ЛАБОРАТОРНАЯ РАБОТА №9 ОБРАБОТКА СТРОК (РАБОТА С ТЕКСТОВЫМИ ДАННЫМИ)

Цель работы – практическое знакомство со способами эффективной обработки текста при помощи интерфейса командной строки и набора стандартных утилит

1 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Ввод и вывод. Перенаправление ввода и вывода

Каждая программа работает с данными определенного типа: текстовыми, графическими, звуковыми и т. п. Основной интерфейс управления системой в Linux - это *терминал*, который предназначен для передачи текстовой информации от пользователя системе и обратно. Поскольку ввести с *терминала* и вывести на *терминал* можно только текстовую информацию, то ввод и вывод программ, связанных с терминалом, тоже должен быть текстовым. Однако необходимость оперировать с текстовыми данными не ограничивает возможности управления системой, а, наоборот, расширяет их. Пользователь может прочитать вывод любой программы и проанализировать, что происходит в системе, а разные программы оказываются совместимыми между собой, поскольку используют один и тот же вид представления данных - текстовый.

"Текстовость" данных - всего лишь договоренность об их формате. Никто не мешает выводить на экран нетекстовый файл, однако пользы в этом будет мало. Во-первых, раз уж файл содержит не текст, то не предполагается, что пользователь сможет что-либо понять из его содержимого. Во-вторых, если в нетекстовых данных, выводимых на *терминал*, случайно встретится *управляющая последовательность*, *терминал* ее выполнит. Например, если в скомпилированной программе записано некоторое число в виде четырех байтов: 27, 91, 49 и 74, соответствующий им **текст** состоит из четырех *символов* ASCII: "esc", "[", "1" и "J", и при *выводе* файла на виртуальную консоль Linux в этом месте выполнится очистка экрана, так как "^[[1J" - именно такая *управляющая последовательность* для виртуальной консоли. Не все *управляющие последовательности* столь безобидны, поэтому использовать нетекстовые данные в качестве текстов не рекомендуется.

Если содержимое нетекстового файла все-таки желательно просмотреть (то есть превратить в **текст**), можно воспользоваться утилитой **hexdump** с ключом **-C**, которая выдает содержимое файла в виде шестнадцатеричных ASCII-кодов, или **strings,** показывающей только те части файла, которые **могут** быть представлены в виде текста:

Пример 1. Использование hexdump

```
[student@localhost root]$ hexdump -C /bin/cat | less
  . . .
[student @localhost root]$ strings -n3 /bin/cat | less
```

В приведенном примере 7.1 утилита hexdump с ключом "-C" выводит в правой стороне экрана текстовое представление данных, заменяя непечатные *символы* точками (чтобы среди выводимого текста не встретилось *управляющей последовательности*). Наименьшая длина строки передается strings ключом "-n".

Для того чтобы записать данные в файл или прочитать их оттуда, процессу необходимо сначала открыть этот файл (при открытии на запись, возможно, придется предварительно создать его). При этом процесс получает **дескриптор** (описатель) открытого файла - уникальное для этого процесса число, которое он и будет использовать во всех операциях записи. Первый открытый файл получит дескриптор 0, второй - 1 и так далее. Закончив работу с файлом, процесс закрывает его, при этом дескриптор освобождается и может быть использован повторно. Если процесс завершается, не закрыв файлы, за него это делает система. Строго говоря, только в операции открытия дескриптора указывается, какой именно файл будет задействован. В качестве "файла" используются и обычные файлы, и устройства (чаще всего - *терминалы*), и *каналы*. Дальнейшие операции - чтение, запись и закрытие - работают с дескриптором, как с потоком данных, а куда именно ведет этот поток, неважно.

Каждый процесс Linux получает при старте три "файла", открытых для него системой. Первый из них (дескриптор 0) открыт на чтение, это стандартный *ввод* процесса. Именно со стандартным *вводом* работают все операции чтения, если в них не указан дескриптор файла. Второй (дескриптор 1) - открыт на **запись**, это *стандартный вывод* процесса. С ним работают все операции записи, если дескриптор файла не указан в них явно. Наконец, третий поток данных (дескриптор 2) предназначается для *вывода диагностических сообщений*, он называется ***стандартный вывод ошибок***. Поскольку эти три дескриптора уже открыты к моменту запуска процесса, первый файл, открытый **самим** процессом, будет, скорее всего, иметь дескриптор 3.

Дескриптор - это описатель потока данных, открытого процессом. Дескрипторы нумеруются, начиная с 0. При открытии нового потока данных его дескриптор получает наименьший из неиспользуемых в этот момент номеров. Три заранее открытых дескриптора - стандартный *ввод* (0), *стандартный вывод* (1) и *стандартный вывод ошибок* (2) - выдаются при запуске.

Механизм копирования *окружения* подразумевает, в числе прочего, копирование всех открытых дескрипторов родительского процесса

дочернему. В результате и родительский, и дочерний процесс имеют под одинаковыми дескрипторами одни и те же потоки данных. Когда запускается *стартовый командный интерпретатор*, все три заранее открытых дескриптора связаны у него с *терминалом* (точнее, с соответствующим устройством типа tty): пользователь вводит команды с клавиатуры и видит сообщения на экране. Следовательно, любая команда, запускаемая из командной оболочки, будет выводить на тот же *терминал*, а любая команда, запущенная интерактивно (не в фоне) - вводить оттуда.

Стандартный вывод

Стандартный вывод (standard output, stdout) - это поток данных, открываемый системой для каждого процесса в момент его запуска и предназначенный для данных, выводимых процессом.

Некоторые утилиты умеют выводить не только на *терминал*, но и в файл. Например, `info` при указании ключа "`-o`" с именем файла выведет текст руководства в файл, вместо того, чтобы отображать его на мониторе. Даже если разработчиками программы не предусмотрен такой ключ, известен и другой способ сохранить *вывод* программы в файле вместо того, чтобы выводить его на монитор: поставить знак ">" и указать после него имя файла.

Подмена *стандартного вывода* - задача командной оболочки (shell). Shell создает пустой файл, имя которого указано после знака ">", и дескриптор этого файла передается программе под номером 1 (*стандартный вывод*). Делается это очень просто. При запуске программы из оболочки после выполнения fork() появляется два одинаковых процесса, один из которых - дочерний - должен запустить вместо себя команду (выполнить exec()). Перед этим он **закрывает стандартный вывод** (дескриптор 1 освобождается) и **открывает** файл (с ним связывается первый свободный дескриптор, т. е. 1), а запускаемой команде ничего знать и не надо: ее *стандартный вывод* уже подменен. Эта операция называется **перенаправлением стандартного вывода**. В том случае, если файл уже существует, shell запишет его заново, полностью уничтожив все, что в нем содержалось до этого. Поэтому, чтобы продолжить записывать данные в textfile, потребуется другая операция - ">>":

Стандартный ввод

Стандартный ввод (standard input, stdin) - поток данных, открываемый системой для каждого процесса в момент его запуска и предназначенный для *ввода* данных.

Для передачи данных на вход программе может быть использован *стандартный ввод* (сокращенно - stdin). При работе с командной строкой *стандартный ввод* - это *символы*, вводимые пользователем с клавиатуры. *Стандартный ввод* можно **перенаправить** при помощи командной

оболочки, подав на него данные из некоторого файла. *Символ* "<" служит для перенаправления содержимого файла на *стандартный ввод* программе. Например, если вызвать утилиту sort без параметра, она будет читать строки со *стандартного ввода*. Команда "sort < имя_файла" подаст на *ввод* sort данные из файла:

Результат действия этой команды аналогичен команде sort textfile - разница лишь в том, что когда используется "<", sort получает данные со *стандартного ввода* ничего не зная о файле "textfile", откуда они поступают. Механизм работы shell в данном случае тот же, что и при перенаправлении *вывода*: shell читает данные из файла "textfile", запускает утилиту sort и передает ей на *стандартный ввод* содержимое файла.

Необходимо помнить, что операция ">" **деструктивна**: она всегда создает файл нулевой длины. Поэтому для, допустим, сортировки данных **в файле** надо применять последовательно sort < файл > новый_файл и mv новый_файл файл. Команда вида команда < файл > тот_же_файл просто урежет его до нулевой длины!

Стандартный вывод ошибок

Для *диагностических сообщений*, информирующих пользователя о ходе выполнения работы: а также для сообщений об ошибках, возникших в ходе выполнения программы, в Linux предусмотрен *стандартный вывод ошибок* (сокращенно - stderr).

Стандартный вывод ошибок (standard error, stderr) - поток данных, открываемый системой для каждого процесса в момент его запуска и предназначенный для *диагностических сообщений*, выводимых процессом.

Использование *стандартного вывода ошибок* наряду со *стандартным выводом* позволяет отделить собственно результат работы программы от разнообразной сопровождающей информации, например, направив их в разные файлы. Стандартный вывод ошибок может быть перенаправлен так же, как и *стандартный ввод/вывод*, для этого используется комбинация *символов* "2>", например info cat > cat.info 2> cat.stderr

На терминал в этом случае ничего не попадет - *стандартный вывод* отправится в файл cat.info, *стандартный вывод ошибок* - в cat.stderr. Вместо ">" и "2>" можно было бы написать "1>" и "2>". Цифры в данном случае обозначают номера дескрипторов открываемых файлов. Если некая утилита ожидает получить открытый дескриптор с номером, допустим, 4, то, для того чтобы ее запустить, **обязательно** потребуется использовать сочетание "4>".

Иногда, однако, требуется объединить *стандартный вывод* и *стандартный вывод ошибок* в одном файле, а не разделять их. В командной оболочке bash для этого имеется специальная последовательность "2>&1". Это означает "направить *стандартный вывод*

ошибок туда же, куда и *стандартный вывод*":

Перенаправление в никуда

Иногда заведомо известно, что какие-то данные, выведенные программой, не понадобятся. Например, предупреждения со *стандартного вывода ошибок*. В этом случае можно перенаправить *стандартный вывод ошибок* на устройство, специально предназначенное для уничтожения данных - /dev/null. Все, что записывается в этот файл, просто будет выброшено и **нигде не сохранится**:

Пример 2. Перенаправление в /dev/null

```
[student@localhost root]$ info cat > cat.info 2> /dev/null
```

Точно таким же образом можно избавиться и от *стандартного вывода*, отправив его в /dev/null.

1.2 Обработка данных в потоке. Конвейер

Нередко возникают ситуации, когда нужно обработать *вывод* одной программы какой-то другой программой. Для решения подобной задачи в bash предусмотрена возможность перенаправления *вывода* можно не только в файл, но и **непосредственно** на *стандартный ввод* другой программе. В Linux такой способ передачи данных называется ***конвейер***.

В bash для перенаправления *стандартного вывода* на *стандартный ввод* другой программе служит *символ* "|". Самый простой и наиболее распространенный случай, когда требуется использовать *конвейер*, возникает, если *вывод* программы не умещается на экране монитора и очень быстро "пролетает" перед глазами, так что человек не успевает его прочитать. В этом случае можно направить *вывод* в программу просмотра (less).

Можно последовательно обработать данные несколькими разными программами, перенаправляя *вывод* на *ввод* следующей программе и организуя сколь угодно длинный *конвейер* для обработки данных. В результате получаются командные строки вида "cmd1 | cmd2 | ... | cmdN".

Организация *конвейера* устроена в shell по той же схеме, что и перенаправление в файл, но с использованием особого объекта системы - *канала*. Можно представить трубу, немедленно доставляющую данные от входа к выходу (английский термин - "pipe"). *Каналом* пользуются сразу два процесса: один пишет туда, другой читает. Связывая две команды *конвейером*, shell открывает *канал* (заводится **два** дескриптора - входной и выходной), подменяет по уже описанному алгоритму *стандартный вывод* первого процесса на входной дескриптор *канала*, а *стандартный ввод* второго процесса - на выходной дескриптор *канала*. После чего остается запустить по команде в этих процессах, и *стандартный вывод* первой попадет на *стандартный ввод* второй.

Канал (pipe) - неделимая пара дескрипторов (входной и выходной), связанных друг с другом таким образом, что данные, записанные во входной дескриптор, будут немедленно доступны на чтение с выходного дескриптора.

1.3 Фильтры

Если программа и вводит данные, и выводит, то ее можно рассматривать как трубу, в которую что-то входит и из которой что-то выходит. Обычно смысл работы таких программ заключается в том, чтобы определенным образом **обработать** поступившие данные. В Linux такие программы называют **фильтрами**: данные проходят через них, причем что-то "застревает" в *фильтре* и не появляется на выходе, а что-то изменяется, что-то проходит сквозь *фильтр* неизменным. *Фильтры* в Linux обычно по умолчанию читают данные со *стандартного ввода*, а выводят на *стандартный вывод*. Простейший *фильтр* - программа cat: собственно, никакой "фильтрации" данных она не производит, она просто копирует *стандартный ввод* на *стандартный вывод*.

Данные, проходящие через *фильтр*, представляют собой текст: в стандартных потоках *ввода-вывода* все данные передаются в виде *символов*, строка за строкой, как и в *терминале*. Поэтому могут быть состыкованы при помощи *конвейера ввод* и *вывод* любых двух программ, поддерживающих стандартные потоки *ввода-вывода*.

В любом дистрибутиве Linux присутствует набор стандартных утилит, предназначенных для работы с файловой системой и обработки текстовых данных. Это who, cat, ls, pwd, cp, chmod, id, sort и др. Каждая из этих утилит предназначена для исполнения какой-то **одной** операции над файлами или текстом: *вывод* списка файлов в каталоге, копирование, сортировка строк, хотя каждая утилита может выполнять свою функцию по-разному, в зависимости от переданных ей ключей и параметров. При этом все они ориентированы на работу с данными в текстовой форме, многие являются *фильтрами*, не имеют графического интерфейса, вызываются из командной строки и работают со стандартными потоками *ввода*/*вывода*, поэтому хорошо приспособлены для построения *конвейеров*.

1.4 Структурные единицы текста

Работу в системе Linux почти всегда можно представить как работу с текстами. Поиск файлов и других объектов системы - это получение от системы текста **особой** структуры - списка имен. Операции над файлами - создание, переименование, перемещение, а также сортировка, перекодировка и прочее - замену одних *символов* и строк другими либо в

каталогах, либо в самих файлах. Работая с текстом в Linux, нужно принимать во внимание, что текстовые данные, передаваемые в системе, структурированы. Большинство утилит обрабатывает не непрерывный поток текста, а последовательность **единиц**. В текстовых данных в Linux выделяются следующие структурные единицы:

Строки

Строка - основная единица передачи текста в Linux. *Терминал* передает данные от пользователя системе строками (командная строка), множество утилит вводят и выводят данные построчно, при работе многих утилит одной строке соответствует один объект системы (имя файла, путь и т. п.), sort сортирует строки. Строки разделяются *символом* конца строки "\n" (newline).

Поля

В одной строке может упоминаться и больше одного объекта. Если понимать объект как последовательность *символов* из определенного набора (например, букв), то строку можно рассматривать как состоящую из слов и разделителей. В этом случае текст от начала строки до первого *разделителя* - это первое *поле*, от первого *разделителя* до второго - второе *поле* и т. д. В качестве *разделителя* можно рассматривать любой *символ,* который не может использоваться в объекте. Например, если в пути "/home/student" *разделителем* является *символ* "/", то первое *поле* пусто, второе содержит слово "home", третье - "student". Некоторые утилиты позволяют выбирать из строк отдельные *поля* (по номеру) и работать со строками как с таблицей.

Символы

Минимальная единица текста - *символ*. ***Символ*** - это одна буква или другой письменный знак. Стандартные утилиты Linux позволяют заменять одни *символы* другими (производить транслитерацию), искать и заменять в строках *символы* и комбинации *символов*.

Символ конца строки в кодировке ASCII совпадает с *управляющей последовательностью* "^J" - "перевод строки", однако в других кодировках он может быть иным. Кроме того, на большинстве *терминалов* - но не на всех! - вслед за переводом строки необходимо выводить еще *символ* возврата каретки ("^M"). Это вызвало путаницу: некоторые системы требуют, чтобы в конце текстового файла стояли **оба этих символа** в определенном порядке. Чтобы избежать путаницы, в Linux было принято единственно верное решение: содержимое файла соответствует кодировке, а при *выводе* на *терминал* концы строки преобразуются в *управляющие последовательности* согласно настройке *терминала*.

В распоряжении пользователя Linux есть ряд утилит, выполняющих элементарные операции с единицами текста: поиск, замену, разделение и объединение строк, *полей, символов*. Эти утилиты, как правило, имеют одинаковое представление о том, как определяются единицы текста: что такое строка, какие *символы* являются *разделителями* и т. п. Во многих случаях их представления можно изменять при помощи настроек. Поэтому такие утилиты легко взаимодействуют друг с другом. Комбинируя их, можно автоматизировать довольно сложные операции по обработке текста.

1.5 Регулярные выражения

Регулярными выражениями называются особым образом составленные наборы символов, выделяющие из текста нужное сочетание слов или символов, которое соответствует признакам, отраженным в регулярном выражении. Иными словами, регулярное выражение — это **фильтр для текста.**

В Linux регулярные выражения используются командой grep, которая позволяет искать файлы с определенным содержанием либо выделять из файлов строки с необходимым содержимым (например, номера телефонов, даты и т. д.). Многие программы, так или иначе работающие с текстом, (текстовые редакторы), поддерживают *регулярные выражения*. К таким программам относятся два "главных" текстовых редактора Linux - Vim и Emacs. Однако нужно учитывать, что в разных программах используются разные диалекты языка *регулярных выражений*, где одни и те же понятия имеют разные обозначения, поэтому **всегда** нужно обращаться к руководству по конкретной программе.

1.5.1 Элементарные регулярные выражения

Элементарная структурная единица регулярного выражения — это символ. Текст можно искать по определенному набору букв и цифр. Рассмотрим пример, в котором с помощью регулярного выражения выделим из данных строк те, которые содержат сочетание букв bc (именно в этом порядке):

Исходный набор строк:
abc
abcd
dcba
adbc
Регулярное выражение:
bc
Результат:
abc
abcd

adbc

1.5.2 Конструкция вида [...]

Рассмотрим другой пример, заменив некоторые буквы в строках на заглавные:

Исходный набор строк:
abC
abcd
dcba
adBc
Регулярное выражение:
bc
Результат: abcd

Результат обработки строк с помощью регулярного выражения изменился. При использовании вышеуказанного регулярного выражения в большинстве случаев чаще всего различают строчные и прописные буквы, что логически обосновано, если не указан соответствующий параметр, предписывающий не различать их. Программы, которые работают с регулярными выражениями чаще всего различают строчные и прописные буквы, если не указана соответствующая опция, предписывающая не различать строчные и прописные буквы.

В результате получается всего одна строка — вторая. Для вывода трех строк, как в первом примере, понадобится **особая конструкция**. Рассмотрим ее.

В основе данной конструкции лежат две квадратные скобки (открывающая и закрывающая), внутри которых расположены символы либо конструкции (последний случай будет описан далее), один из которых может быть на месте этой конструкции в итоговом выражении. Изменим регулярное выражение в предыдущем примере. Теперь задачей является сделать это регулярное выражение более универсальным, чтобы с его помощью можно было найти в исходном наборе строк сочетание bc независимо от того, в каком регистре находятся буквы в конечных выражениях.

Исходный набор строк:
abC
abcd
dcba
adBc
Регулярное выражение:
[Bb][Cc]
Результат:
abC
abcd

adBc

Теперь рассмотрим такой пример: с помощью регулярного выражения необходимо выделить из указанных в предыдущем примере строк те, которые содержат некоторую букву английского алфавита в нижнем регистре и сразу за ней — букву с в нижнем или верхнем регистре. При применении способа перечисления для решения поставленной задачи получится следующее регулярное выражение:
[abcdefghijklmnopqrstuvwxyz] [Cc]

Это верно, но строка получилась длинной, что особенно неудобно при составлении больших регулярных выражений. В подобных случаях можно перечислить все эти 26 символов короче, используя интервалы, то есть указать начальный и конечный символы, поставив между ними знак «тире». Рассмотрим пример:

Исходный набор строк:
abC
abcd
dcba
adBc

Регулярное выражение:
[a-z][Cc]

Результат:
abC
abcd
dcba

Здесь a-z — это и есть нужный интервал. Можно изменить пример так, чтобы первый символ мог быть как строчным, так и прописным, для чего сразу после первого интервала указываем второй:

Исходный набор строк:
abC
abcd
dcba
adBc

Регулярное выражение:
[a-zA-Z][Cc]

Результат:
abC
abcd
dcba
adBc

То же касается и цифр. Границами интервала могут быть любые символы, но последовательности типа [z-a] и [5-1] смысла иметь не будут, так как ASCII-код первого символа должен быть меньше либо равен коду

завершающего. По поводу интервалов с цифрами следует напомнить, что символ нуля идет раньше символов всех остальных цифр. Количество стоящих рядом последовательностей неограниченно.

В этой же конструкции можно обратить (или, как еще говорят, инвертировать) выбор символов, поставив после знака открывающей квадратной скобки символ ^, после чего на месте конструкции будут предполагаться все символы, кроме указанных в ней самой. Рассмотрим это на предыдущем примере.

Из данного набора строк выделим только те, в которых буква b стоит перед любым символом, кроме буквы c.

Исходный набор строк:

abC
abcd
dcba
adBc

Регулярное выражение:

[Bb][^Cc]

Результат:

dcba

1.5.3 Метасимволы

Не все символы можно использовать прямо по назначению. Посмотрите, например, на конструкцию, которая описывалась в предыдущем разделе. Допустим, требуется найти в каком-то файле строки, содержащие следующий набор символов: abc[def. Можно предположить, что регулярное выражение будет составлено по принципам, описанным выше, но это неверно. Открывающая квадратная скобка — это один из символов, который несет для программы, работающей с регулярными выражениями, особый смысл (который был рассмотрен ранее). Такие символы называются **метасимволами.**

Метасимволы бывают разными и служат для различных целей. Из предыдущего материала можно выделить метасимволы открывающей и закрывающей квадратных скобок, а также символ ^. Символ «тире» не рассматривается как метасимвол, так как он имеет особое значение только внутри конструкции с квадратными скобками, а вне такой конструкции специально не применяется.

Рассмотрим те метасимволы, которые предполагают, что в конечном выражении на их месте будет стоять какой-либо символ или символы (табл. 1).

Таблица 1.

Знакозаменяющие метасимволы

Метасим-вол	Описание метасимвола
.(точка)	Предполагает, что в конечном выражении на ее месте будет стоять любой символ. Продемонстрируем это на примере набора английских слов: **Исходный набор строк:** wake make machine cake maze **Регулярное выражение:** ma.e **Результат:** make maze
\w	Замещает любые символы, которые относятся к буквам, цифрам и знаку подчеркивания. Пример: **Исходный набор строк:** abc a$c a1c a c **Регулярное выражение:** a\wc **Результат:** abc a1c
\W	Замещает все символы, кроме букв, цифр и знака подчеркивания (то есть является обратным метасимволу \w). Пример: **Исходный набор строк:** abc a$c a1c a c **Регулярное выражение:** a\Wc **Результат:** a$c a c

Метасим-вол	Описание метасимвола
\d	Замещает все цифры. Продемонстрируем его действие на том же примере: **Исходный набор строк:** abc a$c a1c a c **Регулярное выражение:** a\dc **Результат:** alc
\D	Замещает все символы, кроме цифр, например: **Исходный набор строк:** abc a$c alc a c
\D	**Регулярное выражение:** a\Dc **Результат:** abc a$c a c
[\b]	Замещает символ перевода курсора на один влево (возврат курсора)
\r	Замещает символ перевода курсора в начало строки
\n	Замещает символ переноса курсора на новую строку
\t	Замещает символ горизонтальной табуляции
\v	Замещает символ вертикальной табуляции
\f	Замещает символ перехода на новую страницу
\s	Равнозначен использованию пяти последних метасимволов, то есть вместо метасимвола \s можно написать [\r\n\t\v\f], что, однако, не так удобно
\S	Является обратным метасимволу \ s

Для лучшего понимания рассмотрим, что такое символ перевода курсора в начало строки, переноса курсора на новую строку и т. д. В конце каждой строки находятся один или два символа, которые указывают на необходимость перехода на новую строку. Начиная с операционной системы MS-DOS и до настоящего времени в операционных системах Microsoft используются два символа для обозначения переноса строки — сам символ переноса и символ возврата курсора в начало строки

(заменяются метасимволами \n и \r соответственно), хотя в отдельности эти символы почти не применяются. В Linux используется только символ перехода на новую строку (заменяется метасимволом \n). Это следует учесть при составлении регулярных выражений.

Рассмотрим интересную группу символов, для чего поставим следующую задачу. Количество символов, которые должны быть в конечном тексте, не всегда известно (примером может быть распознавание имени веб-сайта), поэтому при помощи вышеописанных конструкций и метасимволов написать действительно универсальные регулярные выражения невозможно. Рассмотрим еще одну группу символов, которые помогут решить подобные проблемы. Они используются сразу после символа, метасимвола либо конструкции, количество вхождения которых они должны описать (табл. 2).

Таблица 2.

Метасимволы количества повторений

Символ	Описание метасимвола
?	Указывает обработчику регулярных выражений на то, что предыдущий символ, метасимвол или конструкция могут вообще не существовать в конечном тексте либо присутствовать, но иметь не более одного вхождения. Рассмотрим пример. Из данного набора строк требуется найти только те, в которых символу c может (но не обязательно) предшествовать один символ a, перед чем должен стоять символ b*:* **Исходный набор строк:** acbd aabc caab ecad bcde abac **Регулярное выражение:** b[Aa]?c **Результат:** aabc bcde abac
*****	Означает, что впереди стоящие символ, метасимвол либо конструкция могут как отсутствовать, так и быть в конечном выражении, причем количество вхождений неограниченно. **Пример**: из данного набора строк выделим те, в которых есть по крайней мере две буквы a, между которыми может быть либо отсутствовать некоторое количество цифр: **Исходный набор строк:** a123a

Символ	Описание метасимвола
	a12a al23b alc3b b12a aaa **Регулярное выражение:** [Aa]\d*[Aa] **Результат:** al23a al2a aaa
+	Действие схоже с действием предыдущего символа с тем отличием, что впередистоящие метасимвол, символ или конструкция должны повторяться как минимум один раз. Рассмотрим предыдущий пример, но чтобы между буквами а была хотя бы одна цифра: **Исходный набор строк:** a123a a12a al23b alc3b b12a aaa **Регулярное выражение:** [Aa]\d+[Aa] **Результат:** a123a a12a
{min, max}	Иногда первых трех способов указания количества вхождений бывает недостаточно, так как они описывают количество не детально. Решить проблему можно следующим способом. Для указания количества вхождений символа либо конструкции после них ставят открывающие фигурные скобки и пишут минимальное количество вхождений. Если это количество фиксированное (то есть должно быть не больше и не меньше вхождений символа), то скобку закрывают; если должно быть не меньше указанного количества вхождений, то ставят запятую и закрывают скобку; если существует предельное количество вхождений, то после запятой указывают его и закрывают скобку. Так, эквивалентом знаку вопроса является конструкция {0,1}, знаку звездочки — { 0, }, знаку «плюс» — {1, }. Рассмотрим пример: **Исходный набор строк:** a123a a12a al23b alc3b b12a aaa **Регулярное выражение:**

Символ	Описание метасимвола
	[Aa]\d{2,}[Aa] **Результат:** a123a a12a

Символ \b

Описание метасимвола:

Играет большую роль при разборе выражений. Его функция заключается в следующем. Обычно программы, которые работают с регулярными выражениями (в том числе и grep), ищут сходные выражения в тексте, не определяя, является ли выражение словом и может ли быть расположено конечное выражение в начале или конце слова. Однако часто требуется найти именно конкретное слово, что позволяет сделать данный метасимвол. Он ставится на том месте, где слово должно начинаться или заканчиваться. Рассмотрим пример, в котором попытаемся в данном наборе слов найти начинающиеся на букву s и заканчивающиеся на букву r (для сравнения здесь будут приведены примеры регулярного выражения с использованием метасимвола /b и без него):

Исходный набор строк:

starfish
starless
stellar
ascender
sacrifice
scalar

Регулярное выражение:

[Ss]\w*[Rr]

Результат:

starfish
starless
stellar
a**scender**
sacrifice
scalar

Были выбраны слова, которые не соответствуют условиям. Изменим регулярное выражение, добавив метасимвол \b:

Регулярное выражение:

\b[Ss]\w*[Rr]\b

Результат:

stellar
scalar

1.5.4 Группировка выражений

Особым приемом при составлении регулярных выражений является группировка нескольких его составляющих в одну единицу. Рассмотрим пример, в котором требуется выделить из текста обычный телефонный номер в его записи без кода города, когда весь номер разбивается на три части, между которыми ставится дефис. Для этого подойдет такое регулярное выражение:

\d{l,3}-\d{l,3}-\d{l,3}

Это регулярное выражение можно сократить. В нем есть два идентичных блока текста, стоящих подряд. В этом случае всегда можно сделать выражение короче, заключив повторяющиеся фрагменты в круглые скобки и указав после них количество повторений. Это можно сделать любым способом. Вот один из вариантов:

(\d{1,3}-){2}\d{l,3}

Теперь все выражение, заключенное в скобки, считается единым целым.

Внутри скобок также можно использовать прием, который позволяет выбирать между несколькими выражениями. Вот простой пример. Дано несколько чисел. Необходимо с помощью регулярного выражения выделить из них те, в которых есть цифра 7, после которой находится одна из цифр — 1, 3 или 5. Задачу легко решить с помощью конструкции с квадратными скобками, однако сделаем по-другому. Для выбора между несколькими выражениями требуется заключить их в круглые скобки и поставить между каждыми двумя выражениями символ вертикальной черты. Посмотрим, как таким способом решить поставленную задачу.

Исходный набор строк:

123
178
176
755
713
873

Регулярное выражение:

(7(1|3|5))

Результат:

755
713
873

Данную задачу было бы правильнее решить с помощью конструкции с квадратными скобками, при использовании которой регулярное выражение получилось бы короче. Следует отметить, что в скобках можно также выполнять операцию группировки выражений неограниченное по

глубине количество раз, то есть выражение типа (a | (b | (c | d))) будет верным.

1.5.5 Использование зарезервированных символов

Выше упоминалось, что некоторые символы имеют для программы, работающей с регулярными выражениями, особый смысл. Это, например, косая черта, точка, круглая, фигурная и квадратная скобки, звездочка и т. д. Однако не исключено, что в целевом выражении также могут быть эти символы, и их наличие нужно будет определить в регулярном. В данном случае эти символы нужно указать особым образом («защитить» их). При этом перед нужным символом ставят косую черту, то есть, чтобы указать наличие в конечном тексте символа звездочки, в регулярном выражении на соответствующем месте следует написать *. Рассмотрим пример.

С помощью регулярного выражения требуется найти строки, где между некоторыми буквами или цифрами в круглые или квадратные скобки заключено несколько цифр.

Исходный набор строк:

ab(123)cd
a[12]d
al23]d
a(12d
l[123]d

Регулярное выражение:

\w+(\[|\()\d+(\] |\))\w+

Результат:

ab(123)cd
a[12]d
1[123]d

Это регулярное выражение несколько нелогично, так как позволяет сделать открывающей круглую скобку, а закрывающей — квадратную, и наоборот. Попробуйте придумать свое, более универсальное регулярное выражение, которое исправило бы этот недостаток.

1.5.6 Примеры использования регулярных выражений

Рассмотрим несколько примеров на основе изученного материала. В первом примере попытаемся из текста выделить IP-адреса. Следует напомнить, что правильным IP-адресом являются четыре числа, в каждом из которых содержится не более трех цифр, причем эти числа разделены точками. Регулярное выражение будет таким:

\d{l,3}\.\d{l,3}\.\d{l,3}\.\d{l,3}\

В нем содержится три одинаковых блока, стоящих один за другим. Их можно сгруппировать в один блок следующим образом:

(\d{l,3}\.){3}\d{l,3}

Однако это выражение также нельзя назвать верным: ни одно из чисел, входящих в состав IP-адреса, не может быть больше 255. По этой причине составить регулярные выражения не так просто. Следует разбирать каждое число посимвольно. Рассмотрим возможные варианты и регулярные выражения, соответствующие им (табл.3).

Таблица 3.

Перебор регулярных выражений для определения IP-адреса

Описание	Регулярное выражение
Один или два символа цифр. В таком случае эти цифры могут быть любыми	\d{l,2}
Три символа цифр, причем первая — нуль или единица. В таком случае две другие цифры могут быть любыми	(0|l)\d{2)
Три символа цифр, причем первая двойка, а вторая меньше пяти. В таком случае третья цифра может быть любой	2[0-4]\d
Три символа цифр, причем вторая — пятерка. Третья цифра должна быть меньше или равна пяти	25[0-5]

Осталось сгруппировать выражения, приведенные в таблице, в одно, которое позволило бы выделить из текста число, меньшее либо равное 255:
((\d{l,2})I{(0|l)\d{2}) | (2[0-4]\d) | (25[0-5]))

В итоге получаем регулярное выражение для поиска IP-адреса:
(((\d{1,2}) | ((0|1) \d{2}) | (2[0-4]\d) | (25[0-5]))\.) {3}
((\d{l,2}) | ((0|l)\d{2}) | (2[0-4]\d) | (25 [0-5]))

Продемонстрируем работу этого регулярного выражения на примере.

Пример:

Исходный набор строк:

127.0.0.1
255.255.255.255
12.34.56
123.256.0.0
1.23.099.255

Регулярное выражение:

(((\d{1,2}) | ((0 | l)\d{2}) | (2[0-4]\d) | (25 [0-5])) \.) {3}\d{l,3}

Результат:

127.0.0.1
255.255.255.255
1.23.099.255

Рассмотрим пример, связанный с выделением из текста электронных почтовых адресов (то есть адресов e-mail). Для начала определим все возможные варианты, которые должны быть отражены в регулярном выражении. Существует мнение, что не так сложно учесть все варианты, как не допустить, чтобы регулярное выражение соответствовало неправильному конечному тексту. Давайте определим все возможные варианты.

Самым обычным считается адрес типа: username@foobarwebsite.com.

Может показаться, что для выделения адреса e-mail будет достаточно такого регулярного выражения:

\w+@\w+\.\w+

Этого достаточно, чтобы распознать вышеуказанный формат адреса, но данное регулярное выражение не универсально. Следующий адрес корректен, однако вышеуказанное регулярное выражение распознать его не сможет:

my.user.name@sub.foobar.website.com.

В качестве тренировки подумайте, каким будет регулярное выражение. Ответ прост: чтобы регулярное выражение в данном случае было универсальным, следует учесть возможное наличие точек в имени пользователя и домена. Решение будет следующим:

(\w+\ .) *\w+@ (\w+\ .)+\w+

Это не совсем правильный вариант. В имени домена первого уровня могут быть только буквы, но не цифры или другие символы. В связи с этим можно изменить регулярное выражение следующим образом:

(\w+\.)*\w+@(\w+\.) + [A-Za-z]+

Продемонстрируем работу регулярного выражения на примере.

Исходный набор строк:

my@email.com
another.my@email.com
not.my@email.address.com
wrong.address.com
another.wrong.address.com

Регулярное выражение:

(\w+\.)*\w+@(\w+\.)+[A-Za-z]+

Результат:

my@email.com
another.my@email.com
not.my@email.address.com

Закончим еще одним классическим примером — распознаванием адреса страницы в Интернете. Определим критерии, по которым будем искать адрес:

- в начале выражения может быть **http://, https://** или **www.,** причем последнее выражение, если оно существует, должно быть позже двух

первых, а http:// и https:// не могут присутствовать одновременно; составными частями выражения могут быть буквы, цифры и знаки подчеркивания, причем эти составные части разделяются косыми чертами;

- в конце выражения может стоять косая черта или имя файла (выделяем только адрес страницы и не учитываем случай, когда передаются какие-либо параметры).

На основе этого попытаемся составить регулярное выражение. Разобьем его на составляющие и посмотрим, что описывает каждая его часть (табл. 4).

Таблица 4.

Части, составляющие адрес страницы

Выражение	Описание
(https?://)?	Здесь предположим, что в конечном тексте может быть http:// либо https://
(www\.)?	Предполагаем, что в конечном тексте может быть www
(\w+\.) +	С помощью этой фразы выделяются имена доменов второго и последующих уровней
[A-Za-z]+	Здесь выделяем имя домена первого уровня. Это не совсем верное решение, так как на данный момент существуют только домены первого уровня, длина которых не превышает четырех символов, и при пользовании этой фразой есть шанс вместе с адресами сайтов выделить «мусор»
(/+\w+)*	Предполагаем, что может быть также указан путь к определенному каталогу
(\.\w+)?	Здесь предполагаем, что в адресе может стоять имя файла. В этой фразе учитываем только возможность наличия расширения, так как само имя файла будет учтено предыдущей

В результате получим регулярное выражение:
(https?://) ?(www\.)?(\w+\.) + [A-Za-z] + (/ + \w+)* (\.\w+) ?

Теперь, применив выражение к некоторому тексту, вы сможете выделить из него адреса страниц.

1.6 Примеры задач

Этот раздел посвящен нескольким примерам использования стандартных утилит для решения разных типичных (и не очень) задач. Примеры не следует воспринимать как исчерпывающий список возможностей, они приведены просто для демонстрации того, как можно организовать обработку данных при помощи *конвейера*. Чтобы освоить их, нужно читать руководства и экспериментировать.

1.6.1 Подсчет

Часто бывает необходимо подсчитать количество определенных элементов текстового файла. Для подсчета строк, слов и символов служит стандартная утилита - wc (от англ. "word count" - "подсчет слов"). Используя текстовый вывод утилит, можно посчитать свои файлы:

Пример 3. Подсчет файлов при помощи find и wc

```
[student@localhost root]$ find . | wc -l
42
```

Для подсчета файлов в примере использована команда find - инструмент поиска нужных файлов в системе. Команда find вызвана с одним параметром - каталогом, с которого надо начинать поиск. find выводит список найденных файлов по одному на строку, а поскольку критерии поиска в данном случае не уточнялись, то find просто вывела список всех файлов во всех подкаталогах текущего каталога. Этот список передан утилите wc для посчета количества полученных строк "-l". В ответ wc выдала искомое число - "42".

Задав find критерии поиска, можно посчитать и что-нибудь менее тривиальное, например, файлы, которые создавались или были изменены в определенный промежуток времени, файлы с определенным режимом доступа, с определенным именем и т. п. Узнать обо всех возможностях поиска при помощи find и подсчета при помощи wc можно из руководств по этим программам.

1.6.2 Отбрасывание ненужного

Иногда пользователя интересует только часть из тех данных, которые собирается выводить программа. Утилита head нужна, чтобы вывести только первые несколько строк файла. Не менее полезна утилита tail (англ. "хвост"), выводящая только последние строки файла. Если же пользователя интересует только определенная часть каждой строки файла - поможет утилита cut.

Допустим, пользователю потребовалось получить список всех файлов и подкаталогов в "/etc", которые принадлежат группе "adm". При этом ему нужно, чтобы найденные файлы в списке были представлены не полным путем, а только именем файла (это требуется для последующей автоматической обработки полученного списка):

Пример 4. Извлечение отдельного поля

```
[student@localhost root]$ find /etc -maxdepth 1 -group adm 2> /dev/null \
> | cut -d / -f 3
syslog.conf
anacrontab
```

Если команда получается такой длинной, что ее неудобно набирать в одну строку, можно разбить ее на несколько строк, не передавая системе:

для этого в том месте, где нужно продолжить набор со следующей строки, достаточно поставить *символ* "\" и нажать Enter. При этом в начале строки bash выведет *символ* ">", означающий, что команда еще не передана системе и набор продолжается. Вид этого приглашения определяется значением переменной *окружения* "PS2". Для системы безразлично, в сколько строк набрана команда, возможность набора в несколько строк нужна только для удобства пользователя.

Пользователь получил нужный результат, задав параметры find - каталог, где нужно искать и параметр поиска - наибольшую допустимую глубину вложенности и группу, которой должны принадлежать найденные файлы. Ненужные *диагностические сообщения* о запрещенном доступе он отправил в /dev/null, а потом указал утилите cut, что в полученных со *стандартного ввода* строках нужно считать *разделителем символ* "/" и вывести только третье *поле*. Таким образом, от строк вида "/etc/filename" осталось только "filename" Как уже указывалось, первым *полем* считается текст от начала строки до первого *разделителя*; в приведенном примере первое *поле* - пусто, "etc" - содержимое второго *поля*, и т. д.

1.6.3 Выбор нужного. Поиск

Зачастую пользователю нужно найти только упоминания чего-то конкретного среди данных, выводимых утилитой. Обычно эта задача сводится к поиску строк, в которых встречается определенное слово или комбинация *символов*. Для этого подходит стандартная утилита grep, которая может искать строку в файлах, а может работать как *фильтр*: получив строки со *стандартного ввода*, она выведет на *стандартный вывод* только те строки, где встретилось искомое сочетание *символов*. В следующем примере анализируются процессы bash, которые выполняются в системе:

Пример 5. Поиск строки в выводе утилиты

```
 [student@ localhost root]$ ps aux | grep bash
student  3459 0.0 3.0 2524 1636 tty2  S  14:30  0:00 -bash
student  3734 0.0 1.1 1644  612 tty2  S  14:50  0:00 grep bash
```

Первый аргумент команды grep - та строка, которую нужно искать в *стандартном вводе*, в данном случае это "bash", а поскольку ps выводит сведения по строке на каждый процесс, на экран будут выведены только процессы, в имени которых есть "bash". Однако неожиданно в списке выполняющихся процессов получены две строки, в которых встретилось слово "bash", т. е. два процесса: один - искомый - командный интерпретатор bash, а другой - процесс поиска строки "grep bash", запущенный **после** ps. Это произошло потому, что после разбора командной строки bash запустил **оба** дочерних процесса, чтобы организовать *конвейер*, и на момент выполнения команды ps процесс grep bash уже был запущен и тоже попал в *вывод* ps. Чтобы в этом примере

получить правильный результат, необходимо добавить в *конвейер* еще одно звено: | grep -v grep, эта команда **исключит** из конечного *вывода* все строки, в которых встречается "grep".

1.6.4 Поиск по регулярному выражению

Очень часто точно не известно, какую именно комбинацию *символов* нужно будет найти. Точнее, известно только то, как примерно должно выглядеть искомое слово, что в него должно входить и в каком порядке. Так обычно бывает, если некоторые фрагменты текста имеют строго определенный формат. Например, в руководствах, выводимых программой info, принят такой формат ссылок: "*Note название_узла::". В этом случае нужно искать не конкретное сочетание *символов*, а "Строку "*Note", за которой следует название узла (одно или несколько слов и пробелов), оканчивающееся *символами* "::"". Утилита grep может выполнить такой запрос, если его сформулировать на языке регулярных выражений.

Пример 6. Поиск ссылок в файле info

```
[student@ localhost root]$ info grep > grep.info 2> /dev/null
[student@ localhost root]$ grep -on "\*Note[^:]*::" grep.info
324:*Note Grep Programs::
684:*Note Invoking::
[student@ localhost root]$
```

Первый параметр grep, который взят в кавычки - это и есть *регулярное выражение* для поиска ссылок в формате info, второй параметр - имя файла, в котором нужно искать. Ключ "-o" заставляет grep выводить строку не целиком, а только ту часть, которая совпала с *регулярным выражением* (шаблоном поиска), а "-n" - выводить номер строки, в которой встретилось данное совпадение.

В регулярном выражении большинство *символов* обозначают сами себя, как если бы мы искали обыкновенную текстовую строку, например, "Note" и "::" в регулярном выражении соответствуют строкам "Note" и "::" в тексте. Однако некоторые *символы* обладают специальным значением, самый главный из таких *символов* - звездочка ("*"), поставленная после элемента регулярного выражения, обозначает, что могут быть найдены тексты, где этот элемент повторен любое количество раз, в том числе и ни одного, т. е. просто отсутствует. В нашем примере звездочка встретилась дважды: в первый раз нужно было включить в регулярное выражение именно *символ* "звездочка", для этого потребовалось лишить его специального значения, поставив перед ним "\" (см. раздел 1.5).

Вторая звездочка обозначает, что стоящий перед ней элемент может быть повторен любое количество раз от нуля до бесконечности. В нашем случае звездочка относится к выражению в квадратных скобках - "[^:]", что означает "любой *символ*, кроме ":"". Целиком регулярное выражение можно прочесть так: "Строка "*Note", за которой следует ноль или больше

любых *символов*, кроме ":", за которыми следует строка "::"". Особенность работы "*" состоит в том, что она пытается выбрать совпадение максимальной длины. Именно поэтому элемент, к которому относилась "*", был задан как "не ":"". Выражение "ноль или более **любых символов**" (оно записывается как ".*") в случае, когда, например, в одной строке встречается **две** ссылки, вбирает подстроку от конца **первого** "*Note" до начала **последнего** "::" (*символы* ":", поместившиеся внутри этой подстроки, распознаются как "любые").

Регулярные выражения позволяют резко повысить эффективность работы, хорошо интегрированы в рабочую среду в системе Linux.

1.6.5 Замены

Удобство работы с потоком не в последнюю очередь состоит в том, что можно не только выборочно передавать результаты работы программ, но и автоматически заменять один текст другим прямо в потоке.

Для замены одних *символов* другими предназначена утилита tr (сокращение от англ. "translate" - "преобразовывать, переводить"), работающая как *фильтр*. В примере 7.7 утилита применена прямо по назначению – с ее помощью выполнена транслитерация - замена латинских *символов* близкими по звучанию русскими:

Пример 7. Замена символов (транслитерация)

```
 [student@ localhost root]$ cat cat.info | tr
abcdefghijklmnopqrstuvwxyz абцдефгхийклмнопкрстуввсиз \
> | tr ABCDEFGHIJKLMNOPRSTUVWXYZ
АБЦДЕФГХИЙКЛМНОПКРСТУВВСИЗ | head -4
Филе: цореутилс.инфо, Ноде: цат инвоцатион, Нест: тац инвоцатион,
Тп: Оутпут оф ентире филес
'цат': Цонцатенате анд врите филес
……
[student@ localhost root]$
```

Два параметра для утилиты tr: соответствия латинских букв кириллическим. Первый *символ* из первого параметра tr заменяет первым *символом* второго, второй - вторым и т. д. Пользователь обработал поток *фильтром* tr дважды: сначала чтобы заменить строчные буквы, а затем - прописные. Он мог бы сделать это и за один проход (просто добавив к параметрам прописные после строчных), но не захотел выписывать столь длинные строки. Полученному на выходе тексту вряд ли можно найти практическое применение, однако транслитерацию можно употребить и с пользой. Если не указать tr второго параметра, то все *символы*, перечисленные в первом, будут заменены на "ничто", т. е. попросту удалены из потока. При помощи tr можно также удалить дублирующиеся *символы* (например, лишние пробелы или переводы строки), заменить пробелы переводами строк и т. п.

1.7 Потоковый редактор sed

Помимо простой замены отдельных *символов*, возможна замена последовательностей (слов). Специально для этого предназначен потоковый редактор sed (сокращение от англ. «stream editor»). Он работает как фильтр и выполняет редактирование поступающих строк: замену одних последовательностей *символов* другими, причем можно заменять и регулярные выражения.

Например, с помощью sed можно сделать более понятным для непривычного читателя список файлов, выводимый ls:

Пример 8. Замена по регулярному выражению

```
[student@ localhost root]$ ls –l | sed s/^-[-rwx]*/Файл:/ | sed s/^d[-
rwx]*/Каталог:/
итого 124
Файл: 1 student 2693 Ноя 15 16:09 cat.info
Файл: 1 student   69 Ноя 15 16:08 cat.stderr
Каталог: 2 student 4096 Ноя 15 12:56 Documents
Каталог: 3 student 4096 Ноя 15 13:08 examples
Файл: 1 student 83459 Ноя 15 16:11 grep.info
Файл: 1 student    26 Ноя 15 13:08 loop
Файл: 1 student   23 Ноя 15 13:08 script
Файл: 1 student    33 Ноя 15 16:07 textfile
Каталог: 2 student 4096 Ноя 15 12:56 tmp
Файл: 1 student    32 Ноя 15 13:08 to.sort
[student @ localhost root]$
```

У sed очень широкие возможности, но довольно непривычный синтаксис, например, замена выполняется командой «s/что_заменять/на_что_заменять/». Чтобы в нем разобраться, нужно обязательно прочесть руководство sed(1) и знать регулярные выражения.

1.8 Упорядочивание

Для того чтобы разобраться в данных, нередко требуется их упорядочить: по алфавиту, по номеру, по количеству употреблений. Основной инструмент для упорядочивания - утилита sort. Рассмотрим ее использование в сочетании с несколькими другими утилитами:

Пример 9. Получение упорядоченного по частотности списка словоупотреблений

```
 [student@ localhost root]$ cat grep.info | tr "[:upper:]" "[:lower:]" | tr
"[:space:][:punct:]" "\n" \
> | sort | uniq -c | sort -nr | head -8
 15233
```

```
  720 the
  342 of
  251 to
  244 a
  213 and
  180 or
  180 is
[student@ localhost root]$
```

В примере 7.9 выполняется подсчет, сколько раз какие слова были употреблены в файле "grep.info" и вывод самых часто употребляемых с указанием количества употреблений в файле. Для этого потребовалось сначала заменить все большие буквы маленькими, чтобы не было разных способов написания одного слова, затем заменить все пробелы и знаки препинания концом строки (*символ* "\n"), чтобы в каждой строке было ровно по одному слову (пользователь всюду взял параметры tr в кавычки, чтобы bash не понял их неправильно). Потом список был отсортирован, все повторяющиеся слова заменены одним словом с указанием количества повторений ("uniq -c"), затем строки снова отсортированы по убыванию чисел в начале строки ("sort -nr") и выведены первые 8 строк ("head -8").

2 МЕТОДИКА ВЫПОЛНЕНИЯ

1. Используя утилиты hexdump и strings, вывести на экран содержимое одного из перечисленных ниже файлов из каталога /bin. Позиция файла для распечатки определяется номером бригады. Имена файлов для выполнения задания 1:

tar, sort, sed, ping, vi, unlink, uname, touch, sleep, sty.

2. Подсчитать общее количество файлов (каталогов) в одном из перечисленных ниже каталогов. Каталог для подсчета количества определяется номером бригады. Имена каталогов для выполнения задания 2:

/bin, /etc, /lib, /proc, /usr, /var, /dev, /sbin, /sys, /root

3. Найти общее количество процессов, выполняющихся в системе в данный момент.
4. Вывести список выполняющихся процессов, в именах которых присутствует слово manager и отсутствует слово grep
5. Создать текстовый файл, содержащий набор строк вида:

123
178
176
755
713
873

С помощью утилиты grep найти строки, в которых есть цифра 7, после которой находится одна из цифр — 1, 3 или 5.

6. Создать текстовый файл, содержащий набор строк вида:

starfish
starless
samscripter
stellar
microsrar
ascender
sacrifice
scalar

С помощью утилиты grep найти строки, начинающиеся на букву s и заканчивающиеся на букву r

7. Создать текстовый файл, содержащий простейшие адреса электронной почты вида username@website.com.

С помощью утилиты grep найти строки, содержащие правильные простейшие адреса. Проверить возможность использования более сложного регулярного выражения для распознавания адресов, содержащих другие допустимые символы.

8. На произвольном примере продемонстрировать работу утилиты tr

Создать текстовый файл, содержащий допустимые и недопустимые IP-адреса, например 127.0.0.1

255.255.255.255
12.34.56
123.256.0.0
1.23.099.255
0.79.378.111

С помощью утилиты grep и руководства man найти строки, содержащие допустимые четырехбайтовые IP адреса.

9. Создать текстовый файл, содержащий корректные и некорректные номера телефонов ведомственной АТС объемом 399 номеров, номера с 000 до 399 – корректные, 0, 400, 900 –некорректные.

С помощью утилиты grep и руководства man найти строки, содержащие допустимые номера телефонов.

3 ОТЧЕТ О РАБОТЕ

Готовится в письменном виде один на бригаду. Содержание отчета:

1. Командные строки, использованные при выполнении заданий 1 - 9 и результаты выполнения заданий (в текстовом виде или в виде снимков экрана).
2. Письменный ответ на контрольный вопрос (номер вопроса определяется выражением номер бригады + 5).

4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Вывод на экран содержимого нетекстового файла с помощью утилит hexdump и strings.
2. Стандартный ввод, вывод, стандартный вывод ошибок.
3. Конвейер и канал.
4. Фильтры.
5. Структурные единицы текста. Подсчет количества единиц текста.
6. Элементарные регулярные выражения.
7. Знакозаменяющие метасимволы.
8. Метасимволы количества повторений в регулярных выражениях.
9. Группировка выражений в регулярных выражениях.
10. Использование зарезервированных символов в регулярных выражениях.
11. Подсчет количества элементов текстового файла.
12. Назначение утилит head, tail, cut.
13. Назначение и использование утилиты grep.
14. Выполнение транслитерации.
15. Назначение потокового редактора sed.