а не простыми значениями. В таких случаях функция оценивается и возвращается результат функции.

5.1 Имена слов

Слова состоят из букв, цифр и любых из следующих символов:

```
? ! . ' + - * & | = _ ~
```

Слово не может начинаться с числа, а также существуют некоторые ограничения на слова, которые можно интерпретировать как числа. Например, -1 и +1 - это числа, а не слова.

Конец слова отмечается пробелом, новой строкой или одним из следующих символов:

```
[ ] ( ) { } " : ; /
```

Таким образом, квадратные скобки блока не являются частью слова. Например, следующий блок содержит слово test :

```
[test]
```

В словах нельзя использовать следующие символы, так как они могут привести к неправильной интерпретации слов или возникновению ошибки:

```
@ # $ % ^ ,
```

Слова могут быть любой длины, но слова не могут выходить за конец строки:

```
это-очень-длинное-слово-используется-как-пример
```

Следующие строки содержат примеры допустимых слов:

```
Copy print test
number?  time?  date!
image-files  l'image
++ -- == +-
***** *new-line*
left&right left|right
```

REBOL не чувствителен к регистру. Все следующие слова относятся к одному и тому же слову:

```
синий
Синий
СИНИЙ
```

Регистр слова сохраняется при печати.

Слова можно использовать повторно. Значение слова зависит от его контекста, поэтому слова можно повторно использовать в разных контекстах. В REBOL нет ключевых слов. Вы можете повторно использовать любое слово, даже те, которые предопределены в REBOL. Например, вы можете использовать слово **if** в вашем коде иначе, чем интерпретатор REBOL использует это слово.

Выбирайте хорошие слова

Тщательно выбирайте слова, которые вы используете. Слова используются для обозначения смысла. Если вы правильно подберете слова, вам и другим будет легче понять ваш сценарий.

5.2 Использование слов

Слова используются двумя способами: как ***символы*** или как ***переменные***. В следующем блоке слова используются как символы для цветов.

```
[red green blue]
```

В следующей строке:

```
print second [red green blue]
green
```

слова не имеют другого значения, кроме их использования в качестве названий цветов. Все слова, используемые в блоках, служат символами до тех пор, пока не будут оценены.

Когда слово оценивается, оно используется как переменная. В предыдущем примере слова **print** и **second** - это переменные, которые содержат собственные функции, выполняющие необходимую обработку.

Слово может быть написано четырьмя способами, чтобы указать, как с ним следует обращаться, подробней показано в форматах слова.

Формат	Что делает
слово	Оценивает слово. Это наиболее естественный и распространённый способ написания слов. Если слово содержит функцию, она будет оценена(выполнена). В противном случае будет возвращено значение слова.
слово:	Определяет или устанавливает значение слова. Слову присваивается новое значение. Значение может быть любым, включая функцию. См. Раздел "Установка слов" ниже.
:слово	Получает значение слова, но не оценивает его. Это полезно для обращения к функциям и другим данным без их оценки. См. Раздел "Получение слов" ниже.
'слово	Считает слово набором символов и не оценивает его. Само слово - это значение.

5.3 Установка слов

Слово, за которым следует двоеточие (:), используется для определения или установки его значения:

```
возраст: 42
обед: 12:32
день рождения: 20 марта 1990 г.
город: "Додж Сити"
тест: %stuff.r
```

Вы можете установить для слова значение любого типа. В предыдущих примерах слова определены как целые числа, время, дата, строка и значения файла. Вы также можете сделать слова более сложными типами значений. Например, для значений блоков и функций используются следующие слова:

```
towns: ["Ukiah" "Willits" "Mendocino"]
code: [if age > 32 [print town]]
say: func [item] [print item]
```

Почему слова установлены таким образом

Во многих языках слова ставятся со знаком равенства, например:

```
возраст = 42
```

В REBOL слова начинаются с двоеточия. Причина этого важна. Он превращает операцию множества над словами в одно лексическое значение. Представление операции над множествами является ***атомарным***.

Разницу между двумя подходами можно увидеть на этом примере:

```
print length? [возраст: 42]
2
print length? [возраст = 42]
3
```

REBOL - это ***рефлексивный*** язык, он умеет манипулировать собственным кодом. Этот метод установки значений позволяет вам писать код, который легко управляет операциями с `набором слов` как единым целым.
Конечно, другая причина в том, что знак равенства (=) используется в качестве оператора сравнения.

За один раз можно задать несколько слов путем каскадирования определений слов. Например, каждое из следующих слов имеет значение `42:`

```
возраст: номер: размер: 42
```

Слова также могут быть установлены с помощью функции `set`:

```
set 'time 10:30
```

В этом примере строка устанавливает слово `time` значению 10:30. Слово `time` записывается как буквальное (с использованием одинарной кавычки (') в начале слова), поэтому оно не будет оцениваться.

Функция **set** также может устанавливать несколько слов:

```
set [number num ten] 10

print [number num ten]
10 10 10
```

В приведённом выше примере обратите внимание, что слова не нужно заключать в кавычки, потому что они находятся внутри блока, который не оценивается. Функция **print** показывает, что каждое слово имеет целое число `10`.

Если функцией **set** задан блок значений, каждое из отдельных значений устанавливается в слова. В этом примере один, два и три установлены на 1, 2 и 3:

```
set [один два три] [1 2 3]

print три
3
print [один два три]
1 2 3
```

См. Раздел «Слова» в Приложении «Значения» для получения дополнительной информации о настройке слов.

5.4 Получение слов

Чтобы получить значение слова, которое было определено ранее, поместите двоеточие (:) перед словом. Слово с префиксом двоеточие получает значение слова, но не выполняет его дальнейшую оценку, если это функция. Например, следующая строка:

```
drucken: :print
```

определяет новое слово, `drucken` (что по-немецки означает печать), для обозначения той же функции, которую выполняет **print**. Это возможно, потому что **:print** возвращает функцию для печати , но не оценивает ее.

Теперь, `drucken` выполняет ту же функцию, что и **print**:

```
drucken "test"
test
```

Для **print** и `drucken` установлено одно и то же значение - функция, выполняющая печать.

Это также можно сделать с помощью функции **get** . Когда дано буквальное слово, **get** возвращает его значение, но не оценивает его:

```
stampa: get 'print

stampa "test"
test
```

Способность узнать значение слова также важна, если вы хотите определить значение слова, не оценивая его. Например, вы можете определить, является ли слово собственной функцией, используя следующую строку:

```
print native? :if
true
```

Здесь get возвращает функцию для **if**. Функция **if** не оценивается, а передаётся встроенной функции **native?**, которая проверяет, является ли это собственным типом данных. Без двоеточия функция if будет оценена, и, поскольку у неё нет аргументов, возникнет ошибка.

5.5 Буквальные слова

Умение воспринимать слово как буквальное полезно. И **set**, и **get**, а также другие функции, такие как **value?**, **unset**, **protect**, и **unprotect**, ожидают буквального значения.
Буквальные слова могут быть записаны одним из двух способов: поставив перед словом одинарную кавычку, также известную как галочка, (') или поместив слово в блок.
Вы можете поставить галочку перед оцениваемым словом:

```
word: 'this
```

В приведённом выше примере для переменной `word` установлено буквальное слово `this`, а не значение `this`. Слово переменной `word` просто использует имя символически. В приведённом ниже примере показано, что если вы напечатаете значение `word`, вы увидите `this`:

```
print word
this
```

Вы также можете получить буквальные слова из неоцененного блока. В следующем примере функция **first** выбирает первое слово из блока. Затем это слово присваивается в виде значения переменной word.

```
word: first [this and that]
```

Любое слово может использоваться как буквальное. Это может относиться или не относиться к значению. Например, в примере ниже слово `here` не имеет значения. Слово **print** имеет значение, но его можно использовать как литерал, потому что буквальные слова не оцениваются.

```
word: 'here
print word
here
word: 'print
print word
print
```

Следующий пример иллюстрирует важность буквальных значений:

```
video: [
    title "День Независимости"
    length 2:25:24
    date   4/july/1996
]
print select video 'title
День Независимости
```

В этом примере слово `title` ищется в блоке. Если галочка отсутствовала в заголовке, будет использовано её естественное значение. Если `title` не имеет естественного значения, отображается ошибка.
См. Раздел «Слова» в приложении «Значения» для получения дополнительной информации о словесных литералах..

5.6 Неустановленные слова

Слово, не имеющее значения, не установлено . Если оценивается неустановленное `(unset)` слово, возникает ошибка:

```
>> outlook
** Ошибка сценария: Outlook не имеет значения.
** Где: outlook
```

Сообщение об ошибке в предыдущем примере указывает, что для слова не задано значение. Значение слова не установлено. Не путайте это со словом, для которого установлено значение `none`, что является допустимым значением.
Ранее определённое слово можно сбросить в любой момент с помощью команды **`unset`**:

```
unset 'word
```

Когда слово не установлено, его значение теряется.

Чтобы определить, было ли установлено слово, используйте функцию **`value?`**, которая принимает буквальное слово в качестве аргумента:

```
if not value? 'word [print "слово word не заданно"]
слово word не заданно
```

Определение того, установлено ли слово, может быть полезно в сценариях, вызывающих другие сценарии. Например, скрипт может установить параметр по умолчанию, который ранее не был установлен:

```
if not value? 'test-mode [test-mode: on]
```

5.7 Защита слов

Вы можете предотвратить установку слова с помощью функции **protect**:

```
protect 'word
```

Попытка переопределить защищённое слово вызывает ошибку:

```
word: "here"
** Ошибка скрипта: Слово Word защищено, его нельзя изменить.
** Где: слово: "here"
```

Слово также можно снять с защиты с помощью **unprotect**:

```
unprotect 'word
word: "here"
```

Функции **protect** и **unprotect** также принимают блок слов:

```
protect [this that other]
```

Важные функции и системные слова могут быть защищены с помощью функции **protect-system**. Защита функциональных и системных слов особенно полезна для новичков, которые могут случайно задать важные слова. Если **protect-system** помещается в ваш пользовательский файл `user.r`, тогда все предопределённые слова будут защищены.

6. Условная оценка

Как упоминалось ранее, блоки обычно не оцениваются. Для принудительной оценки блока требуется функция **do**. Бывают случаи, когда вам может потребоваться условная оценка блока. В следующем разделе описано несколько способов сделать это.

6.1 Условные блоки

Функция **if** принимает два аргумента. Первый аргумент - это условие, а второй аргумент - это блок. Если условие `истинна`(`true`), блок оценивается (выполняется), в противном случае он не оценивается (не выполняется).

```
if now/time > 12:00 [print "после полудня"]
после полудня
```

Условие обычно является выражением, которое принимает `истинное` (`true`) или `ложное` (`false`) значение; однако можно указать и другие значения. Только значение `false` или `none` препятствует оценке блока. Все остальные значения (включая ноль(!)) обрабатываются как истинные и приводят к оценке блока. Это может быть полезно для проверки результатов **find**, **select**, **next** и других функций, которые не возвращают `ничего` (`none`):

```
string: "давайте поговорим о REBOL"
if find string "поговорим" [print "найдено"]
найдено
```

Функция **either**(либо) расширяет функцию **if**(если). Она включает третий аргумент, который является блоком для оценки, когда условие ложно:

```
either now/time > 12:00 [
    print "после полудня"
][
    print "до полудня"
]
после полудня
```

Функция **either** также интерпретирует значение none как false.

Обе функции **if** и **either** возвращают результат оценки своих блоков. В случае **if** значение блока возвращается только в том случае, если блок оценивается; в противном случае возвращается none. Функция **if** полезна для условной инициализации переменных::

```
flag: if time > 13:00 ["обед уже съеден"]

print flag
обед уже съеден
```

Используя результат функции **either**, предыдущий пример можно переписать следующим образом:

```
flag: either now/time > 13:00 [
    "обед уже съеден"
][
    "обед ещё цел"
]

print flag
обед уже съеден
```

Поскольку и **if**, и **either** являются функциями, их аргументы блока могут быть любым выражением, результатом которого является блок при оценке. В следующих примерах слова используются для представления аргумента блока для **if** и **either**.

```
notice: [print "Проснись!"]
if now/time > 7:00 notice
Wake up!
notices: [
    [print "Уже восход!"]
    [print "Уже полдень!"]
    [print "Уже закат!"]
]
if now/time > 12:00 second notices
Уже полдень!
sleep: [print "...продолжай спать..."]
either now/time > 7:00 notice sleep
Проснись!
```

Условные выражения, используемые для первого аргумента как **if**, так и **either**, могут быть составлены из большого количества функций сравнения и логики. Обратитесь к главе «Математика» за дополнительной информацией.

Избегайте этой распространённой ошибки

Наиболее частая ошибка в REBOL - это забыть второй блок для **either** или добавить второй блок для **if**. Эти примеры как создатели трудно найти ошибки:

```
either age > 10 [print "старше"]

if age > 10 [print "старше"] [print "моложе"]
```

Эти типы ошибок сложно обнаружить, поэтому имейте это в виду, если кажется, что эти функции не работают так, как вы ожидаете.

6.2 Any (любые) и all (все)

Функции **any**(любой) и **all**(все) позволяют быстро оценивать некоторые типы условных выражений. Эти функции можно использовать по-разному: вместе с **if**, **either** и другими условными функциями, либо отдельно.

И **any**, и **all** принимают блок выражений, которые оцениваются по одному выражению за раз. Функция **any** возвращает первое истинное(true) выражение, а функция **all** возвращает первое ложное(false) выражение. Имейте в виду, что ложное выражение также может быть none, а истинное выражение - это любое значение, кроме false или none.

Функция **any** возвращает первое значение, которое не является ложным, в противном случае она не возвращает none. Функция **all** возвращает последнее значение, если все выражения не ложны, в противном случае возвращает none.

Обе функции **any** и **all** оценивают ровно столько, сколько им нужно. Например, если какое-либо выражение обнаружило истинное выражение, ни одно из оставшихся выражений не оценивается.

Вот пример использования **any**::

```
size: 50
if any [size < 10 size > 90] [
    print "Размер вне допустимого диапазона."
]
```

Поведение **any** также полезно для установки значений по умолчанию. Например, в следующих строках устанавливается число 100, но только если его значение none:

```
number: none
print number: any [number 100]
100
```

Точно так же, если у вас есть различные потенциальные значения, вы можете использовать первое, у которого действительно есть значение (не none):

```
num1: num2: none
num3: 80
print number: any [num1 num2 num3]
80
```

Вы можете использовать **any** с такими функциями, как **find**, чтобы всегда возвращать действительный результат:

```
data: [123 456 789]
print any [find data 432 999]
999
```

Точно так же **all** можно использовать для условий, требующих, чтобы все выражения были true(истинными):

```
if all [size > 10 size < 90] [print "Размер в требуемом диапазоне"]
Размер в требуемом диапазоне
```

Вы можете убедиться, что значения были установлены до оценки функции::

```
a: "REBOL/"
b: none
probe all [string? a string? b append a b]
none
b: "Core"
probe all [string? a string? b append a b]
REBOL/Core
```

6.3 Условные циклы

Функции **until** and **while** fповторяют оценку блока до тех пор, пока не будет выполнено условие.

Функция **until** повторяет блок до тех пор, пока оценка блока не вернёт true (то есть не false или none). Блок оценки всегда оценивается хотя бы один раз. Функция **until** возвращает значение своего блока.

В приведённом ниже примере будет напечатано каждое слово в цветовом блоке. Блок начинается с печати первого слова блока. Затем он переходит к следующему цвету для каждого цвета в блоке. Функция **tail?**(хвост?) проверяет конец блока и вернёт true, что приведёт к завершению работы функции **until**.

```
color: [красный зелёный синий]
until [
    print first color
    tail? color: next color
]
красный
зелёный
синий
```

Функция **break**(прервать) может быть использована, чтобы уйти из цикла **until** в любое время.

Функция **while** повторяет оценку двух своих аргументов блока, в то время как первый блок возвращает true. Первый блок - это блок условий, второй блок - блок оценки. Когда блок условий возвращает false или none, блок выражения больше не будет оцениваться(выполняться), и цикл завершится.

Вот пример, аналогичный показанному выше. Цикл **while** будет продолжать печатать цвет до тех пор пока не в цвета ещё напечатаны.

```
color: [красный зелёный синий]
while [not tail? color] [
    print first color
    color: next color
]
красный
зелёный
синий
```

Блок условия может содержать любое количество выражений, если последнее выражение возвращает условие. Чтобы проиллюстрировать это, в следующем примере в блок условия добавляется печать. Это напечатает значение индекса цвета. Затем он проверит наличие хвоста цветового блока, что является условием, используемым для цикла.

```
color: [red green blue]
while [
    print index? color
    not tail? color
][
    print first color
    color: next color
]
1
red
2
green
3
blue
4
```

Последнее значение блока возвращается из функции **while**.
Break можно использовать для выхода из цикла в любой момент времени.

6.4 Распространённые ошибки

Условные выражения являются ложными только тогда, когда они возвращают `false` или `none`, и они `true`, когда они возвращают любое другое значение. Все условные выражения в следующих примерах возвращают `true`, даже нулевые и пустые значения блоков:

```
if true [print "ага"]
ага
if 1 [print "ага"]
ага
if 0 [print "ага"]
ага
if [] [print "ага"]
ага
```

Следующие условные выражения возвращают `false`:

```
if false [print "ага"]

if none [print "ага"]
```

Не заключайте условные выражения в блок. Условные выражения, заключённые в блоки, всегда возвращают `true` результат:

```
if [false] [print "ara"]
ara
```

Не путайте **either** с **if**. Например, если вы собираетесь написать:

```
either some-condition [a: 1] [b: 2]
```

но напишите вместо этого:

```
if some-condition [a: 1] [b: 2]
```

Функция **if** будет игнорировать второй блок. Это не вызовет ошибки, но второй блок никогда не будет оценён.

Обратное тоже верно. Если вы напишете следующую строку, опуская второй блок:

```
either some-condition [a: 1]
```

Функция **either** не будет оценивать правильный код и может привести к ошибочному результату.

7. Повторная оценка

Функции **while** и **until**, указанные выше, используются для цикла до тех пор, пока не будет выполнено условие. Также есть несколько функций, которые позволяют выполнять цикл заданное количество раз.

7.1 Loop

Функция **loop** оценивает блок заданное количество раз. В следующем примере выводится строка из 40 тире:

```
loop 40 [prin "-"]
----------------------------------------
```

Обратите внимание, что функция **prin** похожа на функцию **print** , но выводит свой аргумент без завершения строки.

Функция **loop** возвращает значение окончательной оценки блока:

```
i: 0
print loop 40 [i: i + 10]
400
```

7.2 Repeat

Функция **repeat** расширяет **loop**, позволяя вам контролировать счётчик цикла. В **repeat** первый аргумент функции является слово, которое будет использоваться для хранения значения счётчика:

```
repeat count 3 [print ["count:" count]]
count: 1
count: 2
count: 3
```

Также возвращается окончательное значение блока:

```
i: 0
print repeat count 10 [i: i + count]
55
```

В предыдущих примерах слово `count` имеет своё значение только в блоке **repeat**. Другими словами, значение `count` является локальным для блока. После завершения **repeat** `count` возвращается к предыдущему установленному значению.

7.3 For

Функция **for** расширяет **repeat**, позволяя указать начальное значение, конечное значение и приращение к значению. Любое из значений может быть положительным или отрицательным.

Пример ниже начинается с нуля и отсчитывается до 50 с увеличением на 10 каждый раз в цикле.

```
for count 0 50 10 [print count]
0
10
20
30
40
50
```

Функция **for** проходит цикл до конечного значения включительно. Однако, если счётчик превышает конечное значение, цикл все равно завершается. В приведённом ниже примере указано конечное значение 55. Это значение никогда не будет достигнуто, потому что цикл увеличивается на 10 каждый раз. Цикл останавливается на 50.

```
for count 0 55 10 [prin [count " "]]
0 10 20 30 40 50
```

В следующем примере показано, как вести обратный отсчет. Он начинается с четырех и отсчитывается до нуля по одному.

```
for count 4 0 -1 [print count]
4
3
2
1
0
```

Функция **for** также работает с десятичными числами, деньгами, временем, датами, сериями и символами. Убедитесь, что начальное и конечное значения имеют один и тот же тип данных. Вот несколько примеров использования цикла **for** с другими типами данных.

```
for count 10.5 0.0 -1 [prin [count " "]]
10.5 9.5 8.5 7.5 6.5 5.5 4.5 3.5 2.5 1.5 0.5
for money $0.00 $1.00 $0.25 [prin [money " "]]
$0.00 $0.25 $0.50 $0.75 $1.00
for time 10:00 12:00 0:20 [prin [time " "]]
10:00 10:20 10:40 11:00 11:20 11:40 12:00
for date 1-jan-2000 4-jan-2000 1 [prin [date " "]]
1-Jan-2000 2-Jan-2000 3-Jan-2000 4-Jan-2000
for char #"a" #"z" 1 [prin char]
abcdefghijklmnopqrstuvwxyz
```

Функция **for** также работает с сериями. В следующем примере **for** используется для строкового значения. Слово `end` определяется как строка с текущим индексом у символа `d`.

Функция **for** перемещается по серии строк по одному символу за раз и останавливается, когда достигает позиции символа, определённой как `end`:

```
str: "abcdef"
end: find str "d"
for s str end 1 [print s]
abcdef
bcdef
cdef
def
```

7.4 Foreach

Функция **foreach** предоставляет удобный способ повторить вычисление блока для каждого элемента серии. Он работает для всех типов блочных и строчных серий.

В примере ниже каждое слово в блоке будет напечатано:

```
цвета: [красный зелёный синий]
foreach цвет цвета [print цвет]
красный
зелёный
красный
```

В следующем примере каждый символ в строке будет напечатан:

```
строка: "REBOL"
foreach символ строка [print символ]
R
E
B
O
L
```

В приведённом ниже примере будет напечатано каждое имя файла, содержащее ".t" в текущем каталоге:

```
files: read %.
foreach file files [
    if find file ".t" [print file]
]
file.txt
file2.txt
newfile.txt
output.txt
```

Когда блок содержит группы связанных значений, функция **foreach** может получить все значения группы одновременно. Например, вот блок, который содержит время, строку и цену. Предоставляя функции **foreach** блок слов для группы, можно выбрать и распечатать каждое из их значений.

```
Фильмы: [
     8:30 "Контакт"        $4.95
    10:15 "Планета Земля" $3.25
    12:45 "Матрица"        $4.25
]

foreach [время название цена] фильмы [
    print ["Сегодня в кино фильм " название " в " время ", цена билета " цена]
]
Сегодня в кино фильм  Контакт  в  8:30 цена билета  $4.95
Сегодня в кино фильм  Планета Земля  в  10:15 цена билета  $3.25
Сегодня в кино фильм  Матрица  в  12:45 цена билета  $4.25
```

В приведённом выше примере блок значений **foreach**, [`время название цена`], указывает, что три значения должны быть извлечены из `фильмы` для каждой оценки блока (каждого круга).

Переменные, используемые для хранения значений **foreach**, являются локальными для блока. Их значение устанавливается только в внутри повторяющегося блока. После выхода из цикла переменные возвращаются к своим ранее установленным значениям.

7.5 Forall и Forskip

Подобно **foreach**, функция **forall** оценивает блок для каждого значения в серии. Однако есть несколько важных отличий. Функция **forall** передаёт серию, установленную в начало цикла. По мере прохождения цикла **forall** изменяет позицию в серии.

```
цвета: [красный зелёный синий]
forall цвета [print first цвета]
красный
зелёный
синий
```

В приведённом выше примере после каждой оценки блока серия продвигается к следующей позиции. Когда **forall** возвращается, индекс `цвета` находится в конце ряда.

Чтобы продолжить использовать серию, вам нужно будет вернуть ее в исходное положение с помощью следующей строки:

```
цвета: head цвета
```

Функция **forskip** оценивает блок для групп значений в серии. Второй аргумент **forskip** - это количество элементов, которые нужно продвинуть вперёд после каждого цикла цикла.

Как и **forall**, **forskip** передаётся серия с индексом серии, установленным в том месте, где она должна начинаться. Затем **forskip** изменяет позицию индекса, продолжая цикл. После каждой оценки основного блока индекс серии продвигается на величину пропуска до следующей позиции индекса.

Следующий пример демонстрирует **forskip**:

```
Фильмы: [
     8:30 "Контакт"       $4.95
    10:15 "Планета Земля" $3.25
    12:45 "Матрица"       $4.25
]

forskip Фильмы 3 [print second Фильмы]
Контакт
Планета Земля
Матрица
```

В приведённом выше примере **forskip** возвращается с серией фильмов в хвосте. Вам нужно будет использовать функцию **head**, чтобы вернуть серию в исходное положение.

7.6 Forever

Функция **forever** оценивает блок бесконечно или до тех пор, пока он не встретит функцию **break**.

В следующем примере функция **forever** используется для проверки существования файла каждые десять минут:

```
forever [
    if exists? %datafile [break]
    wait 0:10
]
```

7.7 Break

Вы можете остановить повторную оценку блока с помощью функции **break**. Функция **break** полезна, когда встречается особое условие и цикл необходимо остановить. Функция **break** работает со всеми типами циклов.

В следующем примере цикл прерывается (**break**), если число больше 5.

```
repeat count 10 [
    if (random count) > 5 [break]
    print "проверка"
]
проверка
проверка
проверка
```

Функция **break** не возвращает значение из цикла, если не используется уточнение(опция) **/return**:

```
print repeat count 10 [
    if (random count) > 5 [break/return "стоп тут"]
    print "проверка"
    "нормальный выход"
]
проверка
проверка
проверка
стоп тут
```

В приведённом выше примере, если повтор завершается без возникновения условия, блок возвращает строку `нормальный выход`. В противном случае команда **break/return** вернёт строку `стоп тут`.

8. Выборочная оценка

Есть несколько методов выборочной оценки выражений в REBOL. Эти методы предоставляют возможность ветвления оценки множеством различных способов на основе значения ключа.

8.1 Select

Функция **select** часто используется для получения определённого значения или блока по заданному значению. Если вы определяете блок значений и действий, вы можете использовать **select** для поиска действия, соответствующего значению.

```
cases: [
    center [print "center"]
    right  [print "right"]
    left   [print "left"]
]
action: select cases 'right
if action [do action]
right
```

В предыдущем примере функция `select` находит слово `right` и возвращает следующий за ним блок. (Если по какой-то причине блок не был найден, то возвращается `none`.) Затем блок оценивается. Значения, используемые в примере, являются словами, но они могут быть любыми значениями:

```
cases: [
     5:00 [print "везде"]
    10:30 [print "здесь"]
    18:45 [print "там"]
]
action: select cases 10:30
if action [do action]
здесь
```

8.2 Switch

Функция **select** используется так часто, что существует её специальная версия, с именем **switch**, которая включает оценку результирующего блока. Функция **switch** упрощает выполнение встроенной выборочной оценки. Например, чтобы включить простой числовой регистр:

```
switch 22 [
    11 [print "здесь"]
    22 [print "там"]
]
там
```

Функция **switch** также возвращает значение блока, которое она оценивает, поэтому предыдущий пример также можно записать как:

```
str: copy "прямо "

print switch 22 [
    11 [join str "там"]
    22 [join str "тут"]
]
прямо тут
```

и:

```
автомобиль: pick [Форд Хонда Лада] random 3
print switch автомобиль [
    Форд  [351 * 1.4]
    Хонда [454 * 5.3]
    Лада [154 * 3.5]
]
2406.2
```

Регистры могут быть любым допустимым типом данных, включая числа, строки, слова, даты, время, URL-адреса и файлы. Вот некоторые примеры:
Строки:

```
персона: "ребёнок"
switch персона [
    "папа" [print "тут"]
    "мама" [print "там"]
    "ребёнок" [print "везде"]
]
везде
```

Слова:

```
person: 'kid
switch person [
    dad [print "here"]
    mom [print "there"]
    kid [print "everywhere"]
]
everywhere
```

Тип данных:

```
person: 123
switch type?/word [
    string! [print "это строка"]
    binary! [print "это двоичные"]
    integer! [print "это целое число"]
    decimal! [print "это десятичное число"]
]
это целое число
```

Файлы:

```
file: %rebol.r
switch file [
    %user.r [print "тут"]
    %rebol.r [print "везде"]
    %file.r [print "там"]
]
везде
```

Интернет адреса (URLs):

```
url: ftp://ftp.rebol.org
switch url [
    http://www.rebol.com [print "тут"]
    http://www.cnet.com [print "там"]
    ftp://ftp.rebol.org [print "везде"]
]
везде
```

Тэги:

```
tag: <LI>
print switch tag [
    <PRE>   ["Предварительно отформатированный текст"]
    <TITLE> ["Заголовок страницы"]
    <LI>    ["Элемент маркированного списка"]
]
Элемент маркированного списка
```

Время:

```
time: 12:30
switch time [
     8:00 [send wendy@domain.dom "Эй! Просыпайся!!!"]
    12:30 [send cindy@rebol.dom "Пойдём обедать."]
    16:00 [send group@every.dom "Ужин будет у кого?"]
]
```

8.2.1 Случай по умолчанию

Случай по умолчанию может быть указан, если ни один из других вариантов не соответствует. Используйте **default**, чтобы указать значение по умолчанию:

```
time: 7:00
switch/default time [
     5:00 [print "везде"]
    10:30 [print "тут"]
    18:45 [print "там"]
] [print "нигде"]
нигде
```

8.2.2 Общие случаи

Если у вас есть общие случаи, когда результат будет одинаковым для нескольких значений, вы можете определить слово для хранения общего блока кода:

```
case1: [print length? url]   ; общий блок кода

url: http://www.rebol.com
switch url [
    http://www.rebol.com case1
    http://www.cnet.com [print "тут"]
    ftp://ftp.rebol.org case1
]
20
```

8.2.3 Другие случаи

Для кейсов можно оценивать не только блоки. В этом примере оценивается файл, соответствующий дню недели: переключиться сейчас / будний день [1% понедельник.р 5% пятница.р 6% суббота.р] Итак, если сегодня пятница, проверяется файл пятницы .r, и его результат возвращается переключателем. Этот тип оценки также работает для URL-адресов: время переключения [8:30 ftp://ftp.rebol.org/wakeup.r 10:30 http://www.rebol.com/break.r 18:45 ftp://ftp.rebol.org/sleep. r] Варианты для switch заключены в блок и поэтому могут быть определены отдельно от оператора switch:

Для кейсов можно оценивать не только блоки. В этом примере оценивается(выполняются) файл, соответствующий дню недели:

```
switch now/weekday [
    1 %понедельник.r
    5 %пятница.r
    6 %суббота.r
]
```

Итак, если сегодня пятница, проверяется скрипт `пятница.r`, и его результат возвращается переключателем. Этот тип оценки также работает для URL-адресов:

```
switch time [
     8:30 ftp://ftp.rebol.org/wakeup.r
    10:30 http://www.rebol.com/break.r
    18:45 ftp://ftp.rebol.org/sleep.r
]
```

Варианты для **switch** заключены в блок и поэтому могут быть определены отдельно от оператора **switch**:

```
расписание: [
     8:00 [send wendy@domain.dom "Подъём"]
    12:30 [send cindy@dom.dom "Пойдём обедать."]
    16:00 [send group@every.dom "Угостишь ужином?"]
]

switch 8:00 расписание
```

9. Остановка оценки

Оценка сценария может быть остановлена в любой момент, нажав клавишу выхода (ESC) на клавиатуре или используя функции **halt** и **quit**.
Функция **halt** function stops evaluation and returns you to the REBOL console prompt:

```
if time > 12:00 [halt]
```

The **quit** останавливает оценку и возвращает вас к приглашению консоли REBOL:

```
if error? try [print test] [quit]
```

10. Тестирование блоков

Бывают случаи, когда вы хотите оценить блок, но в случае возникновения ошибки вы не хотите останавливать оценку остальной части вашего скрипта.
Например, вы можете выполнять деление чисел, но не хотите, чтобы ваш скрипт останавливался, если происходит деление на ноль.
Функция **try** позволяет обнаруживать ошибки во время оценки блока. Это почти идентично функции **do**.
Функция **try** обычно возвращает результат блока; однако в случае возникновения ошибки вместо этого будет возвращено значение ошибки.
В следующем примере, когда происходит деление на ноль, сценарий передаст ошибку обратно в функцию **try**, и оценка продолжится с этой точки.

```
for num 5 -5 -1 [
    if error? try [print 10 / num] [print "ошибка"]
]
2
2
2.5
3.33333333333333
5
10
ошибка
-10
-5
-3.33333333333333
-2.5
-2
```

Подробнее об обработке ошибок можно узнать в Приложении «Ошибки»х.

Глава 5 - Скрипты

Содержание:

1. Обзор
  1.1 Суффикс файла
  1.2 Структура
2. Заголовки
3. Аргументы сценария
  3.1 Параметры программы
4. Запуск сценариев
  4.1 Загрузка сценариев
  4.2 Сохранение сценариев
  4.3 Комментирование сценариев
5. Руководство по стилям
  5.1 Форматирование
  5.2 Имена слов
  5.3 Заголовки сценариев
  5.4 Заголовки функций
  5.5 Имена файлов сценариев
  5.6 Встроенные примеры
  5.7 Встроенная отладка
  5.8 Сворачивание глобальных объектов
6. Очистка скрипта

1. Обзор

Термин ***script***(скрипт/сценарий) относится не только к отдельным файлам, которые оцениваются, но и к ***исходному*** тексту, встроенному в другие типы файлов (например, веб-страницы), или ***фрагментам*** исходного текста, которые сохраняются как файлы данных или передаются как сообщения.

1.1 Суффикс файла

Сценарии REBOL обычно добавляют суффикс(расширение) .r к именам файлов; однако это соглашение не требуется. Интерпретатор читает файлы с любым суффиксом и просматривает их содержимое на предмет наличия допустимого заголовка сценария REBOL.

1.2 Структура

Структура скрипта произвольная. Отступы и интервалы могут использоваться для пояснения структуры и содержания сценария. Кроме того, вам рекомендуется использовать стандартный стиль написания сценариев, чтобы сделать сценарии более удобочитаемыми. См. Руководство по стилю для получения дополнительной информации.

2. Заголовки

Непосредственно перед телом сценария каждый сценарий должен иметь заголовок, определяющий его назначение и другие атрибуты. Заголовок может содержать название сценария, имя автора, дату, версию, имя файла и дополнительную информацию. Файлы данных REBOL, которые не предназначены для прямой оценки, не требуют заголовка.

Заголовки полезны по нескольким причинам.

- Они идентифицируют сценарий как действительный исходный текст пригодный для интерпретации REBOL.
- Интерпретатор использует заголовок, чтобы распечатать заголовок сценария и определить, какие ресурсы и версии ему необходимы, прежде чем оценивать сценарий.
- Заголовки предоставляют стандартный способ сообщить название, смысл программы, автора и другие детали. Часто по заголовку сценария можно определить, интересен ли он вам.
- Архивы сценариев и веб-сайты используют заголовки для создания каталогов сценариев, категорий и перекрёстных ссылок.
- Некоторые текстовые редакторы получают доступ и обновляют заголовок скрипта, чтобы отслеживать такую информацию, как автор, дата, версия и история.

Общая форма заголовка скрипта:

```
REBOL [block]
```

Чтобы интерпретатор мог распознать заголовок, блок должен сразу следовать за словом **REBOL** . Между словом **REBOL** и блоком разрешены только пробелы (пробелы, табуляции и строки).

Блок, следующий за словом **REBOL**, представляет собой определение объекта, описывающего сценарий. Предпочтительный минимальный заголовок:

```
REBOL [
    Title:  "Сканер Web сайтов"
    Date:   2-Feb-2000
    File:   %webscan.r
    Author: "Jane Doer"
    Version: 1.2.3
]
```

Когда скрипт загружается, блок заголовка оценивается, и его словам присваиваются определённые значения. Эти значения используются интерпретатором, а также могут использоваться самим сценарием.

Обратите внимание, что слова, определённые как одно значение, также можно определить как несколько значений, предоставив их в блоке:

```
REBOL [
    Title: "Сканер Web сайтов"
    Date:   12-Nov-1997
    Author: ["Ema User" "Wasa Writer"]
]
```

Заголовки могут быть более сложными, предоставляя информацию об авторе, авторских правах, форматировании, требованиях к версии, истории изменений и многом другом. Поскольку блок используется для создания объекта заголовка, он также может быть расширен новой информацией. Это означает, что сценарий может расширять заголовок по мере необходимости, но это следует делать с осторожностью, чтобы избежать неоднозначной или избыточной информации.

Полный заголовок может выглядеть примерно так:

```
REBOL [
    Title:   "Full REBOL Header Example"
    Date:    8-Sep-1999
    Name:    'Full-Header  ; For window title bar

    Version: 1.1.1
    File:    %headfull.r
    Home:    http://www.rebol.com/rebex/

    Author:  "Carl Sassenrath"
    Owner:   "REBOL Headquarters"
    Rights:  "Copyright (C) Carl Sassenrath 1999"

    Needs:   [2.0 ODBC]
    Tabs:    4

    Purpose: {
        The purpose or general reason for the program
        should go here.
    }

    Note: {
        An important comment or notes about the program
        can go here.
    }

    History: [
        0.1.0 [5-Sep-1999 "Created this example" "Carl"]
        0.1.1 [8-Sep-1999 {Moved the header up, changed
            comment on extending the header, added
            advanced user comment.} "Carl"]
    ]

    Language: 'English
]
```

Предисловие сценария

Текст сценария не обязательно должен начинаться с заголовка. Скрипты могут начинаться с любого текста, что позволяет вставлять их в сообщения электронной почты, веб-страницы и другие файлы.

Заголовок отмечает начало сценария, а текст, который следует за ним, является телом сценария. Текст, который появляется перед заголовком, называется ***предисловием*** и игнорируется во время оценки.

```
Текст, который появляется перед заголовком, игнорируется.
от REBOL и может использоваться для комментариев, заголовков писем,
HTML-теги и т.д.

REBOL [
  Title:   "Пример предисловия"
  Date:    8-Jul-1999
]

print "Перед заголовком этого файла есть предисловие."
```

Встраивание скриптов в текст

Если за сценарием должен следовать другой текст, не связанный с самим сценарием, сценарий должен быть заключён в квадратные скобки [ ]:

```
Какой-то текст перед сценарием.
[
    REBOL [
        Title:   "Встроенный сценарий"
        Date:    8-Nov-1997
    ]
    print "done"
]
Какой-то текст после сценария.
```

Между начальной скобкой и словом REBOL допускается только пробел.

3. Аргументы скрипта

Когда сценарий оценивается/выполняется, он имеет доступ к информации о себе. Он находится в объекте `system/script`. Объект содержит поля, перечисленные в полях объекта для system/script.

Заголовок Родитель Путь Args

Header(Заголовок)	Объект заголовка скрипта. Это может быть использовано для доступа к заголовку скрипта, автору, версии, дате и другим полям.
Parent(Родитель)	Если сценарий был оценён/запущен из другого сценария, это объект system/script от родительского сценария.
Path(Путь)	Путь к каталогу файла или URL-адрес оцениваемого сценария.
Args(Аргументы)	Аргументы сценария. Они передаются из командной строки операционной системы или из функции **do**, которая использовалась для оценки/запуска скрипта.

Примеры использования объекта скрипта:

```
print system/script/title

print system/script/header/date

do system/script/args

do system/script/path/script.r
```

В последнем примере оценивается сценарий с именем `script.r` в том же каталоге, что и сценарий, который выполняется в данный момент.

3.1 Опции программы

Главная Путь к файлу определяется средой вашей операционной системы. Это путь, заданный вПеременная среды HOME или системный реестр для систем, которые ее поддерживают. Это путь, по которомуrebol.r и user.r файлы. Сценарий Имя файла исходного сценария, предоставленное при запуске интерпретатора. Путь Путь к текущему каталогу. Args Первоначальные аргументы, предоставляемые интерпретатору в командной строке. Do-arg C system / options

Сценарии также имеют доступ к параметрам, предоставленным интерпретатору REBOL при его запуске. Они находятся в объекте system/options. Объект содержит поля, перечисленные в полях объекта для system/options.

Home	Путь к файлу определяется вашей операционной системой. Это путь, заданный в переменной среды HOME или системный реестр для систем, которые его поддерживают. Это путь, по которому `rebol.r` и `user.r` файлы.
Script	Имя файла исходного сценария, указанного при запуске интерпретатора.
Path	Путь текущего каталога.
Args	Первоначальные аргументы, указанные интерпретатору в командной строке.
Do-arg	Строка, предоставленная в качестве аргумента для параметр --do в командной строке.

Объект system/options object также содержит дополнительные параметры, которые были предоставлены в командной строке. Тип

```
probe system/options
```

для проверки содержимого объекта параметров.

Примеры:

```
print system/options/script

probe system/options/args

print read system/options/home/user.r
```

4. Запуск сценариев

Есть два способа запустить сценарий: в качестве исходного сценария, указанного в командной строке, в виде аргумента, при запуске интерпретатора REBOL или из функции **do**.

Чтобы запустить сценарий при запуске интерпретатора, укажите имя сценария в командной строке после имени программы REBOL:

```
rebol script.r
```

Как только интерпретатор инициализируется, выполняется оценка сценария.

В функции **do** укажите имя файла сценария или URL-адрес в качестве аргумента. Файл загружается в интерпретатор и оценивается:

```
do %script.r

do http://www.rebol.com/script.r
```

Функция **do** возвращает результат скрипта, когда он завершает оценку.

Обратите внимание, что файл сценария должен включать допустимый заголовок REBOL.

4.1 Загрузка скриптов

Файлы сценариев могут быть загружены как данные с помощью функции **load**. Эта функция читает сценарий и переводит его в значения, слова и блоки, но не оценивает (не запускает) сценарий. Результатом функции **load** является блок, если было загружено только одно значение, тогда это значение возвращается.

Аргументом для функции **load** является имя файла, URL-адрес или строка.

```
load %script.r
load %datafile.txt
load http://www.rebol.org/script.r
load "print now"
```

Функция **load** выполняет следующие шаги:

- Читает текст из файла, URL-адреса или строки.
- Ищет заголовок скрипта, если он есть.
- Переводит данные, начинающиеся после заголовка, если они найдены.
- Возвращает блок, содержащий переведённые значения.

Например, если файл скрипта buy.r содержал текст:

```
Buy 100 shares at $20.00 per share
```

он может быть загружен строкой:

```
data: load %buy.r
```

что приведёт к формированию в переменной `data` блока:

```
probe data
[Buy 100 shares at $20.00 per share]
```

Следует отметить, что приведённый выше пример «Buy» - это диалект REBOL, а не непосредственно исполняемый код. См. Главу 4 о выражениях или главу 15 о синтаксическом анализе для получения дополнительной информации.

Обратите внимание, что файл не требует загрузки заголовка. Заголовок необходим только в том случае, если файл будет запускаться как сценарий.

Функция **load** поддерживает несколько уточнений (дополнительных параметров). В разделе «Усовершенствования функций загрузки» перечислены уточнения и описание их функций:

/header	Включает заголовок, если он есть.
/next	Загружает только следующее значение, по одному за раз. Это полезно для разбора скриптов REBOL.
/markup	Обрабатывает файл как файл HTML или XML и возвращает блок, содержащий его теги и текст.

Обычно **load** не возвращает заголовок из сценария. Но, если используется уточнение **/header**, возвращаемый блок содержит объект заголовка в качестве первого аргумента.

Уточнение **/next** загружает следующее значение и возвращает блок, содержащий два значения. Первое возвращаемое значение-это следующее значение из серии. Второе возвращаемое значение-это позиция строки, следующая сразу за последним загруженным элементом.

Уточнение **/markup** позволяет загрузить HTML и данные XML как блок тегов и строк. Все теги являются типами данных тегов. Все остальные данные обрабатываются как строки.

Если следующее содержимое файла загружено с **load/markup:**

```
<title>Это пример</title>
```

будет создан блок:

```
probe data
[<title> "Это пример" </title>]
```

4.2 Сохранение скриптов

Данные могут быть сохранены в файл сценария в формате, который можно загрузить в REBOL с помощью функции **load**. Это полезный способ сохранения значений данных и блоков данных. Таким образом можно создавать целые мини-базы данных.

Функция **save** ожидает два аргумента: имя файла и либо блок, либо значение для сохранения:

```
data: [Buy 100 shares at $20.00 per share]

save %data.r data
```

Данные записываются в формате исходного текста REBOL, который можно загрузить позже с помощью:

```
data: load %data.r
```

Также можно сохранять и загружать простые значения. Например, отметку даты можно сохранить с помощью:

```
save %date.r now
```

а затем перезагружен:

```
stamp: load %date.r
```

В предыдущем примере, поскольку `stamp` является единственным значением, при загрузке оно не заключено в блок.
Чтобы сохранить файл сценария с заголовком, заголовок может быть предоставлен в уточнении как объект или блок:

```
header: [Title: "Это пример"]
save/header %data.r data header
```

4.3 Комментирование скриптов

Комментирование полезно для пояснения назначения разделов скрипта. Заголовки сценария содержат высокоуровневое описание сценария, а комментарии содержат краткое описание функций. Также рекомендуется комментировать другие части кода.

Однострочный комментарий записывается после точки с запятой. Все, что стоит после точки с запятой и до конца строки, является комментарием, который полностью игнорируется интерпретатором:

```
zertplex: 10   ; установили значение на высшее качество
```

Вы также можете использовать несколько строк для комментариев. Например, вы можете создавать многострочные комментарии со строками, заключёнными в фигурные скобки:

```
{
    Это длинный
    при длинный
    многострочный
    комментарий.
}
```

Этот метод комментирования работает только в том случае, если строка не интерпретируется как аргумент функции. Если вы хотите быть уверенным в том, что многострочный комментарий распознаётся как комментарий и не интерпретируется как код, поставьте перед строкой слово `comment`:

```
comment {
    Это длинный
    при длинный
    многострочный
    комментарий.
}
```

Функция `comment` сообщает REBOL игнорировать следующий блок или строку. Обратите внимание, что строковые и блочные комментарии фактически являются частью блока скрипта. Следует проявлять осторожность, чтобы не помещать их в блоки данных, поскольку они могут появиться как часть данных.

5. Руководство по стилю

Скрипты REBOL имеют произвольную форму. Вы можете написать сценарий, используя отступы, интервалы, длину строки и терминаторы строки, которые вы предпочитаете. Вы можете поместить каждое слово в отдельную строку или объединить их в одну длинную строку.

Хотя форматирование вашего скрипта не влияет на интерпретатор, оно влияет на его удобочитаемость. По этой причине REBOL Technologies рекомендует вам следовать стандартному стилю сценариев, описанному в этом разделе.

Конечно, вам не обязательно следовать ни одному из этих предложений. Однако стиль написания сценариев важнее, чем кажется на первый взгляд. Это может иметь большое значение для удобочитаемости и повторного использования скриптов. Пользователи могут судить о качестве ваших скриптов по ясности их стиля. Неряшливые сценарии часто означают небрежный код. Опытные авторы сценариев обычно обнаруживают, что чистый, последовательный стиль облегчает создание, поддержку и исправление их кода.

5.1 Форматирование

Для ясности используйте следующие рекомендации по форматированию скриптов REBOL.

5.1.1 Отступ содержания для ясности

Содержимое блока имеет отступ, а квадратные скобки блока [] - нет. Это потому, что квадратные скобки относятся к предыдущему уровню синтаксиса, поскольку они определяют блок, но не являются его содержимым. Кроме того, легче заметить разрывы между соседними блоками, когда скобки выступают наружу.

По возможности открывающая квадратная скобка остается на строке со связанным с ней выражением. За закрывающей скобкой могут следовать другие выражения того же уровня. Эти же правила в равной степени применяются к круглым скобкам () и фигурным скобкам {}.

```
if check [сделайте то и то]

if check [
    сделай это и сделай то
    сделать другое дело
    сделай еще несколько операций
]

either check [сделайте что-нибудь короткое][
    сделай что-нибудь другое]

either check [
    когда выражение расширяется
    после конца блока ...
][
    это помогает сохранить
    читаемость и наглядность.
]

while [
    сделай более длинное выражение
    чтобы увидеть,
    правда ли это
][
    конец последнего блока
    и начало нового
    находятся на одном уровне
    и это визуально их разделяет
    между собой продолжая указывать
    на то, что они оба относятся
    к функции while
]

adder: func [
    "Это пример функции"
    arg1 "это первый аргумент"
    arg2 "это второй аргумент"
][
    arg1 + arg2
]
```

Исключение делается для выражений, которые обычно принадлежат одной строке, но распространяются на несколько строк:

```
if (это длинное условное выражение, которое
    делится на несколько строк и имеет отступ
)[
    так что это выглядит немного странно
]
```

Это также относится к сгруппированным значениям, которые принадлежат друг другу, но должны быть заключены в оболочку, чтобы поместиться в строке:

```
[
    "Hitachi Precision Focus" $1000 10-Jul-1999
        "Computers Are Us"

    "Nuform Natural Keyboard" $70 20-Jul-1999
        "The Keyboard Store"
]
```

5.1.2 Стандартный размер отступа (табуляции)

Стандартный размер табуляции REBOL - четыре пробела. Поскольку люди используют разные редакторы и читатели для сценариев, вы можете использовать пробелы, а не клавишу табуляции.

5.1.3 "Детаб" перед публикацией

Символ табуляции (ASCII 9) не делает отступ строго в четыре пробела во многих программах просмотра, браузерах или оболочках, поэтому используйте редактор или REBOL, чтобы заменить символы табуляции на пробелы перед его публикацией в сети. Следующая функция заменяет в файле символы табуляции на четыре пробела:

```
detab-file: func [file-name [file!]] [
    write file-name detab read file-name
]
detab-file %script.r
```

Следующая функция преобразует табуляцию с восемью пробелами в табуляцию с четырьмя пробелами:

```
detab-file: func [file-name [file!]] [
    write file-name detab entab/size read file-name 8
]
```

5.1.4 Ограничение длины строки до 80 символов.

Для удобства чтения и редактирования в разных пограммах рекомендуется ограничивать длину строки до 80 символов. Длинные строки, уходящие за край экрана, трудно и неудобно читать.

5.2 Имена слов

Слова - это первое знакомство пользователя с вашим кодом, поэтому очень важно тщательно выбирать слова. Сценарий должен быть чётким и лаконичным. По возможности, слова должны относиться к своему английскому или другому эквиваленту на человеческом языке простым и прямым образом. Ниже приведены стандартные соглашения об именах для REBOL.

5.2.1 Используйте самое короткое слово, передающее смысл

По возможности лучше всего подходят короткие, чёткие слова:

```
size  time  send  wait  make  quit
```

Локальные имена переменных часто можно сократить до одного слова. Более длинные и описательные слова лучше подходят для глобальных переменных.

5.2.2 По возможности используйте целые слова

То, что вы экономите при сокращении слов, редко того стоит. Лучше введите слово "дата_файла" полностью, а не "дф" или "файл_с_изображением", а не "фи". Это очень поможет при работе с исходным кодом вашего сценария.

5.2.3 Расстановка нескольких слов через дефис

Стандартный стиль - использовать дефисы или "нижний дефис" (символ подчёркивания), а не регистр символов, чтобы различать слова.

```
group-name image-file  clear-screen  bake-cake
файл_изображения очистка_экрана     общая_сумма
```

5.2.4 Начинайте имена функций с глагола

Имена функций начинаются с глагола, а за ними следует существительное, наречие или прилагательное. Некоторые существительные также могут использоваться как глаголы.

```
make  print  scan  find  show  hide  take
rake-coals  find-age  clear-screen
```

Избегайте лишних слов. Смысл слова "выход" также понятен, как и "выход-из-программы".

При использовании существительного в качестве глагола используйте специальные символы, например? где применимо. Например, функция для получения длины серии - это `длина?`. Другие функции REBOL, использующие это соглашение об именах:

```
size?  dir?  time?  modified?
```

5.2.5 Начинайте слова данных с существительных

Слова для объектов или переменных, содержащих данные, должны начинаться с существительного. При необходимости они могут включать модификаторы (прилагательные):

```
image  sound  big-file  image-files  start-time
```

5.2.6 Использование стандартных имён

В REBOL есть стандартные имена, которые следует использовать для аналогичных типов операций. Например:

```
make-blub            ;создание чего-то нового
free-blub            ;высвобождение ресурсов чего-то
copy-blub            ;копирование содержимого чего-либо
to-blub              ;преобразование во что-то
insert-blub          ;вставка чего-то
remove-blub          ;удаление чего-то
clear-blub           ;очистка чего-то
```

5.3 Заголовки скрипта

Преимущество использования заголовков очевидно. Заголовки предоставляют пользователям сводку сценария и позволяют другим сценариям обрабатывать информацию (например, сценарий каталогизации). Минимальный заголовок содержит заголовок, дату, имя файла и цель. Также могут быть предоставлены другие поля, такие как автор, примечания, использование и потребности/зависимости.

```
REBOL [
    Title: "Local Area Defringer"
    Date:  1-Jun-1957
    File:  %defringe.r
    Purpose: {
        Stabilize the wide area ignition transcriber
        using a double ganged defringing algorithm.
    }
]
```

5.4 Заголовки функций

Полезно предоставить описание в блоках спецификации функций. Ограничьте такой текст одной строкой из 70 символов или меньше. В описании укажите, какой тип значения обычно возвращает функция.

```
набор: func [
    "Вернуть стоимость набора текста"
    size "Размер рукописного текста"
    time "Время выполнения набора"
    /cost num "Максимальная стоимость"
    /compound "Составной расчёт"
][
    ...код функции...
]
```

5.5 Имена файлов сценариев

Лучший способ дать файлу имя - подумать о том, как лучше всего найти этот файл через несколько месяцев. Часто бывает достаточно коротких и понятных имён. Следует избегать множественного числа, если оно не имеет смысла.
Кроме того, при присвоении имени сценарию следует учитывать, как имя будет отсортировано в каталоге. Например, объедините связанные файлы, начав их с общего слова.

```
%net-start.r
%net-stop.r
%net-run.r
```

5.6 Встроенные примеры

При необходимости предоставьте примеры в сценарии, чтобы показать, как работает сценарий, и дать пользователям быстрый способ проверить правильность работы сценария в их системе.

5.7 Встроенная отладка

Часто бывает полезно встроить функции отладки как часть сценария. Это особенно верно в отношении сетевых сценариев и сценариев обработки файлов, в которых нежелательно отправлять и записывать файлы во время работы в тестовом режиме. Такие тесты можно включить с помощью контрольной переменной вначале скрипта.

```
отладка: вкл
check-data: off
```

5.8 Свернуть глобальные объекты

В больших скриптах и по возможности избегайте использования глобальных переменных, которые передают своё внутреннее состояние от одного модуля или функции к другому. Для коротких сценариев это не всегда практично. Но помните, что короткие сценарии со временем могут стать более длинными.

Если у вас есть набор глобальных переменных, которые тесно связаны между собой, рассмотрите возможность использования объекта для их представления:

```
пользователь: make object! [
    имя:       "Fred Dref"
    возраст: 94
    телефон: 707-555-1234
    почта:          dref@fred.dom
]
```

6. Очистка скрипта

Вот короткий скрипт, который можно использовать для очистки отступов скрипта. Он работает, анализируя синтаксис REBOL и реконструируя каждую строку скрипта. Этот пример можно найти в библиотеке сценариев REBOL на сайте www.REBOL.com.

```
out: none ; вывод текста через
spaced: off ; добавить дополнительный отступ
indent: "" ; содержит вкладки с отступом

emit-line: func [] [append out newline]

emit-space: func [pos] [
    append out either newline = last out [indent] [
        pick [#" " ""] found? any [
            spaced
            not any [find "[(" last out
                     find ")]" first pos]
        ]
    ]
]

emit: func [from to] [
    emit-space from append out copy/part from to
]
```

```
clean-script: func [
    "Возвращает новый текст скрипта со стандартным интервалом."
    script "Исходный текст сценария"
    /spacey "Необязательные пробелы возле скобок"
    /local str new
] [
    spaced: found? spacey
    out: append clear copy script newline
    parse script blk-rule: [
        some [
            str:
            newline (emit-line) |
            #";" [thru newline | to end] new:
                (emit str new) |
            [#"[" | #"("]
                (emit str 1 append indent tab)
                blk-rule |
            [#"]" | #")"]
                (remove indent emit str 1) |
            skip (set [value new]
                load/next str emit str new) :new
        ]
    ]
    remove out ; remove first char
]

script: clean-script read %script.r

write %new-script.r script
```

Глава 6 – Серии

Содержание:

1. Основные концепции
- 1.1 Переход к серии
- 1.2 Переход по порядку
- 1.3 Извлечение значений
- 1.4 Извлечение подсерии
- 1.5 Вставка и добавление
- 1.6 Удаление значений
- 1.7 Изменение значений

2. Серийные функции
- 2.1 Функции создания
- 2.2 Функции навигации
- 2.3 Информационные функции
- 2.4 Функции извлечения
- 2.5 Функции изменения
- 2.6 Функции поиска
- 2.7 Функции упорядочивания
- 2.8 Функции набора данных (группы)

3. Типы данных серий
- 3.1 Типы блоков
- 3.2 Типы строк
- 3.3 Псевдотипы
- 3.4 Функции проверки типов

4. Информация о серии
- 4.1 Length? (Длинна)
- 4.2 Head? (Голова)
- 4.3 Tail? (Хвост)
- 4.4 Index? (Индекс)
- 4.5 Offset? (Смещение)

5. Создание и копирование серии
- 5.1 Частичные Копии
- 5.2 Глубинные Копии
- 5.3 Исходные Копии

6. Серия Итерационные
- 6.1 Цикл Foreach
- 6.2 Цикл While
- 6.3 Цикл Forall
- 6.4 Цикл Forskip
- 6.5 Функция Break

7. Поиск в серии
- 7.1 Простой поиск
- 7.2 Доработка резюме
- 7.3 Частичные поиск
- 7.4 Позиция хвоста
- 7.5 Обратный поиск
- 7.6 Повторный поиск
- 7.7 Сопоставление
- 7.8 Поиск по шаблону
- 7.9 Выбор
- 7.10 Поиск и Замена

8. Сортировка серий
- 8.1 Простая сортировка
- 8.2 Групповая сортировка
- 8.3 Функции сравнения

9. Серия как набор данных
9.1 Unique (Уникальные)
9.2 Intersect (Пересечение)
9.3 Union (Объединение)
9.4 Exclude (Исключение)
9.5 Difference (Разница)
9.6 Exclude (Исключение)
10. Переменные нескольких серий
11. Уточнения модификации
11.1 Part (Часть)
11.2 Only (Только)
11.3 Dup (Дублирование)

1. Основные концепции

Концепция серии проста, и это фундаментальная концепция, которая используется везде и почти для всего в REBOL. Чтобы понять REBOL, вы должны понимать, как создавать и управлять сериями.

Серии представляет собой **набор** значений , расположенных в **определённом** порядке.

Например, это все серии:

```
1 2 3 4

A B C D

"ABCD"

10:30 4:20 7:11
```

В REBOL есть много типов серий. Блок, строка, список, URL-адрес, путь, электронная почта, файл, тег, двоичный файл, битовый набор, порт, хэш и изображение - все это серии, к которым можно получить доступ и обрабатывать одинаковым образом с использованием небольшого набором последовательных функций.

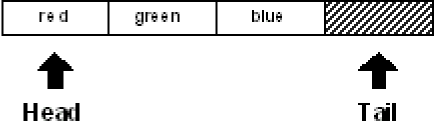
1.1 Переход по серии

Поскольку ряд представляет собой упорядоченный набор значений, вы можете перемещаться по нему из одной позиции в другую. В качестве примера возьмём серию из трёх цветов, определённых следующим блоком:

```
colors: [red green blue]
```

В этом блоке нет ничего особенного. Это серия из трёх слов. У него есть набор значений: red (красный), green (зелёный) и blue (синий). Значения организованы в определённом порядке: красный - первый, зелёный - второй, синий - третий..

Первая позиция блока называется `head` (голова). Эту позицию занимает слово `красный`. Последняя позиция блока называется `tail` (хвост). Это позиция сразу после последнего слова в блоке. Если бы вы нарисовали схему блока, это выглядело бы так:



Обратите внимание, что хвост находится `за концом` блока. Вскоре важность этого станет более ясной.

Переменная colors используются для обозначения блока. В настоящее время он установлен в заголовок блока:

```
print head? colors
true
```

Индекс переменной colors в первой позиции индекса блока.

```
print index? colors
1
```

Блок имеет длину в три значения:

```
print length? colors
3
```

Первый элемент в блоке:

```
print first colors
red
```

Второй элемент в блоке:

```
print second colors
green
```

ы можете изменить положение индекса/позиции переменной colors в блоке с помощью различных функций. Чтобы переместить переменную цветов на следующую позицию в блоке цветов, используйте функцию **next**:

```
colors: next colors
```

Функция **next** перемещает индекс вперёд, на одно значение в блоке и возвращает значение на этой позиции, в качестве результата. Переменная colors теперь установлено в этой новой позиции:

Положение переменной colors изменилось. Теперь переменная больше не находится в голове (head) блока:

```
print head? colors
false
```

Он находится на второй позиции в блоке:

```
print index? colors
2
```

Однако, если вы получите первый предмет цветов, вы получите:

```
print first colors
green
```

Положение значения, возвращаемого функцией **first**, зависит от положения, которое цвета имеют в блоке. Возвращаемое значение - это не первый цвет в блоке, а первый цвет, следующий сразу за текущей позицией блока.

Точно так же, если вы спросите длину или второй цвет, вы обнаружите, что они также относительны:

```
print length? Colors
2
print second colors
blue
```

Вы можете перейти к следующей позиции и получить аналогичный набор результатов:

```
colors: next colors

print index? colors
3
print first colors
blue
print length? colors
1
```

Блок-схема теперь выглядит так::

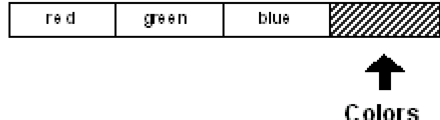
Индекс в переменной `colors` теперь указывает на последний цвет в блоке, но это ещё не в положение хвоста.

```
print tail? colors
false
```

Чтобы дотянуться до хвоста, его нужно переместить в следующую позицию:

```
colors: next colors
```

Теперь индекс переменной `colors` находится в хвосте блока. Он больше не имеет допустимого цвета(значения). Это за концом блока.



Если вы попробуете свой код, вы получите:

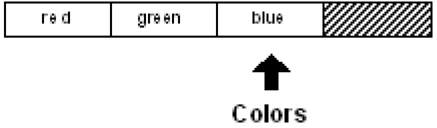
```
print tail? colors
true
print index? colors
4
print length? Colors
0
print first colors
** Ошибка сценария: за пределами допустимого диапазона или за пределом.
** Где: print first colors
```

В последнем случае вы получите сообщение об ошибке, потому что, когда вы пройдёте конец блока, нет действительного первого элемента.

Также возможно перемещение назад в блоке. Если вы напишете:

```
colors: back colors
```

вы переместите переменную `colors` назад на одну позицию в серии:

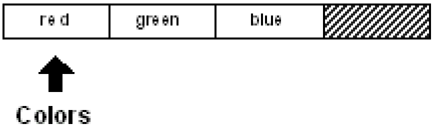


Все тот же код будет работать как раньше:

```
print index? colors
3
print first colors
blue
```

1.2 Пропуск вокруг

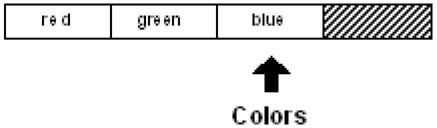
Предыдущие примеры перемещаются по серии по одному элементу за раз. Однако бывают случаи, когда вы хотите пропустить несколько элементов, используя функцию **skip**. Предположим, что переменная `colors` находится в начале ряда:



Вы можете пропустить два пункта, используя:

```
colors: skip colors 2
```

Функция **skip** аналогична функции **next** в том, что **skip** возвращает серию с новой позицией индекса.



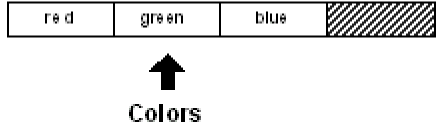
Следующий код подтверждает новую позицию:

```
print index? colors
3
print first colors
blue
```

Чтобы двигаться назад, используйте **skip** с отрицательным значениям:

```
colors: skip colors -1
```

Это похоже на **back**. В приведённом выше примере пропуск -1 возвращает индекс на один элемент назад.




```
print first colors
green
```

Обратите внимание, что вы не можете пропустить хвост или начало серии. Если вы попытаетесь это сделать, пропустите только элементы серии на столько, насколько это возможно. Это не вызовет ошибки.

Если вы пропустите слишком далеко вперёд, команда **skip** вернёт конец серии:

```
colors: skip colors 20

print tail? colors
true
```

Если вы выполните **skip** слишком далеко назад, то **skip** возвращает индекс в начало серии:

```
colors: skip colors -100

print head? colors
true
```

Для того, чтобы перейти непосредственно к главе серии, используйте функцию **head**:

```
colors: head colors

print head? colors
true
print first colors
red
```

Вы можете вернуться к хвосту с помощью функции **tail**:

```
colors: tail colors

print tail? colors
true
```

1.3 Извлечение значений

Некоторые из предыдущих примеров использовали **first**(первое) и **second**(второе) порядковые функции для извлечения определённых значений из ряда. Полный набор порядковых функций:

```
first      (первый)
second  (второй)
third   (третий)
fourth  (четвёртый)
fifth   (пятый)
last    (последний_
```

Порядковые функции предоставляются для удобства и используются для выбора значений из наиболее распространённой позиции в серии.

Вот некоторые примеры:

```
colors: [red green blue gold indigo teal]

print first colors
red
print third colors
blue
print fifth colors
indigo
print last colors
teal
```

Чтобы извлечь из числовой позиции, используйте функцию **pick**:

```
print pick colors 3
blue
print pick colors 5
indigo
```

Сокращённое обозначение для выбора - использовать путь (через слеш):

```
print colors/3
blue
print colors/5
indigo
```

Помните, как показано ранее, извлечение выполняется `относительно` указанной вами переменной ряда (индекса). Если бы индекс переменной `colors` находился бы на другом месте в серии, результаты были бы другими.

Извлечение значения за концом его ряда генерирует ошибку в случае порядковых функций и не возвращает ничего (**none**) в случае функции **pick** или пути выбора:

```
print pick colors 10
none
print colors/10
none
```

1.4 Извлечение подсерии

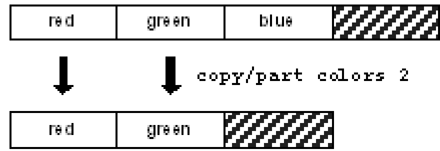
Вы можете извлечь несколько значений из ряда с помощью функции **copy**(копирования). Для этого используйте **copy** с уточнением **/part**, которое указывает количество значений, которые вы хотите извлечь:

```
colors: [red green blue]

sub-colors: copy/part colors 2

probe sub-colors
[red green]
```

Графически это выглядело бы так:

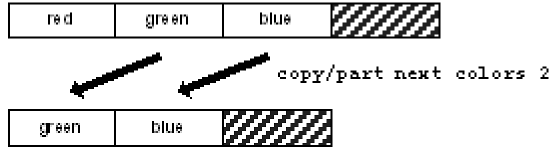


Чтобы скопировать подсерию из любой позиции в серии, сначала перейдите в исходную позицию (установите индекс). В следующем примере перед копированием выполняется переход ко второй позиции в серии с помощью команды **next**:

```
sub-colors: copy/part next colors 2

probe sub-colors
[green blue]
```

Это можно представить как:



Длина копируемой серии может быть указана как конечная позиция, а также как количество копий. Обратите внимание, что позиция указывает, где должна `остановиться` копия, а не конечная позиция.

```
probe copy/part colors next colors
[red]
probe copy/part colors back tail colors
[red green]
probe copy/part next colors back tail colors
[green]
```

Это может быть полезно, когда конечная позиция найдена в результате функции **find**:

```
file: %image.jpg

print copy/part file find file "."
image
```

1.5 Вставка и добавление

Вы можете вставить одно или несколько новых значений в любую часть ряда с помощью функции **insert**. Когда вы вставляете значение в позицию в ряду, пространство создаётся за счёт смещения его предыдущих значений в сторону конца ряда.

Например, блок:

```
colors: [red green]
```

будет отображаться как:

Чтобы вставить новое значение в начало блока, где сейчас расположен индекс переменной colors:

```
insert colors 'blue
```

Слова red и green сдвинулись в право, и слово blue(в начале которого одинарная кавычка, потому что это слово и не должно быть оценено) вставляются во главе списка.

Обратите внимание, что индекс переменной colors остаётся в начале списка.

```
probe colors
[blue red green]
```

Также обратите внимание, что возврат из функции **insert** не использовался, потому что он не был установлен для перемен:

```
colors: insert colors 'blue
```

эффект на блок был бы таким же, но положение индекса переменной colors изменилось бы в результате установки возвращаемого значения. Позиция, возвращаемая при вставке, следует сразу за точкой вставки.

Вставка может быть сделана в любом месте серии. Положение вставки может быть указано, и она может включать хвост. Вставка в хвост имеет эффект добавления к серии.

```
colors: tail colors
insert colors 'gold
probe colors

[blue red green gold]
```

Перед вставкой:

после вставки:

Слово gold было вставлено в конце серии.

Другой способ вставить в конец ряда - использовать функцию **append**. Функция добавления работает так же, как и **insert**, но всегда вставляет в конце. Предыдущий пример будет выглядеть так:

```
append colors 'gold
```

Результат такой же, как и в предыдущем примере.

Функции **insert** и **append** также могут принимать блок аргументов для вставки. Например:

```
colors: [red green]

insert colors [blue yellow orange]

probe colors
[blue yellow orange red green]
```

Если вы хотите вставить новые значения между red и green словами:

```
colors: [red green]

insert next colors [blue yellow orange]

probe colors
[red blue yellow orange green]
```

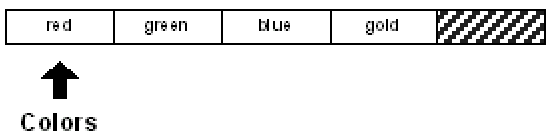
Функции **insert** и **append** имеют другие возможности, которые более подробно рассматриваются в следующем разделе.

1.6 Удаление значений

Вы можете удалить одно или несколько значений из любой части ряда с помощью функции **remove**. Для примера начнём со следующего блока:

```
colors: [red green blue gold]
```

Как показано здесь:



Вы можете удалить первое значение из блока строкой:

```
remove colors
```

Блок станет таким:

Его можно распечатать с помощью:

```
probe colors
[green blue gold]
```

Функция **remove** удаляет значения относительно положения индекса переменной `colors`. Вы можете удалить значения из любого места в серии, установив позицию.

```
remove next colors
```

Блок сейчас:

Несколько значений можно удалить, указав уточнение **/part**.

```
remove/part colors 2
```

Это удаляет оставшиеся значения, оставляя пустой блок:

Как и с **insert/part**, аргумент для **remove/part** также может быть позицией внутри блока.

Удаление всех оставшихся значений - обычная операция. Функция **clear** предназначена для того, чтобы сделать это более простым. **Clear** удаляет все значения от текущей позиции до хвоста. Например:

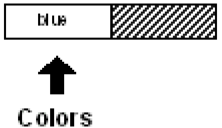
```
Colors: [blue red green gold]
```

Выглядит это так:

Все после синего(blue) можно удалить с помощью:

```
clear next colors
```

Тогда блок станет таким:



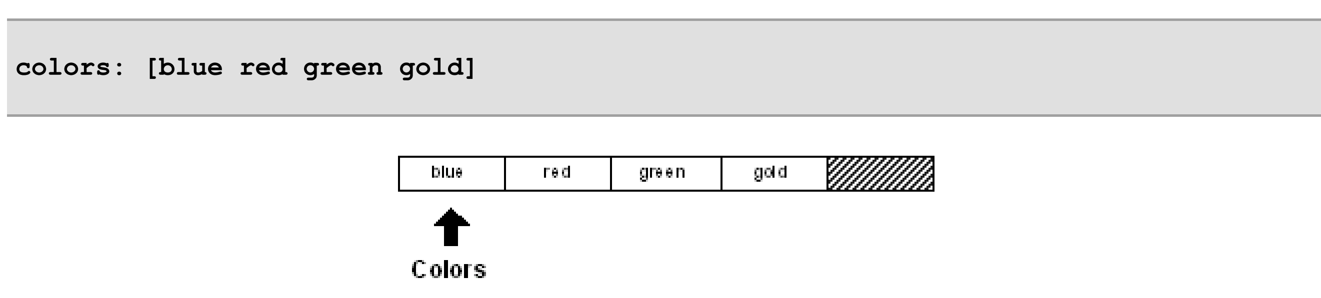
Вы можете легко очистить весь блок с помощью:

```
clear colors
```

1.7 Изменение значений

Предусмотрен один дополнительный набор функций для изменения значений в серии. Функция **change** заменяет одно или несколько значений новыми значениями. Хотя этого можно достичь путём удаления и вставки значений, более эффективно использовать **change**..

Определение блока:



```
colors: [blue red green gold]
```


Его второе значение можно изменить строчкой:

```
change next colors 'yellow
```

И блок станет такой:

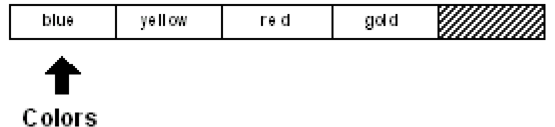
Теперь напечатаем блок:

```
probe colors
[blue yellow green gold]
```

Функция **poke** позволяет указать, что изменение происходит в определённой позиции относительно индекса переменной colors. Функция **pick** аналогична функции **pick**, описанной ранее.

```
poke colors 3 'red
```

Теперь блок:



И выведем его:

```
probe colors
[blue yellow red gold]
```

Функция **change** имеет дополнительные уточнения, которые описаны далее в этой главе.

2. Функции серии

Вот сводка функций, которые работают с сериями. Большинство из них было подробно описано в предыдущем разделе. Другие будут рассмотрены более подробно в этом разделе.

2.1 Функции создания

Функция	Описание
make	Создаёт новую серию данного типа.
copy	Копирует серию.

2.2 Функции навигации

Функция	Описание
next	Возвращает следующую позицию в серии.
back	Возвращает предыдущую позицию в серии.
head	Возвращает позицию заголовка серии.
tail	Возвращает конечную позицию ряда.
skip	Возвращает позицию плюс или минус целое число.
at	Возвращает позицию плюс или минус целое число, но использует ту же индексацию, что и **pick**.

2.3 Информационные функции

Функция	Описание
head?	Возвращает true, если индекс находится в начале серии.
tail?	Возвращает true, если индекс находится в конце серии.
index?	Возвращает смещение от заголовка ряда. (индекс)
length?	Возвращает длину серии от текущей позиции.
offset?	Возвращает расстояние между двумя позициями в серии.
empty?	Возвращает true, если с этой позиции серия пуста.

2.4 Функции извлечения

Функция	Описание
pick	Извлекает одно значение из позиции индекса в серии.
copy/part	Извлекает подсерию из серии.
first	Извлекает первое значение из ряда.
second	Извлекает второе значение из ряда.
third	Извлекает третье значение из ряда.
fourth	Извлекает четвёртое значение из ряда.
fifth	Извлекает пятое значение из ряда.
last	Извлекает последнее значение из ряда.

2.5 Функции модификации

Функция	Описание
insert	Вставляет значения в ряд.
append	Добавляет значения в конец ряда.
remove	Удаляет значения из серии.
clear	Очищает значения до конца ряда.
change	Изменяет значения в серии.
poke	Изменяет значения в позиции в серии.

2.6 Функции поиска

Функция	Описание
find	Находит значение в серии.
select	Находит связанное значение в серии.
replace	Ищет и заменяет значения в серии.
parse	Разбирает значения в серии.

2.7 Функции сортировки

Функция	Описание
sort	Сортирует значения в серии по порядку.
reverse	Изменить порядок значений в серии на обратный

2.8 Функции набора данных (группы)

Функция	Описание
`unique`	Возвращает уникальный набор значений, удаляя дубликаты.
`intersect`	Возвращает только значения, найденные в обеих сериях.
`union`	Возвращает объединённые значения из двух серий.
`exclude`	Возвращает одну серию меньше другой.
`difference`	Возвращает значения, не найденные ни в одной из серий.

3. Типы данных серии

Все типы данных серии можно разделить на два больших класса. Каждый включает значение типа данных и функцию проверки типа.

3.1 Типы блоков

Тип блока	Описание
`Block!`	Блоки значений
`Paren!`	Блоки значений, заключённые в круглые скобки
`Path!`	Пути
`List!`	Связанные списки
`Hash!`	Ассоциативные массивы

3.2 Типы строк

Тип строки	Описание
`String!`	Строка символов
`Binary!`	Строка байт (двоичных данных)
`Tag!`	HTML и XML теги
`File!`	Имена файлов
`URL!`	Адрес интернет ресурсов
`Email!`	Адрес электронной почты
`Image!`	Данные изображения
`Issue!`	Коды последовательности

3.3 Псевдотипы

Серийные типы данных сгруппированы в несколько псевдотипов, что упрощает проверку аргументов функций и типов:

Псевдотип	Описание
`Series!`	Тип данных серии
`Any-block!`	Любой из блочных типов данных
`Any-string!`	Любой из строковых типов данных

3.4 Функции типовых испытаний

Тесты блочного типа:

```
Block? Paren? Path? List? Hash?
```

Тесты строкового типа:

```
String? Binary? Tag? File? URL?

Email? Image? Issue?
```

Типовые испытания других серий:

```
Series? Any-block? Any-string?
```

4. Информация о серии

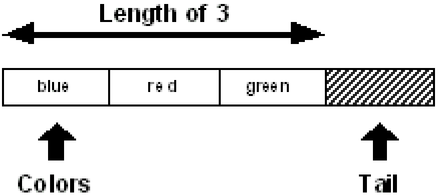
4.1 Length?

Длина серии - это количество элементов (значений для блока или символов для строки) от текущей позиции (индекса) до конца . Если текущая позиция является заголовком серии, то длина - это количество элементов во всей серии.

Функция `length?` возвращает количество элементов до хвоста.

```
colors: [blue red green]
print length? colors
3
```

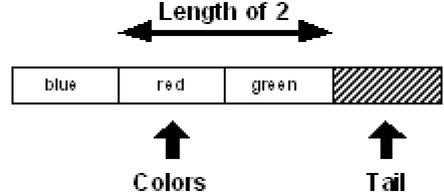
Все три значения являются частью длины:



Если положение индекса переменной `color` продвигается к следующему значению:

```
color: next color
print length? color
2
```

То длинна становится равна двум:



Другие примеры с **length?**:

```
print length? "Ukiah"
5
print length? []
0
print length? ""
0
data: [1 2 3 4 5 6 7 8]
print length? data
8
data: next data
print length? data
7
data: skip data 5
print length? data
2
```

4.2 Head?

Голова ряда - это позиция его первого значения. Если индекс серии в голове, то функция **head?** возвращает `true`:

```
data: [1 2 3 4 5]
print head? data
true
data: next data
print head? data
false
```

4.3 Tail?

Хвостовая часть серии - это позиция, следующая сразу `за последним` значением. Если переменная ряда находится в хвосте, функция **tail?** возвращает `true`:

```
data: [1 2 3 4 5]
print tail? data
false
data: tail data
print tail? data
true
```

Функция **empty?** эквивалентна функции **tail?**.

```
print empty? data
true
```

Если **empty?** возвращает true, это означает, что между текущей позицией и хвостом нет значений; однако в серии все ещё могут быть значения. Значения могут присутствовать перед текущей позицией. Если вам нужно определить, пуста ли серия от головы до хвоста, используйте:

```
print empty? head data
false
```

4.4 Index?

Индекс - положение в серии, относительно головы серии. Чтобы определить позицию индекса для серии, используйте функцию **index?**:

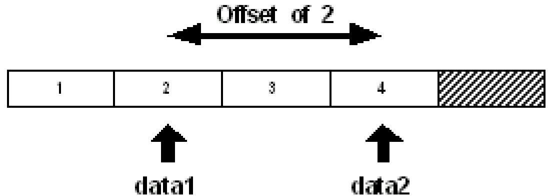
```
data: [1 2 3 4 5]
print index? data
1
data: next data
print index? data
2
data: tail data
print index? data
6
```

4.5 Offset?

Расстояние между двумя позициями в серии можно определить с помощью функции **offset?**.

```
data: [1 2 3 4]
data1: next data
data2: back tail data
print offset? data1 data2
4
```

В этом примере смещение - это разница между позицией 2 и позицией 4:



5. Создание и копирование серий

Новые серии создаются с функциями **make** и **copy**.

Используйте функцию **make** для создания новой серии на основе типа данных серии и начального размера. Размер - это приблизительный размер, необходимый для серии. Если исходный размер слишком мал, серия автоматически расширится, но с небольшой потерей производительности.

```
block: make block! 50
string: make string! 10000
list: make list! 128
file: make file! 64
```

Функция **copy** создаёт новую серию, копируя существующую серию:

```
string: copy "Записка запечатана в бутылке"
new-string: copy string
block: copy [1 2 3 4 5]
new-block: copy block
```

Копирование также важно для использования с функциями, которые изменяют содержимое серии. Например, если вы хотите изменить регистр строки без изменения оригинала, используйте **copy:**

```
string: uppercase copy "Записка запечатана в бутылке"
```

5.1 Частичные копии

Функция **copy** имеет уточнение **/part**, которое принимает один аргумент, являющийся либо целым числом, определяющее количество элементов для копирования, либо позицию в серии, указывающею последнюю позицию для копирования.

```
str: "Записка запечатана в бутылке"
print str
Записка запечатана в бутылке
print copy/part str find str " "
Записка
new-str: copy/part (find str "запечатана") (find str "бутылке")
print new-str
запечатана в
blk: [ages [10 12 32] sizes [100 20 30]]
new-blk: copy/part blk 2
probe new-blk
[ages [10 12 32]]
```

5.2 Глубокие копии

Многие блоки содержат другие блоки и строки. Когда такой блок копируется, его подсерии не копируются. Подсерии упоминаются напрямую и представляют собой те же данные серии, что и исходный блок. Если вы измените любую из этих подсерий, вы также измените их в исходном блоке.

В **copy/deep** уточняющих силах копии всех значений ряда в пределах блока:

```
blk-one: ["abc" [1 2 3]]
probe blk-one
["abc" [1 2 3]]
```

Следующий пример присваивает нормальную копию blk-one к blk-two:

```
blk-two: copy blk-one
probe blk-one
["abc" [1 2 3]]
probe blk-two
["abc" [1 2 3]]
```

Если изменяется строка или блок, содержащиеся в blk-two, значения серии в blk-one также изменяются..

```
append blk-two/1 "DEF"
append blk-two/2 [4 5 6]
probe blk-one
["abcDEF" [1 2 3 4 5 6]]
probe blk-two
["abcDEF" [1 2 3 4 5 6]]
```

Использование **copy/deep** делает копию всех значений серии, найденных в блоке:

```
blk-two: copy/deep blk-one
append blk-two/1 "ghi"
append blk-two/2 [7 8 9]
probe blk-one
["abcDEF" [1 2 3 4 5 6]]
probe blk-two
["abcDEFghi" [1 2 3 4 5 6 7 8 9]]
```

5.3 Начальные копии

При инициализации строки или серии блоков используйте **copy**, чтобы сделать уникальную серию:

```
str: copy ""
blk: copy []
```

Использование **copy** гарантирует, что новый ряд создаётся для слова каждый раз, когда слово инициализируется. Вот пример того, почему это важно.

```
print-it: func [/local str] [
    str: ""
    insert str "ha"
    print str
]

print-it
ha
print-it
haha
print-it
hahaha
```

В этом примере, поскольку **copy** не использовалась, серия пустых строк изменяется при каждом вызове print-it. Строка series ha вставляется в str каждый раз, когда она вызывается print-it. Изучение источника функции в том виде, в каком он сейчас существует, выявляет корень проблемы:

```
source print-it
print-it: func [/local str] [
    str: "hahaha"
    insert str "ha"
    print str
]
```

Хотя str является локальной переменной, её строковое значение является глобальным. Чтобы избежать этой проблемы, функция **copy** должна скопировать пустую строку или использовать **make** в строке.**make**.

```
print-it: func [/local str] [
    str: copy ""
    insert str "ha"
    print str
]

print-it
ha
print-it
ha
print-it
ha
```

6. Итерация серии

Вы можете использовать цикл для обхода серии. Есть несколько функций цикла, которые могут помочь автоматизировать итерационный процесс.

6.1 Foreach

Цикл **foreach** проходит через серию, устанавливая слово или несколько слов в значения в серии.

Цикл **foreach** принимает три аргумента: слово или блок слов, содержащий значения для каждой итерации, серию и блок, оцениваемый для каждой итерации.

```
colors: [red green blue yellow orange gold]
foreach color colors [print color]
red
green
blue
yellow
orange
gold
foreach [c1 c2] colors [print [c1 c2]]
red green
blue yellow
orange gold
foreach [c1 c2 c3] colors [print [c1 c2 c3]]
red green blue
yellow orange gold
```

Это очень полезно для блоков, которые содержат связанные значения:

```
people: [
    "Bob" bob@example.com 12
    "Tom" tom@example.net 40
    "Sam" sam@example.org 22
]
foreach [name email age] people [
    print [name email age]
]
Bob bob@example.com 12
Tom tom@example.net 40
Sam sam@example.org 22
```

Обратите внимание, что цикл **foreach** не продвигает текущий индекс по серии, поэтому нет необходимости сбрасывать его переменную серии.

6.2 While

Наиболее гибкий подход заключается в использовании цикла **while**, который позволяет делать почти все для серии без проблем.

```
colors: [red green blue yellow orange]

while [not tail? colors] [
    print first colors
    colors: next colors
]
red
green
blue
yellow
orange
```

Показанный ниже метод позволяет вставлять значения, не нажимать дважды: The method shown below allows you to **insert** values without hitting a value twice:

```
colors: head colors

while [not tail? colors] [
    if colors/1 = 'yellow [
        colors: insert colors 'blue
    ]
    colors: next colors
]
```

Этот пример показывает, что **insert** возвращает позицию сразу после вставки.

Чтобы удалить (**remove**) значение без случайного пропуска значения, используйте следующий код:

```
colors: head colors

while [not tail? colors] [
    either colors/1 = 'blue [
        remove colors
    ][
        colors: next colors
    ]
]
```

Обратите внимание, что если удаление выполнено, функция **next** не выполняется.

6.3 Цикл Forall

Цикл **forall** похожа на **while** но устраняет некоторые из требуемых усилий. Цикл **forall** начинается с текущего индекса и последовательно продвигается к своему хвосту, оценивая блок для каждого значения.

Forall принимает два аргумента: переменную серии и блок, оцениваемый для каждой итерации.

```
colors: [red green blue yellow orange]

forall colors [print first colors]
red
green
blue
yellow
orange
```

forall перемещает позицию индекса через ряд, поэтому, когда он заканчивается, индекс остаётся на хвосте:

```
print tail? colors
true
```

Следовательно, перед повторным использованием переменную необходимо сбросить:

```
colors: head colors
```

Кроме того, если блок изменяет серию, будьте осторожны, чтобы не пропустить или не повторить значение. В некоторых случаях цикл **forall** может зациклится, поэтому если вы не уверены, используйте **while**.

```
forall colors [
    if colors/1 = 'blue [remove colors]
    print first colors
]
red
green
yellow
orange
```

6.4 Цикл Forskip

Подобно **forall**, цикл **forskip** продвигается по серии, начиная с текущей позиции, но каждый раз пропускает указанное количество значений.

Цикл **forskip** принимает три аргумента: переменную серии, пропуск между каждой итерацией и блок для оценки для каждой итерации.

```
colors: [red green blue yellow orange]

forskip colors 2 [print first colors]
red
blue
orange
```

Цикл **forskip** оставляет серию в хвосте, требуя, чтобы вы её сбросили.

```
print tail? colors
true
colors: head colors
```

6.5 Функция прерывания

Любой из циклов можно остановить в любое время, оценив функцию **break** (прерывания) из блока оценки/цикла. См. Главу «Выражения» для получения дополнительной информации о функции **break**.

7. Поиск в серии

Функция **find** выполняет поиск значения или шаблона в блоках или сериях строк. Эта функция имеет множество усовершенствований, которые позволяют изменять параметры поиска в широком диапазоне.

7.1 Простой поиск

Самый простой и распространённый способ использования **find** - поиск значения в блоке или строке. В этом случае для поиска требуется только два аргумента: серия для поиска и значение для поиска:

```
colors: [red green blue yellow orange]
where: find colors 'blue
probe where
[blue yellow orange]
print first where
blue
```

Функция **find** также может искать значения по типу данных. Это может быть весьма полезно.

```
items: [10:30 20-Feb-2000 Cindy "United"]
where: find items date!
print first where
20-Feb-2000
where: find items string!
print first where
United
```

Пример поиска в строке:

```
colors: "red green blue yellow orange"
where: find colors "blue"
print where
blue yellow orange
```

Если поиск не удался, возвращается **none**.

```
colors: [red green blue yellow orange]
probe find colors 'indigo
none
```

7.2 Уточнения в find

В Find есть множество уточнений, которые поддерживают широкий спектр параметров поиска:

Уточнение	Описание
/part	Ограничивает поиск в серии заданной продолжительностью или конечной позицией.
/only	Обрабатывает значение ряда как одно значение.
/case	Использует сравнение строк с учётом регистра.
/any	Разрешает использование шаблонов подстановочных знаков, позволяющих проводить сопоставления с любым символом. Звёздочка (*) в шаблоне соответствует любой строке, а вопросительный знак (?) В шаблоне соответствует любому символу.
/with	Позволяет использовать подстановочные знаки шаблона с разными символами, кроме звёздочки (*) и (?). Это позволяет шаблону содержать звёздочки и вопросительные знаки.
/match	Соответствует шаблону, начинающемуся с позиции текущего ряда, а не находит первое вхождение значения или строки. Возвращает позицию хвоста, если совпадение найдено.
/tail	Возвращает хвостовую позицию совпадения при успешном поиске, а не точку, в которой было найдено совпадение.
/last	Поиск совпадения в обратном направлении, начиная с конца серии.
/reverse	Поиск совпадения в обратном направлении, начиная с текущей позиции.

7.3 Частичный поиск

Уточнение **/part** позволяет ограничить часть серии, в которой выполняется поиск. Например, вы можете захотеть ограничить поиск заданной строкой или частью текста.

Подобно **insert/part** и **remove/part**, **find/part** принимает либо счётчик, либо конечную позицию. В следующем примере используется счетчик и ограничивается поиск первыми тремя элементами:

```
colors: [red green blue yellow blue orange gold]
probe find/part colors 'blue
[blue yellow blue orange gold]
```

Следующий поиск ограничен первыми 15 символами:

```
text: "Keep things as simple as you can."
print find/part text "as" 15
as simple as you can.
```

В следующем примере используется конечная позиция. Поиск ограничен одной строкой текста:

```
text: {
    This is line one.
    This is line two.
}

start: find text "this"
end: find start newline
item: find/part start "line" end
print item
line one.
```

7.4 Позиция хвоста

Функция **find** возвращает позицию в серии, где был найден элемент. Уточнение **/tail** возвращает позицию сразу после пункта , который был найден. Вот пример:

```
filename: %script.txt

print find filename "."
.txt
print find/tail filename "."
txt
clear change find/tail filename "." "r"
print filename
script.r
```

В этом примере **clear** необходимо удалить xt, который следует за t.

7.5 Обратный поиск

Последний пример в предыдущем разделе завершится ошибкой, если в имени файла будет более одной точки. Например:

```
filename: %new.script.txt
print find filename "."
.script.txt
```

В этом примере нам нужно последнее вхождение точки в строке, которое можно найти с помощью уточнения **/last**. В **/last** уточняющие поиски отсталые через ряд. Уточнение **/last** выполняет поиск в обратном направлении через ряд.

```
print find/last filename "."
.txt
```

Уточнение **/last** может быть объединено с **/tail** к производству:

```
print find/last filename "."
txt
```

Если вы хотите продолжить поиск в обратном направлении по строке, вам потребуется уточнение **/reverse**. Это уточнение выполняет поиск от текущей позиции обратно к голове, а не вперёд к хвосту.

```
where: find/last filename "."
print where
.txt
print find/reverse where "."
.script.txt
```

Обратите внимание, что **/reverse** продолжает поиск непосредственно перед позицией последнего совпадения. Это не позволяет ему снова найти тот же период.

7.6 Повторные поиски

Вы легко можете повторить функцию **find** для поиска нескольких вхождений значения или строки. Вот пример, который напечатает все строки, найденные в блоке:

```
blk: load %script.r
while [blk: find blk string!] [
    print first blk
    blk: next blk
]
```

В следующем примере подсчитывается количество новых строк в скрипте. Он использует уточнение **/tail** для предотвращения бесконечного цикла и возвращает позицию сразу после совпадения.

```
text: read %script.r
count: 0
while [text: find/tail text newline] [count: count + 1]
```

Чтобы выполнить повторный поиск в обратном направлении, используйте уточнение **/reverse**. В следующем примере печатаются все позиции индекса в обратном порядке для текста сценария.

```
while [text: find/reverse tail text newline] [
    print index? text
]
```

7.7 Соответствие

Уточнение **/match** изменяет поведение **find** для выполнения сопоставления с образцом на текущей позиции ряда. Это уточнение позволяет выполнять операции синтаксического анализа, сопоставляя следующую часть серии с ожидаемыми шаблонами. См. Главу о синтаксическом анализе, чтобы узнать о другом способе сопоставления серий.

Вот простой пример сопоставления:

```
blk: [1342 "Franklin Pike Circle"]
probe find/match blk integer!
["Franklin Pike Circle"]
probe find/match blk 1432
["Franklin Pike Circle"]
probe find/match blk "test"
none
str: "Keep things simple."
probe find/match str "keep"
" things simple."
print find/match str "things"
none
```

Обратите внимание, что в этом примере поиск не выполняется. Начало серии либо совпадает, либо нет. Если он совпадает, серия перемещается на позицию, следующую сразу за точкой совпадения, что позволяет сопоставить следующую последовательность.

Вот простой парсер, написанный с помощью **find/match**:

```
grammar: [
    ["keep" "make" "trust"]
    ["things" "life" "ideas"]
    ["simple" "smart" "happy"]
]

parse-it: func [str /local new] [
    foreach words grammar [
        foreach word words [
            if new: find/match str word [break]
        ]
       if none? new [return false]
       str: next new  ;skip space
   ]
   true
]

print parse-it "Keep things simple"
true
print parse-it "Make things smart"
true
print parse-it "Trust life well"
false
```

Сопоставление можно сделать с учётом регистра с помощью уточнения **/case**.

Возможность **/match** может быть значительно расширена добавлением уточнения **/any**, как описано ниже.

7.8 Поиск по шаблону

Уточнение **/any** позволяет подстановочные сопоставление с образцом. Знак вопроса (?) И звёздочка (*) действуют как подстановочные знаки для соответствия любому одиночному символу или любому количеству символов соответственно. Уточнение **/any** может быть использовано в сочетании **find** с или без **/match** утонченности.

Примеры:

```
str: "abcdefg"
print find/any str "c*f"
cdefg
print find/any str "??d"
bcdefg
email-list: [
    mack@rebol.dom
    judy@somesite.dom
    jack@rebol.dom
    biff@rebol.dom
    jenn@somesite.dom
]
foreach email email-list [
    if find/any email *@rebol.dom [print email]
]
mack@rebol.dom jack@rebol.dombiff@rebol.dom
```

В следующем примере используется уточнение **/match**, чтобы попытаться сопоставить шаблон со следующей частью серии:

```
file-list: [
    %rebol.exe
    %notes.html
    %setup.html
    %feedback.r
    %nntp.r
    %rebdoc.r
    %rebol.r
    %user.r
]

foreach file file-list [
    if find/match/any file %reb*.r [print file]
]

rebdoc.r rebol.r
```

Если какой-либо из подстановочных знаков является частью того, что должно быть сопоставлено, заменяющие подстановочные символы могут быть предоставлены с использованием уточнения **/with**.

7.9 Select

Полезной разновидностью функции **find** является функция **select**, которая возвращает значение, следующее за найденным. Функция **select** часто используется для поиска значения в помеченных блоках данных. Функция **select** принимает те же аргументы, что и **find**: серия для поиска и значение **find**. Однако, в отличие от **find** , который возвращает позицию ряда, функция **select** возвращает значение, следующее за совпадением.

```
colors: [red green blue yellow orange]
print select colors 'green
blue
```

Имея простую базу данных, функцию **select** можно использовать для доступа к её значениям:

```
email-book: [
    "George" harrison@guru.org
    "Paul" lefty@bass.edu
    "Ringo" richard@starkey.dom
    "Robert" service@yukon.dom
]
```

Следующий код находит адрес электронной почты соответствующий конкретному имени:

```
print select email-book "Paul"
lefty@bass.edu
```

Используйте функцию **select**, чтобы найти блок выражений для оценки. Например, учитывая следующие данные:

```
cases: [
    10 [print "ten"]
    20 [print "twenty"]
    30 [print "thirty"]
]
```

блок можно оценить на основе селектора:

```
do select cases 10
ten
do select cases 30
thirty
```

7.10 Поиск и замена

Чтобы заменить значения в серии, вы можете использовать функцию **replace**. Эта функция ищет определённое значение в серии, а затем заменяет его новым значением.

Функция **replace** принимает три аргумента: серию для поиска, значение для замены и новое значение.

```
str: "hello world hello"
probe replace str "hello" "aloha"
"aloha world hello"
data: [1 2 8 4 5]
probe replace data 8 3
[1 2 3 4 5]
probe replace data 4 `four
[1 2 3 four 5]
probe replace data integer! 0
[0 2 3 four 5]
```

Используйте уточнение **/all**, чтобы заменить все вхождения значения от текущей позиции до хвоста.

```
probe replace/all data integer! 0
[0 0 0 four 0]
code: [print "hello" print "world"]
replace/all code 'print 'probe
probe code
[probe "hello" probe "world"]
do code
helloworld
str: "hello world hello"
probe replace/all str "hello" "aloha"
"aloha world aloha"
```

8. Сортировка серий

Функция **sort** предлагает простой и быстрый метод сортировки серий. Это наиболее полезно для блоков данных, но также может использоваться для строк символов.

8.1 Простая сортировка

Простые примеры сортировки:

```
names: [Ева Люк Зафод Адам Мэтт Бетти]
probe sort names
[Адам Бетти Ева Зафод Люк Мэтт]
print sort [321.3 78 321 42 321.8 12 98]
12 42 78 98 321 321.3 321.8
print sort "plosabelm"
abellmops
```

Обратите внимание на то, что **sort** разрушительна для своей серии аргументов. Он переупорядочивает исходные данные. Чтобы предотвратить это, используйте **copy**, как в следующем примере:

```
probe sort copy names
```

По умолчанию при сортировке регистр не учитывается:

```
print sort ["Fred" "fred" "FRED"]
Fred fred FRED
print sort "G4C28f9I15Ed3bA076h"
0123456789AbCdEfGhI
```

Уточнение **/case** делает сортировку с учётом регистра:

```
print sort/case "gCcAHfiEGeBIdbFaDh"
ABCDEFGHIabcdefghi
print sort/case ["Fred" "fred" "FRED"]
FRED Fred fred
print sort/case "g4Dc2BI8fCF9i15eAd3bGaE07H6h"
0123456789ABCDEFGHIabcdefghi
```

Можно отсортировать многие другие типы данных:

```
print sort [1.3.3.4 1.2.3.5 2.2.3.4 1.2.3.4]
1.2.3.4 1.2.3.5 1.3.3.4 2.2.3.4
print sort [$4.23 $23.45 $62.03 $23.23 $4.22]
$4.22 $4.23 $23.23 $23.45 $62.03
print sort [11:11:43 4:12:53 4:14:53 11:11:42]
4:12:53 4:14:53 11:11:42 11:11:43
print sort [11-11-1999 10-11-9999 11-4-1999 11-11-1998]
11-Nov-1998 11-Apr-1999 11-Nov-1999 10-Nov-9999
print sort [john@doe.dom jane@doe.dom jack@jill.dom]
jack@jill.dom jane@doe.dom john@doe.dom
print sort [%user.r %rebol.r %history.r %notes.html]
history.r notes.html rebol.r user.r
```

8.2 Групповая сортировка

Часто бывает необходимо отсортировать набор данных, который имеет более одного значения на запись. Уточнении **/skip** применяется для сортировки таких записей, которые имеют фиксированную длину. Уточнение принимает один дополнительный аргумент: целое число, определяющее длину каждой записи.

Вот пример сортировки блока, который содержит имя, фамилию, возраст и адрес электронной почты. Блок сортируется по его первому столбцу, имени.

```
names: [
    "Evie" "Jordan" 43 eve@jordan.dom
    "Matt" "Harrison" 87 matt@harrison.dom
    "Luke" "Skywader" 32 luke@skywader.dom
    "Beth" "Landwalker" 104 beth@landwalker.dom
    "Adam" "Beachcomber" 29 adam@bc.dom
]
sort/skip names 4
foreach [first-name last-name age email] names [
    print [first-name last-name age email]
]
Adam Beachcomber 29 adam@bc.dom
Beth Landwalker 104 beth@landwalker.dom
Evie Jordan 43 eve@jordan.dom
Luke Skywader 32 luke@skywader.dom
Matt Harrison 87 matt@harrison.dom
```

8.3 Функции сравнения

Уточнение **/compare** позволяет выполнять пользовательские сравнения для сортировки данных. Это уточнение принимает дополнительный аргумент - функцию сравнения, используемую для упорядочивания данных.

Функция сравнения записывается как обычная функция, которая принимает два аргумента. Эти аргументы являются значениями для сравнения. Функция сравнения возвращает `true` (истина), если первое значение следует поместить перед вторым значением, и `false` (ложь), если первое значение следует поместить после второго значения.

При обычном сравнении данные располагаются в порядке возрастания:

```
ascend: func [a b] [a < b]
```

Если первое значение меньше второго, функция возвращает `true` (истину), и первое значение помещается перед вторым значением.

```
data: [100 101 -20 37 42 -4]
probe sort/compare data :ascend
[-20 -4 37 42 100 101]
```

Так же:

```
descend: func [a b] [a > b]
```

Если первое значение больше второго, возвращается `true`, и данные сначала сортируются с более высокими значениями. Сортировка будет происходить от больших значений.

```
probe sort/compare data :descend
[101 100 42 37 -4 -20]
```

Обратите внимание, что в обоих случаях функция сравнения передаётся с указанием имени с двоеточием. Имя, которому предшествует двоеточие, приводит к тому, что функция передаётся на сортировку без предварительной оценки. Функция сравнения также может быть предоставлена напрямую:

```
probe sort/compare data func [a b] [a > b]
[101 100 42 37 -4 -20]
```

9. Серии как наборы данных

Есть несколько функций, которые работают с сериями как с наборами данных. Эти функции позволяют выполнять такие операции, как поиск объединения или пересечения двух серий.

9.1 Unique (Уникальный)

Функция **`unique`** возвращает уникальный набор, который не содержит повторяющихся значений.

Примеры:

```
data: [Bill Betty Bob Benny Bart Bob Bill Bob]
probe unique data
[Bill Betty Bob Benny Bart]
print unique "abracadabra"
abrcd
```

9.2 Intersect (Пересечение)

Функция **`intersect`** принимает две серии и возвращает серию, содержащую значения, присутствующие в обеих сериях.

Примеры:

```
probe intersect [Bill Bob Bart] [Bob Ted Fred]
[Bob]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe intersect lunch dinner
[ham carrot]
print intersect [1 3 2 4] [3 5 4 6]
3 4
string1: "CBAD"    ; A B C D scrambled
string2: "EDCF"    ; C D E F scrambled
print sort intersect string1 string2
CD
```

Пересечение можно найти между битовыми наборами:

```
all-chars: "ABCDEFGHI"
charset1: charset "ABCDEF"
charset2: charset "DEFGHI"
charset3: intersect charset1 charset2

print find charset3 "E"
true
print find charset3 "B"
false
```

Уточнение `/case` выполняет регистрозависимое пересечение:

```
probe intersect/case [Bill bill Bob bob] [Bart bill Bob]
[bill Bob]
```

9.3 Union (Объединение)

Функция `union` принимает две серии и возвращает серию, которая содержит все значения из обеих серий, но не дублирует.

Примеры:

```
probe union [Bill Bob Bart] [Bob Ted Fred]
[Bill Bob Bart Ted Fred]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe union lunch dinner
[ham cheese bread carrot salad rice]
print union [1 3 2 4] [3 5 4 6]
1 3 2 4 5 6
string1: "CBDA"    ; A B C D scrambled
string2: "EDCF"    ; C D E F scrambled
print sort union string1 string2
ABCDEF
```

Функция `union` также может быть использована с битовыми наборами:

```
charset1: charset "ABCDEF"
charset2: charset "DEFGHI"
charset3: union charset1 charset2

print find charset3 "C"
true
print find charset3 "G"
true
```

Уточнение `/case` позволяет регистрозависимые союзы:

```
probe union/case [Bill bill Bob bob] [bill Bob]
[Bill bill Bob bob]
```

9.4 Exclude (Исключить)

Функция **exclude** принимает две серии и возвращает серию, которая содержит все значения первой серии за вычетом значений второй.

```
probe exclude [1 2 3 4] [1 2 3 5]
[4]
probe exclude [Bill Bob Bart] [Bob Ted Fred]
[Bill Bart]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe exclude lunch dinner
[cheese bread]
string1: "CBAD"    ; A B C D scrambled
string2: "EDCF"    ; C D E F scrambled
print sort difference string1 string2
AB
```

Уточнение **/case** позволяет выполнить регистрозависимое исключение:

```
probe exclude/case [Bill bill Bob bob] [Bart bart bill Bob]
[Bill bob]
```

9.5 Difference (Разница)

Функция **difference** принимает две серии и возвращает серию, которая содержит все значения, не совпадающие с обеими сериями.

Примеры:

```
probe difference [1 2 3 4] [1 2 3 5]
[4 5]
probe difference [Bill Bob Bart] [Bob Ted Fred]
[Bill Bart Ted Fred]

lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe difference lunch dinner
[cheese bread salad rice]
string1: "CBAD"    ; A B C D scrambled
string2: "EDCF"    ; C D E F scrambled
print sort difference string1 string2
ABEF
```

Уточнение **/case** позволяет регистрозависимые различия.

```
probe difference/case [Bill bill Bob bob] [Bart bart bill Bob]
[Bill bob Bart bart]
```

9.6 Exclude (Исключить)

Разновидностью функции **difference** является функция **exclude**. Она возвращает значения, которые находятся в первой серии, но не найдены во второй серии. Например:

```
probe exclude [1 2 3 4] [1 2 3 5]
[4]
```

Обратите внимание, что приведённый выше результат не содержит 5, как это было в случае с **difference** (разница) в предыдущем разделе.

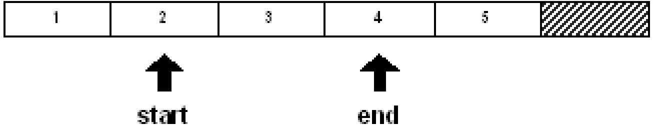
```
probe exclude [Bill Bob Bart] [Bob Ted Fred]
[Bill Bart]
probe exclude "abcde" "ace"
"bd"
```

10. Переменные нескольких серий

Несколько переменных могут относиться к одной и той же серии. Например:

```
data: [1 2 3 4 5]
start: find data 3
end: find start 4
print first start
2
print first end
4
```

И переменная start, и переменная end относятся к серии. У них разные позиции, но серия, на которую они ссылаются, одна.



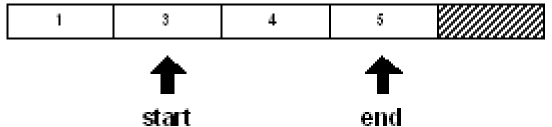
Если для серии выполняется функция **insert** (вставить) или **remove** (удалить), то значения в серии сдвигаются, и переменные start и end могут больше не относиться к одним и тем же значениям. Например, если значение удаляется из серии в начальной позиции:

```
remove start

print first start
3

print first end
5
```

Ряд сдвинулся влево, и теперь переменные относятся к разным значениям.



Обратите внимание, что позиции индекса переменных не изменились, но значения в серии изменились. Такая же ситуация может возникнуть при использовании **insert** (вставки).

Иногда этот побочный эффект будет работать вам на пользу. Иногда этого не происходит, и вам нужно будет исправить это в своем коде.

11. Уточнения модификации

Функции **change** (изменение), **insert** (вставка) и **remove** (удаление) могут потребовать дополнительных уточнений, чтобы откорректировать их работу.

11.1 Part

Уточнение **/part** принимает подсчёт или позицию в серии и использует её, чтобы ограничить эффект функции.

Например, используя следующую серию:

```
str: "abcdef"
blk: [1 2 3 4 5 6]
```

вы можете изменить часть `str` и `blk`, используя **change/part**:

```
change/part str [1 2 3 4] 3
probe str

1234def

change/part blk "abcd" 3
probe blk

["abcd" 4 5 6]
```

Вы можете вставить часть серии в хвост `str` и `blk` с помощью **insert/part**.

```
insert/part tail str "-ghijkl" 4
probe str
1234def-ghi
insert/part tail blk ["--" 7 8 9 10 11 12] 4
probe blk
["abcd" 4 5 6 "--" 7 8 9]
```

Чтобы удалить часть серий `str` и `blk`, используйте **remove/part**. Обратите внимание, как **find** используется для получения позиции в серии:

```
remove/part (find str "d") (find str "-")
probe str
1234-ghi
remove/part (find blk 4) (find blk "--")
probe blk
["abcd" "--" 7 8 9]
```

11.2 Only

Уточнение **/only** позволяет изменять или вставлять блок в виде блока, а не его отдельных значений. Примеры:

```
blk: [1 2 3 4 5 6]
```

Вы можете заменить 2 в blk блоком [a b c] и вставить блок [$1 $2 $3] в позицию 5.

```
change/only (find blk 2) [a b c]
probe blk
[1 [a b c] 3 4 5 6]
insert/only (find blk 5) [$1 $2 $3]
probe blk
[1 [a b c] 3 4 [$1.00 $2.00 $3.00] 5 6]
```

11.3 Dup

Уточнение **/dup** вставляет значение заданное число раз.

Примеры:

```
str: "abcdefghi"
blk: [1 2 3 4 5 6]
```

Вы можете изменить первые четыре значения в строке или серии блоков на звездочку (*) с помощью:

```
change/dup str "*" 4
probe str
****efghi
change/dup blk "*" 4
probe blk
["*" "*" "*" "*" 5 6]
```

Чтобы вставить дефис (-) четыре раза перед последним значением в строке или блоке:

```
insert/dup (back tail str) #"-" 4
probe str
****efgh----i
insert/dup (back tail blk) #"-" 4
probe blk
["*" "*" "*" "*" 5 #"-" #"-" #"-" #"-" 6]
```

Глава 7 - Серия блоков

Содержание:

1. Блоки блоков
2. Пути для вложенных блоков
3. Массивы
   3.1 Создание массивов
   3.2 Начальные значения
4. Составление блоков

1. Блоки блоков

Когда блок отображается как значение в другом блоке, он считается одним значением независимо от того, сколько значений он содержит. Например:

```
values: [
    "new" [1 2]
    %file1.txt ["one" ["two" %file2.txt]]
]
probe values
["new" [1 2] %file1.txt ["one" ["two" %file2.txt]]]
```

Функция **length?** возвращает четыре `значения`. Второе и четвертое значения считаются отдельными значениями:

```
print length? values
4
```

Значения блоков в пределах `values` блока могут быть использованы в качестве `блока`, а также. В следующих примерах **second** используется для извлечения второго значения из `values`.

Чтобы распечатать блок, введите:

```
probe second values
[1 2]
```

Чтобы узнать длину блока, введите

```
print length? second values
2
```

Чтобы распечатать тип данных блока, введите:

```
print type? second values
block
```

Таким же образом операции с рядами могут выполняться с другими типами значений ряда в блоках. В следующих примерах **pick** используется для извлечения %file1.txt из values.

Чтобы посмотреть значение, введите:

```
probe pick values 3
%file1.txt
```

Чтобы посмотреть значение, введите:

```
print length? pick values 3
9
```

чтобы увидеть тип данных значения:

```
print type? pick values 3
file
```

2. Пути для вложенных блоков

Обозначение пути полезно для вложенных блоков.

Четвертое значение в values - это блок, содержащий другой блок. В следующих примерах используется путь для получения информации об этом значении.

Чтобы просмотреть вложенные значения, введите:

```
probe values/4
["one" ["two" %file2.txt]]

probe values/4/2
["two" %file2.txt]
```

Чтобы получить длину вложенных значений, введите:

```
print length? values/4
2

print length? values/4/2
2
```

Чтобы узнать, какой тип данных вложенного значения, введите:

```
print type? values/4
block
print type? values/4/2
block
```

Также можно получить доступ к двум значениям серий в блоке четвертого значения.

Чтобы посмотреть значения, введите:

```
probe values/4/2/1
two
probe values/4/2/2
%file2.txt
```

Чтобы получить длины значений:

```
print length? values/4/2/1
3
print length? values/4/2/2
9
```

Чтобы узнать, к какому типу данных относятся значения:

```
print type? values/4/2/1
string

print type? values/4/2/2
file
```

Чтобы изменить значения:

```
change (next values/4/2/1) "o"
probe values/4/2/1
too

change/part (next find values/4/2/2 ".") "r" 3
probe values/4/2/2
%file2.r
```

Приведенные выше примеры иллюстрируют способность REBOL работать со значениями, вложенными внутри блоков. Обратите внимание, что в последней серии примеров **`change`** используется для изменения строки и файловой серии на трех уровнях `values`. Распечатка блока значений дает:

```
probe values
["new" [1 2] %file1.txt ["one" ["too" %file2.r]]]
```

3. Массивы

Блоки используются для массивов.
Пример статически определенного двумерного массива:

```
arr: [
    [1   2   3  ]
    [a   b   c  ]
    [$10 $20 $30]
]
```

Вы можете получить значения массива с помощью функций извлечения ряда:

```
probe first arr
[1 2 3]

probe pick arr 3
[$10.00 $20.00 $30.00]

probe first first arr
1
```

Вы также можете использовать пути для получения значений из массива:

```
probe arr/1
[1 2 3]

probe arr/3
[$10.00 $20.00 $30.00]

probe arr/3/2
$20.00
```

Пути также можно использовать для изменения значений в массиве:

```
arr/1/2: 20
```

probe arr/1 == [1 20 3]

```
arr/3/2: arr/3/1 + arr/3/3
```

probe arr/3/2 == $40.00

3.1 Создание массивов

Функция **array** создает массивы динамически. Функция принимает аргумент, который является целым числом или блоком целых чисел, и возвращает блок, который является массивом. По умолчанию элементы массива инициализируются как `none`. Чтобы инициализировать ячейки массива каким-либо другим значением, используйте уточнение **/initial**, описанное в следующем разделе.

Когда в массив передается одно целое число, возвращается одномерный массив такого размера:

```
arr: array 5
probe arr
[none none none none none]
```

Когда предоставляется блок целых чисел, массив имеет несколько измерений. Каждое целое число обеспечивает размер соответствующего измерения.

Вот пример двумерного массива с шестью ячейками, двумя строками по три столбца:

```
arr: array [2 3]
probe arr
[[none none none] [none none none]]
```

Его можно превратить в трехмерный массив, добавив в блок еще одно целое число:

```
arr: array [2 3 2]
foreach lst arr [probe lst]
[[none none] [none none] [none none]]
[[none none] [none none] [none none]]
```

Блок целых чисел, передаваемый в **array**, может быть настолько большим, насколько поддерживает ваша память.

3.2 Начальные значения

Чтобы инициализировать ячейки массива значением, отличным от none , используйте уточнение **/initial**. Это уточнение принимает один аргумент: начальное значение. Вот некоторые примеры:

```
arr: array/initial 5 0
probe arr
[0 0 0 0 0]
arr: array/initial [2 3] 0
probe arr
[[0 0 0] [0 0 0]]
arr: array/initial 3 "a"
probe arr
["a" "a" "a"]
arr: array/initial [3 2] 'word
probe arr
[[word word] [word word] [word word]]
arr: array/initial [3 2 1] 11:11
probe arr
[[[11:11] [11:11]] [[11:11] [11:11]] [[11:11] [11:11]]]
```

4. Составные блоки

Функция **compose** удобна для создания блоков из динамических значений. Его можно использовать для создания как данных, так и кода.

Функция **compose** принимает блок в качестве аргумента и возвращает блок, который имеет каждое значение в блоке аргументов. Значения в скобках оцениваются перед возвратом блока. Например:

```
probe compose [1 2 (3 + 4)]
[1 2 7]
probe compose ["The time is" (now/time)]
["The time is" 10:32:45]
```

Если значения в круглых скобках возвращают блок, используются отдельные значения этого блока:

```
probe compose [a b ([c d])]
[a b c d]
```

Чтобы этого не произошло, нужно заключить результат в дополнительный блок:

```
probe compose [a b ([[c d]])]
[a b [c d]]
```

Пустой блок ничего не вставляет:

```
probe compose [a b ([]) c d]
[a b c d]
```

Когда compose дается блок, содержащий субблоки, субблоки не оцениваются, даже если они содержат круглые скобки:

```
probe compose [a b [c (d e)]]
[a b [c (d e)]]
```

Если вы хотите, чтобы субблоки оценивались, используйте уточнение **/deep**. Оно рафинирование вызывает все круглые скобки должны быть оценены, независимо от того, где они находятся:

```
probe compose/deep [a b [c (d e)]]
[a b [c d e]]
```

Глава 8 - Серия строк

Содержание:

1. Строковые функции
2. Преобразование значений в строки
  2.1 Join
  2.2 Rejoin
  2.3 Form
  2.4 Reform
  2.5 Mold
  2.6 Remold
  2.7 String Spacing Functions
  2.8 Uppercase and Lowercase
  2.9 Checksum
  2.10 Compression and Decompression
  2.11 Number Base Conversion
  2.12 Internet Hexadecimal Decoding

1. Строковые функции

Существует множество функций, которые работают со строками или создают их. Доступны функции для изменения строк, поиска строк, сжатия и распаковки строк, изменения интервала между строками, синтаксического анализа строк и преобразования строк. Эти функции работают со всеми типами данных, связанными со строками, такими как `string!`, `binary!`, `tag!`, `file!`, `URL!`, `email!`, и `issue!`.

Функции создания, изменения и поиска строк описаны в главе «Серии». Они включают элементы, перечисленные в строковых функциях.

copy	скопировать всю строку или ее часть
make	выделить память для строки
insert	вставить символ или подстроку в другую строку
remove	удалить один или несколько символов из строки
change	изменить один или несколько символов в строке
append	вставить символ или подстроку в конец строки
find	найти или сопоставить символ или строку в другой строке
replace	найти строку и заменить ее другой строкой

Кроме того, были охвачены такие функции перемещения, как `next`, `back`, `head`, и `tail`. Они используются для изменения положения струн. Кроме того, функции последовательного тестирования позволяют определить вашу позицию в строке.

В этой главе будут представлены функции, которые преобразуют значения REBOL в строки. Эти функции используются часто, а также функциями `print` и `probe`. Они включают:

form	преобразовывать значения с пробелами и в удобочитаемом формате
mold	конвертировать значения в читаемый формат REBOL
join	преобразовывать значения без пробелов
reform	уменьшает значения перед их формированием
remold	уменьшает значения перед их формованием
rejoin	уменьшает значения перед присоединением к ним

В этой главе также будут описаны следующие строковые функции:

detab	заменить табуляцию пробелами
entab	заменить пробелы табуляцией
trim	удалить пробелы или линии вокруг строк
uppercase	преобразовать строку в верхний регистр
lowercase	преобразовать строку в нижний регистр
checksum	вычислить контрольную сумму для строки
compress	запоковать/сжать строку
decompress	рапаковать строку
enbase	преобразовать строку в базовое значение
debase	спреобразовать базовое значение в строку
dehex	преобразовать шестнадцатеричные значения ASCII в символы

2. Преобразование значений в строки

2.1 Join

Функция `join` fпринимает два аргумента и объединяет их в одну серию.

Тип данных возвращаемой серии основан на значении первого аргумента. Когда первым аргументом является значение серии, возвращается этот тип серии.

```
str: "abc"
file: %file
url: http://www.rebol.com/

probe join str [1 2 3]
abc123
probe join file ".txt"
%file.txt
probe join url %index.html
http://www.rebol.com/index.html
```

Когда первый аргумент не является рядом, `join` сначала преобразует его в строку, а затем выполняет добавление:

```
print join $11 " долларов"
$11.00 долларов
print join 9:11:01 " elapsed"
9:11:01 elapsed
print join now/date " -- сегодня"
30-Jun-2000 -- сегодня
print join 255.255.255.0 " netmask"
255.255.255.0 netmask
print join 412.452 " световых лет от нас"
412.452 световых лет от нас
```

Когда вторым аргументом для `join` является блок, значения этого блока оцениваются и добавляются к возвращаемой серии.

```
print join "a" ["b" "c" 1 2]
abc12

print join %/ [%dir1/ %sub-dir/ %filename ".txt"]
%/dir1/sub-dir/filename.txt
```

```
print join 11:09:11 ["AM" " on " now/date]
11:09:11AM on 30-Jun-2000

print join 312.423 [123 987 234]
312.423123987234
```

2.2 Rejoin

Функция **rejoin** идентична **join**, за исключением того, что принимает один аргумент, блок.

```
print rejoin ["try" 1 2 3]
try123
print rejoin ["h" 'e #"l" (to-char 108) "o"]
hello
```

2.3 Form

Функция **form** преобразует значение в строку:

```
print form $1.50
$1.50
print type? $1.50
money
print type? form $1.50
string
```

В следующем примере **form** используется для поиска числа по его десятичному значению:

```
blk: [11.22 44.11 11.33 11.11]
foreach num blk [if find form num ".11" [print num]]

44.11
11.11
```

Когда **form** используется в блоке, все значения в блоке преобразуются в строковые значения с пробелами между каждым значением:

```
print form [11.22 44.11 11.33]
11.22 44.11 11.33
```

Функция **form** не оценивает значения блока. Это приводит к преобразованию слов в строковые значения:

```
print form [a block of undefined words]
a block of undefined words
print form [33.44 num "-- unevaluated string:" str]
33.44 num -- unevaluated string: str
```

2.4 Reform

Функция `reform` похожа на `form`, за исключением того, что блоки уменьшаются перед преобразованием.

```
str1: "Сегодняшняя дата:"
str2: "Текущее время:"
print reform [str1 now/date newline str2 now/time]
Сегодняшняя дата: 30-Jun-2000 Текущее время: 14:41:44
```

Функция `print` основана на функции `reform`.

2.5 Mold

Функция `mold` преобразует значение в строку, которую может использовать REBOL. Строки, созданные с помощью `mold`, можно преобразовать обратно в значения с помощью функции `load`.

```
blk: [[11 * 4] ($15 - $3.89) "eleven dollars"]
probe blk
[[11 * 4] ($15.00 - $3.89) "eleven dollars"]
molded-blk: mold blk
probe molded-blk
{[[11 * 4] ($15.00 - $3.89) "eleven dollars"]}
print type? blk
block
print type? molded-blk
string
probe first blk
[11 * 4]
<A name=pgfId-539552>probe first molded-blk
#"["
```

Строки, возвращенные из `mold`, могут быть загружены REBOL:

```
new-blk: load molded-blk
probe new-blk
[[11 * 4] ($15.00 - $3.89) "eleven dollars"]

print type? new-blk
block

probe first new-blk
[11 * 4]
```

Функция `mold` не оценивает значения блока.

```
money: $11.11
sub-blk: [inside another block mold this is unevaluated]
probe mold [$22.22 money "-- unevaluated block:" sub-blk]
{[$22.22 money "-- unevaluated block:" sub-blk]}

probe mold [a block of undefined words]
[a block of undefined words]
```

2.6 Remold

Функция **remold** работает так же, как и **mold**, за исключением того, что блоки уменьшаются до преобразования.

```
str1: "Сегодняшняя дата::"
probe remold [str1 now/date]
{["Сегодняшняя дата:" 30-Jun-2000]}
```

2.7 Функции разделения строк

2.7.1 Trim

Функция **trim** удаляет лишние пробелы из строки.

Функция **trim** по умолчанию заключается в удалении лишних пробелов в начале и конце строки:

```
str: "  строка текста с пробелами в начале и конце    "
print trim str
строка текста с пробелами в начале и конце
```

Обратите внимание, что строка изменяется в процессе:

```
print str
строка текста с пробелами в начале и конце
```

Чтобы обрезать копию строки, напишите:

```
print trim copy str
Чтобы обрезать копию строки, напишите:
```

Trim включает в себя ряд уточнений, чтобы указать, где нужно удалить пробел в строке:

/head	удаляет пробел в начала строки
/tail	удаляет пробел в конце строки
/auto	удаляет пробел из каждой строки относительно первой строки
/lines	удаляет символы новой строки, заменяя их пробелами
/all	- удаляет все пробелы
/with	удаляет все указанные символы

Используйте уточнения **/head** и **/tail** для обрезки с любого конца строки:

```
probe trim/head copy str
      строка текста с пробелами в начале и конце

probe trim/tail copy str
строка текста с пробелами в начале и конце
```

Используйте уточнение **/auto**, чтобы вырезать ведущие пробелы из нескольких строк, не трогая отступы:

```
str: {
    indent text
        indent text
            indent text
        indent text
    indent text
}

print str
indent text
    indent text
        indent text
    indent text
indent text

probe trim/auto copy str
{indent text
    indent text
        indent text
    indent text
indent text
}
```

Используйте **/lines** для обрезки начала и конца строк, а также преобразования новой строки в пробелы:

```
probe trim/lines copy str
{indent text indent text indent text indent text indent text}
```

Используйте **/all**, чтобы удалить все пробелы:

```
probe trim/all copy str
indenttextindenttextindenttextindenttextindenttext
```

Уточнением **/with** удалит все символы, которые вы укажете. В следующем примере удалены пробелы, разрывы строк и символы `e` и `t`:

```
probe trim/with copy str " ^/et"
indnxindnxindnxindnxindnx
```

2.7.2 Detab и Entab

Функции **detab** и **entab** преобразует символы табуляции в пробелы и пробелы в закладках.

```
str:
{^(tab)line one
^(tab)^(tab)line two
^(tab)^(tab)^(tab)line three
^(tab)line^(tab)full^(tab)of^(tab)tabs}
```

```
print str
line one
        line two
            line three
    line    full    of  tabs
```

По умолчанию функция **detab** преобразует табуляторы в четыре пробела (стандартный интервал REBOL). Все табуляции в строке будут преобразованы в пробелы, независимо от того, где они расположены.

```
probe detab str
{    line one
        line two
            line three
    line    full    of  tabs}
```

Обратите внимание, что функции **detab** и **entab** влияют на строку, указанную в качестве аргумента. Чтобы изменить копию исходной строки, используйте функцию копирования.

Функция **entab** преобразует пробелы в табуляции. Каждые четыре пробела будут преобразованы в одну табуляцию. Только пробелы в начале строки будут преобразованы в табуляцию.

```
probe entab str
{^-line one
^-^-line two
^-^-^-line three
^-line^-full^-of^-tabs}
```

Вы можете использовать уточнение **/size**, чтобы указать количество пробелов в табуляции. Например, если вы хотите преобразовать каждую табуляцию в восемь пробелов или преобразовать каждые восемь пробелов в табуляцию, вы можете использовать этот пример:

```
probe detab/size str 8
{        line one
                line two
                        line three
        line    full    of      tabs}
probe entab/size str 8
{^-line one
^-^-line two
^-^-^-line three
^-line^-full^-of^-tabs}
```

2.8 Uppercase и Lowercase

Есть две функции для изменения регистра символов: **uppercase** и **lowercase**. Функция **uppercase** принимает строковый аргумент и преобразует его символы в верхний регистр:

```
print uppercase "SamPle TExT, tO test CASES"
SAMPLE TEXT, TO TEST CASES
```

Функция lowercase преобразует символы в нижний регистр:

```
print lowercase "Sample TEXT, tO teST Cases"
sample text, to test cases
```

Чтобы преобразовать только часть строки, используйте уточнение /part:

```
print upppercase/part "ukiah" 1
Ukiah
```

2.9 Checksum

Функция checksum возвращает контрольную сумму строкового значения. Существует три типа контрольных сумм, которые можно вычислить:

CRC	24-битная контрольная сумма с круговой избыточностью
TCP	стандартная 16-битная контрольная сумма Интернета
Secure	криптографически безопасная контрольная сумма

По умолчанию вычисляется контрольная сумма CRC:

```
print checksum "hello"
52719
print checksum (read http://www.rebol.com/)
356358
```

Чтобы вычислить 16-битную контрольную сумму TCP, используйте уточнение /tcp refinement:

```
print checksum/tcp "hello"
10943
```

Безопасная контрольная сумма вернет двоичное значение, а не целое число. Используйте уточнение /secure для вычисления безопасной контрольной суммы:

```
print checksum/secure "hello"
#{AAF4C61DDCC5E8A2DABEDE0F3B482CD9AEA9434D}
```

2.10 Compression и Decompression

Функция compress сжимает строку и возвращает двоичный тип данных. В следующем примере показывается разница в размере сжатого и исходного текста, помещённого в переменную str:

```
Str:
{I wanted the gold, and I sought it,
  I scrabbled and mucked like a slave.
Was it famine or scurvy -- I fought it;
  I hurled my youth into a grave.
I wanted the gold, and I got it --
```

```
  Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
  And somehow the gold isn't all.}

print [size? str "bytes"]
306 bytes
bin: compress str

print [size? bin "bytes"]
156 bytes
```

Обратите внимание, что результатом сжатия является двоичный тип данных.

Функция **decompress** распаковывает ранее сжатую строку.

```
print decompress bin
I wanted the gold, and I sought it,
  I scrabbled and mucked like a slave.
Was it famine or scurvy -- I fought it;
  I hurled my youth into a grave.
I wanted the gold, and I got it --
  Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
  And somehow the gold isn't all.
```

Сохраните ваши данные

Всегда храните несжатые резервные копии сжатых данных. Если вы потеряете только один байт из сжатого двоичного файла, восстановить данные может быть сложно. Не храните файловые архивы в сжатом формате, если у вас нет несжатых копий.

2.11 Преобразование числовой базы

Для отправки в виде текста двоичные строки должны быть преобразованы в шестнадцатеричную кодировку или кодировку `base64`. Это часто делается для электронной почты и содержания групп новостей.

Функция **enbase** закодирует двоичную строку:

```
line: to-binary "No! There's a land!"
e-line: enbase line

== "Tm8hIFRoZXJlJ3MgYSBsYW5kIQ=="
```

Закодированные строки можно декодировать с помощью функции **debase**. Обратите внимание, что строка сначала была преобразованна в двоичные данные, затем они были закодированны и результатом кодирования двочных данных является строка. Чтобы после раскодирования преобразовать их обратно в строку, используйте функцию **to-string**.

```
b-line: debase e-line
print type? b-line
binary
probe b-line
#{4E6F2120546865726527732061206C616E6421}
print to-string b-line
No! There's a land!
```

Уточнение **/base** может быть использовано с **enbase** и **debase**, чтобы указать base2 (двоичный), base16 (шестнадцатеричный), или кодирование base64.

Вот несколько примеров использования base2:

```
e2-str: enbase/base "a" 2
print e2-str
01100001
b2-str: debase/base e2-str 2
print type? b2-str
binary
probe b2-str
#{61}
print to-string b2-str
a
```

Вот несколько примеров использования base16:

```
e16-line: enbase/base line 16
print e16-line
4E6F212054686572652773206120616E6421
b16-line: debase/base e16-line 16
print type? b16-line
binary
probe b16-line
#{4E6F212054686572652773206120616E6421}
print to-string b16-line
No! There's a land!
```

2.12 Шестнадцатеричное декодирование в Интернете

Функция **dehex** преобразует символы в шестнадцатеричной кодировке в стиле URL-адреса Интернета и CGI в строки. Шестнадцатеричные представления ASCII появляются в строке URL или CGI как **%xx**, где **xx** - шестнадцатеричное значение..

```
str: "there%20seem%20to%20be%20no%20spaces"
print dehex str
there seem to be no spaces

print dehex "%68%65%6C%6C%6F"
hello
```

Глава 9 - Функции

Содержание:

1. Обзор
2. Оценка функций
  2.1 Аргументы
  2.2 Типы данных аргументов
  2.3 Уточнения
  2.4 Значения функций
3. Определение функций
  3.1 Спецификации интерфейса
  3.2 Литеральные аргументы
  3.3 Получение аргументов
  3.4 Определение уточнений
  3.5 Локальные переменные
  3.6 Локальные переменные, содержащие ряд
  3.7 Возвращение значения
  3.8 Возврат нескольких значений
4. Вложенные функции
5. Безымянные функции
6. Условные функции
7. Атрибуты функций
8. Прямые ссылки
9. Область переменных
10. Отражающие свойства
11. Справка по функциям в Интернете
12. Просмотр исходного кода

1. Обзор

REBOL предоставляет несколько видов функций:

Native	Функция, которая оценивается непосредственно процессором. Это функции самого нижнего уровня языка.
Function	Функция более высокого уровня, которая определяется блоком и оценивается путём оценки функций внутри блока. Также называется пользовательскими функциями.
Mezzanine	Имя для функций более высокого уровня, которые являются стандартной частью языка. Это не собственные функции.
Operator	Функция, которая используется как инфиксный оператор. Примеры +, -, * и /.
Routine	Функция, которая используется для вызова функций внешней библиотеки (функция REBOL/Command).

2. Оценка функций

В главе «Выражения»были описаны общие детали оценки. Способ вычисления аргументов функции определяет общий порядок слов и значений в языке. В этом разделе более подробно рассказывается о том, как оцениваются функции.

2.1 Arguments

отправить другу ** Ошибка сценария: для send отсутствует аргумент сообщения. ** Куда: отправить друга Если указано слишком много аргументов, дополнительные значения игнорируются.

Функции получают аргументы и возвращают результаты. Большинству функций требуется один или несколько аргументов; хотя некоторые функции, например **now** (текущая дата и время), не требуют аргументов вовсе.

Аргументы, которые передаются функции, обрабатываются интерпретатором и затем передаются в функцию. Аргументы обрабатываются одинаково, независимо от типа вызываемой функции, будь то собственная функция, оператор, пользовательская функция или иное. Например, функция **send** (отправить) ожидает два аргумента:

```
приятель: luke@rebol.com
сообщение: "Записка в бутылке"

send приятель сообщение
```

Сначала оценивается слово «приятель», и его значение (luke@rebol.com) предоставляется в качестве первого аргумента для **send**. Затем оценивается слово «сообщение», и его значение становится вторым аргументом. Представьте, что значения переменных «приятель» и «сообщение» подставляются в строку перед **send** (отправить):

```
send luke@rebol.com "Записка в бутылке"
```

Если вы предоставите слишком мало аргументов функции, будет возвращено сообщение об ошибке. Например, функция отправки ожидает два аргумента, и если вы отправляете один, возвращается ошибка.

```
send приятель
** Ошибка сценария: для send отсутствует аргумент сообщения.
** Где: send приятель
```

Если указано слишком много аргументов, дополнительные значения игнорируются.

```
send приятель сообщение "что-то ещё"
```

В предыдущем примере **send** уже имеет два аргумента, поэтому строка, которая является третьим аргументом, игнорируется. Обратите внимание, что сообщение об ошибке не появляется. В этом случае не было функций, ожидающих третьего аргумента. Однако в некоторых случаях третий аргумент может принадлежать другой функции, которая была оценена перед **send**.

Аргументы функции оцениваются слева направо. Этот порядок соблюдается даже тогда, когда аргументы сами являются функциями. Например, если вы напишете:

```
send приятель detab copy сообщение
```

второй аргумент должен быть вычислен путём вычисления функции **detab** и функции **copy**. Результат копирования будет передан в **detab**, а результат **detab** будет передан в **send**. В предыдущем

примере функция **copy** (копирование) принимает единственный аргумент, `сообщение`, и возвращает его копию. Скопированное сообщение передаётся в функцию **detab**, которая удаляет символы табуляции и возвращает отредактированное (без символов табуляции) сообщение, которое передаётся в функцию **send**. Обратите внимание на то, как результаты функций перетекают справа налево при вычислении выражения.
Происходящую здесь оценку можно показать с помощью скобок, чтобы пояснить, что оценивается в первую очередь. (Однако круглые скобки не обязательны и фактически немного замедляют оценку.)

```
send приятель (detab (copy сообщение))
```

Каскадный эффект результатов, передаваемых функциям, весьма полезен. Вот пример, в котором дважды используется **insert** (вставка) в одном и том же выражении:

```
file: %image
insert tail insert file %graphics/ %.jpg
print file
graphics/image.jpg
```

В следующем примере к основному имени файла добавляются имя каталога и суффикс. Скобки можно использовать для пояснения порядка оценки:

```
insert (tail (insert file %graphics/)) %.jpg
```

Примечание о круглых скобках

Скобки - это хорошие «обучающие колеса», чтобы начать писать REBOL. Однако не пройдёт много времени, прежде чем вы сможете отказаться от этой помощи и писать выражения прямо без скобок. Отсутствие круглых скобок позволяет интерпретатору быстрее вычислять выражения.

2.2 Типы данных аргумента

Функции обычно требуют аргументов определённого типа данных. Например, первым аргументом функции **send** может быть только адрес электронной почты или блок адресов электронной почты. Любой другой тип значения вызовет ошибку:

```
send 1234 "цифры"
** Ошибка сценария: send ожидает адресный аргумент типа: блок электронной почты.
** Где: send 1234 "цифры"
```

В предыдущем примере сообщение об ошибке сообщает вам, что аргумент адреса функции **send** должен быть либо адресом электронной почты, либо блоком.
Чтобы быстро узнать, какие типы аргументов принимает функция, введите в командной строке консоли следующее:

```
help send
ИСПОЛЬЗОВАНИЕ:
    SEND адрес сообщение /only /header header-obj
ОПИСАНИЕ:
    Отправляет "сообщение" на указанный "адрес" (или блок с адресами)
    SEND is a function value.
АРГУМЕНТЫ:
    адрес -- Адрес электронной почты (e-mail) или блок с адресами (Тип: email
блок)
    сообщение -- Текст сообщения. Первая строка это тема письма. (Тип: любой)
УТОЧНЕНИЯ:
    /only -- отправить только одно сообщение на несколько адресов.
    /header -- указать свой собственный заголовок
        header-obj - заголовок для использования (Тип: объект)
```

В разделе АРГУМЕНТЫ указывается тип данных каждого аргумента. Обратите внимание, что второй аргумент может иметь любой тип данных. Итак, можно написать:

```
send luke@rebol.com $1000.00
```

2.3 Уточнения

Уточнение определяет вариант нормальной оценки функции. Уточнения также позволяют указывать необязательные аргументы. Уточнения доступны как для собственных, так и для пользовательских функций.

Уточнения указываются после имени функции с дробной чертой (/) и имени уточнения. Например:

```
copy/part  (копировать только часть строки)

find/tail  (вернуть хвост совпадения)

load/markup  (возврат XML/HTML тегов и строк)
```

Функции также могут включать несколько уточнений:

```
find/case/tail (совпадение регистра и возврат хвоста)

insert/only/dup (вставить весь блок несколько раз)
```

Вы видели функцию **copy** (копирование), используемую для копирования строки. По умолчанию **copy** возвращает копию своего аргумента:

```
строка: "нет такого времени, как настоящее"
print copy строка
нет такого времени, как настоящее
```

Используя уточнение **/part**, функция **copy** возвратит часть строки:

```
print copy/part строка 10
нет такого
```

В предыдущем примере уточнение **/part** указывает, что копируются только десять символов строки.

Чтобы узнать, какие уточнения разрешены для такой функции, как **copy**, используйте встроенную справку:

```
help copy
ИСПОЛЬЗОВАНИЕ:
    COPY значение /part диапазон /deep
ОПИСАНИЕ:
     Возвращает копию "значения".
     COPY is an action value.
АРГУМЕНТЫ:
     значение - Обычно серия (Тип: битовый набор последовательного порта)
УТОЧНЕНИЯ:
     /part -- Ограничивает заданную длину или позицию.
         диапазон - (Тип: номер серия порт)
     /deep -- также копирует значения серий внутри блока.
```

Обратите внимание, что для уточнения **/part** требуется дополнительный аргумент. Не все уточнения требуют дополнительных аргументов. Например, уточнение **/deep** указывает, что **copy** делает копии всех своих подблоков. Никаких других аргументов не требуется.

Когда с функцией используется несколько уточнений, порядок дополнительных аргументов определяется порядком, в котором указаны уточнения. Например:

```
str: "test"
insert/dup/part str "this one" 4 5
print str
this this this this test
```

При изменении порядка уточнения **/dup** и **/part** изменяется порядок аргументов. Вы можете увидеть разницу:

```
str: "test"
insert/part/dup str "this one" 4 5
print str
thisthisthisthisthistest
```

Уточнения указывают порядок аргументов.

2.4 Значения функций

Предыдущие примеры описывают, как функции возвращают значения при их оценке. Однако иногда вы хотите получить функцию как значение, а не возвращаемое значение. Это можно сделать, поставив перед именем функции двоеточие или используя функцию **get**. Например, чтобы установить слово pr для функции **print** (печать), вы должны написать:

```
pr: :print
```

Вы также можете написать:

```
pr: get `print
```

Теперь **pr** эквивалентна функции **print**:

```
pr "this is a test"
this is a test
```

3. Определение функций

Вы можете определять функции, которые работают так же, как собственные функции. Это называются ***пользовательскими функциями***. Пользовательские функции относятся к типу данных **function!**.

Вы можете создавать простые функции, не требующие аргументов, с помощью функции **does**. В этом примере определяется новая функция, которая печатает текущее время:

```
print-time: does [print now/time]
print-time
10:30
```

Функция **does** возвращает значение, которое является новой функцией. В этом примере слово `print-time` установлено на функцию. Однако это значение функции может быть установлено в слово, передано другой функции, возвращено как результат функции, сохранено в блоке или немедленно оценено.

Функции, требующие аргументов, выполняются с помощью функции **func**, которая принимает два аргумента:

```
func spec body
```

Первый аргумент - это блок, определяющий интерфейс функции. Он включает описание функции, её аргументов, типы, разрешённые для аргументов, описания аргументов и другие элементы. Второй аргумент - это блок кода, который оценивается всякий раз, когда оценивается функция.
Вот пример новой функции под названием `sum:`:

```
sum: func [arg1 arg2] [arg1 + arg2]
```

Вновь определённая функция принимает два аргумента, как указано в первом блоке. Второй блок - это тело функции, которая при вычислении складывает два аргумента вместе. Новая функция возвращается как значение от **func**, и ей присваивается слово `sum`. Вот он в употреблении:

```
print sum 123 321
444
```

Результат сложения `arg1` и `arg2` возвращается и печатается.

Func определён в REBOL

Func - это функция, которая выполняет другие функции. Он выполняет тип данных **function!**. **Func** определяется как:

```
func: make function! [args body] [
    make function! args body
]
```

3.1 Спецификации интерфейса

Первый блок определения функции называется ***спецификацией интерфейса***. Этот блок включает описание функции, её аргументов, типы данных, разрешённые для аргументов, описания аргументов и другие элементы.

Спецификация интерфейса - это диалект REBOL (потому что у него другие правила оценки, чем у обычного кода). Блок спецификации имеет формат:

```
[
    "описание функции"
    [необязательные атрибуты]

    аргумент-1 [необязательный-тип]
    "описание аргумента"

    аргумент-2 [необязательный-тип]
    "описание аргумента"
```

```
    ...

    /уточнение
    "описание уточнения"

    уточнение-аргумент-1 [необязательный-тип]
    "описание аргумента уточнения"

    ...
]
```

Поля блока спецификации:

Описание	Краткое описание функции. Это строка, к которой могут обращаться другие функции, такие как `help` (помощь), чтобы вывести описания функций.
Атрибуты	Блок, описывающий особые свойства функции, например её поведение при ошибках. В будущем он может быть расширен за счёт включения флагов для оптимизации.
Аргумент	Переменная, которая используется для доступа к аргументу из тела функции.
Тип аргумента	Блок, который определяет типы данных, которые принимает функция. Если в функцию передаётся тип данных, не идентифицированный в этом блоке, произойдёт ошибка.
Описание аргумента	Краткое описание аргумента. Как и описание функции, это может быть доступно другим функциям, таким как `help`.
Уточнение	Уточняющее слово, указывающее на особое поведение функции.
Описание уточнения	Краткое описание уточнения.
Аргумент уточнения	Переменная, используемая при уточнении.
Тип аргумента уточнения	Блок, который определяет типы данных, которые принимаются уточнением.
Описание аргумента уточнения	Краткое описание аргумента уточнения.

Все эти поля необязательны.

Например, блок аргументов функции суммы (определённой в предыдущем примере) расширен, чтобы ограничить тип принимаемых аргументов. Он также включает описание функции и её ожидаемых аргументов.

```
sum: func [
    "Возвращает сумму двух чисел."
    arg1 [number!] "первое число"
    arg2 [number!] "второе число"
][
    arg1 + arg2
]
```

Теперь тип данных аргументов автоматически проверяется, выявляя такие ошибки, как:

```
print sum 1 "test"
** Ошибка скрипта: sum ожидает аргумент arg2 типа: число.
** Где: print sum 1 "test"
```

Чтобы разрешить дополнительные типы данных аргумента, можно указать их несколько разделив пробелом:

```
sum: func [
    "Возвращает сумму двух чисел."
    arg1 [number! tuple! money!] "первое число"
    arg2 [number! tuple! money!] "второе число"
][
    arg1 + arg2
]

print sum 1.2.3 3.2.1
4.4.4
print sum $1234 100
$1334.00
```

Теперь функция `sum` принимает в качестве аргументов число (number!), кортеж (tuple!) или денежное (money!) значение. Если внутри функции вам нужно различать, какой тип данных был передан, вы можете использовать функции проверки типа данных:

```
if tuple? arg1 [print arg1]

if money? arg2 [print arg2]
```

Поскольку функция `sum` предоставила строки описания, функция **`help`** теперь предоставляет полезную информацию о ней:

```
help sum
ИСПОЛЬЗОВАНИЕ:
    SUM arg1 arg2
ОПИСАНИЕ:
     Возвращает сумму двух чисел.
     SUM is a function value.
АРГУМЕНТЫ:
     arg1 -- первое число (Тип: номер кортежа денег)
     arg2 -- второе число (Тип: номер кортежа денег)
```

3.2 Буквальные аргументы

Как описано ранее, интерпретатор оценивает аргументы функций и передаёт их в тело функции. Однако бывают случаи, когда вы не хотите, чтобы аргументы функции оценивались. Например, если вам нужно передать слово и получить к нему доступ из тела функции, вы не хотите, чтобы оно оценивалось как аргумент. Функция **`help`**, ожидающая слова, является хорошим примером:

```
help print
```

Чтобы предотвратить оценку **`print`**, функция **`help`** должна указать, что её аргумент не должен оцениваться.

Чтобы указать, что аргумент не оценивается, поставьте перед именем аргумента одинарную кавычку (обозначает буквальное слово). Например:

```
zap: func [`var] [set var 0]
test: 10
zap test
print test
10
```

Аргумент `var` указан с одиночной кавычки, который инструктирует интерпретатор, что это аргумент передаётся без оценки перед передачей. Аргумент передаётся как слово. Например:

```
say: func [`var] [probe var]
say test
test
```

В примере печатается слово, переданное в качестве аргумента.

Другой пример - функция, которая увеличивает переменную на единицу и возвращает ее результат (аналогично функции приращения ++ в С):

```
++: func ['word] [set word 1 + get word]
count: 0
++ count
print count
1
print ++ count
2
```

3.3 Получение аргументов

Аргументы функции также могут указывать, что значение слова выбирается, но не оценивается. Это похоже на буквальные аргументы, описанные выше, но вместо передачи слова значение слова передаётся без оценки.

Чтобы указать, что аргумент должен быть выбран, но не вычислен, перед именем аргумента поставьте двоеточие. Например, следующая функция принимает функции в качестве аргументов:

```
print-body: func [:fun] [probe second :fun]
```

Функция-образец печатает тело переданной ей функции. Аргументу предшествует двоеточие, которое указывает, что значение слова должно быть получено, но не оцениваться в дальнейшем.

```
print-body reform
[form reduce value]

print-body rejoin
[
    if empty? block: reduce block [return block]
    append either series? first block [copy first block] [
        form first block] next block
]
```

3.4 Определение уточнений

Уточнения могут использоваться для указания вариации в обычном вычислении функции, а также для предоставления необязательных аргументов. Уточнения добавляются в блок спецификации функции в виде слова, которому предшествует дробная черта (/).

В теле функции уточняющее слово используется как логическое значение, чтобы определить, было ли уточнение предоставлено при вызове функции.

Например, следующий код добавляет уточнение к функции `sum`, которая была определена в предыдущем примере:

```
sum: func [
    "Возвращает сумму двух чисел."
    arg1 [number!] "первое число"
    arg2 [number!] "второе число"
    /average "вернуть среднее значение чисел"
][
    either average [arg1 + arg2 / 2][arg1 + arg2]
]
```

Функция `sum` указывает уточнение `/average` (среднее). В теле функции слово проверяется функцией **`either`** (либо), которая возвращает истину, если указано уточнение.

```
print sum/average 123 321
222
```

Чтобы указать уточнение, которое принимает дополнительные аргументы, следуйте уточнению с определениями аргументов:

```
sum: func [
    "Возвращает сумму двух чисел."
    arg1 [number!] "первое число"
    arg2 [number!] "второе число"
    /times "умножить результат"
    amount [number!] "сколько раз"
][
    either times [arg1 + arg2 * amount][arg1 + arg2]
]
```

Сумма умножится (`amount`) только тогда, когда указанно `times` верно. Вот пример:

```
print sum/times 123 321 10
4440
```

Не забудьте проверить уточняющее слово перед использованием дополнительных аргументов. Если аргумент уточнения используется без указания уточнения, он будет иметь значение `none`.

3.5 Локальные переменные

Локальная переменная - это слово, значение которого определяется в рамках функции. Изменения локальной переменной влияют только на функцию, в которой определена переменная. Если то же слово используется вне функции, на него не повлияют изменения в локальной переменной с тем же именем.

Переменные аргумента и уточнения являются локальными переменными. Их значения определяются в рамках функции. По соглашению дополнительные локальные переменные могут быть указаны с уточнением **/local**. За уточнением **/local** следует список слов, которые используются в качестве локальных переменных в функции.

```
average: func [
    block "Block of numbers"
    /local total length
][
    total: 0
    length: length? block
    foreach num block [total: total + num]
    either length > 0 [total / length][0]
]
```

Здесь слова total и length являются локальными для функции.

Другой метод создания локальных слов - использовать функцию **function**, которая идентична **func** , но принимает отдельный блок, содержащий локальные слова:

```
average: function [
    block "Block of numbers"
][
    total length
][
    total: 0
    length: length? block
    foreach num block [total: total + num]
    either length > 0 [total / length][0]
]
```

В этом примере обратите внимание, что уточнение **/local** не используется с функцией **function**. Функция **function** создаёт уточнения для вас.

Если локальная переменная используется до того, как её значение было установлено в теле функции, она будет иметь значение none.

```
>> b: function [c][a][a: c / 2 c + a]
>> a: 100
== 100
```

Переменная a равна 100, а внутри функции **b** ей присваивается новое значение, однако, т.к. оно объявлено локальным, то перед вызовом функции **b** интерпретатор запомнит текущее значение переменной a и по завершению функции востановит его:

```
>> print b 3
4.5
>> print a
100
```

Если не указывать a как локальную переменную внутри функции **b**, то её значение не сохраниться перед вызовом функции **b**

```
>> b: func [c][a: c / 2 c + a]
>> print b 3
4.5
>> print a
1.5
```

Переменные, указанные в качестве аргументов функций всегда являются локальными, для функции, т.е. если у них есть значения, то после оценки функции они останутся прежними, невзирая но то, что внутри функции они имеют другие значения

```
>> c: 123
== 123
>> print b 3
4.5
>> print c
123
```

3.6 Локальные переменные, содержащие серии

Локальные переменные, содержащие серии, необходимо скопировать, если серия используется несколько раз. Например, если вы хотите, чтобы строка `stars` была одинаковой каждый раз, когда вы вызываете функцию `start-name`, вы должны написать:

```
star-name: func [name] [
    stars: copy "**"
    insert next stars name
    stars
]
```

В противном случае, если вы напишете:

```
star-name: func [name] [
    stars: "**"
    insert next stars name
    stars
]
```

вы будете использовать одну и ту же строку каждый раз, и каждый раз, когда используется функция, в результате будет появляться предыдущее имя.

```
print star-name "test"
*test*
print star-name "this"
*thistest*
```

Это важно

Описанную выше концепцию важно помнить. Если вы его забудете, вы увидите странные результаты в своих программах.

3.7 Возврат значения

Как вы знаете из главы о выражениях, блоки возвращают своё последнее значение, когда они оцениваются:

```
do [1 + 3  5 + 7]
12
```

Это также верно для функций. Последне значение возвращается как значение функции:

```
sum: func [a b] [
    print a
    print b
    a + b
]

print sum 123 321

123
321
444
```

Кроме того, функцию **return** можно использовать, чтобы остановить оценку функции в любой момент и вернуть значение:

```
find-value: func [series value] [
    forall series [
        if (first series) = value [
            return series
        ]
    ]
    none
]

probe find-value [1 2 3 4] 3
[3 4]
```

В этом примере, если значение найдено, функция возвращает серию в позиции совпадения. В противном случае функция возвращает `none`.

Чтобы остановить оценку функции без возврата значения, используйте функцию **exit**:

```
source1: func [
    "Print the source code for a word"
    'word [word!]
][
    prin join word ": "
    if not value? word [print "undefined" exit]
    either any [
        native? get word op? get word action? get word
    ][
        print ["native" mold third get word]
    ][print mold get word]
]
```

3.8 Возврат нескольких значений

Чтобы вернуть более одного значения из функции, используйте блок. Вы можете сделать это легко, вернув блок, который был уменьшен.

Например:

```
find-value: func [series value /local count] [
    forall series [
        if (first series) = value [
            reduce [series  index? series]
        ]
    ]
    none
]
```

Функция возвращает блок, содержащий серию и значение индекса, в котором было найдено значение.

```
probe find-value [1 2 3 4] 3
[[3 4] 3]
```

Функция **reduce** была необходима для возврата нескольких значений из функции для версии 2.3 включительно. Для последней версии Rebol 2.7, можно её не использовать:

```
>> f: func [a] [ reduce [ a + a a * a ] ]
>> print f 3
6 9
>> f: func [a] [ [ a + a a * a ] ]
>> print f 4
8 16
```

Чтобы легко установить переменные в возвращаемое значение функции, используйте **set**:

```
set [block index] find-value [1 2 3 4] 3
print block
3 4
print index
3
```

4. Вложенные функции

Функции могут определять другие функции. Подфункции могут быть глобальными, локальными или возвращаться в результате, в зависимости от их назначения.

Например, чтобы создать глобальную функцию внутри функции, присвойте её глобальной переменной:

```
make-timer: func [code] [
    timer: func [time] code
]
make-timer [wait time]
timer 5
```

Чтобы создать локальную функцию, присвойте её локальной переменной:

```
do-timer: func [code delay /local timer] [
    timer: func [time] code
    timer delay
    timer delay
]
do-timer [wait time] 5
```

Функция `timer` существует только в период, когда оценивается функция `do-timer`.

Чтобы вернуть функцию в результате:

```
make-timer: func [code] [
    func [time] code
]
timer: make-timer [wait time]
timer 5
```

Используйте правильные локальные переменные

Вам следует избегать использования переменных, которые являются локальными для функции верхнего уровня, в качестве неоцененной части вложенной функции. Например:

```
make-timer: func [code delay] [
    timer: func [time] [wait time + delay]
]
```

В этом примере слово `delay` динамически принадлежит функции `make-timer`. Этого следует избегать, поскольку значение `delay` будет изменяться при последующих вызовах `make-timer`.

5. Безымянные функции

Имена функций - это переменные. В REBOL переменная - это переменная, независимо от того, что она содержит. В функциональных переменных нет ничего особенного.

Кроме того, функциям не требуются имена. Вы можете создать функцию и сразу оценить её, сохранить в блоке, передать в качестве аргумента функции или вернуть её как результат функции. Такие функции безымянны.
Вот пример, который создаёт блок безымянных функций:

```
funcs: []
repeat n 10 [
    append funcs func [t] compose [t + (n * 100)]
]
print funcs/1 10
110

print funcs/5 10
510
```

Функции также можно создавать и передавать другим функциям. Например, когда вы используете **sort** с вашим собственным сравнением, вы предоставляете функцию в качестве аргумента:

```
sort/compare data func [a b] [a > b]
```

6. Условные функции

Поскольку функции создаются динамически путём оценки, вы можете определить, как вы хотите создать функцию, на основе другой информации. Это способ предоставить условный код, который находится в подъязыках макросов или препроцессоров других языков программирования. В языке REBOL этот тип условного кода выполняется с помощью обычного кода REBOL.

Например, вы можете создать отладочную версию функции, которая выводит дополнительную информацию:

```
test-mode: on

timer: either test-mode [
    func [delay] [
        print "delaying..."
        wait delay
        print "resuming"
    ]
][
    func [delay] [wait delay]
]
```

Здесь вы создадите одну из двух функций в зависимости от того, в каком тестовом режиме вы работаете. Это также можно записать короче:

```
timer: func [delay] either test-mode [[
    print "delaying..."
    wait delay
    print "resuming"
]][[wait delay]]
```

7. Функциональные атрибуты

Атрибуты функции обеспечивают контроль над определённым поведением функции, например, методом, который функция использует для обработки ошибок или для выхода. Атрибуты - это необязательный блок слов в спецификациях интерфейса.

В настоящее время существует два атрибута функции: **catch** и **throw**.

Сообщения об ошибках обычно отображаются, когда они возникают внутри функции. Если указан атрибут **catch** , ошибки, возникающие внутри функции, автоматически перехватываются функцией. Ошибки отображаются не внутри функции, а в той точке, где функция использовалась. Это полезно, если вы предоставляете библиотеку функций (мезонинные функции) и не хотите, чтобы ошибка отображалась внутри вашей функции, а там, где она была вызвана:

```
root: func [[catch] num [number!]] [
    if num < 0 [
        throw make error! "только положительные числа"
    ]
    square-root num
]

root 4
2
root -4
** Ошибка пользователя: только положительные числа
** Где: root -4
```

Обратите внимание, что в этом примере ошибка возникает там, где был вызван `root`, даже если фактическая ошибка была сгенерирована в теле функции. Это потому, что использовался атрибут **catch** .

Без атрибута **catch** ошибка возникла бы в функции `square-root`:

```
root: func [num [number!]] [
    square-root num
]
root -4
** Математическая ошибка: требуется положительное число.
** Где: square-root num
```

Пользователь может ничего не знать о внутреннем устройстве функции `root`. Таким образом, сообщение об ошибке может сбивать с толку. Пользователь знает только о `root`, но ошибка была с **square-root**.

Не путайте атрибут **catch** с функцией **catch** . Хотя они похожи, функция **catch** может быть применена к любому блоку, который оценивается.

Атрибут **throw** позволяет создавать свои собственные функции управления, такие как **for**, **foreach**, **if**, **loop**, и **forever**, позволяя ваши функции передать **return** и **exit** операции. Например, эта функция цикла:

```
loop-time: func [time block] [
    while [now/time < time] block
]
```

оценивает блок до тех пор, пока не наступит или не пройдёт определённое время. Затем этот цикл можно использовать в функции:

```
do-job: func [job][
    loop-time 10:30 [
        if error? try [page: read http://www.rebol.com]
            [return none]
    ]
    page
]
```

Что же происходит при вычислении блока `[return none]`? Поскольку этот блок оценивается функцией `loop-time`, возврат происходит в этой функции, а не в `do-job`.

Этого можно избежать атрибутом **throw**:

```
loop-time: func [[throw] time block] [
    while [now/time < time] block
]
```

Атрибут **throw** вызывает **return** или **exit**, который произошёл в блок, чтобы быть выброшен до прежнего уровня, что следующая функция вызываетdo-job для возврата.

8. Перспективные ссылки

Иногда скрипту необходимо обратиться к функции до того, как она будет определена. Это может быть сделано до тех пор, пока переменная для функции не вычисляется до её определения.

```
buy: func [item] [
    append own item
    sell head item    ; появляется до того, как он будет определён
]

sell: func [item] [
    remove find own item
]
```

9. Объем переменных

Контекст переменных называется их областью действия. Широкий диапазон переменных - это глобальные и локальные. REBOL использует форму статической области видимости, которая называется определением области действия. Область действия переменной определяется при определении её контекста. В случае функции это определяется тем, когда функция определена.

Все локальные переменные, определённые в функции, имеют область видимости относительно этой функции. Вложенные функции и объекты могут получить доступ к словам своих родителей.

```
a-func: func [a] [
    print ["a:" a]
    b-func: func [b] [
        print ["b:" b]
        print ["a:" a]
        print a + b
    ]
    b-func 10
]
a-func 11
a: 11
b: 10
a: 11
21
```

Обратите внимание, что b-func имеет доступ к переменной a-func.

Слова, привязанные вне функции, сохраняют эти привязки даже при вычислении внутри функции. Это результат статической области видимости, и он позволяет вам писать свои собственные функции оценки блоков (например **if**, **while**, **loop**).

Например, вот функция **if**, которая оценивает один из трёх блоков на основе знака условного значения:

```
ifs: func [
    "Если значение положительное, то выполнится block 1, равно нулю block 2,
отрицательное ,block 3"
    condition block1 block2 block3
][
    if positive? condition [return do block1]
    if negative? condition [return do block3]
    return do block2
]

print ifs 12:00 - now/time ["утро"]["полдень"]["вечер"]
вечер
```

Переданные блоки могут содержать те же слова, которые используются в функции `ifs`, не влияя на слова, определённые локально для функции. Это потому, что слова, переданные в функцию, не связаны с функцией.

Следующий пример передаёт слова `block1`, `block2` и `block3` к `ifs`, как предварительно определённых слов. Функция `ifs` не путается между словами, переданными в качестве аргументов, и словами с тем же именем, определёнными локально:

```
block1: "сейчас утро"
block2: "только что миновал полдень"
block3: "вечернее время"

print ifs (12:00 - now/time) [block1][block2][block3]
вечернее время
```

10. Отражающие свойства.

Спецификация всех функций может быть получена и изменена во время выполнения. Например, вы можете распечатать блок спецификации для функции с помощью:

```
probe third :if
[
    "Если условие ИСТИНА, оценивает блок"
    condition
    then-block [block!]
    /else "Если не верно, оценить этот блок"
    else-block [block!]
]
```

Основной код функций можно получить с помощью:

```
probe second :append
[
    head either only [
        insert/only tail series :value
    ][
        insert tail series :value
    ]
]
```

Функции могут запрашиваться динамически во время оценки. Так работают функции **help** (справки) и **source** (исходный код) и как форматируются сообщения об ошибках.

Кроме того, эта функция полезна для создания ваших собственных уникальных версий существующих функций. Например, может быть создана определяемая пользователем функция print (печать), которая имеет точно такую же спецификацию, что и **print**, но отправляет свой вывод в строку, а не на дисплей:

```
output: make string! 1000

print-str: func third :print [
    repend output [reform :value newline]
]
```

Имя аргумента, используемого для print-str, берётся из спецификации интерфейса для print. Вы можете изучить эту спецификацию с помощью:

```
probe third :print
[
    «Выводит значение, за которым следует разрыв строки».
    value «Значение для печати»
]
```

11. Встроенная справка

Полезную информацию обо всех функциях системы можно получить с помощью функции **help**:

```
help send
ИСПОЛЬЗОВАНИЕ:
    SEND адрес сообщение /only /header header-obj
ОПИСАНИЕ:
     Отправить сообщение на адрес (или блок адресов)
     SEND - значение функции.
АРГУМЕНТЫ:
     адрес - адрес или блок адресов (Тип: блок электронной почты)
     сообщение - Текст сообщения. Первая строка - тема. (Тип: любой)
УТОЧНЕНИЯ:
     / only - отправить только одно сообщение на несколько адресов.
     / header - укажите свой собственный заголовок
         header-obj - заголовок для использования (Тип: объект)
```

Вся эта информация исходит из определения функции. Справку можно получить по всем типам функций, а не только по родным или встроенным функциям. Функция справки также может использоваться для пользовательских функций. Документация, которая отображается о функции, предоставляется при её определении.

Вы также можете выполнить поиск по функциям, содержащим различные шаблоны. Например, в командной строке вы можете ввести

```
Help "path"
Found these words:
     clean-path      (function)
     lit-path!       (datatype)
```

```
    lit-path?        (action)
    path!            (datatype)
    path-thru        (function)
    path?            (action)
    set-path!        (datatype)
    set-path?        (action)
    split-path       (function)
    to-lit-path      (function)
    to-path          (function)
    to-set-path      (function)
```

для отображения всех слов, содержащих `path` к строке.

Чтобы просмотреть список всех функций, доступных в REBOL, введите **`what`** в командной строке.

```
what
* [value1 value2]
** [number exponent]
+ [value1 value2]
- [value1 value2]
/ [value1 value2]
// [value1 value2]
< [value1 value2]
<= [value1 value2]
<> [value1 value2]
= [value1 value2]
== [value1 value2]
=? [value1 value2]
> [value1 value2]
>= [value1 value2]
? ['word]
?? ['name]
about []
abs [value]
absolute [value]
...
```

12. Просмотр исходного кода

Другой метод изучения REBOL и экономии времени при написании собственной функции - это посмотреть, сколько мезонинных функций REBOL определено. Для этого вы можете использовать функцию **`source`**.

```
source source
source: func [
    "Prints the source code for a word."
    'word [word!]
][
    prin join word ": "
    if not value? word [print "undefined" exit]
    either any [native? get word op? get word action? get word] [
        print ["native" mold third get word]
    ] [print mold get word]
]
```

Здесь функция **`source`** используется для печати собственного исходного кода.

Обратите внимание, что вы не можете увидеть исходный код для собственных функций, потому что они существуют только как машинный код. Однако исходная функция будет отображать спецификацию интерфейса собственной функции. Например:

```
source add
add: native [
    "Returns the result of adding two values."
    value1 [number! pair! char! money! date! time! tuple!]
    value2 [number! pair! char! money! date! time! tuple!]
]
```

Глава 10 - Объекты

Содержание:

1. Обзор
2. Создание объектов
3. Клонирование объектов
4. Доступ к объектам
5. Функции объекта
6. Прототип объектов
7. Обращение к себе
8. Инкапсуляция
9. Отражающие свойства

1. Обзор

Объекты группируют значения в общий контекст. Объект может включать в себя скалярные значения, серии, функции и другие объекты. Объекты полезны при работе со сложными структурами, поскольку они позволяют инкапсулировать связанные данные и код и передавать их функциям как одно значение.

2. Создание объектов

Новые объекты создаются с помощью функции **make**. Функция **make** требует два аргумента и возвращает новый объект. Формат функции **make**:

```
new-object: make parent-object new-values
```

Первый аргумент, `parent-object`, - это родительский объект, из которого создаётся новый объект. Если родительский объект недоступен, как при определении начального объекта, используйте тип данных **object!**, как показано ниже:

```
new-object: make object! new-values
```

Второй аргумент, `new-values`, представляет собой блок, который определяет дополнительные переменные и начальные значения для нового объекта. Каждая переменная, которая определена в блоке, является переменной экземпляра объекта. Например, если блок содержит два определения переменных, то они будут переменными объекта:

```
example: make object! [
    var1: 10
    var2: 20
]
```

У объекта `example` есть две переменные, которые содержат два целых числа.

Блок оценивается, поэтому он может включать любой тип выражения для вычисления значений переменных:

```
example: make object! [
    var1: 10
    var2: var1 + 10
    var3: now/time
]
```

После того, как объект создан, он может служить прототипом для создания новых объектов:

```
example2: make example []
```

В приведённом выше примере создаётся второй экземпляр объекта `example`. Новый объект является клоном первого объекта. В блоке задаются новые значения для второго объекта:

```
example2: make example [
    var1: 30
    var2: var1 + 10
]
```

В приведённом выше примере объект `example2` имеет другие значения, чем исходный объект `example` для двух своих переменных.

Объект `example2` также может расширять определение объекта, добавляя к нему новые переменные:

```
example2: make example [
    var4: now/date
    var5: "пример"
]
```

В результате получается объект с пятью переменными: три исходные из исходного объекта `example` и две новые.
Процесс расширения определения объекта можно повторять любое количество раз.

Вы также можете создать объект, содержащий переменные, которые инициализируются некоторым общим значением. Это можно сделать с помощью каскадного набора определений слов:

```
example3: make object! [
    var1: var2: var3: var4: none
]
```

В приведённом выше примере для четырёх переменных в новом объекте установлено значение `none`.
Подводя итог, процесс создания объекта включает следующие шаги:

- Использовать **make** для создания нового объекта на основе родительского объекта или `object!` тип данных.
- Добавьте к новому объекту любые новые переменные, которые определены в блоке.
- Оцените блок, в результате чего переменные, определённые в блоке, будут установлены в значения в новом объекте.
- В результате возвращается новый объект.

3. Клонирование объектов

Когда вы используете родительский объект для создания нового объекта, родительский объект клонируется, а не наследуется. Это означает, что если родительский объект изменяется, это не влияет на дочерний объект.
В качестве примера следующий код создаёт объект банковского счета, переменные которого пусты:

```
bank-account: make object! [
    first-name:
    last-name:
    account:
    balance: none
]
```

Чтобы использовать новый объект, можно указать значения для создания учётной записи для клиента:

```
luke: make bank-account [
    first-name: "Luke"
    last-name: "Lakeswimmer"
    account: 89431
    balance: $1204.52
]
```

Поскольку новые учётные записи создаются на регулярной основе, для их создания полезно использовать функцию и некоторые глобальные переменные:

```
last-account: 89431
bank-bonus: $10.00
make-account: func [
    "Возвращает новый объект учётной записи"
    f-name [string!] "Имя"
    l-name [string!] "Фамилия"
    start-balance [money!] "Начальный баланс"
][
    last-account: last-account + 1
    make bank-account [
        first-name: f-name
        last-name: l-name
        account: last-account
        balance: start-balance + bank-bonus
    ]
]
```

Теперь новый объект учётной записи для `Fred` (Фреда) потребует только:

```
fred: make-account "Fred" "Smith" $500.00
```

4. Доступ к объектам

Доступ к переменным внутри объектов осуществляется с помощью путей. Путь состоит из имени объекта, за которым следует имя переменной. Например, следующий код обращается к переменным в объекте `example`:

```
example/var1
example/var2
```

Вот примеры использования объекта bank-account:

```
print luke/last-name
Lakeswimmer
print fred/balance
$510.00
```

Используя путь, можно также изменить переменные объекта:

```
fred/balance: $1000.00
print fred/balance
$1000.00
```

Вы можете использовать функцию **in** для доступа к объектным переменным, извлекая их слова из их объектного контекста:

```
print in fred 'balance
balance
```

Слово balance возвращается имеет объект fred в качестве контекста. Вы можете получить значение, которое он содержит, используя **get:**:

```
print get in fred 'balance
$1000.00
```

Второй аргумент функции **in** - буквальное слово. Это позволяет вам динамически менять слова в зависимости от того, что вам нужно:

```
words: [first-name last-name balance]
foreach word words [print get in fred word]
FredSmith
$1000.00
```

Каждое слово в блоке используется для получения своего значения в объекте.

В функции **in** также могут быть использованы для установки переменных объекта.

```
set in fred 'balance $20.00
print fred/balance
$20.00
```

Если слово не определено в объекте, функция **in** возвращает none. Это полезно для определения наличия переменной в объекте.

```
if get in fred 'bank [print fred/bank]
```

5. Функции объекта

Объект может содержать переменные, которые относятся к функциям, которые определены в контексте объекта. Это полезно, потому что функции инкапсулированы в контексте объекта и могут обращаться к другим переменным объекта напрямую, без необходимости указывать путь.
В качестве простого примера объект `example` может включать в себя функции для вычисления новых значений внутри объекта:

```
example: make object! [
    var1: 10
    var2: var1 + 10
    var3: now/time
    set-time: does [var3: now/time]
    calculate: func [value] [
        var1: value
        var2: value + 10
    ]
]
```

Обратите внимание, что в этом примере функции могут ссылаться на переменные объекта напрямую, а не как пути. Это возможно, потому что функции определены в том же контексте, что и переменные, к которым они обращаются.
Чтобы установить новое время, используйте:

```
example/set-time
```

В этом примере оценивается функция, которая устанавливает `var3` на текущее время.

Чтобы вычислить новые значения для `var1` и `var2`, используйте:

```
example/calculate 100
print example/var2
110
```

В случае объекта `bank-account` к текущему определению могут быть добавлены функции депозита (`deposit`) и снятия (`withdraw`) средств:

```
bank-account: make bank-account [
    deposit: func [amount [money!]] [
        balance: balance + amount
    ]
    withdraw: func [amount [money!]] [
        either negative? balance [
            print ["Отказано. Перерасход "
                absolute balance]
        ][balance: balance - amount]
    ]
]
```

Обратите внимание, что в этом примере функции могут ссылаться на `balance` (баланс) непосредственно внутри объекта. Это потому, что функции являются частью контекста объекта.

Теперь, если создаётся новый счёт, он будет содержать функции для ввода и вывода денег. Например:

```
lily: make-account "Lily" "Lakeswimmer" $1000

print lily/balance
$1010.00
lily/deposit $100

print lily/balance
$1110.00
lily/withdraw $2000

print lily/balance
-$890.00
lily/withdraw $2.10
Отказано. Перерасход $890.00
```

6. Прототип объектов

Любой объект может служить прототипом для создания новых объектов. Например, ранее определенный объект учетной записи `lily` можно использовать для создания новых объектов с такой строкой, как:

```
maya: make lily []
```

Это создаёт экземпляр объекта. Объект является копией объекта клиента и имеет идентичные значения:

```
print lily/balance
-$890.00
print maya/balance
-$890.00
```

Вы можете изменить новый объект при его создании, указав новые значения в блоке определения:

```
maya: make lily [
    first-name: "Maya"
    balance: $10000
]

print maya/balance
$10000.00
maya/deposit $500

print maya/balance
$10500.00
print maya/first-name
Maya
```

Объект `lily` служит прототипом для создания нового объекта. Любые слова, которые не были переопределены для нового объекта, продолжают иметь значения старого объекта:

```
print maya/last-name
Lakeswimmer
```

Аналогичным образом к объекту добавляются новые слова:

```
maya: make lily [
    email: maya@example.com
    birthdate: 4-July-1977
]
```

7. Обращение к себе

Каждый объект включает предопределённую переменную под названием `self`. В контексте объекта переменная `self` относится к самому объекту. Его можно использовать для передачи объекта другим функциям или для возврата его в результате выполнения функции.

В следующем примере функции `how-date` требуется объект в качестве аргумента, и ей передаётся `self`:

```
show-date: func [obj] [print obj/date]

example: make object! [
    date: now
    show: does [show-date self]
]

example/show
16-Jul-2000/11:08:37-7:00
```

Другой пример использования переменной В следующем примере функции `how-date` требуется объект в качестве аргумента, и ей передаётся `self`: - это функция, которая клонирует себя:

```
person: make object! [
    name: days-old: none
    new: func [name' birthday] [
        make self [
            name: name'
            days-old: now/date - birthday
        ]
    ]
]

lulu: person/new "Lulu Ulu" 17-May-1980

print lulu/days-old
7366
```

8. Инкапсуляция

Объект предоставляет хороший способ инкапсулировать группу переменных, которые не должны появляться на глобальном уровне. Когда функциональные переменные определены как глобальные, они могут быть непреднамеренно изменены другими функциями.

Решение этой проблемы глобальных переменных состоит в том, чтобы обернуть объект как переменными, так и функцией. Когда это будет сделано, функция все ещё может получить доступ к переменным, но переменные не будут доступны глобально.

Например:

```
Bank: make object! [

    last-account: 89431
    bank-bonus: $10.00

    set 'make-account func [
        "Возвращает новый объект учётной записи"
        f-name [string!] "Имя"
        l-name [string!] "Фамилия"
        start-balance [money!] "Начальный баланс"
    ][
        last-account: last-account + 1
        make bank-account [
            first-name: f-name
            last-name: l-name
            account: last-account
            balance: start-balance + bank-bonus
        ]
    ]
]
```

В этом примере переменные защищены от случайного изменения. Обратите внимание, что функция `make-account` была установлена для переменной с помощью функции **`set`**, а не с использованием определения переменной. Это было сделано для того, чтобы сделать его глобальной функцией. Функцию можно использовать так же, как функции, заданные с помощью определения переменной, но для неё не требуется путь к объекту:

```
bob: make-account "Bob" "Baker" $4000
```

9. Отражающие свойства.

Как и в случае со многими другими типами данных REBOL, вы можете получить доступ к компонентам объектов таким образом, чтобы вы могли писать полезные инструменты и утилиты для их создания, мониторинга и отладки.

Функции **`first`** и **`second`** позволяет получить доступ к компонентам объекта. Функция **`first`** возвращает слова, определённые для объекта. Функция **`second`** возвращает значения, объекты, установленные в этих словах. На следующей диаграмме показана взаимосвязь между возвращаемыми значениями **`first`** и **`second`**:

first	second
firstname	"Luke"
lastname	"Lakeswimmer"
account	89431
balance	$1204.52

Преимущество использования **`first`** заключается в том, что он позволяет вам получить список слов для функции, ничего не зная о функции:

```
probe first luke
[self first-name last-name account balance]
```

Обратите внимание, что в приведённом выше примере список содержит слово `self`, которое является ссылкой на сам объект. Вы можете исключить `self` при получении списка слов объекта, используя **`next`**:

```
probe next first luke
[first-name last-name account balance]
```

Теперь у вас есть способ написать функцию, которая может проверять содержимое объекта:

```
probe-object: func [object][
    foreach word next first object [
        print rejoin [word ":" tab get in object word]
    ]
]

probe-object fred
first-name: Luke
last-name: Lakeswimmer
account: 89431
balance: $1204.52
```

При доступе к объектам таким образом следует проявлять осторожность, чтобы избежать бесконечных циклов. Например, если вы попытаетесь исследовать определённые объекты, содержащие ссылки на самих себя, ваш код может начать бесконечный цикл. Это причина, по которой вы не можете напрямую исследовать системный объект **`system`**. Объект **`system`** содержит много ссылок на себя.

Глава 11 - Математика

Содержание:

1. Обзор
2. Скалярные типы данных
3. Порядок оценки
4. Стандартные функции и операторы
4.1 absolute
4.2 add
4.3 complement
4.4 divide
4.5 multiply
4.6 negate
4.7 random
4.8 remainder
4.9 subtract
5. Преобразование типов
6. Функции сравнения
6.1 equal
6.2 greater
6.3 greater-or-equal
6.4 lesser
6.5 lesser-or-equal
6.6 not equal to
6.7 same
6.8 strict-equal
6.9 strict-not-equal
7. Логарифмические функции
7.1 exp
7.2 log-10
7.3 log-2
7.4 log-e
7.5 power
7.6 square-root
8. Тригонометрические функции
8.1 arccosine
8.2 arcsine
8.3 arctangent
8.4 cosine
8.5 sine
8.6 tangent
9. Логические функции
9.1 and
9.2 or
9.3 xor
9.4 complement
9.5 not
10. Ошибки
10.1 Попытка деления на ноль
10.2 Математическое или числовое переполнение
10.3 Требуется положительное число
10.4 Невозможно используйте оператор для значений типа datatype!

1. Обзор

REBOL предоставляет полный набор математических и тригонометрических операций. Многие из этих операторов могут обрабатывать несколько типов данных, включая целые, десятичные, деньги, кортежи, время и дату. Некоторые из этих типов данных могут даже быть смешанными или принудительными.

2. Скалярные типы данных

Математические функции REBOL работают согласованно с широким спектром скалярных (числовых) типов данных. Эти типы данных включают:

Тип данных	Описание
`Integer!`(Целое!)	32-битные числа без десятичной точки
`Decimal!`(Десятичный!)	64-битные числа с плавающей запятой
`Money!`(Деньги!)	валюта с 64-битным числом с плавающей запятой
`Time!`(Время!)	часы, минуты, секунды и доли секунды
`Date!`(Дата!)	день, месяц, год, время, часовой пояс
`Pair!`(Пара!)	графическое положение или размер
`Tuple!`(Кортеж!)	версии, цвета, сетевые адреса

Ниже приведены несколько примеров, демонстрирующих ряд математических операций над скалярными типами данных. Обратите внимание, что операторы дают полезные результаты для каждого типа данных.

В `integer` (целочисленных) и `decimal` (десятичных) типах данных:

```
print 2 + 1
3
print 2 - 1
1
print 2 * 10
20
print 20 / 10
2
print 21 // 10
1
print 2.2 + 1
3.2
print 2.2 - 1
1.2
print 2.2 * 10
22
print 2.2 / 10
0.22
print random 10
5
```

В **time** (время) типе данных:

```
print 2:20 + 1:40
4:00
print 2:20 + 5
2:20:05
print 2:20 + 60
2:21
print 2:20 + 2.2
2:20:02.2
print 2:20 - 1:20
1:00
print 2:20 - 5
2:19:55
print 2:20 - 120
2:18
print 2:20 * 2
4:40
print 2:20 / 2
1:10
print 2:20:01 / 2
1:10:00.5
print 2:21 // 2
0:00
print - 2:20
-2:20
print random 10:00
5:30:52
```

В **date** (дата) типе данных:

```
print 1-Jan-2000 + 1
2-Jan-2000
print 1-Jan-2000 - 1
31-Dec-1999
print 1-Jan-2000 + 31
1-Feb-2000
print 1-Jan-2000 + 366
1-Jan-2001
birthday: 7-Dec-1944
print ["Я живу " (now/date - birthday) " дней."]
Я живу 20305 дней.
print random 1-1-2000
29-Apr-1695
```

В **money** (деньги) типе данных:

```
print $2.20 + $1
$3.20
print $2.20 + 1
$3.20
print $2.20 + 1.1
$3.30
print $2.20 - $1
$1.20
print $2.20 * 3
$6.60
```

```
print $2.20 / 2
$1.10
print $2.20 / $1.10
2
print $2.21 // 2
$0.21
print random $10.00
$6.00
```

В **pair** (пара) типе данных:

```
print 100x200 + 10x20
110x220
print 10x10 + 3
13x13
print 10x20 * 2x4
20x80
print 100x100 * 3
300x300
print 100x30 / 10x3
10x10
print 100x30 / 10
10x3
print 101x32 // 10x3
1x2
print 101x32 // 10
1x2
print random 100x20
67x12
```

В **tuple** (кортеж) типе данных:

```
print 1.2.3 + 3.2.1
4.4.4
print 1.2.3 - 1.0.1
0.2.2
print 1.2.3 * 3
3.6.9
print 10.20.30 / 10
1.2.3
print 11.22.33 // 10
1.2.3
print 1.2.3 * 1.2.3
1.4.9
print 10.20.30 / 10.20.30
1.1.1
print 1.2.3 + 7
8.9.10
print 1.2.3 - 1
0.1.2
print random 10.20.30
8.18.12
```

3. Порядок оценки

При вычислении математических выражений следует помнить два правила:
- Выражения: оцениваются слева направо.
- Операторы имеют приоритет над функциями.

Оценка выражений слева направо не зависит от типа используемого оператора. Например:

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

Обратите внимание, что в приведённом выше примере результат не равен семи, как в случае, если бы умножение имело приоритет перед сложением.

Важная заметка

Способ вычисления математических выражений слева направо независимо от оператора отличается от многих других компьютерных языков. Во многих языках есть правила приоритета, которые вы должны помнить, определяющие порядок вычисления операторов. Например, умножение выполняется перед сложением. В некоторых языках есть 10 или более таких правил.

В REBOL вместо того, чтобы требовать от пользователей запоминания приоритета операторов, вам нужно помнить только правило письма слева направо. Что ещё более важно, для расширенного кода, такого как Выражения:, которые обрабатывают Выражения: (например, в отражении), вам не нужно переупорядочивать термины на основе приоритета. Порядок оценки остаётся простым.

Для большинства математических выражений правило вычисления слева направо работает достаточно хорошо и его легко запомнить. Однако, поскольку это правило отличается от других языков, оно может быть источником ошибок программирования, так что будьте осторожны.

Лучшее решение - проверить свою работу. Вы также можете использовать круглые скобки, если необходимо, чтобы пояснить своё выражение (см. ниже), и вы всегда можете ввести своё выражение в консоли, чтобы проверить свой результат.

Если вам нужно выполнить оценку в другом порядке, измените порядок Выражения: или используйте круглые скобки:

```
print 2 * 3 + 1
7
print 1 + (2 * 3)
7
```

Когда функции смешиваются с операторами, сначала оцениваются операторы, а затем функции:

```
print absolute -10 + 5
5
```

В приведённом выше примере сначала выполняется сложение, а его результат предоставляется функции **absolute**.

В следующем примере:

```
print 10 + sine 30 + 60
11
```

выражение оценивается в следующем порядке:

```
30 + 60 => 90
sine 90 => 1
10 + 1 => 11
print
```

Чтобы изменить порядок так, чтобы сначала использовался **синус** 30, используйте круглые скобки:

```
print 10 + (sine 30) + 60
70.5
```

или измените порядок Выражения:

```
print 10 + 60 + sine 30
70.5
```

4. Стандартные функции и операторы

В этом разделе описаны стандартные математические функции и операторы, используемые в REBOL.

4.1 absolute (абсолютное)

Выражения:

```
absolute value

abs value
```

вернуть абсолютное (по модулю, т.е. положительное) значение `value`.

Работает со следующими типами данных - **integer**(целыми), **decimal**(десятичными), **money**(деньги), **time**(время), **pair**(пары).

```
print absolute -10
10
print absolute -1.2
1.2
print absolute -$1.2
$1.20
print absolute -10:20
10:20
print absolute -10x-20
10x20
```

4.2 add (добавить)

Выражения:

```
value1 + value2

add value1 value2
```

возвращает результат добавления `value1` к `value2`.

Работает со следующими типами данных **integer**(целыми), **decimal**(десятичными), **money**(деньги), **time**(время), **tuple**(кортеж), **pair**(пары), **date**(дата), **char**(символ).

```
print 1 + 2
3
print 1.2 + 3.4
4.6
print 1.2.3 + 3.4.5
4.6.8
print $1 + $2
$3.00
print 1:20 + 3:40
5:00
print 10x20 + 30x40
40x60
print #"A" + 10
K
print add 1 2
3
```

4.3 complement (дополнение)

Выражение:

```
complement value
```

Возвращает числовое дополнение (побитовое дополнение) значения.

Возвращает значение дополнения до единицы для value.

Работает со следующими типами данных **integer**(целыми), **decimal**(десятичными), **tuple**(кортеж).

```
print complement 10
-11
print complement 10.5
-11
print complement 100.100.100
155.155.155
```

4.4 divide (деление)

Выражения:

```
value1 / value2
divide value1 value2
```

возвращает результат деления `value1` на `value2`.

Работает со следующими типами данных **integer**(целыми), **decimal**(десятичными), **money**(деньги), **time**(время), **tuple**(кортеж), **pair**(пары), **char**(символ).

```
print 10 / 2
5
print 1.2 / 3
0.4
print 11.22.33 / 10
1.2.3
print $12.34 / 2
$6.17
print 1:20 / 2
0:40
print 10x20 / 2
5x10
print divide 10 2
5
```

4.5 multiply (умножение)

Выражения:

```
value1 * value2
multiply value1 value2
```

возвращает результат умножения `value1` на `value2`.

Работает со следующими типами данных: **`integer`**(целыми), **`decimal`**(десятичными),**`money`**(деньги), **`time`**(время), **`tuple`**(кортеж),**`pair`**(пары), **`char`**(символ).

```
print 10 * 2
20
print 1.2 * 3.4
4.08
print 1.2.3 * 3.4.5
3.8.15
print $10 * 2
$20.00
print 1:20 * 3
4:00
print 10x20 * 3
30x60
print multiply 10 2
20
```

4.6 negate (отрицание)

Выражения:

```
- value

negate value
```

изменить знак `value`.

Работает со следующими типами данных: `integer`(целыми), `decimal`(десятичными),`money`(деньги), `time`(время),`pair`(пары).

```
print - 10
-10
print - 1.2
-1.2
print - $10
-$10.00
print - 1:20
-1:20
print - 10x20
-10x-20
print negate 10
-10
```

4.7 random (случайный)

Выражение:

```
random value
```

возвращает случайное значение, которое меньше или равно заданному значению.

Обратите внимание, что для целых чисел `random` начинается с 1, а не с 0, и включает данное значение. Это позволяет использовать `random` напрямую с такими функциями, как `pick`.

Когда используется десятичное число, результатом является десятичный тип данных, округлённый до целого числа.

Уточнение `/seed` перезапускает генератор случайных чисел. Используйте уточнение `/seed` если вы хотите генерировать уникальные случайные числа. Вы можете использовать текущую дату и время, чтобы сделать начальное число более случайным:

```
random/seed now
```

Работает со следующими типами данных: `integer`(целыми), `decimal`(десятичными),`money`(деньги), `time`(время), `tuple`(кортеж),`pair`(пары), `char`(символ), `string`(строка), `block`(блок).

```
print random 10
5
print random 10.5
2
print random 100.100.100
79.95.66
print random $100
$32.00
print random 10:30
6:37:33
print random 10x20
2x4
print random 30-Jun-2000
27-Dec-1171
```

4.8 remainder (остаток)

Выражения:

```
value1 // value2

remainder value1 value2
```

Возвращает остаток от первого значения, делённого на второе.

Работает со следующими типами данных: **integer**(целыми), **decimal**(десятичными),**money**(деньги), **time**(время), **tuple**(кортеж),**pair**(пары).

```
print 11 // 2
1
print 11.22.33 // 10
1.2.3
print 11x22 // 2
1x0
print remainder 11 2
1
```

4.9 subtract (вычесть)

Выражения:

```
value1 - value2

subtract value1 value2
```

возвращает результат вычитания `value2` из `value1`.

Работает со следующими типами данных: **integer**(целыми), **decimal**(десятичными),**money**(деньги), **time**(время), **tuple**(кортеж),**pair**(пары), **char**(символ), **data**(дата).

```
print 2 - 1
1
print 3.4 - 1.2
2.2
print 3.4.5 - 1.2.3
2.2.2
print $2 - $1
$1.00
print 3:40 - 1:20
2:20
print 30x40 - 10x20
20x20
print #"Z" - 1
Y
print subtract 2 1
1
```

5. Преобразование типов

Когда между типами данных выполняются математические операции, обычно возвращается нецелочисленный или недесятичный тип данных. Когда целые числа комбинируются с десятичными числами, возвращается десятичный тип данных.

6. Функции сравнения

Все функции сравнения возвращают либо `true` (истина) или `false` (ложь).

6.1 equal (равно)

Выражения:

```
value1 = value2

equal? value1 value2
```

вернёт `true` (истину) если первое и второе значения равны..

Работает со следующими типами данных: **`integer`**(целыми), **`decimal`**(десятичными),**`money`**(деньги), **`time`**(время), **`tuple`**(кортеж), **`char`**(символ), **`data`**(дата) и **`series`** (серия).

```
print 11-11-99 = 11-11-99
true
print equal? 111.112.111.111 111.112.111.111
true
print #"B" = #"B"
true
print equal? "a b c d" "A B C D"
true
```

6.2 greater (больше)

Выражения:

```
value1 > value2

greater? value1 value2
```

Возвращает `true` истину если первое значение больше второго.

Работает со следующими типами данных: **`integer`**(целыми), **`decimal`**(десятичными),**`money`**(деньги), **`time`**(время), **`tuple`**(кортеж), **`char`**(символ), **`data`**(дата) и **`series`** (серия).

```
print 13-11-99 > 12-11-99
true
print greater? 113.111.111.111 111.112.111.111
true
```

```
print #"C" > #"B"
true
print greater? [12 23 34] [12 23 33]
true
```

6.3 greater-or-equal (больше или равно)

Выражения:

```
value1 >= value2

greater-or-equal? value1 value2
```

Возвращает `true` истину если первое значение больше или равно второму.

Работает со следующими типами данных: **integer**(целыми), **decimal**(десятичными),**money**(деньги), **time**(время), **tuple**(кортеж), **char**(символ), **data**(дата) и **series** (серия).

```
print 11-12-99 >= 11-11-99
true
print greater-or-equal? 111.112.111.111 111.111.111.111
true
print #"B" >= #"A"
true
print greater-or-equal? [b c d e] [a b c d]
true
```

6.4 lesser (меньше)

Выражения:

```
value1 < value2

lesser? value1 value2
```

Вернёт `true` если первое значение меньше второго.

Работает со следующими типами данных: **integer**(целыми), **decimal**(десятичными),**money**(деньги), **time**(время), **tuple**(кортеж), **char**(символ), **data**(дата) и **series** (серия).

```
print 25 < 50
true
print lesser? 25.3 25.5
true
print $2.00 < $2.30
true
print lesser? 00:10:11 00:11:11
true
```

6.5 lesser-or-equal (меньше или равно)

Выражения:

```
value1 <= value2

lesser-or-equal? value1 value2
```

Вернёт `true` если первое значение меньше или равно второму value.

Работает со следующими типами данных: **`integer`**(целыми), **`decimal`**(десятичными),**`money`**(деньги), **`time`**(время), **`tuple`**(кортеж), **`char`**(символ), **`data`**(дата) и **`series`** (серия).

```
print 25 <= 25
true
print lesser-or-equal? 25.3 25.5
true
print $2.29 <= $2.30
true
print lesser-or-equal? 11:11:10 11:11:11
true
```

6.6 not equal to (не равно)

Выражения:

```
value1 <> value2

not-equal? value1 value2
```

Вернёт `true` если первое значение не равно второму.

Работает со следующими типами данных: **`integer`**(целыми), **`decimal`**(десятичными),**`money`**(деньги), **`time`**(время), **`tuple`**(кортеж), **`char`**(символ), **`data`**(дата) и **`series`** (серия).

```
print 26 <> 25
true
print not-equal? 25.3 25.5
true
print $2.29 <> $2.30
true
print not-equal? 11:11:10 11:11:11
true
```

6.7 same (то же)

Выражения:

```
value1 =? value2
same? value1 value2
```

Вернёт `true` если два слова относятся к одному и тому же значению. Например, если вы хотите увидеть, ссылаются ли два слова на один и тот же индекс в серии.

Работает со всеми типами данных.

```
reference-one: "abcdef"
reference-two: reference-one
print same? reference-one reference-two
true
reference-one: next reference-one
print same? reference-one reference-two
false
reference-two: next reference-two
print same? reference-one reference-two
true
reference-two: copy reference-one
print same? reference-one reference-two
false
```

6.8 strict-equal (строго равное)

Выражения:

```
value1 == value2

strict-equal? value1 value2
```

вернуть `true`, если первое и второе значения строго совпадают. Может использоваться как эквивалентная версия **`equal?`** (с учётом регистра) (**=**) для строк и различать целые и десятичные числа, когда их значения совпадают.

Работает со всеми типами данных.

```
print strict-equal? "abc" "ABC"
false
print equal? "abc" "ABC"
true
print strict-equal? "abc" "abc"
true
print strict-equal? 1 1.0
false
print equal? 1 1.0
true
print strict-equal? 1.0 1.0
true
```

6.9 strict-not-equal (строго-не-равный)

Выражения:

```
strict-not-equal? value1 value2
```

Возвращает `true`, если первое и второе значения строго не совпадают. Может использоваться как **`not-equal?`** (чувствительная к регистру) версия неравенства (**<>**) для строк и различать целые и десятичные числа, когда их значения совпадают.

Работает со всеми типами данных.

```
print strict-not-equal? "abc" "ABC"
true
print not-equal? "abc" "ABC"
false
print strict-not-equal? "abc" "abc"
false
print strict-not-equal? 1 1.0
true
print not-equal? 1 1.0
false
print strict-not-equal? 1.0 1.0
false
```

7. Логарифмические функции.

7.1 exp

Выражения:

```
exp value
```

Возвращает экспоненту E (натуральное число) в степени `value`.

7.2 log-10

Выражения:

```
log-10 value
```

Возвращает десятичный логарифм числа `value`.

7.3 log-2

Выражения:

```
log-2 value
```

возвращает логарифм значения по основанию 2 для `value`.

7.4 log-e

Выражения:

```
log-e value
```

Возвращает натуральный логарифм числа `value`.

7.5 power

Выражения:

```
value1 ** value2
power value1 value2
```

Возвращает результат возведения в степень `value2` числа `value1`.

7.6 square-root

Выражения:

```
square-root value
```

Возвращает квадратный корень числа `value`.

8. Тригонометрические функции

Тригонометрические функции имеют дело с градусами. Используйте уточнение **`/radians`** с любой из тригонометрических функций, чтобы работать и возвращать радианы.

8.1 arccosine

Выражения:

```
arccosine value
```

Возвращает значение арккосинуса числа `value`.

8.2 arcsine

Выражения:

```
arcsine value
```

Возвращает значение арксинуса числа `value`.

8.3 arctangent

Выражения:

```
arctangent value
```

Возвращает значение арктангенса числа `value`.

8.4 cosine

Выражения:

```
cosine value
```

Возвращает значение косинуса числа`value`.

8.5 sine

Выражения:

```
sine value
```

Возвращает значение синуса числа `value`.

8.6 tangent

Выражения:

```
tangent value
```

Возвращает значение тангенса `value`.

9. Логические функции

Логические функции могут выполняться над логическими значениями и некоторыми скалярными значениями, включая целые числа (**`integer`**), символы (**`char`**), кортежи (**`tuple`**) и биты (**`bitset`**). При работе с логическими значениями логические функции возвращают логические значения. При работе с другими типами значений логика работает с битами.

9.1 and (и)

Функция **`and`** сравнивает два логических значения и возвращает `true`(истину), если они оба верны:

```
print (1 < 2) and (2 < 3)
true
print (1 < 2) and (4 < 3)
false
```

При использовании целых чисел, функция **`and`** сравнивает биты одного числа с битами второго и возвращает число, в котором установлены в единицу только те биты, в которых была единица в обоих числах одновременно:

```
print 3 and 5
1
```

9.2 or (или)

Функция **`or`** сравнивает два логических значения и возвращает `true` если хоть одно из значений `true`:

```
print (1 < 2) or (2 < 3)
true
print (1 < 2) or (4 < 3)
true
print (3 < 2) or (4 < 3)
false
```

При использовании целых чисел **or** сравнивает биты обоих чисел и если хоть в одном из них бит в единице, то в результате бит так же устанавливается в 1:

```
print 3 or 5
7
```

9.3 xor

Функция **xor** сравнивает два логических значения и возвращает true (истину) тогда и только тогда, когда одно из значений истинно, а другое ложно.

```
print (1 < 2) xor (2 < 3)
false
print (1 < 2) xor (4 < 3)
true
print (3 < 2) xor (4 < 3)
false
```

При использовании с целыми числами **xor** сравнивает бит с битом и возвращает 1 тогда и только тогда, когда один бит равен 1, а другой 0 . В противном случае возвращается 0:

```
print 3 xor 5
6
```

9.4 complement

Функция **complement** возвращает логическое или побитовое дополнение значения. Он используется для целых чисел битовой маски и инвертирования битовых наборов.

```
print complement true
false
print complement 3
-4
```

9.5 not (нет)

Для логического значения **not** возвращает true (истина), если значение ложно, и false (ложь), если значение истинно. Он не выполняет числовые побитовые операции.

```
print not true
false
print not false
true
```

10. Ошибки

Математические ошибки сообщаются, когда выполняется недопустимая операция или когда происходит переполнение или потеря значимости. В математических операциях могут встречаться следующие ошибки.

10.1 Попытка деление на ноль

Была сделана попытка разделить число на 0.

```
1 / 0
** Math Error: Attempt to divide by zero
** Where: connect-to-link
** Near: 1 / 0
```

10.2 Математическое или числовое переполнение

Была сделана попытка обработать число, слишком большое для обработки в REBOL.

```
1E+300 + 1E+400
** Math Error: Math or number overflow
** Where: connect-to-link
** Near: 1 / 0
```

10.3 Требуется положительное число

Была предпринята попытка обработать отрицательное число математическим оператором, который принимает только положительные числа.

```
log-10 -1
** Math Error: Positive number required
** Where: connect-to-link
** Near: log-10 -1
```

10.4 Невозможно использовать оператор для значения данного типа данных.

Была сделана попытка обработать несовместимые типы данных. Тип данных второго аргумента в операции возвращается, как указано.

```
10:30 + 1.2.3
** Script Error: Cannot use add on time! value
** Where: connect-to-link
** Near: 10:30 + 1.2.3
```

Глава 12 - Файлы

Содержание:

1. Обзор
2. Имена и пути
    2.1 Имена файлов
    2.2 Строки пути
    2.3 Чувствительность к регистру
    2.4 Функции имени файла
3. Чтение файлов
    3.1 Чтение текстовых файлов
    3.2 Чтение двоичных файлов
    3.3 Чтение по сети
4. Запись файлов
    4.1 Запись текстовых файлов
    4.2 Запись двоичных файлов
    4.3 Запись файлов в сеть
5. Преобразование строк
6. Блоки строк
7. Информация о файлах и каталогах
    7.1 Проверка каталога
    7.2 Существование файла
    7.3 Размер файла
    7.4 Дата изменения файла
    7.5 Информация о каталоге
8. Каталоги
    8.1 Чтение каталога
    8.2 Создание каталога
    8.3 Переименование каталогов и файлов
    8.4 Удаление каталогов и файлов
    8.5 Текущий каталог
    8.6 Изменение текущего каталога
    8.7 Список текущего каталога

1. Обзор

Важным аспектом возможностей REBOL является его способность манипулировать файлами и каталогами. REBOL предоставляет широкий спектр функций, позволяющих выполнять операции от простого чтения файлов до прямого доступа к файлам и каталогам. Дополнительные сведения о прямом доступе к файлам и каталогам см. В главе «Порты».

2. Имена и пути

REBOL предоставляет стандартное машинно-независимое соглашение об именах файлов и путей.

2.1 Имена файлов

В сценариях имена файлов и пути записываются со знаком процента (%), за которым следует последовательность символов:

```
%examples.r
%big-image.jpg
%graphics/amiga.jpg
%/c/plug-in/video.r
%//sound/goldfinger.mp3
```

Знак процента необходим для предотвращения интерпретации имен файлов как слов в пределах языка.
Хотя это не очень хорошая практика, в имена файлов можно включать пробелы, заключив имя файла в двойные кавычки (" "). Двойные кавычки не позволяют интерпретировать имя файла как несколько слов:

```
%"this file.txt"
%"cool movie clip.mpg"
```

Стандартное Интернет-соглашение об использовании знака процента (%) и шестнадцатеричного кода также разрешено для кодировки символов. Когда это будет сделано, кавычки не требуются. Вышеупомянутые имена файлов также могут быть записаны как:

```
%this%20file.txt
%cool%20movie%20clip.mpg
```

Обратите внимание, что стандартным суффиксом файла для сценариев REBOL является «.r». В системах, где это соглашение противоречит другому типу файлов, вместо него может использоваться суффикс «.reb».

2.2 Строки пути

Пути к файлам записываются со знаком процента (%), за которым следует последовательность имен каталогов, каждое из которых разделено косой чертой (/).

```
%dir/file.txt
%/file.txt
%dir/
%/dir/
%/dir/subdir/
%../dir/file.txt
```

Стандартным символом для разделения каталогов является косая черта (/), а не обратная косая черта (\). Если обнаружены обратные косые черты, они преобразуются в прямые:

```
probe %\some\cool\movie.mpg
%/some/cool/movie.mpg
```

REBOL предоставляет стандартный, независимый от операционной системы метод определения путей к каталогам. Пути могут быть относительными к текущему каталогу или абсолютными от файловой структуры верхнего уровня операционной системы.

Пути к файлам, которые не начинаются с косой черты (/), являются относительными.

```
%docs/intro.txt
%docs/new/notes.txt
%"new mail/inbox.mbx"
```

Также поддерживается стандартное соглашение об использовании двойных точек (..) для обозначения родительского каталога или одиночной точки (.) Для обозначения текущего каталога. Например:

```
%.
%./
%./file.txt
%..
%../
%../script.r
%../../plans/schedule.r
```

В путях к файлам используется стандартное Интернет-соглашение о начале абсолютных путей с косой черты (/). Косая черта указывает на начало с верхнего уровня файловой системы. (Как правило, следует избегать абсолютных путей, чтобы гарантировать машинно-независимые сценарии.) Пример:

```
%/home/file.txt
```

будет ссылаться на том или раздел диска с именем `home`. Другие примеры:

```
%/ram/temp/test.r
%/cd0/scripts/test/files.r
```

Для обозначения тома `C`, который часто используется в Windows, используются следующие обозначения:

```
%/C/docs/file.txt
%"/c/program files/qualcomm/eudora mail/out.mbx"
```

Обратите внимание, что в приведенных выше строках дисковый том `C` не записывается как:

```
%c:/docs/file.txt
```

Приведенный выше пример не является машинно-независимым форматом и вызывает ошибку.
Если имя первого каталога отсутствует, а путь начинается с двойной косой черты (//), то путь к файлу указывается относительно текущего тома:

```
%//docs/notes
```

2.3 Чувствительность к регистру

В REBOL имена файлов по умолчанию ***не*** чувствительны к регистру. Однако, когда новые файлы создаются языком, они сохраняют тот регистр, в котором были введены:

```
write %Script-File.r file-data
```

В приведенном выше примере создается имя файла с буквами `S` и `F` в верхнем регистре.

Кроме того, когда имена файлов считываются из каталогов, регистр сохраняется:

```
print read %/home
```

Для систем с учетом регистра, таких как UNIX, REBOL находит наиболее близкое совпадение регистра с указанным файлом. Например, если сценарий просит прочитать `%test.r`, но находит только `%TEST.r`, будет прочитан файл `%TEST.r`. Такое поведение необходимо, чтобы разрешить машинно-независимые сценарии

2.4 Функции имени файла

Предусмотрены различные функции, которые помогут вам создавать имена и пути к файлам. Они перечислены ниже в разделе «Функции имени файла».

to-file	Преобразует строки и блоки в имя файла или путь к файлу.
split-path	Разбивает путь на часть каталога и имя файла.
clean-path	Возвращает абсолютный путь, эквивалентный любому заданному пути, содержащему двойную точку (..) или точку (.).
what-dir	Возвращает абсолютный путь к текущему каталогу.

3. Чтение файлов

Файлы читаются как последовательность текстовых символов или как двоичные байты. Источником файла является либо локальный файл в вашей системе, либо файл из сети.

3.1 Чтение текстовых файлов

Чтобы прочитать локальный текстовый файл, используйте функцию **`read`**:

```
text: read %file.txt
```

Функция **`read`** возвращает строку, содержащую весь текст файла. В приведенном выше примере переменный `text` относится к этой строке.

В строке, возвращаемой методом **`read`**, окончание строки преобразуются в символы новой строки `newline`, независимо от того, какой стиль завершения строки используется в вашей операционной системе. Это позволяет вам писать сценарии, которые ищут новую строку (`newline`), не заботясь о том, какой конкретный символ или символы составляют завершение строки.

```
next-line: next find text newline
```

Файл также можно читать как отдельные строки, которые хранятся в блоке

```
lines: read/lines %file.txt
```

Смотрите раздел конверсионной линии для получения дополнительной информации о новых строках (`newline`) и чтении строк.
Чтобы читать файл по частям, используйте функцию **`open`**, как описано в главе «Порты».
Чтобы просмотреть содержимое текстового файла, вы можете прочитать его с помощью функции **`read`** и распечатать с помощью функции **`print`**:

```
print read %service.txt
I wanted the gold, and I sought it,I scrabbled and mucked like
a slave.
```

3.2 Чтение двоичных файлов

Чтобы прочитать двоичный файл, такой как изображение, программа или звук, используйте **`read/binary`**:

```
data: read/binary %file.bin
```

Функция **`read/binary`** кода возвращает двоичный ряд, содержащий все содержимое файла. В приведенном выше примере переменные `data` относятся к двоичной серии. Никакого преобразования в файл не производится.
Чтобы читать двоичный файл по частям, используйте функцию **`open`**, как описано в главе «Порты».

3.3 Чтение по сети

Файлы можно читать из сети. Например, чтобы просмотреть текстовый файл из сети по протоколу HTTP:

```
print read http://www.rebol.com/test.txt
Hellotherenewuser!
```

Файл можно было записать локально со строкой:

```
write %test.txt read http:/www.rebol.com/test.txt
```

В процессе записи конец строки файла будет преобразован в то, что используется вашей операционной системой.
Чтобы прочитать и сохранить двоичный файл, например изображение, используйте следующую строку:

```
write %image.jpg
read/binary http:/www.rebol.com/image.jpg
```

Обратитесь к главе «Сетевые протоколы» для получения дополнительной информации и примеров доступа к файлам в сети.

4. Запись файлов

Вы можете записать файл как серию текстовых символов или как двоичные байты. Местоположение файла может быть либо локальным файлом в вашей системе, либо файлом в сети.

4.1 Запись текстовых файлов

Чтобы написать локальный текстовый файл, используйте следующую строку кода:

```
write %file.txt "sample text here"
```

Это запишет весь текст в файл.

Если файл содержит символы новой строки (`newline`), они будут преобразованы в символы, используемые вашей локальной файловой системой. Это позволяет работать с файлами в последовательном порядке, но записывать их с помощью конвенции, которая является стандартной для вашей файловой системы.

Например, следующая строка преобразует любой текстовый файл из однострочного формата завершения (UNIX, Macintosh, PC, Amiga) в формат, используемый вашей локальной системой:

```
write %newfile.txt read %file.txt
```

Вышеупомянутая строка считывает весь файл при преобразовании окончания строки в стандарт REBOL, а затем записывает файл, преобразовывая его в формат локальной операционной системы.

Чтобы добавить в конец файла, используйте уточнение **`/append`**:

```
write/append %file.txt "more text"
```

Файл также можно записать из отдельных строк, которые хранятся в блоке.

```
write/lines %file.txt lines
```

Чтобы записать файл по частям, используйте функцию **`open`**, как описано в главе «Порты».

4.2 Запись двоичных файлов

Чтобы записать двоичный файл, такой как изображение, программа, звук, используйте **`write/binary`**:

```
write/binary %file.bin data
```

Функция **`write/binary`** файла создает файл, если он не существует, или перезаписывает файл, если он уже существует. Никакого преобразования в файл не производится.
Чтобы записать двоичный файл по частям, используйте функцию **`open`**, как описано в главе «Порты».

4.3 Запись файлов в сеть

Файлы также можно записывать в сеть. Например, чтобы записать текстовый файл в сеть с помощью FTP, используйте:

```
write ftp://ftp.domain.com/file.txt "save this text"
```

Файл можно прочитать локально и записать в сеть с помощью такой строки, как:

```
write ftp://ftp.domain.com/file.txt read %file.txt
```

При этом окончание строки файла преобразуется в стандартный формат CRLF.
Чтобы записать двоичный файл, например изображение, в сеть, используйте следующие строки кода:

```
write/binary ftp://ftp.domain.com/file.txt/image.jpg
    read/binary %image.jpg
```

См. Главу «Сетевые протоколы» для получения дополнительной информации и примеров доступа к файлам из сетей.

5. Преобразование строк

Когда файл читается как текст, все символы конца строки преобразуются в символы новой строки (`newline`, "перевода строки"). Перевод строки (используемый в качестве ограничителя строки в системах Amiga, Linux и UNIX), возврат каретки (используемый в качестве ограничителя строки в Macintosh) и комбинация CR / LF (ПК и Интернет) преобразуются в эквивалентные символы новой строки.
Использование в сценариях стандартного терминатора строки позволяет им работать машинно-независимым образом. Например, чтобы найти и подсчитать все символы новой строки в текстовом файле:

```
text: read %file.txt
count: 0
while [spot: find text newline][
    count: count + 1
    text: next spot
]
```

Преобразование строк также полезно для чтения сетевых файлов:

```
text: read ftp://ftp.rebol.com/test.txt
```

Когда файл записывается, символ новой строки преобразуется в стандарт стиля завершения строки для используемой операционной системы. Например, символ новой строки преобразуется в CRLF на ПК, LF в UNIX или Amiga или CR в Macintosh. Сетевые файлы пишутся с помощью CRLF.

Следующая функция преобразует любой текстовый файл с любым стилем терминатора в файл, используемый локальной операционной системой:

```
convert-lines: func [file] [write file read file]
```

Файл читается, и все символы конца строки преобразуются, затем файл записывается, и символы новой строки преобразуются в стиль локальной операционной системы

Преобразование строк можно отключить, прочитав файл как двоичный. Например, следующая строка:

```
write/binary %newfile.txt read/binary %oldfile.txt
```

сохраняет символы окончания строки текстового файла.

6. Блоки линий

Доступ к текстовым файлам и управление ими можно выполнять как отдельные строки текста, а не как одну серию символов. Например, чтобы прочитать файл как блок строк:

```
lines: read/lines %service.txt
```

В приведенном выше примере возвращается блок, содержащий серию строк (по одной для каждой строки) без символов оканчания строки. Пустые строки представлены пустыми строками.

Чтобы напечатать определенную строку, вы используете следующий код:

```
print first lines
print last lines
print pick lines 100
print lines/500
```

Чтобы напечатать все строки файла, используйте следующую строку кода:

```
foreach line lines [print line]
I wanted the gold, and I sought it,
I scrabbled and mucked like a slave.
Was it famine or scurvy -- I fought it;
I hurled my youth into a grave.
I wanted the gold, and I got it --
Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
And somehow the gold isn't all.
```

Чтобы напечатать все строки, содержащие строку `gold`, используйте следующую строку кода:

```
foreach line lines [
   if find line "gold" [print line]
]
I wanted the gold, and I sought it,
I wanted the gold, and I got it --
And somehow the gold isn't all.
```

Вы можете записать текстовый файл в виде строк, используя функцию **`write`**:

```
write/lines %output.txt lines
```

Чтобы записать определенные строки из блока, используйте:

```
write/lines %output.txt [
    "line one"
    "line two"
    "line three"
]
```

Фактически, функции **read/lines** (чтения/строк) и **write/lines** (записи/строк) могут быть объединены для обработки файлов по одной строке за раз. Например, следующий код удаляет все комментарии из сценария REBOL:

```
script: read/lines %script.r
foreach line script [
    where: find line ";"
    if where [clear where]
]
write/lines %script1.r script
```

Образец сценария в приведенном выше примере предназначен только для демонстрационных целей. Помимо удаления комментариев, пример сценария также удаляет действительные точки с запятой в строках, заключенных в кавычки.

Файлы также можно читать как строки из сети:

```
data: read/lines http://www.rebol.com
print pick (read/lines ftp://ftp.rebol.com/test.txt) 3
new
```

Уточнение **/lines** может быть использовано с функцией **open**, чтобы прочитать строку в то время от консольного ввода. Дополнительную информацию см. В главе «Порты».
Кроме того, **/lines** можно использовать с **/append** для добавления строк из блока в файл.

7. Информация о файлах и каталогах

Существует ряд функций, которые предоставляют полезную информацию о файле, например, существует ли он, размер его файла в байтах, когда он был в последний раз изменен и является ли он каталогом.

7.1 Проверка каталога

Чтобы определить, совпадает ли имя файла с именем каталога, используйте команду функции **dir?**:

```
print dir? %file.txt
false
print dir? %.
true
```

Функция **dir?** также работает с некоторыми сетевыми протоколами:

```
print dir? ftp://www.rebol.com/pub/
true
```

7.2 Существование файла

Чтобы определить, существует ли файл, используйте команду функции b>exists?:

```
print exists? %file.txt
```

Чтобы определить, существует ли файл, прежде чем вы его прочитаете, используйте:

```
if exists? file [text: read file]
```

Чтобы избежать перезаписи файла, с помощью которого вы можете проверить его, используйте:

```
if not exists? file [write file data]
```

Функция **exists?** также работает с некоторыми сетевыми протоколами:

```
print exists? ftp://www.rebol.com/file.txt
```

7.3 Размер файла

Чтобы получить размер файла в байтах, используйте функцию **size?**:

```
print size? %file.txt
```

Функция **size?** также работает с некоторыми сетевыми протоколами:

```
print size? ftp://www.rebol.com/file.txt
```

7.4 Дата изменения файла

Чтобы получить дату последнего изменения файла, используйте функцию **modified?**:

```
print modified? %file.txt
30-Jun-2000/14:41:55-7:00
```

Не все операционные системы отслеживают дату создания файла, поэтому для того, чтобы сценарии REBOL оставались независимыми от операционной системы, доступна только дата последнего изменения.

Функция **modified?** также работает с некоторыми сетевыми протоколами:

```
print modified? ftp://www.rebol.com/file.txt
```

7.5 Справочная информация

Функция **info?** получает всю информацию о каталоге или файле одновременно. Информация возвращается в виде объекта:

```
probe info? %file.txt

make object! [
    size: 306
    date: 30-Jun-2000/14:41:55-7:00
    type: 'file
]
```

Чтобы распечатать информацию обо всех файлах в текущем каталоге, используйте:

```
foreach file read %. [
    info: info? file
    print [file info/size info/date info/type]
]

build-guide.r 22334 30-Jun-2000/14:24:43-7:00 file
code/ 11 11-Oct-1999/18:37:04-7:00 directory
data.r 41 30-Jun-2000/14:41:36-7:00 file
file.txt 306 30-Jun-2000/14:41:55-7:00 file
```

Функция `info?` также работает с некоторыми сетевыми протоколами:

```
probe info? ftp://www.rebol.com/file.txt
```

8. Директории

Есть несколько простых в использовании функций для чтения каталогов, управления подкаталогами, создания новых каталогов, переименования файлов и удаления файлов. Кроме того, есть стандартные функции для получения, изменения и вывода текущего каталога. Дополнительную информацию о прямом доступе к каталогам см. В главе «Порты».

8.1 Чтение каталога

Каталоги читаются так же, как файлы. Функция `read` возвращает блок имен файлов, а не текстовые или двоичные данные.

Чтобы прочитать все имена файлов из текущего каталога, используйте следующую строку кода:

```
read %.
```

В приведенном выше примере считывается весь каталог и возвращается блок имен файлов.

Чтобы распечатать имена всех файлов в каталоге `intro`, используйте следующую строку кода:

```
print read %intro/
CVS/ history.t intro.t overview.t quick.t
```

В возвращаемом блоке имена каталогов обозначаются косой чертой в конце. Чтобы напечатать каждое имя файла в отдельной строке, используйте:

```
foreach file read %intro/ [print file]
CVS/
history.t
intro.t
overview.t
quick.t
```

Вот простой способ распечатать только найденные каталоги:

```
foreach file read %intro/ [
    if #"/" = last file [print file]
]
CVS/
```

Если вы хотите прочитать каталог из сети, не забудьте включить косую черту в конце URL-адреса, чтобы указать протокоу, что вы ссылаетесь на каталог:

```
print read ftp://ftp.rebol.com/
```

8.2 Создание каталога

Функция **make-dir** создает новый каталог. Новое имя каталога может быть указанно как относительно текущего каталога, так и содержать абсолютный пути.

```
make-dir %new-dir
make-dir %local-dir/
make-dir %/work/docs/old-docs/
```

Завершающая косая черта не является обязательной для этой функции.

Внутренне вызовы функции **make-dir** это вызов функции **open** с уточнением **/new**. Строка:

```
close open/new %local-dir/
```

также создает новый каталог. Завершающая косая черта важна в этом примере, указывая на то, что создается каталог, а не файл.
Если вы используете функцию **make-dir** для создания каталога, который уже существует, произойдет ошибка. Ошибка может быть обнаружена с помощью функции **try**. Каталог можно заранее проверить с помощью функции **exists?**.

8.3 Переименование каталогов и файлов

Чтобы переименовать файл, используйте функцию **rename**:

```
rename %old-file %new-file
```

Старое имя файла может включать полный путь к файлу, но новое имя файла не должно включать путь. Это связано с тем, что функция **rename** не предназначена для перемещения файлов между каталогами (различные операционные системы не поддерживают эту функцию).

```
rename %../docs/intro.txt %conclusion.txt
```

Если старое имя файла каталога (обозначается слэш), то функция **rename** переименовывает каталог:

```
rename %../docs/ %manual/
```

Если файл не может быть переименован, произойдет ошибка. Ошибка может быть обнаружена с помощью функции **try**.

8.4 Удаление каталогов и файлов

Файлы можно удалить с помощью функции **delete**:

```
delete %file
```

Удаляемый файл может включать путь:

```
delete %source/docs/file.txt
```

Также можно удалить блок файлов в том же каталоге:

```
delete [%file1 %file2 %file3]
```

Группа файлов может быть удалена с помощью подстановочного знака и уточнения **/any**:

```
delete/any %file*
delete/any %secret.?
```

Подстановочный знак звездочка (*) соответствует всем символам, а подстановочный знак вопросительного знака (?) соответствует одному символу.

Чтобы удалить каталог, укажите в конце косую черту

```
delete %dir/
delete %../docs/old/
```

Если файл не может быть удален, произойдет ошибка. Ошибка может быть обнаружена с помощью функции **try**.

8.5 Текущий каталог

Используйте функцию **what-dir**, чтобы определить текущий каталог:

```
print what-dir
/work/REBOL/
```

Функция **what-dir** ссылается на путь к каталогу текущего скрипта, который находится в `system/script/path`.

8.6 Изменение текущего каталога

Изменение текущего каталога

```
change-dir %new-path/to-dir/
```

Если косая черта в конце не указана, функция добавляет ее.

8.7 Список текущего каталога

Чтобы вывести список содержимого текущего каталога, используйте:

```
list-dir
```

Количество столбцов, используемых для отображения каталога, зависит от размера окна консоли и максимальной длины имени файла.

Глава 13 - Сетевые протоколы

Содержание:

1. Обзор
2. Основы работы с сетью в REBOL
  2.1 Режимы работы
  2.2 Указание сетевых ресурсов
  2.3 Схемы, обработчики и протоколы
  2.4 Обработчики мониторинга
3. Первоначальная настройка
  3.1 Основные параметры сети
  3.2 Параметры прокси
  3.3 Другие параметры
  3.4 Доступ к настройкам
4. DNS - Domain Name Service
5. Протокол Whois
6. Протокол Finger
7. Daytime - Протокол сетевого времени
8. HTTP - Протокол передачи гипертекста
  8.1 Чтение веб-страницы
  8.2 Сценарии на веб-сайтах
  8.3 Загрузка страниц разметки
  8.4 Другие функции
  8.5 Работа как браузер
  8.6 Отправка запросов CGI
9. SMTP - простой протокол передачи почты
  9.1 Отправка электронной почты
  9.2 Несколько получателей
  9.3 Массовая рассылка
  9.4 Тема и заголовки
  9.5 Отладка ваших сценариев
10. POP - протокол почтового отделения
  10.1 Чтение электронной почты
  10.2 Удаление электронной почты
  10.3 Обработка заголовков электронной почты
11. FTP - протокол передачи файлов
  11.1 Использование FTP
  11.2 URL-адреса FTP
  11.3 Передача текстовых файлов
  11.4 Передача двоичных файлов
  11.5 Добавление к файлам
  11.6 Чтение каталогов
  11.7 Информация о файлах
  11.8 Создание каталогов
  11.9 Удаление файлов
  11.10 Удаление файлов
  11.11 О паролях
  11.12 Передача больших файлов
12. NNTP - сетевой протокол передачи новостей
  12.1 Чтение списка групп новостей
  12.2 Чтение всех сообщений
  12.3 Чтение отдельных сообщений
  12.4 Обработка заголовков новостей
  12.5 Отправка новостного сообщения

13. CGI - общий интерфейс шлюза
- 13.1 Настройка сервера CGI
- 13.2 Сценарии CGI
- 13.3 Создание содержимого HTML
- 13.4 CGI среда
- 13.5 Запросы CGI
- 13.6 Обработка HTML-форм

14. TCP - протокол управления передачей
- 14.1 Создание клиентов
- 14.2 Создание серверов
- 14.3 Крошечный сервер
- 14.4 Тестирование кода TCP

15. UDP - протокол дейтаграмм пользователя

1. Обзор

REBOL включает несколько встроенных основных протоколов интернет-сервисов. Эти протоколы легко использовать в ваших сценариях; они не требуют дополнительных библиотек или включаемых файлов, и многие полезные операции могут быть выполнены с помощью только одной строчки исходного кода.

Поддерживаются протоколы, перечисленные в Сетевые протоколы:

DNS	Domain Name Service: Служба доменных имен: переводит имена компьютеров в адреса и адреса в имена.
Finger	Получает информацию о пользователе из его профиля.
Whois	Получает информацию о регистрации домена.
Daytime	Сетевой протокол времени. Получает время с сервера.
HTTP	Hypertext Transfer Protocol. Протокол передачи гипертекста. Используется для Интернета.
SMTP	Simple Mail Transfer Protocol. Простой протокол передачи почты. Используется для отправки электронной почты.
POP	Post Office Protocol. Почтовый протокол. Используется для получения электронной почты.
FTP	File Transfer Protocol. Протокол передачи файлов. Обменивается файлами с сервером.
NNTP	Network News Transfer Protocol. Протокол передачи сетевых новостей. Публикует или читает новости Usenet.
TCP	Transmission Control Protocol. Протокол управления передачей. Базовый интернет-протокол.
UDP	User Datagram Protocol. Протокол пользовательских датаграмм. Пакетный протокол.

Кроме того, вы можете создавать обработчики для других Интернет-протоколов или создавать свои собственные протоколы.

2. Основы работы с сетью REBOL

2.1 Режимы работы

Существует два основных режима работы сети: атомарный и портовый.

`Атомарные` сетевые операции - это операции, которые выполняются с помощью одной функции. Например, вы можете прочитать всю веб-страницу с помощью одного вызова функции чтения. Нет необходимости отдельно открывать соединение или настраивать чтение. Все это делается автоматически как часть чтения. Например, вы можете ввести:

```
print read http://www.rebol.com
```

Хост найден и открыт, его веб-страница перенесена, а соединение закрыто.

На основе `портов` режим работы один, который использует более традиционный программный подход. Он включает в себя открытие порта и выполнение с ним различных последовательных операций. Например, если вы хотите читать свою электронную почту с POP-сервера по одному сообщению за раз, вы должны использовать этот метод. Вот пример, который читает и отображает всю вашу электронную почту:

```
pop: open pop://user:pass@mail.example.com
forall pop [print first pop]
close pop
```

Атомарный метод работы проще, но он более ограничен. Метод на основе портов позволяет выполнять больше типов операций, но также требует более глубокого понимания работы в сети.

2.2 Указание сетевых ресурсов

REBOL предлагает два подхода к определению сетевых ресурсов: URL-адреса и спецификации портов. `Унифицированные` указатели ресурсов (URL) используются в Интернете для идентификации сетевого ресурса, такого как веб-страница, FTP-сайт, адрес электронной почты, файл или другой ресурс или услуга. URL-адреса являются неотъемлемой частью работы REBOL, и они могут быть выражены непосредственно на языке.
Стандартное обозначение URL-адресов состоит из `схемы` (`scheme`), за которой следует спецификация:

```
scheme:specification
```

Схема часто представляет собой имя протокола, например HTTP, FTP, SMTP и POP; однако это не является обязательным требованием. Схема может быть любым именем, которое идентифицирует метод, используемый для доступа к ресурсу.

Формат спецификации схемы зависит от схемы; однако в большинстве схем используется общий формат для определения сетевых хостов, имен пользователей, паролей, номеров портов и путей к файлам. Вот несколько часто используемых форматов:

```
scheme://host

scheme://host:port

scheme://user@host

scheme://user:pass@host

scheme://user:pass@host:port

scheme://host/path

scheme://host:port/path

scheme://user@host/path

scheme://user:pass@host/path

scheme://user:pass@host:port/path
```

В Спецификации сетевых ресурсов перечислены поля, используемые в вышеуказанных форматах.

scheme Имя, используемое для идентификации типа ресурса, часто такое же, как протокол. Например, HTTP, FTP и POP.

host Сетевое имя или адрес машины. Например, www.rebol.com, cnn.com, 192.168.0.1.

port Номер порта на хост-машине для используемой схемы. Обычно для этого есть значение по умолчанию, поэтому в большинстве случаев это не требуется. Примеры: 21, 23, 80, 8000.

user Имя пользователя для доступа к ресурсу.

pass Пароль для проверки имени пользователя.

path Путь к файлу или другой способ обращения к ресурсу. Это зависит от схемы (scheme). Некоторые схемы включают шаблоны и аргументы сценария (например, CGI).

Другой способ идентифицировать ресурс - использовать спецификацию `port` (порт) REBOL . Фактически, когда используется URL-адрес, он автоматически преобразуется в спецификацию порта. Спецификация порта может принимать намного больше аргументов, чем URL-адрес, но для ее выражения требуется несколько строк.

Спецификация порта записывается как определение объектного блока, которое предоставляет все параметры, необходимые для доступа к сетевому ресурсу. Например, URL-адрес для доступа к веб-сайту:

```
read http://www.rebol.com/developer.html
```

но это также можно записать как:

```
read [
    scheme: 'HTTP
    host: "www.rebol.com"
    target: %/developer.html
]
```

URL-адрес для чтения FTP может быть:

```
read ftp://bill:vbs@ftp.example.com:8000/file.txt
```

но это также можно записать как:

```
read [
    scheme: 'FTP
    host: "ftp.example.com"
    port-id: 8000
    target: %/file.txt
    user: "bill"
    pass: "vbs"
]
```

Кроме того, есть много других полей порта, которые можно указать, например тайм-аут, тип доступа и безопасность.

2.3 Схемы, обработчики и протоколы

Сеть REBOL работает с использованием `схем` (schemes) для идентификации `обработчиков` (handlers), которые обмениваются данными с `протоколами` (protocols).

В REBOL `схема` (schemes) используется для идентификации метода доступа к ресурсу. Этот метод использует объект кода, который называется `обработчиком` (handlers). Каждая из схем URL-адресов, поддерживаемых REBOL (например, HTTP, FTP), имеет обработчик. Список схем можно получить с помощью:

```
probe next first system/schemes
[default Finger Whois Daytime SMTP POP HTTP FTP NNTP]
```

Кроме того, есть имена схем более низкого уровня, которые здесь не показаны. Например, схемы TCP и UDP используются для прямой связи более низкого уровня.

В этот список можно добавлять новые схемы. Например, вы можете определить свою собственную схему, называемую FTP2, которая предоставляет специальные функции для доступа по FTP, такие как автоматическое указание вашего имени пользователя и пароля, поэтому его не нужно включать в каждый URL-адрес FTP.

Большинство обработчиков используются для обеспечения интерфейса для сетевого протокола. `Протокол` (protocol) используется для обмена данными между различными устройствами, в том числе клиентов и серверов.

Несмотря на то, что каждый протокол очень отличается по способу взаимодействия, у него есть некоторые общие черты с другими протоколами. Например, для большинства протоколов требуется, чтобы сетевое соединение было открыто (opened), прочитано (read), записано (written) и закрыто (closed). Эти общие операции выполняются обработчиком по умолчанию в REBOL. Этот обработчик упрощает реализацию таких протоколов, как finger, whois и daytime.

Обработчики схем записываются как объекты. Обработчик по умолчанию служит корневым объектом для всех остальных обработчиков. Когда обработчику требуется определенное поле, например значение тайм-аута для чтения данных, если значение не определено в конкретном обработчике, оно будет предоставлено обработчиком по умолчанию. Следовательно, обработчики накладывают друг на друга свои поля и значения. Вы также можете создать обработчики, которые используют другие обработчики для значений по умолчанию. Например, вы можете создать обработчик FTP2, который сначала ищет отсутствующие поля в обработчике FTP, а затем в обработчике по умолчанию.

Когда порт используется для доступа к сетевым ресурсам, он связан с определенным обработчиком. Обработчик и порт вместе образуют единицу, которая используется для предоставления данных, кода и информации о состоянии для обработки всех протоколов.

Исходный код обработчиков можно получить из объекта system/scheme (система/схема). Это может быть полезно, если вы хотите изменить поведение обработчика или создать свой собственный обработчик. Например, чтобы просмотреть код обработчика whois, введите:

```
probe get in system/schemes 'whois
```

Обратите внимание, что вы видите составную часть обработчика по умолчанию и обработчика whois. Фактический исходный код, который используется для создания обработчика whois, состоит всего из нескольких строк:

```
make Root-Protocol [
    open-check:  [[any [port/user ""]] none]
    net-utils/net-install Whois make self [] 43
]
```

2.4 Обработчики мониторинга

В целях отладки вы можете отслеживать действия любого обработчика. У каждого обработчика есть свои собственные выходные данные отладки, чтобы указать, какие операции выполняются. Чтобы включить отладку сети, включите трассировку сети с помощью строки:

```
trace/net on
```

Чтобы отключить отладку сети, используйте:

```
trace/net off
```

Вот пример:

```
read pop://carl:poof@zen.example.com
URL Parse: carl poof zen.example.com none none none
Net-log: ["Opening tcp for" POP]
connecting to: zen.example.com
Net-log: [none "+OK"]
Net-log: {+OK QPOP (version 2.53) at zen.example.com starting.}
Net-log: [["USER" port/user] "+OK"]
Net-log: "+OK Password required for carl."
Net-log: [["PASS" port/pass] "+OK"]
** User Error: Server error: tcp -ERR Password supplied for "carl"
is incorrect.
** Where: read pop://carl:poof@zen.example.com
```

3. Начальная настройка

Сеть REBOL встроена. Для создания сценариев, использующих сетевые протоколы, вам не нужны специальные включаемые файлы или библиотеки. Единственное требование - предоставить основную информацию, необходимую для подключения протоколов к серверам или через брандмауэры и прокси. Например, для отправки электронной почты протоколу SMTP необходимо имя SMTP-сервера и адрес электронной почты для ответа.

3.1 Основные настройки сети

Когда вы запускаете REBOL в первый раз, вам будет предложено ввести необходимые сетевые настройки, которые хранятся в файле `user.r`. REBOL использует этот файл для загрузки необходимых сетевых настроек при каждом запуске. Если `user.r` не создан и REBOL не может найти существующий файл в своих путях, никакие настройки не загружаются. См. Главу «Эксплуатация» для получения дополнительной информации.

Чтобы изменить настройки сети, введите в командной строке **set-user**. Это запускает тот же сценарий конфигурации сети, который запускался при первом запуске REBOL. Этот скрипт загружается из файла `user.r`. Если этот файл не может быть найден или если вы хотите отредактировать настройку напрямую, вы можете использовать текстовый редактор для `user.r`.

В файле `user.r` настройки сети находятся в блоке, который следует за функцией **set-net**. Как минимум, блок должен содержать два элемента:

- Ваш адрес электронной почты для использования в полях отправителя и ответа в сообщениях электронной почты, а также для анонимного входа по FTP.
- Ваш сервер по умолчанию; это также ваш основной почтовый сервер.

Кроме того, вы можете указать еще несколько пунктов:

- Другой сервер входящей электронной почты (для POP)
- Прокси-сервер (для подключения к сети)
- Номер порта прокси
- Тип прокси (см. Настройки прокси ниже).

Вы также можете добавить строки после функции **set-net** для настройки других значений протокола. Например, вы можете установить значения тайм-аута для протоколов, установить пассивный режим FTP, установить идентификатор HTTP-агента пользователя, настроить отдельные прокси для разных протоколов и многое другое.

Пример **set-net**:

```
set-net [user@domain.dom mail.server.dom]
```

В первом поле указана ваша электронная почта с адреса, а во втором поле указан ваш сервер по умолчанию (обратите внимание, что здесь кавычки не требуются). Для большинства сетей этого достаточно, и никаких других настроек не требуется (если вам не нужен прокси). Также ваш сервер по умолчанию используется всякий раз, когда конкретный сервер не предоставляется.

Кроме того, если вы используете POP-сервер (для входящей электронной почты), отличный от вашего SMTP-сервера (для исходящей электронной почты), вы также можете указать это:

```
set-net [
    user@domain.dom
    mail.server.dom
    pop.server.dom
]
```

Однако, если у вас одинаковые SMTP и POP-серверы, в этом нет необходимости.

3.2 Настройки прокси

Если вы используете прокси или брандмауэр, вы можете предоставить функцию set-net с настройками вашего прокси. Это может включать имя или адрес прокси-сервера, номер порта прокси для доступа к серверу и необязательный тип прокси.

Например:

```
set-net [
    email@addr
    mail.example.com
    pop.example.com
    proxy.example.com
    1080
    socks
]
```

В этом примере будет использоваться прокси-сервер с именем `proxy.example.com` на его TCP-порту 1080 с методом прокси-сервера `socks`. Чтобы использовать прокси-сервер `socks4`, используйте слово `socks4`, а не `socks`. Чтобы использовать общий сервер CERN, используйте слово `generic`.

Вы также можете настроить прокси на разные машины для разных схем (протоколов). Каждый протокол имеет свой собственный прокси-объект, где вы можете установить значения прокси только для этой схемы. Вот пример настройки прокси для FTP:

```
system/schemes/ftp/proxy/host: "proxy2.example.com"

system/schemes/ftp/proxy/port-id: 1080

system/schemes/ftp/proxy/type: 'socks
```

В этом случае только FTP использует специальный прокси-сервер. Обратите внимание, что имя компьютера должно быть строкой, а тип прокси - буквальным словом.
Вот еще два примера. В первом примере прокси-сервер для HTTP устанавливается как общий (CERN) метод прокси-сервера:

```
system/schemes/http/proxy/host: "wp.example.com"

system/schemes/http/proxy/port-id: 8080

system/schemes/http/proxy/type: 'generic
```

В приведенном выше примере все HTTP-запросы проходят через общий прокси-сервер на `wp.example.com` с использованием TCP-порта 8080.
Если вы хотите отключить настройки прокси для определенной схемы, вы можете установить для полей прокси значение false.

```
system/schemes/smtp/proxy/host: false

system/schemes/smtp/proxy/port-id: false

system/schemes/smtp/proxy/type: false
```

В приведенном выше примере вся исходящая электронная почта не проходит через прокси. Значение `false` предотвращает использование даже прокси по умолчанию. Если вы установите для этих полей значение `none`, то будет использоваться прокси по умолчанию, если он настроен.

Если вы хотите обойти настройки прокси-сервера для определенных машин, например, в вашей локальной сети, вы можете предоставить список обхода. Вот список обхода прокси по умолчанию:

```
system/schemes/default/proxy/bypass:
    ["host.example.net" "*.example.com"]
```

Обратите внимание, что для сопоставления с образцом можно использовать символы звездочки (*) и вопросительного знака (?). Звездочка (*), используемая в приведенном выше примере, позволяет обойти все машины, заканчивающиеся на `example.com`.

Чтобы установить список обхода только для схемы HTTP, введите:

```
system/schemes/http/proxy/bypass:
    ["host.example.net" "*.example.com"]
```

3.3 Другие настройки

Помимо настроек прокси, вы можете установить значения тайм-аута сети для всех схем (по умолчанию) или для определенных схем. Например, чтобы увеличить время ожидания для всех схем, вы можете написать:

```
system/schemes/default/timeout: 0:05
```

Это устанавливает тайм-аут сети на 5 минут.

Если вы хотите увеличить тайм-аут только для SMTP, вы должны написать:

```
system/schemes/smtp/timeout: 0:10
```

В некоторых схемах (schemes) есть настраиваемые поля. Например, схема FTP позволяет установить пассивный режим для всех передач:

```
system/schemes/ftp/passive: on
```

Пассивный режим FTP полезен, потому что FTP-серверы, которые установлены в пассивный режим, не пытаются подключиться обратно через ваш брандмауэр.

При выполнении HTTP-доступа к веб-сайтам вы можете захотеть использовать другое поле user-agent в HTTP-запросе, чтобы получить лучшие результаты на нескольких сайтах, которые определяют тип браузера:

```
system/schemes/http/user-agent: "Mozilla/4.0"
```

3.4 Доступ к настройкам

Каждый раз, когда REBOL запускается, он читает файл `user.r`, чтобы найти сетевые настройки. Эти настройки выполняются с помощью функции **`set-net`**. Скрипты имеют доступ к этим настройкам через объект `system/schemes`.

```
system/user/email ; used for email from and reply

system/schemes/default/host - your primary server

system/schemes/pop/host - your POP server

system/schemes/default/proxy/host - proxy server

system/schemes/default/proxy/port-id - proxy port

system/schemes/default/proxy/type - proxy type
```

Ниже приведена функция, которая возвращает блок, содержащий настройки сети, в том же порядке, в котором их принимает **set-net**:\\

```
get-net: func [][
    reduce [
        system/user/email
        system/schemes/default/host
        system/schemes/pop/host
        system/schemes/default/proxy/host
        system/schemes/default/proxy/port-id
        system/schemes/default/proxy/type
    ]
]

probe get-net
```

4. DNS - служба доменных имен (Domain Name Service)

DNS - это сетевая служба, которая переводит доменные имена в связанные с ними IP-адреса. Кроме того, вы можете использовать DNS для поиска машины и доменного имени по IP-адресу.

Протокол DNS можно использовать тремя способами: вы можете найти первичный IP-адрес в имени машины, вы можете найти в доменном имени IP-адрес и вы можете найти имя и IP-адрес вашей локальной системы.

Чтобы найти основной IP-адрес определенного компьютера в определенном домене, введите:

```
print read dns://www.rebol.com
207.69.132.8
```

Вы также можете получить доменное имя, связанное с определенным IP-адресом:

```
print read dns://207.69.132.8
rebol.com
```

Обратите внимание, что этот обратный поиск DNS нередко возвращает значение «none». Есть машины, у которых нет имен хостов.

```
print read dns://11.22.33.44
none
```

Чтобы узнать имя хоста вашей системы, прочтите пустой URL-адрес DNS в форме:

```
print read dns://
crackerjack
```

Возвращаемые здесь данные зависят от типа машины. Это может быть неполное имя хоста, как показано выше, но также может быть полное имя хоста, `crackerjack`.example.com. Это зависит от операционной системы и конфигурации сети в операционной системе.

Вот пример, который ищет и распечатывает IP-адреса для ряда веб-сайтов:

```
domains: [
    www.rebol.com
    www.rebol.org
    www.mochinet.com
    www.sirius.com
]

foreach domain domains [
    print ["address for" domain "is:"
        read join dns:// domain]
]
address for www.rebol.com is: 207.69.132.8
address for www.rebol.org is: 207.66.107.61
address for www.mochinet.com is: 216.127.92.70
address for www.sirius.com is: 205.134.224.1
```

5. Протокол Whois

Протокол whois получает информацию о доменных именах из центрального реестра. Сервис whois предоставляется организациями, работающими в Интернете. Whois часто используется для получения регистрационной информации о домене или сервере в Интернете. Он может сказать вам, кто владеет доменом, как можно связаться с их техническим контактом, а также другую информацию.

Для получения информации используйте функцию **read** (чтение) с URL-адресом whois. Этот URL-адрес должен содержать имя домена и имя сервера Whois, разделенные знаком (@). Например, чтобы получить информацию о `example.com` из реестра Internic:

```
print read whois://example.com@rs.internic.net
connecting to: rs.internic.net
Whois Server Version 1.1
Domain names in the .com, .net, and .org domains can now be
registered with many different competing registrars. Go to
http://www.internic.net for detailed information.
Domain Name: EXAMPLE.COM
Registrar: NETWORK SOLUTIONS, INC.
Whois Server: whois.networksolutions.com
Referral URL: www.networksolutions.com
Name Server: NS.ISI.EDU
Name Server: VENERA.ISI.EDU
Updated Date: 17-aug-1999
>>> Last update of whois database: Sun, 16 Jul 00 03:16:34 EDT <<<
The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains
and Registrars.
```

Приведенный выше код является только примером. Детали возвращаемой информации и серверов, поддерживающих whois, со временем меняются.

Если вместо имени домена вы укажете слово, будут возвращены все записи, соответствующие этому слову:

```
print read whois://example@rs.internic.net
connecting to: rs.internic.net
Whois Server Version 1.1
Domain names in the .com, .net, and .org domains can now be
registered with many different competing registrars. Go to
http://www.internic.net for detailed information.
EXAMPLE.512BIT.ORG
EXAMPLE.ORG
EXAMPLE.NET
EXAMPLE.EDU
EXAMPLE.COM
To single out one record, look it up with "xxx", where xxx is one
of the of the records displayed above. If the records are the same, look them
up with "=xxx" to receive a full display for each record.
>>> Last update of whois database: Sun, 16 Jul 00 03:16:34 EDT <<<
The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains
and Registrars.
```

Протокол whois не принимает URL-адреса, такие как `www.example.com`, если только URL-адрес не является частью названия компании регистранта.

6. Протокол Finger

Протокол finger извлекает информацию о пользователе, хранящуюся в файле журнала пользователя.

Чтобы запросить информацию о пользователе с сервера, он должен работать по протоколу finger. Информация запрашивается путем чтения пальца (finger) URL-адреса, который содержит имя пользователя и имя домена в формате стиля электронной почты:

```
print read finger://username@example.com
```

В приведенном выше примере извлекается информация о пользователе по адресу `username@example.com`. Возвращаемая информация зависит от информации, предоставленной пользователем, и настроек сервера finger. Кроме того, детали возвращаемой информации зависят от каждого сервера; приведенные ниже примеры описывают только типичные серверы. Многие серверы могут вести себя нестандартно на своих цифровых портах.

Например, может быть возвращена следующая информация:

```
Login: username
Name: Firstname Lastname
Directory: /home/user
Shell: /usr/local/bin/tcsh
Office: City, State +1 555 555 5555
Last login Wed Jul 28 01:10 (PDT) on ttyp0 from some.example.com
No Mail.
No Plan.
```

Обратите внимание, что finger сообщает, когда пользователь в последний раз входил в систему с машины и есть ли у пользователя ожидающая почта. Если пользователь читает электронную почту из

этой учетной записи, finger иногда сообщает, когда почта была получена и когда пользователь в последний раз получал ее:

```
New mail received Sun Sep 26 11:39 1999 (PDT)
Unread since Tue Sep 21 04:45 1999 (PDT)
```

Сервер finger может также сообщать о содержимом файла `plan` (план) и файла `project` (проект), если они существуют. Пользователи могут включать любую информацию в файл плана или проекта.

Также возможно получить информацию о пользователях, используя их настоящее имя или фамилию. Некоторые серверы finger требуют, чтобы вы использовали заглавные буквы в именах точно так, как они отображаются в файле входа или в файле, используемом онлайн-сервером finger, для получения информации о пользователе. Другие finger-серверы более либеральны в отношении капитализации. Сервер finger будет отвечать на запросы настоящего имени, возвращая все списки, соответствующие критериям запроса. Например, если на хосте несколько пользователей с именем `zaphod`, то ввод запроса

```
print read finger://Zaphod@main.example.com
```

будет извлекать всех таких пользователей, чье имя или фамилия `Zaphod`.

Некоторые серверы finger возвращают список пользователей, если имя пользователя опущено. Например,

```
print read finger://main.example.com
```

получает список всех пользователей, вошедших в систему, если это позволяет служба finger, установленная на хост-машине.

Некоторые хост-машины ограничивают сервисы finger по соображениям безопасности. Они могут потребовать действительное имя пользователя и вернуть только информацию об этом пользователе. Если вы укажете такой сервер без предоставления информации о пользователе, сервер сообщит, что ему требуется конкретная информация о пользователе.

Если система не поддерживает протокол finger , то REBOL вернёт ошибку:

```
print read finger://host.dom
connecting to: host.dom
Access Error: Cannot connect to host.dom.
Where: print read finger://host.dom
```

7. Daytime - протокол сетевого времени.

Дневной протокол извлекает текущий день и время. Чтобы подключиться к дневному серверу, используйте **`read`** (чтение) с URL. URL-адрес содержит имя сервера, с которого следует читать дату:

```
print read daytime://everest.cclabs.missouri.edu
Fri Jun 30 16:40:46 2000
```

Формат информации, возвращаемой серверами, может различаться в зависимости от сервера. Обратите внимание, что часовой пояс может отсутствовать.

Если выбранный вами сервер не поддерживает дневное время, REBOL возвращает ошибку:

```
print read daytime://www.example.com
connecting to: www.example.com
** Access Error: Cannot connect to www.example.com.
** Where: print read daytime://www.example.com
```

8. HTTP - протокол передачи гипертекста (Hyper Text Transfer Protocol)

Всемирная паутина управляется двумя фундаментальными технологиями: HTTP и HTML. HTTP - это протокол передачи гипертекста, который контролирует, как веб-серверы и веб-браузеры взаимодействуют друг с другом. HTML - это язык гипертекстовой разметки, который определяет структуру и содержимое веб-страницы.

Чтобы получить веб-страницу, браузер отправляет запрос на веб-сервер по протоколу HTTP. При получении запроса сервер интерпретирует его, иногда используя сценарий CGI (см. CGI - Общий интерфейс шлюза), и отправляет данные обратно. Эти данные могут быть чем угодно, включая HTML, текст, изображения, программы и звук.

8.1 Чтение веб-страницы

Чтобы прочитать веб-страницу, используйте функцию **read** (чтение) с URL-адресом HTTP. Например:

```
page: read http://www.rebol.com
```

Это вернет веб-страницу www.rebol.com. Обратите внимание, что при чтении возвращается строка, содержащая HTML-код страницы. Никакая графика или другая информация не загружаются. Для этого вам нужно будет предоставить дополнительные чтения. Страница может отображаться как HTML-код с помощью **print**, ее можно записать в файл с помощью **write** или отправить по электронной почте с помощью **send**.

```
print page

write %index.html page

send zaphod@example.com page
```

Страницу можно обрабатывать различными способами с помощью различных функций REBOL, таких как **parse** (синтаксический анализ), **find** (поиск) и **load** (загрузка).

Например, чтобы найти на веб-странице все вхождения слова REBOL, вы можете написать:

```
parse read http://www.rebol.com [
    any [to "REBOL" copy line to newline (print line)]
]
```

8.2 Скрипты на веб-сайтах

Веб-сервер может предоставлять больше, чем просто сценарии HTML. Веб-серверы также весьма полезны для предоставления сценариев REBOL.

Вы можете загружать сценарии REBOL прямо с веб-сервера с помощью `load`:

```
data: load http://www.rebol.com/data.r
```

Вы также можете оценивать скрипты прямо с сервера с помощью `do`:

```
data: do http://www.rebol.com/code.r
```

Предупреждение

Делайте это осторожно. Оценка произвольных сценариев на открытых серверах Интернета вызывает проблемы. Оценивайте сценарий только в том случае, если вы полностью доверяете его источнику, полностью проверили его источник или установили максимальные параметры безопасности REBOL.

Кроме того, веб-страницы, содержащие HTML, могут содержать встроенные сценарии REBOL, и их можно запускать с помощью:

```
data: do http://www.rebol.com/example.html
```

Чтобы определить, существует ли сценарий на странице перед его оценкой, используйте функцию `script?` (сценарий?).

```
if page: script? http://www.rebol.com [do page]
```

функция `script?` читает страницу с веб-сайта и возвращает страницу в позиции заголовка REBOL.

8.3 Загрузка страниц разметки

Страницы HTML и XML можно быстро преобразовать в блок REBOL с помощью функции `load/markup` (загрузки/разметки). Эта функция возвращает блок, состоящий из всех тегов и строк, найденных на странице. Все интервалы и разрывы строк остаются без изменений.
Чтобы отфильтровать все теги веб-страницы и просто распечатать её текст, введите:

```
tag-text: load/markup http://www.rebol.com
text: make string! 2000

foreach item tag-text [
    if string? item [append text item]
]
print text
```

Затем вы можете искать в этом тексте строковые шаблоны. Он будет содержать все пробелы и разрывы строк исходного HTML-файла.

Вот еще один пример, который проверяет все ссылки, найденные на веб-странице, чтобы убедиться, что страницы, на которые они ссылаются, существуют:

```
REBOL []

page: http://www.rebol.com/developer.html
set [path target] split-path page
system/options/quiet: true  ; turn off connection msgs
tag-text: load/markup page
links: make block! 100

foreach tag tag-text [  ; find all anchor href tags
    if tag? tag [
        if parse tag [
            "A" thru "HREF="
            [{"} copy link to {"} | copy link to ">"]
            to end
        ][
            append links link
        ]
    ]
]

print links

foreach link unique links [  ; try each link
    if all [
        link/1 <> #"#"
        any [flag: not find link ":"
             find/match link "http:"]
    ][
        link: either flag [path/:link][to-url link]
        prin [link "... "]
        print either error? try [read link]
            ["failed"]["OK"]
    ]
]
```

8.4 Другие функции

Чтобы проверить, существует ли веб-страница, используйте команду **exists?**, которая возвращает истину (`true`), если страница существует.

```
if exists? http://www.rebol.com [
    print "page still there"
]
```

Примечание. Во многих случаях обычно быстрее просто прочитать страницу, чем сначала проверить, существует ли она. В противном случае сценарий должен дважды связаться с сервером, что может занять много времени.

Чтобы запросить дату последнего изменения веб-страницы, используйте функция **modified?** (измененный?):

```
print modified? http://www.rebol.com/developer.html
```

Однако обратите внимание, что не все веб-серверы предоставляют информацию о дате изменения. Динамически генерируемые веб-страницы обычно не возвращают дату изменения.

Еще один способ определить, изменилась ли веб-страница, - это время от времени опрашивать ее и проверять. Удобный способ проверить, что веб-страница изменилась, - использовать функцию **checksum** (контрольная сумма). Если ранее вычисленная контрольная сумма веб-страницы отличается от ее текущего значения, значит, веб-страница была изменена с момента последней проверки. Вот пример, в котором используется эта техника. Он проверяет страницу каждые восемь часов.

```
forever [
    page: read http://www.rebol.com
    page-sum: checksum page
    if any [
        not exists? %page-sum
        page-sum <> (load %page-sum)
    ][
        print ["Page changed" now]
        save %page-sum page-sum
        send luke@rebol.com page
    ]
    wait 8:00
]
```

Каждый раз, когда страница изменяется, она отправляется Люку по электронной почте.

8.5 Действовать как браузер

Обычно REBOL идентифицирует себя для сервера при чтении с веб-сайта. Однако некоторые серверы запрограммированы так, чтобы отвечать только определенным браузерам. Если запрос к серверу не создает правильную веб-страницу, вы можете изменить запрос, чтобы он выглядел так, как будто он поступил из какого-либо другого типа веб-браузера. Многие программы притворяются веб-браузером, чтобы заставить веб-сайты правильно реагировать. Однако эта практика в конечном итоге сводит на нет цель идентификации браузера.

Чтобы изменить HTTP-запросы так, чтобы они выглядели так, как будто они отправляются Netscape 4.0, вы можете изменить пользовательский агент в обработчике HTTP:

```
system/options/http/user-agent: "Mozilla/4.0"
```

Установка этой переменной влияет на все последующие HTTP-запросы.

8.6 Отправка запросов CGI

Запросы HTTP CGI могут быть отправлены двумя способами. Вы можете включить данные запроса CGI в URL-адрес или предоставить данные запроса с помощью операции HTTP post.

В запросе URL CGI используется обычный URL. Пример ниже отправляет сценарию CGI `test.r` значение 10 переменной data.

```
read http://www.example.com/cgi-bin/test.r?data=10
```

Пост-запрос CGI требует, чтобы вы предоставили данные CGI как часть уточнения **/custom** функции чтения **read**. В приведенном ниже примере показано, как данные отправляются в CGI:

```
read/custom http://www.example.com/cgi-bin/test.r [
    post "data: 10"
]
```

В этом примере уточнение **/custom** используется для предоставления дополнительной информации для чтения. Второй аргумент - это блок, который начинается со слова **post**, за которым следует строка для отправки.

Метод post полезен для простой отправки кода и данных REBOL на веб-сервер, на котором работает CGI. Следующий пример иллюстрирует это:

```
data: [sell 10 shares of "ACME" at $123.45]

read/custom http://www.example.com/cgi-bin/test.r reduce [
    `post mold data
]
```

Функция mold создаст правильную строку REBOL, которая будет отправлена на сервер.

9. SMTP - простой протокол передачи почты (Simple Mail Transport Protocol)

Простой протокол передачи почты (SMTP) контролирует передачу сообщений электронной почты в Интернете. SMTP определяет взаимодействие между интернет-узлами, которые участвуют в пересылке электронной почты от отправителя к месту назначения.

9.1 Отправка электронной почты

Электронная почта отправляется через SMTP с использованием функции **send** (отправка). Эта функция может отправить сообщение электронной почты на один или несколько адресов электронной почты. Для правильной работы **send** необходимо настроить вашу сеть. Функция отправки **send** требует, чтобы вы указали свой адрес электронной почты От и почтовый сервер по умолчанию. См. Раздел «Начальная установка» выше.

Функция отправки **send** принимает два аргумента: адрес электронной почты и сообщение. Например:

```
send user@example.com "Hi from REBOL"
```

Первый аргумент должен быть типом данных электронной почты или блока. Второй аргумент может быть любым типом данных.

```
send luke@rebol.com $1000.00
send luke@rebol.com 10:30:40
send luke@rebol.com bill@ms.dom
send luke@rebol.com [Today 9-Apr-99 10:30]
```

Каждое из этих простых сообщений электронной почты можно интерпретировать на стороне получателя (с помощью REBOL) или просмотреть с помощью обычной программы электронной почты.

Вы можете отправить файл целиком, прочитав файл и передав его в качестве второго аргумента функции отправки:

```
send luke@rebol.com read %task.txt
```

Двоичные данные, такие как изображение или исполняемый файл, также могут быть отправлены:

```
send luke@rebol.com read/binary %rebol
```

Двоичные данные закодированы, чтобы их можно было передавать в виде текста.

Чтобы отправить самораспаковывающееся двоичное сообщение, вы можете написать:

```
send luke@rebol.com join "REBOL for the job" [
    newline "REBOL []" newline
    "write/binary %rebol decompress "
    compress read/binary %rebol
]
```

Когда сообщение получено, файл можно извлечь с помощью функции `do`.

9.2 Несколько получателей

Для отправки нескольким получателям вы можете указать блок имен электронной почты:

```
send [luke@rebol.com ben@example.com] message
```

В этом случае каждое сообщение адресуется индивидуально, а в поле "Кому" (`To`) отображается только имя получателя (аналогично адресации BCC).

Блок адресов электронной почты может быть любого размера или даже быть загруженным вами файлом. Просто убедитесь, что это действительные адреса, а не строки. Строки игнорируются.

```
friends: [
    bob@cnn.dom
    betty@cnet.dom
    kirby@hooya.dom
    belle@apple.dom
    ...
]

send friends read %newsletter.txt
```

9.3 Массовая рассылка

Если вы отправляете электронную почту большой группе, вы можете снизить нагрузку на свой сервер, доставив всем в группе одно сообщение. Это цель уточнения `/only`. Он использует функцию SMTP для отправки только одного сообщения на несколько адресов электронной почты. Используя список друзей из предыдущего примера:

```
send/only friends message
```

Сообщения не адресуются индивидуально. Возможно, вы видели этот режим в некоторых массовых рассылках, которые вы получаете. Когда вы получаете массовую рассылку писем, ваш адрес не отображается в поле `To` (Кому).

Режим массовой рассылки SMTP следует использовать для списков рассылки, а не для рассылки спама. Электронная почта со спамом не является надлежащим сетевым этикетом, в некоторых странах и штатах она является незаконной, а спам приведет к блокировке вашего интернет-провайдера и других сайтов.

9.4 Тема и заголовки

По умолчанию функция **send** использует первую строку сообщения в качестве темы. Чтобы указать собственную строку темы, вам необходимо указать заголовок электронной почты для функции **send**. В дополнение к теме вы можете указать организацию, дату и даже свои собственные настраиваемые поля.

Чтобы включить заголовок, используйте уточнение **/header** функции **send** и включите объект заголовка. Объект заголовка должен быть создан из объекта `system/standard/email`. Например:

```
header: make system/standard/email [
    Subject: "Seen REBOL yet?"
    Organization: "Freedom Fighters"
]
```

Обратите внимание, что стандартные поля, такие как адрес `From` (От), не являются обязательными и автоматически предоставляются функцией **send**.

Затем заголовок предоставляется в качестве аргумента для **send/header**:

```
send/header friends message header
```

Электронное письмо выше отправляется с использованием настраиваемого заголовка для каждого сообщения.

9.5 Отладка ваших скриптов

При тестировании сценариев электронной почты рекомендуется сначала отправить электронное письмо самому себе, прежде чем отправлять его другим. Внимательно изучите тестовое электронное письмо, чтобы убедиться, что оно именно то, что вам нужно. Часто возникают такие ошибки, как отправка имени файла, а не его содержимого. Например, вы можете написать:

```
send person %the-data-file.txt
```

Это отправляет имя файла, а не сам файл.

10. POP - протокол почтового отделения (Post Office Protocol)

Протокол почтового отделения (POP) позволяет получать электронную почту, ожидающую в почтовом ящике почтового сервера. POP определяет ряд операций для доступа и хранения электронной почты на вашем сервере.

10.1 Чтение электронной почты

Вы можете прочитать всю свою электронную почту в одной строке, не удаляя ее с почтового сервера: это делается путем чтения из URL-адреса POP, в котором вы указали свое имя пользователя, пароль и адрес электронной почты.

```
mail: read pop://user:pass@mail.example.com
```

Сообщения возвращаются в виде блока строк, который вы можете обрабатывать по одному сообщению за раз, используя такой код, как:

```
foreach message mail [print message]
```

Чтобы читать отдельные сообщения электронной почты с сервера, вам необходимо открыть соединение порта с сервером и обрабатывать каждое сообщение по одному. Чтобы открыть порт POP:

```
mailbox: open pop://user:pass@mail.example.com
```

В этом примере доступ к `mailbox` (почтовому ящику) можно получить как серию. Он отвечает на многие стандартные функции серии, такие как **length?** (длина?), **first** (первый), **second** (второй), **third** (третий), **pick** (выбрать), **next** (следующий), **back** (назад), **head** (голова), **tail** (хвост), **head?** (голова?) , **tail?** (хвост?), **remove** (удалить) и **clear** (очистить).

Чтобы определить количество почтовых сообщений, находящихся на сервере, используйте **length?** (длина?) функция.

```
print length? mailbox
37
```

Кроме того, вы можете узнать общий размер всех сообщений и отдельные размеры сообщений с помощью:

```
print mailbox/locals/total-size
print mailbox/locals/sizes
```

Чтобы отобразить первое, второе и последнее сообщение, вы можете написать:

```
print first mailbox
print second mailbox
print last mailbox
```

Вы также можете использовать **pick**, чтобы получить конкретное сообщение:

```
print pick mailbox 27
```

Вы можете получать и отображать каждое сообщение от самого старого до самого нового, используя цикл, идентичный тому, который используется для других типов серий:

```
while [not tail? mailbox] [
    print first mailbox
    mailbox: next mailbox
]
```

Вы также можете читать свою электронную почту от самой новой до самой старой с помощью цикла, например:

```
mailbox: tail mailbox

while [not head? mailbox] [
    mailbox: back mailbox
    print first mailbox
]
```

Когда вы закончите, не забудьте закрыть (`close`) почтовый ящик (mailbox). Это можно сделать с помощью такой строки, как:

```
close mailbox
```

10.2 Удаление электронной почты

Как и в случае с сериями, функцию `remove` (удаления) можно использовать для удаления одного сообщения, а функцию `clear` (очистка) можно использовать для удаления всех сообщений от текущей позиции до конца почтового ящика.

Например, чтобы прочитать сообщение, сохранить его в файл и удалить с сервера:

```
mailbox: open pop://user:pass@mail.example.com
write %mail.txt first mailbox
remove mailbox
close mailbox
```

Сообщение удаляется с сервера после **закрытия** (`close`).

Чтобы удалить 22-е сообщение электронной почты с сервера, вы можете написать:

```
user:pass@mail.example.com
remove at mailbox 22
close mailbox
```

Вы можете удалить ряд сообщений, используя уточнение `/part` с функцией удаления (remove):

```
remove/part mailbox 5
```

Чтобы удалить все сообщения в почтовом ящике, используйте функцию `clear` **(очистка):**

```
mailbox: open pop://user:pass@example.com
clear mailbox
close mailbox
```

функция `clear` также может быть использована в различных позициях в почтовом ящике, чтобы удалить сообщения в конце почтового ящика.

10.3 Обработка заголовков электронной почты

Сообщения электронной почты всегда включают заголовок. Заголовок содержит такую информацию, как отправитель, получатель, тема, дата и другие поля.

В REBOL заголовки электронной почты обрабатываются как объекты, содержащие все необходимые поля. Чтобы преобразовать сообщение электронной почты в объект заголовка, вы можете использовать функцию `import-email`. Например:

```
msg: import-email first mailbox
print first msg/from  ; the email address
print msg/date
print msg/subject
print msg/content
```

Вы можете легко написать фильтр, который сканирует вашу электронную почту на предмет сообщений, начинающихся с определенной строки темы:

```
mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    msg: import-email first mailbox
    if find/match msg/subject "[REBOL]" [
        print msg/subject
    ]
    mailbox: next mailbox
]

close mailbox
```

Вот ещё один пример, который информирует вас о получении электронного письма от группы друзей:

```
friends: [orson@rebol.com hans@rebol.com]

messages: read pop://user:pass@example.com

foreach message messages [
    msg: import-email message
    if find friends first msg/from [
        print [msg/from newline msg/content]
        send first msg/from "Got your email!"
    ]
]
```

Этот спам-фильтр удаляет с сервера все сообщения, в которых отсутствует ваше имя электронной почты:

```
mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    mailbox: either find first mailbox user@example.com
        [next mailbox][remove mailbox]
]

close mailbox
```

Вот простой сервер списков рассылки, который принимает сообщения и отправляет их группе. Сервер принимает электронную почту только от людей в группе.

```
group: [orson@rebol.com hans@rebol.com]

mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    message: import-email first mailbox
    mailbox: either find group first message/from [
        send/only group first mailbox
        remove mailbox
    ][next mailbox]
]

close mailbox
```

11. FTP - протокол передачи файлов (File Transfer Protocol)

Протокол передачи файлов (FTP) широко используется в Интернете для передачи файлов на удаленный хост и с него. FTP обычно используется для загрузки страниц на веб-сайт и для предоставления онлайн-архивов файлов.

11.1 Использование FTP

В REBOL операции с файлами FTP обрабатываются почти так же, как операции с локальными файлами. Такие функции, как `read` (чтение), `write` (запись), `load` (загрузка), `save` (сохранение), `do` (выполнение), `open` (открытие), `close` (закрытие), `exists?` (существуют?), `size?` (размер?), `modified?` (модифицированный?), и другие используются с FTP. REBOL различает локальные файлы и файлы, доступные по FTP через FTP-URL.

Доступ к FTP-серверам может быть открытым или закрытым. Открытый доступ позволяет любому войти на сайт и скачивать файлы. Это называется анонимным доступом и часто используется для общедоступных файловых архивов. Закрытый доступ требует, чтобы вы указали имя пользователя и пароль для загрузки и выгрузки файлов. Это режим работы для загрузки веб-страниц на веб-сайт.

Хотя FTP не требует настройки вашей сети REBOL, если вы хотите использовать анонимный доступ, требуется адрес электронной почты. Этот адрес находится в объекте `system/user/email`. Обычно, когда вы загружаете REBOL, это поле устанавливается из вашего пользовательского файла `user.r`. См. Дополнительные сведения в разделе « Начальная настройка».

Если вы используете FTP через прокси-сервер или брандмауэр, возможно, FTP должен работать в пассивном режиме. Пассивный режим не требует обратных подключений от FTP-сервера к клиенту для передачи данных. Этот режим устанавливает только исходящие соединения с вашего компьютера и обеспечивает более высокий уровень безопасности. Для включения пассивного режима вам необходимо установить флаг в обработчике протокола FTP:

```
system/schemes/ftp/passive: true
```

Если вы не знаете, нужно ли это, сначала попробуйте FTP без него. Если это не сработает, попробуйте установить пассивный флаг.

11.2 URL-адреса FTP

URL-адрес FTP имеет базовую форму:

```
ftp://user:pass@host/directory/file
```

Для анонимного доступа имя пользователя и пароль можно не указывать:

```
ftp://host/directory/file
```

В большинстве примеров в этом разделе для простоты используется эта форма; однако они также работают с именем пользователя и паролем.

Чтобы получить доступ к удаленному каталогу, завершите URL-адрес косой чертой, например:

```
ftp://user:pass@host/directory/
ftp://host/directory/
ftp://host/
```

Подробнее о доступе к каталогу показано ниже.

Удобно поместить URL-адрес в переменную и использовать пути для указания имен файлов. Это позволяет вам ссылаться на URL-адрес одним словом. Например:

```
site: ftp://ftp.rebol.com/pub/
read site/readme.txt
```

Этот метод используется в некоторых следующих разделах.

11.3 Передача текстовых файлов

FTP различает текстовые файлы и двоичные файлы. При передаче текстовых файлов FTP преобразует символы разрыва строки. Это нежелательно для двоичных файлов.

Чтобы прочитать текстовый файл, укажите для функции `read` (чтение) URL-адрес FTP:

```
file: read ftp://ftp.site.com/file.r
```

Это помещает содержимое файла в строку. Чтобы записать файл локально, используйте эту строку:

```
write %file.r read ftp://ftp.site.com/file.r
```

Также можно использовать многие из усовершенствований `read`. Например, вы можете использовать `read/lines` с:

```
data: read/lines ftp://ftp.site.com/file.r
```

В этом примере возвращается блок строк для файла. См. Главу «Файлы» для получения дополнительной информации об усовершенствованиях функции `read`.

Чтобы записать текстовый файл на сервер, используйте функцию `write` (запись):

```
write ftp://ftp.site.com/file.r read %file.r
```

Функция записи (`write`) также может включать уточнения. См. Главу "Файлы".

Как и при обычной передаче текстовых файлов, все окончания строк будут правильно преобразованы во время передачи по FTP.

Вот простой сценарий, который обновляет файлы на вашем веб-сайте:

```
site: ftp://wwwuser:secret@www.site.dom/pages

files: [%index.html %home.html %info.html]

foreach file files [write site/:file read file]
```

Его не следует использовать для передачи графических или звуковых файлов, поскольку они являются двоичными. Используйте технику, показанную в разделе «Передача двоичных файлов».

В дополнение к функциям чтения (`read`) и записи (`write`) вы можете использовать функции загрузки (`load`), сохранения (`write`) и выполнения (`do`) с FTP.

```
data: load ftp://ftp.site.com/database.r

save ftp://ftp.site.com/data.r data-block

do ftp://ftp.site.com/scripts/test.r
```

11.4 Передача двоичных файлов

Чтобы избежать преобразования окончания строки при передаче двоичных файлов (изображений, архивов, исполняемых файлов), используйте уточнение `/binary`. Например, чтобы прочитать двоичный файл с FTP-сервера:

```
data: read/binary ftp://ftp.site.com/file
```

Чтобы сделать локальную копию файла:

```
write/binary %file read/binary ftp://ftp.site.com/file
```

Чтобы сделать локальную копию файла:

```
write/binary ftp://ftp.site.com/file read/binary %file
```

Преобразования завершения строки не выполняются.
Чтобы передать набор графических файлов на веб-сайт, используйте этот сценарий:

```
site: ftp://user:pass@ftp.site.com/www/graphics
files: [%icon.gif %logo.gif %photo.jpg]

foreach file files [
    write/binary site/:file read/binary file
]
```

11.5 Добавление к файлам

FTP также позволяет добавлять текст и данные в существующий файл. Для этого используйте уточнение `write/append` (запись/добавление), как описано в главе «Файлы».

```
write/append ftp://ftp.site.com/pub/log.txt reform
    ["Log entry date:" now newline]
```

Это также можно использовать с двоичными файлами.

```
write/binary/append ftp://ftp.site.com/pub/log.txt
    read/binary %datafile
```

11.6 Чтение каталогов

Чтобы прочитать имена файлов в каталоге FTP, после имени каталога используйте косую черту:

```
print read ftp://ftp.site.com/

pub-files: read ftp://ftp.site.com/pub/
```

Конечная косая черта (/) указывает, что это доступ к каталогу, а не к файлу. Косая черта не всегда требуется, но рекомендуется, если вы знаете, что получаете доступ к каталогу.

Возвращаемый блок файлов включает все файлы в каталоге. В этом блоке имена каталогов обозначаются косой чертой после их имен. Например:

```
foreach file read ftp://ftp.site.com/pub/ [
    print file
]
readme.txt
rebol.r
rebol.exe
library/docs/
```

Вы также можете использовать `dir?` над файлом, чтобы определить, является ли он каталогом.

11.7 Информация о файле

Те же функции, которые предоставляют информацию о файлах, также предоставляют информацию о файлах FTP. Сюда входит `modified?` (модифицированный?), `size?` (размер?), `exists?` (существует?), `dir?` (директорий?) и `info?` (информация?) функции.

Вы можете использовать функцию `exists?`, чтобы определить, существует ли файл:

```
if exists? ftp://ftp.site.com/pub/log.txt [
    print "Log file is there"
]
```

Это также работает для каталогов, но включает косую черту в конце имени каталога:

```
if exists? ftp://ftp.site.com/pub/rebol/ [
    print read ftp://ftp.site.com/pub/rebol/
]
```

Чтобы получить размер или дату изменения файла:

```
print size? ftp://ftp.site.com/pub/log.txt
print modified? ftp://ftp.site.com/pub/log.txt
```

Чтобы определить, действительно ли файл является каталогом:

```
if dir? ftp://ftp.site.com/pub/text [
    print "It's a directory"
]
```

Вы можете получить всю эту информацию в едином доступе с помощью `info?` функция:

```
file-info: info? ftp://ftp.site.com/pub/log.txt

probe file-info

print file-info/size
```

Чтобы выполнить ту же операцию с каталогом:

```
probe info? ftp://ftp.site.com/pub/
```

Чтобы распечатать список каталогов:

```
files: open ftp://ftp.site.com/pub/

forall files [
    file: first files
    info: info? file
    print [file info/date info/size info/type]
]
```

11.8 Создание каталогов

Новые каталоги FTP можно создать с помощью функции **make-dir**:

```
make-dir ftp://user:pass@ftp.site.com/newdir/
```

11.9 Удаление файлов

При соответствующих настройках разрешений файлы могут быть удалены с удаленного FTP-сервера с помощью функции `delete` (удаление):

```
delete ftp://user:pass@ftp.site.com/upload.txt
```

Вы также можете удалить каталоги:

```
delete ftp://user:pass@ftp.site.com/newdir/
```

Обратите внимание, что для этого каталог должен быть пустым.

11.10 Переименование файлов

Вы можете переименовать файл используя функцию `rename`:

```
rename ftp://user:pass@ftp.site.com/foo.r %bar.r
```

Новое имя файла будет `bar.r`.

FTP также позволяет переместить файл в другой каталог с помощью:

```
rename ftp://user:pass@ftp.site.com/foo.r %pub/bar.r
```

Чтобы переименовать каталог на FTP-сайте, не забудьте поставить после имени каталога косую черту:

```
rename ftp://user:pass@ftp.site.com/rebol/ rebol-old/
```

11.11 О паролях

Приведенные выше примеры включают пароль в свои URL-адреса, но если вы планируете поделиться своим скриптом, вы, вероятно, не хотите, чтобы эта информация была известна. Вот простой способ запросить пароль и создать правильный URL:

```
pass: ask "Password? "

data: read join ftp://user: [pass "@ftp.site.com/file"]
```

Или вы можете запросить имя пользователя и пароль:

```
user: ask "Username? "
pass: ask "Password? "
data: read join ftp:// [
    user ":" pass "@ftp.site.com/file"
]
```

Вы также можете открывать FTP-соединения, используя спецификацию порта, а не URL-адрес. Это позволяет использовать любой пароль, даже те, которые содержат специальные символы, которые нелегко записать в URL-адресах.

Пример спецификации порта для открытия FTP-соединения:

```
ftp-port: open [
    scheme: `ftp
    host: "ftp.site.com"
    user: ask "Username? "
    pass: ask "Password? "
]
```

См. «Указание сетевых ресурсов» выше для получения более подробной информации.

11.12 Передача больших файлов

Передача больших файлов требует особого внимания. Вы можете передавать файл по частям, чтобы уменьшить объем памяти, необходимый вашему компьютеру, и обеспечить обратную связь с пользователем во время передачи.

Вот пример, который загружает очень большой двоичный файл по частям.

```
inp: open/binary/direct ftp://ftp.site.com/big-file.bmp
out: open/binary/new/direct %big-file.bmp
buf-size: 200000
buffer: make binary! buf-size + 2

while [not zero? size: read-io inp buffer buf-size][
    write-io out buffer size
    total: total + size
    print ["transferred:" total]
]
```

Обязательно используйте уточнение `/direct`, иначе весь файл будет буферизован внутренне REBOL. Функции `read-io` и `write-io` позволяют повторно использовать уже выделенную буферную память. Другие функции, такие как `copy` (копирование), выделяют дополнительную память.

Если передача не удалась, вы можете перезапустить FTP с того места, где она была прервана. Для этого изучите выходной файл или переменную размера, чтобы определить, где перезапустить передачу. Откройте файл еще раз с уточнением `/custom` (настраиваемым), которое указывает `restart` (перезапуск) и расположение, с которого следует начать чтение.

Вот пример функции open для использования, когда переменная total указывает уже прочитанную длину:

```
inp: open/binary/direct/custom

ftp://ftp.site.com/big-file.bmp

reduce ['restart total]
```

Обратите внимание, что перезапуск работает только для двоичных передач. Его нельзя использовать с передачей текста, потому что выполняемое преобразование ограничителя строки приведет к неправильным смещениям.

12. NNTP - протокол передачи сетевых новостей (Network News Transfer Protocol)

Протокол передачи сетевых новостей (NNTP) является основой для десятков тысяч групп новостей, которые обеспечивают общественный форум для миллионов пользователей Интернета. REBOL включает два уровня поддержки NNTP.

Встроенная поддержка NNTP, обеспечивающая очень ограниченную функциональность и доступ. Это схема NTTP.

Расширенный уровень функциональности, обеспечиваемый схемой новостей, реализованной в файле, распространяемом как **nntp.r**.

12.1 Чтение списка групп новостей

NNTP состоит из двух компонентов: списка групп новостей, поддерживаемых определенным сервером группы новостей (группы новостей обычно выбираются поставщиком услуг Интернета); и база данных сообщений, которые в настоящее время доступны для любой конкретной группы новостей.

Чтобы получить список всех групп новостей с определенного сервера новостей, используйте функцию read (чтение) с URL-адресом NNTP, например:

```
groups: read nntp://news.example.com
```

Это может занять некоторое время, в зависимости от вашего подключения; есть тысячи групп новостей.

12.2 Чтение всех сообщений

Если вы используете быстрое соединение, вы можете прочитать все ожидающие сообщения для группы новостей с помощью:

```
messages: read nntp://news.example.com/alt.test
```

Однако рекомендуется соблюдать осторожность. Некоторые группы новостей могут содержать тысячи сообщений. Загрузка всех сообщений может занять много времени, и вам может не хватить памяти для их хранения.

12.3 Чтение одиночных сообщений

Чтобы читать отдельные сообщения, откройте NNTP как порт и используйте последовательные функции для доступа к сообщениям. Это похоже на то, как вы читаете электронную почту через порт POP. Например:

```
group: open nntp://news.example.com/alt.test
```

Вы можете использовать функцию `length?` (длина?) для определения количества сообщений, доступных в группе новостей:

```
print length? group
```

Чтобы прочитать первое сообщение, доступное в группе новостей, используйте **first**:

```
message: first group
```

Чтобы выбрать конкретное сообщение в группе по индексу, используйте команду **pick**:

```
message: pick group 37
```

Чтобы создать простой цикл, который просматривает все сообщения на предмет ключевого слова, используйте:

```
forall group [
    if find msg: first first group "REBOL" [
        print msg
    ]
]
```

Помните, что когда цикл завершается, указатель серии group располагается в хвосте. Если нужно вернуться к началу группы:

```
group: head group
```

Обязательно закройте (**close**) порт, когда закончите его использовать:

```
close group
```

12.4 Обработка заголовков новостей

Новостные сообщения всегда включают заголовок. Заголовок содержит такую информацию, как отправитель, сводка, ключевые слова, тема, дата и другие поля.
Заголовки обрабатываются как объекты. Чтобы преобразовать новостное сообщение в объект заголовка новости, вы можете использовать функцию **import-email**. Например:

```
message: first first group
header: import-email message
```

Теперь вы можете получить доступ к полям новостного сообщения:

```
print [header/from header/subject header/date]
```

Разные группы новостей и клиенты групп новостей используют разные поля в своих заголовках. Чтобы просмотреть поля, доступные для конкретного сообщения, отобразите первый элемент объекта заголовка:

```
print first header
```

12.5 Отправка новостного сообщения

Прежде чем вы сможете отправить новостное сообщение, вам необходимо создать для него заголовок. Вот общий заголовок, который можно использовать для новостей:

```
news-header: make object! [
    Path: "not-for-mail"
    Sender: Reply-to: From: system/user/email
    Subject: "Test message"
    Newsgroups: "alt.test"
    Message-ID: none
    Organization: "Docs For All"
    Keywords: "Test"
    Summary: "A test message"
]
```

Прежде чем вы сможете его отправить, вам необходимо создать для него уникальный глобальный идентификационный номер. Вот функция, которая это делает:

```
make-id: does [
    rejoin [
        "<"
        system/user/email/user
        "."
        checksum form now
        "."
        random 999999
        "@"
        read dns://
        ">"
    ]
]

print news-header/message-id: make-id
<carl.4959961.534798@fred.example.com>
```

Теперь вы можете объединить заголовок с сообщением. Они должны быть разделены хотя бы одной пустой строкой. Содержание сообщения читается из файла.

```
write nntp://news.example.net/alt.test rejoin [
    net-utils/export news-header
    newline newline
    read %message.txt
    newline
]
```

13. CGI - Общий интерфейс шлюза (Common Gateway Interface)

Общий интерфейс шлюза используется со многими веб-серверами для обеспечения обработки, выходящей за рамки обычного веб-интерфейса HTTP. Запросы CGI отправляются из веб-браузеров на веб-серверы. Когда сервер получает запрос CGI, он обычно выполняет сценарий для обработки запроса и возврата результата в браузер. Эти сценарии CGI могут быть написаны на разных языках, и REBOL предоставляет один из наиболее простых способов работы с CGI.

13.1 Настройка сервера CGI

Настройка доступа CGI отличается для каждого веб-сервера. См. Инструкции, прилагаемые к вашему серверу.

Обычно сервер имеет возможность включить работу CGI. Вам необходимо включить эту опцию и указать путь к каталогу, в котором находятся ваши сценарии CGI. Общий каталог для сценариев CGI находится в `cgi-bin`.

На серверах Apache опция `ExecCGI` включает сценарии CGI, и вы можете предоставить каталог (`cgi-bin`) для своих сценариев. Обычно это устанавливается при установке Apache по умолчанию.

Чтобы настроить CGI для Microsoft IIS, перейдите в свойства cgi-bin и нажмите кнопку конфигурации. На панели конфигурации нажмите «Добавить» и введите путь к вашему файлу rebol .exe. Формат для этого:

```
C:\rebol\rebol.exe -cs %s %s
```

Два символа %s необходимы для правильной передачи аргументов сценария и командной строки в REBOL. Добавьте расширение для файлов REBOL (`.r`) и установите последнее поле в `PUT`, `DELETE`. Механизм сценария выбирать не нужно.

Параметр -`cs`, предоставляемый REBOL, включает работу CGI и позволяет сценарию получить доступ ко всем файлам. (!! См. Примечания ниже о том, как сценарии могут ограничивать доступ к файлам для выбранных каталогов).

С веб-серверами, отличными от описанных выше, серверу требуется конфигурация для выполнения исполняемого файла REBOL для `.r` и запустите REBOL с необходимой опцией **`-cs`**.

13.2 Скрипты CGI

Прежде чем сценарий может быть запущен на большинстве серверов CGI, он должен иметь правильные права доступа к файлам. В системах типа UNIX или тех, которые используют сервер Apache, вам необходимо изменить разрешения, чтобы сценарий был доступен для чтения и выполнения всеми пользователями. Это можно сделать с помощью функции `chmod`. Если вы новичок в этой концепции, вам следует прочитать руководство по своей операционной системе или поговорить с системным администратором, прежде чем изменять права доступа к файлам.

Чтобы Apache и другие веб-серверы могли запускать сценарии REBOL, вам необходимо указать правильный заголовок в верхней части каждого файла сценария. В заголовке указывается путь к исполняемому файлу REBOL и опция `-cs`. За ним может следовать обычный заголовок сценария REBOL. Вот простой сценарий CGI, который печатает строку, `hello!`.

```
#!/path/to/rebol -cs
REBOL [Title: "CGI Test Script"]
print "Content-Type: text/plain"
print ""  ; required
print "Hello!"
```

Есть много вещей, которые мешают правильной работе CGI-скрипта. Прежде чем пробовать более сложные сценарии, запустите этот простой сценарий. Если ваш сценарий не работает, проверьте следующее: * На вашем веб-сервере включен CGI. * Первая строка начинается с #! и правильный путь к REBOL. * Опция -cs входит в REBOL. * Сценарий начинается с печати «Content-Type:». (!!см. ниже) * Скрипт находится в правильном каталоге. (обычноcgi-bin каталог) * У скрипта есть правильные права доступа к файлу (все могут читать и исполнять). * Сценарий содержит правильные символы разрыва строки. Некоторые серверы не запускают сценарии, содержащие символ CR для разрыва строки. Возможно, вам потребуется преобразовать файл. (Используйте REBOL, чтобы сделать это в одну строку: записать файл, прочитать файл). * Скрипт не содержит ошибок. Протестируйте его без CGI, чтобы убедиться, что скрипт загружается (не имеет синтаксических ошибок) и работает правильно. Предоставьте образцы данных и проверьте их. * Все файлы, к которым обращается сценарий, имеют правильные права доступа к файлам.

Часто один или несколько из вышеперечисленных пунктов неверны и мешают запуску вашего скрипта. Вы можете увидеть ошибку при просмотре веб-страницы. Если там написано «Ошибка сервера» или «Ошибка CGI», то обычно это связано с разрешениями или настройкой скрипта. Если отображается сообщение об ошибке REBOL, значит, сценарий запущен, но в нем есть ошибка.

В примере сценария, показанном выше, критически важна строка `Content-Type`. Это часть HTTP-заголовка, который возвращается браузеру, и он сообщает браузеру тип доставляемого контента. После этого следует пустая строка, чтобы отделить ее от фактического содержимого.

Может быть доставлено много различных типов контента. Предыдущий пример был простым текстом, но вы также можете доставлять HTML, как показано в следующем примере. (Дополнительную информацию о типах содержимого см. В руководстве к веб-серверу.)

Тип содержимого и пустую строку можно объединить в одну строку. Символ каретки (`^/`) обеспечивает дополнительный разрыв строки, чтобы отделить его от содержимого.

```
print "Content-Type: text/plain^/"
```

Рекомендуется всегда выводить эту строку сразу из сценария. Это позволяет браузеру видеть сообщения об ошибках, если ваш скрипт обнаруживает ошибку.

Вот простой CGI-скрипт, который печатает время:

```
#!/path/to/rebol -cs
REBOL [Title: "Time Script"]
print "Content-Type: text/plain^/"
print ["The time is now" now/time]
```

13.3 Создание HTML-содержимого

Существует столько же способов создания содержимого HTML, сколько способов создания строк.

Эта страница создает страницу, на которой отображается счетчик посещений страницы:

```
#!/path/to/rebol -cs
REBOL [Title: "HTML Example"]
print "Content-Type: text/html^/"
count: either exists? %counter [load %counter][0]
save %counter count: count + 1
```

```
print [
    {<HTML><BODY><H2>Web Counter Page</H2>
    You are visitor} count {to this page!<P>
    </BODY></HTML>}
]
```

Сценарий в приведенном выше примере загружается и сохраняет в текстовый файл счетчика. Для того, чтобы этот файл был доступен, необходимо установить соответствующие разрешения, чтобы разрешить доступ всем пользователям.

13.4 Среда CGI

Когда запускается сценарий CGI, сервер предоставляет REBOL информацию о запросе CGI и его аргументах. Вся эта информация предоставляется в виде объекта в объекте `system/options`. Чтобы просмотреть поля объекта, введите:

```
probe system/options/cgi

make object! [
    server-software: none
    server-name: none
    gateway-interface: none
    server-protocol: none
    server-port: none
    request-method: none
    path-info: none
    path-translated: none
    script-name: none
    query-string: none
    remote-host: none
    remote-addr: none
    auth-type: none
    remote-user: none
    remote-ident: none
    Content-Type: none
    content-length: none
    other-headers: []
]
```

Конечно, ваш сценарий проигнорирует большую часть этой информации, но некоторые из них могут быть полезны. Например, вы можете создать файл журнала, в котором будет записан сетевой адрес системы, отправившей запрос, или проверить тип используемого браузера.

Чтобы создать CGI-страницу, отображающую этот контент в вашем браузере:

```
#!/path/to/rebol -cs

REBOL [Title: "Dump CGI Server Variables"]

print "Content-Type: text/plain^/"

print "Server Variables:"

probe system/options/cgi
```

Если вы хотите использовать эту информацию в журнале, вы можете записать ее в файл. Например, чтобы регистрировать адреса посетителей вашей CGI-страницы, вы можете написать:

```
write/append/lines %cgi.log
    system/options/cgi/remote-addr
```

/Append и **/lines** уточнения вызывает запись, чтобы быть в хвосте файла и включает в себя разрыв строки. Вот еще один подход, при котором несколько элементов помещаются в одну строку:

```
write/append %cgi.log reform [
    system/options/cgi/remote-addr
    system/options/cgi/remote-ident
    system/options/cgi/content-type
    newline
]
```

13.5 Запросы CGI

CGI может предоставить данные запроса вашим скриптам двумя способами: GET и POST.

Метод GET кодирует данные CGI в URL-адрес. Это используется для предоставления информации серверу. Возможно, вы уже заметили, что некоторые URL-адреса выглядят так:

```
http://www.example.com/cgi-bin/test.r?&data=test
```

Строка, следующая за вопросительным знаком (?), Предоставляет аргументы CGI. Иногда они могут быть довольно длинными. Эта строка предоставляется вашему скрипту при его запуске. Его можно получить из поля `cgi/query-string`. Например, чтобы распечатать строку из скрипта:

```
print system/options/cgi/query-string
```

Данные в строке могут включать любые требуемые данные. Однако, поскольку строка является частью URL-адреса, данные должны быть закодированы. Существуют ограничения на разрешенные символы. Кроме того, когда данные создаются HTML-формами, они кодируются стандартным способом. Эти данные могут быть декодированы и помещены в объект с помощью кода:

```
cgi: make object! decode-cgi-query
    system/options/cgi/query-string
```

Функция `decode-cgi-query` возвращает блок, содержащий имена переменных и их значения. См. Пример HTML-формы в следующем разделе.
Метод POST предоставляет данные CGI в виде строки. Данные не нужно кодировать. Он может быть в любом желаемом формате и даже в двоичном формате. Почтовые данные считываются со стандартного устройства ввода. Вам нужно будет прочитать его из ввода с помощью такой строки, как:

```
data: make string! 2002
read-io system/ports/input data 2000
```

Это будет читать до первых 2000 байтов данных POST и помещать их в строку.

Хороший формат для данных POST - использовать диалект REBOL и создать простой синтаксический анализатор. Данные POST можно загружать и анализировать как блок. См. Главу «Анализ».

Предупреждение о блоках

Не рекомендуется передавать блоки REBOL для непосредственной оценки, поскольку это может представлять угрозу безопасности. Например, кто-то может POST блок, который читает или удаляет файлы на сервере. Однако можно безопасно передавать блоки, которые интерпретируются вашим скриптом (диалектом).

Вот пример сценария, который отображает данные публикации в вашем браузере:

```
#!/path/to/rebol -cs

REBOL [Title: "Show POST data"]

print "Content-Type: text/html^/"
data: make string! 10000
foreach line copy system/ports/input [
    repend data [line newline]
]

print [
    <HTML><BODY>
    {Here is the posted data.}
    <HR><PRE>data</PRE>
    </BODY></HTML>
]
```

13.6 Обработка HTML-форм

CGI часто используется для обработки HTML-форм. Формы принимают ввод из различных полей и отправляют их на веб-сервер как HTML-метод get или post.

Вот пример, который использует CGI get для обработки формы и отправки электронного письма в результате. Это состоит из двух частей: HTML-страницы и CGI-скрипта.

Вот HTML-страница, содержащая форму:

```
<HTML><BODY>

<FORM ACTION="http://example.com/cgi-bin/send.r" METHOD="GET">

<H1>CGI Emailer</H1><HR>

Enter your email address:<P>

<INPUT TYPE="TEXT" NAME="email" SIZE="30"><P>

<TEXTAREA NAME="message" ROWS="7" COLS="35">
Enter message here.
</TEXTAREA><P>

<INPUT TYPE="SUBMIT" VALUE="Submit">

</FORM>
</BODY></HTML>
```

При отправке вышеуказанного сценария ему требуется сценарий CGI для обработки его результатов. Вот пример такого сценария. Этот пример сценария декодирует данные формы и отправляет электронное письмо. Он возвращает страницу подтверждения.

```
#!/path/to/rebol -cs

REBOL [Title: "Send CGI Email"]

print "Content-Type: text/html^/"

cgi: make object! decode-cgi-query
    system/options/cgi/query-string

print {<HTML><BODY><H1>Email Status</H1><HR><P>}

failed: error? try [send to-email cgi/email cgi/message]

print either failed [
    {The email could not be sent.}
][
    [{The email to} cgi/email {was sent.}]
]

print {</BODY><HTML>}
```

Этот сценарий должен называться `send.r` и храниться в каталоге `cgi-bin`. Его разрешения должны быть доступны для чтения и выполнения для всех.
Этот сценарий будет запущен после отправки формы браузером. Он декодирует строку запроса CGI в объект cgi. Теперь у объекта есть переменные электронной почты и сообщения, которые используются для функции `send` (отправка). Перед отправкой поле электронной почты преобразуется из строки в тип данных `email` (электронной почты).
Функция `send` помещается в блок `try`, чтобы отловить ошибки, если они возникают при отправке электронного письма. Для переменной с ошибкой устанавливается значение true, если произошла ошибка, и создается соответствующее сообщение.
Другие примеры CGI можно найти в библиотеке сценариев REBOL по адресу **http://www.rebol.com/library/library.html**.

14. TCP - протокол управления передачей (Transmission Control Protocol)

В дополнение к ранее описанным протоколам вы можете создавать свои собственные сетевые серверы и клиенты с протоколом управления передачей TCP.

14.1 Создание клиентов

TCP-порты можно открывать так же, как и другие протоколы REBOL, используя TCP-URL. Чтобы открыть TCP-соединение с HTTP (веб-сервером) на TCP-порту 80:

```
http-port: open tcp://www.example.com:80
```

Другой способ открытия TCP-соединения - напрямую указать спецификацию порта. Это замена URL-адреса и часто бывает весьма полезной:

```
http-port: open [
    scheme: 'tcp
    host: "www.example.com"
    port-id: 80
]
```

Поскольку порты являются последовательными, вы можете использовать одни и те же последовательные функции для отправки и получения данных. В приведенном ниже примере запрашивается HTTP-сервер, открытый в предыдущем примере. Он использует функцию insert (вставить) для помещения данных в серию портов, которая отправляет их на сервер:

```
insert http-port "GET / HTTP/1.0^/^/"
```

Два символа новой строки используются, чтобы сообщить серверу, что заголовок был отправлен.

Символы новой строки автоматически преобразуются в последовательности CR LF, поскольку порт был открыт в текстовом режиме.

Всервер обрабатывает HTTP-запрос и возвращает результат в портов серию. Чтобы прочитать результат, воспользуйтесь функцией **copy**:

```
while [data: copy http-port] [prin data]
```

Этот цикл будет продолжать выборку данных до тех пор, пока из копии не будет возвращено **none**. Это поведение различается в зависимости от протокола. Нет возвращается, потому что сервер закрывает соединение. Другие протоколы могут отправлять специальный символ для обозначения конца передачи.

Теперь, когда все данные получены, следует закрыть HTTP-порт:

```
close http-port
```

Вот еще один пример подключения к порту POP на сервере:

```
pop: open/lines tcp://fred.example.com:110
```

В этом примере используется уточнение /lines. Теперь соединение будет линейным. Данные будут записаны и прочитаны как строки. Чтобы прочитать первую строку с сервера:

```
print first pop
+OK QPOP (version 2.53) at fred.example.com starting.
```

Чтобы отправить серверу имя пользователя для входа в систему по протоколу POP:

```
insert pop "user carl"
```

Поскольку порт работает в линейном режиме, терминатор строки отправляется после вставки. Ответ сервера можно прочитать с помощью:

```
print first pop
+OK Password required for carl.
```

А в остальном общение будет происходить следующим образом:

```
insert pop "pass secret"

print first pop
+OK carl has 0 messages (0 octets).
insert pop "quit"

first pop
+OK Pop server at fred.example.com signing off.
```

Теперь соединение должно быть закрыто:

```
close pop
```

14.2 Создание серверов

Чтобы создать сервер, вам нужно дождаться соединений и отвечать на них по мере их возникновения. Чтобы настроить порт на вашем компьютере, который можно использовать для ожидания входящих подключений:

```
listen: open tcp://:8001
```

Обратите внимание, что вы не указываете имя хоста, а только номер порта. Этот тип порта называется портом **listen**. Теперь система принимает соединения через порт 8001.

Чтобы дождаться соединения с другой машиной, выполните wait на слушающем порту.

```
wait listen
```

Эта функция не возвращается, пока не будет установлено соединение.

*ПРИМЕЧАНИЕ. Есть и другие варианты ожидания (**wait**). Например, вы можете подождать на нескольких портах или на тайм-аут.*

Теперь вы можете открыть порт подключения с машины, которая связалась с вашей системой:

```
connection: first listen
```

Это возвращает соединение, которое было выполнено с портом прослушивания. Это порт, как и все остальные, и теперь его можно использовать для приема и отправки данных с помощью функций **insert** (вставка), **copy** (копирование), **first** (первый) и других последовательных функций:

```
insert connection "you are connected^/"

while [newline <> char: first connection] [
    print char
]
```

Когда связь будет завершена, соединение должно быть закрыто:

```
close connection
```

Теперь вы готовы к следующему подключению к порту прослушивания. Вы можете снова подождать (`wait`) и снова использовать `first`, чтобы установить соединение.

Когда вы закончите с обслуживанием, вы можете закрыть порт прослушивания с помощью:

```
close listen
```

14.3 Крошечный сервер

Вот полезный сервер REBOL, для которого требуется всего несколько строк кода. Этот сервер оценивает любой отправленный ему код REBOL. Строки REBOL читаются от клиента до тех пор, пока не возникнет ошибка. Каждая строка должна быть полным выражением REBOL. Они могут быть любой длины, но должны состоять из одной строки.

```
server-port: open/lines tcp://:4321

forever [
    connection-port: first server-port
    until [
        wait connection-port
        error? try [do first connection-port]
    ]
    close connection-port
]
close server-port
```

В случае ошибки соединение закрывается, и сервер ожидает следующего соединения.

Вот пример клиентского скрипта, который позволяет удаленно вводить командные строки REBOL:

```
server: open/lines tcp://localhost:4321
until [error? try [insert server ask "R> "]]
close server
```

Здесь запрос используется, чтобы определить, было ли соединение закрыто из-за ошибки.

14.4 Тестирование TCP-кода

Чтобы протестировать серверный код, подключитесь со своего собственного компьютера, вместо того, чтобы требовать и сервера, и клиента. Это можно сделать из двух отдельных процессов REBOL или даже из одного и того же процесса.

Чтобы подключиться к локальному компьютеру, вы можете использовать такую строку, как:

```
port: open tcp://localhost:8001
```

Вот пример, в котором два порта подключаются друг к другу в линейном режиме. Это своего рода эхо-порт (echo), поскольку вы отправляете данные самому себе. Он обеспечивает хороший тест вашего кода и сети:

```
listen: open/lines tcp://:8001
remote: open/lines tcp://localhost:8001
local: first listen
insert local "How are you?"
print first remote  ; response
close local
close remote
close listen
```

15. UDP - протокол дейтаграмм пользователя (User Datagram Protocol)

Протокол дейтаграмм пользователя - это еще один протокол транспортного уровня, который обеспечивает связь между машинами без установления соединения. Он позволяет отправлять дейтаграммы, пакеты между машинами.
Работа UDP сильно отличается от TCP. UDP проще, но по сути ненадежен. Нет никакой гарантии, что пакет когда-либо достигнет места назначения. Кроме того, UDP не имеет управления потоком. Если вы отправляете сообщения слишком быстро, пакеты могут быть потеряны.
Как и TCP, функция wait (ожидание) может использоваться для ожидания прибытия следующего пакета, а функция copy используется для возврата данных. Если данных нет, копирование ждет, пока они не появятся. Обратите внимание, однако, что insert (вставка) никогда не ждет.
Вот пример простого скрипта UDP-сервера:

```
udp: open udp://:9999
wait udp
print copy udp
insert udp "response"
close udp
```

Сообщения, вставленные здесь сервером, отправляются клиенту, от которого сервер в последний раз получил сообщение. Это позволяет отправлять ответы на входящие сообщения. Однако, в отличие от TCP, у вас нет постоянного соединения между машинами. Каждая пакетная передача - это отдельный обмен.
Клиентский сценарий для связи с указанным выше сервером будет:

```
udp: open udp://localhost:9999
insert udp "Test"
wait udp
print copy udp
close udp
```

Вы должны знать, что максимальный размер пакета UDP зависит от операционной системы. 32 КБ и 64 КБ являются общими значениями. Чтобы отправлять большие объемы данных, вам необходимо буферизовать данные, разделив их на более мелкие части. Однако требуется тщательное программирование, чтобы гарантировать получение каждой части данных. Помните, что с UDP нет никаких гарантий.

Глава 14 – Порты

Содержание:

1. Обзор
2. Открытие порта
    2.1 Функция открытия
    2.2 Открытые уточнения
3. Закрытие порта
4. Чтение из порта
5. Запись в порт
6. Обновление порта
7. Ожидание порта
8. Другие режимы порта
    8.1 Линейный режим
    8.2 Только чтение и запись
    8.3 Прямой доступ к портам
    8.4 Пропуск данных
9. Права доступа к файлам
10. Порты каталогов

1. Обзор

Порты обращаются к внешним (***external***) сериям, таким как файлы, сети, консоли, события, базы данных, кодировщики данных и декодеры данных. Данные порта обрабатываются с использованием стандартных функций серии REBOL, как описано в главе «Серии».

Порты используются как для ввода, так и для вывода. Тип данных, которые обрабатывает порт, зависит от того, как порт открыт. Возможны три типа данных:

String	серия байтов, преобразует разрывы строк (по умолчанию)
Binary	серия байтов, без преобразования данных
Block	серия значений REBOL

Порт можно открыть в одном из двух режимов буферизации:

Buffered	все данные хранятся в памяти (по умолчанию)
Direct	данные не хранятся в памяти

Кроме того, порт можно открыть с помощью:

Wait	порт будет ждать данных (по умолчанию)
No-wait	порт не будет ждать данных

2. Открытие порта

2.1 Функция открытия

Функция **open** инициализирует доступ к порту в соответствии с указанными параметрами. Функция может быть снабжена именем файла, URL-адресом или объектом. Кроме того, есть несколько уточнений, которые повлияют на операцию открытия или доступ к данным порта.

Самый простой способ использования **open** - предоставить ему имя файла или URL в качестве аргумента. В приведенном ниже примере открывается файловый порт:

```
fp: open %file.txt
```

FP переменная относится к порту. Если порт не открылся, произойдет ошибка. При необходимости ошибку можно отловить с помощью функции **try**.

По умолчанию файл открывается как буферизованный. Это означает, что к файлу осуществляется доступ и модифицируется в памяти, и изменения в файл не записываются, пока порт не будет закрыт или обновлен.

Для файлов функция **open** автоматически создаст файл, если он еще не существует.

```
close open %somefile.txt
if exists? %somefile.txt [print "somefile exists"]
somefile exists
```

Уточнение **/new** может быть использовано для перезаписи существующего файла.

```
write %somefile.txt "text in some file"
print read %somefile.txt
text in some file
close insert open/new %somefile.txt "new data"
print read %somefile.txt
new data
```

После того, как порт открыт, последовательные операции, такие как копирование (**copy**), вставка (**insert**), удаление (**remove**), очистка (**clear**), первый (**first**), следующий (**next**) и длина? (**length?**) может использоваться для доступа и изменения содержимого порта.

2.2 Уточнения функции open

Функция **open** принимает ряд уточнений, которые можно использовать для изменения ее работы:

/binary	данные порта являются двоичными
/string	данные порта - текст, перевод всех символов окончаний строк
/with	указать альтернативное завершение строки
/lines	обрабатывать данные построчно или как блок строк
/direct	не буферизировать порт
/new	создать или воссоздать цель порта
/read	открыт для операции только для чтения
/write	открыт только для записи
/no-wait	не ждать данных

/skip	пропустить часть данных
/allow	указать атрибуты защиты файлов
/custom	разрешить особые уточнения

3. Закрытие порта

Доступ к порту завершается функцией **close** (закрыть). Все несохраненные буферизованные данные будут записаны в целевой файл. В приведенном ниже примере будет закрыт порт, открытый ранее:

```
close fp
```

Если вы попытаетесь закрыть закрытый порт, произойдет ошибка.
Закрытый порт можно снова открыть с помощью функции **open**:

```
open fp
```

4. Чтение из порта

Функция **copy** используется для чтения данных из открытого порта:

```
print copy fp
I wanted the gold, and I sought it,I scrabbled and mucked like
a slave....
```

Эта функция будет ждать данных порта. Если вы не хотите ждать данных, откройте порт с уточнением **/no-wait**.
Чтобы прочитать только часть данных порта, используйте **copy/part**:

```
print copy/part fp 35
I wanted the gold, and I sought it,
```

Обратите внимание, что вторым аргументом для копирования может быть длина или позиция в порту.

Вы можете использовать функции поиска (**find**) и копирования (**copy**) серий, чтобы прочитать только часть данных порта:

```
a: find fp "famine"
print copy/part a find a newline
famine or scurvy -- I fought it;
```

В порте также могут использоваться функции **first**, **next** и другие позиционные последовательные функции:

```
print first fp
I
print first next next fp
w
```

Функция **copy** не вернет **none**, когда все данные будут прочитаны из порта. Когда не работает в **/no-wait** режима, то функция **copy** возвращает пустую строку, если нет данных для порта.

```
tp: open/direct/binary/no-wait tcp://system:8000

content: make binary! 1000

while [wait tp  data: copy tp] [append content data]

close tp
```

5. Запись в порт

Функция вставки (**insert**) используется для записи в порт.

```
insert fp "I was a fool to seek it."
```

Если порт находится в буфере, изменение будет происходить извне, когда порт будет закрыт или обновлен (с функцией **update**). Если порт открыт с помощью **/direct**, изменение произойдет немедленно.

Все уточнения функции **insert** могут быть использованы для порта. Например, чтобы записать в порт 20 пробелов:

```
insert/dup fp " " 20
```

Вы также можете использовать для порта функции **remove** (удаления), **clear** (очистки), **change** (изменения), **append** (добавления), **replace** (замены) и другие функции изменения серии.

Например, чтобы удалить один символ или несколько символов:

```
remove fp

remove/part fp 20
```

и чтобы удалить все оставшиеся символы, напишите:

```
clear fp
```

6. Обновление порта

Функция **update** заставляет порт обновлять свой статус по отношению к внешнему устройству. Например, при записи буферизованного файла функцию обновления можно использовать для принудительного вывода буфера данных в файл. При чтении можно использовать функцию **update**, чтобы быть уверенным, что все ожидающие данные были прочитаны в память.

```
update fp
```

7. Ожидание порта

Функция **wait** (ожидание) важна для программ, которым необходимо обрабатывать асинхронную передачу данных. С помощью **wait** вы можете дождаться данных на одном или нескольких портах или истечения времени ожидания.

Функция **wait** принимает один порт:

```
wait port
```

или может быть предоставлен целый блок портов:

```
wait [port1 port2 port3]
```

Кроме того, значение тайм-аута может быть указано в секундах или в виде значения времени:

```
wait [port1 port2 10]
wait [port1 port2 0:00:05]
```

В первом примере время ожидания истекает через десять секунд. Во втором примере тайм-аут будет через пять секунд.

Функция **wait** вернет порт, который готов или none, если истекло время ожидания.

```
ready: wait [port1 port2 10]
if ready [data: copy ready]
```

В приведенном выше примере данные будут считываться из первого готового порта, если тайм-аут не наступил.

Чтобы получить блок всех готовых портов, используйте уточнение **/all**.

```
ready: wait/all [port1 port2 10]
if ready [
    foreach port ready [
        append data copy port
    ]
]
```

В этом примере данные со всех готовых портов будут добавлены в одну серию.

Вы также можете использовать функцию отправки (**dispatch**) для оценки блока или функции на основе результатов ожидания (**wait**) на нескольких портах.

```
dispatch [
    port1 [print "port1 awake"]
    port2 [print "port2 awake"]
    10 [print "timeout!"]
]
```

Используйте /No-wait и /Direct

Чтобы использовать **wait** с большинством портов, вам нужно будет указать уточнения / no-wait и / direct как часть open. Это указывает на то, что обычные функции доступа к данным не должны блокироваться и что данные не буферизуются.

```
port1: open/no-wait/direct tcp://system:8000
```

8. Другие режимы порта

8.1 Линейный режим

Функция **open** позволяет открывать порты для линейного доступа. В строчном режиме первая функция вернет строку текста, а не символ. В приведенном ниже примере файл читается по одной строке за раз:

```
fp: open/lines %file.txt
print first fp
I wanted the gold, and I got it --
print third fp
Yet somehow life's not what I thought it,
```

Уточнение **/lines** также полезно для интернет-протоколов, ориентированных на линию (строк).

```
tp: open/lines tcp://server:8000
print first tp
```

8.2 Только чтение и запись

Вы можете использовать уточнение **/read**, чтобы открыть порт только для чтения:

```
fp: open/read %file.txt
```

Изменения, внесенные в буфер порта, не записываются обратно в файл.
Чтобы открыть только для записи, используйте уточнение **/write**
:

```
fp: open/write %file.txt
```

Файловые порты, открытые с уточнением **/write**, не будут читать текущие данные при открытии порта.

Закрытие или обновление файлового порта только для записи приведет к перезаписи существующих данных в файле:

```
insert fp "This is the law of the Yukon..."
close fp
print read %file.txt
This is the law of the Yukon...
```

8.3 Прямой доступ к порту

Уточнение **/direct** открывает небуферизованный порт. Это полезно для доступа к файлам по частям, например, когда файл слишком велик для хранения в памяти.

```
fp: open/direct %file.txt
```

Чтение данных с помощью функции **copy** переместит голову указателя порта вперед:

```
print copy/part fp 40
I wanted the gold, and I sought it,^/ I
print copy/part fp 40
scrabbled and mucked like a slave.^/Was i
```

В прямом режиме указатель порт всегда будет в исходном положении:

```
print head? fp
true
```

Функция **copy** вернет none, когда порт достигнет своего конца.

Вот пример, который использует прямые порты для копирования файла любого размера:

```
from-port: open/direct %a-file.jpg
to-port: open/direct %a-file.jpg
while [data: copy/part from-port 100000 ][
    append to-port data
]
close from-port
close to-port
```

8.4 Пропуск данных

Есть два способа пропустить данные, существующие в порту. Во-первых, вы можете открыть порт с уточнением **/skip**. Эта функция **open** автоматически перейдет к определенной точке порта. Например:

```
fp: open/direct/skip %file.big 1000000
fp: open/skip http://www.example.com/bigfile.dat 100000
```

Вы также можете использовать функцию **skip** порта. Для файлов, которые открываются с помощью **/direct** и **/binary**, операция пропуска идентична операции поиска в файловой системе. Данные не считываются в память. Это невозможно в режиме **/string**, потому что разрывы строк влияют на размер пропуска.

```
fp: open/direct/binary %file.dat
fp: skip fp 100000
```

9. Права доступа к файлам

Когда файлы создаются REBOL, устанавливаются права доступа по умолчанию. В системах Windows и Macintosh файлы создаются с правами полного доступа. В системах UNIX файлы создаются с разрешениями, установленными на текущую настройку umask.
При использовании `open` или `write` для доступа к файлу уточнение `/allow` используется для установки разрешений на доступ к файлу.
Уточнение `/allow` принимает блок в качестве аргумента. Этот блок может состоять из любого или всех трех слов `read`, `write` и `execute`.

Ограничения операционной системы

Уточнение `/allow` только установить разрешения на операционных системах , поддерживающих заданную настройку разрешений. Если операционная система не поддерживает используемый параметр разрешений, он будет проигнорирован. Например, файлы в системах UNIX могут быть установлены как исполняемые (`execute`), но операционные системы Windows и Macintosh не поддерживают это. При работе с системами UNIX разрешения, установленные с помощью `/allow`, будут устанавливать только права пользователя. Использование `/allow` приведет к удалению всех прав доступа для пользователей и других лиц.

Чтобы сделать файл доступным только для чтения, используйте `open/allow` или `write/allow` с блоком чтения.

```
write/allow %file.txt [read]
```

Чтобы сделать файл читаемым и исполняемым:

```
open/allow %file.txt [read execute]
```

Вы можете установить аналогичные разрешения для доступа на запись:

```
write/allow %file.txt [read write]
```

Чтобы предотвратить любой доступ к файлу (для операционных систем, где это имело бы значение), предоставьте пустой блок разрешений:

```
write/allow %file.txt []
```

Чтобы разрешить полный доступ:

```
write/allow %file [read write execute]
```

10. Порты каталогов

Порты каталогов позволяют открывать прямой доступ к каталогам файлов. В системе именно так создается большинство других функций каталогов.
Когда вы открываете каталог, вы получаете прямой доступ к нему в виде блока имен файлов:

```
mydir: open %intro/
forall mydir [print first mydir]
CVS/
history.t
intro.t
overview.t
quick.t
close mydir
```

Вы можете перейти к определенной позиции в серии каталогов и удалить файл с таким кодом, как:

```
dir: open %.
remove next dir
close dir
```

Это удаляет второй файл в текущем каталоге. По аналогии,

```
remove at dir 5
```

удалит пятый файл в каталоге и:

```
clear dir
```

удалит все файлы в каталоге.

Чтобы удалить все файлы, содержащие слово "junk", вы можете написать:

```
dir: open %intro/
while [not tail? dir] [
    either find first dir "junk" [remove dir][
        dir: next dir
    ]
]
close dir
```

Изменения, внесенные в каталог, производятся при закрытии каталога или при его обновлении. Чтобы действие было выполнено немедленно, используйте такую строку, как:

```
update dir
```

Метод доступа к каталогам также может использоваться для изменения имен файлов. После **open** строка:

```
change at dir 3 %newname.txt
```

переименует третий файл в каталоге. Аналогичным образом можно изменить имена любого из файлов в каталоге.
Вот пример, который переименовывает все файлы в каталоге, добавляя слово REBOL к их именам:

```
dir: open %intro/
forall dir [insert first dir "REBOL"]
close dir
```

Глава 15 - Разбор

Содержание:

[1. Обзор]
[2. Простое разбиение]
[3. Правила грамматики]
[4. Пропуск ввода]
[5. Типы соответствий]
[6. Рекурсивные правила]
[7. Оценка]
7.1 Возвращаемое значение
7.2 Выражения в правилах
7.3 Копирование ввода
7.4 Пометка ввода
7.5 Изменение строки
7.6 Использование объектов
7.7 Отладка
[8. Работа с пробелами]
[9. Блоки синтаксического анализа и диалекты]
9.1 Сравние слов
9.2 Сравнение дат
9.3 Недопустимые символы
9.4 Примеры диалектов
9.5 Анализ субблоков
[10. Сводка операций синтаксического анализа]
10.1 Основные формы
10.2 Указание количества
10.3 Пропуск значений
10.4 Получение значений
10.5 Использование слов
10.6 Соответствие значений (только анализ блока)
10.7 Слова типа данных

1. Обзор

При синтаксическом анализе последовательность символов или значений разбивается на более мелкие части. Его можно использовать для распознавания символов или значений, которые встречаются в определенном порядке. Помимо предоставления мощного, удобочитаемого и удобного в обслуживании подхода к сопоставлению с образцом регулярных выражений, синтаксический анализ позволяет вам создавать собственные настраиваемые языки для конкретных целей.

Функция `parse` имеет общий вид:

```
parse series rules
```

`Серия` аргумента является входом, который разбирается и может быть строка или блок. Если аргумент является строкой, он анализируется по символам. Если аргумент является блоком, он анализируется по значению.

Правила аргумента определяет , как серия будет разобранный аргумент. Правила аргумент может быть строкой для простых типов разбора или блока для сложного анализа.

Функция **parse** также принимает два уточнения: **/all** и **/case**. Уточнение **/all** разбирает все символы в строке, включая все разделители, такие как пробел, табуляция, перевод строки, запятой и точкой с запятой. Уточнение **/case** разбирает строку, основанную на случае. Если **/case** не указан, верхний и нижний регистры обрабатываются одинаково.

2. Простое разделение

Простая форма **parse** предназначена для разделения строк:

```
parse string none
```

Функция **parse** разбивает входной аргумент, строку, на блок из нескольких строк, разбивая каждую строку там, где она встречается с разделителем, например пробелом, табуляцией, новой строкой, запятой или точкой с запятой. Аргумент none указывает на то, что никакие другие разделители кроме них. Например:

```
probe parse "The trip will take 21 days" none
["The" "trip" "will" "take" "21" "days"]
```

По аналогии,

```
probe parse "here there,everywhere; ok" none
["here" "there" "everywhere" "ok"]
```

Обратите внимание, что в приведенном выше примере из результирующих строк были удалены запятые и точки с запятой.

Вы можете указать свои собственные разделители во втором аргументе для синтаксического анализа (**parse**). Например, следующий код анализирует номер телефона с разделителями тире (-):

```
probe parse "707-467-8000" "-"
["707" "467" "8000"]
```

В следующем примере в качестве разделителей используется равенство (=) и двойная кавычка ("):

```
probe parse <IMG SRC="test.gif" WIDTH="123"> {="}
["IMG" "SRC" "test.gif" "WIDTH" "123"]
```

В следующем примере строка анализируется только на основе запятых; любые другие разделители игнорируются. Следовательно, пробелы внутри строк не удаляются:

Обычно при синтаксическом анализе строк любые пробелы (пробел, табуляция, строки) автоматически обрабатываются как разделители. Чтобы избежать этого действия, вы можете использовать уточнение /any.

Сравните эти два примера:

```
parse "Test This" ""
["Test" "This"]
parse/all "Test This" ""
["Test This"]
```

Во втором случае вы можете видеть, что пробел не рассматривался как разделитель.

Вот еще один пример:

```
probe parse/all "Harry, 1011 Main St., Ukiah" ","
["Harry" " 1011 Main St." " Ukiah"]
```

Вы также можете анализировать строки, содержащие нулевые символы в качестве разделителей (например, определенные типы файлов данных):

```
parse/all nulled-string "^(null)"
```

3. Грамматические правила

Функция `parse` принимает грамматические правила, написанные на ***диалекте*** REBOL. Диалекты - это подъязыки REBOL, которые используют одну и ту же лексическую форму для всех типов данных, но допускают различный порядок значений внутри блока. В рамках этого диалекта грамматика и словарь REBOL изменены, чтобы сделать его похожим по структуре на хорошо известную BNF (Backus-Naur Form), которая обычно используется для определения языковых грамматик, сетевых протоколов, форматов заголовков и т.д..

Чтобы определить правила, используйте блок, чтобы указать последовательность входных данных. Например, если вы хотите проанализировать строку и вернуть символы "the phone", вы можете использовать правило:

```
parse string ["the phone"]
```

Чтобы разрешить любое количество пробелов или отсутствие пробелов между словами, напишите правило следующим образом:

```
parse string ["the" "phone"]
```

Вы можете указать альтернативные правила с помощью вертикальной черты (|). Например:

```
["the" "phone" | "a" "radio"]
```

принимает строки, соответствующие любому из следующего:

```
the phone
a radio
```

Правило может содержать блоки, которые рассматриваются как вспомогательные правила. Следующая строка:

```
[ ["a" | "the"] ["phone" | "radio"] ]
```

принимает строки, соответствующие любому из следующего:

```
a phone
a radio
the phone
the radio
```

Для повышения удобочитаемости запишите подправила как отдельный блок и дайте им имя, которое поможет указать их цель:

```
article: ["a" | "the"]
device: ["phone" | "radio"]
parse string [article device]
```

В дополнение к сопоставлению одного экземпляра строки вы можете указать счетчик или диапазон, который повторяет совпадение. В следующем примере показано количество:

```
[3 "a" 2 "b"]
```

который принимает строки, которые соответствуют:

```
aaabb
```

В следующем примере представлен диапазон:

```
[1 3 "a" "b"]
```

который принимает строки, соответствующие любому из следующего:

```
ab aab aaab
```

Начальная точка диапазона может быть нулевой, что означает, что это необязательно.

```
[0 3 "a" "b"]
```

принимает строки, соответствующие любому из следующего:

```
b ab aab aaab
```

Используйте **some**, чтобы указать, что совпадают один или несколько символов. Используйте **any**, чтобы указать, что совпадают ноль или более символов. Например, **some** используются в следующей строке:

```
[some "a" "b"]
```

принимает строки, содержащие один или несколько символов `a` и `b:`:

```
ab aab aaab aaaab
```

В следующем примере используются **any**:

```
[any "a" "b"]
```

который принимает строки, содержащие ноль или более символов `a` или `b`:

```
b ab aab aaab aaaab
```

Слова **some** и **any** также могут использоваться в блоках. Например:

```
[some ["a" | "b"]]
```

принимает строки, содержащие любую комбинацию символов `a` и `b`.

Другой способ выразить необязательность символа - предоставить альтернативный вариант `none`:

```
["a" | "b" | none]
```

Этот пример принимает строки , которые содержат `a` или `b` или `none`.

Параметр `none` полезен для указания дополнительных шаблонов или для выявления случаев ошибок, когда ни один шаблон не совпадает.

4. Пропуск ввода

Skip, **to** и **thru** слова разрешают вводу быть пропущен.

Используйте **skip** (пропустить), чтобы пропустить один символ, или используйте его с **repeat** (повторять), чтобы пропустить несколько символов:

```
["a" skip "b"]
["a" 10 skip "b"]
["a" 1 10 skip]
```

Чтобы пропустить, пока не будет найден конкретный символ, используйте **to:** (для):

```
["a" to "b"]
```

В предыдущем примере анализ начинается с `a` и заканчивается на `b`, но не включает `b`.

Чтобы включить конечный символ, используйте **thru** (через):

```
["a" thru "b"]
```

В предыдущем примере анализ начинается с `a`, заканчивается на `b` и включает `b`.

Следующее правило находит заголовок HTML-страницы и распечатывает его:

```
page: read http://www.rebol.com/
parse page [thru <title> copy text to </title>]
print text
REBOL Technologies
```

Первый **thru** находит тег заголовка (title) и сразу проходит мимо него. Затем входная строка копируется в переменную с именем `text` до тех пор, пока не будет найден конечный тег (/title) (но он не пройдет мимо него, иначе текст будет включать тег).

5. Типы соответствия

При синтаксическом анализе строк эти типы данных и слова могут использоваться для сопоставления символов во входной строке:

Тип соответствия	Описание
"abc"	соответствовать всей строке
#"c"	соответствовать одному символу
tag	сопоставить строку тега
end	совпадать с концом ввода
(bitset)	соответствует любому указанному символу в наборе

Чтобы использовать все эти слова (кроме bitset, который объясняется ниже) в одном правиле, используйте:

```
[<B> ["excellent" | "incredible"] #"!" </B> end]
```

В этом примере анализируются входные строки:

```
<B>excellent!</B>
<B>incredible!</B>
```

В **end** указывает, что ничего не следует во входном потоке. Все входные данные были проанализированы. Это необязательно в зависимости от того, нужно ли проверять возвращаемое значение функции **parse**. Обратитесь к разделу "Оценка" ниже для получения дополнительной информации.

Тип данных **bitset** заслуживает более подробного объяснения. Битовые наборы используются для эффективного задания наборов символов. Функция **charset** позволяет вам указывать отдельные символы или диапазоны символов. Например, строка:

```
digit: charset "0123456789"
```

определяет набор символов, содержащий цифры. Это допускает такие правила, как:

```
[3 digit "-" 3 digit "-" 4 digit]
```

который может анализировать телефонные номера в форме:

```
707-467-8000
```

Чтобы принять любое количество цифр, обычно пишут правило:

```
digits: [some digit]
```

Набор символов (character) также может указывать диапазоны символов. Например, набор символов `digit` можно было бы записать как:

```
digit: charset [#"0" - #"9"]
```

В качестве альтернативы вы можете комбинировать определенные символы и диапазоны символов:

```
the-set: charset ["+-." #"0" - #"9"]
```

Чтобы расширить это, вот буквенно-цифровой набор символов:

```
alphanum: charset [#"0" - #"9" #"A" - #"Z" #"a" - #"z"]
```

Наборы символов также могут быть изменены с помощью функций **insert** (вставить) и **remove** (удалить), или комбинации наборов могут быть созданы с помощью функций **union** (объединение) и **intersect** (пересечение). Эта строка копирует набор символов цифр и добавляет к нему точку:

```
digit-dot: insert copy digit "."
```

Следующие строки определяют полезные наборы символов для синтаксического анализа:

```
digit: charset [#"0" - #"9"]
alpha: charset [#"A" - #"Z" #"a" - #"z"]
alphanum: union alpha digit
```

6. Рекурсивные правила

Вот пример набора правил, который анализирует математические выражения и дает приоритет (приоритет) используемым математическим операторам:

```
expr:    [term ["+" | "-"] expr | term]
term:    [factor ["*" | "/"] term | factor]
factor:  [primary "**" factor | primary]
primary: [some digit | "(" expr ")"]
digit:   charset "0123456789"
```

Теперь мы можем анализировать многие типы математических выражений. Следующие примеры возвращают `true` (истину), указывая, что выражения действительны:

```
probe parse "1 + 2 * ( 3 - 2 ) / 4" expr
true
probe parse "4/5+3**2-(5*6+1)" expr
true
```

Обратите внимание, что в примерах некоторые правила относятся к самим себе. Например, `expr` правило включает в себя `expr` . Это полезный метод определения повторяющихся последовательностей и комбинаций. Правило ***recursive*** (рекурсивное) - оно относится к самому себе.

При использовании рекурсивных правил необходимо соблюдать осторожность, чтобы предотвратить бесконечную рекурсию. Например:

```
expr: [expr ["+" | "-"] term]
```

создает бесконечный цикл, потому что первое, что делает `expr`, - снова использует `expr`.

7. Оценка

Обычно вы анализируете строку, чтобы получить какой-то результат. Вы хотите сделать больше, чем просто проверить правильность строки, вы хотите что-то сделать во время ее анализа. Например, вы можете выбрать подстроки из различных частей строки, создать блоки связанных значений или вычислить значение.

7.1 Возвращаемое значение

Примеры в предыдущих главах показали, как анализировать строки, но результатов не было. Это делается только для проверки того, что строка соответствует указанной грамматике; значение, возвращаемое cb>`parse`, указывает на его успех. Следующие примеры показывают это:

```
probe parse "a b c" ["a" "b" "c"]
true
probe parse "a b" ["a" "c"]
false
```

Функция **`parse`** возвращает `true` (истину), только если достигает конца входной строки. Неудачное совпадение останавливает разбор серии. Если с**`parse`** исчерпывает значения для поиска до достижения конца серии, он не просматривает серию и возвращает **`false`**:

```
probe parse "a b c d" ["a" "b" "c"]
false
probe parse "a b c d" [to "b" thru "d"]
true
probe parse "a b c d" [to "b" to end]
true
```

7.2 Выражения в правилах

Внутри правила вы можете включить выражение REBOL, которое будет оцениваться, когда **`parse`** достигает этой точки в правиле. Круглые скобки используются для обозначения таких выражений:

```
string: "there is a phone in this sentence"
probe parse string [
    to "a"
    to "phone" (print "found phone")
    to end
]
found phone
true
```

В приведенном выше примере анализируется строка `a phone` и печатается сообщение `found phone` после завершения сопоставления. Если строки `a` или `phone` отсутствуют и анализ не может быть выполнен, выражение не оценивается.

Выражения могут появляться в любом месте правила, а несколько выражений могут встречаться в разных частях правила. Например, следующий код печатает разные строки в зависимости от того, какие входные данные были найдены:

```
parse string [
    "a" | "the"
    to "phone" (print "answer") |
    to "radio" (print "listen") |
    to "tv"    (print "watch")
]
answer
string: "there is the radio on the shelf"

parse string [
    "a" | "the"
    to "phone" (print "answer") |
    to "radio" (print "listen") |
    to "tv"    (print "watch")
]
listen
```

Вот пример, который подсчитывает, сколько раз тег предварительного форматирования HTML появляется в текстовой строке:

```
count: 0
page: read http://www.rebol.com/docs/dictionary.html
parse page [any [thru <pre> (count: count + 1)]]
print count
777
```

7.3 Копирование ввода

Наиболее распространенное действие, выполняемое с помощью **parse**, - это выбор части анализируемой строки. Это делается с помощью **copy**, за которым следует имя переменной, в которую вы хотите скопировать строку. В следующем примере анализируется заголовок веб-страницы:

```
parse page [thru <title> copy text to </title>]
print text
REBOL/Core Dictionary
```

В этом примере текст пропускается до тех пор, пока не будет найден тег <title>. Вот где он начинает делать копию входного потока и устанавливать переменную с именем `text` для его хранения. Операция копирования продолжается до тех пор, пока не будет найден закрывающий тег <title>.

Действие копирования также можно использовать с целыми блоками правил. Например, для правила:

```
[copy heading ["H" ["1" | "2" | "3"]]
```

строка заголовка содержит всю строку `H1`, `H2` или `H3`. Это также работает для больших правил с несколькими блоками.

7.4 Маркировка ввода

Действие **copy** создает копию найденной подстроки, но это не всегда желательно. В некоторых случаях лучше сохранить текущую позицию входного потока в переменной.

ПРИМЕЧАНИЕ. Слово **copy**, используемое при синтаксическом анализе, отличается от функции **copy**, используемой в выражениях REBOL. Анализировать использует диалект REBOL, и **copy** имеет различное значение в пределах этого диалекта.

В следующем примере переменная `begin` содержит ссылку на строку ввода `page` сразу после <title>. `Ending` ссылается на `page` строки непосредственно перед <title>. Эти переменные можно использовать так же, как и с любыми другими сериями.

```
parse page [
    thru <title> begin: to </title> ending:
    (change/part begin "Word Reference Guide" ending)
]
```

Вы можете видеть, что приведенное выше выражение синтаксического анализа фактически изменило содержимое заголовка:

```
parse page [thru <title> copy text to </title>]
print text
Word Reference Guide
```

Вот еще один пример, который отмечает позицию каждого тега таблицы в файле HTML:

```
page: read http://www.rebol.com/index.html
tables: make block! 20
parse page [
    any [to "<table" mark: thru ">"
        (append tables index? mark)
    ]
]
```

Блок `tables` теперь содержит позицию каждого тега:

```
foreach table tables [
    print ["table found at index:" table]
]
table found at index: 836
table found at index: 2076
table found at index: 3747
table found at index: 3815
table found at index: 4027
table found at index: 4415
table found at index: 6050
table found at index: 6556
table found at index: 7229
table found at index: 8268
```

ПРИМЕЧАНИЕ. Текущая позиция во входной строке также может быть изменена. В следующем разделе объясняется, как это делается.

7.5 Изменение строки

Теперь, когда вы знаете, как получить позицию входного ряда, вы также можете использовать для него другие функции ряда, включая вставку (**insert**), удаление (**remove**) и изменение (**change**). Чтобы написать сценарий, в котором все вопросительные знаки (?) Заменяются восклицательными знаками (!), Напишите:

```
str: "Where is the turkey? Have you seen the turkey?"
parse str [some [to "?" mark: (change mark "!") skip]]
print str
Where is the turkey! Have you seen the turkey!
```

Skip в хвосте опережает ввод над новым персонажем, который не является необходимым в этом случае, но это хорошая практика.
В качестве другого примера, чтобы вставить текущее время везде, где в тексте встречается слово `time` (время), напишите:

```
str: "at this time, I'd like to see the time change"
parse str [
    some [to "time"
        mark:
        (remove/part mark 4  mark: insert mark now/time)
        :mark
    ]
]
print str
at this 14:42:12, I'd like to see the 14:42:12 change
```

Обратите внимание на слово `:mark`, использованное выше. Он устанавливает вход в новую позицию. Функция **insert** возвращает новую позицию сразу после вставки текущего времени. Слово `:mark` используется для установки ввода в эту позицию.

7.6 Использование объектов

При разборе большой грамматики из набора правил используются переменные, чтобы сделать грамматику более читаемой. Однако переменные являются глобальными и могут быть перепутаны с другими переменными, имеющими такое же имя где-то еще в программе.
Решение этой проблемы - использовать объект, чтобы сделать все слова правила локальными для контекста. Например:

```
tag-parser: make object! [
    tags: make block! 100
    text: make string! 8000
    html-code: [
        copy tag ["<" thru ">"] (append tags tag) |
        copy txt to "<" (append text txt)
    ]
    parse-tags: func [site [url!]] [
        clear tags clear text
        parse read site [to "<" some html-code]
        foreach tag tags [print tag]
        print text
    ]
]
tag-parser/parse-tags http://www.rebol.com
```

7.7 Отладка

Поскольку правила написаны, иногда требуется отладка. В частности, вы можете захотеть узнать, как далеко вы продвинулись в синтаксическом анализе правила.

Функция **trace** может быть использована для наблюдения за прогрессом операции синтаксического анализа, но это может выводить тысячи строк, которые трудно рассмотреть.

Лучше всего вставить отладочные выражения в правила синтаксического анализа. Например, для отладки правила:

```
[to "<IMG" "SRC" "=" filename ">"]
```

вставьте функцию **print** после ключевых разделов, чтобы следить за тем, как вы выполняете правило:

```
[to "<IMG" (print 1) "SRC" "=" (print 2)
    filename (print 3) ">"]
```

В этом примере печатаются 1, 2 и 3 по мере обработки правила.

Другой подход - распечатать часть входной строки по мере выполнения синтаксического анализа:

```
[
   to "<IMG" here: (print here)
   "SRC" "=" here: (print here)
    filename here: (print here) ">"
]
```

Если это делается часто, вы можете создать для этого правило:

```
here: [where: (print where)]

[
   to "<IMG" here
   "SRC" "=" here
    filename here ">"
]
```

Функция **copy** также может использоваться, чтобы указать, какие подстроки были проанализированы при обработке правила.

8. Работа с пространствами

Функция **parse** обычно игнорирует все пробелы между сканируемыми шаблонами. Например, правило:

```
["a" "b" "c"]
```

возвращает строки, которые соответствуют:

```
abc
a bc
ab c
a b c
a  b  c
```

и другие аналогично расположенные комбинации.
Чтобы обеспечить соблюдение определенного соглашения об интервале, используйте b>`parse` с уточнением **`/all`**. В предыдущем примере это уточнение заставляет **`parse`** соответствовать только первому регистру (abc).

```
parse/all "abc" ["a" "b" "c"]
```

Указание **`/all`** уточнения заставляют обрабатывать каждый символ во входном потоке, включая разделители по умолчанию, такие как пробел, табуляция, новая строка.
Чтобы обрабатывать пробелы в ваших правилах, создайте набор символов, который определяет допустимые символы пробела:

```
spacer: charset reduce [tab newline #" "]
```

Если вы хотите, чтобы между каждой буквой был один пробел, напишите:

```
["a" spacer "b" spacer "c"]
```

Чтобы разрешить использование нескольких пробелов, напишите:

```
spaces: [some spacer]
["a" spaces "b" spaces "c"]
```

Для более сложных грамматик создайте набор символов, который позволяет сканировать строку до символа пробела.

```
non-space: complement spacer
to-space: [some non-space | end]
words: make block! 20
parse/all text [
    some [copy word to-space (append words word) spacer]
]
```

В предыдущем примере строится блок из всех слов. Функция **`complement`** инвертирует набор символов. Теперь он содержит все, кроме пробелов (`non-space`), которые вы определили ранее. Набор символов без пробела содержит все символы, кроме символов пробела. `to-space` правило принимает один или несколько символов до символа пробела или конца входного потока. Основное правило предполагает, что нужно начинать со слова, копировать это слово до пробела, затем пропускать пробел и начинать следующее слово.

9. Блоки синтаксического анализа и диалекты

Блоки анализируются аналогично строкам. Набор правил определяет порядок ожидаемых значений. Однако, в отличие от синтаксического анализа строк, синтаксический анализ блоков не связан с символами или разделителями. Анализ блоков выполняется на уровне значений, что упрощает определение правил грамматики и во много раз ускоряет работу.

Блочный синтаксический анализ - это самый простой способ создания ***диалектов*** REBOL . Диалекты - это подъязыки REBOL, которые используют одну и ту же лексическую форму для всех типов данных, но допускают различный порядок значений в блоке. Значения не обязательно должны соответствовать нормальному порядку, требуемому аргументами функции REBOL. Диалекты способны обеспечить большую выразительность в определенных областях использования. Например, сами правила парсера указаны как диалект.

9.1 Соответствующие слова

При синтаксическом разборе блока для сопоставления со словом укажите слово как литерал:

```
'name
'when
'empty
```

9.2 Соответствие типов данных

Вы можете сопоставить значение любого типа данных, указав слово типа данных. См. Раздел «Соответствие типов данных» ниже.

Тип данных слова	Описание
string!	соответствует любой строке в кавычках
time!	соответствует в любому времени
date!	соответствует любой дате
tuple!	соответствует любому кортежу

ПРИМЕЧАНИЕ: не забывайте "!" это часть имени, иначе будет вызвона ошибка.

9.3 Запрещенные символы

Для блоков разрешены операции `parse` которые имеют дело с конкретными символами. Например, нельзя указать совпадение ни с первой буквой слова или строки, ни с пробелами или символами новой строки.

9.4 Примеры диалектов

Несколько кратких примеров помогают проиллюстрировать парсинг блоков:

```
block: [when 10:30]
print parse block ['when 10:30]
print parse block ['when time!]
parse block ['when set time time! (print time)]
```

Обратите внимание, что конкретное слово можно сопоставить, используя его буквальное слово в правиле (как в случае `'when`). Можно указать тип данных, а не значение, как в строках выше, содержащих **`time!`**. Кроме того, переменной можно присвоить значение с помощью операции **`set`**. Как и в случае со строками, при синтаксическом анализе блоков можно указать альтернативные правила:

```
rule: [some [
    'when set time time! |
    'where set place string! |
    'who set persons [word! | block!]
]]
```

Эти правила позволяют вводить информацию в любом порядке:

```
parse [
    who Fred
    where "Downtown Center"
    when 9:30
] rule
print [time place persons]
```

В этом примере можно было бы использовать присвоение переменных, но он показывает, как обеспечить альтернативный порядок ввода.
Вот еще один пример, оценивающий результаты синтаксического анализа:

```
rule: [
    set count integer!
    set str string!
    (loop count [print str])
]
parse [3 "great job"] rule
parse [3 "hut" 1 "hike"] [some rule]
```

Наконец, вот более сложный пример:

```
rule: [
    set action ['buy | 'sell]
    set number integer!
    'shares 'at
    set price money!
    (either action = 'sell [
            print ["income" price * number]
            total: total + (price * number)
        ][
            print ["cost" price * number]
            total: total - (price * number)
        ]
    )
]
total: 0
parse [sell 100 shares at $123.45] rule
print ["total:" total]
total: 0
parse [
    sell 300 shares at $89.08
    buy  100 shares at $120.45
    sell 400 shares at $270.89
] [some rule]
print ["total:" total]
```

Следует отметить, что это один из способов ***оценки*** выражений, использующих концепцию ***диалекта***, впервые описанную в главе 4.

9.5 Разбор субблоков

Если при синтаксическом анализе блока найден подблок, он обрабатывается как одно значение блока - тип данных **block!**. Однако для синтаксического анализа субблока необходимо рекурсивно вызывать синтаксический анализатор для этого субблока. Слове **into** обеспечивает эту возможность. Ожидается, что следующее значение во входном блоке будет подблоком для анализа. Это как бы тип данных **block!** был предоставлен. Если **into** значение не тип данных **block!**, сравнение не удается, и в поиск чередуется или выходит из этого правила. Если следующим значением является блок, то правило синтаксического анализатора, которое следует за словом **into**, используется для начала синтаксического анализа субблока. Он обрабатывается так же, как и подправило.

```
rule: [date! into [string! time!]]
data: [10-Jan-2000 ["Ukiah" 10:30]]
print parse data rule
```

Все обычные операции синтаксического анализа могут быть применены к **into**.

```
rule: [
    set date date!
    set info into [string! time!]]
]
data: [10-Jan-2000 ["Ukiah" 10:30]]
print parse data rule

print info

rule: [date! copy items 2 into [string! time!]]
data: [10-Jan-2000 ["Ukiah" 10:30] ["Rome" 2:45]]
print parse data rule

probe items
```

10. Краткое описание операций синтаксического анализа

10.1 Общие формы

Оператор	Описание
|	альтернативное правило
[блок]	подправило
(выражение)	оценить выражение REBOL

10.2 Указание количества

Оператор	Описание
none	ничего не совпадает
opt	ноль или один раз
some	один или несколько раз
any	ноль или более раз
12	повторить шаблон 12 раз
1 12	повторить шаблон от 1 до 12 раз
0 12	повторить шаблон от 0 до 12 раз

10.3 Пропуск значений

Оператор	Описание
skip	пропустить значение (или несколько, если задано повторение)
to	предварительный ввод значения или типа данных
thru	предварительный ввод через значение или тип данных

10.4 Получение значений

Оператор	Описание
set	установить следующее значение переменной
copy	скопировать следующую последовательность совпадений в переменную

10.5 Использование слов

Оператор	Описание
word	значение слова
word:	отметить текущую позицию входной серии
:word	установить текущую позицию входной серии
'word	буквально соответствует слову (блок синтаксического анализа)

10.6 Соответствие значений (только анализ блока)

Оператор	Описание
"fred"	совпадает со строкой "fred"
%data	совпадает с именем файла %data
10:30	совпадает с временем 10:30
1.2.3	соответствует кортежу 1.2.3

10.7 Слова типа данных

Слово	Описание
type!	соответствует чему-либо с заданным типом данных

Приложение 1 - Значения

Содержание:

1. Числовые значения
- 1.1 Десятичные
- 1.2 Целое число

2. Серии значений
- 2.1 Двоичный
- 2.2 Блок
- 2.3 Электронная почта
- 2.4 Файл
- 2.5 Хэш
- 2.6 Изображение
- 2.7 Проблема
- 2.8 Список
- 2.9 Парен
- 2.10 Путь
- 2.11 Строка
- 2.12 Тег
- 2.13 URL

3. Другие значения
- 3.1 Символ
- 3.2 Дата
- 3.3 Логика
- 3.4 Деньги
- 3.5 None
- 3.6 Пара
- 3.7 Уточнение
- 3.8 Время
- 3.9 Кортеж
- 3.10 Слова

1. Числовые значения

1.1 Десятичный

1.1.1 Концепция

Тип данных `decimal!` основан на 64-битных стандартных числах с плавающей запятой IEEE. Они отличаются от целых чисел десятичной точкой (точка или запятая разрешены для международного использования, см. Примечания ниже).

1.1.2 Формат

Десятичные значения представляют собой последовательность числовых цифр, за которыми следует десятичная точка, которая может быть точкой (.) или запятой (,), за которой следует несколько цифр. Знак плюс (+) или минус (-) непосредственно перед первой цифрой обозначает знак числа. Начальные нули перед десятичной точкой игнорируются. Лишние пробелы, запятые и точки не допускаются.

```
1.23
123.
123.0
0.321
0.123
1234.5678
```

Вместо точки можно использовать запятую для обозначения десятичной точки (что является стандартом во многих странах):

```
1,23
0,321
1234,5678
```

Используйте одинарные кавычки (`) для разделения цифр в длинных десятичных дробях. Одиночные кавычки могут появляться в любом месте после первой цифры числа, но не перед первой цифрой.

```
100'234'562.3782
100'234'562,3782
```

Не используйте запятые или точки для разделения цифр в десятичном значении.

Научная нотация может использоваться для указания степени числа, добавляя к числу букву **E** или **e**, за которой следует последовательность цифр. Показатель степени может быть положительным или отрицательным числом.

```
1.23E10
1.2e007
123.45e-42
56,72E300
-0,34e-12
0.0001e-001
```

Десятичные числа варьируются от 2,2250738585072e-308 до 1,7976931348623e + 308 и могут содержать до 15 цифр точности.

1.1.3 Создание

Используйте функцию `to-decimal` (в-десятичное) для преобразования типов данных `string!` (строка!), `integer!` (целое!), `block!` (блок!), или `decimal!` (десятичное) в десятичное число:

```
probe to-decimal "123.45"
123.45
probe to-decimal 123
123
probe to-decimal [-123 45]
-1.23E+47
probe to-decimal [123 -45]
1.23E-43
probe to-decimal -123.8
-123.8
probe to-decimal 12.3
12.3
```

Если в выражении объединены десятичное и целое число, целое число преобразуется в десятичное число:

```
probe 1.2 + 2
3.2
probe 2 + 1.2
3.2
probe 1.01 > 1
true
probe 1 > 1.01
false
```

1.1.4 Связанные

Используйте фенкцию **decimal?** (десятичное?) чтобы определить, является ли значение типа данных **decimal!** (десятичное!).

```
print decimal? 0.123
true
```

Используйте функции **form**, **print** и **mold** с целочисленным аргументом, чтобы распечатать десятичное значение в его простейшей форме:

- целое число. Если его можно представить как единое целое.
- десятичная дробь без экспоненты. Если он не слишком большой или слишком маленький.
- научная нотация. Если он слишком большой или маленький.

Например,

```
probe mold 123.4
123.4
probe form 2222222222222222
2.22222222222222E+15
print 1.00001E+5
100001
```

Одиночные кавычки (`) и ведущий знак плюс (+) не отображаются в десятичном выводе:

```
print +1'100'200.222'112
1100200.222112
```

1.2 Целое число

1.2.1 Концепция

Тип данных **integer!** включает 32-битные положительные и отрицательные числа и ноль. В отличие от десятичных чисел, целые числа не содержат десятичной точки.

1.2.2 Формат

Целочисленные значения состоят из последовательности цифр. Знак плюс (+) или минус (-) непосредственно перед первой цифрой обозначает знак. Между знаком и первой цифрой не может быть пробела. Начальные нули игнорируются.

```
0 1234 +1234 -1234 00012 -0123
```

Не используйте запятые или точки в целых числах. Если запятая или точка находятся внутри целого числа, это интерпретируется как десятичное значение. Однако вы можете использовать одиночные кавычки (`) для разделения цифр в длинных целых числах. Одиночные кавычки могут появляться в любом месте после первой цифры числа, но не перед первой цифрой.

```
2'147'483'647
```

Целые числа охватывают диапазон от -2147483648 до 2147483647.

1.2.3 Создание

Используйте функцию **to-integer** (в-целое) для преобразования **string!** (строка!), **logic!** (логическле!), **decimal!** (десятичное!), или **integer!** (целое!) типов данных к целому числу:

```
probe to-integer "123"
123
probe to-integer false
0
probe to-integer true
1
probe to-integer 123.4
123
probe to-integer 123.8
123
probe to-integer -123.8
-123
```

Если в выражении объединены десятичное и целое число, целое число преобразуется в десятичное:

```
probe 1.2 + 2
3.2
probe 2 + 1.2
3.2
probe 1.01 > 1
true
probe 0 < .001
true
```

1.2.4 Связанные

Используйте **integer?** чтобы определить, является ли значение целым числом - тип данных **integer!**.

```
probe integer? -1234
true
```

Используйте функции **form**, **print** и **mold** с целочисленным аргументом, чтобы вывести целочисленное значение в виде строки:

```
probe mold 123
123

probe form 123
123

print 123
123
```

Целые числа, которые находятся вне диапазона или не могут быть представлены в 32-битном формате, помечаются как ошибка.

2. Значения серий

2.1 Двоичный

2.1.1 Концепция

Двоичные значения содержат двоичные данные любого произвольного типа. Может быть сохранена любая последовательность байтов, например изображение, аудио, исполняемый файл, сжатые данные и зашифрованные данные. Исходным форматом двоичных данных может быть base-2 (двоичный), base-16 (шестнадцатеричный) и base-64. Базой по умолчанию для двоичных данных в REBOL является base-16.

2.1.2 Формат

Двоичные строки записываются в виде знака числа (#), за которым следует строка, заключенная в фигурные скобки. Символы в строке кодируются в одном из нескольких форматов, определяемых необязательным числом перед знаком числа. Base-16 - это формат по умолчанию.

```
#{3A18427F 899AEFD8} ; default base-16

2#{10010110110010101001011011001011} ; base-2

64#{LmNvbSA8yw9CB0aGvXmgUkVCu2Uz934b} ; base-64
```

Внутри строки разрешены пробелы, табуляции и новые строки. Двоичные данные могут занимать несколько строк.

```
probe #{
    3A
    18
    92
    56
}

#{3A189256}
```

Строки, в которых отсутствует правильное количество символов для создания правильного двоичного результата, дополняются справа.

2.1.3 Создание

Функция **to-binary** (в-двоичное) преобразует данные в тип данных **binary!** (двоичные) в базе по умолчанию, установленной в **system/options/binary-base**:

```
probe to-binary "123"
#{313233}
probe to-binary "today is the day..."
#{746F6461792069732074686520646179E2E2E}
```

Чтобы преобразовать целое число в его двоичное значение, передайте его в блоке:

```
probe to-binary [1]
#{01}
probe to-binary [11]
#{0B}
```

Преобразование серии целых чисел в двоичное возвращает преобразование битов для каждого целого числа, объединенного в одно двоичное значение:

```
probe to-binary [1 1 1 1]
#{01010101}
```

2.1.4 Связанные

Использовать **binary?** (двоичный?), чтобы определить является ли значение **binary!** (двоичное!) типом данных.

```
probe binary? #{616263}
true
```

Двоичные значения - это тип серии:

```
probe series? #{616263}
true
probe length? #{616263} ; три шестнадцатеричных значения в этом двоичном формате
3
```

Тесно связано с работой с типом данных **binary!** функции **enbase** и **debase**. Функция **enbase** преобразует строки в их представления base-2, base-16 или base-64 как строки. Функция **debase** преобразует enbased строку в двоичное значение базы, указанной в **system/options/binary-base**.

2.2 Блок

2.2.1 Концепция

Блоки - это группы значений и слов. Блоки используются везде, от самого сценария до блоков данных и кода, представленных в сценарии.

Значения блоков обозначаются открывающими и закрывающими квадратными скобками ([ ]) с любым количеством данных, содержащихся между ними.

```
[123 data "hi"]   ; блок с данными
[]                ; пустой блок
```

Блоки могут содержать записи информации:

```
woodsmen: [
    "Paul" "Bunyan" paul@bunyan.dom
    "Grizzly" "Adams" grizzly@adams.dom
    "Davy" "Crocket" davy@crocket.dom
]
```

Блоки могут содержать код:

```
[print "this is a segment of code"]
```

Блоки также являются типом серий, и поэтому все, что можно сделать с серией, можно сделать с помощью значения блока.

Блоки можно искать:

```
probe copy/part (find woodsmen "Grizzly") 3
[
    "Grizzly" "Adams" grizzly@adams.dom]
```

Блоки можно модифицировать:

```
append woodsmen [
    "John" "Muir" john@muir.dom
]
probe woodsmen
[
    "Paul" "Bunyan" paul@bunyan.dom
    "Grizzly" "Adams" grizzly@adams.dom
    "Davy" "Crocket" davy@crocket.dom
    "John" "Muir" john@muir.dom
]
```

Блоки можно оценивать:

```
blk: [print "data in a block"]
do blk
data in a block
```

Блоки могут содержать любое количество любых блоков:

```
blks: [
    [print "block one"]
    [print "block two"]
    [print "block three"]
]
foreach blk blks [do blk]
block one
block two
block three
```

2.2.2 Формат

Блоки могут содержать любое количество значений или вообще не содержать значений. Они могут занимать несколько строк и включать в себя значения любого типа, включая другие блоки.

Пустой блок:

```
[ ]
```

Блок целых чисел:

```
[24 37 108]
```

Заголовок REBOL:

```
REBOL [
    Title: "Test Script"
    Date: 31-Dec-1998
    Author: "Ima User"
]
```

Блок условия и оценки функции:/p>

```
while [time < 10:00] [
    print time
    time: time + 0:10
]
```

Слова в блоке определять не нужно:

```
blk: [undefined words in a block]
probe value? pick blk 1
false
```

Блоки допускают любое количество строк, пробелов или табуляции. Строки и пробелы можно размещать в любом месте блока, если они не разделяют одно значение.

2.2.3 Создание

Функция **to-block** (в-блок) преобразует данные в тип данных **block!**:

```
probe to-block luke@rebol.com
[luke@rebol.com]
probe to-block {123 10:30 "string" luke@rebol.com}
[123 10:30 "string" luke@rebol.com]
```

2.2.4 Связанные

Использовать **block?** чтобы определить, является ли значение типом данных **block!**.

```
probe block? [123 10:30]
true
```

Поскольку блоки являются подмножеством псевдотипа **series!** (серия!), использовать **series?** чтобы проверить это:

```
probe series? [123 10:30]
true
```

Использование **form** для значения блока создаёт строку из содержимого, содержащегося в блоке:

```
probe form [123 10:30]
123 10:30
```

Использование **mold** для значения блока создаёт строку из значения блока и его содержимого, что позволяет перезагрузить его как значение блока REBOL:

```
probe mold [123 10:30]
[123 10:30]
```

Тесно связанные типы данных - это **hash!** (хэш!) и **list!** (список!). Они используются почти так же, как значения блоков, но имеют особые возможности. Значения списков предназначены для более быстрой обработки изменений списков, чем значения блоков, а значения хэшей предназначены для обработки поиска и индексации данных по хешам. Это полезно при работе с большими наборами данных.

2.3 Электронная почта

2.3.1 Концепция

Адрес электронной почты - это тип данных. Тип данных **email!** позволяет легко выражать адреса электронной почты:

```
send luke@rebol.com {some message}

emails: [
    john@keats.dom
```

```
    lord@byron.dom
    edger@guest.dom
    alfred@tennyson.dom
]
mesg: {poetry reading at 8:00pm!}
foreach email emails [send email mesg]
```

Электронная почта тоже входит в тип данных **series!**, поэтому те же правила, которые применяются к сериям, применяются к электронным письмам:

```
probe head change/part jane@doe.dom "john" 4
john@doe.dom
```

2.3.2 Формат

Стандартный формат адреса электронной почты - это имя, за которым следует знак (@), за которым следует домен. Адрес электронной почты может быть любой длины, но не должен включать какие-либо запрещенные символы, такие как квадратные скобки, кавычки, фигурные скобки, пробелы, символы новой строки и т.д..

Следующее типы данных **email!** имеют допустимые форматы:

```
info@rebol.com
123@number-mail.org
my-name.here@an.example-domain.com
```

В адресах электронной почты сохраняются прописные и строчные буквы.

2.3.3 Доступ

Уточнения можно использовать со значением электронной почты, чтобы получить имя пользователя или домен. Уточнения:

- /user - получить имя пользователя.
- /host - получить домен.

Вот как работают эти уточнения:

```
email: luke@rebol.com
probe email/user
luke
probe email/host
rebol.com
```

2.3.4 Создание

Функция **to-email** (в-электронную-почту) преобразует данные в тип данных **email!**:

```
probe to-email "info@rebol.com"
info@rebol.com

probe to-email [info rebol.com]
info@rebol.com
```

```
probe to-email [info rebol com]
info@rebol.com

probe to-email [user some long domain name out there dom]
user@some.long.domain.name.out.there.dom
```

2.3.5 Связанные

Использовать **email?** чтобы определить, является ли значение типом данных **email!**.

```
probe email? luke@rebol.com
true
```

Поскольку электронные письма - это подмножество псевдотипа **series!**, использовать **series?** чтобы определить, является ли значение серией:

```
probe series? luke@rebol.com
true

probe pick luke@rebol.com 5
#"@"
```

2.4 Файл

2.4.1 Концепция

Тип данных **file!** может быть именем файла, именем каталога или путем к каталогу.

```
%file.txt
%directory/
%directory/path/to/some/file.txt
```

Значения файлов являются подмножеством серий, поэтому ими можно управлять как сериями:

```
probe find %dir/path1/path2/file.txt "path2"
%path2/file.txt
f: %dir/path/file.txt
probe head remove/part (find f "path/") (length? "path/")
%dir/file.txt
```

2.4.2 Формат

Файлы обозначаются знаком процента (%), за которым следует последовательность символов:

```
load %image.jpg
prog: load %examples.r
save %this-file.txt "This file has few words."
files: load %../programs/
```

Необычные символы в именах файлов должны быть закодированы с помощью % и шестнадцатеричного числа, что является соглашением в Интернете. Имя файла с пробелом (шестнадцатеричное 20) будет выглядеть так:

```
probe %cool%20movie%20clip.mpg
%cool%20movie%20clip.mpg
print %cool%20movie%20clip.mpg
cool movie clip.mpg
```

Другой формат - заключить имя файла в кавычки:

```
probe %"cool movie clip.mpg"
%cool%20movie%20clip.mpg
print %"cool movie clip.mpg"
cool movie clip.mpg
```

Стандартным символом для разделения каталогов в пути является косая черта (/), а не обратная косая черта (\). Однако язык REBOL автоматически преобразует обратную косую черту в именах файлов в прямую косую черту:

```
probe %\some\path\to\some\where\movieclip.mpg
%/some/path/to/some/where/movieclip.mpg
```

2.4.3 Создание

Функция **to-file** (в-файл) преобразует данные в тип данных **file!**:

```
probe to-file "testfile"
%testfile
```

При передаче блока элементы в блоке объединяются в путь к файлу с последним элементом, используемым в качестве имени файла:

```
probe to-file [some path to a file the-file.txt]
%some/path/to/a/file/the-file.txt
```

2.4.4 Связанные

Используйте **file?** чтобы определить, является ли значение типом данных **file!**.

```
probe file? %rebol.r
true
```

Поскольку файлы являются частью псевдотипа **series!**, использовать **series?** чтобы проверить это:

```
probe series? %rebol.r
true
```

2.5 Хеш

2.5.1 Концепция

Хеш - это блок, специально организованный для ускорения поиска данных. Когда поиск выполняется по хэш-блоку, поиск выполняется с использованием хеш-таблицы для поиска. Для больших блоков это может ускорить поиск в сотни раз.

2.5.2 Формат

Блоки хеширования должны быть созданы с использованием **make** или **to-hash**. У них нет лексического формата.

2.5.3 Создание

Используйте **make** для инициализации хеш-блока:

```
hsh: make hash! 10 ; выделение места для 10 элементов
```

Функция **to-hash** (в-хеш) преобразует данные в тип данных **hash!**.

Преобразовать блок:

```
blk: [1 "one" 2 "two" 3 "three" 4 "four"]
probe hash: to-hash blk
make hash! [1 "one" 2 "two" 3 "three" 4 "four"]
print select hash 2
two
```

Преобразуйте различные значения:

```
probe to-hash luke@rebol.com

probe to-hash 123.5

probe to-hash {123 10:30 "string" luke@rebol.com}
```

2.5.4 Связанные

Использовать **hash?** чтобы проверить тип данных.

```
hsh: to-hash [1 "one" 2 "two" 3 "three" 4 "four"]
probe hash? Hsh
true
```

Поскольку хеши - это подмножество псевдотипа **series!**, использовать **series?** чтобы проверить это:

```
probe series? hsh
true
```

При формировании хеш-значения создается строка из содержимого, содержащегося в хеш-значении:

```
probe form hsh
"1 one 2 two 3 three 4 four"
```

Формирование хеш-значения создает строку самого хеш-значения и его содержимого, что позволяет перезагрузить его как хеш-значение REBOL:

```
probe mold hsh
make hash! [1 "one" 2 "two" 3 "three" 4 "four"]
```

2.6 Изображение

2.6.1 Концепция

Тип данных **image!** (изображение!)- это серия, содержащая изображения RGB. Этот тип данных используется с REBOL/View.

Поддерживаемые форматы изображений: GIF, JPEG и BMP. Загруженным изображением можно управлять как серией.

2.6.2 Формат

Изображения обычно загружаются из файла. Однако они также могут быть выражены в исходном коде путем создания изображения. Предоставленный блок включает размер изображения и его данные RGB.

```
image: make image! [192x144 #{
    B34533B44634B44634B54735B7473
    84836B84836B84836BA4837BA4837
    BC4837BC4837BC4837BC4837BC483 ...
}
```

2.6.3 Создание

Пустые изображения могут быть созданы с помощью **make** или **to-image:**:

```
empty-img: make image! 300x300
empty-img: to-image 150x300
```

Размер изображения указан.

Изображения также могут быть сделаны из снимков лица. Это также делается с помощью **make** или **to-image:**:

```
face-shot: make image! face
face-shot: to-image face
```

Используйте **load** для загрузки файла изображения. Если формат изображения не поддерживается, загрузить его не удастся.

Загрузка изображения:

```
img: load %bay.jpg
```

2.6.4 Связанные

Использовать **image?** чтобы определить, является ли значение типом данных **image!**:

```
probe image? img
```

Изображения - это часть псевдотипа **series!**:

```
probe series? img
```

Используйте уточнение **/size**, чтобы вернуть размер изображения в пикселях в виде парного значения::

```
probe img/size
```

Значения пикселей изображения получаются с помощью **pick** и изменяются с помощью **poke**. Значение, возвращаемое командой **pick**, является значением кортежа RGB. Значение, замененное на **poke**, также должно быть значением кортежа.

Выбор конкретных пикселей:

```
probe pick img 1

probe pick img 1500
```

Меняем конкретные пиксели:

```
poke img 1 255.255.255
probe pick img 1

poke img 1500 0.0.0
probe pick img 1500
```

2.7 Номер

2.7.1 Концепция

Issue! представляет собой последовательность символов, используемых для упорядочивания символов или идентификаторов таких вещей, как номера телефонов, номера моделей, серийные номера и номера кредитных карт.

Значения номера представляют собой подмножество серий, поэтому ими можно управлять как сериями:

```
probe copy/part find #888-555-1212 "555" 3
#555
```

2.7.2 Формат

Номер начинаются с цифрового знака (#) и продолжаются до тех пор, пока не будет достигнут первый ограничивающий символ (например, пробел).

```
#707-467-8000
#A-0987654321-CD-09876
#1234-5678-4321-8765
#MG82/32-7
```

Значения, содержащие символы-разделители, следует записывать в виде строк, а не вопросов.

2.7.3 Создание

Функция **to-issue** преобразует данные в тип данных **issue!**:

```
probe to-issue "1234-56-7890"
#1234-56-7890
```

2.7.4 Связанные

Использовате **issue?** чтобы определить, является ли значение типом данных **issue!**.

```
probe issue? #1234-56-7890
true
```

Поскольку номер являются подмножеством псевдотипа сериала, используйте **series?** чтобы проверить это:

```
probe series? #1234-56-7890
true
```

Функция **form** возвращает проблему в виде строки без знака числа (#):

```
probe form #1234-56-7890
1234-56-7890
```

Функция **mold** возвращает проблему в виде строки, которую REBOL может прочитать как значение проблемы:

```
probe mold #1234-56-7890
#1234-56-7890
```

Фнкция **print** выводит вопрос на стандартный вывод после выполнения **reform** на нем:

```
print #1234-56-7890
1234-56-7890
```

2.8 Список

2.8.1 Концепция

Списки представляют собой блоки связанных списков, которые позволяют быстрее и эффективнее вставлять и удалять их значения. Их можно использовать в случаях, когда выполняется большое количество операций вставки или удаления больших блоков.

2.8.2 Формат

Блоки списка должны быть созданы с помощью `make` или `to-list`. У них нет лексического формата.

Значения списков не являются прямой заменой блоков. Между блоками и списками есть несколько отличий:

- При вставке в список ссылка на него изменяется сразу после точки вставки.
- Удаление элемента, на который в настоящее время имеется ссылка в списке, приводит к сбросу ссылки в конец списка.

Следующие примеры показывают разницу в поведении между вставкой в список и в блок.

Инициализация блока и списка:

```
blk: [1 2 3]

lst: to-list [1 2 3]
```

Вставка в блок и список:

```
insert blk 0

insert lst 0
```

Смотрим на слово после блока и список после вставки. Обратите внимание, что `blk` указывает на голову, как и до вставки `0`, но `lst` указывает сразу после точки вставки:

```
print blk
0 1 2 3

print lst
1 2 3

print head lst
0 1 2 3
```

Следующие примеры показывают разницу в поведении между удалением элемента из списка и блока.

Инициализация блока и списка:

```
blk: [1 2 3]

lst: to-list [1 2 3]
```

Удаление из блока и списка:

```
remove blk

remove lst
```

Смотрим на слово после удаления значения. Обратите внимание, что теперь `lst` указывает на конец серии:

```
print blk
2 3

print tail? lst
true

print head lst
2 3
```

Если вы не хотите, чтобы слово было в конце после удаления значения, сделайте шаг вперед и удалите значение за текущим индексом. Следующие примеры демонстрируют это.

Инициализация списка:

```
lst: to-list [1 2 3]
```

Шаг вперед и удаление значения за текущим индексом:

```
remove back (lst: next lst)
```

Глядя на слово после удаления значения:

```
probe lst
make list! [2 3]
```

2.8.3 Создание

Используйте `make` для инициализации значения списка:

```
lst: make list! 10 ; allocating space for 10 elements
```

Функция `to-list` преобразует данные в тип данных `list!`:

Преобразовать блок:

```
blk: [1 "one" 2 "two" 3 "three" 4 "four"]
probe to-list blk
```

2.8.4 Связанные

Используйте list? чтобы определить, является ли значение типом данных list!.

```
lst: to-list [1 "one" 2 "two" 3 "three" 4 "four"]
probe list? Lst
true
```

Поскольку списки - это подмножество типа данных series!, используйте series? чтобы проверить, является ли список серией:

```
probe series? lst
true
```

Использование form для значения списка создает строку из содержимого, содержащегося в списке:

```
probe form lst
"1 one 2 two 3 three 4 four"
```

Использование mold для значения списка создает строку самого значения списка и его содержимого, что позволяет перезагрузить его как значение списка REBOL:

```
probe mold lst
make list! [1 "one" 2 "two" 3 "three" 4 "four"]
```

2.9 Скобки

2.9.1 Концепция

Тип данных paren! - это блок, который немедленно оценивается. Он идентичен блоку во всех отношениях, за исключением того, что он оценивается, когда он встречается, и его результат возвращается.

При использовании в вычисляемом выражении paren! позволяет контролировать порядок оценки:

```
print 1 + (2 * 3)
7
print 1 + 2 * 3
9
```

Значение paren! можно получить доступ и изменить так же, как и любой блок. Однако когда речь идет о paren! необходимо соблюдать осторожность, чтобы предотвратить оценку if. Если вы храните парен в переменной, вам нужно будет использовать форму получения слова (:слово), чтобы предотвратить ее оценку.

Скобки - это тип серии, поэтому все, что можно сделать с серией, можно сделать с помощью парных значений.

```
paren: first [(1 + 2 * 3 / 4)]

print type? :paren
paren!
```

```
print length :paren
7

print first :paren
1

print last :paren
4

insert :paren [10 + 5 *]
probe :paren
(10 + 5 * 1 + 2 * 3 / 4)

print paren
12.75
```

2.9.2 Формат

Скобки обозначаются открывающими и закрывающими круглыми скобками. Они могут занимать несколько строк и содержать любые данные, включая другие скобки значения.

2.9.3 Создание

Функция `make` может быть использована для выделения PAREN значения:

```
paren: make paren! 10
insert :paren 10
insert :paren `+
insert :paren 20

print :paren
20 + 10
print paren
30
```

Функция `to-paren` преобразует данные в тип данных `paren!`:

```
probe to-paren "123 456"
(123 456)
probe to-paren [123 456]
(123 456)
```

2.9.4 Связанные

Используйте `paren?` чтобы проверить тип данных.

```
blk: [(3 + 3)]
probe pick blk 1
(3 + 3)
probe paren? pick blk 1
true
```

Поскольку скобки - это подмножество псевдотипа **series!**, использовать **series?** чтобы проверить это:

```
probe series? pick blk 1
true
```

Использование **form** в парном значении создает строку из содержимого, содержащегося в парном значении:

```
probe form pick blk 1
3 + 3
```

2.10 Путь

2.10.1 Концепция

Пути - это набор слов и значений, обозначенных косой чертой (/). Пути используются для навигации или поиска чего-либо. Слова и значения пути называются уточнениями, и они объединяются, чтобы обеспечить средства навигации по значению или функции.

Пути можно использовать для блоков, файлов, строк, списков, хэшей, функций и объектов. Как работает путь, зависит от используемого типа данных.

Пути можно использовать для выбора значений из блоков, выбора символов из строк, доступа к переменным в объектах, уточнения работы функции:

```
USA/CA/Ukiah/size (block selection)

names/12          (string position)

account/balance   (object function)

match/any         (function option)
```

Пример ниже показывает простоту использования пути для доступа к мини-базе данных, созданной из нескольких блоков:

```
towns: [
    Hopland [
        phone #555-1234
        web   http://www.hopland.ca.gov
    ]

    Ukiah [
        phone #555-4321
        web   http://www.ukiah.com
        email info@ukiah.com
    ]
]

print towns/ukiah/web
http://www.ukiah.com
```

Резюме конструкций пути:

Действие	Типовые слова	Типовое испытание	Преобразование
path/word:	set-path!	set-path?	to-set-path
path/word	path!	path?	to-path
'path/word	lit-path!	lit-path?	to-lit-path

Примеры путей:

Оцените функцию объекта:

```
obj: make object! [
    hello: func [] [print "hello! hello!"]
]
obj/hello
hello! hello!
```

Оцените слово объекта:

```
obj: make object! [
    text: "do you believe in magic?"
]
probe obj/text
do you believe in magic?
```

Доработки функций:

```
hello: func [/again] [
    print either again ["hello again!"]["hello"]
]
hello/again
hello again!
```

Выберите из блоков или нескольких блоков:

```
USA: [
    CA [
        Ukiah [
            population 15050
            elevation [610 feet]
        ]
        Willits [
            population 5073
            elevation [1350 feet]
        ]
    ]
]
print USA/CA/Ukiah/population
15050
print form USA/CA/Willits/elevation
1350 feet
```

Выберите элементы из серий и встроенных серий по их числовому положению:

```
string-series: "abcdefg"
block-series: ["John" 21 "Jake" 32 "Jackson" 43 "Joe" 52]
block-with-sub-series: [ "abc" [4 5 6 [7 8 9]]]
probe string-series/4
#"d"
probe block-series/3
Jake
probe block-series/6
43
probe block-with-sub-series/1/2
#"b"
probe block-with-sub-series/2/2
5
probe block-with-sub-series/2/4/2
8
```

Слова, представленные как пути, являются символическими и поэтому не оцениваются. Это необходимо для обеспечения наиболее интуитивно понятной формы ссылок на объекты. Чтобы использовать ссылку на слово, требуется явная ссылка на значение слова:

```
city: 'Ukiah
probe USA/CA/:city
[
    population 15050
    elevation "610 feet"
]
```

Пути в блоках, хэшах или объектах оцениваются путем сопоставления слова на верхнем уровне пути и проверки слова как **block!**, **hash!** или **object!** значение. Затем ищется следующее слово в пути как слово, выраженное в блоке, хэше или объекте, и выполняется неявный выбор. Возвращается значение, следующее за совпавшим словом. Когда возвращаемое значение является блоком, хешем или объектом, путь может быть расширен:

Получение значения, связанного с **CA** в **USA:**

```
probe USA/CA
[
    Ukiah [
        population 15050
        elevation "610 feet"
    ]
    Willits [
        population 9935
        elevation "1350 feet"
    ]
]
```

Получение значения, связанного с **Willits** в **USA/CA:**

```
probe USA/CA/Willits
[
    population 9935
    elevation "1350 feet"
]
```

Получение значения, связанного с **population** в **USA/CA/Willits**:

```
probe USA/CA/Willits/population
9935
```

Когда слово используется в пути, которого нет в данной точке структуры, возникает ошибка:

```
probe USA/CA/Mendocino
** Script Error: Invalid path value: Mendocino.
** Where: probe USA/CA/Mendocino
```

Пути можно использовать для изменения значений в блоках и объектах:

```
USA/CA/Willits/elevation: "1 foot, after the earthquake"
probe USA/CA/Willits
[
    population 9935
    elevation "1 foot, after the earthquake"
]
obj/text: "yes, I do believe in magic."
probe obj
make object! [
    text: "yes, I do believe in magic."
]
```

Блоки, хэши, функции и объекты могут быть смешаны в путях.

Выбор из элементов в блоке внутри объекта:

```
obj: make object! [
    USA: [
        CA [
            population "too many"
        ]
    ]
]
probe obj/USA/CA/population
too many
```

Использование уточнений функций внутри объекта:

```
obj: make object! [
    hello: func [/again] [
        print either again [
            "hello again"
        ] [
            "oh, hello"
        ]
    ]
]
obj/hello/again
hello again
```

Пути - это тип серии, поэтому все, что можно сделать с серией, можно сделать с помощью значений пути:

```
root: [sub1 [sub2 [
    word "a word at the end of the path"
    num 55
]   ]   ]
path: 'root/sub1/sub2/word
probe :path
root/sub1/sub2/word
```

В предыдущем примере обозначение `:path` использовалось для получения самого `path`, а не его значения:

```
probe path
a word at the end of the path
```

Смотрим на длину пути:

```
probe length? :path
4
```

Поиск слова в пути:

```
probe find :path 'sub2
sub2/word
```

Изменение слова в пути:

```
change find :path 'word 'num
probe :path
root/sub1/sub2/num
probe path
55
```

2.10.2 Формат

Пути выражаются относительно корневого слова с помощью ряда уточнений, каждое из которых разделено косой чертой (/). Эти уточнения могут быть словами или значениями. Их конкретная интерпретация зависит от типа данных корневого значения.

Слова, представленные как уточнения в путях, являются символическими и не оцениваются. Это необходимо для обеспечения наиболее интуитивно понятной формы ссылок на объекты. Чтобы использовать ссылку на слово, требуется явная ссылка на значение слова:

```
root/:word
```

В этом примере используется значение переменной, а не её имя.

2.10.3 Создание

Вы можете создать пустой путь заданного размера с помощью **make**:

```
path: make path! 10
insert :path `test
insert tail :path `this
print :path
test/this
```

Функция **to-path** преобразует данные в тип данных **path!**:

```
probe to-path [root sub]
root/sub
probe to-path "root sub"
root/sub
```

Функция **to-set-word** преобразует другие значения в тип данных **set-word**.

```
probe to-set-path "root sub"
root/sub:
```

Функция **to-lit-word** преобразует другие значения в тип данных **lit-word**.

```
probe to-lit-path "root sub"
'root/sub
```

2.10.4 Связанные

Использовать **path?**, **set-path?**, и **lit-path?** для определения типа данных значения.

```
probe path? second [1 two "3"]
false
blk: [sub1 [sub2 [word 1]]]
blk2: [blk/sub1/sub2/word: 2]
if set-path? (pick blk2 1) [print "it is set"]
it is set
probe lit-path? first ['root/sub]
true
```

Поскольку пути - это подмножество псевдотипа **series!**, использовать **series?** чтобы проверить это:

```
probe series? pick [root/sub] 1
true
```

Использование **form** в значении пути создает строку из пути:

```
probe form pick [root/sub] 1
root/sub
```

Использование **mold** для значения пути создает строку самого значения пути, что позволяет перезагрузить ее как значение пути REBOL:

```
probe mold pick [root/sub] 1
root/sub
```

2.11 Строка

2.11.1 Концепция

Строки - это набор символов. Все операции, выполняемые над значениями серий, могут выполняться над строками.

2.11.2 Формат

Строковые значения записываются как последовательность символов, заключенных в двойные кавычки " " или фигурные скобки {}. Строки, заключенные в двойные кавычки, ограничиваются одной строкой и не должны содержать непечатаемых символов.

```
"This is a short string of characters."
```

Строки, заключенные в фигурные скобки, используются для больших разделов текста, занимающих несколько строк. Все символы строки, включая пробелы, табуляторы, кавычки и символы новой строки, являются частью строки.

```
{This is a long string of text that will
not easily fit on a single line of source.
These are often used for documentation
purposes.}
```

Фигурные скобки подсчитываются в строке, поэтому строка может включать другие фигурные скобки, если количество закрывающих фигурных скобок совпадает с количеством открывающих фигурных скобок.

```
{
This is another long string of text that would
never fit on a single line. This string also
includes braces { a few layers deep { and is
valid because there are as many closing braces }
as there are open braces } in the string.
}
```

Вы можете включать в строки специальные символы и операции, добавляя к ним знак вставки (^). К специальным символам относятся:

Сочитание	Определение
^"	Вставляет двойную кавычку (").
^}	Вставляет закрывающую скобку (}).
^^	Вставляет курсор (^).

^/	Начинает новую строку.
^(line)	Начинает новую строку.
^-	Вставляет табуляцию.
^(tab)	Вставляет табуляцию.
^(page)	Начинает новую страницу.
^(back)	Удаляет один символ слева от места вставки.
^(null)	Вставляет нулевой символ.
^(escape)	Вставляет escape-символ.
^(letter)	Вставляет указанную букву (A-Z).
^(xx)	Вставляет символ ASCII в виде шестнадцатеричного числа (xx). его формат допускает расширение в символы Юникода в будущем.

2.11.3 Создание

Используйте **make**, чтобы создать заранее выделенное пространство для пустой строки:

```
make string! 40'000 ; space for 40k characters
```

Функция **to-string** преобразует данные других типов в тип данных **string!**:

```
probe to-string 29-2-2000
"29-Feb-2000"
probe to-string 123456.789
"123456.789"
probe to-string #888-555-2341
"888-555-2341"
```

Преобразование блока данных в строку с **to-string** имеет эффект **rejoin** (повторное соединение), но без оценки содержимого блока:

```
probe to-string [123 456]
"123456"
probe to-string [225.225.225.0 none true 'word]
"225.225.225.0nonetrueword"
```

2.11.4 Связанные

Использовать **string?** или **series?** чтобы определить, является ли значение типом данных **string!**:

```
print string? "123"
true
print series? "123"
true
```

Функции **form** и **mold** тесно связаны со строками, поскольку они создают строки из других типов данных. Функция **form** создает удобочитаемую версию указанного типа данных, в то время как **mold** создает читаемую версию REBOL.

```
probe form "111 222 333"
"111 222 333"
probe mold "111 222 333"
{"111 222 333"}
```

2.12 Тег

2.12.1 Концепция

Теги используются в HTML и других языках разметки, чтобы указать, как следует обрабатывать текстовые поля. Например, тег `<HTML>` в начале файла указывает, что он должен быть проанализирован в соответствии с правилами языка гипертекстовой разметки. Тег с косой чертой (/), например `</HTML>`, указывает на закрытие тега.

Теги являются подмножеством серий, поэтому ими можно управлять как таковые:

```
a-tag: <img src="mypic.jpg">
probe a-tag
<img src="mypic.jpg">
append a-tag { alt="My Picture!"}
probe a-tag
<img src="mypic.jpg" alt="My Picture!">
```

2.12.2 Формат

Допустимые теги начинаются с открытой угловой скобки (<) и заканчиваются закрывающей скобкой (>). Например:

```
<a href="index.html">
<img src="mypic.jpg" width="150" height="200">
```

2.12.3 Создание

Функция `to-tag` преобразует данные в тип данных `tag!`:

```
probe to-tag "title"
<title>
```

Используйте `build-tag` для создания тегов, включая их атрибуты. Функция `build-tag` принимает один аргумент - блок. В этом блоке первое слово используется как имя тега, а остальные слова обрабатываются как пары значений атрибутов:

```
probe build-tag [a href http://www.rebol.com/]
<a href="http://www.rebol.com/">
probe build-tag [
    img src %mypic.jpg width 150 alt "My Picture!"
]
<img src="mypic.jpg" width="150" alt="My Picture!">
```

2.12.4 Связанные

Использовать **tag?** чтобы определить, является ли значение типом данных **tag!**.

```
probe tag? <a href="http://www.rebol.com/">

true
```

Поскольку теги являются подмножеством псевдотипа серии, используйте **series?** чтобы проверить это:

```
probe series? <a href="http://www.rebol.com/">

true
```

Функция **form** возвращает тег в виде строки:

```
probe form <a href="http://www.rebol.com/">

{<a href="http://www.rebol.com/">}
```

Функция **mold** возвращает тег в виде строки:

```
probe mold <a href="http://www.rebol.com/">

{<a href="http://www.rebol.com/">}
```

Функция **print** печатает метки на стандартный вывод после выполнения **reform** на нем:

```
print <a href="http://www.rebol.com/">

<a href="http://www.rebol.com/">
```

2.13 URL

2.13.1 Концепция

URL - это аббревиатура от Uniform Resource Locator, интернет-стандарта, используемого для доступа к таким ресурсам, как веб-страницы, изображения, файлы и электронная почта по сети. Наиболее известная схема URL-адресов используется для таких веб-сайтов, как **http://www.REBOL.com.**.

Значения URL-адресов являются подмножеством серий, поэтому ими можно управлять как сериями:

```
url: http://www.rebol.com/reboldoc.html
probe to-file find/reverse (tail url) "rebol"
%reboldoc.html
```

2.13.2 Формат

Первая часть URL-адреса указывает его протокол связи, называемый схемой. Язык поддерживает несколько схем, в том числе: веб - страниц (**HTTP:**), передача файлов (**FTP:**), группы новостей (**NNTP:**), электронная почта (**MAILTO:**), файлы (**FILE:**), finger (**FINGER:**), Whois (**WHOIS:**), сетевое время (**DAYTIME:**), почтовое отделение (**POP:**), управление передачей (**TCP:**) и служба доменных имен (**DNS:**). За этими именами схем следуют символы, зависящие от используемой схемы.

```
http://host.dom/path/file
ftp://host.dom/path/file
nntp://news.some-isp.net/some.news.group
mailto:name@domain
file://host/path/file
finger://user@host.dom
whois://rebol@rs.internic.net
daytime://everest.cclabs.missouri.edu
pop://user:passwd@host.dom/
tcp://host.dom:21
dns://host.dom
```

Некоторые поля необязательны. Например, за хостом может следовать номер порта, если он отличается от значения по умолчанию. URL-адрес FTP предоставляет пароль по умолчанию, если он не указан:

```
ftp://user:password@host.dom/path/file
```

Символы в URL-адресе должны соответствовать Интернет-стандартам. Запрещенные символы должны быть закодированы в шестнадцатеричном формате, перед ними должен стоять escape-символ %:

```
probe http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt
http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt

print http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt
http://www.somesite.dom/odd(dir)/odd{file}.txt
```

2.13.3 Создание

Функция **to-url** преобразует блоки в тип данных **url!**, первый элемент в блоке - это схема, второй элемент - это домен (с указанием именем пользователя, порт или без), остальные элементы - это путь и файл:

```
probe to-url [http www.rebol.com reboldoc.html]
http://www.rebol.com/reboldoc.html

probe to-url [http www.rebol.com %examples "websend.r"]
http://www.rebol.com/examples/websend.r

probe to-url [http usr:pass@host.com:80 "(path)" %index.html]
http://usr:pass@host.com:80/%28path%29/index.html
```

2.13.4 Связанные

Слово типа данных - **url!**.

Использовать **url?** чтобы проверить тип данных..

```
probe url? ftp://ftp.rebol.com/
true
```

Поскольку URL-адреса являются подмножеством псевдотипа серии, используйте **series?** чтобы проверить это:

```
probe series? http://www.rebol.com/
true
 
```

3. Прочие типы

3.1 Символ

3.1.1 Концепция

Символы - это не строки; это отдельные значения, из которых строятся строки. Символ может быть печатным, непечатаемым или управляющим.

3.1.2 Формат

Char! значение записывается в виде знака числа (#), за которым следует строка, заключенная в двойные кавычки. Знак числа необходим, чтобы отличить символ от строки:

```
#"R"    ; символ "R"
"R"     ; строка с буквой "R"
```

Символы могут включать escape-последовательности, которые начинаются с символа вставки (^) и сопровождаются одним или несколькими символами кодировки. Эта кодировка может включать символы от **#"^A"** до **#"^Z"** для элемента **control A** для **control #"^Z"** (верхний и нижний регистры одинаковы) (...или сочетание славиш от Ctrl+A до Ctrl+Z) :

```
#"^A" #"^Z"
```

Кроме того, если внутри символа используются скобки, они указывают специальное значение. Например, null можно записать как:

```
"^@"
"^(null)"
"^(00)"
```

Последняя строка записывается в шестнадцатеричном формате (основание 16). Квадратные скобки вокруг значения позволяют в будущем расширить его до 16-битных символов Юникода.

Ниже приводится таблица управляющих символов, которые можно использовать в REBOL.

Запись	Определение
`#"(null)" or #"@"`	null (ноль)
`#"(line)", #"/" or, #"."`	конец линии
`#"(tab)" or #"-"`	горизонтальная табуляция
`#"(page)"`	новая страница (и извлечение страницы)
`#"(esc)"`	esc
`#"(back)"`	backspace
`#"(del)"`	delete
`#"^"`	символ каретки
`#"^""`	кавычка
`#"(00)" to #"(FF)"`	шестнадцатеричные формы символов

3.1.3 Создание

Символы могут быть преобразованы в другие типы данных и обратно с помощью функции **to-char**:

```
probe to-char "a"
#"a"
probe to-char "z"
#"z"
```

Символы соответствуют стандарту ASCII и могут быть созданы путем указания числового эквивалента символа:

```
probe to-char 65
#"A"
probe to-char 52
#"4"
probe to-char 52.3
#"4"
```

Другой способ получения символа - получить первый символ из строки:

```
probe first "ABC"
#"A"
```

Хотя символы в строках не чувствительны к регистру, сравнение между отдельными символами чувствительно к регистру:

```
probe "a" = "A"
true
probe #"a" = #"A"
false
```

Однако при использовании во многих типах функций сравнение не чувствительно к регистру, если вы не укажете этот параметр. Примеры:

```
select [#"A" 1] #"a"
1

select/case [#"A" 1] #"a"
none

find "abcde" #"B"
"bcde"

find/case "abcde" #"B"
none

switch #"A" [#"a" [print true]]
true
```

3.1.4 Связанные

Используйте **char?** чтобы определить, является ли значение типом данных **char!**.

```
probe char? "a"
false

probe char? #"a"
true
```

Используйте функцию **form** для печати символа без знака числа:

```
probe form #"A"
"A"
```

Используйте команду **mold** on, чтобы напечатать символ со знаком числа и двойными кавычками (и escape-последовательностями для тех символов, которые этого требуют):

```
probe mold #"A"
{#"A"}
```

3.2 Дата

3.2.1 Концепция

Во всем мире даты записываются в самых разных форматах. Однако в большинстве стран используется формат **"день-месяц-год"**. Одно из немногих исключений - США, где обычно используется формат **месяц-день-год**. Например, дата, записанная в цифровом формате как 01.02.1999, неоднозначна. Месяц можно интерпретировать как февраль или январь. В некоторых странах используется тире (-), в некоторых - косая черта (/), а в других в качестве разделителя используется точка (.). Наконец, компьютерщики часто предпочитают даты в формате «год-месяц-день» (ISO), чтобы их можно было легко отсортировать.

3.2.2 Формат

Язык REBOL гибкий, позволяющий типы данных **date!** быть выраженой в различных форматах. Например, первый день марта может быть выражен в любом из следующих форматов:

```
probe 1/3/1999
1-Mar-1999
probe 1+++1999
1-Mar-1999
probe 1999+++1  ;ISO format
1-Mar-1999
```

Год может охватывать от 9999 до 1. Високосные дни (29 февраля) могут быть записаны только для високосных лет:

```
probe 29-2-2000
29-Feb-2000
```

Поля дат можно разделять косой чертой (/) или тире (-). Даты могут быть записаны в формате год-месяц-день или день-месяц-год:

```
probe 1999-10-5
5-Oct-1999
probe 1999/10/5
5-Oct-1999
probe 5-10-1999
5-Oct-1999
probe 5/10/1999
5-Oct-1999
```

Поскольку международные форматы даты, которые широко не используются в США, также можно использовать название месяца или аббревиатуру месяца:

```
probe 5/Oct/1999
5-Oct-1999
probe 5-October-1999
5-Oct-1999
probe 1999/oct/5
5-Oct-1999
```

Когда год является последним полем, его можно записать как четырехзначное или двузначное число:

```
probe 5/oct/99
5-Oct-1999
probe 5/oct/1999
5-Oct-1999
```

Однако желательно указывать год полностью. В противном случае возникнут проблемы с операциями сравнения дат и сортировки. В то время как две цифры могут использоваться для обозначения года, интерпретация двухзначного года относится к текущему году и действительна только в течение 50 лет в будущем или в прошлом:

```
probe 28-2-66   ; refers to 1966
28-Feb-1966
probe 12-Mar-20 ; refers to 2020
12-Mar-2020
probe 11+++45   ; refers to 2045, not 1945
11-Mar-2045
```

Рекомендуется использовать год из четырех цифр, чтобы избежать потенциальных проблем. Чтобы представить даты в первом веке (что делается редко, потому что григорианского календаря не существовало), используйте ведущие нули для обозначения века (как в **9-4-0029**).

Даты также могут включать необязательное поле времени и необязательный часовой пояс. Время отделяется от даты косой чертой (/). Часовой пояс добавляется с помощью плюса (+) или минуса (-), пробелы не допускаются. Часовые пояса записываются как сдвиг во времени (плюс или минус) от GMT. Разрешение часового пояса до получаса. Если временной сдвиг является целым числом, предполагается, что это часы:

```
probe 4/Apr/2000/6:00+8:00
4-Apr-2000/6:00+8:00
probe 1999-10-2/2:00-4:00
2-Oct-1999/2:00-4:00
probe 1/1/1990/12:20:25-6
1-Jan-1990/12:20:25
```

В дате не должно быть пробелов. Например:

```
10 - 5 - 99
```

будет интерпретироваться как выражение вычитания, а не как дата.

3.2.3 Доступ

Уточнения можно использовать со значением даты, чтобы получить любое из его определенных полей:

Уточнение	Описание
/day	Получает день.
/month	Получает месяц.
/year	Получает год.
/julian	Получает день года.
/weekday	Получает день недели (1-7 / пн-вс).
/time	Получает время (если есть).
/hour	Получает час времени (если есть)
/minute	Получает минуты времени (если есть).
/second	Получает секунды времени (если есть).
/zone	Получает часовой пояс (если есть).

Вот как работают эти уточнения:

```
some-date: 29-Feb-2000
probe some-date/day
29
probe some-date/month
2
probe some-date/year
2000
days: ["Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"]
probe pick days some-date/weekday
Tue
```

Когда время присутствует, можно использовать уточнения, связанные со временем b>/hour, **/minute** и **/second**, используются с уточнением **/time**, что изолирует отрезок времени значения даты для их работы по:

```
lost-time: 29-Feb-2000/11:33:22.14-8:00
probe lost-time/time
11:33:22.14
probe lost-time/time/hour
11
probe lost-time/time/minute
33
probe lost-time/time/second
22.14
probe lost-time/zone
-8:00
```

3.2.4 Создание

Используйте функцию **to-date** для преобразования значений в **date!:**

```
probe to-date "5-10-1999"
5-Oct-1999
probe to-date "5 10 1999 10:30"
5-Oct-1999/10:30
probe to-date [1999 10 5]
5-Oct-1999
probe to-date [5 10 1999 10:30 -8:00]
5-Oct-1999/10:30-8:00
```

[!Примечание! При преобразовании в **date!**, год должен быть указан в виде четырех цифр.

Преобразования можно применять к различным математическим операциям над датами:

```
probe 5-Oct-1999 + 1
6-Oct-1999
probe 5-10-1999 - 10
25-Sep-1999
probe 5-Oct-1999/23:00 + 5:00
6-Oct-1999/4:00
```

3.2.5 Связанные

Используйте **date?**чтобы определить является ли у значения тип данных **date!**.

```
probe date? 5/1/1999
true
```

Связанная функция **to-idate** возвращает стандартную строку даты в Интернете. Формат даты в Интернете - день, дата, месяц, год, время (в 24-часовом формате) и смещение часового пояса от GMT.

```
probe to-idate now
Fri, 30 Jun 2000 14:42:26 -0700
```

Now возвращает текущую дату и время в полном формате, включая смещение часового пояса:

```
probe now
30-Jun-2000/14:42:26-7:00
```

3.3 Логика

3.3.1 Концепция

Тип данных **logic!** состоит из двух состояний, представляющих **true** (истину) и **false** (ложь). Они часто возвращаются из таких сравнений, как:

```
age: 100
probe age = 100
true
time: 10:31:00
probe time < 10:30
false
str: "this is a string"
probe (length? str) > 10
true
```

Тип данных **logic!** чаще всего используется в качестве параметров условных функций, таких как **if**, **while** и **until**:

```
if age = 100 [print "Centennial human"]
Centennial human

while [time > 6:30] [
    send person "Wake up!"
    wait [0:10]
]
```

Дополнение к логическому значению получается из функции **not** :

```
there: place = "Ukiah"
if not there [...]
```

3.3.2 Формат

Обычно логические значения извлекаются из оценки выражений сравнения. Однако, слова могут быть установлены на логическое значение и используются для слова **on** и **off**:

```
print-me: false
print either print-me ["turned on"]["turned off"]
turned off
print-me: true
print either print-me ["turned on"]["turned off"]
turned on
```

Значение **false** не соответствует числу ноль или значению **none**. Однако в условных выражениях **false** и **none** имеют такой же эффект:

```
print-me: none
print either print-me ["turned on"]["turned off"]
turned off
```

Практически любое значение, присвоенное слову, имеет тот же эффект, что и **true**:

```
print-me: "just a string"
print either print-me ["turned on"]["turned off"]
turned on
print-me: 11-11-1999
print either print-me ["turned on"]["turned off"]
turned on
```

Следующие слова предопределены для хранения логических значений:

```
true
on      ;чтото истенное
yes     ;чтото истенное
false
off     ;чтото ложное
no      ;чтото ложное
```

Таким образом, вместо **true** и **false** , когда это имеет смысл, можно использовать слова **on** (вкл) и **off** (выкл) или **yes** (да) и **no** (нет):

```
print-me: yes
print either print-me ["turned on"]["turned off"]
turned on
print-me: no
print either print-me ["turned on"]["turned off"]
turned off
print-me: on
print either print-me ["turned on"]["turned off"]
turned on
print-me: off
print either print-me ["turned on"]["turned off"]
turned off
```

3.3.3 Создание

Функция **to-logic** преобразует **integer!** (целое число) или **none!** (нечего) значение к типу данных **logic!**:

```
probe to-logic 0
false
probe to-logic 200
true
probe to-logic none
false
```

```
probe to-logic []
true
probe to-logic "a"
true
probe to-logic none
false
```

3.3.4 Связанные

Используйте `logic?` чтобы определить, является ли значение типо данных `logic!`.

```
probe logic? 1
false
probe logic? on
true
probe logic? false
true
```

Используйте функции `form`, `print` и `mold` для вывода логического значения:

```
probe form true
true
probe mold false
false
print true
true
```

3.4 Деньги

3.4.1 Концепция

Существует множество международных символов для денежных купюр. Некоторые символы используются перед суммой, а некоторые - после. В качестве стандарта для представления международных денежных значений язык REBOL использует денежный формат США, но допускает включение определенных номиналов.

3.4.2 Формат

Тип данных `money!` использует стандартные числа с плавающей запятой IEEE, допускающие до 15 знаков точности, включая центы.

Язык ограничивает длину до 64 символов. Значения, которые находятся вне диапазона или не могут быть представлены 64 символами, помечаются как ошибка.

Денежные значения имеют префикс с необязательным обозначением валюты, за которым следует знак доллара ($). Знак «плюс» (+) или «минус» (-) может стоять непосредственно перед первым символом (обозначением валюты или знаком доллара) для обозначения знака.

```
$123
-$123
$123.45
US$12
US$12.34
-US$12.34
```

```
$12,34
-$12,34
DEM$12,34
```

Чтобы разбить длинные числа на читаемые сегменты, можно поставить одинарную кавычку (`) в любом месте между двумя цифрами в пределах суммы, но не перед суммой.

```
probe $1'234.56
$1234.56
probe $1'234'567,89
$1234567.89
```

Не используйте запятые и точки для разделения больших сумм, поскольку оба эти символа представляют собой десятичные точки.

Тип данных **money!** - это гибридный тип данных. Концептуально деньги скалярны - это сумма денег. Однако, поскольку обозначение валюты хранится в виде строки, тип данных **money!** состоит из двух элементов:

- string! - Строка обозначения валюты, которая может содержать максимум 3 символа.
- decimal! - Сумма денег.

Чтобы продемонстрировать это, следующие деньги указаны с префиксом USD:

```
my-money: USD$12345.67
```

Вот два компонента:

```
probe first my-money
USD
probe second my-money
12345.67
probe pick my-money 3        ; only two components
none
```

Если обозначение валюты не используется, строка обозначения валюты пуста:

```
my-money: $12345.67

probe first my-money
""
probe second my-money
12345.67
```

В обозначении валюты могут быть указаны различные международные валюты, например:

```
my-money: DKM$12'345,67

probe first my-money
DKM
probe second my-money
12345.67
```

3.4.3 Создание

Используйте функцию **to-money**, чтобы конвертировать деньги из **string!**, **integer!**, **decimal!** или з**block!**.

```
probe to-money 123
$123.00
probe to-money "123"
$123.00
probe to-money 12.34
$12.34
probe to-money [DEM 12.34]
DEM$12.34
probe to-money [USA 12 34]
USA$12.34
```

Деньги можно складывать, вычитать и сравнивать с другими деньгами той же валюты. Ошибка возникает, если для таких операций используется другая валюта (автоматические преобразования в настоящее время не предусмотрены).

```
probe $100 + $10
$110.00
probe $100 - $50
$50.00
probe equal? DEM$100.11 DEM$100.11
true
```

Деньги можно умножать и делить на целые и десятичные дроби. Деньги также можно разделить на деньги, получив целое или десятичное число.

```
probe $100 + 11
$111.00
probe $100 / 4
$25.00
probe $100 * 5
$500.00
probe $100 - 20.50
$79.50
probe 10 + $1.20
$11.20
probe 10 - $0.25
$9.75
probe $10 / .50
$20.00
probe 10 * $0.75
$7.50
```

3.4.4 Связанные

Используйте **money?** чтобы определить, является ли тип данных **money!**.

```
probe money? USD$12.34
true
```

Используйте функции form, print и mold с аргументом типа money! для печати денежного значения с обозначением валюты и знаком доллара ($) в виде десятичного числа с двумя цифрами десятичной точности.

```
probe form USD$12.34
USD$12.34
probe mold USD$12.34
USD$12.34
print USD$12.34
USD$12.34
```

3.5 None

3.5.1 Концепция

Тип данных none! содержит одно значение, которое ничего не содержит или не представляет.

Концепция none отличается от пустого блока, пустой строки или нулевого символа. Это реальная значение, олицетворяющее "нечего".

Значение none! может быть возвращено из различных функций, в первую очередь из тех, которые связаны с рядами (например, pick и find).

В REBOL слово none определяется как тип данных none! и содержит значение none!. Слово none не эквивалентно нулю или false. Однако none не интерпретируется многими функциями как false.

Значение none! имеет множество применений, таких как возвращаемое значение из функций ряда, таких как pick, find и select:

```
if (pick series 30) = none [...]
```

В базах данных none может быть заполнителем для пропущенных значений:

```
email-database: [
    "Bobby" bob@rebol.com 40
    "Linda" none 23
    "Sara"  sara@rebol.net 33
]
```

Его также можно использовать как логическое значение:

```
secure none
```

3.5.2 Формат

Слово none предопределено для хранения значения none.
Хотя none не эквивалентен нулю или false, он действителен в условных выражениях и имеет тот же эффект, что и false:

```
probe find "abcd" "e"
none
if find "abcd" "e" [print "found"]
```

3.5.3 Создание

Функция **to-none** всегда возвращает **none**.

3.5.4 Связанные

Используйте **none?** чтобы определить, является ли значение типом данных **none!**.

```
print none? 1
false

print none? find [1 2 3] 4
true
```

Функции **form**, **print** и **mold** выводят значение **none** при передаче аргумента **none**.

```
probe form none
none

probe mold none
none

print none
none
```

3.6 Пара

3.6.1 Концепция

Тип данных **pair!** используется для обозначения пространственных координат, например положения на дисплее. Они используются как для позиций, так и для размеров. Пары используются в основном в REBOL/View.

3.6.2 Формат

Пара указывается как целые числа, разделенные символом **x**.

```
100x50

1024x800

-50x200
```

3.6.3 Создание

Используйте **to-pair** для преобразования блочных или строковых значений в тип данных **pair!**:

```
p: to-pair "640x480"
probe p
640x480
p: to-pair [800 600]
probe p
800x600
```

3.6.4 Связанные

Используйте **pair?** чтобы определить, является ли значение типом данных **pair!**:

```
probe pair? 400x200
true

probe pair? pair
true
```

Пары можно использовать с большинством целочисленных математических операторов:

```
100x200 + 10x20

10x20 * 2x4

100x30 / 10x3

100x100 * 3

10x10 + 3
```

Пары можно просмотреть по их индивидуальным координатам:

```
pair: 640x480
probe first pair
640

probe second pair
480
```

Все значения пар поддерживают уточнения **/x** и **/y**. Эти уточнения позволяют просматривать и изменять координаты отдельных пар.

Просмотр индивидуальных координат:

```
probe pair/x
640

probe pair/y
480
```

Изменение индивидуальных координат:

```
pair/x: 800
pair/y: 600
probe pair

800x600
```

3.7 Уточнение

3.7.1 Концепция

Уточнения - это модификаторы, похожие на прилагательные, используемые в естественных (человеческих) языках. Уточнение указывает на вариант использования или расширение значения функции, объекта, имени файла, URL-адреса или пути. Уточнения всегда имеют символическое значение.

Доработки используются для функций:

```
block: [1 2]

append/only block [3 4]
```

объектов:

```
print system/version
```

файлов:

```
dir: %docs/core

print read dir/file.txt
```

urls:

```
site: http://www.rebol.com

print read site/index.html
```

3.7.2 Формат

Уточнения состоят из косой черты, за которой следует допустимое слово REBOL (определение см. В разделе слов ниже). Примеры:

```
/only
/test1
/save-it
```

Уточнения обычно присоединяются к другим словам, например, в случае:

```
port: open/binary file
```

Но уточнения также могут быть написаны отдельно, как это делается при указании уточнений функции:

```
save-data: function [file data /limit /reload] ...
```

3.7.3 Создание

Уточнения можно создавать буквально в исходном коде:

```
/test
```

или может состоять из слова **to-refinement**:

```
probe to-refinement "test"
/test
```

3.7.4 Связанные

Чтобы проверить уточнение, используйте функцию **refinement?**:

```
probe refinement? /test
true
probe refinement? 'word
false
```

3.8 Время

3.8.1 Концепция

Язык REBOL поддерживает стандартное выражение времени в часах, минутах, секундах и подсекундах. Допускается как положительное, так и отрицательное время.

Тип данных **time!** использует относительное, а не абсолютное время. Например, **10:30** - это 10 часов 30 минут, а не 10:30 AM или PM (до_полудня или после_полудня).

3.8.2 Формат

Время выражается в виде набора целых чисел, разделенных двоеточием (:) .. Часы и минуты являются обязательными, но секунды - необязательными. Внутри каждого поля ведущие нули игнорируются:

```
10:30
0:00
18:59
23:59:50
8:6:20
8:6:2
```

Поля минут и секунд могут содержать значения больше 60. Значения больше 60 преобразуются автоматически. Например, **0:120:00** совпадает с **2:00**.

```
probe 00:120:00
2:00
```

Подсекунды указываются с использованием десятичной дроби в поле секунд. В качестве десятичной точки используйте точку или запятую. Поля часов и минут становятся необязательными, если присутствует десятичная дробь. Подсекунды кодируются с точностью до наносекунды или одной миллиардной секунды:

```
probe 32:59:29.5
32:59:29.5
probe 1:10,25
0:01:10.25
probe 0:0.000000001
0:00:00.000000001
probe 0:325.2
0:05:25.2
```

После времени можно указывать **AM** или **PM**, но пробелы не допускаются. **PM** добавляет к времени 12 часов:

```
probe 10:20PM
22:20
probe 3:32:20AM
3:32:20
```

Время выводится в стандартном формате часов, минут, секунд и подсекунд, независимо от того, как они введены:

```
probe 0:87363.21
24:16:03.21
```

3.8.3 Доступ

Значения времени имеют три уточнения, которые можно использовать для возврата конкретной информации о значении:

Уточнение	Описание
/hour	Получает значение часа.
/minute	Получает значение минуты.
/second	Получает значение секунды.

Вот как использовать уточнения значения времени:

```
lapsed-time: 91:32:12.14
probe lapsed-time/hour
91
probe lapsed-time/minute
32
probe lapsed-time/second
12.14
```

Время с часовыми поясами можно использовать только с д**date!**.

3.8.4 Создание

Время можно преобразовать с помощью функции **to-time**:

```
probe to-time "10:30"
10:30
probe to-time [10 30]
10:30
probe to-time [0 10 30]
0:10:30
probe to-time [10 30 20.5]
10:30:20.5
```

В предыдущих примерах значения не оцениваются. Чтобы оценить значения как математические выражения, используйте функцию уменьшения:

```
probe to-time reduce [10 30 + 5]
10:35
```

В различных математических операциях, связанных со значениями времени, значения времени, целые или десятичные числа преобразуются, как показано ниже:

```
probe 10:30 + 1
10:30:01
probe 10:00 - 10
9:59:50
probe 0:00 - 10
-0:00:10
probe 5:10 * 3
15:30
probe 0:0:0.000000001 * 1'500'600
0:00:00.0015006
probe 8:40:20 / 4
2:10:05
probe 8:40:20 / 2:20:05
3
probe 8:40:20 // 4:20
0:00:20
```

3.8.5 Связанные

Используете **time?** чтобы определить, является ли значение типом данных **time!**:

```
probe time? 10:30
true
probe time? 10.30
false
```

Используйте функцию **now** с уточнением **/time**, чтобы вернуть текущую местную дату и время:

```
print now/time
14:42:15
```

Используйте функцию **wait**, чтобы дождаться продолжительности, порта или того и другого. Если значение - это тип данных **time!**, **wait** (ждать) делает задержку в течение этого периода времени. Если значением является **date!/time!**, **wait** ожидает, пока не истечет указанная дата и время. Если значение **integer!** (целое!) или **decimal!** (десятичный!), функция ждет указанное количество секунд. Если значением является **port** (порт), функция будет ожидать события от этого порта. Если блок указан, он будет ждать, пока не произойдет какое-либо время или порты. Он возвращает порт, который вызвал завершение ожидания, или возвращает **none**, если истекло время ожидания. Например,

```
probe now/time
14:42:16
wait 0:00:10
probe now/time
14:42:26
```

3.9 Кортеж

3.9.1 Концепция

Обычно номера версий, адреса в Интернете и значения цветов RGB представляют как последовательность из трех или четырех целых чисел. Эти типы чисел называются **tuple!**(кортежами!) и представлены как набор целых чисел, разделенных точками.

```
1.3.0 2.1.120 1.0.2.32     ; версия
199.4.80.250 255.255.255.0 ; сетевой адрес и маска
0.80.255 200.200.60        ; RGB цвета
```

3.9.2 Формат

Каждое целочисленное поле типа данных **tuple!** может находиться в диапазоне от 0 до 255. Отрицательные целые числа вызывают ошибку.
В кортеже можно указать от трех до десяти целых чисел. В случае, когда задано только два целых числа, должно быть не менее двух точек, в противном случае значение рассматривается как десятичное.

```
probe 1.2     ; is decimal
1.2
probe type? 1.2
decimal!
probe 1.2.3   ; is tuple
1.2.3
probe 1.2.    ; is tuple
1.2.0
probe type? 1.2.
tuple!
```

3.9.3 Создание

Используйте функцию **to-tuple** для преобразования данных в тип данных **tuple!**:

```
probe to-tuple "12.34.56"
12.34.56
probe to-tuple [12 34 56]
12.34.56
```

3.9.4 Связанные

Используйте **tuple?** чтобы определить, является ли значение типом данных **tuple!**.

```
probe tuple? 1.2.3.4
true
```

Используйте функцию **form** для печати кортежа в виде строки:

```
probe form 1.2.3.4
1.2.3.4
```

Используйте функцию **mold**, чтобы преобразовать кортеж в строку, которую можно будет прочитать обратно в REBOL как кортеж:

```
probe mold 1.2.3.4
1.2.3.4
```

Используйте функцию **print**, чтобы распечатать кортеж на стандартный вывод после использования функции **reform**:

```
print 1.2.3.4
1.2.3.4
```

3.10 Слова

3.10.1 Концепция

Слова - это символы, используемые REBOL. Слово может быть или не быть переменной, в зависимости от того, как оно используется. Слова часто используются непосредственно как символы.

REBOL не имеет ключевых слов, нет ограничений на то, какие слова используются или как они используются. Например, вы можете определить свою собственную функцию с именем **print** и использовать ее вместо предопределенной функции для печати значений.

В зависимости от требуемой операции существует четыре различных формата использования слов.

Действие	Создание	Проверка	ПРеобразование
word:	set-word!	set-word?	to-set-word
:word	get-word!	get-word?	to-get-word
word	word!	word?	to-word
'word	lit-word!	lit-word?	to-lit-word

3.10.2 Формат

Слова состоят из букв, цифр и любых из следующих символов:

```
? ! . ' + - * & | = _ &#126;
```

Слово не может начинаться с числа, а также существуют некоторые ограничения на слова, которые можно интерпретировать как числа. Например, -1 и +1 - это числа, а не слова.

Конец слова отмечается пробелом, новой строкой или одним из следующих символов:

```
[ ] ( ) { } " : ; /
```

Таким образом, квадратные скобки блока не являются частью слова:

```
[test]
```

В словах нельзя использовать следующие символы:

```
@ # $ % ^ ,
```

Слова могут быть любой длины, но не могут выходить за пределы конца строки.

```
this-is-a-very-long-word-used-as-an-example
```

Примеры слов:

```
Copy print test

number?  time?  date!

image-files  l'image

++ -- == +-

***** *new-line*

left&right left|right
```

В языке REBOL регистр не учитывается. Слова следующие за словами:

```
blue

Blue

BLUE
```

все относятся к одному и тому же слову. Регистр слова сохраняется при печати.

Слова можно использовать повторно. Значение слова зависит от его контекста, поэтому слова можно повторно использовать в разных контекстах. Вы можете повторно использовать любое слово, даже заранее определенные слова REBOL. Например, слово REBOL `if` может использоваться в вашем коде иначе, чем оно используется интерпретатором REBOL.

3.10.3 Создание

Функция `to-word` преобразует значения в тип данных `word!`.

```
probe to-word "test"
test
```

Функция `to-set-word` преобразует значения в тип данных`set-word!`.

```
probe make set-word! "test"
test:
```

Функция `to-get-word` преобразует значения в тип данных `get-word!`.

```
probe to-get-word "test"
:test
```

Функция `to-lit-word` преобразует значения в световое тип данных `lit-word!`.

```
probe to-lit-word "test"
'test
```

3.10.4 Связанные

Используйте `word?`, `set-word?`, `get-word?` и `lit-word?` чтобы проверить тип данных.

```
probe word? second [1 two "3"]
true
if set-word? first [word: 10] [print "it is set"]
it is set
probe get-word? second [pr: :print]
true
probe lit-word? first ['foo bar]
true
```

Приложение 2 - Ошибки

Содержание:

1. Обзор
2. Категории ошибок
  2.1 Синтаксические ошибки
  2.2 Ошибки сценария
  2.3 Математические ошибки
  2.4 Ошибки доступа
  2.5 Ошибки пользователя
  2.6 Внутренние ошибки
3. Перехват ошибок
4. Объект ошибки
5. Генерация ошибок
6. Сообщения об ошибках
  6.1 Синтаксические ошибки
  6.2 Ошибки сценария
  6.3 Ошибки доступа
  6.4 Внутренние ошибки

1. Обзор

Ошибки - это исключения, которые возникают при возникновении определенных нестандартных условий. Эти условия варьируются от синтаксических ошибок до ошибок доступа к файлам или сети. Вот несколько примеров:

```
12-30
** Syntax Error: Invalid date -- 12-30.
** Where: (line 1) 12-30
1 / 0
** Math Error: Attempt to divide by zero.
** Where: 1 / 0
read %nofile.r
** Access Error: Cannot open /example/nofile.r.
** Where: read %nofile.r
```

Ошибки обрабатываются внутри системы как значения типов данных **error!**. Ошибка - это объект, который при оценке выводит сообщение об ошибке и останавливается. Вы также можете обнаруживать ошибки и обрабатывать их в своем скрипте. Ошибки можно передавать в функции, возвращать из функций и присваивать переменным.

2. Категории ошибок

Есть несколько категорий ошибок.

2.1 Ошибки синтаксиса

Синтаксические ошибки возникают, когда сценарий неправильно использует синтаксис REBOL. Например, если закрывающая скобка отсутствует или в строке отсутствует закрывающая кавычка, произойдет синтаксическая ошибка. Эти ошибки возникают только во время загрузки или оценки файла или строки.

2.2 Ошибки скрипта

Ошибки сценария - это общие ошибки времени выполнения. Например, неверный аргумент функции вызовет ошибку сценария.

2.3 Математические ошибки

Математические ошибки возникают, когда математическая операция не может быть обработана. Например, при попытке разделить на ноль возникнет ошибка.

2.4 Ошибки доступа

Ошибки доступа возникают, когда возникает проблема с доступом к файлу, порту или сети. Например, ошибка доступа произойдет при попытке прочитать несуществующий файл.

2.5 Ошибки пользователей

Пользовательские ошибки генерируются явным образом сценарием, создавая значение ошибки и возвращая его.

2.6 Внутренние ошибки

Внутри интерпретатора REBOL возникают внутренние ошибки.

3. Выявление ошибок

Вы можете отловить ошибки с помощью функции **try**. Функция **try** аналогична функции **do**. Он оценивает блок, но всегда возвращает значение, даже если возникает ошибка.
Если ошибки не возникает, **try** возвращает значение блока. Например:

```
print try [100 / 10]
10
```

При возникновении ошибки попытка возвращает ошибку. Если вы напишете:

```
print try [100 / 0]
** Math Error: Attempt to divide by zero.
** Where: 100 / 0
```

ошибка возвращается при **try**, и функция печати **print** не может ее обработать.
Чтобы обрабатывать ошибки в сценарии, вы должны предотвратить оценку ошибки REBOL. Вы можете предотвратить оценку ошибки, передав её функции. Например, функция **error?** вернет истину при возниктовении ошибки:

```
print error? try [100 / 0]
true
```

Вы также можете распечатать тип данных значения, возвращенного при попытке:

```
print type? try [100 / 0]
error!
```

Функция **disarm** (снятия с охраны) преобразует ошибку в объект ошибки, который можно исследовать. В приведенном ниже примере переменная ошибки содержит объект ошибки:

```
error: disarm try [100 / 0]
```

Когда ошибка снята с охраны, это будет тип данных object!, а не тип данных error!. Оценка снятого с охраны объекта не вызовет ошибки:

```
probe disarm try [100 / 0]
make object! [
    code: 400
    type: 'math
    id: 'zero-divide
    arg1: none
    arg2: none
    arg3: none
    near: [100 / 0]
    where: none
]
```

Значения ошибок могут быть установлены в слово перед снятием с охраны. Чтобы присвоить слову ошибку, ему должна предшествовать функция, предотвращающая дальнейшее распространение ошибки. Например:

```
disarm err: try [100 / 0]
```

Установка переменной позволяет вам получить доступ к значению блока позже. В приведенном ниже примере будет напечатано значение ошибки или не ошибка:

```
either error? result: try [100 / 0] [
    probe disarm result
][
    print result
]
```

4. Объект ошибки

Показанный выше объект ошибки имеет структуру:

```
make object! [
    code: 400
    type: 'math
    id: 'zero-divide
    arg1: none
    arg2: none
    arg3: none
    near: [100 / 0]
    where: none
]
```

Где находятся поля:

code	Номер кода ошибки. Они устарели и не должны использоваться.
type	Поле типа определяет категорию ошибки. Это всегда словесный тип данных syntax (синтаксис), script (сценарий), math (математика), access (доступ), user (пользователь) и internal (внутренний).
id	Поле id - это название ошибки в виде слова. Он определяет конкретную ошибку, которая произошла в категории ошибок.
arg1	Это поле содержит первый аргумент сообщения об ошибке. Например, он может включать тип данных значения, вызвавшего ошибку.
arg2	Это поле содержит второй аргумент сообщения об ошибке.
arg3	Это поле содержит третий аргумент сообщения об ошибке.
near	Ближнее поле - это фрагмент кода, который показывает, где произошла ошибка.
where	Поле where зарезервировано.

Вы можете написать код, который проверяет любое из полей объекта ошибки. В этом примере ошибка выводится только тогда, когда идентификатор ошибки указывает на ошибку деления на ноль:

```
error: disarm try [1 / 0]
if error/id = 'zero-divide [
    print {It is a Divide by Zero error}
]
It is a Divide by Zero error
```

Слово идентификатора ошибки также предоставляет блок ошибки, который будет напечатан интерпретатором. Например:

```
error: disarm try [print zap]
probe get error/id
[:arg1 "has no value"]
```

Этот блок определяется объектом система/ошибки.

5. Создание ошибок

Могут возникнуть ошибки пользователя. Самый простой способ сгенерировать ошибку - это **make**. Вот пример:

```
make error! "this is an error"
** User Error: this is an error.
** Where: make error! "this is an error"
```

Любая из существующих ошибок может быть сгенерирована путем внесения ошибки с помощью аргумента блока. Этот блок содержит имя категории ошибки и имя идентификатора сообщения об ошибке. Если для ошибки требуются аргументы, аргументы следуют за именем идентификатора сообщения. Аргументы определяют значения `arg1`, `arg2` и `arg3` в объекте ошибки. Вот пример:

```
make error! [script expect-set series! number!]
** Script Error: Expected one of: series! - not: number!.
** Where: make error! [script expect-set series! number!]
```

Пользовательские ошибки могут быть внесены в категорию `user` объекта `system/error` (система/ошибка). Это делается путем создания новой категории пользователей с новыми записями. Эти записи используются при создании ошибок. Например, следующий пример вводит ошибку в категорию пользователя:

```
system/error/user: make system/error/user [
    my-error: "a simple error"
]
```

Теперь ошибка может быть сгенерирована с использованием идентификатора сообщения `my-error`:

```
if error? err: try [
    make error! [user my-error]
] [probe disarm err]
make object! [
    code: 803
    type: 'user
    id: 'my-error
    arg1: none
    arg2: none
    arg3: none
    near: [make error! [user my-error]]
    where: none
]
```

Чтобы создать более информативные ошибки, определите ошибку, которая использует данные, доступные при ее создании. Эти данные включаются в снятый с охраны объект ошибки и печатаются как часть сообщения об ошибке. Например, чтобы использовать все три пробела аргументов в объекте ошибки:

```
system/error/user: make system/error/user [
    my-error: [:arg1 "doesn't go into" :arg2 "using" :arg3]
]

if error? err: try [
    make error! [user my-error [this] "that" my-function]
] [probe disarm err]
make object! [
    code: 803
    type: 'user
    id: 'my-error
    arg1: [this]
    arg2: "that"
    arg3: 'my-function
    near: [make error! [user my-error [this] "that" my-function]]
    where: none
]
```

Сообщение об ошибке, созданное для `my-error`, можно распечатать, не останавливая скрипт:

```
disarmed: disarm err
print bind (get disarmed/id) (in disarmed 'id)
this doesn't go into that using my-function
```

Новая категория библиотеки может быть создана, если есть необходимость сгруппировать серию ошибок вместе, создав новую категорию в `system/error:`:

```
system/error: make system/error [
    my-errors: make object! [
        code: 1000
        type: "My Error Category"
        error1: "a simple error"
        error2: [:arg1 "doesn't go into" :arg2 "using" :arg3]
    ]
]
```

Тип определен в объекте ошибки будет напечатан типом ошибки , когда генерируются ошибка. В следующем примере показано создание ошибки как из `error1`, так и из `error2` в категории `my-error`.

Генерация ошибки из `error1`. Эта ошибка не требует аргументов:

```
disarmed: disarm try [make error! [my-errors error1]]
print get disarmed/id
a simple error
```

Для генерации ошибки из `error2` требуются три аргумента:

```
disarmed: disarm try [
make error! [my-errors error2 [this] "that" my-function]]
print bind (get disarmed/id) (in disarmed 'id)
this doesn't go into that using my-function
```

Наконец, описание, которое возвращает ошибки, определенные в `my-error`, можно получить с помощью:

```
probe get in get disarmed/type 'type
My Error Category
```

6. Сообщения об ошибках

Ниже приведен список всех ошибок, определенных в каталоге ошибок объекта system/error.

6.1 Синтаксические ошибки

6.1.1 недействительно

Данные не могут быть преобразованы в допустимый тип данных REBOL. Другими словами, было оценено искаженное значение.

Сообщение:

```
["Invalid" :arg1 "--" :arg2]
```

Пример:

```
filter-error try [load "1024AD"]
** Syntax Error: Invalid integer -- 1024AD
** Where: (line 1) 1024AD
```

6.1.2 отсутствует

Блок, строка или выражение-скобка остались незакрытыми.

Сообщение:

```
["Missing" :arg2 "at" :arg1]
```

Пример:

```
filter-error try [load "("]
** Syntax Error: Missing ) at end-of-script
** Where: (line 1) (
```

6.1.3 заголовок

Была сделана попытка оценить файл как сценарий REBOL, но файл не имел заголовка REBOL.

Сообщение:

```
Script is missing a REBOL header
```

Пример:

```
write %no-header.r {print "data"}
filter-error try [do %no-header.r]
** Syntax Error: Script is missing a REBOL header
** Where: do %no-header.r
```

6.2 Ошибки скрипта

6.2.1 без значения

Была сделана попытка оценить неопределенное слово.

Сообщение:

```
[:arg1 "has no value"]
```

Пример:

```
filter-error try [undefined-word]
** Script Error: undefined-word has no value
** Where: undefined-word
```

6.2.2 need-value (потребность)

Была сделана попытка определить слово ни к чему. Установочное слово использовалось без аргументов.

Сообщение:

```
[:arg1 "needs a value"]
```

Пример:

```
filter-error try [set-to-nothing:]
** Script Error: set-to-nothing needs a value
** Where: set-to-nothing:
```

6.2.3 no-arg

Функция была оценена без предоставления всех ожидаемых аргументов.

Сообщение:

```
[:arg1 "is missing its" :arg2 "argument"]
```

Пример:

```
f: func [b][probe b]
filter-error try [f]
** Script Error: f is missing its b argument
** Where: f
```

6.2.4 expect-arg (ожидаемый аргумент)

Функция предоставила аргумент типа данных, которого она не ожидала.

Сообщение:

```
[:arg1 "expected" :arg2 "argument of type:" :arg3]
```

Пример:

```
f: func [b [block!]][probe b]
filter-error try [f "string"]
** Script Error: f expected b argument of type: block
** Where: f "string"
```

6.2.5 expect-set (ожидаемый набор)

Два ряда значений использовались вместе несовместимым образом. Например, при попытке объединения (**union**) строки и блока.

Сообщение:

```
["Expected one of:" :arg1 "- not:" :arg2]
```

Пример:

```
filter-error try [union [a b c] "a b c"]
** Script Error: Expected one of: block! - not: string!
** Where: union [a b c] "a b c"
```

6.2.6 invalid-arg (неверный аргумент)

Это общая ошибка обработки значений, которые использовались неправильно. Например, когда заданное слово используется внутри блока спецификации функции.

Сообщение:

```
["Invalid argument:" :arg1]
```

Пример:

```
filter-error try [f: func [word:][probe word]]
** Script Error: Invalid argument: word
** Where: func [word:] [probe word]
```

6.2.7 invalid-op (недействительный)

Была сделана попытка использовать переопределенный оператор. Используемый оператор больше не является допустимым оператором.

Сообщение:

```
["Invalid operator:" :arg1]
```

Пример:

```
*: "operator redefined to a string"
filter-error try [5 * 10]
** Script Error: Invalid operator: *
** Where: 5 * 10
```

6.2.8 no-op-arg

Был использован математический оператор или оператор сравнения без предоставления второго аргумента.

Сообщение:

```
Operator is missing an argument
```

Пример:

```
filter-error try [1 +]
** Script Error: Operator is missing an argument
** Where: 1 +
```

6.2.9 no-return (невозврат)

Функция, ожидающая, что блок вернет значение, но ничего не вернула. Например, при использовании функции **while** или **until**.

Сообщение:

```
Block did not return a value
```

Примеры:

```
filter-error try [ ; first block returns nothing
    while [print 10][probe "ten"]
]
10
** Script Error: Block did not return a value
** Where: while [print 10] [probe "ten"]
filter-error try [
    until [print 10] ; block returns nothing
]
10
** Script Error: Block did not return a value
** Where: until [print 10]
```

6.2.10 not-defined (не определено)

Используемое слово не было определено ни в каком контексте.

Сообщение:

```
[:arg1 "is not defined in this context"]
```

6.2.11 no-refine (без уточнения)

Была сделана попытка использовать уточнение функции, которой не было для этой функции.

Сообщение:

```
[:arg1 "has no refinement called" :arg2]
```

Пример:

```
f: func [/a] [if a [print "a"]]
filter-error try [f/b]
** Script Error: f has no refinement called b
** Where: f/b
```

6.2.12 invalid-path (неверный путь)

Была сделана попытка получить доступ к значению блока или объекта, используя путь, которого не было в этом блоке или объекте.

Сообщение:

```
["Invalid path value:" :arg1]
```

Пример:

```
blk: [a "a" b "b"]
filter-error try [print blk/c]
** Script Error: Invalid path value: c
** Where: print blk/c
obj: make object! [a: "a" b: "b"]
filter-error try [print obj/d]
** Script Error: Invalid path value: d
** Where: print obj/d
```

6.2.13 cannot-use (нельзя использовать)

Была сделана попытка выполнить операцию со значением несовместимого типа данных. Например, при попытке добавить строку к числу.

Сообщение:

```
["Cannot use" :arg1 "on" :arg2 "value"]
```

Пример:

```
filter-error try [1 + "1"]
** Script Error: Cannot use add on string! value
** Where: 1 + "1"
```

6.2.14 already-used (уже используется)

Была предпринята попытка присвоить псевдоним (**alias**) слову, которое уже было псевдонимом.

Сообщение:

```
["Alias word is already in use:" :arg1]
```

Пример:

```
alias 'print "prink"
filter-error try [alias 'probe "prink"]
** Script Error: Alias word is already in use: prink
** Where: alias 'probe "prink"
```

6.2.15 out-of-range (вне допустимого диапазона)

Была сделана попытка изменить недопустимый индекс серии.

Сообщение:

```
["Value out of range:" :arg1]
```

Пример:

```
blk: [1 2 3]
filter-error try [poke blk 5 "five"]
** Script Error: Value out of range: 5
** Where: poke blk 5 "five"
```

6.2.16 past-end

Была сделана попытка получить доступ к данным ряда за пределами длины ряда.

Сообщение:

```
Out of range or past end
```

Сообщение:

```
blk: [1 2 3]
filter-error try [print fourth blk]
** Script Error: Out of range or past end
** Where: print fourth blk
```

6.2.17 no-memory (без памяти)

Системе не хватило памяти при попытке завершить операцию.

Сообщение:

```
Not enough memory
```

6.2.18 wrong-denom (неправильный деном)

Была произведена математическая операция с денежными знаками двух разных номиналов. Например, при попытке прибавить 1 доллар США к 1,50 DEN.

Сообщение:

```
[:arg1 "not same denomination as" :arg2]
```

Пример:

```
filter-error try [US$1.50 + DM$1.50]
** Script Error: US$1.50 not same denomination as DM$1.50
** Where: US$1.50 + DM$1.50
```

6.2.19 bad-press (плохая пресса)

Была сделана попытка распаковать двоичное значение, которое было повреждено или не было сжатым форматом.

Сообщение:

```
["Invalid compressed data - problem:" :arg1]
```

Пример:

```
compressed: compress {some data}
change compressed "1"
filter-error try [decompress compressed]
** Script Error: Invalid compressed data - problem: -3
** Where: decompress compressed
```

6.2.20 bad-port-action (плохой порт-действие)

Была сделана попытка выполнить неподдерживаемое действие над портом. Например, при попытке использовать `find` на TCP-порту.

Сообщение:

```
["Cannot use" :arg1 "on this type port"]
```

6.2.21 needs (потребности)

Была предпринята попытка запустить сценарий, для которого требовалась либо новая версия REBOL, либо файл, который не удалось найти. Эта информация будет найдена в заголовке сценария REBOL.

Сообщение:

```
["Script needs:" :arg1]
```

6.2.22 locked-word (заблокированное слово)

Была сделана попытка изменить защищенное слово. Слово будет защищено функцией **protect** (защиты).

Сообщение:

```
["Word" :arg1 "is protected, cannot modify"]
```

Пример:

```
my-word: "data"
protect 'my-word
filter-error try [my-word: "new data"]
** Script Error: Word my-word is protected, cannot modify
** Where: my-word: "new data"
```

6.2.23 dup-vars

Была оценена функция, в которой в блоке спецификации было несколько вхождений слова. Например, если слово `arg` было определено как аргумент один и два.

Сообщение:

```
["Duplicate function value:" :arg1]
```

Пример:

```
filter-error try [f: func [a /local a][print a]]
** Script Error: Duplicate function value: a
** Where: func [a /local a] [print a]
```

6.3 Ошибки доступа

6.3.1 cannot-open (не открывается)

Нет доступа к файлу. Это может быть локальный или сетевой файл. Наиболее частая причина этой ошибки - несуществующий каталог.

Сообщение:

```
["Cannot open" :arg1]
```

Пример:

```
filter-error try [read %/c/path-not-here]
** Access Error: Cannot open /c/path-not-here
** Where: read %/c/path-not-here
```

6.3.2 not-open (не открыто)

Была сделана попытка использовать закрытый порт.

Сообщение:

```
["Port" :arg1 "not open"]
```

Сообщение:

```
p: open %file.txt
close p
filter-error try [copy p]
** Access Error: Port file.txt not open
** Where: copy p
```

6.3.3 already-open (уже открыт)

Была сделана попытка открыть (`open`) порт, который уже был открыт.

Сообщение:

```
["Port" :arg1 "already open"]
```

Пример:

```
p: open %file.txt
filter-error try [open p]
** Access Error: Port file.txt already open
** Where: open p
```

6.3.4 already-closed (уже закрыто)

Была предпринята попытка закрыть (`close`) уже закрытый порт.

Сообщение:

```
["Port" :arg1 "already closed"]
```

Пример:

```
p: open %file.txt
close p
filter-error try [close p]
** Access Error: Port file.txt not open
** Where: close p
```

6.3.5 invalid-spec (недействительная спецификация)

Была сделана попытка создать порт с помощью **make**, используя спецификацию, по которой порт не может быть построен.

Сообщение:

```
["Invalid port spec:" :arg1]
```

Пример:

```
filter-error try [p: make port! [scheme: 'naughta]]
** Access Error: Invalid port spec: scheme naughta
** Where: p: make port! [scheme: 'naughta]
```

6.3.6 socket-open (открыт сокет)

В операционной системе закончились выделенные сокеты.

Сообщение:

```
["Error opening socket" :arg1]
```

6.3.7 no-connect (без подключения)

Не удалось подключиться к другому хосту. Это общая ошибка, охватывающая ряд причин сбоя подключения. Когда станет известно больше информации о причине сбоя подключения, будет выдана более конкретная ошибка.

Сообщение:

```
["Cannot connect to" :arg1]
```

Пример:

```
filter-error try [read http://www.host.dom/]
** Access Error: Cannot connect to www.host.dom
** Where: read http://www.host.dom/
```

6.3.8 no-delete (запретить удаление)

Была предпринята попытка удалить файл (**delete**), который был заблокирован или защищен.

Сообщение:

```
["Cannot delete" :arg1]
```

Пример:

```
p: open %file.txt
filter-error try [delete %file.txt]
** Access Error: Cannot delete file.txt
** Where: delete %file.txt
```

6.3.9 no-rename (без переименования)

Была сделана попытка переименовать файл (**rename**), который был заблокирован или защищен.

Сообщение:

```
["Cannot rename" :arg1]
```

Пример:

```
p: open %file.txt
filter-error try [rename %file.txt %new-name.txt]
** Access Error: Cannot rename file.txt
** Where: rename %file.txt %new-name.txt
```

6.3.10 no-make-dir

Была сделана попытка создать каталог по пути к файлу, который не существует или был защищен от записи.

Сообщение:

```
["Cannot make directory" :arg1]
```

Пример:

```
filter-error try [make-dir %/c/no-path/dir]
** Access Error: Cannot make directory /c/no-path/dir/
** Where: m-dir path return path
```

6.3.11 timeout

Время ожидания ответа от другого хоста истекло. Этот тайм-аут устанавливается в атрибуте timeout порта.

Сообщение:

```
Network timeout
```

6.3.12 new-level (новый уровень)

В сценарии была сделана попытка изменить безопасность на более низкий уровень безопасности, в котором было отказано. Это означает, что всякий раз, когда сценарий запрашивает более низкие настройки безопасности, а пользователь отклоняет запрос, возникает эта ошибка.

Сообщение:

```
["Attempt to change security level to" :arg1]
```

Пример:

```
secure quit
filter-error try [secure none] ; denied request

secure none
```

6.3.13 security (безопасность)

Произошло нарушение безопасности. Это произойдет, когда будет сделана попытка получить доступ к файлу или сети, когда для параметра **secure** установлено значение throw.

Сообщение:

```
REBOL - Security Violation
```

Пример:

```
secure throw
filter-error try [open %file.txt]
** Access Error: REBOL - Security Violation
** Where: open %file.txt
secure none
```

6.3.14 invalid-path (неверный путь)

Использован неверный путь к файлу.

Сообщение:

```
["Bad file path:" :arg1]
```

Пример:

```
filter-error try [read %/]
```

6.4 Внутренние ошибки

6.4.1 bad-path (неверный путь)

Был оценен путь, который начинается с недопустимого слова.

Сообщение:

```
["Bad path:" arg1]
```

Пример:

```
path: make path! [1 2 3]
filter-error try [path]
** Internal Error: Bad path: 1
** Where: path
```

6.4.2 not-here (не здесь)

Была сделана попытка использовать функцию REBOL/Command или REBOL/View из REBOL/Core.

Сообщение:

```
[arg1 "not supported on your system"]
```

6.4.3 stack-overflow (переполнение стека)

Стек системной памяти переполнен при попытке выполнить операцию.

Сообщение:

```
["Stack overflow"]
```

Пример:

```
call-self: func [][call-self]
filter-error try [call-self]
** Internal Error: Stack overflow
** Where: call-self
```

6.4.4 globals-full (глобальное переполнение)

Превышено максимально допустимое количество определенных глобальных слов.
Сообщение:

```
["No more global variable space"]
```

Приложение 3 – Консоль

Содержание:

1. Командная строка
2. Индикатор результата
3. Вызов истории
4. Индикатор занятости
5. Расширенные операции с консолью
5.1 Последовательности ввода с клавиатуры
5.2 Последовательности вывода терминала

1. Командная строка

Приглашение командной строки по умолчанию - ">>". Вы можете изменить приглашение с помощью такого кода, как:

```
system/console/prompt: "Input: "
```

Тогда приглашение станет:

```
Input:
```

Подсказка может быть блоком, который оценивается каждый раз. Эта строка печатает текущее время:

```
system/console/prompt: [reform [now/time " >> "]]
```

Это приведет к появлению запроса:

```
10:30 >>
```

2. Индикатор результата

Индикатор результата по умолчанию - "==" и может быть изменен с помощью такой строки, как:

```
system/console/result: "Result: "
```

Эти настройки можно передать `user.r`, чтобы сделать их постоянными.

3. Напоминание истории

Каждая строка, введенная в REBOL в командной строке, сохраняется в блоке истории, и ее можно вызвать позже, используя клавиши со стрелками вверх и вниз. Например, однократное нажатие стрелки вверх вызывает предыдущую строку ввода.

Доступ к блоку истории, содержащему все строки ввода, осуществляется из объекта системной консоли:

```
probe system/console/history
```

Вы можете сохранить блок истории в виде файла:

```
save %history.r system/console/history
```

и его можно перезагрузить позже с помощью:

```
system/console/history: load %history.r
```

Эти строчки можно вводить `user.r` для сохранения и перезагрузки вашей истории между сессиями REBOL.

4. Индикатор занятости

Когда REBOL ожидает завершения сетевой операции, на экране появляется индикатор занятости, указывающий на то, что что-то происходит. Вы можете изменить индикатор с помощью такой строки:

```
system/console/busy: "123456789-"
```

Когда REBOL работает в тихом режиме, индикатор занятости отображаться не будет.

5. Расширенные операции с консолью

Консоль предоставляет возможность «виртуального терминала», которая позволяет выполнять такие операции, как перемещение курсора, адресация курсора, редактирование строки, очистка экрана, ввод управляющих клавиш и запрос положения курсора.

Последовательности управления с консоли соответствуют стандарту ANSI. Эти функции предоставляют вам возможность писать собственные терминальные программы, не зависящие от платформы, такие как текстовые редакторы, почтовые клиенты или эмуляторы telnet.

Функции консоли применимы как к вводу, так и к выводу. При вводе функциональные клавиши будут преобразованы в многосимвольные escape-последовательности. На выходе можно использовать многосимвольные escape-последовательности для управления отображением текста в окне консоли. И входная, и выходная последовательности начинаются с escape-символа ANSI, 27 десятичного числа (1B в шестнадцатеричном формате). Следующий символ в последовательности указывает управляющие клавиши на входе или операцию управления терминалом на выходе.

Управляющие символы ANSI чувствительны к регистру и обычно требуют символа верхнего регистра.

5.1 Последовательности ввода с клавиатуры

Последовательности ввода для функциональных клавиш перечислены в таблице ниже (в системах и оболочках, которые их поддерживают, таких как Linux, BSD и т.д.).

Чтобы получить эти последовательности в виде потока необработанных символов, отключите режим обработки строки входного порта:

```
set-modes system/ports/input [lines: false]
```

Теперь вы можете получить ввод из порта (с помощью COPY или READ-IO) или использовать такую функцию, как INPUT, для получения каждого символа:

```
while [
    code: input
    code <> 13  ; ENTER
][
    probe code
]
```

Вот несколько общих кодов входных функций ANSI:

Функциональная клавиша	Escape Code	REBOL Block
F1	ESC O P	[27 79 80]
F2	ESC O Q	[27 79 81]
F3	ESC O R	[27 79 82]
F4	ESC O S	[27 79 83]
F5	ESC [1 5 ~	[27 91 49 53 126]
F6	ESC [1 7 ~	[27 91 49 55 126]
F7	ESC [1 8 ~	[27 91 49 56 126]
F8	ESC [1 9 ~	[27 91 49 57 126]
F9	ESC [2 0 ~	[27 91 50 48 126]
F10	ESC [2 1 ~	[27 91 50 49 126]
F11	ESC [2 2 ~	[27 91 50 50 126]
F12	ESC [2 3 ~	[27 91 50 51 126]
Home	ESC [1 ~	[27 91 49 126]
End	ESC [4 ~	[27 91 52 126]
Page-up	ESC [5 ~	[27 91 53 126]
Page-down	ESC [6 ~	[27 91 54 126]
Insert	ESC [2 ~	[27 91 50 126]
Up	ESC [A	[27 91 65]
Down	ESC [B	[27 91 66]
Left	ESC [D	[27 91 68]
Right	ESC [C	[27 91 67]

5.2 Последовательности выходных символов

Существует несколько вариантов последовательностей выходных символов управления терминалом. Некоторым кодам команд предшествует число (отправленное в формате ASCII), указывающее, что операция должна выполняться указанное количество раз. Например, команде перемещения курсора могут предшествовать два числа, разделенных точкой с запятой, чтобы указать позицию строки и столбца, к которым нужно перейти. Командные символы курсора (требуется верхний регистр) включены в следующую таблицу:

Последовательность вывода	Описание
(1B)	Используйте этот escape-код перед следующими кодами
D	Перемещает курсор на одну позицию влево
C	Перемещает курсор на одну позицию вправо
A	Перемещает курсор на одну позицию вверх
B	Перемещает курсор на одну позицию вниз
n D	Перемещает курсор на n пробелов влево
n C	Перемещает курсор на n пробелов вправо
n A	Перемещает курсор на n пробелов вверх
n B	Перемещает курсор на n пробелов вниз
r ; c H	Перемещает курсор в строку r, столбец c *
H	Перемещает курсор в верхний левый угол (домой) *
P	Удаляет один символ справа в текущем местоположении
n P	Удаляет n символов справа в текущем местоположении
@	Вставляет одно пустое пространство в текущее местоположение
n @	Вставляет n пробелов в текущее местоположение
J	Очищает экран и перемещает курсор в верхний левый угол (домой) *
K	Очищает от текущей позиции до конца текущей строки
6n	Помещает текущую позицию курсора во входной буфер
7n	Помещает размеры экрана во входной буфер

Верхний левый угол определяется как строка 1, столбец 1

В следующем примере курсор перемещается на десять правых пробелов:

```
print "^(1B)[10CHi!"
Hi
```

В этом примере курсор перемещается на семь левых пробелов и очищается оставшаяся часть строки:

```
cursor: func [parm [string!]][join "^(1B)[" parm]
print ["How are you" cursor "7D" cursor "K"]
How a
```

Чтобы узнать текущий размер окна консоли, вы можете использовать этот пример:

```
cons: open/binary [scheme: 'console]

print cursor "7n"
screen-dimensions: next next to-string copy cons
33;105R
close cons
```

В приведенном выше примере открывается консоль, отправляется управляющий символ во входной буфер и копируется возвращаемое значение. Он считывает значение (размеры экрана), которое возвращается после управляющего символа, и закрывает консоль. Возвращаемое значение - высота и ширина, разделенные точкой с запятой (;), за которой следует R. В приведенном выше примере экран имеет высоту 33 и ширину 105.

Автопрокрутка

Печать символа в правом нижнем углу некоторых терминалов приведет к появлению новой строки, которая будет прокручивать экран. Других не будет. Это несоответствие между типами консольных терминалов необходимо учитывать при написании скриптов REBOL, предназначенных для кроссплатформенности.

Updated 8-Apr-2005 - Copyright REBOL Technologies - Formatted with MakeDoc2