Алексей Петровский

Командный язык программирования Tcl (Tool Command Language)

Москва

Литературное агентство «Бук-Пресс»
2006

УДК 681.3
ББК 32.973.26-018.2
П308

Петровский А.И.

П308 Командный язык программирования Tcl (Tool Command Language). - М.: Бук-пресс, 2006. - 192 с.

Tcl — язык программирования высокого уровня, который представляет собой текстовый язык с простым синтаксисом, в первую очередь предназначенный для подачи команд интерактивным приложениям, таким как текстовые редакторы, отладчики, иллюстрационные приложения и оболочки. Его легко изучать, а достигнув определенного уровня знакомства с языком, можно очень быстро создавать добротные приложения. На этом языке также можно программировать процедуры, тем самым, дополняя множество встроенных команд языка.

УДК 681.3
ББК 32.973.26-018.2

Ведение

Что такое Tcl/Tk?

Систему программирования Tcl/Tk разработал Джон Аустерхаут в то время, когда он работал в университете Калифорнии в Беркли. Она состоит из языка сценариев Tcl (Tool Command Language) и интерпретатора этого языка.

Язык программирования Tcl является основой системы и кроме собственно языка также включает в себя библиотеку. Язык Tcl представляет собой текстовый язык с простым синтаксисом, в первую очередь предназначенный для подачи команд интерактивным приложениям, таким как текстовые редакторы, отладчики, иллюстрационные приложения и оболочки. Его легко изучать, а достигнув определенного уровня знакомства с языком, можно очень быстро создавать добротные приложения. На этом языке также можно программировать процедуры, тем самым, дополняя множество встроенных команд языка.

Библиотечный пакет Tcl можно встраивать в прикладные программы. Библиотека Tcl состоит из анализатора языка Tcl, подпрограмм, реализующих встроенные команды, и процедур, позволяющих приложениям расширять Tcl дополнительными командами для работы этого приложения. Такое приложение генерирует команды Tcl и передает их анализатору Tcl для исполнения. Можно генерировать команды с помощью чтения данных из входного источника или при помощи привязки строк команд к элементам пользовательского интерфейса приложения, например, кнопкам, пунктам меню или комбинациям клавиш. Библиотека Tcl раскладывает полученные команды на составляющие поля и непосредственно исполняет встроенные команды. Для исполнения команд, реализуемых приложением,

Tcl делает вызов в приложение. Во многих случаях команды рекурсивно стартуют копии интерпретатора Tcl.

Все процедуры, команды циклов и условий работают таким образом.

Язык можно использовать для соединения воедино блоков, выполненных на языках системного программирования. В приложении эти блоки приобретают вид команд языка сценариев. Tcl можно легко встроить в существующую программу, за счет чего станет возможным управлять поведением этой программы и встраивать в нее другие блоки, например, графический интерфейс.

Прикладная программа получает три преимущества при использовании Tcl в качестве командного языка. Во-первых, Tcl предоставляет стандартный синтаксис и пользователи, знающие Tcl, смогут легко давать команды любому, основанному на Tcl, приложению. Во-вторых, на Tcl можно программировать само приложение: все, что требуется от приложения — это предоставить несколько своих специфических команд низкого уровня. Tcl предоставляет много команд-утилит и кроме этого, общий интерфейс программирования для создания сложных командных процедур. Используя все это, прикладные программы будут избавлены от необходимости самостоятельно воспроизводить такую же функциональность заново. В-третьих, Tcl можно использовать в качестве общего языка для общения приложений между собой. Коммуникации между приложениями не встроены в существующее ядро языка, но разнообразные дополнительные библиотеки, такие, как инструментальный набор Tk, позволяют приложениям подавать команды друг другу. Например, одно приложение может программно управлять работой другого. Все это позволяет программам работать совместно на гораздо более высоком качественном уровне, чем это было возможно ранее.

Tk дополняет Tcl средствами построения развитого графического интерфейса пользователя, состоящими из

примитивов (widgets). Примитивы Tk схожи с другими аналогичными наборами, а отличаются от других наборов тем, что для работы с примитивами Tk не нужно использовать C или C++.

Команды Tk создают примитивы и управляют ими, за счет чего программирование графического интерфейса сочетает в себе простоту и возможность тонко управлять деталями. В частности, очень мощные средства — текстовые примитивы и примитивы канвы (canvas). Текстовые примитивы при помощи механизма тегов поддерживают множественные шрифты: разные участки текста могут иметь разный вид и изменять его в зависимости от заданных событий. Примитивы канвы работают с элементами графических изображений, например: линиями, окружностями, дугами, прямоугольниками, овалами, изображениями, и также имеют теги.

Для выполнения Tcl-программы файл со скриптом следует передать интерпретатору. Также интерпретатор может работать в интерактивном режиме, когда пользователь с клавиатуры вводит команды скрипта в командной строке интерпретатора.

Применяются два стандартных интерпретатора языка: **wish** и **tclsh**. Интерпретатор **wish** обрабатывает команды и процедуры не только Tcl, но и Tk. Поэтому с его помощью можно работать с приложениями, имеющими графический интерфейс пользователя. Другой стандартный интерпретатор Tc — **tclsh** — может обрабатывать только команды Tcl и поэтому с его помощью нельзя создавать программы с графическим интерфейсом. Интерпретаторы Tcl написаны на C.

Существуют версии пакета Tcl для разных аппаратных платформ и операционных систем (ОС), и потому приложения на Tcl имеют хорошие предпосылки к достижению мобильности.

Однако, использование в приложении индивидуальных особенностей ОС с целью более полной реализации имеющихся

возможностей в итоге может ограничить мобильность приложения.

Обычно Tcl-программы выполняет интерпретатор, поэтому они работают не так быстро, как эквивалентные им программы на C. Для многих приложений это не критично, учитывая большую вычислительную мощность современных микропроцессоров. В тех случаях, когда важна скорость исполнения, можно воспользоваться компилятором Tcl или написать вычислительную часть программы на компилируемом языке, например, C или C++, а интерфейс пользователя написать на Tcl.

Возможно, этот язык не полностью подходит для создания больших сложных программ, от которых требуется исключительная надежность и устойчивость. Но несомненно, он будет очень удобен для создания программ, которым необходим хороший пользовательский интерфейс.

Язык и интерпретатор Tcl/Tk

Продукт Tcl/Tk в действительности представляет собой два связанных программных пакета, которые совместно обеспечивают возможность разработки и использования приложений с развитым графическим пользовательским интерфейсом. Название Tcl относится к «командному языку инструментальных средств — tool command language», и, как не странно, его рекомендуется произносить «тикл». Это простой командный язык для управления приложениями и расширения их возможностей. Язык Tcl является «встраиваемым»: его интерпретатор реализован в виде библиотеки функций языка Си, так что интерпретатор может быть легко пристыкован к любой прикладной программе, написанной на языке Си.

Tk (рекомендуемое произношение — «ти-кей») является библиотекой Си-функций, ориентированной на облегчение создания пользовательских графических интерфейсов в среде оконной системы X (т.е., по сути дела, некоторый аналог Xt

Intrinsics). С другой стороны, аналогично тому, как это делается в командных языках семейства shell, функции библиотеки Tk являются командами языка Tcl, так что любой программист может расширить командный репертуар языка Tcl путем написания новой функции на языке Си.

Совместно, Tcl и Tk обеспечивают четыре преимущества для разработчиков приложений и пользователей. Во-первых, наличие командного языка Tcl дает возможность в каждом приложении использовать мощный командный язык. Все, что требуется от разработчика приложения, чтобы удовлетворить его специфические потребности, — это создать несколько новых команд Tcl, требующихся приложению (и, возможно, другим приложениям — явно традиционный стиль командного программирования в ОС UNIX). После этого нужно связать прикладную программу с интерпретатором Tcl и пользоваться полными возможностями командного языка.

Вторым преимуществом использования Tcl/Tk является возможность быстрой разработки графических интерфейсов. Многие интересные оконные приложения могут быть написаны в виде скриптов языка Tcl без привлечения языков Си или Си++ (а Tcl позволяет скрыть многие несущественные детали). Как утверждают разработчики Tcl/Tk, пользователи оказываются способными к созданию новых графических интерфейсов уже после нескольких часов знакомства с продуктом. Другой особенностью языка Tcl, способствующей быстрой разработке оконных приложений, является то, что язык является интерпретируемым. Можно опробовать новую идею интерфейса, выражающуюся в сотнях или тысячах строк кода на языке Tcl, без потребности вызова новых программных средств, путем простого нажатия на клавишу мыши (не наблюдая существенных задержек при использовании современных рабочих станций).

Третьим преимуществом языка Tcl является то, что его можно применять в качестве языка «склейки» приложений.

Например, любое основанное на Tcl и использующее Tk оконное приложение может направить свой скрипт любому другому аналогично ориентированному приложению. С использованием Tcl/Tk можно создавать приложения, работающие в стиле мультимедиа, и опять же они смогут обмениваться скриптами, поскольку пользуются общим интерпретатором командного языка Tcl и общей внешней библиотекой Tk.

Наконец, четвертым удобством интегрированного пакета Tcl/Tk является удобство пользователей. Для написания нового приложения в среде Tcl/Tk достаточно выучить несколько совершенно необходимых команд, и этого окажется достаточно. Другими словами оказывается возможным инкрементальный (пошаговый) стиль погружения в предмет. Такая возможность всегда радует сердце и греет душу.

Впрочем, заметим, что далеко не все программисты разделяют выраженное выше глубоко радостное отношение разработчиков Tcl/Tk к своему продукту.

Язык системной интеграции

Языки программирования принято делить на 6 поколений. Однако в эту классификацию не укладывается большая группа так называемых скрипт-языков (СЯ). Программы на СЯ обычно вставляются в исходных текстах в приложения для их гибкой настройки, или, наоборот, служат мощным средством для объединения ПО, написанного на других языках. Но большинство разработчиков считают СЯ только вспомогательным инструментом. Например, JavaScript обычно воспринимается как урезанная Java и используется в основном для создания бегущих строчек в браузере. Программисты иногда даже возмущаются отсутствием в JavaScript типизации, что говорит о полном непонимании предназначения этого языка. Ведь JavaScript обладает уникальными возможностями по интеграции объектов Windows — например, автоматическим

управлением Internet Explorer, который является OLE-объектом. При работе с ним использовать переменные конкретных типов неудобно или невозможно.

Джон Аустираут, создатель скрипт-языка Tcl/Tk, назвал СЯ языками системной интеграции, потому что они ориентированы прежде всего на работу не с элементарными данными, а с объектами операционной среды, что позволяет очень эффективно использовать возможности ОС, интегрируя ее ресурсы с помощью языков, которые на высоком уровне, без написания тысяч строк кода, двумя-тремя командами позволяют легко манипулировать системными объектами и «склеивать» их в одно целое. По мнению Аустираута, принципы ООП не завоевали должной популярности только потому, что их пытаются реализовывать на уровне примитивных операторов языков третьего поколения (3GL) типа Си++, что приводит к громоздкому и неуклюжему коду, а не на уровне свойств и методов самих объектов. В отличие от 3GL-языков, в современных СЯ парадигма ООП воплощена наиболее полно. Она присутствовала даже в старых СЯ типа командных языков MS DOS или UNIX, которые позволяют довольно эффективно работать с объектами ОС — файлами и программами.

Практически все СЯ — интерпретируемого типа (в последнее время для ускорения интерпретации все более активно используется компиляция в промежуточный код). Это связано с тем, что для работы с объектами ОС язык должен обладать высокой гибкостью, а процесс выполнения системных программ часто зависит от многих факторов, которые трудно предусмотреть на этапе компиляции. Для повышения гибкости в СЯ обычно отсутствует строгая типизация. Основной элемент данных в этих языках — строка, которую при работе с двоичными объектами нередко надо интерпретировать как последовательность байт. Кроме того, для настройки крупной системы постоянно возникает необходимость быстрого создания или модификации небольших программ, для чего удобнее использовать интерпретаторы. Конечно, СЯ проигрывают

компилируемым 3GL-языкам в производительности (оператор Си транслируется в среднем в 3-7 машинных команд, а для выполнения одного оператора СЯ требуются сотни тысяч команд), но позволяют разрабатывать ПО в десятки раз быстрее. К тому же они более легки в изучении, что очень важно при подготовке персонала для сопровождения ОС, сетей и т. д.

История создания

Tcl представляет собой язык, синтаксически немного похожий на Perl. Он ориентирован преимущественно на автоматизацию рутинных процессов ОС и крупных программных систем и состоит из мощных команд, ориентированных на работу с абстрактными нетипизированными объектами. Принципиальное отличие Tcl от командных языков ОС в независимости от типа системы (когда не надо утруждать себя изучением нового командного языка) и, самое главное, он позволяет создавать переносимые программы с графическим интерфейсом (GUI). Можно, например, написать утилиту просмотра всех каталогов и подкаталогов с уничтожением временных файлов, приделать к ней симпатичную оболочку с кнопочками, меню, картинками, диалогом ввода расширения уничтожаемых файлов, и эта утилита будет работать практически в любой ОС!

Tcl — расширяемый язык. Можно самостоятельно определять новые команды языка (как в Форте). Классический пример определения команды Tcl:

```
button .b -text Hello! -font {Times 16} -command {puts hello}
```

Команда **button** предназначена для создания кнопки (**.b**) с надписью «Hello!» (можно указать картинку), которая пишется шрифтом **Times** размером **16** пунктов. При нажатии на кнопку в стандартное устройство вывода посылается строка **hello** (**puts hello**). Набором таких команд можно очень компактно и наглядно описать GUI с полным функциональным

наполнением. Эти команды можно вызывать из других языков или на их основе писать ПО полностью на Tcl. В последнем случае приложение считается «Pure Tcl 100%».

Tcl/Tk разрабатывался одновременно как язык и библиотека. Эта идея была позже реализована в Java, языке фирмы Sun, в которой Аустирает трудится с середины 90-х годов. Библиотека Tk содержит стандартизованный набор команд поддержки GUI в стиле Motif. Управляющие элементы, хранящиеся в Tk, называются виджетами (widgets). Большое количество нетиповых виджетов можно найти в Сети. На основе Tk создаются переносимые приложения, например, на Си, с графическим интерфейсом.

На Tcl написано очень много мобильных приложений с GUI, которые используются в самых разных областях — для управления ОС, администрирования сетей, обслуживания Web-серверов и т.д. В Tk имеются команды для работы с сокетами, с помощью которых можно создавать приложения для Сети. Для браузеров разработаны подключаемые модули, позволяющие вставлять Tcl-скрипты (тиклеты) в HTML-страницы. Известно использование Tcl/Tk в масштабных российских проектах, в частности, в кодогенераторах фирмы DataX/Florin для поддержки платформо-независимого пользовательского интерфейса.

Tcl/Tk распространяется в исходных текстах бесплатно. Он может легко интегрироваться с Си++, Адой, Прологом, Модулой-3, Perl и многими другими языками. Tcl реализован для Windows 3.1/95/98/NT, Macintosh, AIX 3.x, VMS, для практически всех UNIX-платформ — SunOS, SCO Unix, HP-UX, BSDI, Linux, QNX, а также для суперкомпьютеров Cray и NEC. На Pure Tcl 100% написана оболочка Visual Tcl, которая позволяет разрабатывать кросс-платформенное ПО для UNIX, Windows и Macintosh. Версию Tcl/Tk для OS/2 Presentation Manager написал некий Илья Захаревич из штата Огайо (не исключено, наш соотечественник, так как в этом штате работает

много российских программистов). Имеется расширение языка incrTcl/incrTk, ориентированное на создание больших приложений.

С начала 90-х годов проект Tcl взялась поддерживать фирма Sun. Была создана рабочая группа SunScript (название языка осталось прежним). В середине 90-х годов в этой группе работает Джон Аустирает. SunScript создает коммерческие версии Tk, в частности, для Windows и Macintosh. Несколько месяцев назад эта группа приступила к интеграции Tcl и Java. Разработана версия Tcl, написанная на Java — Jacl (JAva Command Language). Расширение языка Tcl Blend позволяет работать с Java VM и интегрировать Tcl-скрипты с Java-кодом.

В восьмой версии Tcl/Tk интерпретация заменена компиляцией в байт-код «на лету» (по аналогии с Java). Полностью переделана обработка строк. Работа с двоичными объектами и большими текстовыми массивами по сравнению с версией 7.x ускорилась в 2–20 раз. Добавлены новые команды быстрой сортировки, ввода/вывода двоичных данных с поддержкой различных типов периферийных устройств, поддерживается протокол http, реализованы новые средства обеспечения безопасности. Увеличена точность при работе с числами с плавающей запятой до 12 цифр вместо 6. Подробно документирован интерфейс с Си. Полностью переписана Tk. Работа с объектами управления стала более удобной, усовершенствована работа с меню и шрифтами, добавлены новые команды работы с графическими файлами и др. Наконец, приложения на Tcl/Tk могут сопровождаться логотипом «Tcl Powered» (по лицензии Sun). Появилась, правда, небольшая несовместимость с программами, написанными на старых версиях Tcl. Теперь даты, выраженные двумя цифрами года, трактуются как даты следующего столетия. Например, 17 год означает не 1917, а 2017.

Учитывая, что Tcl/Tk активно поддерживается Sun, можно предсказать этому языку очень хорошее будущее. А история

создания Tcl/Tk является классическим примером на тему «как стать знаменитым». Хотите им стать? Разработайте свой скрипт-язык для использования в какой-нибудь быстро развивающейся области ИТ, положите исходные тексты интерпретатора на свою страничку, напишите в конференцию по программированию, и ждите, когда вас пригласят в Sun или Microsoft...

Основные понятия и элементы

Общая характеристика языка Tcl/Tk

Язык Tcl/Tk обладает многими свойствами обычных процедурных языков и имеет следующие основные особенности:

- это язык высокого уровня, что выражается в значительно меньшем количестве непосредственного программирования для решения задачи;
- это интерпретируемый язык: программы на Tcl готовы к выполнению без компилирования и компоновки, хотя существует и компилятор Tcl;
- это расширяемый язык: в него легко добавляются команды пользователя, написанные на C или Tcl;
- его можно встраивать в создаваемые приложения — интерпретатор Tcl представляет собой просто набор функций C, которые можно вызывать из кода приложения. В результате Tcl может служить языком приложения, подобно макроязыку электронных таблиц;
- Tcl работает на многих распространенных платформах;
- Tcl предоставляет возможность автоматической загрузки пользовательских библиотек Tcl. Также возможна автоматическая загрузка по мере надобности динамических библиотек DLL, если ОС поддерживает такую загрузку.

Типы данных Tcl

Tcl поддерживает только один тип данных: строки. Все команды, все аргументы команд, результаты их выполнения и значения переменных суть строки. В тех случаях, когда командам требуются числовые аргументы или они возвращают числовые результаты, аргументы и результаты передаются в виде строк. Многие команды предполагают соответствие их аргументов определенным форматам, но это суть индивидуальные особенности команд. Например, часто аргументы содержат командные строки, которые могут быть обработаны в качестве частей команды. Самый простой путь для понимания интерпретатора Tcl — это помнить, что все в нем представляет собой операции над строками. Во многих случаях конструкции Tcl будут выглядеть подобно более структурированным конструкциям из других языков. Однако конструкции Tcl не являются структурированными вообще: они являются лишь строками символов, и поэтому они ведут себя иначе, чем те конструкции, на которые они могут быть похожи.

Способ интерпретации строки Tcl зависит от конкретного интерпретатора, однако существует три общих формы строк: команды, выражения и списки.

Основы синтаксиса команд

Синтаксически язык Tcl похож одновременно на оболочки Unix и lisp. Тем не менее, интерпретация команд в Tcl отличается от обоих систем. Команда Tcl состоит из одной или нескольких команд, разделенных символами новой строки или точками с запятой. Каждая команда состоит из набора полей, разделенных пустым промежутком (пробелами или табуляцией). Первое поле должно быть именем команды, а необязательные остальные поля — суть аргументы, передаваемые этой команде. Например, команда:

```
set a 22
```


имеет три поля: первое, **set**, есть имя команды, а остальные два, **a** и **22**, будут переданы в качестве аргументов команде **set**. Имя команды должно быть именем встроенной команды Tcl, дополнительной команды, созданной для данного приложения процедурой **Tcl_CreateCommand**, или командной процедурой, определенной с помощью встроенной команды **proc**.

Аргументы передаются текстовыми строками в буквальном смысле. Команды пользуются этими строками так, как им требуется. Например, команда **set** считает свой первый аргумент именем переменной, а второй — строковым значением для присвоения этой переменной. Другие команды могут считать свои аргументы целыми числами, списками, именами файлов или командами Tcl.

Обычно имена команд должны быть напечатаны полностью, без сокращений. В тех случаях, когда интерпретатор Tcl не может узнать имя команды, он вызовет специальную команду по имени **unknown**, которая попытается найти или создать указанную команду. Например, во многих случаях команда **unknown** будет искать команду в каталогах библиотек и если найдет, то создаст ее в виде процедуры Tcl. Часто команда **unknown** обеспечивает выполнение команд, заданных сокращенным именем, но обычно только тех, которые поданы в интерактивном режиме работы. Даже если выполнение такой команды обеспечивается, использование сокращенной формы имен команд в командных скриптах и других вещах, которые будут в использовании спустя время, не рекомендуется: изменения в наборах команд последующих версий языка могут привести к неоднозначной интерпретации и вызванными этим ошибками в работе скриптов.

Комментарии

При изображении первого символа команды, кроме пробела, в виде #, все последующие символы в этой строке до символа новой строки включительно считаются комментарием и игнорируются. Когда комментарии встроены во вложенных

командах (например, поля, заключенные в фигурные скобки), они должны иметь парные фигурные скобки. Это необходимо потому, что когда Tcl анализирует команду верхнего уровня, он еще не знает, что вложенное поле будет использовано как команда, и поэтому не может обработать вложенный символ комментария как символ комментария.

Группирование аргументов с помощью двойных кавычек

Обычно каждое поле аргумента заканчивается последующим пробелом, однако двойные кавычки можно использовать для задания аргументов с пробелами внутри. Если поле аргумента начинается с двойных кавычек, то этот аргумент не будет заканчиваться пробелом (включая символы новой строки) или точкой с запятой (использование точек с запятой описано ниже). Такой аргумент будет заканчиваться только следующим символом двойных кавычек. Символы двойных кавычек не будут входить в значение аргумента. Например, команда

```
set a "This is a single argument"
```

передаст команде **set** два аргумента: **a** и **This is a single argument**. Выполнение подстановок команд, подстановок переменных и подстановок с обратным слешем внутри двойных кавычек будет сохранено. Если первый символ в поле команды не двойные кавычки, тогда при анализе этого поля двойные кавычки не имеют специальной интерпретации.

Группирование аргументов с помощью фигурных скобок

Для группирования аргументов можно также использовать фигурные скобки. Их действие похоже на двойные кавычки, за исключением двух отличий. Во-первых, они позволяют вложение, поэтому их проще использовать для таких усложненных аргументов, как вложенные командные строки Tcl. Во-вторых, нижеописанные подстановки для команд,

переменных и обратных слешей внутри фигурных скобок не выполняются. Поэтому фигурные скобки можно использовать для того, чтобы избежать ненужных подстановок. Если поле аргумента начинается с открывающей фигурной скобки, то этот аргумент заканчивается соответствующей ей закрывающей фигурной скобкой. Tcl отбросит наружную пару фигурных скобок и передаст команде информацию внутри скобок без какой-либо последующей модификации. Например, в команде

```
set a {xyz a {b c d}}
```

команда **set** получит два аргумента: **a** и **xyz a {b c d}**.

Когда действуют двойные кавычки или фигурные скобки, парная закрывающая скобка или кавычки не должны обязательно быть на той же строке, что и ее открывающая. В этом случае символ новой строки будет включен в поле аргумента вместе со всеми остальными символами до закрывающей скобки или кавычек. Например, команда **eval** использует один аргумент, представляющий собой строку команды: **eval** вызывает интерпретатор Tcl для выполнения командной строки. Команда

```
eval {
set a 22
set b 33
}
```

присвоит значение **22** переменной **a** и значение **33** — переменной **b**.

Если первый символ поля команды — не открывающая фигурная скобка, то ни открывающая, ни закрывающая фигурные скобки в этом поле не будут интерпретироваться специальным образом.

Подстановка команд со скобками

При появлении в поле команды открывающей квадратной скобки выполняется подстановка команды. Все символы внутри

скобок считаются командой, и она исполняется немедленно. Затем результат этого исполнения подставляется вместо текста в скобках. Например, рассмотрим команду

```
set a [set b]
```

Когда у команды **set** задан только один аргумент, тогда это есть имя переменной, и **set** возвращает значение этой переменной. В этом случае, если переменная **b** имеет значение **foo**, то предыдущая команда эквивалентна команде

```
set a foo
```

Квадратные скобки можно использовать более сложными способами. Например, если переменная **b** имеет значение **foo**, а переменная **c** имеет значение **gorp**, то команда

```
set a xyz[set b].[set c]
```

эквивалентна команде

```
set a xyzfoo.gorp
```

Команда в квадратных скобках может содержать несколько команд, разделенных обычным образом — строками или точками с запятой. В этом случае для подстановки используется значение последней команды. Например, последовательность команд

```
set a x[set b 22
expr $b+2]x
```

эквивалентна команде

```
set a x24x
```

Если поле заключено в фигурные скобки, то квадратные скобки и символы между ними не интерпретируются специальным образом, а передаются в аргумент без изменения.

Подстановка переменных с $

Символ доллара **$** можно использовать в качестве краткой формы для подстановки переменных. Если у аргумента, не заключенного в фигурные скобки, имеется символ **$**, то

выполняется подстановка переменной. Символы после **$** вплоть до первого символа, не являющегося цифрой, буквой или подчеркиванием, считаются именем переменной, и строковое значение этой переменной подставляется вместо ее имени. Например, если переменная **foo** имеет значение **test**, то команда:

```
set a $foo.c
```

эквивалентна команде

```
set a test.c
```

Существует две специальные формы для подстановки переменных. Если следующим после имени переменной является открывающая скобка, то переменная считается именем массива, и все символы между открывающей скобкой и следующей закрывающей скобкой считаются индексом внутри этого массива. Команды и переменные, используемые в качестве индекса, обрабатывается перед операцией извлечения элемента массива. Например, если переменная **x** есть массив и один его элемент по имени **first** имеет значение **87**, а второй по имени **14** — значение **more**, то команда:

```
set a xyz$x(first)zyx
```

эквивалентна команде

```
set a xyz87zyx
```

Если переменная **index** имеет значение **14**, то команда:

```
set a xyz$x($index)zyx
```

эквивалентна команде

```
set a xyzmorezyx
```

Массивы

Вторая специальная форма для переменных имеет место тогда, когда после символа доллара следует открывающая фигурная скобка. В этом случае имя переменной состоит из всех символов, заключенных между фигурными скобками. Ссылки на массив в таком случае невозможны: предполагается, что такое

имя есть имя скалярной переменной. Например, если переменная **foo** имеет значение **test**, то команда:

```
set abc${foo}bar
```

эквивалентна команде

```
set abctestbar
```

В аргументе, заключенном в фигурные скобки, не выполняется подстановка переменных: символ доллара и имя переменной передаются аргументу без изменения.

Аббревиатура с символом доллара есть просто сокращенная форма: **$a** полностью эквивалентна **[set a]** и используется только для удобного сокращения количества печатаемых символов.

Разделение команд точкой с запятой

Обычно каждая команда занимает одну строчку (команда заканчивается символом новой строки). Символ точки с запятой также считается разделителем команд: можно поместить несколько команд в одной строке, разделив их точками с запятой. Точки с запятой не считаются разделителями команд, если они находятся внутри фигурных скобок или двойных кавычек.

Подстановки с обратным слешем

Обратный слеш можно использовать для ввода непечатаемых символов в поля команд и для вставки специальных символов (например, фигурных скобок или двойных кавычек) в поля без описанной выше специальной интерпретации этих символов. Ниже перечислены различаемые интерпретатором Tcl последовательности с обратным слешем. В каждом случае последовательность с обратным слешем замещается указанным символом:

- \b — сдвиг на одну позицию влево (0x8);
- \f — прогон листа (0xc);

- \n — новая строка, newline (0xa);
- \r — возврат каретки (0xd);
- \t — табуляция (0x9);
- \v — вертикальная табуляция (0xb);
- \{ — открывающая фигурная скобка;
- \} — закрывающая фигурная скобка;
- \[— открывающая квадратная скобка;
- \] — закрывающая квадратная скобка;
- \$ — символ доллара;
- \sp — пробел, аргумент не прерывается;
- \; — точка с запятой, аргумент не прерывается;
- \" — двойные кавычки;
- \nl — ничего: обратный слеш с последующей новой строкой соединяет две строки в одну. Эта подстановка уникальна тем, что она действует даже внутри фигурных скобок;
- \<newline>пробел — одиночный пробел заменяет обратный слеш, символ новой строки и все пробелы и табуляции после этого символа новой строки. Данная последовательность уникальна в том смысле, что она замещается в отдельном суб-проходе перед тем, как начинается непосредственно анализ команды. Поэтому эта последовательность будет замещена, даже если она стоит между фигурными скобками и результирующий пробел будет интерпретироваться как разделитель слов, кроме того случая, когда он стоит между фигурными скобками или кавычками;
- \\ — обратный слеш (\);

- \ddd — цифры ddd (одна, две или все три) дают восьмеричное значение символа. Нулевой символ нельзя вставить в поле команды: если ddd есть ноль, то последовательность с обратным слешем игнорируется (т.е., преобразовывается в нулевую строку).

  Пример:

```
set a \{x\[\\0yz\141
```

второй аргумент команды **set** будет **{x[\0yza**.

Если следом за обратным слешем стоит какой-либо иной символ, кроме перечисленных выше, то обратный слеш передается в поле аргумента без специальной обработки, и сканер строки Tcl продолжает нормальную работу со следующим символом.

Например, в команде:

```
set \*a \\\{foo
```

первый аргумент для **set** будет ***a**, а второй — **\{foo**.

Если аргумент заключен в фигурные скобки, то последовательности с обратным слешем внутри аргумента будут распознаны, но подстановки не будут выполнены (за исключением пары обратный слеш — новая строка): последовательности с обратным слешем передаются в аргумент как есть, без какой-либо специальной интерпретации символов в последовательностях. В частности, в такой ситуации фигурные скобки не означают поиска соответствующей парной скобки, заканчивающей аргумент. Например, в команде:

```
set a {\{abc}
```

второй аргумент команды **set** будет **\{abc**.

Данный механизм обратного слеша недостаточен для создания аргумента абсолютно любой структуры; он только обеспечивает наиболее общие случаи. Для создания особенно сложного аргумента, возможно, самым простым вариантом будет

использование команду **format** совместно с подстановкой команды.

Выражения

Вторая смысловая форма строк в Tcl — это выражения. Несколько команд, таких как **expr**, **for** и **if**, считают один или несколько своих аргументов выражениями и для вычисления их значения вызывают процессоры выражений Tcl (**Tcl_ExprLong**, **Tcl_ExprBoolean** и т.д.). Разрешенные в Tcl-выражениях операторы составляют подмножество операторов, разрешенных в выражениях C, и они имеют такой же смысл и приоритет выполнения, как и соответствующие им операторы C. Почти всегда значением выражения является число (целое или с плавающей запятой). Например, выражение:

```
8.2 + 6
```

дает результат **14,2**. Выражения Tcl отличаются от выражений C способом описания операндов, а также тем, что поддерживают нечисловые операнды и сравнение строк.

Выражение Tcl состоит из комбинации операндов, операторов и скобок. Между ними всеми можно ставить пробелы, потому что при вычислении значения пробелы игнорируются. По возможности, все операнды интерпретируются как целые числа, если не задано иное. Целые числа могут иметь вид десятичного числа (обычно), восьмеричного (если первая цифра числа есть 0) или шестнадцатеричного (если первые два символа числа — 0x). Если операнд не подпадает ни под один из названных форматов, он считается числом с плавающей запятой, если это возможно. Числа с плавающей запятой можно задавать любым из способов, воспринимаемым совместимым с ANSI компилятором C. Исключение составляет запрет в большинстве версий на суффиксы f, F, l, и L. Примеры правильных чисел с плавающей запятой: 2.1, 3., 6e4, 7.91e+16. Если числовая

интерпретация невозможна, то операнд считается строковым и работать с ним может только ограниченный набор операторов.

Операнды могут быть заданы одним из следующих способов:

- числовым значением — целым или с плавающей запятой;
- переменной Tcl, используя обычную нотацию со знаком $. В качестве операнда будет использовано значение переменной;
- строкой символов, заключенной в двойные кавычки. Анализатор выражения выполнит подстановки переменных, команд и с обратным слешем; полученное значение будет использовано в качестве операнда;
- строкой, заключенной в фигурные скобки. Все символы между открывающей и закрывающей скобками будут считаться операндом без каких-либо подстановок;
- командой Tcl, заключенной в угловые скобки. Команда будет выполнена, и ее результат будет использован в качестве операнда.

Если в выражении имели место подстановки (например, внутри двойных кавычек), то они будут обработаны процессором выражения. Хотя анализатор команд тоже может выполнить свою часть подстановок (дополнительную серию подстановок) до вызова процессора выражения. Поэтому обычно, во избежание выполнения анализатором команд подстановок в содержимое выражения, лучше всего заключать выражение в фигурные скобки.

В качестве примеров рассмотрим простые выражения, в которых переменная **a** имеет значение **3**, а значение переменной **b** есть **6**. Тогда выражение в левой части каждой строки даст значение в ее правой части:

```
3.1 + $a  6.1
2 + " $a.$b" 5.6
```



```
4*[llength " 6 2" ] 8
{word one} < " word $a" 0
```

Списки

Третьей основной смысловой формой строк в Tcl являются списки. Список — это обычная строка с подобной списку структурой, состоящей из полей, разделенных промежутками. Например, строка **Al Sue Anne John** есть список, имеющий четыре элемента (поля). Основная структура списков аналогична структуре командных строк, за исключением того, что символ новой строки служит таким же разделителем, как и пробел с табуляцией. Для списков действуют такие же правила в отношении фигурных скобок, двойных кавычек и обратных слешей, как и для команд. Например, строка:

```
a b\ c {d e {f g h}}
```

есть список из трех элементов: **a**, **b c** и **d e {f g h}**. Всегда, когда из списка извлекается элемент, действуют те же правила относительно фигурных скобок, двойных кавычек и обратных слешей, что и для команд. Таким образом, когда из списка в примере будет извлечен третий элемент, результат будет **d e {f g h}** (потому что при извлечении произошло только отбрасывание внешней пары фигурных скобок). В отношении списков никогда не выполняются подстановки команд и переменных (по крайней мере, командами обработки списков: список всегда может быть передан интерпретатору Tcl для обработки).

Команды Tcl **concat**, **foreach**, **lappend**, **lindex**, **linsert**, **list**, **llength**, **lrange**, **lreplace**, **lsearch** и **lsort** позволяют составлять списки, извлекать из них элементы, просматривать содержимое и выполнять прочие относящиеся к спискам функции.

Регулярные выражения

Tcl предоставляет две команды для сравнения строк регулярных выражений в стиле **egrep**: **regexp** и **regsub**.

Регулярное выражение состоит из ни одной или более **ветвей** (branch), разделенных символом «|». Оно совпадает с любым выражением, которое совпадает с одной из ветвей.

Ветвь состоит из одного или более **кусков** (piece), соединенных друг с другом. Она совпадает с выражением, которое состоит из тождества для первого куска, следом за которым идет тождество для второго куска, и т.д.

Кусок состоит из атома со следующим за ним необязательным символом *****, **+** или **?**. Атом с последующим символом ***** совпадает с последовательностью из одного или более тождества для этого атома. Атом с символом **+** после него совпадает с последовательностью из одного или более тождества для этого атома. Атом с символом **?** после него совпадает с последовательностью из одного тождества для этого атома или пустой строкой.

Атом может быть регулярным выражением в скобках (в этом случае оно совпадает с тождеством для этого регулярного выражения), интервалом, символом «**.**» (совпадающим с одним любым символом), «^» (совпадающим с нулевой строкой в начале вводимой строки), «**$**» (совпадающим с нулевой строкой в конце вводимой строки), «\» с последующим одним символом (совпадающим с этим символом), или одним символом без какого-либо иного смысла (совпадающим с этим символом).

Интервал есть последовательность символов, заключенная в квадратные скобки. Он обычно совпадает с любым символом из этого интервала. Если последовательность начинается с символа ^, то она совпадает с любым символом не из числа остальных символов. Если два символа в последовательности разделены символом **-**, то это краткая форма для обозначения всех символов между этими двумя (например, [0-9] совпадает с

любой десятичной цифрой). Для того, чтобы включить в последовательность символ «**]**», следует поставить его на место первого в последовательности (следом за возможным символом «^»). Для включения в последовательность символа «**-**» следует сделать его первым или последним символом.

Если регулярное выражение совпадает с двумя разными частями строки, она будет совпадать с той, которая раньше начинается. Если обе начинаются в одном и том же месте, то это неопределенный (тяжелый) случай. Его можно объяснить следующим образом.

В общем, возможные исходы в списке ветвей рассматриваются слева направо. Исходы для *****, + и **?** рассматриваются в порядке убывания длины. Вложенные конструкции рассматриваются извне вовнутрь (from the outermost in), и составленные (concatenated) конструкции рассматриваются слева направо. Будет выбрано то из тождеств, которое будет отвечать самому раннему из исходов в первом выборе. Если предстоит несколько выборов, то следующее сравнение будет сделано таким же образом (самый ранний из исходов), в зависимости от решения при первом выборе, и так далее.

Например, **(ab|f)b*c** может совпадать с **abc** одним из двух способов. Первый выбор делается между **ab** и **a**; поскольку **ab** стоит раньше и ведет к успешному совпадению, то оно будет выбрано. Поскольку **b** уже закрыто, то **b*** должно совпасть со своим крайним исходом — пустой строкой, чтобы не конфликтовать с предыдущим выбором.

В частном случае, когда нет ни одного разделителя ветвей «|» и присутствует только один символ *****, **+** или **?**, общий результат будет следующим: будет выбрано самое длинное тождество из всех возможных. Таким образом, сравнение **ab*** с **xabbbby** даст результат **abbbb**. Зато если **ab*** будет сравниваться с **xabyabbbz**, то результатом будет считаться **ab**, находящееся сразу за **x**, в соответствии с правилом первого совпадения из всех

возможных. В действительности, решение о том, откуда начинать сравнивать, есть первый выбор, который надо сделать. Этому выбору должны подчиняться последующие шаги, даже они ведут к менее предпочтительному результату.

Результаты команд

Каждая команда возвращает два результата: код и строку. Код служит для индикации того, успешно или нет закончилась команда, а строка предоставляет дополнительную информацию. Действующие значения кодов определены в файле tcl.h, в виде следующего списка:

TCL_OK

Этот код нормального завершения, возвращается при успешном выполнении команды. Строка содержит возвращаемое командой значение.

TCL_ERROR

Указывает на имевшую место ошибку; строка содержит сообщение, описывающее ошибку. Дополнительно к этому, глобальная переменная **errorInfo** будет содержать словесную информацию о командах и процедурах, выполнявшихся при возникновении ошибки; глобальная переменная **errorCode** будет содержать машинно-читаемые данные об ошибке, если таковые доступны.

TCL_RETURN

Указывает на то, что была вызвана команда **return**, и что текущая процедура (или команда верхнего уровня, или команда source) должна немедленно завершиться. Строка содержит возвращаемое значение для процедуры или команды.

TCL_BREAK

Указывает на то, что была вызвана команда **break**, и поэтому самый внутренний цикл должен немедленно прекратиться. Строка должна всегда оставаться пустой.

TCL_CONTINUE

Указывает на то, что была вызвана команда **continue**, и поэтому самый внутренний цикл должен приступить к следующей итерации. Строка должна всегда оставаться пустой.

Обычно программистам на Tcl не нужно задумываться о кодах возврата, поскольку почти всегда возвращается **TCL_OK**. Если команда возвратила что-либо иное, то интерпретатор Tcl немедленно останавливает обработку команд и возвращается к вызвавшему его событию. Если в некоторый момент имеется несколько вложенных вызовов интерпретатора Tcl, то обычно каждая из вложенных команд вернет ошибку вызывающему ее субъекту, и таким образом сообщение об ошибке достигнет самого верхнего уровня в приложении. После этого приложение выведет пользователю сообщение об ошибке.

В некоторых случаях отдельные команды обрабатывают ошибочные ситуации сами и не сообщают о них наверх. Например, команда **for** проверяет наличие кода возврата **TCL_BREAK** и если находит его, то прекращает выполнение тела цикла и возвращает код **TCL_OK** вызвавшему субъекту. Также команда **for** обрабатывает коды **TCL_CONTINUE**, а Интерпретатор процедур обрабатывает коды **TCL_RETURN**. Команда **catch** позволяет программам Tcl перехватывать ошибки и обрабатывать их без последующего прекращения интерпретации команд.

Процедуры

Tcl позволяет расширить командный интерфейс за счет определения процедур. Процедуру Tcl можно вызвать для исполнения так же, как и любую другую команду Tcl (у нее есть

имя и она получает один или более аргументов). Единственным отличием является то, что она не состоит из кода C, скомпонованного внутрь программы: это строка, содержащая одну или более команд Tcl.

Переменные: скалярные и массивы

В Tcl можно определять переменные и использовать их значения при помощи подстановки переменных с символом **$**, команды **set**, или нескольких иных механизмов. Нет необходимости специально определять переменные: новая переменная будет автоматически определена сразу же, как только будет использовано новое имя переменной.

Tcl поддерживает переменные двух типов: скалярные и массивы (векторные). Скалярная переменная имеет только одно значение в каждый момент времени, тогда как переменная-массив может содержать любое количество элементов, имеющих имя (обычно называемое «индексом») и значение. Индексами массива могут быть произвольные строки, необязательно числового вида. Для ссылки на индексы в командах Tcl используются круглые скобки. Например, команда **set x(first) 44** изменит значение элемента массива **x** по имени **first** на новое: **44**. Двумерные массивы можно имитировать использованием индексов, состоящих из нескольких составленных вместе частей. Например, команды

```
set a(2,3) 1
set a(3,6) 2
```

задают элементы массива **a** с индексами **2,3** и **3,6**.

В общем, элементы массивов можно употреблять везде, где можно употреблять скалярные переменные. Недопустимо наличие скалярной переменной и массива с одним и тем же именем. Нельзя обращаться к скалярной переменной как к элементу массива. Для преобразования скалярной переменной в массив и наоборот следует удалить существующую переменную

при помощи команды **unset**. Команда **array** предоставляет набор средств для работы с массивами, в том числе получение списка всех элементов массива и просмотр значений элементов по одному.

Переменные могут быть локальными или глобальными. Если имя переменной используется тогда, когда не выполняется процедура, то оно автоматически относится к глобальной переменной. Имена переменных, используемых внутри процедуры, обычно ссылаются на локальные переменные, ассоциированные с данным выполнением этой процедуры. Локальные переменные удаляются по окончании работы процедуры. При выполнении процедуры для указания на то, что имя является именем глобальной переменной, может использоваться команда **global** (она в некотором смысле аналогична extern в C).

Операторы

Действующие операторы перечислены ниже в порядке убывания приоритетности исполнения.

-
: Унарный минус

+
: Унарный плюс

~
: Побитовое неравенство **NOT**

!
: Логическое **NOT**

Ни один из этих операторов не может быть использован со строковыми операндами, а побитовое **NOT** может использоваться только с целыми числами.

*****

Умножить

/

Разделить

%

Остаток деления

Ни один из этих операторов не может быть использован со строковыми операндами, а оператор остатка может использоваться только для целых чисел.

+ –

Сложение и вычитание

Могут использоваться для любых числовых операндов.

<< >>

Сдвиг влево и вправо. Операторы можно использовать только с целыми числами. Сдвиг вправо также сдвигает и знаковый бит.

< > <= >=

Операторы булевой алгебры: меньше, больше, не больше, не меньше. Каждый оператор дает результат **1**, если неравенство верно, и **0** — в обратном случае. Кроме числовых операндов, операторы можно применять для строковых выражений, в этом случае выполняется сравнение строк.

== !=

Булевские операторы: равно и не равно. Результат операции — число **0** или **1**. Операторы можно применять с любыми аргументами.

&

Оператор побитового **AND**. Используется только с целыми операндами.

^

Оператор побитового исключающего **OR**. Применяется только с целыми числами.

|

Оператор побитового **OR**. Применяется только с целыми числами.

&&

Оператор логического **AND**. Результат равен **1**, если оба операнда равны **1**, и **0** — в обратном случае. Операндами могут быть любые числа, как целые, так и с плавающей запятой.

||

Оператор логического **OR**. Результат равен **0**, если оба операнда равны **0**, и **1** — в обратном случае. Операндами могут быть любые числа, как целые, так и с плавающей запятой.

x?y:z

Конструкция **if-then-else**, подобная аналогичной конструкции в языке C. Операнд **x** должен иметь числовое значение. Если значение **x** не равно нулю, то результат команды будет **y**. В обратном случае результат будет **z**.

Операторы одного уровня приоритета исполнения выполняются по очереди, слева направо. Например, команда:

```
expr 4*2 < 7
```

возвращает результат **0**.

Подобно языку C, операторы **&&**, **||** и **?:** имеют свойство «ленивого вычисления», т.е., если операнд не нужен для

получения результата, то он не вычисляется. Например, в команде:

```
expr {$v? [a] : [b]}
```

будет вычислено только одно из выражений **[a]** и **[b]**, в зависимости от значения **$v**. Однако, это справедливо только тогда, когда все выражение заключено в фигурные скобки. В противном случае анализатор Tcl сначала вычислит **[a]** и **[b]**, и только потом вызовет команду **expr**.

Математические функции

Tcl поддерживает в выражениях следующие математические функции:

```
acos  cos  hypot  sinh
asin  cosh  log  sqrt
atan  exp  log10  tan
atan2  floor  pow  tanh
ceil  fmod  sin
```

Каждая из этих функций вызывает одноименную функцию из математической библиотеки.

Кроме них можно использовать также перечисленные ниже функции преобразования чисел и генерации случайных чисел.

abs(arg)
Возвращает абсолютное значение аргумента. Аргумент может быть целым или числом с плавающей точкой. Результат возвращается в такой же форме.

double(arg)
Переводит аргумент в десятичное число в плавающей точкой.

int(arg)
Переводит аргумент в целое число, обрезая дробную часть.

rand()
Возвращает случайное десятичное число в интервале **(0,1)**.

Исходное значение, используемое при генерации, берется от внутренних часов или задается с помощью функции **srand**.

round(arg)
Округляет число до целого.

srand(arg)
Аргумент, который должен быть целым числом, используется для генерации последовательности случайных чисел. Возвращает первое случайное число из последовательности. Каждый интерпретатор может использовать собственное значение и порождать собственную последовательность случайных чисел.

Пользовательские приложения могут определять дополнительные функции, используя процедуру **Tcl_CreateMathFunc()**.

Типы данных, точность вычислений и переполнения

Все внутренние вычисления с целыми числами выполняются с С-типом **long**, все внутренние вычисления с числами с плавающей запятой выполняются с С-типом **double**.

В общем случае, обнаружение переполнения и исчезновения значения при операциях с целыми числами зависит от поведения функций конкретной библиотеки С, и потому для промежуточных результатов не может считаться надежным. Переполнение и исчезновение значения при

операциях с числами с плавающей запятой обнаруживаются на аппаратном уровне, что обычно надежно обеспечивается.

При необходимости выполняются преобразования внутренних представлений операндов между целыми, строковыми и с плавающей запятой. При арифметических вычислениях используются целые числа до тех пор, пока не будет подставлено или указано число с плавающей запятой. После этого тип данных будет с плавающей запятой.

Числа с плавающей запятой возвращаются либо с точкой, либо с буквой **e**, так что они заведомо не похожи на целые значения. Например:

```
expr 5 / 4
```

вернет «1», тогда как

```
expr 5 / 4.0
expr 5 / ( [string length "abcd"] + 0.0 )
```

оба вернут **1.25**.

Выражение

```
expr 20.0/5.0
```

вернет **4.0**, а не **4**.

Операции со строками

Операторы сравнения могут работать со строковыми аргументами, хотя при вычислении выражений аргументы по возможности интерпретируются как целые или числа с плавающей запятой. Если один из операндов строковый, а другой — число, то числовой операнд будет конвертирован в строковый. Например, обе команды

```
expr {"0x03" > "2"}
expr {"0y" < "0x12"}
```

вернут **1**. При этом первое сравнение будет выполнено как сравнение чисел, а второе будет выполнено как сравнение

строк, после того как второй операнд будет преобразован в строку **18**.

Если необходимо сравнить аргументы именно как строки, а операнды могут быть восприняты неоднозначно, то рекомендуется использовать команду **string compare** вместо операторов вида ==.

Правила именования файлов

Все команды Tcl и процедуры C, использующие имена файлов в качестве аргументов, позволяют использовать имена в форме, установленной для данной платформы. Кроме того, на всех платформах Tcl поддерживается синтаксис UNIX с целью предоставления удобного способа составления простых имен файлов. Тем не менее, скрипты, предназначенные для переноса между платформами, не должны пользоваться конкретной формой имен файлов. Вместо этого такие скрипты должны использовать команды **file split** и **file join** для преобразования имен к нужной форме.

Типы путей

Все имена файлов поделены на три типа, в зависимости от начальной точки для отсчета пути: абсолютные, относительные и имена внутри тома (volume-relative).

Абсолютные имена являются самодостаточными, они содержат полный путь файла внутри тома и адрес корневого каталога тома. Относительные имена являются неполными, они указывают положение файла по отношению к текущему каталогу. Имена внутри тома занимают промежуточное положение между первыми двумя, они указывают положение файла относительно корневого каталога текущего тома или относительно текущего каталога указанного тома.

Для определения типа указанного пути можно использовать команду **file pathtype**.

Синтаксис путей

Пути формируются различным образом для различных платформ. Текущая платформа определяется по значению переменной **tcl_platform(platform)**:

Для Macintosh-платформ Tcl поддерживает две формы представления путей: с двоеточием, в обычном для Macintosh стиле, и со слешем, в Unix-стиле. Если путь не содержит двоеточий, то он считается путем в в Unix-стиле. При этом «**.**» означает текущий каталог, «**..**» — родительский каталог для текущего каталога. Однако такие имена, как «/» или «/**..**» считаются именами каталогов в Macintosh-стиле. При этом команды, генерирующие имена файлов, возвращают их в Macintosh-стиле, тогда как команды, использующие имена файлов, могут получать их и в Macintosh-стиле, и в Unix-стиле.

На Unix-платформах используются пути, которые содержат компоненты, разделенные символом слеш. Пути могут быть абсолютными или относительными, имена файлов могут содержать любые символы, кроме слеша. Имена файлов являются специальными и обозначают текущий каталог и родительский каталог текущего каталога, соответственно. Несколько слешей подряд понимаются как один разделитель.

Ниже приведено несколько примеров различных типов путей:

/

Абсолютный путь к корневому каталогу.

/etc/passwd

Абсолютный путь к файлу **passwd** к каталогу **etc** в корневом каталоге.

.

Относительный путь к текущему каталогу.

foo

Относительный путь к файлу **foo** в текущем каталоге

foo/bar

Относительный путь к файлу **bar** в подкаталоге **foo** текущего каталога

../foo

Относительный путь к файлу **foo** в каталоге над текущим.

Для Windows-платформ Tcl поддерживает дисковые и сетевые имена. В обоих типах имен можно использовать в качестве разделителя как прямой, так и обратный слеш. Дисковые имена состоят из (при необходимости) имени диска и последующего абсолютного или относительного пути. Сетевые пути обычно имеют вид **\\servername\sharename\path\file**. В обеих формах «**.**» и «**..**» ссылаются соответственно на текущий каталог и его предка.

«Тильда»-подстановки

Tcl позволяет использовать «тильда»-подстановки в стиле cshell. Если имя файла начинается с «~», за которой сразу следует сепаратор, она заменяется на значение переменной окружения **$HOME**. В противном случае символы от «тильды» до следующего разделителя интерпретируются как имя пользователя и заменяются на имя домашнего каталога пользователя.

На Macintosh- и Windows-платформах «тильда»-подстановки с именем пользователя не поддерживаются. При попытке использовать такое имя файла выдается ошибка. Однако «тильда» без имени пользователя заменяется, как и на Unix-платформах, на значение переменной окружения **$HOME**.

При разработке переносимых приложений необходимо учитывать, что не все файловые системы различают заглавные и прописные буквы. Поэтому следует избегать использования имен файлов, различающихся только регистром букв. Кроме

того, необходимо отказаться от использования символов, имеющих специальное назначение хотя бы на одной из платформ, например, <, >, :, ", /\, |. А также, если предполагается использовать программу на Windows 3.1, необходимо учитывать, что имена файлов при этом должны быть ограничены восемью буквами, а расширения — тремя.

Встроенные переменные

env
errorCode
errorInfo

Библиотека Tcl автоматически создает и управляет следующими глобальными переменными. Во всех случаях, кроме специально оговоренных, эти переменные должны использоваться приложениями и пользователями в режиме только для чтения.

env

Эта переменная задается Tcl в виде массива, элементами которого являются переменные окружения процесса. Чтение элемента даст значение соответствующей переменной окружения. Присвоение значения элементу массива присвоит новое значение переменной окружения или создаст ее, если она не была определена. Отмена (удаление) элемента массива **env** удалит соответствующую переменную окружения. Изменения в массиве будут влиять на окружение, переданное дочерним процессам такими командами, как **exec**. Если будет удален весь массив **env**, то Tcl перестанет отслеживать обращения к **env** и перестанет обновлять значения переменных окружения.

errorCode

После возникновения ошибки данная переменная будет определена для содержания дополнительной информации об ошибке в форме, удобной для обработки программами.

Переменная состоит из списка Tcl с одним или более элементами. Первый элемент списка идентифицирует общий класс ошибок и определяет формат остальной части списка. Ядро Tcl использует следующие ниже форматы **errorCode**, отдельные приложения могут задавать свои собственные форматы.

- **CHILDKILLED pid sigName msg**

Эта форма используется тогда, когда дочерний процесс был удален по сигналу. Элементами, начиная со второго, являются: идентификатор процесса (десятичное число); символическое имя сигнала, вызвавшего прекращение процесса; и короткое сообщение пользователю, описывающее сигнал, например для **SIGPIPE**: «Запись в конвейер без чтения».

- **CHILDSTATUS pid code**

Этот формат используется при окончании дочернего процесса с ненулевым статусом выхода. Второй элемент формата есть идентификатор процесса (десятичное число) и третий элемент есть код выхода, возвращенный процессом (тоже десятичное число).

- **CHILDSUSP pid sigName msg**

Формат используется при временной приостановке процесса по сигналу. Вторым элементом является:

  - идентификатор процесса (десятичное число);
  - символическое имя сигнала, вызвавшего приостановку процесса;
  - короткое сообщение пользователю, описывающее сигнал, например: «Фоновое TTY чтение».

- **NONE**

Формат используется для ошибок, для которых нет никакой дополнительной информации, кроме сообщения, возвращенного вместе с ошибками. В таких случаях список

errorCode будет состоять из одного элемента со значением **NONE**.

- **UNIX errName msg**

Если первый элемент **errorCode** есть **UNIX**, то это означает, что ошибка произошла во время обращения к ядру UNIX. Элемент **errName** содержит символическое имя возникшей ошибки, например, **ENOENT**; имена определены в файле errno.h. Элемент **msg** есть соответствующее **errName** сообщение пользователю, например, для случая **ENOENT**: «Нет такого файла или каталога».

Для задания переменной **errorCode** приложения должны использовать такие библиотечные процедуры, как **Tcl_SetErrorCode** и **Tcl_UnixError**, или использовать команду **error**. Если ни один из этих способов не был использован, то после следующей ошибки интерпретатор Tcl установит переменную на **NONE**.

errorInfo

После возникновения ошибки эта строковая переменная будет содержать одну или более строк, идентифицирующих команды и процедуры Tcl, исполнявшиеся в момент возникновения самой последней ошибки. Ее содержимое принимает форму образа стека, со всеми вложенными командами Tcl, которые в момент возникновения ошибки не были закончены.

Стандартные интерпретаторы

Интерпретатор tclsh

Интерпретатор tclsh представляет собой простую оболочку с алфавитно-цифровым интерфейсом пользователя. Интерпретатор может работать в интерактивном или пакетном режиме. В первом случае он считывает команды Tcl со стандартного входа: пользователь вводит команды с клавиатуры, tclsh обрабатывает их и выводит результат или сообщение об ошибке на стандартный вывод. Во втором случае источником команд для обработки служит указанный дисковый файл. Интерпретатор будет работать до тех пор, пока не будет подана команда **exit**, или пока на стандартный вход не поступит символ конца файла.

Для запуска интерпретатора необходимо в ответ на приглашение операционной системы подать команду

```
tclsh ?fileNamearg arg …?
```

Если никаких аргументов указано не было, то интерпретатор запустится в интерактивном режиме, изображая на дисплее приглашение для ввода команд в виде знака процента «**%**». В ответ на приглашение следует ввести команду Tcl и нажать клавишу **Enter**. Затем ввести следующую команду и снова нажать **Enter**.

Если в домашнем каталоге пользователя существует файл **.tclshrc**, то tclsh обработает этот файл как скрипт Tcl до считывания первой команды со стандартного входа.

Когда tclsh запускается с аргументами, то первый аргумент **fileName** является именем файла со скриптом, а все последующие необязательные аргументы передаются скрипту в качестве переменных. Вместо того, чтобы считывать команды со стандартного входа, интерпретатор будет работать в пакетном режиме: считывать их из указанного файла и завершит свою работу по достижении конца файла. В этом случае автоматической обработки файла **.tclshrc** не делается, но если это необходимо, то можно сослаться на него изнутри файла скрипта.

Переменные

Tclsh задает следующие переменные Tcl:

Argc — содержит счетчик количества аргументов **arg** (если ни одного, то 0), исключая имя файла со скриптом.

Argv — содержит список Tcl, элементами которого являются аргументы **arg**, в порядке их следования, или нулевую строку, если нет ни одного аргумента.

argv0 — содержит **fileName**, если он был задан. В обратном случае содержит имя, при помощи которого был вызван tclsh.

tcl_interactive — содержит 1, если tclsh работает в интерактивном режиме (не было задано **fileName** и стандартный вход есть терминальное устройство). В противном случае содержит 0.

Интерпретатор wish

Интерпретатор wish представляет собой простую программу с двумя рабочими окнами, главным и выходным. Интерпретатор может работать в интерактивном или пакетном режиме. В первом случае он считывает команды Tcl со стандартного входа: пользователь вводит команды с клавиатуры в главном окне, wish обрабатывает их и выводит результат или сообщение об ошибке в выходном окне.

Во втором случае источником команд для обработки служит указанный дисковый файл. В обоих режимах интерпретатор будет работать до тех пор, пока не будут удалены все окна приложения, или пока на стандартный вход не поступит символ конца файла.

Для запуска интерпретатора необходимо в ответ на приглашение операционной системы подать команду

```
wish ?fileName arg arg …?
```

Если никаких аргументов указано не было, или если первый аргумент начинается с символа «**-**», то интерпретатор запустится в интерактивном режиме, изображая на дисплее приглашение для ввода команд в виде знака процента «**%**». В ответ на приглашение следует ввести команду Tcl и нажать клавишу **Enter**. Затем ввести следующую команду и снова нажать **Enter**.

Если в домашнем каталоге пользователя существует файл **.wishrc**, то wish обработает этот файл как скрипт Tcl до считывания первой команды со стандартного входа.

Когда wish запускается с первым аргументом **fileName**, то аргумент считается именем файла со скриптом. Интерпретатор обработает файл **fileName** (создающий преимущественно интерфейс пользователя) в пакетном режиме и затем будет откликаться на события до тех пор, пока все окна не будут удалены. Команды со стандартного входа считываться не будут. В этом случае автоматической обработки файла **.wishrc** не делается, но если это необходимо, то файл скрипта может сослаться на него сам.

Wish автоматически обработает все необязательные аргументы **arg** командной строки из нижеприведенного списка.

-colormap new — указывает на то, что окно должно иметь новую собственную цветовую схему (colormap), а не использовать цветовую схему, заданную по умолчанию.

-display display — задает дисплей (и экран) для отображения окна.

-geometry geometry — задает начальную геометрию для использования в окне. Если этот параметр задан, то его значение сохраняется в глобальной переменной **geometry** интерпретатора Tcl, работающего с приложением.

-name name — задает использование **name** в качестве заголовка окна и имени интерпретатора для использования в командах **send**.

-sync — задает синхронное исполнение команд X сервера, так что все ошибки докладываются немедленно. Это приведет к гораздо более медленному исполнению, но такой режим полезен при отладке.

-use id — дает директиву встроить главное окно приложения в окно с идентификатором **id**, а не создавать его как отдельное самостоятельное окно верхнего уровня. Идентификатор **id** необходимо описать таким же образом, как и значения параметра **-use** для примитивов верхнего уровня (т.е., в форме, возвращаемой командой **winfo id**).

-visual visual — описывает используемый для окна **visual**. Этот параметр может иметь любую форму, поддерживаемую процедурой **Tk_GetVisual**.

-- — дает директиву передать все остальные параметры в переменную скрипта **argv** без интерпретации. Это обеспечивает способ передачи аргументов в скрипт вместо интерпретации их wish.

Любые аргументы командной строки не из этого списка передаются скрипту с помощью переменных **argc** и **argv**.

Имя и класс приложения

Имя приложения (используемое для таких целей, как команды **send**) берется из параметра **-name**, если он был задан. В противном случае оно берется из **fileName**, если оно было

указано, или из командной строки, вызвавшей wish. В последних двух случаях, если имя содержит символ «/» (слеш), то только символы после последнего слеша используются в качестве имени приложения.

Класс приложения (используемый для таких целей, как описание параметров при помощи свойства **RESOURCE_MANAGER** или файла **.Xdefaults**) совпадает с его именем за исключением первой буквы, которая делается заглавной.

Переменные

Wish — задает следующие переменные Tcl:

argc, argv0, tcl_interactive — эти три переменные выполняют для wish точно такую же роль, какую выполняют одноименные переменные для интерпретатора tclsh.

Argv — содержит список Tcl, элементами которого являются аргументы **arg**, которые следуют в командной строке за параметром — или не входят в приведенный выше список различаемых параметров wish, в порядке их следования, или содержит нулевую строку, если нет ни одного аргумента.

geometry — если параметр **-geometry** был задан, то wish копирует его значение в эту переменную. Если после обработки **fileName** эта переменная еще существует, то wish использует значение этой переменной в команде **wm geometry** для задания геометрии главного окна.

Дополнительные возможности

Интерпретаторы можно запускать не только в режиме командной строки, но также и из скрипта. Также можно изменить вид приглашения (символ «%»).

Если создать файл скрипта (этот и следующий примеры даны для интерпретатора tclsh, для wish надо заменить в строках имя tclsh на wish) с первой строкой вида

```
#!/usr/local/bin/tclsh
```

и пометить файл скрипта, как исполняемый, то будет можно запускать интерпретатор прямо из оболочки Unix. В этом случае предполагается, что интерпретатор (tclsh, wish) был установлен в стандартный каталог usr/local/bin; если он был установлен в какое-нибудь иное место, то надо будет откорректировать путь в названной выше строке. При этом эта строка должна удовлетворять возможным требованиями операционной системы Unix на длину строк, начинающихся с **#!** (не более 30 символов).

Еще удобнее будет начинать файлы со скриптами следующими тремя строками:

```
#!/bin/sh
# the next line restarts using tclsh\
exec tclsh "$0" "$@"
```

Этот второй способ имеет три преимущества перед предыдущим способом:

- местонахождение исполняемого файла интерпретатора не нужно прописывать в скрипте: исполняемый файл может находиться в любом месте в пределах пути поиска оболочки пользователя;
- нет необходимости учитывать ограничение на длину строк;
- этот способ будет работать даже тогда, когда файл tclsh или wish сам является скриптом оболочки (иногда так поступают для того, чтобы иметь возможность работать с несколькими архитектурами или операционными системами: скрипт tclsh (wish) выберет для запуска один из нескольких исполняемых файлов).

Эти три строчки позволяют обрабатывать скрипт и оболочке sh, и интерпретатору (tclsh или wish), но при этом **exec** будет запущен только оболочкой sh. Сначала скрипт обрабатывает sh, она считает вторую строку комментарием и исполняет третью строку. Утверждение с **exec** остановит обработку оболочкой и вместо этого запустит интерпретатор для повторной обработки всего скрипта.

Когда интерпретатор запустится, он сочтет все три строки комментариями, потому что обратный слеш в конце второй строки означает для него, что третья строка есть продолжение комментария на второй строке.

Изменить символ приглашения можно при помощи переменных **tcl_prompt1** и **tcl_prompt2**. Если переменная **tcl_prompt1** существует, то она должна содержать скрипт Tcl для вывода приглашения; вместо вывода своего приглашения, интерпретатор будет исполнять скрипт в **tcl_prompt1**. Переменная **tcl_prompt2** используется аналогичным образом, когда при вводе была начата новая строка, но вводимая команда еще не была закончена. Если переменная **tcl_prompt2** не была задана, то для незаконченных команд не будет выводиться никакого приглашения.

Встроенные команды Tcl

Библиотека Tcl предоставляет набор встроенных команд. Эти команды будут доступны в любом приложении, использующем Tcl. Дополнительно к этим командам приложения могут определять свои собственные команды, а также команды, определенные в виде процедур Tcl.

after

Команда **after** указывает, что некоторая команда должна быть выполнена с задержкой по времени.

Синтаксис

```
after ms
after ms?script script script...?
after cancel id
after cancel script script script...
after idle?script script script...?
after info?id?
```

Описание

Команда используется для того, чтобы отложить выполнение программы или выполнить указанную команду в фоновом режиме в некоторый момент в будущем. У команды есть несколько форм, зависящих от ее первого аргумента.

after ms

Ms — целое число, задающее задержку в миллисекундах. Команда обеспечивает задержку на соответствующее число миллисекунд до своего завершения. Во время задержки приложение не реагирует на события.

```
after ms?script script script...?
```


В этой форме команда завершается немедленно, но при этом она обеспечивает выполнение Tcl-команды **script script script** через соответствующее время. Команда выполняется ровно один раз в установленный момент времени. Команда формируется объединением всех перечисленных аргументов **script** таким же образом, как при выполнении команды **concat**. Команда выполняется на глобальном уровне (вне контекста какой-либо Tcl-процедуры). Ошибки во время выполнения команды (если они происходят) обрабатываются с помощью процедуры **bgerror**. Команда **after** в этой форме возвращает идентификатор, который может быть использован для отмены отложенной команды с помощью команды **after cancel**.

after cancel id

Отменяет исполнение ранее заявленной отложенной команды. **Id** определяет, какая именно команда будет отменена. Значение **id** должно совпадать со значением, возвращенным предыдущей командой **after**. Если соответствующая команда уже выполнена, команда **after cancel** игнорируется.

after cancel script script...

Эта команда также отменяет выполнение ранее заявленной отложенной команды. Все перечисленные скрипты **script** объединяются через пробел таким же образом, как при выполнении команды **concat**. После чего ищется отложенная команда с аналогичным скриптом. Если такая команда будет найдена, ее исполнение будет отменено. В противном случае команда **after cancel** игнорируется.

after idle script?script script...?

Все перечисленные скрипты объединяются через пробел таким же образом, как при выполнении команды **concat**. Сформированная таким образом Tcl команда выполняется позже. Она выполняется ровно один раз в первом цикле обработчика событий, в котором не будет других

необработанных событий. Команда **after idle** возвращает идентификатор, который может быть использован для отмены отложенной команды. Ошибки во время выполнения команды (если они происходят) обрабатываются с помощью процедуры **bgerror**.

after info?id?

Эта команда используется для получения информации об отложенных командах. Если аргумент **id** отсутствует, то возвращается список идентификаторов отложенных команд. Если аргумент **id** указан и соответствует идентификатору отложенной команды, которая не отменена и еще не выполнена, возвращается список из двух элементов. Первый элемент списка – Tcl-скрипт соответствующей команды, второй — **idle** или **timer** в зависимости от того, по какому событию предполагается вызов команды.

Команды **after ms** и **after idle** предполагают, что приложение управляется событиями. Отложенные команды не выполняются, если в приложении не активизирован цикл обработки событий. В приложениях, в которых он обычно не активизирован, таких как **tclsh**, цикл обработки событий может быть активизирован с помощью команд **vwait** и **update**.

append

Команда дописывает значения аргументов к значению переменной.

Синтаксис

```
append varName?value value value...?
```

Описание

Команда **append** добавляет все аргументы **value** к значению переменной **varName**. Если такой переменной не было, она будет создана, и ее значение будет равно соединению значений

аргументов **value**. Эта команда предоставляет удобный способ постепенного наращивания длинных переменных. Если переменная **a** содержит длинное значение, то команда **append a $b** выполняется значительно быстрее, чем **set a ab**.

array

Синтаксис

```
array option arrayName?arg arg...?
array anymore arrayName searchId
array donesearch arrayName searchId
array exists arrayName
array get arrayName?pattern?
array names arrayName?pattern?
array nextelement arrayName searchId
array set arrayName list
array size arrayName
array startsearch arrayName
```

Описание

Эта команда предназначена для выполнения перечисленных ниже операций с массивами. Если иное не оговорено специально, **arrayName** должно быть именем существующего массива. Аргумент **option** определяет конкретную операцию. Для команды определены перечисленные ниже опции.

array anymore arrayName searchId

Возвращает «**1**» если при выполнении команды поиска остались невыбранные элементы массива, и «**0**» в противном случае. **SearchId** указывает операцию поиска, информация о которой запрашивается (величина **searchId** возвращается при выполнении команды **array startsearch**). Эта опция особенно

удобна, если массив содержит элемент с пустым именем, поскольку команда **array nextelement** не позволяет в таком случае определить, закончен ли поиск.

array donesearch arrayName searchId

Команда прерывает поиск элементов массива и удаляет всю связанную с поиском информацию. **SearchId** указывает операцию поиска, информация о которой удаляется (величина **searchId** возвращается при выполнении команды **array startsearch**). Команда возвращает пустую строку.

array exists arrayName

Возвращает «**1**», если **arrayName** есть имя массива, и «**0**», если такой переменной не существует или она является скалярной переменной.

array get arrayName?pattern?

Возвращает список, содержащий пары элементов. Первый элемент пары — имя элемента массива **arrayName**, второй элемент пары — значение этого элемента. Порядок пар не определен. Если шаблон не задан, то все элементы массива будут включены в результат. Если шаблон задан, то в результат будут включены только те элементы, чьи имена соответствуют шаблону (используя те же правила, что и в команде **glob**). Если **arrayName** не является переменной массива или массив не содержит элементов, то возвращается пустой список.

array names arrayName?pattern?

Возвращает список, содержащий имена всех элементов массива, соответствующих шаблону (используя те же правила, что и в команде **glob**). Если шаблона нет, то команда возвращает имена всех элементов массива. Если в массиве нет элементов, соответствующих шаблону или **arrayName** не является именем переменной массива, то возвращается пустая строка.

array nextelement arrayName searchId

Возвращает имя следующего элемента массива **arrayName** или пустую строку, если все элементы массива уже возвращены. **SearchId** указывает операцию поиска, (величина **searchId** возвращается при выполнении команды **array startsearch**). Предупреждение: если в массив внесен новый элемент или из массива удален один из элементов, то все операции поиска в этом массиве автоматически заканчиваются, как если бы была выполнена команда **array donesearch**. Соответственно, попытка выполнить после этого команду **array nextelement** приведет к ошибке.

array set arrayName list

Устанавливает значение одного или нескольких элементов массива **arrayName**. Список **list** должен иметь такую же структуру, как список, возвращаемый командой **array get**, то есть состоять из четного числа элементов. Все нечетные элементы списка рассматриваются как имена элементов массива **arrayName**, а следующие за ними четные элементы — как новые значения соответствующих элементов.

array size arrayName

Возвращает строку, содержащую десятичное число, равное количеству элементов указанного массива. Если **arrayName** не является именем массива, возвращается «**0**».

array startsearch arrayName

Эта команда инициализирует процесс поиска элементов указанного массива. После этого имя каждого следующего элемента массива можно получить с помощью команды **array nextelement**. По завершении поиска необходимо выполнить команду **array donesearch**. Команда **array startsearch** возвращает идентификатор процесса поиска, который должен использоваться в командах **array nextelement** и **array donesearch**. Благодаря этому механизму возможно проведение нескольких

процессов поиска элементов одного и того же массива одновременно.

bgerror

Команда **bgerror** предназначена для обработки фоновых ошибок (background errors).

Синтаксис

```
bgerror  сообщение
```

Описание

В Tcl нет встроенной команды **bgerror**. Если в приложении тем не менее необходимо обрабатывать фоновые ошибки, пользователь может определить собственную команду **bgerror**, например, как Tcl-процедуру.

Фоновые ошибки — это ошибки в командах, которые не вызваны непосредственно из приложения. Например, фоновыми являются ошибки в командах, вызванных с помощью конструкции **after**. Для нефоновых ошибок сообщение об ошибке возвращается через вложенные вызовы команд, пока не достигнет верхнего уровня приложения. После этого приложение выдает сообщение об ошибке в одной из команд верхнего уровня. При фоновой ошибке подобный процесс не достигает команд верхнего уровня и формирование сообщения об ошибке оказывается затруднительным.

Когда Tcl обнаруживает фоновую ошибку, он сохраняет информацию об ошибке и вызывает команду **bgerror** с помощью обработчика событий. Перед вызовом **bgerror** восстанавливаются значения переменных **errorInfo** и **errorCode**, которые были при обнаружении ошибки. После этого вызывается команда **bgerror** с единственным аргументом — сообщением об ошибке. Предполагается, что в приложении определена команда **bgerror** и что она выдает сообщение об ошибке надлежащим образом.

Если при выполнении команды **bgerror** не произошло новой ошибки, возвращаемый ею результат игнорируется.

Если при исполнении команды **bgerror** произошла новая ошибка (например, если эта команда не существует), сообщение об ошибке поступает в канал вывода ошибок.

Если до вызова обработчиком событий команды **bgerror** произошло несколько фоновых ошибок, то, как правило, команда будет вызвана для каждой из обнаруженных ошибок. Однако, если сама команда **bgerror** возвращает код **break**, последующие ее вызовы пропускаются.

В чисто Tcl-приложениях команда **bgerror** не реализована. Однако, в Tk-приложениях определена процедура **bgerror**, которая выводит сообщение об ошибке в диалоговое окно и позволяет пользователю просмотреть стек, описывающий, где именно эта ошибка произошла.

binary

Команда вставляет и извлекает поля из двоичных строк.

Синтаксис

```
binary  format     formatString?arg  arg...?

binary  scan  string  formatString?varName  varName...?
```

Описание

Команда **binary** предоставляет средства для манипулирования двоичными данными. Первая из форм команды конвертирует обычные Tcl-значения **arg** в двоичное число. Например, вызванная с аргументами **16** и **22**, она вернет 8-байтовую двоичную строку, состоящую из двух 4-байтовых чисел, представляющих соответственно **16** и **22**. Вторая форма команды выполняет обратное действие, она извлекает из двоичной строки данные и возвращает их как обычные Tcl строки.

binary format

Команда **binary format** создает двоичную строку по правилам, заданным с помощью шаблона **formatString**. Содержание этой строки задается дополнительными аргументами **arg**. Команда возвращает сформированную двоичную строку.

Шаблон **formatString** содержит последовательность из нуля или более спецификаторов преобразования, разделенных одним или более пробелами. Каждый спецификатор преобразования состоит из буквы, за которой может следовать число **count**. Как правило, спецификатор использует один из аргументов **arg** чтобы получить величину для форматирования. Буква в спецификаторе указывает тип преобразования (форматирования). Число **count** обычно указывает сколько объектов для форматирования выбирается из значения **arg**. Соответственно, **count** должно быть неотрицательным десятичным числом. Если значение **count** равно «*****», это обычно указывает, что надо использовать все значение аргумента. Если число аргументов **arg** не соответствует числу спецификаторов, требующих для себя дополнительного аргумента, выдается ошибка.

Обычно результат каждого нового преобразования дописывается в конец возвращаемой строки. Однако, с помощью специальных спецификаторов точку ввода нового значения (курсор) можно передвигать по формируемой строке.

Ниже приведены допустимые значения спецификаторов преобразований и описаны соответствующие преобразования.

- **a?count?**

Передает в выходную строку **count** символов из соответствующего аргумента **arg**. Если в **arg** содержится меньше **count** байт, добавляются нулевые байты. Если в **arg** содержится больше **count** байт, «лишние» байты игнорируются. Если **count**

равно «*****», используются все байты из **arg**. Если **count** отсутствует, используется один байт. Например, команда:

```
binary format a7a*a alpha bravo charlie
```

вернет строку, эквивалентную **alpha\000\000bravoc**.

- **A?count?**

То же, что и **a**, за исключением того, что для заполнения используются не нулевые байты, а пробелы. Например, команда:

```
binary format A6A*A alpha bravo charlie
```

вернет **alpha bravoc**.

- **b?count?**

Передает в выходную строку **count** бит в порядке от младших к старшим в каждом байте. Аргумент **arg** должен состоять из последовательности нулей и единиц. Порядок байтов в выходной строке тот же, что и во входной информации. Лишние биты отсекаются, недостающие дополняются нулями. По умолчанию форматируется один бит. Если число форматируемых битов не соответствует целому числу байтов, недостающие биты дополняются нулями. Например, команда:

```
binary format b5b* 11100 111000011010
```

вернет строку, эквивалентную **\x07\x87\x05**.

- **B?count?**

Тоже, что и **b**, но биты выдаются в порядке от старших к младшим. Например, команда:

```
binary format B5B* 11100 111000011010
```

вернет строку, эквивалентную **\xe0\xe1\xa0**.

- **h?count?**

Передает в выходную строку **count** шестнадцатеричных чисел в порядке от младших к старшим в каждом байте. Аргумент **arg** должен состоять из последовательности символов,

содержащихся в множестве: «0123456789abcdefABCDEF». Порядок байтов в выходной строке тот же, что и во входной информации. Лишние символы отсекаются, недостающие дополняются нулями. По умолчанию форматируется одно шестнадцатеричное число. Если число форматируемых чисел не соответствует целому числу байтов, недостающие биты дополняются нулями.

Например, команда:

```
binary format h3h* AB def
```

вернет строку, эквивалентную **\xba\xed\x0f**.

- **H?count?**

Тоже, что и **h**, но биты выдаются в порядке от старших к младшим. Например, команда:

```
binary format H3H* ab DEF
```

вернет строку, эквивалентную **\xab\xde\xf0**.

- **c?count?**

Передает в выходную строку одно или несколько восьмибитных чисел. Если **count** не указан, аргумент **arg** должен состоять из одного целого числа. В противном случае он должен состоять из списка, содержащего не менее **count** целых чисел. Последние 8 бит каждого числа сохраняются как один байт в выходной строке. Если **count** равно «*», форматируются все числа в списке. Если в списке меньше чисел, чем **count**, выдается ошибка. Лишние числа в списке игнорируются.

Например, команда:

```
binary format c3cc* {3 -3 128 1} 257 {2 5}
```

вернет строку, эквивалентную **\x03\xfd\x80\x01\x02\x05**, тогда как команда:

```
binary format c {2 5}
```

вернет ошибку.

- **s?count?**

Эта форма аналогична **c**, за исключением того, что она сохраняет 16-ти битовые числа. Последние шестнадцать бит числа сохраняются как два байта, последний байт первым. Например, команда:

```
binary format s3 {3 -3 258 1}
```

вернет строку, эквивалентную **\x03\x00\xfd\xff\x02\x01**.

- **S?count?**

Эта форма аналогична **s**, за исключением того, что порядок байтов противоположный. Например, команда:

```
binary format S3 {3 -3 258 1}
```

вернет строку, эквивалентную **\x00\x03\xff\xfd\x01\x02**.

- **i?count?**

Эта форма аналогична **c**, за исключением того, что она сохраняет 32-х разрядные числа. Например, команда:

```
binary format i3 {3 -3 65536 1}
```

вернет строку, эквивалентную **\x03\x00\x00\x00\xfd\xff\xff\xff\x00\x00\x10\x00**.

- **I?count?**

Эта форма отличается от **i** порядком байтов в числе. Например, команда:

```
binary format I3 {3 -3 65536 1}
```

вернет строку, эквивалентную **\x00\x00\x00\x03\xff\xff\xff\xfd\x00\x10\x00\x00**.

- **f?count?**

Эта форма аналогична **c**, за исключением того, что она сохраняет числа с плавающей запятой в машинном представлении (используемом для конкретной платформы).

Например, на Windows платформе с процессором Pentium команда:

```
binary format f2 {1.6 3.4}
```

вернет строку, эквивалентную **\xcd\xcc\xcc\x3f\x9a\x99\x59\x40**.

- **d?count?**

Эта форма аналогична **f**, за исключением того, что используется представление для десятичных чисел с двойной точностью. Например, на Windows платформе с процессором Pentium команда:

```
binary format d1 {1.6}
```

вернет строку, эквивалентную **\x9a\x99\x99\x99\x99\x99\xf9\x3f**.

- **x?count?**

Записывает в выходную строку **count** нулевых байтов. Если **count** не задан, записывает один нулевой байт. Если **count** равно «*», выдает ошибку. Ни один из аргументов **arg** при форматировании не используется. Например, команда:

```
binary format a3xa3x2a3 abc def ghi
```

вернет строку, эквивалентную **abc\000def\000\000ghi**.

- **X?count?**

Передвигает место будущей вставки в выходную строку (курсор) на count байтов назад. Если аргумент **count** равен «*» или больше текущей позиции, передвигает курсор на начало строки. Если **count** не указан, передвигает его на один байт. Ни один из аргументов **arg** при форматировании не используется. Например, команда

```
binary format a3X*a3X2a3 abc def ghi
```

вернет **dghi**.

- **@ ?count?**

Передвигает место будущей вставки в выходную строку (курсор) на абсолютную позицию **count**. Позиция 0

соответствует первому байту в строке. Если **count** больше текущей длины строки, она дополняется до необходимой позиции нулевыми байтами. «*» указывает на конец строки. Если аргумент **count** не указан, команда возвращает ошибку. Ни один из аргументов **arg** при форматировании не используется. Например, команда:

```
binary format a5@2a1@*a3@10a1 abcde f ghi j
```

вернет **abfdeghi\000\000j**.

binary scan

Команда **binary scan** просматривает двоичную строку **string**, возвращая число выполненных преобразований. Строка **formatString** содержит последовательность спецификаторов преобразования. Каждый из аргументов **varName** содержит имя переменной, в которую записывается результат очередного преобразования.

Поскольку спецификаторы преобразований строятся по тем же правилам, что и для **binary format**, ниже приведены примеры их использования с минимальными пояснениями. Принципиальным моментом при исполнении команды **binary scan** является то, что если для очередного преобразования, указанного в **formatString**, требуется больше байтов, чем осталось до конца строки, то команда немедленно завершается, а соответствующая переменная (в которую должен был записываться результат преобразования) остается неизмененной. Если для записи результата очередного преобразования не осталось переменной, команда возвращает ошибку.

- **a?count?**

Команда:

```
binary scan abcde\000fghi a6a10 var1 var2
```

вернет значение **1**. В переменной **var1** будет записана строка **abcde\000**, а переменная **var2** не изменится.

- **A?count?**

  Команда

  ```
  binary scan "abc efghi\000" a* var1
  ```

вернет **1**. В переменной **var1** будет записана строка **abc efghi**.

- **b?count?**

  Команда

  ```
  binary scan\x07\x87\x05 b5b* var1 var2
  ```

вернет **2**. В переменной **var1** будет записано **11100**, в переменной **var2** будет записано **1110000110100000**.

- **B?count?**

  Команда

  ```
  binary scan\x70\x87\x05 b5b* var1 var2
  ```

вернет **2**. В переменной **var1** будет записано **01110**. В переменной **var2 — 1000011100000101**.

- **h?count?**

  Команда

  ```
  binary scan\x07\x86\x05 h3h* var1 var2
  ```

вернет **2**. В переменной **var1** будет записано **706**. В переменной **var2 — 50**.

- **H?count?**

  Команда

  ```
  binary scan\x07\x86\x05 H3H* var1 var2
  ```

вернет **2**. В переменной **var1** будет записано **078**. В переменной **var2 — 05**.

- **c?count?**

  Команда

  ```
  binary scan\x07\x86\x05 c2c* var1 var2
  ```

вернет **2**. В переменной **var1** будет записана строка **7 -122**. В переменной **var2 — 5**. Обратите внимание, что команда возвращает числа со знаком. Чтобы преобразовать их в числа без знака можно использовать выражение:

```
expr ( $num + 0x100 ) % 0x100
```

- **s?count?**

  Команда

  ```
  binary scan\x05\x00\x07\x00\xf0\xff s2s* var1 var2
  ```

вернет **2**. В переменной **var1** будет записана строка **5 7**. В переменной **var2 — -16**. Обратите внимание, что команда возвращает числа со знаком. Чтобы преобразовать их в числа без знака можно использовать выражение:

```
expr ( $num + 0x10000 ) % 0x10000
```

- **S?count?**

  Команда

  ```
  binary scan\x00\x05\x00\x07\xff\xf0 S2S* var1 var2
  ```

вернет **2**. В переменной **var1** будет записана строка **5 7**. В переменной **var2 — -16**.

- **i?count?**

  Команда

  ```
  binary can\x05\x00\x00\x00\x07\x00\x00\x00\xf0\xff\xff\xff i2i*
  var1 var2
  ```

вернет **2**. В переменной **var1** будет записана строка **5 7**. В переменной **var2 — -16**. Обратите внимание, что команда возвращает числа со знаком. В Tcl нет возможности перевести его в число без знака.

- **I?count?**

  Команда

  ```
  binary\x00\x00\x00\x05\x00\x00\x00\x07\xff\xff\xff\xf0 I2I* var1
  var2
  ```

вернет **2**. В переменной **var1** будет записана строка **5 7**. В переменной **var2 — -16**.

- **f?count?**

  На Windows платформе с процессором Pentium команда

  ```
  binary scan\x3f\xcc\xcc\xcd f var1
  ```

вернет **1**. В переменной **var1** будет записано **1.6000000238418579**.

- **d?count?**

  На Windows платформе с процессором Pentium команда

  ```
  binary scan\x9a\x99\x99\x99\x99\xf9\x3f d var1
  ```

вернет **1**. В переменной **var1** будет записано **1.6000000000000001**.

- **x?count?**

  Команда

  ```
  binary scan\x01\x02\x03\x04 x2H* var1
  ```

вернет **1**. В переменной **var1** будет **0304**. Преобразование не требует дополнительной переменной **varName**.

- **X?count?**

  Команда

  ```
  binary scan\x01\x02\x03\x04 c2XH* var1 var2
  ```

вернет **2**. В переменной **var1** будет **1 2**. В переменной **var2 — 020304**. Преобразование не требует дополнительной переменной **varName**.

- **@?count?**

  Команда

  ```
  binary scan\x01\x02\x03\x04 c2@1H* var1 var2
  ```

вернет **2**. В переменной **var1** будет **1 2**. В переменной **var2 — 020304**. Преобразование не требует дополнительной переменной **varName**.

break

Команда прекращает выполнение цикла.

Синтаксис

```
break
```

Описание

Обычно данная команда помещается внутрь цикла, например созданного командами **for**, **foreach** или **while**. Команда возвращает код **TCL_BREAK**, который вызывает завершение исполнения наименьшего охватывающего ее цикла. Цикл завершается нормальным образом (без ошибки) как если бы он отработал до конца. Код **TCL_BREAK** обрабатывается также в некоторых других ситуация — при исполнении команды **catch**, при обработке событий и в скрипте самого верхнего уровня.

case

Команда **case** исполняет один из нескольких скриптов в зависимости от полученного значения.

Синтаксис

```
case string?in? patList body?patList body...?
case string?in? {patList body? body...?}
```

Описание

Замечание: команда **case** — устаревшая, она поддерживается только для совместимости с предыдущими версиями. В последующих версиях она может быть опущена. Поэтому предпочтительнее использование команды **switch**.

Команда сравнивает **string** со всеми аргументами **patList** по очереди. Каждый аргумент **patList** представляет собой один или несколько (список) образцов. Если **string** соответствует одному из образцов, то **case** рекурсивно вызывает интерпретатор Tcl, передает ему следующий за этим списком образцов скрипт **body** для выполнения и возвращает результат этого выполнения. Каждый аргумент **patList** состоит из одного или нескольких образцов. Каждый образец может содержать спецсимволы, как в команде **string match**. Кроме того, есть специальный образец **default**. Соответствующий ему скрипт выполняется, если **string** не соответствует никакому другому образцу.

Если **string** не соответствует ни одному из образцов, а образец **default** не используется, то **case** вернет пустую строку.

У команды есть две формы записи для аргументов **patList** и **body**. Первая использует отдельные аргументы для каждого шаблона и команды. Эта форма более удобна, если в образцах или командах желательно выполнить какие-либо подстановки. Во второй форме все шаблоны и команды объединены в один аргумент, который должен быть списком, состоящим из шаблонов и команд. Вторая форма позволяет проще записывать многострочные команды, поскольку при этом фигурные скобки вокруг списка позволяют не ставить обратный слеш в конце каждой строки. Однако из-за этих фигурных скобок подстановки в шаблонах и командах не производятся. Поэтому одна и та же команда, записанная в различных формах, может работать по-разному.

catch

Команда выполняет скрипт и обрабатывает ошибки, если они возникают.

Синтаксис

```
catch script?varName?
```

Описание

Данную команду можно использовать для того, чтобы не дать возникшим ошибкам прекратить процесс интерпретации команд. Для исполнения скрипта **script** команда **catch** рекурсивно вызывает интерпретатор Tcl и всегда возвращает код **TCL_OK**, независимо от возможно возникших при исполнении скрипта **script** ошибок.

Команда **catch** возвращает десятичную строку, содержащую код, возвращаемый Tcl-интерпретатором по исполнению скрипта. Если при исполнении скрипта не возникло ошибок, возвращается код **0** (**TCL_OK**). В противном случае возвращается ненулевое значение, соответствующее коду прерывания. Если задан аргумент **varName**, он определяет имя переменной, которой присваивается значение, возвращаемое скриптом (результат выполнения или сообщение об ошибке).

Команда **catch** обрабатывает все прерывания, в том числе от команд **break** и **continue**.

cd

Команда предназначена для перехода в другой каталог.

Синтаксис

```
cd ?dirName?
```

Описание

Команда делает текущим каталогом каталог **dirName**, или, если параметр **dirName** не указан, каталог, заданный в

переменной окружения **HOME**. Команда возвращает пустую строку.

clock

С помощью команды **clock** можно получить и преобразовать значение времени.

Синтаксис

```
clock option?arg arg...?
```

Описание

Команда выполняет одно из перечисленных ниже действий, с помощью которых можно получить и преобразовать строки или значения, являющиеся той или иной формой записи времени. Параметр **option** определяет выполняемое командой действие, одно из следующего списка (параметр может быть сокращен):

clock clicks

Возвращает целое число, с высокой точностью представляющее машинное время. Единица измерения зависит от операционной системы и компьютера и представляет собой минимальную доступную величину, например, счетчик циклов процессора. Эта величина может использоваться только для относительного измерения пройденного времени.

clock format clockValue?-format string??-gmt boolean?

Переводит целое число, возвращаемое командами **clock seconds**, **clock scan** или командами **file atime**, **file mtime** и **file ctime** в удобочитаемую форму. Если в команде присутствует аргумент **-format**, следующий аргумент должен быть строкой, описывающей формат представления времени. Строка состоит из описаний полей, состоящих из символа «**%**» и буквы. Все остальные символы в строке просто копируются в результат. Ниже перечислены допустимые описания полей.

- %% — вставляет %.
- %a — сокращенное название дня недели (Mon, Tue и т.д.).
- %A — полное название дня недели (Monday, Tuesday и т.д.).
- %b — сокращенное название месяца (Jan, Feb и т.д.).
- %B — полное название месяца.
- %c — локальные дата и время.
- %d — день месяца (01–31).
- %H — часы в двадцатичетырехчасовом формате (00–23).
- %I — часы в двенадцатичасовом формате (00–12).
- %j — день года (001–366).
- %m — номер месяца (01–12).
- %M — минуты (00–59).
- %p — AM/PM индикатор (до/после полудня).
- %S — секунды (00–59).
- %U — неделя года (01–52), Воскресенье — первый день недели.
- %w — номер дня недели (Воскресенье = 0).
- %W — неделя года (01–52), Понедельник первый день недели.
- %x — локальный формат даты.
- %X — локальный формат времени.
- %y — год без столетия (00–99).
- %Y — год со столетием (например, 1990).
- %Z — имя часового пояса.

Кроме того, в некоторых операционных системах могут поддерживаться:

- %D — дата в формате %m/%d/%y.
- %e — день месяца (1–31), без нулей впереди.
- %h — сокращенное имя месяца.
- %n — новая строка.
- %r — время в формате %I:%M:%S %p.
- %R — время в формате %H:%M.
- %t — табуляция.
- %T — время в формате %H:%M:%S.

Если аргумент **-format** не задан, используется формат **%a %b %d %H:%M:%S %Z %Y**. Если задан аргумент **-gmt**, следующий аргумент должен быть булевой величиной. Значение **true** означает, что используется время по Гринвичу, значение **false** означает, что используется время по локальному часовому поясу, который задан для операционной системы.

clock scan dateString?-base clockVal??-gmt boolean?

Переводит дату **dateString** в целое число. Команда может перевести в число любую стандартную строку, содержащую время и/или дату, включая название часового пояса. Если строка содержит только время, предполагается текущая дата. Если название часового пояса не указано, предполагается локальный часовой пояс (если значение опции **-gmt** не равно **true**. В этом случае предполагается, что время задано по Гринвичу).

Если в команде указан флаг **-base**, следующий аргумент должен содержать время в виде целого числа. По этому числу определяется дата и используется вместо указанной в строке или текущей. Такая возможность полезна при необходимости перевести в целое число время на заданную дату.

Аргумент **dateString** должен состоять из одной или более спецификаций следующих видов:

time — время суток в форме: **hh?:mm?:ss???meridian??zone?** или в форме **hhmm?meridian??zone?**, где **meridian** — индикатор **AM** или **PM**, **zone** — имя часового пояса. Если индикатор **meridian** не определен, **hh** считается числом часов в двадцатичетырехчасовом формате.

Date — месяц, день и, возможно, год. Допустимые форматы **mm/dd?/yy?**, **monthname dd?, yy?, dd monthname?yy?** и **day, dd monthname yy**. По умолчанию год считается текущим годом. Если год меньше 100, то года 00-38 считаются годами в диапазоне 2000-2038, а года 70-99 считаются годами в диапазоне 1970-1999. Года 39-70 могут быть недопустимыми на некоторых платформах. Для тех платформ, для которых они определены, они считаются годами в диапазоне 1939-1999.

relative time — время относительно текущего момента. Формат — число единица измерения. Возможные единицы измерения **year**, **fortnight**, **month**, **week**, **day**, **hour**, **minute** (или **min**), и **second** (или **sec**). Единицы измерения могут указываться во множественном числе, например **3 weeks**. Кроме того, могут использоваться модификаторы: **tomorrow**, **yesterday**, **today**, **now**, **last**, **this**, **next**, **ago**.

Реальная дата вычисляется в следующей последовательности. Сначала определяется абсолютная дата и/или время, которые переводятся в целое число. Это число используется как базис, к которому добавляется заданный день недели. Далее используется относительное время. Если задана дата, а время (абсолютное или относительное) отсутствует, считается, что это полночь. И последним шагом производится коррекция вычисленной даты, при которой учитываются летнее время и число дней в различных месяцах.

clock seconds

Возвращает время в секундах от начала «эпохи». Может использоваться для вычисления разности времен. «Эпоха» зависит от используемой операционной системы и компьютера.

close

Команда закрывает открытый канал.

Синтаксис

```
close channelId
```

Описание

Команда **close** закрывает канал, идентификатор которого задан аргументом **channelId**. Идентификатор **channelId** возвращается командами **open** и **socket** при открытии канала.

Команда отправляет все накопившиеся в выходном буфере данные на выходное устройство канала, удаляет все данные во входном буфере, закрывает назначенное каналу устройство или файл. Доступ к каналу прекращается.

Для каналов в блокирующем и неблокирующем режимах действие команды несколько различно. Если канал находится в блокирующем режиме, команда завершается только после завершения вывода данных из буфера. В противном случае команда завершается немедленно, а вывод данных из буфера производится в фоновом режиме. Канал закрывается после завершения вывода.

Если канал в блокирующем режиме открыт для конвейера, команда **close** завершается после завершения порожденного процесса.

Если канал совместно используется несколькими интерпретаторами, то команда делает канал **channelId** недоступным в вызвавшем команду интерпретаторе, но не оказывает никакого другого действия на канал, пока все

использующие канал интерпретаторы не закроют его. При выполнении команды в последнем из интерпретаторов, использовавших его, выполняются описанные выше действия.

Команда **close** возвращает пустую строку. Она может порождать ошибку, если при выводе данных произошла ошибка.

concat

Команда соединяет списки в один общий список.

Синтаксис

```
concat?arg arg...?
```

Описание

Аргументы **arg** считаются списками, команда **concat** объединяет их в один общий список. При этом она удаляет пробелы в начале и конце **arg**, и вставляет по одному пробелу между ними. Команда допускает произвольное число аргументов. Например, команда:

```
concat a b {c d e} {f {g h}}
```

возвращает список **a b c d e f {g h}**.

Если не задано никаких аргументов, то команда возвращает пустую строку.

continue

Команда **continue** прекращает выполнение текущего шага цикла.

Синтаксис

```
continue
```

Описание

Обычно данная команда помещается внутрь цикла, например, созданного командами **for**, **foreach** или **while**. Команда

возвращает код **TCL_CONTINUE**, который вызывает завершение текущей итерации наименьшего охватывающего ее цикла. Управление возвращается команде цикла, которая начинает следующий шаг цикла. Код **TCL_CONTINUE** обрабатывается также в некоторых других ситуациях — при исполнении команды **catch** и в скрипте самого верхнего уровня.

eof

Команда проверяет в канале условие конца файла.

Синтаксис

```
eof channelId
```

Описание

Команда **eof** возвращает **1**, если во время последней операции ввода в канале **channelId** произошло условие конца файла, и **0** — в противном случае.

error

Команда генерирует ошибку.

Синтаксис

```
error message?info??code?
```

Описание

Команда возвращает код **TCL_ERROR**, прерывающий интерпретацию команды. Строка **message** возвращается приложению, чтобы указать, что именно произошло.

Если задан непустой аргумент **info**, его значение присваивается глобальной переменной **errorInfo**.

Переменная **errorInfo** обычно используется для формирования сведений о вложениях команды, в которой произошла ошибка. Другими словами, как только оказывается,

что невыполненная команда была вложена в другую команду, информацию об этой команде добавляется к **errorInfo**. Если же аргумент **info** был задан, этого не происходит. Эта особенность позволяет при использовании команды **error** совместно с командой **catch** выдать информацию о реальном месте ошибки (а не о месте вызова команды **error**). Для этого можно использовать следующую конструкцию:

```
catch {...} errMsg
set savedInfo $errorInfo
...
error $errMsg $savedInfo
```

Если задан аргумент **code**, то его значение будет присвоено глобальной переменной **errorCode**. Эта переменная предназначена для хранения машинного описания ошибки в тех случаях, когда такое описание возможно. Если аргумент не задан, переменной **errorCode** в процессе обработки Tcl-интерпретатором ошибки, порожденной командой, присваивается значение **NONE**.

eval

Команда исполняет Tcl-скрипт.

Синтаксис

```
eval arg?arg...?
```

Описание

Аргументы команды **eval** вместе образуют Tcl-скрипт, состоящий из одной или нескольких команд. Команда соединяет аргументы (подобно команде **concat**) и передает их рекурсивно запущенному интерпретатору Tcl. Команда возвращает результат работы интерпретатора (или обнаруженную им ошибку).

exec

Команда запускает подпроцессы.

Синтаксис

```
exec?switches? arg?arg...?
```

Описание

Аргументы команды определяют один или нескольких подпроцессов, которые необходимо выполнить. Аргументы принимают форму конвейера, в котором каждый **arg** задает одно слово команды, а каждая отдельная команда порождает подпроцесс.

Если первые аргументы команды начинаются со знака «**-**», они считаются ключами команды, а не частью описания конвейера.

Возможные ключи:

-keepnewline — сохраняет конечные пробелы в выходных данных конвейера. Обычно они отсекаются.

- — отмечает конец ключей. Аргумент, следующий за этим ключом, рассматривается как первый аргумент **arg**, даже если он начинается со знака «**-**».

Перечисленные ниже аргументы (пары аргументов) используют не для определения подпроцессов, а для перенаправления потоков ввода и вывода между ними. В выражения типа **< fileName fileName** может писаться как отдельно, так и слитно (**<fileName>**).

| — разделяет команды в конвейере. Стандартный вывод предыдущей команды направляется на стандартный вход следующей команды.

|& — разделяет команды в конвейере. Стандартный вывод и стандартный вывод ошибок предыдущей команды направляются на стандартный вход следующей команды. Такое выражение перебивает выражения типа **2>** и **>&**.

< fileName — файл **fileName** отрывается и используется как стандартный ввод для первой команды конвейера.

<@ fileId — в этой форме **fileId —** это идентификатор файла, открытого с помощью команды **open**. Он используется как стандартный ввод для первой команды конвейера. Файл должен быть открыт для чтения.

<< value — **value** используется как стандартный ввод для первой команды конвейера.

> fileName — стандартный вывод последней команды перенаправляется в файл **fileName** и перезаписывает его содержимое.

2> fileName — стандартный вывод ошибок всех команд в конвейере перенаправляется в файл **fileName** и перезаписывает его содержимое.

>& fileName — стандартный вывод последней команды и стандартный вывод ошибок всех команд в конвейере перенаправляется в файл **fileName** и перезаписывает его содержимое.

>>fileName — стандартный вывод последней команды перенаправляется в файл **fileName** и добавляется к его прежнему содержимому.

2>>fileName — стандартный вывод ошибок всех команд в конвейере перенаправляется в файл **fileName** и добавляется к его прежнему содержимому.

>>& fileName — стандартный вывод последней команды и стандартный вывод ошибок всех команд в конвейере перенаправляется в файл **fileName** и добавляется к его прежнему содержимому.

>@ fileId — в этой форме **fileId** — это идентификатор файла, открытого с помощью команды **open**. Стандартный вывод последней команды перенаправляется в файл **fileName** и перезаписывает его содержимое. Файл должен быть открыт для записи.

2>@ fileId — в этой форме **fileId** — это идентификатор файла, открытого с помощью команды **open**. Стандартный вывод ошибок всех команд в конвейере перенаправляется в файл **fileName** и перезаписывает его содержимое. Файл должен быть открыт для записи.

>&@ fileId — в этой форме **fileId** — это идентификатор файла, открытого с помощью команды **open**. Стандартный вывод последней команды и стандартный вывод ошибок всех команд в конвейере перенаправляется в файл **fileName** и перезаписывает его содержимое. Файл должен быть открыт для записи.

Если стандартный вывод последней команды конвейера не был перенаправлен, то команда **exec** возвращает его значение. Если одна из команд конвейера вернула код ошибки, была прервана или приостановлена, то команда **exec** вернет код ошибки. При этом сообщение об ошибке будет включать стандартный вывод конвейера и сообщение об ошибке. В переменной **errorCode** будет записана дополнительная информация о последней встреченной ошибке. Если хотя бы одна из команд конвейера пишет информацию об ошибках в файл и стандартный вывод ошибок не перенаправлен, команда **exec** вернет ошибку, сообщение об ошибке будет включать в себя стандартный вывод конвейера, дополненный сообщениями об ошибках (если их было несколько) и стандартным выводом ошибок.

Если последний символ результата исполнения конвейера или сообщения об ошибке — перевод каретки, то он будет удален из результата или сообщения соответственно. Это соответствует общему правилу Tcl, по которому возвращаемая величина, как правило, не оканчивается символом перевода каретки. Однако, если указана опция **-keepnewline**, символ перевода каретки в конце сохраняется.

Если стандартный ввод конвейера не перенаправлен с помощью символов <, << или <@, стандартный ввод в первую

команду конвейера осуществляется со стандартного ввода приложения.

Если последним аргументом конвейера является **&**, конвейер выполняется в фоновом режиме. В этом случае команда **exec** возвращает список идентификаторов всех процессов конвейера. Стандартный вывод последней команды конвейера, если он не перенаправлен, выводится на стандартный вывод приложения. Стандартный вывод ошибок, если он не перенаправлен, осуществляется в стандартный вывод ошибок приложения.

Первое слово в каждой команде считается именем команды. В нем выполняются тильда-подстановки. Если получившийся при этом результат не содержит слешей, соответствующая команда ищется в каталогах, перечисленных в переменной окружения **PATH**. Если имя команды после подстановок содержит слеши, оно должно указывать на исполняемый файл, доступный из текущего каталога. Никакие другие подстановки в командах, например, принятые в shell подстановки «*****» и «**?**», не выполняются.

exit

Команда **exit** прекращает исполнение приложения.

Синтаксис

```
exit?returnCode?
```

Описание

Прекращает процесс и возвращает системе **returnCode** в качестве кода завершения. Если **returnCode** не определен, то команда использует значение по умолчанию **0**.

expr

Команда **expr** вычисляет значение выражения.

Синтаксис

```
expr arg?arg arg...?
```

Описание

Команда соединяет аргументы **arg** через пробел в одно Tcl-выражение, вычисляет и возвращает его значение. Допустимые в Tcl-выражениях операторы математических действий составляют подмножество операторов языка C, и имеют такое же значение и порядок выполнения, что и соответствующие операторы C. Почти всегда результатом вычисления является число: целое или с плавающей запятой. Например, результат выражения

```
expr 8.2 + 6
```

равен **14.2**.

Выражения в Tcl отличаются от выражений в C способом описания операндов. Кроме того, Tcl-выражения поддерживают нечисловые операнды и сравнение строк.

fblocked

Проверяет, что предыдущая операция ввода исчерпала всю информацию для ввода.

Синтаксис

```
fblocked channelId
```

Описание

Команда **fblocked** возвращает **1**, если последняя операция ввода по каналу **channelId** возвратила меньше информации, чем было запрошено, поскольку вся доступная информация уже исчерпана. Например, если команда **gets** вызвана, когда для

ввода доступны только три символа без символов конца строки, команда **gets** вернет пустую строку, последующий вызов команды **fblocked** вернет **1**.

fconfigure

Команда **fconfigure** — устанавливает и читает опции канала.

Синтаксис

```
fconfigure channelId
fconfigure channelId name
fconfigure channelId name value?name value...?
```

Описание

Команда **fconfigure** устанавливает и читает опции каналов. Аргумент **channelId** определяет конкретный канал, с которым работает команда. Если аргументы **name** и **value** отсутствуют, команда возвращает список, содержащий поочередно имена опций и их значения для канала. Если имя опции указано, а значение — нет, команда возвращает текущее значение указанной опции. Если в команде указаны одна или больше пар имен и значений, то команда устанавливает каждой из перечисленных опций указанное значение. В этом случае команда возвращает пустую строку.

Описанные ниже опции поддерживаются для всех типов каналов. Кроме того, каждый тип канала может иметь дополнительно собственные опции. Они приведены в описаниях команд создания каналов, например, в описании команды **socket**.

-blocking boolean

Опция **-blocking** определяет, вызывают ли команды ввода/вывода блокировку процесса. Величина этой опции должна быть правильным булевым выражением. Обычно каналы

открываются в блокирующем режиме. Если канал переведен в неблокирующий режим, это окажет влияние на выполнение команд **gets**, **read**, **puts**, **flush** и **close**. Чтобы работа в неблокирующем режиме выполнялась корректно, в приложении должен быть запущен обработчик событий (например, с помощью команды **vwait**).

-buffering newValue

Если аргумент **newValue** равен **full**, система ввода-вывода будет накапливать вывод в буфере, пока буфер не заполнится целиком. После этого буфер будет выдан в канал.

Если аргумент **newValue** равен **line**, система ввода-вывода будет выдавать буфер в канал каждый раз при поступлении символа конца строки.

Если аргумент **newValue** равен **none**, система ввода-вывода будет выводить каждый символ сразу после его поступления.

Значение по умолчанию **full** для всех каналов, кроме устройств типа терминалов. Для них значение по умолчанию **line**.

-buffersize newSize

Аргумент **newSize** должен быть целым числом. Его значение определяет размер буфера (в байтах), используемый для данного канала. Величина **newSize** должна быть в пределах от десяти до миллиона, что позволяет задавать величину буфера от десяти до миллиона байт.

-eofcharchar
-eofchar {inChar outChar}

Эти опции поддерживают структуру файлов DOS, в которой символ **Control-z** (**\x1a**) используется как символ конца файла. Если аргумент **char** не равен пустой строке, то этот символ, если он встречается во вводе, означает конец файла. При выводе информации символ конца файла выдается при

закрытии канала. Если аргумент **char** равен пустой строке, то специальный символ конца файла отсутствует. Для каналов ввода-вывода элементы списка **{inChar outChar}** определяют символы конца файла для ввода и вывода соответственно. Пользователь может указать в списке единственный символ, который будет использоваться и для ввода, и для вывода. Однако, при запросе команда возвратит текущие установки в виде списка из двух одинаковых элементов. Значения по умолчанию для символов конца файла — пустая строка всегда, кроме файлов Windows. В этом случае **inChar** равно **Control-z** (**\x1a**), **outChar** — пустой строке.

-translation mode
-translation{inMode outMode}

В Tcl-скриптах конец строки всегда представляется единственным символом новой строки (**\n**). Однако, в реальных файлах и устройствах конец строки может быть представлен разными символами или наборами символов в зависимости от используемой платформы или даже конкретного устройства. Например, на UNIX-платформах символ новой строки используется в файлах, в то время как для сетевых соединений используется последовательность «возврат каретки — новая строка». При вводе данных (например, при исполнении команд **gets** или **read**) система ввода-вывода Tcl сама автоматически преобразует внешнее представление конца строк во внутреннее представление (символ новой строки). При выводе (например, при команде **puts**) также происходит преобразование ко внешнему формату представления конца строки. Значение аргумента по умолчанию равно **auto**. При этом символы конца строки для большинства ситуаций правильно определяются автоматически. Однако, опция **-translation** позволяет при необходимости задать соответствующие символы в явном виде.

Аргумент **mode** задает представление конца строки для каналов, открытых только на чтение или только на запись. Список **{inMode outMode}** определяет представление конца

строки для каналов ввода-вывода. Пользователь может указать в списке единственный символ, который будет использоваться и для ввода, и для вывода. Однако, при запросе команда возвратит текущие установки в виде списка из двух одинаковых элементов.

Поддерживаются следующие значения опции:

- **auto**

При вводе в качестве конца строки могут использоваться символ возврата каретки (**cr**), символ новой строки (**lf**) или их последовательность (**crlf**). Причем разные строки могут заканчиваться по-разному. Все эти представления будут переведены в символ новой строки. При выводе используется разное представление для различных платформ и каналов. Для сетевых соединений на всех платформах используется **crlf**, для всех Unix-платформ — **lf**, для Macintosh — **cr**, а для всех Windows-платформ — **crlf**.

- **binary**

Никакого преобразования символов конца строки не производится. Это значение опции очень схоже со значением **lf**, однако, при значении **binary** пустая строка воспринимается как конец файла.

- **cr**

В качестве символа конца строки используется возврат каретки (**cr**). Соответственно, при вводе символ возврата каретки преобразуется в символ новой строки, а при выводе, наоборот, символ новой строки преобразуется в символ возврата каретки. Это значение опции используется, обычно, на Macintosh-платформах.

- **crlf**

В качестве символа конца строки используется последовательность «возврат каретки — новая строка» (**crlf**). Соответственно, при вводе последовательность «возврат каретки — новая строка» преобразуется в символ новой строки,

а при выводе, наоборот, символ новой строки преобразуется в последовательность «возврат каретки — новая строка». Это значение опции используется обычно на Windows-платформах и при сетевых соединениях.

- **lf**

В качестве символа конца строки используется символ новой строки (**lf**). Никакого преобразования символов конца строки при вводе и выводе не происходит. Это значение опции используется, обычно, на UNIX-платформах.

fcopy

Копирует данные из одного канала в другой.

Синтаксис

```
fcopy inchan outchan?-size size??-command callback?
```

Описание

Команда **fcopy** копирует данные из одного канала ввода-вывода, заданного идентификатором канала **inchan**, в другой канал ввода-вывода, заданный идентификатором канала **outchan**. Команда позволяет упростить буферизацию и избежать излишнего копирования в Tcl-системе ввода-вывода, а также избежать использования больших объемов памяти при копировании данных по таким медленным каналам, как сетевые соединения.

Команда **fcopy** передает данные из канала **inchan**, пока не будет достигнут конец файла или не будет передано **size** байтов. Если аргумент **-size** не задан, передается весь файл. Если опция **-command** не задана, команда блокирует процесс до завершения копирования и возвращает число переданных байтов.

При наличии аргумента **-command** команда **fcopy** работает в фоновом режиме. Она завершается немедленно, а команда **callback** вызывается позже, когда завершается процесс

копирования. Команда **callback** вызывается с одним или двумя дополнительными аргументами, которые указывают число переданных байтов. Если при исполнении фонового процесса произошла ошибка, второй аргумент — строка описания ошибки. При фоновом выполнении копирования каналы **inchan** и **outchan** не обязательно открывать в неблокирующем режиме, команда **fcopy** выполнит это автоматически. Однако при этом необходимо организовать обработку событий, например, с помощью команды **vwait** или используя Tk.

Не допускается выполнение других операций ввода-вывода с теми же каналами во время фонового копирования. Если один из каналов во время копирования будет закрыт, процесс копирования будет прерван и вызова команды **callback** не произойдет. Если будет закрыт канал ввода данных, то все полученные данные, хранящиеся в очереди, будут выданы в выходной канал.

Необходимо отметить, что канал **inchan** может стать открытым на чтение во время копирования. Все обработчики файловых событий во время фонового копирования должны быть выключены, чтобы они не создавали помех копированию. Любые попытки ввода-вывода с помощью обработчиков файловых событий будут завершены с ошибкой «канал занят».

Команда **fcopy** преобразует символы конца строк в соответствии со значениями опций **-translation** для соответствующих каналов. Преобразование означает, в частности, что число прочитанных и число переданных символов может отличаться. В синхронном режиме команда возвращает только число переданных в **outchan** канал символов. В фоновом режиме только это число подается на вход команды **callback**.

Примеры

Первый пример показывает, как в фоновом режиме получить число переданных байтов. Конечно, это показательный

пример, поскольку то же самое может быть сделано проще без использования фонового режима.

```
proc Cleanup {in out bytes {error {}}} {
global total
set total $bytes
close $in
close $out
if {[string length $error]!= 0} {
# error occurred during the copy
}
}
#### Открыть файл на чтение
set in [open $file1]
#### Открыть сетевое соединение
set out [socket $server $port]
#### Скопировать, по окончании копирования вызвать Cleanup
fcopy $in $out -command [list Cleanup $in $out]
#### Ожидать завершения копирования
vwait total
```

Второй пример показывает, как можно организовать копирование файла по фрагментам и проверять конец файла.

```
proc CopyMore {in out chunk bytes {error {}}} {
global total done
incr total $bytes
if {([string length $error]!= 0) || [eof $in] {
set done $total
close $in
close $out
} else {
fcopy $in $out -command [list CopyMore $in $out $chunk] \
```

```
-size $chunk
}
}
set in [open $file1]
set out [socket $server $port]
#### Установить размер фрагмента для копирования.
set chunk 1024
set total 0
fcopy $in $out -command [list CopyMore $in $out $chunk] -size
$chunk
vwait done
```

file

Команда для работы с файлами и их именами.

Синтаксис

```
file option name?arg arg...?
```

Описание

Эта команда осуществляет различные действия с файлами, их именами или свойствами. Аргумент **name** содержит имя файла. Если он начинается с символа «~», то перед выполнением команды выполняются «тильда»-подстановки. Опция команды указывает, какие действия необходимо выполнить с файлом. Ниже приведены возможные опции. В команде их имена могут быть сокращены до уровня, сохраняющего уникальность их имен.

file atimename

Возвращает десятичную строку, содержащую время последнего доступа к файлу **name**. Время представляется стандартным для POSIX образом в числе секунд от фиксированного начального момента (обычно, с 1 января

1970 г.). Если файл не существует или время последнего доступа не может быть получено, выдается сообщение об ошибке.

file attributes name
file attributes name?option?
file attributes name?option value option value...?

Эта подкоманда возвращает или устанавливает зависящие от платформы атрибуты файла. Первая форма возвращает список атрибутов и их значений, вторая возвращает значение указанного атрибута, а третья — позволяет установить значения одного или нескольких атрибутов. Возможные атрибуты перечислены ниже.

На UNIX-платформах:

-group

Возвращает или устанавливает имя группы. В команде группа может быть задана числовым идентификатором, но возвращается всегда имя группы.

-owner

Возвращает или устанавливает имя «хозяина» файла. В команде хозяин может быть задан числовым идентификатором, но возвращается всегда имя хозяина.

-permissions

Возвращает или устанавливает восьмеричный код, используемый командой операционной системы **chmod**. Символьное описание соответствующих атрибутов файла не поддерживается.

На Windows-платформах:

На Windows-платформах поддерживаются атрибуты: **-archive**, **-hidden**, **-longname** (атрибут не может быть установлен), **-readonly**, **-shortname** (атрибут не может быть установлен), **-system**.

На Macintosh-платформах:

На Macintosh-платформах поддерживаются атрибуты: **-creator**, **-hidden**, **-readonly**, **-type**.

file copy?-force??--? source target
file copy ?-force??--? source?source...? targetDir

Первая форма используется для того, чтобы скопировать файл или каталог **source** соответственно в файл или каталог **target**. Вторая форма используется, чтобы скопировать файл(ы) или каталог(и) внутрь существующего каталога **targetDir**. Если один из аргументов **source** есть имя каталога, то этот каталог копируется рекурсивно вместе со всем его содержимым. При копировании существующие файлы не перезаписываются, если только не указана опция **-force**. Попытки перезаписать непустой каталог, а также перезаписать каталог файлом или файл каталогом приводят к ошибке, даже если опция **-force** указана. Аргументы команды обрабатываются в порядке перечисления до первой ошибки. Отметка «**--**» означает конец опций. Следующий аргумент считается именем файла или каталога, даже если он начинается с символа «**-**».

file delete ?-force??--? pathname?pathname...?

Удаляет файлы или каталоги, заданные аргументами **pathname**. Непустые каталоги удаляются, только если задана опция **-force**. Попытка удалить несуществующий файл не рассматривается как ошибка. Попытка удалить файл, доступный только для чтения, приведет к удаления файла, даже если опция **-force** не задана. Аргументы команды обрабатываются в порядке перечисления до первой ошибки. Отметка «**--**» означает конец опций. Следующий аргумент считается именем файла или каталога, даже если он начинается с символа «**-**».

filedirname name

Возвращает имя, составленное из всех элементов **name**, кроме последнего. Если **name** — относительное имя файла и

состоит из единственного элемента, возвращает «**.**» («**:**» для Macintosh). Если имя указывает на корневой каталог, возвращается имя корневого каталога.

Например:

```
file dirname c:/
```

вернет **c:/**.

«Тильда»-подстановки выполняются, только если они необходимы для формирования правильного результата.

Например:

```
file dirname ~/src/foo.c
```

вернет **~/src**, тогда как

```
file dirname ~
```

вернет **/home** (или что-то подобное).

file executable name

Возвращает **1**, если файл **name** исполнимый, и **0** в противном случае.

file exists name

Возвращает **1**, если файл **name** существует, и пользователь имеет права на просмотр каталога, в котором лежит файл, и **0** в противном случае.

file extension name

Возвращает все символы в **name**, начиная с последней точки в последнем элементе. Если в последнем элементе нет точек, возвращается пустая строка.

file isdirectory name

Возвращает **1**, если **name** — имя каталога, и **0** в противном случае.

file isfile name

Возвращает **1**, если файл **name** — регулярный файл, и **0** в противном случае.

file join name?name...?

Соединяет аргументы **name** в одно имя с помощью разделителя, используемого на данной платформе. Если очередной аргумент **name** представляет собой относительное имя, он присоединяется к предыдущим, в противном случае предыдущие аргументы отбрасываются, и процесс формирования имени начинается заново с текущего аргумента.

Например:

```
file join a b /foo bar
```

вернет **/foo/bar**.

Аргументы **name** могут содержать разделитель, это не помешает получить правильный результат для используемой платформы («/» для Unix и Windows, «**:**» для Macintosh).

file lstat name varName

То же самое, что опция **stat**, описанная ниже, за исключением того, что используется команда ядра **lstat** вместо **stat**. Это означает, что если **name** есть имя связи, то команда вернет данные о связи, а не об исходном файле. Для платформ, не поддерживающих связи, команды полностью идентичны.

file mkdir dir?dir...?

Создает все перечисленные каталоги. Для каждого аргумента **dir** команда создает все несуществующие родительские каталоги и сам каталог **dir**. Если указан существующий каталог, ничего не происходит. При этом команда считается выполненной успешно. Попытка перезаписать существующий файл каталогом приведет к ошибке. Аргументы команды обрабатываются в порядке перечисления до первой ошибки.

file mtimename

Возвращает десятичную строку, содержащую время последнего изменения файла **name**. Время представляется стандартным для POSIX образом в числе секунд от фиксированного начального момента (обычно, с 1 января 1970 г.). Если файл не существует или время последнего изменения не может быть получено, выдается сообщение об ошибке.

file nativename name

Возвращает имя файла в виде, характерном для используемой платформы. Опция полезна для подготовки исполнения файла с помощью команды **exec** под Windows.

file owned name

Возвращает **1**, если файл **name** принадлежит пользователю, и **0** в противном случае.

file pathtype name

Возвращает одно из значений **absolute**, **relative**, **volumerelative**. Если **name** указывает на определенный файл в определенном томе, возвращается **absolute**. Если **name** указывает на имя файла относительно текущего рабочего каталога — возвращается **relative**. Если **name** указывает имя файла относительно текущего рабочего каталога в определенном томе или на определенный файл в текущем рабочем томе, возвращается **volumerelative**.

file readable name

Возвращает **1**, если файл name доступен для чтения пользователю, и **0** в противном случае.

file readlink name

Возвращает имя связи **name** (например, имя файла, на который указывает **name**). Если **name** не есть связь, или ее

невозможно прочитать, возвращает ошибку. На платформах, на которых связи не поддерживаются, опция не определена.

file rename ?-force??--? source target
file rename ?-force??--? source?source...? targetDir

Первая форма используется для того, чтобы переименовать файл или каталог **source** соответственно в файл или каталог **target** (и переместить их в соответствующий каталог, если это необходимо). Вторая форма используется, чтобы переместить файл(ы) или каталог(и) внутрь существующего каталога **targetDir**. Если один из аргументов **source** есть имя каталога, то этот каталог перемещается рекурсивно вместе со всем его содержимым. При перемещении существующие файлы не перезаписываются, если только не указана опция **-force**. Попытки перезаписать непустой каталог, а также перезаписать каталог файлом или файл каталогом приводят к ошибке, даже если опция **-force** указана. Аргументы команды обрабатываются в порядке перечисления до первой ошибки. Отметка «**-**» означает конец опций. Следующий аргумент считается именем файла или каталога, даже если он начинается с символа «**-**».

file rootname name

Возвращает все символы в **name** за исключением последней точки в последнем элементе. Если в последнем элементе нет точек, возвращается **name**.

file size name

Возвращает десятичную строку, содержащую размер файла в байтах. Если файл не существует или его размер не может быть получен, выдается ошибка.

file split name

Возвращает список элементов пути **name**. Первый элемент списка при этом имеет тот же тип пути, что и **name**. Все остальные элементы — относительные. Разделители удаляются,

если только они не необходимы для указания на относительный тип элементов. Например, под Unix

```
file split /foo/~bar/baz
```

вернет **/ foo./~bar baz**, чтобы гарантировать, что последующие команды не попытаются выполнить «тильда»-подстановку в третьем элементе.

file stat name varName

Исполняет вызов функции ядра **stat** и записывает возвращаемую информацию о **name** в элементы массива **varName**. Формируются следующие элементы массива: **atime**, **ctime**, **dev**, **gid**, **ino**, **mode**, **mtime**, **nlink**, **size**, **type**, **uid**. Значения всех элементов, кроме **type**, — десятичные строки. Описания их приведены в описании команды ядра **stat**. Элемент **type** содержит тип файла в том же виде, в каком он возвращается командой **file type**. Команда **file stat** возвращает пустую строку.

file tail name

Возвращает все символы в **name** после последнего разделителя каталогов. Если в **name** нет каталогов, возвращает **name**.

file type name

Возвращает строку, содержащую тип файла. Возможные значения **file**, **directory**, **characterSpecial**, **blockSpecial**, **fifo**, **link** или **socket**.

file volume

Возвращает список, содержащий абсолютные пути ко всем подмонтированным томам. На Macintosh-платформах это список всех подмонтированных дисков, локальных и сетевых. На Unix-платформах команда всегда возвращает «/», поскольку все файловые системы монтируются как локальные. На Windows-платформах команда возвращает список локальных дисков (например, **{a:/ c:/}**).

file writable name

Возвращает **1**, если файл **name** доступен для записи, и **0** в противном случае.

fileevent

Исполняет скрипт, когда канал открывается на чтение или запись.

Синтаксис

```
fileevent channelId readable?script?
fileevent channelId writable?script?
```

Описание

Эта команда используется для создания обработчиков файловых событий. Обработчик файловых событий связывает канал и скрипт таким образом, что скрипт исполняется, когда канал открывается на чтение или запись. Обработчики файловых событий используются, чтобы получение данных от другого процесса управлялось событиями. При этом получающий процесс, ожидая поступление данных, сможет продолжать взаимодействовать с пользователем. Если приложение выполняет команду **get** или **read** из блокирующего канала, оно не способно обслуживать другие события, поэтому оно кажется пользователю «замороженным». С использованием файловых событий процесс обратится к команде **get** или **read** только когда информация поступит в канал.

Аргумент **channelId** должен быть идентификатором открытого канала, который вернула предыдущая команда **open** или **socket**. Если в команде присутствует аргумент **script**, команда создает новый обработчик событий: скрипт **script** будет выполнен, когда канал **channelId** откроется на чтение или запись (в зависимости от второго аргумента команды). В такой форме команда возвращает пустую строку. Обработчики для обработки открытия файла на чтение или запись соответственно

независимы и могут создаваться и удаляться по одному, независимо один от другого. Однако для каждого из событий может быть только один обработчик, так что если команда **fileevent** выполняется, когда соответствующий обработчик (в текущем интерпретаторе) уже задан, новый скрипт заменит старый.

Если аргумент **script** не задан, команда **fileevent** возвратит скрипт, заданный для данного события для канала **channelId**, или пустую строку, если скрипт не задан. Обработчик событий удаляется автоматически при закрытии канала или удалении интерпретатора.

Канал считается открытым на чтение, если на соответствующем устройстве есть непрочитанные данные. Также канал считается открытым на чтение, если есть непрочитанные данные во входном буфере, кроме того случая, когда команда **get** не смогла найти в буфере законченную строку. Эта особенность позволяет читать файл построчно в неблокирующем режиме, используя обработчик событий. Канал также считается открытым на чтение, если достигнут конец соответствующего файла или на соответствующем устройстве сгенерирована ошибка. Поэтому скрипт должен уметь распознавать и корректно обрабатывать такие ситуации, чтобы не возникало зацикливаний, когда скрипт не может прочитать данные, завершается и тут же вызывается вновь.

Канал считается открытым на запись, если по крайней мере один байт данных может быть записан в соответствующий файл или передан на соответствующее устройство, или на устройстве (в файле) сгенерирована ошибка.

Событийно управляемый ввод-вывод лучше всего работает с каналами, переведенными в неблокирующий режим с помощью команды **fconfigure**. В блокирующем режиме команды **puts**, **get** или **read** могут заблокировать процесс, если они не могут быть выполнены сразу (например, при попытке прочитать больше данных, чем доступно в настоящий момент). При этом

никакой обработки событий не происходит. В неблокирующем режиме команды **puts**, **get** или **read** никогда не блокируют процесс.

Скрипт обработчика файловых событий выполняется на самом верхнем уровне вне контекста какой-либо процедуры в интерпретаторе, в котором обработчик событий был задан. Если при исполнении скрипта происходит ошибка, сообщение о ней выдается с помощью процедуры **bgerror**. Кроме того, при ошибке соответствующий обработчик событий удаляется. Это делается для того, чтобы избежать зацикливания из-за ошибок в обработчике.

flush

Команда организует немедленную выдачу выходных данных в канал.

Синтаксис

```
flush channelId
```

Описание

Команда направляет накопленные в выходном буфере данные в канал с идентификатором **channelId** (значение идентификатора возвращается командами открытия канала **open** или **socket**), который должен быть открыт для записи. Если канал находится в блокирующем режиме, то команда будет оставаться незавершенной до тех пор, пока все содержимое буфера не будет отправлено в канал. Если канал находится в неблокирующем режиме, то команда может завершиться до окончания отправки выходных данных в канал. Остающиеся данные будут передаваться в канал в фоновом режиме с такой скоростью, с какой назначенный каналу файл или устройство сможет принимать их.

for

Команда **for** организует цикл.

Синтаксис

```
for start test next body
```

Описание

Команда **for** является командой цикла. По структуре команда **for** похожа на аналогичную команду языка C. Здесь аргументы **start**, **next** и **body** должны быть командными строками Tcl, а **test** — строкой выражения. Сначала команда **for** запускает интерпретатор Tcl для выполнения **start**. Затем она вычисляет значение выражения **test**; если оно не равно нулю, то запускает Tcl-интерпретатор для выполнения **body**, затем **next**. Цикл повторяется до тех пор, пока **test** не станет равно **0**. Если при выполнении **body** будет выполнена команда **continue**, то последующие команды в **body** пропускаются и начинает выполняться **next**, затем **test** и т.д. Если при исполнении **body** или **next** встретится команда **break**, исполнение команды **for** немедленно прекращается. Команда **for** возвращает пустую строку.

Строку **test** почти всегда следует помещать в фигурные скобки. В противном случае подстановки переменных будут выполнены до выполнения команды. Из-за этого измененное в ходе цикла значение переменной может перестать передаваться в выражение, что может породить бесконечный цикл. Если же строка **test** заключена в фигурные скобки, подстановка значения переменных выполняется в каждом цикле. Для примера можно выполнить следующий скрипт со скобками и без скобок вокруг выражения **$x<10**:

```
for {set x 0} {$x<10} {incr x} {
puts "x is $x"
}
```

foreach

Команда цикла по элементам одного или нескольких списков.

Синтаксис

```
foreach varname list body
foreach varlist1 list1?varlist2 list2...? body
```

Описание

Команда организует выполнение цикла, в котором переменные цикла последовательно принимают все значения из списков значений. В простейшем случае имеется одна переменная цикла **varname** и один список значений **list** для присвоения переменной цикла. Аргумент **body** есть скрипт Tcl. Для каждого элемента списка **list**, по очереди с первого до последнего, **foreach** присваивает содержимое очередного элемента списка переменной **varname** и затем вызывает интерпретатор Tcl для исполнения **body**.

В общем случае в команде может быть указано несколько списков значений (например, **list1** и **list2**), и каждый из них может быть связан с одной переменной или со списком переменных цикла (например, **varlist1** и **varlist2**). Во время каждой итерации переменные каждого списка переменных принимают значения последовательных элементов соответствующего списка значений. Значения из списков значений используются последовательно от первого до последнего, и каждое значение используется только один раз. Общее число итераций выбирается таким, чтобы использовать все значения из всех списков значений. Если список значений не содержит достаточного числа значений для всех связанных с ним переменных цикла, вместо недостающих элементов используются пустые значения.

Внутри скрипта **body** можно использовать команды **break** и **continue**, аналогично команде **for**.

Команда **foreach** возвращает пустую строку.

Примеры

В цикле используются переменные цикла **i** и **j** для цикла по элементам одного списка:

```
set x {}
foreach {i j} {a b c d e f} {
lappend x $j $i
}
```

В результате величина **x** равна **b a d c f e**.

При вычислении цикла используются три итерации.

В цикле переменные цикла **i** и **j** используются для различных списков значений.

```
set x {}
foreach i {a b c} j {d e f g} {
lappend x $i $j
}
```

В результате величина **x** равна **a d b e c f {} g**.

При вычислении цикла используются четыре итерации.

Обе предыдущие формы скомбинированы в следующем цикле:

```
set x {}
foreach i {a b c} {j k} {d e f g} {
lappend x $i $j $k
}
```

В результате величина **x** равна **a d e b f g c {} {}**.

При вычислении цикла используются три итерации.

format

Команда форматирует строку в стиле процедуры **sprintf**.

Синтаксис

```
format formatString?arg arg...?
```

Описание

Данная команда создает и возвращает программе форматированную строку так же, как это делает процедура ANSI C **sprintf** (эта процедура используется в реализации команды). Подобно **sprintf**, **formatString** указывает с помощью спецификаторов преобразований, как именно сформатировать результат, а возможные дополнительные аргументы предназначены для подстановки в результат.

Команда немного отличается от **sprintf** в части отдельных спецификаторов и ключей.

Команда просматривает строку **formatString** слева направо. Все символы из строки непосредственно переносятся в результирующую строку, кроме символа «%» и следующих непосредственно за ним. Такая последовательность символов рассматривается как спецификатор преобразования. Этот спецификатор управляет преобразованием очередного аргумента **arg** в указанный формат, после чего тот добавляется к результирующей строке вместо соответствующего спецификатора. Если в строке **formatString** содержится несколько спецификаторов, каждый из них управляет преобразованием одного дополнительного аргумента. Число таких аргументов должно быть достаточным для всех спецификаторов в строке.

Каждый из спецификаторов преобразования может содержать до шести различных частей: указатель позиции, набор флагов, минимальная ширина поля, точность, преобразователь длины и тип преобразования. Любые из полей, кроме типа преобразования, могут отсутствовать.

Если за символом «%» следуют целое число и знак «**$**», как например в **%2$d**, то величина для преобразования берется не из следующего аргумента, а из аргумента, занимающего соответствующую позицию в списке (**1** соответствует первому аргументу **arg**). Если спецификатор преобразования требует нескольких аргументов (когда он содержит символ «*****»), то используются последовательные аргументы, начиная с указанного. Если один из спецификаторов содержит указание позиции аргумента, то и все остальные спецификаторы должны его содержать.

Второй раздел спецификатора может содержать в произвольном порядке любые флаги из перечисленных ниже.

—

Указывает, что соответствующий аргумент будет выровнен влево (числа обычно выравниваются вправо с добавлением лидирующих пробелов при необходимости).

+

Указывает, что числа всегда будут вставлены со знаком, даже если они положительные.

space

Указывает, что перед числом будет добавлен пробел, если первый символ не знак.

0

Указывает, что число будет выровнено с добавлением лидирующих нулей.

#

Указывает на использование альтернативной формы вывода. Для «o» и «O» преобразований гарантирует, что первой цифрой всегда будет **0**. Для «x» и «X» преобразований — что **0x** или **0X** соответственно будет добавлен в начало числа. Для «e»,

«E», «f», «g» и «G» — что в числе будет использована десятичная точка. Для «g» и «G» — что конечные нули не будут отброшены.

Третья часть спецификатора преобразования представляет собой число, задающее минимальную ширину поля для данного преобразования. Обычно она используется для формирования данных в таблицу. Если преобразуемое значение не содержит указанного числа символов, одно будет дополнено до необходимого размера. Обычно поле заполняется пробелами слева, однако, указанные выше флаги «**0**» и «**—**» позволяют заполнять его нулями слева или пробелами справа. Если минимальная ширина поля указана как «*****», а не как число, то в качестве числового значения используется значение следующего аргумента в команде.

Четвертая часть спецификатора определяет точность представления чисел. Она состоит из точки и последующего числа. Число имеет различный смысл при различных преобразованиях. Для «e», «E» и «f» преобразования оно определяет число цифр справа от десятичной точки. Для «g» и «G» — общее число чисел слева и справа от десятичной точки (однако, конечные нули будут обрезаться, если не указан флаг «#»). Для целочисленных преобразований оно определяет минимальное число символов (при необходимости будут добавляться лидирующие нули). Для «s» преобразований определяет максимальное число символов, которое будет выводиться. Если строка длиннее, конечные символы будут отброшены. Если точность указана как «*», а не как число, то в качестве числового значения используется значение следующего аргумента в команде.

Пятая часть спецификатора может принимать значения «h» или «l». Значение «h» означает, что все числовые значения предварительно обрезаются до 16 бит. Значение «l» означает, что никаких предварительных преобразований не производится.

Последняя часть спецификатора представляет собой букву, которая определяет тип преобразования. Допускаются следующие значения.

d

Преобразует целое число в десятичную строку со знаком.

u

Преобразует целое число в десятичную строку без знака.

i

Преобразует целое число в десятичную строку со знаком. Целое может быть десятичным, восьмеричным (с 0 вначале) или шестнадцатеричным (с 0x вначале).

o

Преобразует целое число в восьмеричную строку без знака.

x или X

Преобразует целое число в шестнадцатеричную строку без знака. Используются символы **0123456789abcdef** для **x** и **0123456789ABCDEF** для **X**.

c

Преобразует целое число в восьмибитный символ, который оно представляет.

s

Не преобразует, но просто вставляет строку

f

Преобразует число с плавающей точкой в десятичное со знаком в форме **xx.yyy**, где число символов после запятой определяется точностью.

e или E

Преобразует число с плавающей точкой в число в экспоненциальной форме **x.yyye±zz**, где число символов после запятой определяется точностью (по умолчанию 6). Для **E** в записи числа используется **E** вместо **e**.

g или G

Если порядок числа меньше **–4** или больше, чем точность, или равен точности, преобразует число с плавающей точкой как **%e** или **%E**. В противном случае преобразует как **%f**.

%

Просто подставляет символ %.

Для числовых преобразований аргумент должен быть целым числом или десятичным с плавающей точкой. Аргумент преобразуется в двоичное число, а затем преобразуется обратно в строку в соответствии с указанным типом преобразования.

gets

Команда читает строку из канала.

Синтаксис

```
gets channelId?varName?
```

Описание

Команда **gets** читает из канала **channelId** очередную строку символов. Если имя переменной **varName** не задано, тогда команда возвращает полученную строку за исключением символов конца строки. Если **varName** задано, тогда команда записывает полученную строку в переменную и возвращает количество символов в принятой строке.

Если при поиске конца строки был обнаружен конец файла, команда возвращает всю полученную информацию вплоть до конца файла.

Если канал находится в неблокирующем режиме и поступила неполная входная строка, то команда не использует поступившие данные и возвращает пустую строку.

Если указана переменная **varName** и возвращается пустая строка из-за конца файла или из-за неполноты полученной строки, команда возвращает **–1**.

Обратите внимание, что если аргумент **varName** не задан, конец файла и неполная строка приведут к тому же результату, что и строка, состоящая из символа конца строки. Команды **eof** и **fblocked** позволяют различить эти ситуации.

glob

Команда возвращает имена файлов, удовлетворяющих шаблону.

Синтаксис

```
glob ?switches? pattern?pattern …?
```

Описание

Команда **glob** выполняет поиск имен файлов подобно тому, как это делает оболочка csh, и возвращает список имен, удовлетворяющих шаблону **pattern**. Аргументы, начинающиеся со знака «**-**», являются управляющими ключами **switches**.

Возможные ключи:

-nocomplain

Позволяет вернуть пустой список без генерации ошибки. Если ключ не задан, то при пустом списке формируется ошибка.

--

Означает конец ключей. Аргумент после этого ключа считается шаблоном, даже если он начинается с «**-**».

Шаблоны могут включать следующие специальные символы:

- **?** – Удовлетворяет любому символу;
- ***** — Удовлетворяет любой последовательности из нуля или больше символов;
- **[chars]** – Удовлетворяет любому символу из **chars**. Если **chars** включает последовательность символов типа «**a-b**», то удовлетворяет всем символам от «**a**» до «**b**» (включительно).
- **\x** – Удовлетворяет символу «**x**».
- **{a,b,...}** – Удовлетворяет любой из строк «**a**», «**b**» и т.д.

Как и в **csh**, символ «**.**» в начале имени файла или сразу после «/» должен соответствовать явно или с помощью конструкции «{}».

Если первый символ образца «~», то он указывает на домашний каталог пользователя, чье имя указано после «~». Если сразу после «~» идет «/», то используется значение переменной окружения **HOME**.

Действие команды **glob** отличается от работы в **csh** в следующем:

- она не сортирует составленный ей список;
- она возвращает имена только существующих файлов (в csh проверку наличия файлов надо задавать отдельно, если только шаблон не содержит символов «**?**», «*****» или «**[]**»).

В отличие от других Tcl-команд команда **glob** может работать с именами файлов только в нотации, поддерживаемой на той платформе, на которой она исполняется. Кроме того, на Windows-платформах специальные символы не допустимы в сетевых именах.

global

Команда для объявления глобальных переменных.

Синтаксис

```
global varname?varname …?
```

Описание

Данная команда выполняется только при выполнении процедуры, а в остальных случаях игнорируется. Команда объявляет переменные **varname** глобальными. Глобальные переменные — это переменные глобальной области имен.

Только в течение работы данной процедуры и только при выполнении в данной процедуре любая ссылка на значение любого из аргументов **varname** будет указывать на одноименную глобальную переменную.

history

Команда работает со списком выполнявшихся команд.

Синтаксис

```
history?option??arg arg …?
```

Описание

Команда **history** выполняет действия по отношению к недавно выполненным командам, занесенным в журнал. Каждая из этих зарегистрированных команд обозначается термином «событие». Ссылаться на события в команде **history** можно одним из следующих способов:

- **Число**. Если положительное — ссылается на событие с этим номером (все события нумеруются начиная с 1). Если число отрицательное, то оно указывает номер события относительно текущего (**-1** — предыдущее, **-2** — перед предыдущим и т.д.). Событие **0** ссылается на текущее событие.

- **Строка**. Ссылается на наиболее позднее событие, которое удовлетворяет строке. Событие удовлетворяет строке, если оно начинается со строки, или в соответствии с правилами команды **string match**.

Команда **history** может принимать одну из следующих форм.

history

То же самое, что команда **history info**, описанная ниже.

history addcommand?exec?

Добавляет аргумент **command** в журнал как новое событие. Если присутствует аргумент **exec** (или произвольное сокращение), то команда **command** выполняется и возвращается ее результат. В противном случае возвращается пустая строка.

history change newValue?event?

Заменяет описание события **event** на **newValue**. Аргумент **event** определяет событие, описание которого будет заменено. По умолчанию — текущее событие (даже не предыдущее!). Эта форма команды предназначается для использования в тех случаях, когда переформировывается журнал событий и позволяет заменить текущее событие (переформирование журнала) на необходимое. Команда возвращает пустую строку.

history clear

Удаляет журнал событий. Количество запоминаемых событий сохраняется. Нумерация событий начинается сначала.

history event?event?

Возвращает описания события **event**. Значение по умолчанию **-1**.

history info?count?

Возвращает в удобном для чтения виде список, состоящий из номеров и описаний событий (кроме текущего). Если аргумент **count** задан, то только **count** последних событий возвращаются.

history keep?count?

Команда изменяет размер журнала на **count** событий. Исходно в журнале сохраняются 20 последних событий. Если аргумент **count** не указан, команда возвращает текущее значение размера журнала.

history nextid

Возвращает номер следующего события, которое будет записано в журнал. Полезно, например, для вывода номера события в приглашении командной строки.

history redo?event?

Повторно выполняет команду, указанную с помощью аргумента **event**. Значение аргумента по умолчанию **-1**. Эта команда вызывает переформирование журнала.

Переформирование журнала до версии 8.0 имело весьма сложный механизм. Новый механизм несколько сокращен за счет старых опций **substitute** и **words**. (Взамен добавлена новая опция **clear**).

Опция **redo** позволяет переформировывать журнал значительно проще. При ее выполнении последнее событие изменяется таким образом, что удаляется «служебная» команда **history**, которая реально выполнялась, а вместо нее записывается та команда, которая необходима.

Если вы хотите повторить прежнюю команду, не модифицируя журнал, выполните сначала команду **history event**, чтобы извлечь описание команды, а затем **history add**, чтобы выполнить ее.

if

Команда **if** проверяет соблюдение условия в ходе выполнения скрипта.

Синтаксис

```
if  expr1?then?  body1elseifexpr2?then?  body2elseif  …?else??bodyN?
```

Описание

Команда вычисляет значение выражения **expr1** (точно так, как это делает команда **expr**). Это и все остальные выражения **expr** должны быть булева типа (то есть это должна быть числовая величина, причем **0** соответствует **false**, а все остальные значения – **true**, либо строка со значениями **true** или **yes** для **true** и **false** или **no** для **false**). Если выражение равно **true**, то скрипт **body1** передается на выполнение интерпретатору Tcl. Если нет, то вычисляется значение выражения **expr2**, и если оно равно **true**, то исполняется **body2**, и так далее. Если ни одно из выражений не равно **true**, тогда выполняется **bodyN**.

Слова **then** и **else** необязательны и используются только для простоты понимания команды. Аргумент **bodyN** также может отсутствовать, если отсутствует **else**.

Команда возвращает значение выполненного скрипта или пустую строку, если ни одно из выражений не было равно **true**, и **bodyN** отсутствовал.

incr

Команда увеличивает значение переменной на заданную величину.

Синтаксис

```
incr  varName?increment?
```


Описание

Команда **incr** увеличивает значение переменной **varName** на величину **increment**. И значение переменной, и **increment** должны быть целыми числами. По умолчанию **increment** равно **1**. Новое значение переменной сохраняется в виде десятичной строки и одновременно возвращается командой.

info

Команда сообщает сведения о состоянии интерпретатора Tcl.

Синтаксис

```
info  option?arg  arg...?
```

Описание

Эта команда обеспечивает доступ к внутренней информации Tcl-интерпретатора. Ниже перечислены поддерживаемые опции (имена которых могут быть сокращены).

info argsprocname

Возвращает список имен аргументов процедуры **procname** в том порядке, в котором они определены при описании процедуры. Аргумент **procname** должен содержать имя Tcl-процедуры.

info body procname

Возвращает тело процедуры **procname**. Аргумент **procname** должен содержать имя Tcl-процедуры.

info cmdcount

Возвращает полное число команд, введенных в данный интерпретатор.

info commands?pattern?

Если аргумент **pattern** не задан, возвращает полный список команд в текущем пространстве имен, включая как встроенные команды, написанные на C, так и процедуры, заданные с помощью команды **proc**. Если аргумент **pattern** задан, возвращается список только тех имен, которые удовлетворяют шаблону **pattern**. Правила использования шаблонов такие же, как в команде **string match**. Шаблон может быть полным именем, например **Foo::print***. То есть он может задавать определенное пространство имен и шаблон в нем. В этом случае каждая команда в возвращаемом списке будет представлена полным именем с указанием пространства имен.

info complete command

Возвращает **1**, если команда **command** есть завершенная Tcl-команда, то есть не содержит «незакрытых» кавычек, квадратных или фигурных скобок и имен массивов. В противном случае возвращается **0**. Эта команда обычно используется при построчном вводе команд пользователем для того, чтобы позволить ему вводить команды из нескольких строк. Для этого, если введенный скрипт не представляет собой законченную команду, его исполнение откладывается до завершения следующей строки.

info default procname arg varname

Аргумент **procname** должен быть именем Tcl-процедуры, а аргумент **arg** — именем одного из аргументов этой процедуры. Если указанный аргумент не имеет значения по умолчанию, команда возвращает **0**. В противном случае команда возвращает **1** и помещает значение по умолчанию в переменную **varname**.

info exists varName

Возвращает **1**, если переменная **varName** существует в текущем контексте как локальная или как глобальная переменная. В противном случае возвращает **0**.

info globals?pattern?

Если аргумент **pattern** не задан, возвращает список имен определенных в данный момент глобальных переменных (переменных, определенных в глобальном пространстве имен). Если шаблон задан, возвращаются только имена, удовлетворяющие шаблону. Правила использования шаблонов такие же, как в команде **string match**.

info hostname

Возвращает имя компьютера, на котором выполняется этот вызов.

info level ?number?

Если аргумент **number** не задан, возвращает уровень стека выполняемой процедуры, или **0**, если команда выполняется на верхнем уровне. Если аргумент **number** указан, команда возвращает список, состоящий из имени и аргументов процедуры, находящейся в стеке вызовов на соответствующем месте. Если **number** положительное число, оно указывает номер уровня в стеке (**1** — самая верхняя вызванная процедура, **2** — процедура, вызванная из процедуры **1**, и так далее), если же **number** отрицательное, оно указывает уровень относительно уровня выполняемой процедуры (**0** — выполняемая процедура, **-1** — процедура, из которой вызвана исполняемая, и так далее).

info library

Возвращает имя каталога, в котором хранятся стандартные Tcl-скрипты. Обычно совпадает со значением переменной **tcl_library** и может быть изменено с помощью переопределения этой переменной.

info loaded?interp?

Возвращает список библиотек (package), загруженных в интерпретатор с помощью команды **load**. Каждый элемент списка представляет собой подсписок из двух элементов: имени

файла и имени библиотеки. Для статически загруженных библиотек имя файла отсутствует. Если имя интерпретатора **interp** отсутствует, возвращается список всех библиотек, загруженных во все интерпретаторы. Чтобы получить список библиотек, загруженных в текущий интерпретатор, используйте пустую строку в качестве аргумента **interp**.

info locals?pattern?

Если образец не задан, возвращает имена всех определенных в текущий момент локальных переменных, включая аргументы процедуры, если они есть. Переменные, заданные с помощью команд **global** и **upvar**, не возвращаются. Если шаблон задан, возвращаются только имена, удовлетворяющие шаблону. Правила использования шаблонов такие же, как в команде **string match**.

info nameofexecutable

Возвращает полное имя бинарного файла, с помощью которого приложение было запущено. Если Tcl не может определить файл, возвращается пустая строка.

info patchlevel

Возвращает значение глобальной переменной **tcl_patchLevel**.

info procs?pattern?

Если аргумент **pattern** не задан, возвращает полный список Tcl-процедур в текущем пространстве имен. Если аргумент **pattern** задан, возвращается список только тех имен, которые удовлетворяют шаблону **pattern**. Правила использования шаблонов такие же, как в команде **string match**.

info script

Если в данный момент обрабатывается Tcl-скрипт (например, вызванный с помощью команды **source**), то команда

возвращает имя файла, содержащего самый внутренний обрабатываемый скрипт. В противном случае возвращает пустую строку.

info sharedlibextension

Возвращает расширение, используемое на текущей платформе для файлов разделяемых библиотек (например, **.so** для Solaris). Если разделяемые библиотеки на текущей платформе не поддерживаются, возвращает пустую строку.

info tclversion

Возвращает значение глобальной переменной **tcl_version**.

info vars?pattern?

Если аргумент **pattern** не задан, возвращает список имен всех видимых в текущий момент переменных, включая локальные и видимые глобальные. Если аргумент **pattern** задан, возвращается список только тех имен, которые удовлетворяют шаблону **pattern**. Правила использования шаблонов такие же, как в команде **string match**. Шаблон может быть полным именем, например **Foo::option***. То есть он может задавать определенное пространство имен и шаблон в нем. В этом случае каждая команда в возвращаемом списке будет представлена полным именем с указанием пространства имен.

interp

Команда создает и управляет Tcl-интерпретаторами.

Синтаксис

```
interp option?arg arg...?
```

Описание

Эта команда позволяет создавать один или несколько новых Tcl-интерпретаторов, которые сосуществуют в одном

приложении с создавшим их интерпретатором. Создавший интерпретатор называется мастер-интерпретатором, а созданные интерпретаторы называются подчиненными (slave) интерпретаторами. Мастер-интерпретатор может создавать произвольное число подчиненных интерпретаторов, а каждый из подчиненных может в свою очередь создавать подчиненные интерпретаторы, для которых он сам является мастер-интерпретатором. В результате в приложении может создаваться иерархия интерпретаторов.

Каждый интерпретатор независим от остальных. Он имеет собственное пространство имен для команд, процедур и глобальных переменных. Мастер-интерпретатор может создавать связи между подчиненными интерпретаторами и собой, используя механизм алиасов. Алиас — это команда в подчиненном интерпретаторе, которая, при ее вызове, вызывает другую команду в мастер-интерпретаторе или в другом подчиненном интерпретаторе. Кроме механизма алиасов, связь между интерпретаторами поддерживается только через переменные окружения. Массив обычно является общим для всех интерпретаторов в приложении. Необходимо заметить, что идентификаторы каналов (например, идентификатор, возвращаемый командой **open**) больше не разделяются между интерпретаторами, как это было в предыдущих версиях Tcl. Чтобы обеспечить совместный доступ к каналам, необходимо использовать явные команды для передачи идентификаторов каналов из интерпретатора в интерпретатор.

Команда **interp** позволяет также создавать надежные интерпретаторы. Надежный интерпретатор — это интерпретатор с существенно урезанной функциональностью, поэтому он может исполнять ненадежные скрипты без риска нарушить работу вызывающего их приложения. Например, из безопасных интерпретаторов недоступны команды создания каналов и подпроцессов. Опасная функциональность не удалена из безопасных интерпретаторов, но скрыта таким образом, что только надежные интерпретаторы могут получить к ней доступ.

Механизм алиасов может быть использован для безопасного взаимодействия между подчиненным интерпретатором и его мастер-интерпретатором.

Полное имя интерпретатора представляет собой список, содержащий имена его предков в иерархии интерпретаторов и заканчивающийся именем интерпретатора в его непосредственном предке. Имена интерпретаторов в списке — это их относительные имена в их непосредственных мастер-интерпретаторах. Например, если **a** есть подчиненный интерпретатор текущего интерпретатора и, в свою очередь, имеет подчиненный интерпретатор **a1**, а тот, в свою очередь, имеет подчиненный интерпретатор **a11**, то полное имя **a11** в **a** есть список **{a1 a11}**.

В качестве аргумента команды **interp** используется полное имя интерпретатора. Интерпретатор, в котором исполняется команда, всегда обозначается как **{}** (пустая строка). Обратите внимание, что в подчиненном интерпретаторе невозможно сослаться на мастер-интерпретатор кроме как через алиасы. Также нет никакого имени, под которому можно было бы сослаться на мастер-интерпретатор, первым созданный в приложении. Оба ограничения вызваны соображениями безопасности.

Команда **interp** используется для создания, удаления и выполнения команд в подчиненном интерпретаторе, а также для разделения или передачи каналов между интерпретаторами. Она может иметь одну из перечисленных ниже форм в зависимости от значения аргумента **option**.

interp alias srcPath srcCmd

Возвращает список, состоящий из исходной команды и аргументов, связанных с алиасом **srcCmd** в интерпретаторе **srcPath** (возвращаются значения, использовавшиеся при создании алиаса, так как имя команды могло быть изменено с помощью команды **rename**).

interp alias srcPath srcCmd {}

Удаляет алиас **srcCmd** в подчиненном интерпретаторе **srcPath**. Имя **srcCmd** — это имя, под которым алиас был создан. Если созданная команда была переименована, то будет удалена переименованная команда.

interp alias srcPath srcCmd targetPath targetCmd?arg arg...?

Эта команда создает алиас между двумя подчиненными интерпретаторами (для создания алиаса между подчиненным интерпретатором и мастер-интерпретатором используется команда **slave alias**). Оба интерпретатора **srcPath** и **targetPath** должны быть в иерархии интерпретаторов ниже того интерпретатора, в котором выполняется команда. Аргументы **srcPath** и **srcCmd** задают интерпретатор, в котором будет создан алиас и его имя. Аргумент **srcPath** должен быть Tcl-списком, задающим имя существующего интерпретатора. Например, «**a b**» определяет интерпретатор **b**, который является подчиненным интерпретатором интерпретатора **a**, который в свою очередь является подчиненным интерпретатором текущего интерпретатора. Пустой список соответствует текущему интерпретатору (в котором исполняется команда). Аргумент **srcCmd** определяет имя новой команды-алиаса, которая будет создана в интерпретаторе **srcPath**. Аргументы **targetPath** и **targetCmd** определяют целевой интерпретатор и команду, а аргументы **arg**, если они есть, определяют дополнительные аргументы для команды **targetCmd**, которые будут вставлены перед аргументами, заданным при вызове **srcCmd**. Команда **targetCmd** может как существовать, так и не существовать в момент создания алиаса. В последнем случае она не создается командой **interp alias**.

Алиас позволяет использовать команду **targetCmd** в интерпретаторе **targetPath** каждый раз, когда вызывается команда-алиас **srcCmd** в интерпретаторе **srcPath**.

interp aliases?path?

Эта команда возвращает список имен всех команд-алиасов, определенных в интерпретаторе **path**.

interp create?-safe??--??path?

Создает подчиненный интерпретатор с именем **path** и новую команду для работы с этим интерпретатором, называемую также подчиненной (slave) командой. Имя подчиненной команды совпадает с последним элементом списка **path**. Новый подчиненный интерпретатор и подчиненная команда создаются в интерпретаторе, имя которого состоит из всех элементов списка **path**, кроме последнего. Например, если аргумент **path** равен **a b c**, то в результате в интерпретаторе **a b** будет создан Если аргумент **path** отсутствует, Tcl создает уникальное имя в форме **interpx**, где **x** — целое число, и использует его для подчиненного интерпретатора и подчиненной команды. Если в команде указана опция **-safe** или если мастер-интерпретатор сам является безопасным интерпретатором, новый подчиненный интерпретатор будет безопасным, то есть с ограниченной функциональностью. В противном случае новый интерпретатор будет включать полный набор встроенных Tcl-команд и переменных. Аргумент — используется для того, чтобы обозначить конец опций. Следующий аргумент будет использоваться как имя интерпретатора, даже если он равен **-safe**.

interp delete?path...?

Удаляет ноль или больше интерпретаторов с именем **path**. Для каждого удаляемого интерпретатора удаляются также его подчиненные интерпретаторы. Если для одного из аргументов **path** интерпретатора с таким именем не существует, команда генерирует ошибку.

interp eval path arg?arg...?

Команда объединяет все аргументы так же, как команда **concat**, а затем исполняет сформированный скрипт в подчиненном интерпретаторе, заданном аргументом **path**. Результат выполнения (включая информацию об ошибках в переменных **errorInfo** и **errorCode**, если произошла ошибка) возвращается в вызывающий интерпретатор.

interp exists path

Возвращает **1**, если подчиненный интерпретатор с именем **path** существует в его мастер-интерпретаторе. В противном случае возвращает **0**. Если аргумент **path** представляет относительное имя, то он ищется в том интерпретаторе, в котором выполняется команда.

interp expose path hiddenName?exposedCmdName?

Разрешает использование в интерпретаторе **path** скрытой команды **hiddenName** под новым именем **exposedCmdName** (в настоящее время поддерживаются только имена в глобальном пространстве имен, не содержащие «**::**»). Если обычная (не скрытая) команда **exposedCmdName** уже существует, генерируется сообщение об ошибке.

interp hidepath exposedCmdName?hiddenCmdName?

Запрещает использование в интерпретаторе **path** обычной команды **exposedCmdName** и переименовывает ее в скрытую команду **hiddenCmdName** (или в скрытую команду под старым именем, если новое не было задано). Если скрытая команда с заданным именем уже существует, команда возвращает ошибку. В настоящее время **exposedCmdName** и **hiddenCmdName** не могут содержать «**::**». Команды, которые должны быть скрыты с помощью **interp hide**, ищутся только в глобальном пространстве имен, даже если текущее пространство имен не глобальное.

interp hidden path

Возвращает список скрытых команд интерпретатора **path**.

interp invokehidden path?-global? hiddenCmdName?arg...?

Вызывает в интерпретаторе **path** скрытую команду **hiddenCmdName** с перечисленными аргументами. Никаких подстановок или вычислений в аргументах не производится. Если указана опция **-global**, скрытая команда выполняется на глобальном уровне в целевом интерпретаторе. В противном случае она выполняется в текущем контексте и может использовать значения локальных переменных и переменных из вышестоящих стеков.

interp issafe?path?

Возвращает **1**, если интерпретатор **path** безопасный, и **0** в противном случае.

interp marktrusted path

Отмечает интерпретатор **path** как надежный. Не раскрывает скрытые команды. Команда **interp marktrusted** может выполняться только из надежного интерпретатора. Если интерпретатор **path** уже надежный, команда не оказывает никакого воздействия.

interp share srcPath channelId destPath

Позволяет разделить канал ввода-вывода **channelId** между интерпретаторами **srcPath** и **destPath**. Оба интерпретатора после этого будут иметь одинаковые права на канал. Каналы ввода-вывода, доступные в интерпретаторе, автоматически закрываются, когда удаляется интерпретатор.

interp slaves?path?

Возвращает список подчиненных интерпретаторов для интерпретатора **path**. Если аргумент **path** отсутствует, возвращает

список подчиненных интерпретаторов для интерпретатора, в котором выполняется команда.

interp target path alias

Возвращает список, описывающий целевой интерпретатор (интерпретатор, в котором выполняется реальная команда при вызове команды-алиаса) для алиаса **alias**. Алиас задается именем интерпретатора **path** и команды-алиаса **alias** как в команде **interp alias** выше. Имя целевого интерпретатора возвращается как имя интерпретатора относительно имени интерпретатора, в котором выполняется команда. Если это текущий интерпретатор, то возвращается пустой список. Если этот интерпретатор не является потомком интерпретатора, в котором выполняется команда, генерируется ошибка. Реальная команда не обязана быть определена в момент выполнения данной команды.

interp transfer srcPath channelId destPath

Делает канал ввода-вывода **channelId** доступным в интерпретаторе **destPath** и недоступным в интерпретаторе **srcPath**.

join

Команда соединяет элементы списка в одну строку.

Синтаксис

```
join list ?joinString?
```

Описание

Команда возвращает строковое значение, состоящее из всех элементов списка **list**, отделенных друг от друга символом **joinString**. Список **list** должен быть Tcl-списком, **joinString** по умолчанию есть пробел.

lappend

Команда дополняет переменную элементами списка.

Синтаксис

```
lappend varName?value value value …?
```

Описание

Команда добавляет в список **varName** каждый из аргументов **value** как новый элемент, отделенный пробелом. Если **varName** не существует, то он будет создан с элементами, заданными **value**. Команда **lappend** подобна команде **append**, за исключением того, что аргументы добавляются в качестве элементов списка, а не просто текста. С помощью этой команды можно эффективно создавать большие списки. Например, для больших списков команда:

```
lappend a $b
```

намного эффективнее, чем

```
set a [concat $a [list $b]]
```

library

Команды стандартной библиотеки процедур Tcl.

Синтаксис

```
auto_execok cmd
auto_loadcmd
auto_mkindex dir pattern pattern …
auto_reset
parray arrayName
tcl_endOfWordstr start
tcl_startOfNextWord str start
tcl_startOfPreviousWord str start
tcl_wordBreakAfter str start
tcl_wordBreakBefore str start
```

Описание

Tcl содержит библиотеку Tcl-процедур общего назначения.

Местонахождение библиотеки Tcl можно получить с помощью команды **info library**. Обычно, помимо этой библиотеки, приложения имеют собственные библиотеки служебных процедур. Местонахождение процедур приложения обычно содержит глобальная переменная **$app_library**, где **app** — имя приложения. Например, для Tk это переменная **$tk_library**.

Для того, чтобы использовать процедуры из Tcl-библиотеки, приложению необходимо прочитать файл **init.tcl** библиотеки, например, командой:

```
source [file join [info library] init.tcl]
```

Если в приложении процедура **Tcl_AppInit** вызывает библиотечную процедуру **Tcl_Init**, то такая команда выполняется автоматически. Код в файле **init.tcl** определит процедуру **unknown** и позволит остальным процедурам загружаться по требованию при помощи механизма автозагрузки.

Библиотека Tcl предоставляет пользователю следующие процедуры:

auto_execok cmd

Эта команда просматривает каталоги в текущем пути поиска (заданном переменной окружения **PATH**) и проверяет, есть ли в каталогах исполняемый файл по имени **cmd**. Если файл присутствует, то команда возвращает **1**, если нет — то **0**. Команда запоминает сведения о предыдущем поиске в массиве **auto_execs**, это позволяет обходиться без поиска в каталогах при последующих вызовах этой же команды. Для удаления из памяти этой информации можно использовать команду **auto_reset**.

auto_load cmd

Эта команда загружает определение для Tcl-команды **cmd**. Для этого она просматривает путь автозагрузки, являющийся списком из одного или более каталогов. Он содержится в глобальной переменной **auto_path**, если она существует. В противном случае используется переменная окружения **TCLLIBPATH**. Если же и она не существует, список автозагрузки состоит из каталога, в котором находится Tcl-библиотека.

В каждом из каталогов, входящих в путь автозагрузки, должен находиться файл **tclIndex**, список команд, определенных в этом каталоге, и скрипты для загрузки каждой команды. Файл **tclIndex** должен быть создан с помощью команды **auto_mkindex**.

Если команда была успешно загружена, то **auto_load** возвращает **1**. Команда возвращает **0**, если нужная команда не была найдена в списках команд или указанная команда загрузки не позволила создать команду (например, если файл **tclIndex** устарел). Если при выполнении указанного загрузочного скрипта возникла ошибка, команда возвращает ошибку.

Команда **auto_load** читает индексные файлы только один раз и сохраняет полученную информацию в массиве **auto_index**. При последующих обращениях к команде **auto_load** сначала проверяется массив и только если информация о процедуре не найдена, приступает к просмотру индексных файлов. Эта информация может быть удалена с помощью команды **auto_reset**. После команды **auto_reset** следующий вызов **auto_load** к повторному чтению индексных файлов.

auto_mkindexdir pattern pattern...

Команда создает индекс для использования его командой **auto_load**. Для этого команда просматривает каталог **dir** в поисках файлов с именами, удовлетворяющими аргументам **pattern** (сравнение выполняется командой **glob**), и создает индекс всех командных процедур Tcl, определенных в обнаруженных

файлах, и сохраняет индекс в файле **tclIndex** в этой **dir**. Если не задано ни одного шаблона **pattern**, то по умолчанию принимается ***.tcl**. Например, команда

```
auto_mkindex foo *.tcl
```

просматривает все **tcl**-файлы в каталоге **foo** и создает новый индексный файл **foo/tclIndex**.

Команда **auto_mkindex** просматривает Tcl-скрипты очень простым способом: если в очередной строке, начиная с первого символа, написано слово **proc**, считается, что это определение процедуры, а следующее слово есть имя процедуры. Процедуры, определение которых не подходит под описанное (например, если перед словом **proc** стоят пробелы), не попадают в индексный файл.

auto_reset

Команда удаляет всю информацию, накопленную командами **auto_execok** и **auto_load**. При следующем обращении к этой информации она будет считана с диска заново. Эта команда также удаляет все процедуры, перечисленные в массиве **auto_index**, так что при следующем обращении к ним будут загружены новые копии.

parrayarrayName

Команда выдает на стандартный выход имена и значения элементов массива **arrayName**. Массив должен быть доступен в контексте вызова. Он может быть как локальным, так и глобальным.

tcl_endOfWord str start

Возвращает индекс первого конца слова после указанного индекса **start** в строке **str**. Первым концом слова считается первый символ, не принадлежащий слову, следующий за первым после начальной точки символом слова. Возвращает **-1**, если после начальной точки больше нет концов слова.

tcl_startOfNextWordstr start

Возвращает индекс первого начала слова после указанного индекса **start** в строке **str**. Первым началом слова считается первый символ слова, следующий за символом, не принадлежащим слову. Возвращает **-1**, если после начальной точки больше нет начала слова.

tcl_startOfPreviousWordstr start

Возвращает индекс первого начала слова до указанного индекса **start** в строке **str**. Возвращает **-1**, если после начальной точки больше нет начала слова.

tcl_wordBreakAfterstr start

Возвращает индекс первой границы слова после указанного индекса **start** в строке **str**. Возвращает **-1**, если в указанной строке после начальной точки больше нет границ слова. Возвращаемый индекс относится ко второму символу пары, образующей границу.

tcl_wordBreakBeforestr start

Возвращает индекс первой границы слова до указанного индекса **start** в строке **str**. Возвращает **-1**, если в указанной строке до начальной точки больше нет границ слова. Возвращаемый индекс относится ко второму символу пары, образующей границу.

Процедуры Tcl используют или определяют следующие глобальные переменные.

auto_execs — используется командой **auto_execok** для записи информации о том, существуют ли конкретная команда в виде исполняемого файла.

auto_index — используется **auto_load** для сохранения индексной информации, считанной с диска.

auto_noexec — если переменная задана с любым значением, то команда **unknown** не будет пытаться автоматически исполнить какую-либо команду.

auto_noload — если переменная задана с любым значением, то команда **unknown** не будет пытаться автоматически загрузить какую-либо команду.

auto_path — если переменная задана, то она должна содержать Tcl-список с каталогами для просмотра при операциях автозагрузки.

env(TCL_LIBRARY) — если переменная задана, то она указывает местоположение каталога с библиотечными скриптами (команда **info library** возвращает значение этой переменной). Если переменная не определена, то используется значение по умолчанию.

env(TCLLIBPATH) — если переменная задана, то она должна содержать действующий Tcl список каталогов для поиска при операциях автозагрузки. Эта переменная используется только тогда, когда не определена переменная **auto_path**.

tcl_nonwordchars — переменная содержит регулярное выражение, используемое такими процедурами, как **tcl_endOfWord** для определения, является ли символ частью слова или нет. Если образец соответствует символу, то символ считается не принадлежащим к слову. В Unix такими символами являются все символы, кроме цифр, букв и символа подчеркивания.

tcl_wordchars — переменная содержит регулярное выражение, используемое такими процедурами, как **tcl_endOfWord** для определения, является ли символ частью слова или нет. Если образец соответствует символу, то символ считается частью слова. В Unix слова состоят из цифр, букв и символа подчеркивания.

unknown_active — эта переменная служит флагом для индикации активности команды **unknown**: команда сама устанавливает ее. Переменная используется для выявления ошибок, при которых unknown бесконечно рекурсивно обращается к себе. Перед окончанием работы **unknown** переменная сбрасывается.

lindex

Команда извлекает элемент списка.

Синтаксис

```
lindex list index
```

Описание

Команда **lindex** считает **list** Tcl списком и возвращает index-ный элемент этого списка. Для **index** значение **0** соответствует первому элементу списка, а значение **end** — последнему. При выполнении команды соблюдаются общие правила интерпретатора Tcl относительно фигурных скобок, двойных кавычек и обратного слеша, хотя подстановки переменных и команд не происходят.

Если **index** отрицательное число или больше или равно числу элементов, команда возвращает пустое значение. Если значение аргумента **index** равно **end**, команда возвращает последний элемент списка.

linsert

Команда служит для вставки элементов в список.

Синтаксис

```
linsert list index element?element element …?
```

Описание

Данная команда создает из **list** новый список при помощи вставки аргументов **element** непосредственно перед index-ным элементом списка **list**. Каждый из аргументов **element** станет отдельным элементом нового списка.

Если индекс **index** меньше или равен нулю, новые элементы вставляются в начало списка. Если индекс **index** больше или равен числу элементов в списке или равен **end**, новые элементы вставляются в конец списка.

list

Команда создает список.

Синтаксис

```
list?arg arg …?
```

Описание

Команда **list** возвращает новый список из всех элементов arg или пустой список, если аргументы не указаны. При формировании списка по необходимости используются фигурные скобки и обратные слеши, что позволяет потом использовать команду **index** для извлечения исходных аргументов, а также использовать команду **eval** для исполнения результирующего списка, с **arg1**, содержащим имя команды, и остальными **arg** в качестве ее аргументов.

Команда **list** отличается от команды **concat** тем, что команда **concat** удаляет один уровень группирования перед образованием списка, тогда как команда **list** работает непосредственно с исходными аргументами. Например, команда

```
list a b {c d e} {f {g h}}
```

вернет **a b {c d e} {f {g h}}**.

Тогда как команда **concat** с теми же аргументами вернет **a b c d e f {g h}**.

llength

Команда подсчитывает количество элементов в списке.

Синтаксис

```
llength list
```

Описание

Аргумент **list** истолковывается как список; команда возвращает строку с десятичным числом, равным количеству элементов в этом списке.

load

Команда загружает машинный код и инициализирует новые команды.

Синтаксис

```
load fileName
load fileName packageName
load fileName packageName interp
```

Описание

Эта команда загружает двоичный код из файла в адресное пространство приложения и вызывает инициализирующую процедуру библиотеки, чтобы включить ее в интерпретатор. Аргумент **fileName** есть имя файла, содержащего код. Конкретная форма кода различна на разных платформах, но чаще всего он должен быть разделяемой библиотекой, такой как **so**-файлы для Solaris и **dll**-файлы для Windows. Аргумент **packageName** есть имя библиотеки. Оно используется для определения имени инициализационной процедуры. Аргумент **interp** содержит имя интерпретатора. Если аргумент **interp** не указан явно, то по умолчанию библиотека загружается в текущий интерпретатор.

Как только файл загружен в адресное пространство приложения, вызывается одна из двух процедур инициализации нового кода. Обычно процедура инициализации добавляет в интерпретатор новые Tcl-команды. Имя процедуры определяется исходя из имени библиотеки и из того, является интерпретатор, в который будут добавляться команды, безопасным. Для обычного интерпретатора имя процедуры **pkg_Init**, где **pkg** — имя библиотеки, преобразованное следующим образом: первая буква переведена в верхний регистр, а все остальные — в нижний. Например, если имя библиотеки **Foo**, то имя инициализационной процедуры должно быть **Foo_Init**.

Для безопасного интерпретатора имя процедуры должно быть **pkg_SafeInit**. Эта функция должна быть написана очень тщательно, чтобы позволить включить в безопасный интерпретатор только те команды библиотеки, которые безопасны при использовании в ненадежном коде.

Процедура инициализации должна соответствовать следующему прототипу:

```
typedef int Tcl_PackageInitProc(Tcl_Interp *interp);
```

Аргумент **interp** определяет интерпретатор, в который библиотека будет загружена. Процедура инициализации должна вернуть код **TCL_OK** или **TCL_ERROR** чтобы указать, была ли она завершена успешно. В случае ошибки она должна присвоить переменной **interp->result** значение указателя на сообщение об ошибке. Результат процедуры инициализации будет возвращен командой **load** как ее результат.

Реально каждый файл загружается в приложение только один раз. Если команда **load** вызывается несколько раз, чтобы загрузить один и тот же файл в различные интерпретаторы, то реально загрузка кода выполняется только в первый раз. При последующих вызовах будет выполняться только процедура инициализации библиотеки в соответствующем интерпретаторе. Выгрузить или повторно загрузить библиотеку невозможно.

Команда **load** также поддерживает библиотеки, статически связанные с приложением, если они зарегистрированы с использованием процедуры **Tcl_StaticPackage**. Если аргумент **fileName** есть пустая строка, то необходимо указывать имя библиотеки.

Если аргумент **packageName** отсутствует или равен пустой строке, Tcl пытается сам сформировать имя библиотеки. На разных платформах это выполняется по-разному. На Unix-платформах обычно для этого берется последний элемент имени файла. Если первые три буквы в нем — **lib**, то они отбрасываются. После чего берутся все последовательные символы, которые являются буквами или символом подчеркивания. Например, если имя файла **libxyz4.2.so**, то в качестве имени библиотеки будет сформировано **xyz**, а при исполнении команды:

```
load bin/last.so {}
```

сформируется имя библиотеки **last**.

Если аргумент **fileName** равен пустой строке, то должен быть задан аргумент **packageName**. Команда **load** в таком случае ищет сначала статически загружаемую библиотеку (то есть, библиотеку, зарегистрированную с помощью процедуры **Tcl_StaticPackage**) с указанным именем. Если такая будет найдена, то она используется в команде. В противном случае ищется динамически загружаемая процедура с этим именем. Если загружено несколько версий одной и той же библиотеки, то Tcl выбирает ту, которая была загружена первой.

lrange

Команда возвращает один или несколько последовательных элементов списка.

Синтаксис

```
lrange list first last
```

Описание

Команда возвращает новый список — подмножество Tcl-списка **list**, начиная с элемента **first** и заканчивая элементом **last** включительно. Для обозначения последнего элемента списка **list** в аргументе **first** или **last** можно использовать значение **end**. Если аргумент **first** меньше нуля, он считается равным нулю. Если аргумент **last** больше или равен числу элементов в списке, он считается равным **end**. Если аргумент **first** больше, чем **last**, то команда возвращает пустой список.

Команда

```
lrange list first first
```

не всегда возвращает тот же результат, что и

```
lindexlist first
```

(хотя это обычно так при простых элементах списка, которые не заключены в фигурные скобки). Но она всегда возвращает тот же результат, что и команда:

```
list [lindex list first]
```

lreplace

Команда замещает элементы списка новыми элементами.

Синтаксис

```
lreplace list first last?element element …?
```

Описание

Команда **lreplace** возвращает новый список, образованный из списка **list** путем замены его элементов, начиная с **first** до **last**, на аргументы **element**. Если аргумент **first** меньше нуля, он считается равным нулю. Если аргумент **last** больше или равен числу элементов в списке, он считается равным **end**. Если **last** меньше, чем **first**, тогда не будет удалено ни одного прежнего элемента, и новые элементы будут просто вставлены перед **first**.

Для аргументов **first** и **last** можно использовать значение **end** для ссылки на последний элемент списка.

Каждый аргумент **element** становится отдельным элементом списка. Если не было задано ни одного аргумента **element**, тогда прежние элементы списка с **first** по **last** будут просто удалены.

lsearch

Команда проверяет наличие заданного элемента в списке.

Синтаксис

```
lsearch?mode? list pattern
```

Описание

Данная команда просматривает по очереди элементы списка **list** с целью найти первый из них, совпадающий с заданным образцом **pattern**. Команда возвращает индекс первого найденного такого элемента или **-1**, если такой элемент не найден. Аргумент **mode** указывает конкретный способ сравнения элементов списка и образца. Возможные значения аргумента:

-exact

Означает, что элемент списка должен быть в точности равен образцу

-glob

Означает, что используются такие же правила сравнения, как в команде **string match**.

-regexp

Означает, что образец считается регулярным выражением, и для сравнения используются те же правила, что и в команде **regexp**.

Значение по умолчанию **-glob**.

lsort

Команда сортирует элементы списка.

Синтаксис

```
lsort ?options? list
```

Описание

Данная команда сортирует элементы списка **list** и возвращает новый список с уже упорядоченными элементами. По умолчанию используется упорядочивание в порядке возрастания символов в таблице ASCII. Кроме того, для управления процессом сортировки в команде можно указать произвольные из перечисленных ниже опций (допускаются уникальные сокращения).

-ascii

Упорядочивание в порядке возрастания символов в таблице ASCII. Значение по умолчанию.

-dictionary

Словарный режим. Тоже, что и **-ascii**, но со следующими отличиями:

- регистр не учитывается;
- если две строки содержат цифры, то числа сравниваются как целые, а не как символы ASCII.

Например, **bigBoy** окажется между **bigbang** и **bigboy**, а **x10y** между **x9y** и **x11y**.

-integer

Режим целых чисел, когда все элементы списка конвертируются в целые числа и при сравнении трактуются именно как целые числа.

-real

Режим чисел с плавающей запятой; этот режим аналогичен предыдущему и используется для действительных чисел.

-command command

Режим сортировки при помощи произвольной команды пользователя. Для сравнения двух элементов списка выполняется скрипт, состоящий из **command** и дополненный соответствующими элементами списка. Скрипт должен вернуть целое число, большее нуля, равное нулю, или меньшее нуля в зависимости от того, считается ли первый элемент больше второго, равен ему или меньше его.

-increasing

Сортировка в порядке возрастания (от меньших к большим). Это значение по умолчанию.

-decreasing

Сортировка в порядке убывания (от больших к меньшим).

-index index

Эта опция может использоваться, если каждый из элементов списка сам есть список (то есть **list** есть список списков). Опция позволяет отсортировать список по «одной из колонок», то есть по элементам подсписков с фиксированным индексом **index**. Значение аргумента **index**, равное **end**, означает генерацию по последнему элементу подсписков. Например, команда

```
lsort -integer -index 1 {{First 24} {Second 18} {Third 30}}
```

вернет **{Second 18} {First 24} {Third 30}**.

Эта опция значительно более эффективна для данной задачи, чем опция **-command** с соответствующей процедурой сравнения.

namespace

Команда предназначена для создания и управления областями имен для команд и переменных.

Синтаксис

```
namespace ?option??arg …?
```

Описание

Команда позволяет создавать, обращаться и уничтожать отдельные области имен для команд и переменных. Действия, выполняемые командой, зависят от значения аргумента **option**. Ниже приведены возможные опции команды.

namespace children?namespace??pattern?

Позволяет получить список областей-потомков для области имен **namespace**. Если аргумент **namespace** отсутствует, то возвращается список областей-потомков для текущего пространства имен. Команда возвращает полные имена, начинающиеся с «**::**». Если указан аргумент **pattern**, команда возвращает только те имена, которые удовлетворяют шаблону **pattern**. Проверка на соответствие производится по тем же правилам, что и в команде **glob**. При этом шаблон, начинающийся с «**::**», используется непосредственно, в противном случае используемый шаблон составляется из **namespace** (или полного имени текущего пространства имен) и **pattern**.

namespace code script

Позволяет захватить контекст текущей области имен для скрипта **script** с тем, чтобы потом использовать этот контекст при выполнении данного скрипта. Команда возвращает новый скрипт, в котором старый скрипт «завернут» в команду **namespace code**. Новый скрипт имеет две важные особенности: во-первых, он может быть вызван в произвольном пространстве имен, при этом реально он будет выполняться в текущем

пространстве имен (том, в котором выполнялась команда **namespace code**). Во-вторых, к созданному таким образом скрипту можно дописывать произвольные аргументы и они действительно будут использоваться как дополнительные аргументы. Например, предположим, что команда:

```
set script [namespace code {foo bar}]
```

выполняется в пространстве имен **::a::b**. Тогда команда

```
eval "$script x y"
```

может быть выполнена в любом пространстве имен и будет иметь тот же эффект, что и команда

```
namespace eval ::a::b {foo bar x y}
```

Эта команда необходима потому, что расширения Tcl, например, Tk, обычно выполняют вызванные по событиям скрипты в глобальном пространстве имен. Обертывающая команда захватывает необходимую команду вместе с ее пространством имен и позволяет корректно выполнить ее позже по соответствующему вызову.

namespace current

Позволяет получить полное имя текущей области имен. Хотя реальное имя глобального пространства имен пустая строка, эта команда возвращает для него «**::**», поскольку это часто бывает удобнее при программировании.

namespace delete?namespace namespace...?

Позволяет удалить одну или несколько областей имен, при этом удаляются все переменные, процедуры и области-потомки заданной области имен. Если внутри одного из удаляемых пространств имен выполняется какая-либо процедура, оно будет сохранено до завершения процедуры, но помечено, как приготовленное к удалению, чтобы исключить вызов других процедур. Если указанное пространство имен не существует, команда возвращает ошибку. Если ни одно пространство имен не указано, команда не делает ничего.

namespace eval namespace arg?arg...?

Позволяет активизировать пространство имен namespace и выполнить в его контексте заданный код. Если пространство имен namespace не существует, оно будет создано. Если пространство имен должно быть предком несуществующего пространства имен, то оно тоже создается. Если задано более одного аргумента, все они объединяются в одну строку через пробел (как при выполнении команды **eval**) и полученный скрипт выполняется.

namespace export?-clear? ?pattern pattern...?

Позволяет указать, какие команды разрешено экспортировать из данного пространства имен. Эти команды потом могут быть импортированы в другие пространства имен с помощью команды **namespace import**. Можно разрешать экспорт как команд, созданных в данном пространстве имен, так и команд, ранее импортированных из других пространств. Команда, разрешаемая для экспорта, может в данный момент не существовать. Каждый из шаблонов **pattern** может содержать специальные символы как в команде **glob**, но не может содержать имени пространства имен. То есть шаблон может указывать команды только в текущем пространстве имен. Каждый из шаблонов добавляется к списку шаблонов команд, разрешенных для импорта из данного пространства имен. Если в команде указан флаг **-clear**, команда предварительно удаляет старый список. Если не указаны ни флаг, ни шаблоны, команда возвращает текущий список шаблонов.

namespace forget?pattern pattern...?

Удаляет из пространства имен ранее импортированные команды. Каждый образец **pattern** должен быть полным именем команды с указанием хотя бы одного пространства имен и может содержать специальные символы, как в команде **glob**, например: **foo::x** или **a::b::p***. Специальные символы нельзя использовать в именах пространств имен. При выполнении

данной команды сначала ищутся команды, удовлетворяющие шаблону и разрешенные к экспорту из соответствующих пространств имен. Далее проверяется, какие из них были импортированы в текущее пространство имен. После чего импортированные команды удаляются. То есть команда выполняет действия, противоположные действиям команды **namespace import**.

namespace import?-force??pattern pattern...?

Импортирует команды в текущее пространство имен. Каждый образец **pattern** должен быть полным именем команды с указанием хотя бы одного пространства имен и может содержать специальные символы, как в команде **glob**, например **foo::x** или **a::b::p***. Специальные символы нельзя использовать в именах пространств имен. Все команды, которые удовлетворяют шаблонам и разрешены для экспорта, добавляются к текущему пространству имен. Для этого в текущем пространстве имен создается новая команда, которая указывает на экспортируемую команду в исходном пространстве имен. При вызове этой новой команды она вызывает исходную команду. Если в текущем пространстве имен уже есть команда с таким именем, возвращается ошибка, если не указана опция **-force**. В противном случае импортируемая команда заменяет команду, определенную ранее. Команда **namespace import** импортирует в текущее пространство имен только те функции, которые в момент исполнения существуют в соответствующем пространстве имен. Если позже там будут созданы новые команды с именами, удовлетворяющими шаблонам, разрешающим экспорт, и шаблонам, определяющим импорт, то они не импортируются автоматически.

namespace inscope namespace arg?arg...?

Выполняет скрипт в контексте пространства имен **namespace**. Эта команда не предназначена для непосредственного исполнения программистом. Ее вызовы

создаются автоматически при использовании команды **namespace code** для создания скриптов, выполняемых в фоновом режиме, например, для Tk-виджетов. Команда **namespace inscope** похожа на команду **namespace eval**, но отличается от нее наличием возможности указывать дополнительные аргументы и тем, что пространство имен **namespace** должно существовать в момент выполнения команды. При выполнении команды первый аргумент рассматривается как список, к которому остальные аргументы добавляются как элементы списка. Так команда

```
namespace inscope ::foo a x y z
```

эквивалентна

```
namespace eval ::foo [concat a [list x y z]]
```

Такая семантика весьма удобна при формировании скриптов, выполняемых в фоновом режиме.

namespace origincommand

Возвращает полное имя оригинальной команды **command**, от которой происходит заданная импортированная команда. При импорте команды в текущем пространстве имен создается новая команда, которая указывает на экспортируемую команду в исходном пространстве имен. Если команда последовательно импортировать в пространства имен **a, b,...,n**, причем в каждое последующее пространство имен она импортировалась из предыдущего, то **namespace origin** вернет полное имя команды в первом пространстве имен, то есть **a**. Если команда **command** не импортирована, то **namespace origin** вернет ее полное имя.

namespace parent?namespace?

Возвращает полное имя родительского пространства имен для пространства **namespace**. Если аргумент **namespace** не указан, возвращает полное имя предка текущего пространства имен.

namespace qualifiers string

Возвращает полное имя пространства имен для **string**, то есть часть строки **string** от начала до последнего символа «**::**» (но не включая его). Например, для строки **::foo::bar::x** эта команда вернет **::foo::bar**, а для «**::**» — пустую строку. Команда является парной для команды **namespace tail**. При выполнении команды проверка существования соответствующих пространств имен не производится.

namespace tail string

Возвращает простое имя, завершающее полное имя **string**, то есть часть строки **string** от последнего символа «**::**» (но не включая его) до конца строки. Например, для строки **::foo::bar::x** эта команда вернет **x**, а для «**::**» — пустую строку. Команда является парной для команды **namespace qualifiers**. При выполнении команды проверка существования соответствующих пространств имен не производится.

namespace which?-command??-variable? name

Рассматривает **name** как имя команды или переменной (в зависимости от указанной опции; по умолчанию — как имя команды) и возвращает ее полное имя. Например, если **name** не существует в текущем пространстве имен, но существует в глобальном, то возвращает полное имя в глобальном пространстве имен. Если команда или переменная не существует, данная команда возвращает пустую строку.

open

Открывает канал для связи с файлом или программой.

Синтаксис

```
open fileName
open fileName access
open fileName access permissions
```

Описание

Эта команда открывает файл, последовательный порт или командный конвейер и возвращает идентификатор канала, который может использоваться в дальнейшем в таких командах, как **read**, **puts** и **close**. Если первый символ атрибута **fileName** не равен «|», то команда отрывает файл **fileName**, соответственно значение аргумента **fileName** должно соответствовать обычным соглашениям.

Аргумент **access**, если он используется, указывает разрешенные режимы доступа к файлу. Аргумент **access** может указываться в одной из двух нотаций. В первой он может иметь следующие значения:

r

Открывает файл только на чтение. Это значение по умолчанию.

r+

Открывает файл на чтение и запись. Файл должен существовать.

w

Открывает файл только на запись. Удаляет содержимое файла, если он существовал. Если нет, то создает новый файл.

w+

Открывает файл на чтение и запись. Удаляет содержимое файла, если он существовал. Если нет, то создает новый файл.

a

Открывает файл на чтение. Файл должен существовать. Новые данные записываются в конец файла.

a+

Открывает файл на чтение и запись. Если файл не существует, создает новый файл. Новые данные записываются в конец файла.

Во второй нотации аргумент **access** может содержать набор из флагов, описанных ниже. Среди флагов обязательно должен быть один из следующих: **RDONLY**, **WRONLY** или **RDWR**.

RDONLY

Открывает файл только на чтение.

WRONLY

Открывает файл только на запись

RDWR

Открывает файл на чтение и запись.

APPEND

Переставляет указатель в конец файла перед каждой записью.

CREAT

Создает файл, если он не существует. Без этого флага попытка открыть несуществующий флаг приведет к ошибке.

EXCL

Если указан также флаг **CREAT**, то будет сгенерирована ошибка, если файл уже существует.

NOCTTY

Если файл открыт для терминального устройства, этот флаг не позволяет ему стать управляющим терминалом процесса.

NONBLOCK

Позволяет в неблокирующем режиме открыть файл и, возможно, выполнять в этом режиме последующие операции ввода-вывода. Последствия использования этого флага зависят от платформы и устройства. Предпочтительнее вместо него использовать команду **fconfigure**.

TRUNC

Если файл существует, то его содержимое удаляется.

Если файл создается при выполнении команды **open**, то аргумент **permissions** (целое число) используется для установки прав доступа к вновь созданному файлу. Значение по умолчанию **0666**.

package

Команда загрузки пакетов библиотек и контроля версий.

Синтаксис

```
package forget package
package ifneeded package version?script?
package names
package provide package?version?
package require? -exact? package?version?
package unknown? command?
package vcompare version1 version2
package versions package
package vsatisfies version1 version2
```

Описание

Команда поддерживает простую базу данных со сведениями о том, какие пакеты библиотечных функций доступны для использования в данном интерпретаторе, и как их загрузить в интерпретатор. Она поддерживает

многоверсионность пакетов и гарантирует загрузку в приложение необходимой версии пакета. При этом она обеспечивает контроль непротиворечивости версий. Обычно в Tcl-скриптах достаточно использовать команды **package require** и **package provide**. Остальные команды предназначены в первую очередь для системных скриптов, которые поддерживают базу данных пакетов.

Поведение команды **package** определяется ее первым аргументом. Ниже описаны возможные формы команды.

package forget package

Удаляет информацию о пакете из интерпретатора, включая как информацию, возвращаемую командой **package ifneeded**, так и информацию, возвращаемую командой **package provide**.

package ifneeded package version?script?

Команда используется в системных скриптах. Если указана конкретная версия и скрипт, заносит в базу данных информацию о том, что соответствующая версия пакета доступна и может быть загружена в интерпретатор с помощью скрипта **script**. Если в базе данных уже хранится информация о скрипте, она обновляется. Если скрипт не указан, возвращается текущий скрипт.

package names

Возвращает список всех пакетов в интерпретаторе, для которых известна доступная версия (заданная с помощью команды **package provide**) или скрипт загрузки (заданный командой **package ifneeded**).

package provide package?version?

Команда используется для того, чтобы указать, что версия version пакета package загружена в интерпретатор. Если ранее была загружена другая версия, возвращает ошибку. Если версия не указана, возвращает загруженную версию.

package require?-exact? package?version?

Команда используется, если для выполнения дальнейшего кода необходим библиотечный пакет **package** версии **version**. Команда возвращает номер загруженной версии или ошибку. Если присутствуют оба аргумента **-exact** и **version**, то команда загружает именно указанную версию (или выдает ошибку, если эта версия недоступна). Если присутствует номер версии, а **-exact** опущено, то команда загружает указанную версию или более позднюю, но с тем же старшим номером версии (2.7, но не 3.1, когда указана версия 2.5).

Если база данных не содержит необходимой версии, а в интерпретаторе определена команда для **packageunknown**, то она исполняется, после чего повторно проверяется доступность необходимой версии. Если версия по-прежнему недоступна, команда возвращает ошибку.

package unknown ?command?

Команда «последней надежды» при поиске необходимой версии пакета. Если аргумент **command** указан, то он дополняется именем пакета и версии и полученный скрипт выполняется. Если версия не указана, подставляется пустая строка.

Если аргумент **command** не указан, возвращает текущий скрипт, заданный для команды **package unknown**.

package vcompare version1 version2

Команда сравнения версий. Возвращает **-1**, если **version1** более ранняя версия, чем **version2**, **1** — в противном случае, и **0**, если они равны.

package versions package

Возвращает список всех доступных версий пакета (информация о которых занесена в базу данных с помощью команды **package ifneeded**).

package vsatisfies version1 version2

Возвращает **1**, если у **version1** и **version2** совпадают старшие номера, а младший номер у **version1** не меньше, чем у **version2**, и **0** — в противном случае.

pid

Команда сообщает идентификаторы процессов.

Синтаксис

```
pid ?fileId?
```

Описание

Если задан аргумент **fileId**, то он должен указывать на конвейер процессов (**process pipeline**), открытый командой **open**. В этом случае команда представит список идентификаторов всех процессов в конвейере, по порядку. Если **fileId** указывает на открытый файл, не являющийся конвейером процессов, то список будет пустой. Если аргумент **fileId** не задан, то будет возвращен идентификатор текущего процесса. Все идентификаторы представлены десятичными строками.

pkg_mkIndex

Создает индексный файл для автоматической загрузки пакета.

Синтаксис

```
pkg_mkIndex dir pattern?pattern pattern...?
```

Описание

Процедура **pkg_mkIndex** представляет собой утилиту для работы с Tcl-библиотеками. Она обеспечивает создание индексных файлов, необходимых для автоматической загрузки пакетов, когда в приложении встречается команда **package**

require. Для создания автоматически загружаемых пакетов необходимо выполнить следующие действия:

1. Создать один или несколько пакетов. Каждый пакет может состоять из одного или больше файлов с Tcl-скриптами или из бинарных файлов. Бинарные файлы должны быть пригодны для их загрузки с помощью команды **load** с единственным аргументом -именем файла. Например, если в пакет входит файл **test.so**, он должен загружаться командой **load test.so**. Каждый файл Tcl-скриптов должен содержать команду **package provide** с именем пакета и версией. Каждый бинарный файл должен содержать вызов процедуры **Tcl_PkgProvide**.

2. Создать индексные файлы с помощью команды **pkg_mkIndex**. Аргумент **dir** указывает имя каталога, в котором лежат файлы пакета, а шаблоны **pattern**, которые могут содержать специальные символы, как в команде **glob**, указывают на файлы в этом каталоге. Команда **pkg_mkIndex** создаст в каталоге **dir** файл **pkgIndex.tcl**, содержащий информацию обо всех файлах пакета, заданных с помощью аргументов **pattern**. Для этого загружаются все файлы пакета, и определяется, какие новые пакеты и какие новые процедуры появились (поэтому в каждом файле пакета и должна быть команда **package provide** или вызов **Tcl_PkgProvide**).

3. Установить пакет как подкаталог одного из каталогов, перечисленных в переменной **tcl_pkgPath**. Если в списке **$tcl_pkgPath** больше одного каталога, то бинарные файлы с разделяемыми библиотеками обычно устанавливаются в первом каталоге, а библиотеки Tcl-скриптов — во втором. В этих каталогах должны также находиться файлы **pkgIndex.tcl**. Пока пакеты будут размещаться в подкаталогах каталогов, перечисленных в переменной **tcl_pkgPath**, этого будет достаточно для их автоматической загрузки при выполнении команды **package require**.

Если вы установили пакеты в каких-либо других каталогах, то необходимо, чтобы эти каталоги содержались в

переменной **auto_path** или были бы непосредственными подкаталогами одного из содержащихся там каталогов. Переменная **auto_path** содержит список каталогов, которые просматриваются как автозагрузчиком, так и загрузчиком пакетов. По умолчанию он включает **$tcl_pkgPath**. Загрузчик пакетов также просматривает и подкаталоги каталогов, включенных в **auto_path**. Пользователь может в явном виде включить в приложении необходимые каталоги в **auto_path**. А можно включить эти каталоги в переменную окружения **TCLLIBPATH**. Если она существует, то используется для инициализации переменной **auto_path** при запуске приложения.

Если перечисленные выше шаги выполнены, то для использования необходимого пакета достаточно выполнить в приложении команду **package require**.

proc

Создает Tcl-процедуры.

Синтаксис

```
proc name args body
```

Описание

Команда **proc** создает новую Tcl-процедуру с именем **name**, если такой процедуры ранее не было, и замещает ранее существовавшую процедуру или команду с таким именем, если она была. При вызове новой процедуры скрипт **body** передается на выполнение Tcl-интерпретатору. Обычно имя процедуры указывается без указания имени пространства имен. При этом новая процедура создается в текущем пространстве имен. Однако, если пространство имен указано явно, она создается в указанном пространстве. Аргумент **args** определяет формальные аргументы процедуры и представляет собой список (возможно, пустой), каждый элемент которого представляет описание одного формального параметра. Каждое такое описание само

является списком из одного или двух элементов. Первый элемент списка определяет имя формального параметра. Второй элемент списка, если он указан, определяет значение по умолчанию для данного параметра.

При выполнении процедуры для каждого формального параметра создается локальная переменная. Ей присваивается значение соответствующего аргумента, указанного при вызове процедуры, или значение по умолчанию. Аргумент, для которого при определении процедуры указано значение по умолчанию, может не присутствовать в вызове процедуры. Однако общее количество указанных параметров должно быть достаточным для аргументов, не имеющих значения по умолчанию, но не больше общего числа формальных параметров. Если это условие выполнено, все аргументы процедуры собираются в один список (как при исполнении команды **list**). Эта комбинированная величина присваивается локальной переменной **args**.

При исполнении тела процедуры имена переменных обычно считаются именами локальных переменных, которые создаются автоматически по мере необходимости и удаляются после завершения процедуры. По одной локальной переменной создается также для каждого аргумента процедуры. Для использования глобальных переменных необходимо использовать команду **global** или **upvar**. Для использования переменных из пространства имен необходимо использовать команду **variable** или **upvar**.

Команда **proc** возвращает пустую строку. При вызове процедуры возвращается величина, заданная в команде **return**. Если в процедуре не выполнялась явная команда **return**, она возвращает результат выполнения последней команды, выполнявшейся в теле процедуры. Если при выполнении процедуры произошла ошибка, то процедура в целом возвращает эту ошибку.

puts

Команда записывает данные в канал.

Синтаксис

```
puts?-nonewline??channelId?  string
```

Описание

Записывает символы из аргумента **string** в канал **channelId**. Значение **channelId** должно быть идентификатором канала, который вернула предыдущая команда **open** или **socket**. Соответствующий канал должен быть открыт на запись. Если аргумент **channelId** не указан, значение по умолчанию соответствует стандартному выводу. Команда **puts** обычно выдает после **string** символ новой строки, однако, если указана опция **-nonewline**, этого не происходит.

Tcl осуществляет вывод через буфер. Поэтому символы, выданные командой **puts**, могут и не появиться сразу в выходном устройстве или в файле. Обычно вывод откладывается до заполнения буфера или закрытия канала. Чтобы обеспечить немедленную выдачу данных, можно использовать команду **flush**.

Когда буфер заполнится, команда puts обычно блокирует процесс до тех пор, пока все данные не будут переданы операционной системе для дальнейшего вывода. Если канал **channelId** открыт в неблокирующем режиме, процесс не блокируется, даже если операционная система еще не приняла данные. Tcl в этом случае продолжает складывать данные в буфер и в фоновом режиме передает их в соответствующий файл или устройство с той скоростью, с которой они могут принять данные. Чтобы работа в неблокирующем режиме была возможна, необходимо, чтобы был запущен обработчик событий.

При работе в неблокирующем режиме возможен рост буфера, под который будет выделен неоправданно большой объем памяти. Чтобы избежать этого, неблокирующие операции

ввода-вывода лучше делать управляемыми по событиям. При этом новая порция данных не будет передаваться в буфер, пока канал не будет готов к ее приему.

pwd

Команда **pwd** возвращает путь к текущему каталогу.

Синтаксис

```
pwd
```

Описание

Команда **pwd** возвращает полный путь к текущему каталогу.

read

Команда выполняет чтение данных из канала.

Синтаксис

```
read ?-nonewline? channelId
read channelIdnumBytes
```

Описание

Команда **read** зачитывает из канала **channelId** либо весь файл до символа конца файла (при первой форме записи), либо заданное параметром **numBytes** количество байтов (вторая форма). Если во втором случае в файле оказалось меньше байтов, чем задано **numBytes**, тогда возвращаются все байты, что остались.

Если указана опция **-nonewline**, при выполнении команды отбрасывается символ новой строки в конце файла.

Если канал открыт в неблокирующем режиме, команда может прочитать не указанное количество байтов, а только все доступные. После чего она не заблокирует процесс, дожидаясь

дополнительных данных, а завершится. Если команда завершилась до конца файла, то опция **-nonewline** игнорируется.

Команда **read** изменяет во входных данных последовательность, задающую конец строки, в соответствии с опцией канала **-translation option**. Опция может быть изменена с помощью команды **fconfigure**.

regexp

Сравнивает строку и регулярное выражение.

Синтаксис

```
regexp?switches? exp string?matchVar??subMatchVar subMatchVar...?
```

Описание

Команда определяет, соответствует ли регулярное выражение **exp** какой-либо части строки **string** или всей строке, и возвращает **1**, если соответствует, и **0** — в противном случае.

Если в команде указаны дополнительные аргументы после **string**, они считаются именами переменных, в которые возвращается информация о том, какие именно части строки соответствуют регулярным выражениям. Переменной присваивается значение, состоящее из части строки, соответствующей всему регулярному выражению. Самой левой в списке переменной **subMatchVar** присваивается значение, состоящее из части строки, которая соответствует самому левому заключенному в скобки выражению в составе **exp**. Следующей переменной **subMatchVar** присваивается значение, соответствующее следующему заключенному в скобки выражению, и так далее.

Если первые аргументы команды начинаются с «-», они считаются опциями команды. Ниже перечислены возможные опции.

-nocase

При сравнении не различает буквы в верхнем и нижнем регистре.

-indices

В переменных **subMatchVars** сохраняются не части строки, а списки из двух десятичных чисел — индексов начала и конца соответствующей области строки.

--

Означает конец опций. Следующий аргумент будет рассматриваться как **exp**, даже если он начинается с «**-**».

Если в команде указано больше переменных **subMatchVar**, чем выражений в скобках в **exp**, или если для одного из выражений не удалось найти соответствующую ему часть строки, то соответствующей переменной будет присвоено значение «**-1 -1**» или пустая строка, в зависимости от того, была ли задана опция **-indices**.

regsub

Команда выполняет подстановки, используя регулярные выражения.

Синтаксис

```
regsub? switches? exp string subSpec varName
```

Описание

Команда сравнивает регулярное выражение **exp** и строку **string** и копирует **string** в переменную, заданную именем **varName**. Если совпадение найдено, то при копировании часть строки **string**, соответствующая **exp**, замещается на **subSpec**. Если **subSpec** содержит один из символов **&** или **\0**, то он заменяется на часть строки **string**, которая соответствует шаблону **exp**. Если **subSpec** содержит **\n**, где **n** – целое число от 1 до 9, то это

выражение заменяется на часть строки **string**, которая соответствует n-ому заключенному в скобки выражению в **exp**. Если начальные аргументы команды начинаются с символа «**-**», они считаются опциями команды. Ниже приведен список поддерживаемых опций.

-all

Ищутся все подобласти **string**, соответствующие **exp**, и для каждой из них производится замена. Символы **&** и **\n** замещаются на очередной фрагмент **string**, соответствующий **exp**. То есть каждый раз они могут замещаться на различные выражения.

-nocase

При поиске соответствующих фрагментов строки не различаются буквы в верхнем и нижнем регистре. Тем не менее, подстановка производится в исходном регистре.

--

Означает конец опций. Следующий аргумент будет рассматриваться как **exp**, даже если он начинается с «**-**».

Команда возвращает количество найденных (и, соответственно, замещенных) интервалов.

rename

Команда **rename** переименовывает или удаляет команду.

Синтаксис

```
rename oldName newName
```

Описание

Данная команда переименовывает команду по имени **oldName** в **newName**. Если **newName** отсутствует (равно пустой строке), тогда команда **oldName** удаляется.

Имена **oldName** и **newName** могут содержать квалификаторы областей имен (указатели на имена областей имен). Если команда переименовывается в другую область имен, то последующие вызовы этой команды будут происходить в новой области имен. Команда **rename** возвращает пустую строку.

resource

Управляет Macintosh-ресурсами.

Синтаксис

```
resource option?arg arg...?
```

Описание

Команда **resource** позволяет управлять ресурсами на платформах Macintosh. На остальных платформах не поддерживается.

return

Команда осуществляет возврат из процедуры.

Синтаксис

```
return ?-code code?? -errorinfo info?? -errorcode code??string?
```

Описание

Команда немедленно осуществляет возврат из текущей процедуры (или команды верхнего уровня, или команды **source**) со значением, заданным **string**. Если аргумент **string** не задан, возвращает пустую строку.

Обычно опция **-code** не используется, и процедура завершается успешно (с кодом завершения **TCL_OK**). Однако, ее можно использовать для генерации других кодов возврата. Ниже перечислены возможные коды.

ok

Успешное завершение. То же самое, что отсутствие кода.

error

Возвращает ошибку. То же самое, что использовать команду **error** для прекращения выполнения процедуры за исключением обработки переменных **errorInfo** и **errorCode**.

return

Текущая процедура вернет код **TCL_RETURN**, который вызовет немедленный возврат также и из вызывающей процедуры.

break

Текущая процедура вернет код **TCL_BREAK**, который вызовет немедленное прекращение выполнения самого внутреннего из вложенных циклов, из которого была вызвана процедура.

continue

Текущая процедура вернет код **TCL_CONTINUE**, который вызовет немедленное прекращение выполнения текущей итерации самого внутреннего из вложенных циклов, из которого была вызвана процедура.

value

Значение **value** должно быть целым числом. Оно будет возвращено как код выполнения процедуры.

Опция **-code** используется относительно редко. Она предусмотрена для того, чтобы процедуры, реализующие новые управляющие команды, могли вернуть вызывающим их процедурам исключительные условия.

Опции **-errorinfo** и **-errorcode** могут использоваться совместно с **-code error**, чтобы вернуть дополнительную

информацию о сгенерированной ошибке. В остальных случаях они игнорируются.

Опция **-errorinfo** используется для того, чтобы задать исходное значение переменной **errorInfo**. Если она не будет задана, то в переменную **errorInfo** будет включена информация о вызове процедуры, вернувшей ошибку, и о более высоких уровнях стека, но не информация непосредственно об ошибке внутри процедуры. Чаще всего для формирования переменной **info** используется сообщение команды **catch**, обнаружившей ошибку в процедуре.

Если опция **-errorcode** указана, она позволяет задать значение переменной **errorCode**. В противном случае ей будет присвоено значение **NONE**.

scan

Производит разбор строки в стиле процедуры **sscanf**.

Синтаксис

```
scan string format varName?varName...?
```

Описание

Данная команда, подобно ANSI C процедуре **sscanf**, просматривает строку **string**, выбирает поля и преобразует их в соответствии с очередным спецификатором преобразования в строке **format**. Выбранные значения преобразуются обратно в строковый вид и последовательно записываются в переменные **varName**.

Команда **scan** просматривает одновременно строки **string** и **format**. Если очередной символ в строке **format** — пробел или табуляция, то он соответствует любому числу (включая ноль) пробельных символов (пробел, табуляция, новая строка) в строке **string**. Если в строке **format** встретился символ %, он означает начало очередного спецификатора преобразования. Спецификатор преобразования включает в себя до трех полей

после символа %: первое поле может содержать символ *, означающий, что преобразуемая величина будет удалена, а не записана в очередную переменную, второе поле может содержать число, указывающее максимальную ширину поля, и третье поле содержит букву, определяющую тип преобразования. Обязательным является только третье поле.

Когда команда **scan** находит спецификатор преобразования в строке **format**, она пропускает пробельные символы в строке **string**. Затем она выбирает следующую группу непробельных символов и преобразует их в соответствии со спецификатором преобразования и записывает результат в переменную, соответствующую следующему аргументу команды. Поддерживаются следующие типы преобразований, задаваемые соответствующими символами:

d

Входное поле должно быть десятичным числом. Результат записывается как десятичная строка.

o

Входное поле должно быть восьмеричным числом. Результат преобразуется в десятичную строку.

x

Входное поле должно быть шестнадцатеричным числом. Результат преобразуется в десятичную строку.

c

Читается один символ, его двоичная величина преобразуется и записывается в переменную как десятичная строка. Начальные пробельные символы в этом случае не пропускаются. В отличие от ANSI C процедуры **sscanf** входное поле всегда состоит ровно из одного символа, а ширина поля не может быть задана.

s

Входное поле состоит из всех непробельных символов до следующего пробельного. Все символы копируются в переменную.

e или f или g

Входное поле должно быть числом с плавающей точкой, состоящим из знака (не обязательно), строки десятичных цифр, возможно с десятичной точкой и порядка (не обязательно), состоящего из буквы **e** или **E**, знака порядка (не обязательно) и строки десятичных цифр.

[chars]

Входное поле состоит из произвольного числа символов из **chars**. Соответствующая строка записывается в переменную. Если первый символ в скобках **]**, то он рассматривается как часть **chars**, а не как закрывающая скобка для множества символов

[^chars]

Входное поле состоит из произвольного числа символов, не содержащихся в **chars**. Соответствующая строка записывается в переменную. Если первый символ в скобках после ^ есть **]**, то он рассматривается как часть **chars**, а не как закрывающая скобка для множества символов.

Число символов, которое выбирается для преобразования, это максимально возможное число символов для соответствующего преобразования (например, так много десятичных цифр, как это возможно для **%d**, или так много восьмеричных цифр, как это возможно для **%o**). Поле, выбираемое для очередного преобразования, закрывается, как только в строке **string** встречается пробельный символ или как только поле достигает указанного максимального размера (в зависимости от того, что происходит раньше). Если в спецификаторе преобразования для очередного поля

присутствует символ *****, то выбранное значение не присваивает никакой переменной, а очередной аргумент команды **scan** по-прежнему считается неиспользованным.

seek

Команда изменяет позицию доступа открытого канала.

Синтаксис

```
seek channelIdoffset ?origin?
```

Описание

Команда изменяет текущую позицию доступа канала, заданного параметром **channelId**. Значение **channelId** должно быть идентификатором канала, который вернула предыдущая команда **open** или **socket**. Аргументы **origin** и **offset** задают новую позицию, в которой будет выполняться следующая операция чтения или записи. Аргумент **offset** должен быть целым числом (возможно, отрицательным), а аргумент **origin** может принимать одно из перечисленных ниже значений.

start

Следующая позиция будет на расстоянии **offset** байтов от начала соответствующего файла или устройства.

current

Следующая позиция будет на расстоянии **offset** байтов от текущей позиции. Отрицательное значение **offset** передвигает позицию назад.

end

Следующая позиция будет на расстоянии **offset** байтов от конца файла или устройства. Отрицательное значение **offset** указывает на позицию до конца файла, а положительное – на позицию после конца файла.

Значение по умолчанию для аргумента **origin** равно **start**.

Выполнение команды влечет немедленную передачу всех данных из выходного буфера в файл или на выходное устройство. Команда не будет завершена до тех пор, пока все данные не будут переданы, даже если канал находится в неблокирующем режиме. Кроме того, будут удалены все не прочитанные данные из входного буфера. Команда возвращает пустую строку. Если команда используется для файла или канала, для которого не поддерживается произвольный доступ, она вернет ошибку.

set

Команда читает и записывает значения переменных.

Синтаксис

```
set varName?value?
```

Описание

Команда **set** возвращает значение переменной **varName**. Если задан параметр **value**, то команда присваивает переменной **varName** значение **value** и возвращает значение **value**. Если такой переменной не существовало, тогда она создается вновь.

Если **varName** содержит открывающую скобку и заканчивается закрывающей скобкой, тогда это элемент массива. Символы до открывающей скобки являются именем массива, символы между скобками есть индекс этого элемента в массиве. В противном случае команда адресуется к скалярной переменной.

Обычно имя переменной указывается без указания пространства имен, в котором она содержится. При этом соответствующая переменная для чтения или записи ищется в текущем пространстве имен. Если же в имени переменной присутствуют имя пространства имен, то она ищется в указанном пространстве имен.

Если команда используется вне тела процедуры, то **varName** есть имя глобальной переменной (если текущее пространство имен есть глобальное пространство) или переменной текущего пространства имен. В теле процедуры **varName** есть имя параметра или локальной переменной процедуры, если она не объявлена глобальной переменной или переменной пространства имен с помощью команды **global** или **variable** соответственно.

socket

Команда открывает сетевое TCP-соединение.

Синтаксис

```
socket?options? host port
socket -server command?options? port
```

Описание

Эта команда открывает сетевое соединение и возвращает идентификатор канала, который может использоваться в последующих командах **read**, **puts** или **flush**. В настоящее время поддерживается только протокол TCP. Команда может использоваться для открытия соединения как со стороны сервера, так и со стороны клиента.

Для задания дополнительной информации о соединении можно использовать следующие опции.

-myaddr addr

Аргумент **addr** задает доменный или числовой адрес сетевого интерфейса клиентской стороны для упрощения соединения. Эта опция может быть полезна, если на клиентской машине есть несколько сетевых интерфейсов. Если опция не указана, системный интерфейс будет выбран операционной системой.

-myport port

Аргумент **port** задает номер порта для клиентской стороны соединения. Если опция не указана, номер порта для клиента будет определен операционной системой.

-async

Использование опции **-async** приведет к тому, что клиент будет подсоединен в асинхронном режиме. Это значит, что сокет будет создан немедленно, возможно, еще до установления связи с сервером. Если канал открыт в блокирующем режиме, то при выполнении команды **gets** или **flush** по такому сокету, команда завершится только после того, как процесс установления соединения будет завершен. Если канал открыт в неблокирующем режиме, то в этой ситуации команда завершится немедленно, а команда **fblocked** для данного канала возвратит **1**.

source

Команда **source** исполняет скрипт, содержащийся в файле.

Синтаксис

```
source  fileName
```

Описание

Данная команда передает интерпретатору Tcl содержимое названного файла в качестве исполняемого скрипта. Команда возвращает результат выполнения последней выполненной команды скрипта. Если при выполнении скрипта возникла ошибка, то команда **source** возвратит эту ошибку. Если при выполнении скрипта была вызвана команда **return**, то выполнение скрипта прекращается, и команда завершается. При этом она возвращает результат выполнения команды **return**.

Для Macintosh-платформ существуют дополнительные варианты команды, предназначенные для работы с ресурсами.

split

Команда разделяет строку на части и создает из них правильный Tcl-список.

Синтаксис

```
split  string?splitChars?
```

Описание

Команда делит строку **string** в каждом месте, где есть символ, содержащийся в **splitChars**. Каждый элемент списка образован частью исходной строки, заключенной между двумя последовательными вхождениями символов из **splitChars** в строку. В списке формируется пустой элемент, если два символа из **splitChars** встречаются подряд или если первый или последний символ **string** содержится в **splitChars**. Если **splitChars** есть пустая строка, то строка разбивается на отдельные символы. По умолчанию **splitChars** содержит пробельные символы (пробел, табуляция, новая строка).

string

Команда для работы со строками.

Синтаксис

```
string  option  arg?arg...?
```

Описание

Выполняет одну из перечисленных ниже строковых операций в зависимости от заданной опции **option**.

string compare string1 string2

Выполняет посимвольное сравнение строк **string1** и **string2** так же, как С-процедура **strcmp**. Возвращает **-1**, **0** или **1**, в зависимости от того, будет ли строка **string1** больше, равна или меньше (при лексикографическом сравнении) строки **string2**.

string first string1 string2

Ищет в строке **string2** последовательность символов, в точности совпадающую со **string1**. Если такая последовательность есть, возвращает индекс первой буквы в первой найденной последовательности. В противном случае возвращает **-1**.

string index string charIndex

Возвращает **charIndex**-ный символ в строке **string**. Значение **charIndex**, равное **0**, соответствует первому символу в строке. Если значение **charIndex** меньше **0** или не меньше длины строки — возвращает пустую строку.

string last string1 string2

Ищет в строке **string2** последовательность символов, в точности совпадающую со **string1**. Если такая последовательность есть, возвращает индекс первой буквы в последней найденной последовательности. В противном случае возвращает **-1**.

string length string

Возвращает десятичную строку, содержащую число символов в строке.

string match pattern string

Проверяет, соответствует ли строка образцу. Возвращает **1**, если соответствует, и **0** — в противном случае. Соответствие проверяется примерно так же, как в C-shell. Строка соответствует шаблону, если они совпадают посимвольно, за исключением перечисленных ниже специальных случаев:

***** — Удовлетворяет любой последовательности из нуля или больше символов;

? – Удовлетворяет любому символу;

[chars] – Удовлетворяет любому символу из **chars**. Если **chars** включает последовательность символов типа **a-b**, то удовлетворяет всем символам от **a** до **b** (включительно).

\x – Удовлетворяет символу **x**. Обеспечивает возможность избежать в шаблонах специального смысла символов *****, **?**, **[**, **]**, \.

string range string first last

Возвращает подстроку строки **string**, начиная с символа с индексом **first** и кончая символом с индексом **last**. Индекс **0** указывает на первый символ строки. Индекс **end** (или любое его сокращение) указывает на последний символ строки. Если значение **first** меньше **0**, используется значение **0**. Если **last** больше значения индекса последнего символа в строке, используется значение **end**. Если **first** больше, чем **last**, команда возвращает пустую строку.

string tolower string

Возвращает строку, тождественную **string**, за исключением того, что все символы верхнего регистра в ней переведены в нижний регистр.

string toupperstring

Возвращает строку, тождественную **string**, за исключением того, что все символы нижнего регистра в ней переведены в верхний регистр.

string trim string?chars?

Возвращает строку, тождественную **string**, за исключением того, что из нее удалены все начальные и конечные символы, входящие в **chars**. Если аргумент **chars** не указан, удаляются пробельные символы (пробелы, табуляция, символы новой строки).

string trimleft string?chars?

Возвращает строку, тождественную **string**, за исключением того, что из нее удалены все начальные символы, входящие в **chars**. Если аргумент **chars** не указан, удаляются пробельные символы (пробелы, табуляция, символы новой строки).

string trimright string?chars?

Возвращает строку, тождественную **string**, за исключением того, что из нее удалены все конечные символы, входящие в **chars**. Если аргумент **chars** не указан, удаляются пробельные символы (пробелы, табуляция, символы новой строки).

string wordend string index

Возвращает индекс символа, идущего сразу после последнего символа в слове, содержащем **index**-ный символ строки **string**. Словом считается любая непрерывная последовательность из букв, цифр и символа подчеркивания, или любой другой одиночный символ.

string wordstart string index

Возвращает индекс первого символа в слове, содержащем **index**-ный символ строки **string**. Словом считается любая непрерывная последовательность из букв, цифр и символа подчеркивания, или любой другой одиночный символ.

subst

Команда выполняет подстановки переменных, команд и подстановки с обратным слешем.

Синтаксис

```
subst ?-nobackslashes ? ?-nocommands? ?-novariables? string
```

Описание

Команда **subst** выполняет подстановки переменных, подстановки команд и подстановки с обратным слешем в строке

string и возвращает получившуюся строку. Все подстановки выполняются обычным для Tcl образом. В результате подстановки в строке **string** выполняются дважды: один раз — анализатором команд Tcl и второй раз — командой **subst**.

Если задан любой из ключей **-nobackslashes**, **-nocommands** или **-novariables**, то соответствующие подстановки не выполняются.

switch

Команда выполняет один из нескольких скриптов в зависимости от полученного значения.

Синтаксис

```
switch?options? string pattern body?pattern body...?
switch?options? string {pattern body?pattern body...?}
```

Описание

Команда **switch** сравнивает аргумент **string** по очереди с каждым из образцов, заданных аргументами **pattern**. Если строка **string** соответствует очередному образцу, выполняется соответствующий скрипт **body** и команда возвращает результат его выполнения. Если последний из образцов равен **default**, то ему соответствует любая строка. Если строка не соответствует ни одному из образцов (что значит, в частности, что образец **default** отсутствует), то никакой скрипт не выполняется и команда возвращает пустое значение.

Если один или несколько первых аргументов команды начинаются с «-», они считаются опциями команды. Возможные опции перечислены ниже.

-exact

Строка считается соответствующей образцу, только если она в точности с ним совпадает. Этот режим используется по умолчанию.

-glob

При сравнении строки с образцом используются те же правила, что и в команде **string match**.

-regexp

При сравнении строки с образцом используются те же правила, что и в команде **regexp**.

-

Обозначает конец опций. Следующий аргумент считается строкой **string**, даже если он начинается с символа «**-**».

Команда предполагает использование одной из двух синтаксических форм для задания образцов и скриптов. Первая использует отдельные аргументы для каждого образца и скрипта. Эта форма удобна при необходимости выполнить подстановки в образцах и/или скриптах. Во второй форме все они помещаются в один аргумент, который должен быть списком. Элементами этого списка должны быть, соответственно, образцы и скрипты. Эта форма более удобна для длинных команд, не размещающихся в одной строке, поскольку она не требует использовать обратный слеш в конце каждой строки. Но при ее использовании необходимо учитывать, что, поскольку список содержится в фигурных скобках, подстановки команд и переменных в образцах и скриптах не производятся. Вследствие этого результат выполнения команды, записанной в различных формах, может различаться.

Если один из аргументов **body** равен «**-**», это означает, что при совпадении строки с данным образцом будет выполняться скрипт **body** для следующего образца. Если для следующего образца скрипт также равен «**-**», то будет использован скрипт для следующего за ним образца и т.д.

tell

Команда возвращает текущую позицию поиска открытого канала.

Синтаксис

```
tell channelId
```

Описание

Команда **tell** возвращает десятичную строку, указывающую текущую позицию доступа в канале **channelId**. Если канал не поддерживает прямой доступ, то возвращается значение **-1**.

time

Команда выполняет скрипт заданное количество раз.

Синтаксис

```
time script?count?
```

Описание

Команда **time** вызовет интерпретатор Tcl **count** раз для выполнения скрипта **script** или только один раз, если аргумент **count** не задан. Команда возвратит строку вида

```
503 microseconds per iteration
```

отображающую среднее время в микросекундах, израсходованное на одну итерацию. В строке указывается прошедшее время, а не время работы процессора.

trace

Команда отслеживает работу с переменными.

Синтаксис

```
trace option?arg arg...?
```

Описание

Эта команда вызывает выполнение указанных Tcl команд при определенных действиях с переменной. Ниже перечислены возможные опции команды (допускаются сокращения).

trace variable name ops command

Обеспечивает выполнение команды **command** при определенных действиях с переменной **name**. Аргумент **name** может содержать имя простой переменной, имя элемента массива или имя массива. Если **name** содержит имя массива, то команда command выполняется при соответствующих действиях с любым элементом массива.

unknown

Команда **unknown** обрабатывает попытки обратиться к несуществующей команде.

Синтаксис

```
unknown cmdName?arg arg …?
```

Описание

Интерпретатор Tcl выполняет эту команду каждый раз, когда скрипт пытается обратиться к несуществующей команде. Исходный вариант **unknown** не является функцией ядра Tcl; напротив, это библиотечная процедура, определяемая по умолчанию при запуске Tcl. Разработчик может переопределить ее функциональность так, как ему нужно.

Когда Tcl находит имя команды, которому не соответствует ни одной из существующих команд, тогда он проверяет наличие команды unknown. Если команды unknown нет, то он возвращает ошибку. Если такая команда обнаружена, то она будет вызвана с аргументами, состоящими из имени и аргументов исходной несуществующей команды, в которых выполнены все необходимые подстановки.

Команда **unknown** обычно выполняет поиск по библиотечным каталогам процедуры с именем **cmdName**, или поиск полного имени команды, к которой обратились по сокращенному имени, или автоматический запуск неизвестной команды как подпроцесса. При успешном поиске полного имени команды команда **unknown** заменяет имя на полное и вызывает команду с полным именем. Результат работы команды **unknown** используется вместо результата неопределенной команды.

unset

Команда удаляет переменные.

Синтаксис

```
unset name?name name …?
```

Описание

Команда **unset** удаляет переменные **name**. Правила именования переменных точно такие же, как для команды **set**. Если в команде указано имя элемента массива, то этот элемент будет удален из массива, не влияя на остальную часть массива. Если указано имя массива без индекса в скобках, то будет удален весь массив.

Команда возвращает пустую строку. Если одна из переменных не существует, команда вернет ошибку, а последующие переменные не будут удалены.

update

Команда **update** обрабатывает события, находящиеся в состоянии ожидания, и обратные вызовы (**idle callbacks**).

Синтаксис

```
update?idletasks?
```

Описание

С помощью этой команды обновляется состояние приложения, поскольку при ее вызове обрабатываются все необработанные события и выполняются все асинхронные вызовы (**idle callbacks**).

Если в команде задана опция **idletasks**, то новые события и ошибки не обрабатываются, но выполняются все асинхронные вызовы. Команду **update idletasks** удобно использовать тогда, когда нужно выполнить немедленно действия, которые обычно откладываются, например, обновить отображаемые на дисплее данные или окна. Большинство обновлений изображений на дисплее выполняются в виде фоновых вызовов, и эта команда обеспечит их выполнение. Однако, если изменения были вызваны событиями, они не будут выполнены немедленно.

Команда **update** без опций полезна в тех случаях, когда во время долго выполняющихся вычислений необходимо обеспечить оперативную реакцию приложения на события, например, на действия пользователя. Вызов команды **update** и обеспечивает обработку таких событий.

uplevel

Команда выполняет скрипт в контексте, отличном от текущего.

Синтаксис

```
uplevel?level? arg?arg...?
```

Описание

Все аргументы команды объединяются как при выполнении команды **concat**. Получившийся скрипт выполняется в контексте, указанном **level**. Команда возвращает результат выполнения скрипта.

Если аргумент **level** задан как целое число, он указывает на сколько уровней выше уровня контекста текущей процедуры

надо подняться в стеке вызовов перед выполнением скрипта. Если аргумент **level** задан как символ # с последующим целым числом, то он задает абсолютный уровень контекста в стеке. Если аргумент **level** отсутствует, то используется значение по умолчанию **1**. Аргумент **level** должен быть указан, если первый из аргументов **arg** начинается с цифры или символа #.

При выполнении команды **uplevel** контекст вызывающей процедуры временно удаляется из стека вызовов процедур.

upvar

Команда создает связи между переменными различных уровней стека

Синтаксис

```
upvar?level? otherVar myVar?otherVar myVar...?
```

Описание

Команда позволяет одной или больше локальным переменным текущей процедуры ссылаться на переменные процедуры, стоящей выше в стеке, или на глобальные переменные. Аргумент **level** может иметь те же формы, что и в команде **uplevel**, или быть опущен, если первый символ в первой из **otherVar** отличен от цифры и от # (значение по умолчанию **1**). Для каждой пары аргументов **otherVar myVar** команда позволяет сделать переменную с именем **otherVar** из указанного уровня стека (локальную переменную одной из вызывающих процедур или глобальную переменную, если **level** равно **#0**) видимой в исполняемой процедуре под именем **myVar**. Переменная с именем **otherVar** не обязана существовать в момент исполнения команды. При необходимости она будет создана при первом использовании переменной **myVar**. В момент исполнения команды не должно быть доступной переменной **myVar**. Переменная **myVar** всегда считается простой переменной (не массивом и не переменной массива). Даже если

значение **myVar** выглядит как имя элемента массива, например, **a(b)**, создается простая переменная. Значение **otherVar** может быть именем простой переменной, массива или элемента массива. Команда **upvar** всегда возвращает пустую строку.

variable

Команда создает и запускает переменные области имен.

Синтаксис

```
variable?namevalue…?  name?value?
```

Описание

Обычно команду **variable** выполняют внутри команды **namespace eval** для создания одной или нескольких переменных в области имен. Каждая переменная **name** получает начальное значение **value**. Значение для последней переменной можно не указывать.

Если переменная **name** не существует, она будет создана. Если указан аргумент **value**, то переменной присвоится его значение. Если аргумент не указан, то новая переменная останется неопределенной. Если же переменная уже существовала, она сохранит свое значение. Обычно имя создаваемой переменной – это простое имя, не содержащее имя пространств имен. Соответственно переменная создается в текущем пространстве имен. Если имя содержит имена пространств имен, переменная создается в указанном пространстве имен.

Если команда **variable** выполняется внутри Tcl процедуры, она создает локальную переменную, связанную с соответствующей переменной пространства имен. В этом случае команда **variable** напоминает команду **global**, которая, однако, только связывает локальную переменную с глобальной. Если аргумент **value** задан, то он используется для изменения значения соответствующей переменной в пространстве имен.

Если переменная в пространстве имен не существует, она создается и, при необходимости, инициализируется.

Параметр **name** не может указывать на элемент массива. В команде **variable** можно указать только массив в целом, а затем присвоить значения его элементам командами **set** или **array**.

vwait

Команда задает обработку событий до тех пор, пока не будет записано значение переменной.

Синтаксис

```
vwait?varName?
```

Описание

Команда **vwait** активизирует обработчик событий (**event loop**), блокируя приложение до тех пор, пока не в результате какого-либо события не будет присвоено новое значение переменной **varName**. После присвоения значения переменной **varName** команда **vwait** завершит работу сразу после выполнения скрипта, вызванного обработчиком событий.

В некоторых ситуациях команда не завершается сразу после присвоения значения переменной **varName**. Это происходит, например, если вызванный по события скрипт, присвоивший новое значение переменной **varName**, не завершается сразу. Например, если в нем в свою очередь выполняется команда **vwait**, устанавливающая режим ожидания изменения другой переменной. Во время этого ожидания вышестоящая команда **vwait** блокируется как и приложение до выполнения соответствующего события.

while

Команда выполняет скрипт до тех пор, пока не будет выполнено условие.

Синтаксис

```
while test body
```

Описание

Команда **while** вычисляет значение выражения **test** подобно команде **expr**. Значение должно быть булевого типа. Если результат есть «истина», то скрипт **body** передается на выполнение Tcl интерпретатору. После этого выражение **test** снова вычисляется, и процесс повторяется до тех пор, пока его значение не станет «ложь». В тексте скрипта **body** можно использовать команду **continue** для завершения текущего цикла и команду **break** для немедленного завершения команды **while**.

Команда **while** всегда возвращает пустую строку.

Выражение **test** почти всегда лучше заключать в фигурные скобки, иначе подстановки команд и переменных в нем будут выполнены до исполнения команды и никакие изменения значений переменных в скрипте **body** не изменят значения выражения **test**. Это может привести к возникновению бесконечного цикла. Если же выражение **test** помещено в фигурные скобки, подстановки в нем выполняются при каждом вычислении (перед каждым выполнением **body**).

Список используемой литературы

Что такое Tcl/Tk?
© PC Week/RE
© Сергей Бобровский 1997

Как стать знаменитым
Copyright © 1998 Планета КИС
E-Mail: kis@pcweek.ru

Содержание

Ведение

Основные понятия и элементы

Стандартные интерпретаторы

Встроенные команды Tcl

Научно-популярное издание

Петровский Алексей Игоревич

Командный язык программирования Tcl (Tool Command Language)

Подписано в печать 13.02.06.
Формат 60x84/16. Бумага газетная. Печать офсетная.
Кол-во п.л. 12. Тираж 4000 экз.
Заказ

ООО «Литературное агентство «Бук-Пресс».
127591, Москва, Керамический пр., д. 53. кор. 1.
http://www.book-press.ru