3 Основы программирования на языке командного интерпретатора Shell

3.1 Основные понятия языка Shell

Операционная система UNIX предоставляет большие возможности для создания прикладных программ. Сама оболочка операционной системы, в сущности, представляет собой командный язык, который не только обеспечивает интерфейс пользователя с операционной системой, но и позволяет создавать программы (командные процедуры), дополняющие возможности «системных» команд пользовательскими. Вновь создаваемые командные файлы имеют тот же статус, что и «системные» команды. Таким образом, можно построить операционную среду, отвечающую потребностям отдельного человека или группы пользователей и автоматизировать многие рутинные операции администратора.

Достоинства использования интерпретируемых языков программирования очевидны [7, 8]:

- переносимость: написанный в одной операционной системе командный файл может быть выполнен на другой операционной системе семейства UNIX (главное, чтобы в системе был установлен командный интерпретатор для языка, на котором написана командная процедура)
- простота написания кода: нет необходимости специально обучаться сложному программированию на компилирующих языках, таких как C, C++, Pascal, Fortran (такое программирование наиболее очевидно, поскольку программа пишется в пошаговом режиме, то есть человек принимает решение о своем следующем шаге в зависимости от реакции системы на предыдущий шаг)
- быстрота написания кода благодаря простоте синтаксиса языка и мощным средствам отладки
- большие функциональные возможности: с помощью командного языка можно использовать любые существующие файлы и команды системы.

В существующей среде программирования ОС UNIX сложилась методика эффективной разработки программ, включающая в себя следующие рекомендации [7, 8]:

- желательно, чтобы одна программа выполняла одну функцию (необходимо избегать избыточной функциональности);
- необходимо избегать избыточного, хаотичного и неструктурированного вывода в программе (надо помнить, что вывод любой программы может быть вводом для другой);

- по мере возможности необходимо использовать или переделывать уже существующее средство.

3.1.1 Исполнение командных файлов Shell

Файл, написанный для выполнения командным интерпретатором, – это командный файл, представляющий собой текст из последовательного набора команд. Существует несколько способов его вызова на выполнение [3]:

1. явно вызвать оболочку `shell` командой `sh` и передать в качестве параметра имя командного файла:
   ```
   $ sh имя_файла [параметры]
   ```
   или
   ```
   $ sh < имя_файла [параметры]
   ```
   или
   ```
   $ . имя_файла [параметры]
   ```
   (в отличие от первых двух – файл будет исполняться в текущем экземпляре `shell`)
2. сделать командный файл исполняемым, для чего достаточно поменять атрибуты (чтение, запись, выполнение) этого файла командой `chmod`:
   ```
   $ chmod  711  имя_файла
   ```
   а потом вызвать файл на исполнение, просто указав его имя
   ```
   $ имя_файла [параметры]
   ```

При вызове командного файла можно указать параметры, которые в дальнейшем будут доступны внутри командного файла в виде переменных.

При исполнении команд происходит поиск последних по каталогам в следующем порядке (по умолчанию): сначала – текущий каталог; затем – системный `/bin`; в конце – системный `/usr/bin`. Таким образом, если имя пользовательского командного файла дублирует имя команды в системных каталогах, то последняя станет недоступной (если только не набирать её полного имени).

3.1.2 Простейшие средства Shell

3.1.2.1 Комментарии

Как во всяком языке программирования в тексте на языке командного интерпретатора `shell` могут быть использованы комментарии. Для этого используется символ "`#`". Все, что находится в строке (в командном файле) правее этого символа, воспринимается интерпретатором как комментарий. Например,

```
# Это комментарий.
## И это комментарий.
### И это тоже.
```

3.1.2.2 Структура команд

Команды в `shell` имеют аналогичный системным командам формат, то есть после имени команды можно указать параметры и аргументы через пробел:

```
Имя_команды  флаги  аргумент(ы)
```

Например,

```
kill -9  866
```

3.1.2.3 Метасимволы, генерация имён файлов

Метасимволы – символы, имеющие специальное значение для интерпретатора:

`? * ; & ( ) | ^ < > пробел табуляция возврат_каретки`

Однако каждый из этих символов может представлять самого себя, если перед ним стоит \(бэк-слэш). Все символы, заключённые между кавычками ' и ', представляют самих себя. Между двойными кавычками (") выполняются подстановки команд и параметров, а символы \, `, " и $ могут экранироваться предшествующим символом \.

Бэк-слэш (\) экранирует следующий за ним символ, что позволяет использовать специальные символы просто как символы, представляющие сами себя (он может экранировать и сам себя – \\), так же в командном файле бэк-слэш позволяет объединять строки в одну (экранировать конец строки).

Двойные кавычки (") экранируют пробелы, представляя несколько слов, разделённых пробелами, как одно слово. Однако кавычки позволяют подставлять команды, параметры и переменные в строку текста.

После всех подстановок в каждом слове команды ищутся символы *, ?, и [. Если находится хотя бы один из них, то это слово рассматривается как шаблон имён файлов и заменяется именами файлов, удовлетворяющих данному шаблону (в алфавитном порядке). Если ни одно имя файла не удовлетворяет шаблону, то он остаётся неизменным. Значения указанных символов подробно рассмотрено в разделе 2.2.2.8.

3.1.2.4 Вывод на экран

Для вывода текстовой информации на стандартный вывод (по умолчанию – экран терминала) используется команда `echo`, имеющая следующий синтаксис:

```
echo [-n] [строка текста...]
```

В качестве параметра `строка текста` указывается текст, которой надо вывести на экран. При указании параметра `-n` после вывода на экран текста не производится переход на следующую строку. Например, действие команды `echo` без параметра `-n`:

```
$ echo Строка текста
Строка текста
$
```

и действие команды `echo` с параметром `-n`:

```
$ echo -n Строка текста
Строка текста$
```

3.2 Переменные Shell

Командные процедуры могут манипулировать с четырьмя типами переменных [3, 7-11]:

- позиционные параметры;
- специальные параметры;
- именованные (пользовательские) переменные;
- именованные (специальные) переменные.

3.2.1 Позиционные параметры

Позиционные параметры являются переменными в командной процедуре. Их значение устанавливается из аргументов, указанных в командной строке при запуске программы на исполнение. Позиционные параметры нумеруются от `0` до `9`, и на них ссылка производится с помощью символа `$`:

`$0` соответствует имени данного командного файла;
`$1` первый по порядку параметр;
`$2` второй по порядку параметр;
`...` …
`$9` девятый по порядку параметр.

В программе можно обратиться непосредственно только к девяти позиционным параметрам одновременно.

Например, программа вызывается с помощью командной строки, подобной следующей:

```
shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9
```

то позиционному параметру `$1` в программе присваивается значение `pp1`, `$2` – значение `pp2` и т.д. во время вызова программы. Чтобы практически рассмотреть это замещение позиционных параметров, создайте файл `pp`:

```
echo Первый позиционный параметр: $1
echo Второй позиционный параметр: $2
echo Третий позиционный параметр: $3
echo Название файла: $0
```

Если выполнить эту программу с аргументами `один`, `два`, `три`, то будет следующий результат:

```
$ sh pp один два три
Первый позиционный параметр: один
Второй позиционный параметр: два
```

```
Третий позиционный параметр: три
Название файла: pp
$
```

Для примера создадим командный файл `AddToPhones`, который записывает переданные в командной строке параметры (ФИО, пол человека, его телефон) в специальный файл, например `telephones`. Данный командный файл будет содержать только одну строку:

```
echo "$1        $2      $3" >> $HOME/telephones
```

При запуске вводим после имени программы необходимые сведения

```
$ sh AddToPhones "Иванов И.И." муж 12-34-56
$ cat telephones
Иванов И.И.     муж         12-34-56
$
```

В данном случае первый параметр – это фамилия и инициалы человека. Чтобы инициалы из-за разделяющего их с фамилией пробела не были интерпретированы как второй параметр, необходимо заключить их в двойные кавычки. В дальнейшем на других примерах продолжим добавлять функциональность к этой программе.

3.2.2.1 Сдвиг параметров

Командный интерпретатор `Shell` позволяет указывать в командной строке до 128 аргументов, но ссылаться можно не более чем на 9 позиционных параметров. Однако существует команда `shift`, с помощью которой можно сдвинуть имена на остальные аргументы, если их больше 9 (причём окно доступа к переменным остается шириной 9). Синтаксис команды:

```
shift [сдвиг]
```

если параметр сдвиг не указан, то величина сдвига принимается равной одному. Параметр `сдвиг` может быть задан только положительным числом.

Для большего понимания действия данной команды создайте следующий файл с именем `ManyParametrs`:

```
echo "Исходное состояние: $1 $5 $9 "
shift
echo "1 сдвиг: первый=$1 пятый=$5 девятый=$9"
shift 2
echo "1 + 2 = 3 сдвига: первый=$1 пятый=$5 девятый=$9"
```

вызовите его с 13-ю параметрами:

```
ManyParametrs 10 20 30 40 50 60 70 80 90 100 110 120 130
```

Проанализируйте самостоятельно работу данной программы и команды `shift` в частности.

3.2.2.2 Переопределение параметров внутри программы

Своеобразный подход к параметрам даёт команда `set`. Используя данную команду, можно установить новые параметры в уже выполняемой программе. Можно использовать следующие виды синтаксиса:

```
set значения_параметров_через_пробел
```

или

```
set `команда`
```

где после выполнения `команда`, её результат будет рассматриваться выполняющейся программой как параметры.

Например, следующая программа:

```
echo "Первый = $1; Второй = $2"
set два один
echo "Первый = $1; Второй = $2"
```

будет переназначать параметры следующим образом:

```
$ sh SetParametrs один два
Первый = один; Второй = два
Первый = два; Второй = один
$
```

Другой способ использования команды `set` для назначения параметров может быть проиллюстрирован следующим командным файлом:

```
set `date`
cal $6
```

Данный командный файл сначала вызывает команду `date`, возвращающая текущую дату, состоящую из шести слов, последним из которых является текущий год, который, в свою очередь, передаётся команде `cal`, выводящей на экран календарь на указанный год.

3.2.2 Специальные параметры

В командном интерпретаторе доступны специальные параметры, автоматически устанавливаемые самим командным интерпретатором. В таблице 11 перечислены все доступные специальные параметры и дано пояснение по каждому из них.

Таблица 11 – Специальные параметры командного интерпретатора

Параметр	Описание
#	Количество позиционных параметров передаваемых в `shell` (содержит десятичное значение)
-	Флаги, указанные при запуске командного интерпретатора или установленные командой `set`
?	Десятичное значение, возвращённое предыдущей синхронно выполненной командой (код завершения программы – код "`0`" соответствует нормальному завершению про-

Параметр	Описание
	цесса)
$	Номер текущего процесса
!	Номер последнего асинхронного (фонового) процесса
@	Перечень параметров, как совокупность слов (эквивалентно $1 $2 $3 ...)
*	Перечень параметров, как одна строка (эквивалентно "$1 $2 $3 ...")

Чтобы получить значения этих переменных, перед ними нужно поставить знак $.

Разберём действие каждого параметра на простом примере. Создайте следующую программу с именем SpecParametrs:

```
echo Имя программы - $0
echo Код завершения предыдущего процесса - $?
echo Идентификатор процесса - $$
echo Идентификатор последнего фонового процесса - $!
echo Количество параметров - $#
echo Значения параметров, как строки, - $*
echo Значения параметров, как слов, - $@
set -au
echo Режимы работы командного интерпретатора - $-
```

Результат работы программы будет следующий:

```
$ sh SpecParametrs один два три
Имя программы - SpecParametrs
Код завершения предыдущего процесса - 0
Идентификатор процесса - 20255
Идентификатор последнего фонового процесса -
Количество параметров - 3
Значения параметров, как строки, - один два три
Значения параметров, как слов, - один два три
Режимы работы командного интерпретатора – au
$
```

3.2.3 Именованные (пользовательские) переменные

Все переменные в командном языке – текстовые. В Shell используется всего два типа данных: строка символов и текстовый файл. Их имена должны начинаться с буквы или знака подчеркивания и состоять из латинских букв, цифр и знака подчеркивания. Чтобы воспользоваться значением переменной, надо перед ней поставить символ $. Использование значения переменной называется подстановкой.

Определение переменной содержит её имя (без символа $) и значение:

```
переменная=значение
```

Таким образом, для присваивания значений переменным используется оператор присваивания "=". При этом необходимо записывать как переменную, так и её значение без пробелов относительно символа присвоения. Например, следующее объявление

```
var_1=13
```

присвоит переменной значение "13" (однако это не число, а строка из двух цифр).

Если необходимо присвоить значение, включающее в строке пробелы или другие разделяющие знаки, то надо воспользоваться двойными кавычками:

```
var_2="Операционная система FreeBSD"
```

Следующее объявление присвоит переменной пустую строку:

```
var_3=
```

Как отмечалось выше, при обращении к пользовательской переменной необходимо перед именем ставить символ `$`. Например, команды

```
var_2="Операционная система FreeBSD"
echo $var_2
echo var_2
```

выдадут на экран

```
Операционная система FreeBSD
var_2
```

Также необходимо помнить, что не может быть одновременно функции и переменной с одинаковыми именами.

3.2.3.1 Конструкции подстановки переменных

Для того чтобы имя переменной не сливалось со строкой, следующей за именем переменной, используются фигурные скобки. В таблице 12 представлены конструкции, используемые для подстановки значений переменных.

Таблица 12 – Конструкции подстановки

Конструкция	Назначение
`${переменная}`	Если значение `переменная` определено, то оно подставляется. Скобки применяются лишь в том случае, если за переменной следует символ, который без скобок «приклеится» к имени.
`${переменная:-слово}`	Если `переменная` определена и не является пустой строкой, то подставляется её значение; иначе подставляется `слово`.
`${переменная:=слово}`	Если `переменная` не определена или является пустой строкой, ей присваивается

Конструкция	Назначение
	значение `слово`; после этого подставляется её значение.
`${переменная:?слово}`	Если `переменная` определена и не является пустой строкой, то подставляется её значение; иначе на стандартный вывод выводится `слово` и выполнение командного интерпретатора завершается. Если `слово` опущено, то выдается сообщение "`parameter null or not set`".
`${переменная:+слово}`	Если `переменная` определена и не является пустой строкой, то подставляется `слово`; иначе подставляется пустая строка.

Например, можно проверять в программе существует ли переменная и определена ли она с помощью такой конструкции:

```
${var_1:?"Такого параметра нет"}
```

в этом случае на экран выведется соответствующее сообщение и выполнение программы прекратиться, если переменная `var_1` не существует или она не была определена до указанной конструкции.

3.2.3.2 Определение переменных с помощью команд

Возможны и иные способы присваивания значений пользовательским переменным. Так в качестве значения можно присвоить переменной результат выполнения какой-либо команды. Для этого необходимо использовать следующий синтаксис:

```
имя_переменной=`имя_команды`
```

то есть обратные кавычки указывают на то, что сначала должна быть выполнена заключенная в них команда, а результат её выполнения, вместо выдачи на стандартный вывод, присваивается в качестве значения переменной.

Например, запись

```
CurrentPath=`pwd`
```

приводит к тому, что сначала выполняется команда `pwd`, а результат присваивается переменной `CurrentPath`.

3.2.3.3 Ввод значений переменных с клавиатуры

Можно присвоить значение переменной и с помощью команды `read`, которая обеспечивает приём значения переменной с дисплея (клавиатуры) в диалоговом режиме [3, 7-11]. Для этого необходимо использовать следующий синтаксис:

```
read переменная1 переменная2 … переменнаяN
```

Обычно команде `read` в командном файле предшествует команда `echo`, которая позволяет предварительно выдать какое-то поясняющее сообщение на экран. Например:

```
echo -n "Введите трехзначное число: "
read x
```

Введенное с клавиатуры число будет присвоено переменной `x`.

Одна команда `read` может прочитать (присвоить) значения сразу для нескольких переменных. Если переменных в `read` больше, чем их введено (через пробелы), оставшимся присваивается пустая строка. Если передаваемых значений больше, чем переменных в команде `read`, то лишние – игнорируются. На самом деле интерпретатор для продолжения работы ждёт лишь нажатия клавиши `<CR>` (перевод каретки – `Enter`). Введённое число воспринимается им не как число, а как последовательность символов. Интерпретатор не проверяет, что введено, поэтому в качестве значения переменной может оказаться любая введенная «абракадабра» или просто нажатие `<CR>`, как значение пустой строки.

В качестве примера использования команды `read` переработаем ранее рассмотренный пример добавления записи в телефонный справочник. В рассмотренном ранее примере сведения для добавления новой записи берутся из параметров командной строки, что достаточно неудобно для конечного пользователя, поэтому напишем программу, которая будет в интерактивном режиме спрашивать пользователя о добавляемой записи:

```
echo Добавление сведений в телефонный справочник
echo -n Введите имя:
read FIO
echo -n "Введите пол (муж/жен): "
read pol
echo -n Введите телефон:
read phone
echo "$FIO      $pol    $phone" >> $HOME/telephones
```

Использование пользовательских переменных для промежуточного хранения сведений позволит в дальнейшем производить некоторую дополнительную обработку, если потребуется. Вывод программы следующий:

```
$ sh AddToPhones
Добавление сведений в телефонный справочник
Введите имя: Сидоров С.С.
Введите пол: муж
Введите телефон: 98-76-65
$ cat telephones
Иванов И.И.     муж     12-34-56
Сидоров С.С.    муж     98-76-65
$
```

3.2.4 Именованные (специальные) переменные

Ранее в разделах 3.2.2 и 3.2.3 были приведены примеры записи в телефонный справочник, в которых использовалась переменная $HOME. Данная переменная является специальной именованной переменной операционной среды. Эти переменные обычно устанавливаются в файле .profile и rc в регистрационном оглавлении пользователя. Вызвав команду set без параметров, можно получить информацию о стандартных переменных, созданных при входе в систему (и передаваемых далее всем новым процессам "по наследству"), а также переменных, созданных и экспортируемых процессами, созданными самим пользователем. В таблице 13 представлены основные специальные переменные, которыми оперирует операционная среда.

Такие переменные имеют для оболочки Shell особое значение, их не следует использовать для других целей.

3.2.4 Манипуляции с переменными

3.2.4.1 Экспорт переменных

Основополагающим понятием в ОС UNIX, как и в любой другой ОС, является понятие процесса. Процесс возникает тогда, когда запускается на выполнение какая-либо команда. У каждого процесса есть своя среда – множество доступных ему переменных. Переменные локальны в рамках процесса, в котором они объявлены, то есть где им присвоены значения. Для того чтобы они были доступны и другим порождаемым процессам, надо передать их явным образом. Для этого используется встроенная команда export, имеющая следующий синтаксис:

```
export [имя_переменной1 имя_переменной2 …]
```

если параметры не заданы, выдаётся список имён экспортируемых переменных. [3, 7-11]

Механизм работы данной команды рассмотрим на примере.

Пусть командный файл pr1 объявляет и определяет две переменных – var1 и var2, экспортирует переменную var2, а затем последовательно вызывает два других командных файла pr1_1 и pr1_2, которые переопределяют эти переменные:

1. pr1:

```
echo Расчет pr1
var1=1 var2=2
echo var1=$var1 var2=$var2
export var2
sh pr1_1
sh pr1_2
echo Снова расчет pr1: var1=$var1 var2=$var2
```

Таблица 13 – Описание специальных переменных

Переменная	Описание	Пример
$MAIL	При работе с терминалом оболочка перед выдачей очередного приглашения обследует состояние файла, определяемого этой переменной. Если с момента предыдущего обследования в этот файл были внесены изменения, оболочка выдает сообщение you have mail (для Вас почта) перед выдачей приглашения для ввода очередной команды.	MAIL=/var/mail/student
$HOME	Это имя домашнего каталога, в котором пользователь оказывается после входа в систему.	HOME=/home/student
$CDPATH	Список каталогов, просматриваемых командой cd. Имена каталогов разделяются двоеточиями.	CDPATH=:..:$HOME/desk
$PATH	Задаёт последовательность файлов, которые просматривает оболочка в поисках команды. Имена файлов разделяются двоеточиями. Последовательность просмотра соответствует очередности следования имён в переменной. Однако первоначально поиск происходит среди так называемых встроенных команд. В число встроенных команд входят наиболее часто используемые команды, например echo, cd. После набора командной строки и нажатия <CR> оболочка (после выполнения необходимых подстановок) распознаёт имя, соответствующее команде и осуществляет её поиск в директориях, перечисленных в переменной. Если команда размещена вне этих директорий, она не будет	PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:/usr/X11R6/bin:/home/student/bin

Переменная	Описание	Пример
	найдена. Если присутствует несколько команд с одинаковым именем, то вызвана будет та, которая расположена в директории, просматриваемой первой.	
$IFS	Внутренний Разделитель Полей – перечисляет символы, которые служат для разделения слов. Таковыми являются "пробел", "табуляция" и "перевод строки".	IFS=' '
$LOGNAME	Имя входа ("имя" пользователя).	LOGNAME=student
$PWD	Имя текущего каталога	PWD=/usr/home/student/temp
$PS1	Вид промтера – приглашения – которое печатается в командной строке.	PS1='$ '
$PS2	Этот промтер используется как приглашение к продолжению ввода незаконченной команды в очередной строке.	PS2='> '
$SHELL	Указывает оболочку, которую использует пользователь.	SHELL=/bin/sh
$TERM	Указание типа терминала.	TERM=ansi
$TERMCAP	Строка задания параметров терминала.	TERM- CAP=console|con80x25|dumb|linux :li#25:co#80::
$UID	Идентификатор пользователя.	UID=501
$TZ	Определяет зону времени	TZ= EST5EDT

2. pr1_1:

```
    echo "  Расчет pr1_1"
    echo "  var1=$var1 var2=$var2"
    var1=a var2=b
    echo "  var1=$var1 var2=$var2"
    export var1
```

3. pr1_2:

```
    echo "  Расчет pr1_2"
    echo "  var1=$var1 var2=$var2"
    var1=A var2=B
    echo "  var1=$var1 var2=$var2"
    export var2
```

Запустив на исполнение первый файл, получим следующий вывод:

```
    $ sh pr1
    Расчет pr1
    var1=1 var2=2
            Расчет pr1_1
            var1= var2=2
            var1=a var2=b
            Расчет pr1_2
            var1= var2=2
            var1=A var2=B
    Снова расчет pr1: var1=1 var2=2
    $
```

Из примера видно, что значения переменных экспортируются только в вызываемые расчёты. Таким образом, если необходимо чтобы все вновь запускаемые программы «видели» какую-либо переменную, достаточно в командной строке объявить и инициализировать эту переменную, а потом экспортировать её. Например, создадим переменную `MyGlobalVariable`:

```
    $ MyGlobalVariable=true
    $ export MyGlobalVariable
```

Экспортировать переменные можно и командой `set` с параметром `-a`. После вызова такой команды, все вновь объявляемые переменные будут сразу же экспортироваться.

3.2.4.2 Ограничение доступа к переменным

Переменные, значение которых желательно сохранить неизменными, можно объявить доступными только для чтения с помощью ко-

манды `readonly`. Эта команда имеет аналогичный вид команде `export`, а именно:

```
readonly [имя_переменной1 имя_переменной2 …]
```

если параметры не заданы, выдаётся список всех переменных, доступных только для чтения.

Последующие попытки присвоить значение таким переменным отвергаются.

Вопросы для самоконтроля

1. Выведите на экран содержимое файла `/.profile`. Прокомментируйте его содержимое.
2. Что означает параметр `-n` команды `echo`?
3. Каким образом можно переопределить позиционные параметры?
4. Каким образом можно узнать сколько позиционных параметров было передано в командной строке при запуске командного файла?
5. С помощью какой команды производится ввод значений переменных с клавиатуры?
6. Что будет выведено на экран при выполнении следующего командного файла? Почему?

```
Var_1="Yes";
echo ${Var_1:-"No"}
```

7. С помощью какой команды можно вывести на экран все специальные именованные переменные среды?
8. Для чего производится экспорт переменных?
9. Напишите командный файл, который выводит на экран календарь указанного с клавиатуры года.

3.3 Программные структуры

При рассмотрении программных структур, используемых в командном интерпретаторе, необходимо определиться с некоторыми терминами:

Простая команда – это последовательность слов, разделенная пробелами. Первое слово является именем команды, которая будет выполняться, а остальные будут переданы ей как аргументы. Имя команды передаётся ей как аргумент номер `0` (т.е. имя команды является значением параметра `$0`). Значение, возвращаемое простой командой – это её статус завершения, если она завершилась нормально, или (восьмеричное) `200+статус`, если она завершилась аварийно.

Список – это последовательность одного или нескольких конвейеров, разделенных символами `;`, `&`, `&&` или `||` и быть может заканчивающаяся символом `;` или `&`.

Команда – это либо простая команда, либо одна из управляющих конструкций. Кодом завершения команды является код завершения её последней простой команды.

3.3.1 Арифметические операции

Разнообразные возможности для выполнения арифметических операций над числами или значениями переменных имеет команда `expr` [3, 7-11]. Результат вычислений выводится на стандартный вывод. Операнды выражения должны быть разделены пробелами. Метасимволы, используемые в выражениях, должны быть экранированы. Строки, содержащие пробелы или другие специальные символы, должны быть заключены в кавычки. Целые рассматриваются как 32-битные числа.

Список операторов в порядке возрастания приоритета (операции с равным приоритетом заключены в фигурные скобки, перед символами, которые должны быть экранированы, стоит `\)` представлен в таблице 14.

Таблица 14 – Список операторов команды `expr`

Операторы	Описание
`<выр1> | <выр2>`	если `<выр1>` не пустое и не нулевое, то возвращает его, иначе возвращает `<выр2>`
`<выр1> \& <выр2>`	если оба `<выр1>` и `<выр2>` не пустые и не нулевые, то возвращает `<выр1>`, иначе возвращает `0`
`<выр1> { =, \>, \>=, \<, \<=, != } <выр2>`	возвращает результат целочисленного сравнения если оба `<выр1>` и `<выр1>` целые; иначе возвращает результат лексического сравнения
`<выр1> { +, - } <выр2>`	сложение и вычитание целочисленных аргументов
`<выр1> { \*, /, % } <выр2>`	умножение, деление и получение остатка от деления целочисленных аргументов. Операция умножения ("`*`") обязательно должна быть экранирована
`<выр1> : <выр2>`	оператор сопоставления `:` сопоставляет первый аргумент со вторым, который должен быть регулярным выражением. Обычно оператор сравнения возвращает число символов, удовлетворяю-

Операторы	Описание
	щих образцу (0 при неудачном сравнении). Для выделения части первого аргумента могут применяться символы \(и \).

Регулярное выражение строится следующим образом:

. - обозначает любой символ;

* - обозначает предыдущий символ, повторенный несколько раз;

[] - обозначают любой один из указанных между ними символов; группа символов может обозначаться с помощью знака "-"; если после [стоит ^, то это эквивалентно любому символу, кроме указанных в скобках и <CR>; для указания] в качестве образца, надо поставить ее сразу за [(т.е. []...]); . и * внутри квадратных скобок обозначают самих себя.

Все остальные символы (и ^, если стоит не в квадратных скобках) обозначают самих себя. Для указания символов ., *, [и] надо экранировать их.

Примеры использования:

1. a=`expr $a + 1`

производится увеличение на 1 переменной a

2. b=`expr $a : '.*/\(.*\)' \| $a`

выделяет из имени файла короткое имя (т.е. из /usr/util/ena выделяется ena). Надо помнить, что одиночный символ / будет воспринят как знак операции деления.

3. count=`expr $VAR : '.*'`

получение количества символов переменной VAR.

4. a=`expr 'полукеды' : 'пол'`

получение количества одинаковых символов с начала слова.

Обратите внимания, что команда expr при вызове указывается в обратных кавычках.

В качестве побочного эффекта expr возвращает следующие коды завершения:

- 0 – если выражение не нуль и не пустая строка;
- 1 – если выражение нуль или пустая строка;
- 2 – для некорректных выражений.

Команда expr также выдаёт следующие сообщения об ошибках:

- syntax error – для ошибок в операторах или операндах;
- non-numeric argument – для попыток применения арифметических операций к нечисловым строкам.

Вопросы для самоконтроля

1. Напишите программу решения линейного уравнения вида $a \cdot x + b = 0$, где *a*, *b* – целочисленные коэффициенты, задаваемые пользователем с клавиатуры. Ответ вывести в виде обыкновенной дроби.

3.3.2 Пустой оператор

Пустой оператор – эквивалент операции "NOP" (*no op*, нет операции) – имеет формат

```
:
```

Он ничего не делает и возвращает значение 0. Например, в конструкции

```
:
echo $?
```

На экране отобразится ноль.

Данный оператор может использоваться в следующих случаях [10]:

- Для организации бесконечного цикла (рассмотрению использования циклических операторов посвящён раздел 3.3.5 настоящего пособия).
- Как символ-заполнитель в условном операторе (рассмотрению использования условных операторов посвящён раздел 3.3.4 настоящего пособия).
- Как символ-заполнитель в операциях, которые предполагают наличие двух операндов. Например:

```
: ${username=`who am i`}
```

  без символа : выдаётся сообщение об ошибке, если username не является командой.

- Как символ-заполнитель для оператора вложенного документа. Например, можно создать блочный комментарий:

```
: << COMMENTBLOCK
echo "Эта строка не будет выведена."
Эта строка комментария не начинается с символа "#".
&*@!!++= - просто символы
COMMENTBLOCK
...
```

  Между словами COMMENTBLOCK находится многострочный блок комментариев. В данном случае весь блок комментариев рассматривается как встроенный документ и его вывод перенаправляется в пустой оператор :.

- В операциях с подстановкой параметров. Например:

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
  echo
  echo "Имя машины: $HOSTNAME."
```

```
echo "Ваше имя: $USER."
echo "Ваш домашний каталог: $HOME."
echo "Ваш почтовый ящик: $MAIL."
```

Если хотя бы одна переменная не определена, то произойдёт вывод сообщения об ошибке.

- В операциях замены подстроки с подстановкой значений переменных (см. раздел 3.2.3.1).
- В комбинации с оператором перенаправления вывода (>), усекает длину файла до нуля[1]. Если указан несуществующий файл, то он создаётся.

```
: > File.new
```

То же самое можно осуществить и с помощью команды

```
cat /dev/null > File.new
```

Однако при использовании пустого оператора не производится создание нового процесса.

- В комбинации с оператором перенаправления с добавлением в конец файла (>>) обновляет время последнего доступа к файлу, не меняя содержимое указанного файла[1]. Например:

```
: >> File.old
```

Если задано имя несуществующего файла, то он создается.

- Может использоваться как разделитель полей в файле пользовательских паролей `/etc/passwd` и именованных специальных переменных `$PATH` и `$CDPATH` (см. раздел 3.2.4).

3.3.3 Оператор сравнения test ("[ ]")

Команда `test` проверяет выполнение некоторого условия и обычно используется в командных процедурах [3, 7-11]. С использованием этой команды формируются операторы выбора и цикла командного языка. Существует два возможных формата команды:

```
test условие
```

или

```
[ условие ]
```

между скобками и содержащимся в них условием обязательно должны быть пробелы. Также пробелы должны быть и между значениями и символом сравнения или операции (как, кстати, и в команде `expr`).

В результате выполнения команды будет возвращён код завершения: если он равен `0` – то истинность условия подтверждено, если равен `1` – то истинность условия не подтверждена. В случае возникновения ошибки при сравнении код завершения равен `2`.

Имеется четыре типа проверок [3, 7-11]:

- оценка числовых значений;

[1] Такое использование пустого оператора применимо только к обычным файлам и неприменимо к конвейерам, символическим ссылкам и другим специальным файлам.

- оценка типа файла;
- оценка строк;
- сложные операции.

Для каждого типа применяются свои примитивы.

3.3.3.1 Оценка числовых значений

Для сравнения числовых значений используется следующий синтаксис:

```
X op Y
```

где `X`, `Y` - числа или числовые переменные;

`op` – соответствующая операция сравнения, которая может принимать одно из значений, представленных в таблице 15.

Таблица 15 – Операции сравнения числовых значений

Операция	Описание
`X -eq Y`	сравнение на равенство
`X -ne Y`	сравнение на неравенство
`X -gt Y`	`X` больше `Y`
`X -ge Y`	`X` больше или равно `Y`
`X -lt Y`	`X` меньше `Y`
`X -le Y`	`X` меньше или равно `Y`

При сравнении числовых значений с помощью указанных операторов команда `test` воспринимает строки символов как целые числа. Таким образом, если надо обнулить переменную, скажем, `X`, то необходимо присвоить `X=0`. Отметим, что во всех остальных случаях (кроме операций сравнения числовых значений) "нулевому" значению соответствует пустая строка.

Рассмотрим в качестве примера работы команды сравнения числовых значений следующий командный файл:

```
x=abc ; [ abc -eq $x ] ; echo $?
x=10 ; [ 10 -eq $x ] ; echo $?
x=10.1 ; [ 10.1 -eq $x ] ; echo $?
x=10 ; [ 12 -lt $x ] ; echo $?
```

при выполнении которого на стандартном выводе будут получены следующие результаты:

```
[: abc: bad number
2
0
[: 10.1: bad number
2
1
```

Поясним результаты работы команды сравнения. В первой строке выводится сообщение об ошибке (в примере вместо числового зна-

чения передаются текстовые символы), а во второй – код завершения команды сравнения (так как команда равнения завершилась аварийно с ошибками, то код завершения равен 2). В третьей строке вывод результата сравнения – истина. Так как команда сравнения ожидает ввод целых чисел, а 10.1 является дробным числом, то в четвёртой и пятой строке вывода сообщается о соответствующей ошибке. В последней строке выводится результат сравнения – ложь (так как 12 не меньше 10).

3.3.3.2 Оценка типа файла

Для оценки типа файла используют следующий синтаксис:

```
op filename
```

где `op` – операция сравнения, которая может принимать одно из значений, представленных в таблице 16.

Таблица 16 – Основные операции оценки типа файлов

Операция	Описание
`-f filename`	файл `filename` существует и является обычным файлом
`-d filename`	файл `filename` существует и является каталогом
`-c filename`	файл `filename` существует и является специальным символьно-ориентированным файлом
`-b filename`	файл `filename` существует и является специальным блок-ориентированным файлом
`-r filename`	файл `filename` существует и имеется разрешение на чтение
`-w filename`	файл `filename` существует и имеется разрешение на запись в файл
`-x filename`	файл `filename` существует и является выполняемым
`-s filename`	файл `filename` существует и не пустой (имеет ненулевую длину)
`-p filename`	файл `filename` существует и является именованным каналом (pipe)
`-t дескриптор_файла`	открытый файл с указанным дескриптором (по умолчанию 1) существует и ассоциирован с терминалом

Для рассмотрения примера использования операторов оценки типа файла создайте командный файл с именем `File1`, в котором будет только одна команда `date`. Также создайте командный файл

TestFiles, в котором будет выполняться проверка типа файла, переданного в качестве параметра:

```
[ -s $1 ]; echo "Файл существует: $? "
[ -f $1 ]; echo "Файл: $? "
[ -d $1 ]; echo "Каталог: $? "
[ -r $1 ]; echo "Чтение из файла: $? "
[ -w $1 ]; echo "Запись в файл: $? "
[ -x $1 ]; echo "Исполняемый файл: $? "
```

Запустив файл TestFiles с параметром File1, получим следующие результаты:

```
$ sh TestFiles File1
Файл существует: 0
Файл: 0
Каталог: 1
Чтение из файла: 0
Запись в файл: 0
Исполняемый файл: 1
```

С помощью команды chmod измените атрибут файла на исполняемый:

```
chmod  711  File1
```

и снова запустите файл TestFiles. Вывод будет следующий:

```
$ sh TestFiles File1
Файл существует: 0
Файл: 0
Каталог: 1
Чтение из файла: 0
Запись в файл: 0
Исполняемый файл: 0
```

Самостоятельно проверьте вывод результатов работы данного командного файла, если передать в качестве параметра имя несуществующего файла.

3.3.3.3 Оценка строк

Для оценки и сравнения строк синтаксис в зависимости от используемого оператора сравнения следующий:

- унарный оператор сравнения:

```
op1 S
```

- бинарный оператор сравнения:

```
S op R
```

где S, R – строки или строковые переменные;

op, op1 – операции сравнения, которые могут принимать одно из значений, представленных в таблице 17.

Таблица 17 – Операции оценки и сравнения строк

Операция	Описание
`str1 = str2`	строки `str1` и `str2` совпадают
`str1 != str2`	строки `str1` и `str2` не совпадают
`-n str1`	строка `str1` существует (непустая)
`-z str1`	строка `str1` не существует (пустая)

В качестве примера представим следующие команды сравнения строк:

```
x=abc ; [ -n $x ] ; echo $?
```

Результат: `0`

```
x=abc; [ abc = $x ]; echo $?
```

Результат: `0`

```
x="" ; [ -n $x ] ; echo $?
```

Результат: `1`

```
[ abc ] ; echo $?
```

Результат: `0`

```
[ ] ; echo $?
```

Результат: `1`

В двух последних командах такой результат получается в связи с тем, что в первом случае в скобках стоит не пустая строка, а во втором – пустая строка.

Кроме того, существуют два стандартных значения условия, которые могут использоваться вместо условия (для этого не нужны скобки).

```
true ; echo $?
```

Результат: `0`

```
false ; echo $?
```

Результат: `1`

3.3.3.4 Сложные операции

Несколько проверок разных типов могут быть объединены логическими операциями в сложные операции:

`!` – инвертирует значение кода завершения (операция отрицания);

`-o` – дизъюнкция (логическое "ИЛИ");

`-a` – конъюнкция (логическое "И").

Следующие команды иллюстрируют использование сложных операций:

- `[ ! abc ]`
- `[ $x -qe 0 -a $x -le 10 ]`
- `[ -w File2 -a -r File1 ]`
- `[ $x -eq 0 -o $y -eq 0 ]`

Вопросы для самоконтроля

1. В следующих командах используются сложные операции сравнения. Попытайтесь определить коды завершения у каждой команды.

- `x=abc; [ $x -a -f File1 ] ;`
- `x=""; [ "$x" -a -f File1 ] ;`
- `x=""; [ "$x" -a -f File1 -o true ] ;`
- `x=abc; [ "$x" -a -f File1 -o ! true ] ;`

3.3.4 Условное управление

В командном интерпретаторе используются два оператора условного управления [3, 7-11]:

- оператор условного перехода `if`
- оператор выбора `case`

3.3.4.1 Условный оператор if

В общем случае оператор `if` имеет синтаксис:

```
if  условие1
  then  список1
    [elif  условие2
     then  список2]
  [else  список3]
fi
```

Ключевые слова `if`, `then`, `else`, `elif` и `fi` пишутся с начала строки. Допускается вложение произвольного числа операторов `if` (как и других операторов). Разумеется, `список` в каждом случае должен быть осмысленный и допустимый в данном контексте. Если выполнено `условие` (как правило, это команда с кодом завершения `0`), то выполняется `список`, иначе он пропускается.

Например:

```
echo -n " А какую оценку получил на экзамене?: "
read z
if [ $z = 5 ]
    then echo  Молодец !
    elif [ $z = 4 ]
        then echo  Все равно молодец !
        elif [ $z = 3 ]
            then echo  Все равно !
            elif [ $z = 2 ]
                then echo  Все !
                else echo   !
fi
```

Как правило, в качестве команды, входящей в `условие` используют команду `test`. Однако, возможно использование любой другой

команды, главное требование – код завершения должен быть 0, чтобы выполнить список команд, написанных после слова then; либо любой другой код завершения, чтобы выполнить команды после слова else. Например, следующий командный файл:

```
echo "Введите слово и имя файла:"
read word file
if grep $word $file > /dev/null
    then echo "$word есть в $file"
    else echo "$word нет в $file"
fi
```

использует команду grep для поиска слова в файле. Результаты работы grep выводит на экран.

Также можно использовать и сложные условия. Например, в рассматриваемом ранее примере добавления записи в телефонный справочник можно реализовать дополнительную проверку вводимых сведений о человеке:

```
echo Добавление сведений в телефонный справочник
echo -n Введите имя:
read FIO
echo -n "Введите пол (муж/жен): "
read pol
if [ $pol != муж –а $pol != жен ]
    then echo "Неправильно указан пол"
        exit
fi
echo -n Введите телефон:
read phone
echo "$FIO      $pol    $phone" >> $HOME/telephones
```

Если пользователь неправильно введёт обозначение пола человека, то на экран будет выведено соответствующее сообщение и выполнение файла прекратится.

3.3.4.2 Оператор выбора case

Оператор выбора case имеет структуру:

```
case   строка  in
шаблон11|шаблон12)  список_команд1;;
шаблон2)  список_команд2;;
    ...
*)  список_командN;;
esac
```

Здесь case, in и esac – служебные слова, строка (это может быть и один символ) сравнивается с шаблонами. Затем выполняется соответствующий список_команд выбранной строки, и управление передается на команду, идущую за ключевым словом esac. В конце

строк выбора необходимо писать ;;. Для каждой альтернативы может быть выполнено несколько команд. Если эти команды будут записаны в одну строку, то символ ; будет использоваться как разделитель команд. Обычно последняя строка выбора имеет шаблон *, что в структуре case означает "любое значение". Эта строка выбирается, если не произошло совпадение значения переменной ни с одним из ранее записанных шаблонов, ограниченных скобкой). Значения просматриваются в порядке записи. Для организации нескольких вариантов выбора в одном шаблоне необходимо разделять варианты в шаблоне символом | без пробелов между ними. Также в шаблонах могут использоваться и другие метасимволы ?, [].

В качестве примера использования данного оператора модифицируем рассмотренную ранее задачу:

```
echo -n " А какую оценку получил на экзамене?: "
read z
case $z in
    5) echo Молодец !             ;;
    4) echo Все равно молодец !   ;;
    3) echo Все равно !           ;;
    2) echo Все !                 ;;
    *) echo  !                    ;;
esac
```

Как отмечалось выше, в шаблонах могут быть использованы метасимволы. Так, в следующем примере метасимволы применяются для задания диапазона символов:

```
echo "Нажмите клавишу и затем клавишу Enter."
read Key
case $Key in
  [a-z]   ) echo "Латинская буква в нижнем регистре";;
  [A-Z]   ) echo "Латинская буква в верхнем регистре";;
  [0-9]   ) echo "Цифра";;
  *       ) echo "Знак пунктуации, пробел или что-то
другое";;
esac
```

В качестве самостоятельной работы дополните данный командный файл возможностью определения кириллических букв верхнего и нижнего регистра.

Оператор выбора case удобно использовать для реализации простейших меню. Например, функционал рассмотренного ранее командного файла по добавлению сведений в телефонный справочник можно расширить, добавив несколько различный действий в один командный файл:

```
echo Работа с телефонным справочником:
echo "1) Просмотр всех данных."
```

```
echo "2) Просмотр данных по указанному абоненту."
echo "3) Добавление данных."
echo "4) Удаление данных по указанному абоненту."
echo "5) Выход."
echo -n Ваш выбор:
read Action
Telephones=$HOME/telephones
case $Action in
    1)  cat $Telephones ;;
    2)  echo -n Введите имя:
        read FIO
        if grep $FIO $Telephones
            then :
            else echo Сведений о таком абоненте нет
        fi
        ;;
    3)  echo -n Введите имя:
        read FIO
        echo -n "Введите пол (муж/жен): "
        read pol
        if [ $pol != муж –а $pol != жен ]
            then echo "Неправильно указан пол"
                exit
        fi
        echo -n Введите телефон:
        read phone
        echo "$FIO      $pol    $phone" >> $Telephones
        ;;
    4)  echo -n Введите имя:
        read FIO
        cp $Telephones ${Telephones}.temp
        grep -v $FIO ${Telephones}.temp > $Telephones
        rm ${Telephones}.temp
        ;;
    5)  exit ;;
    *)  echo Команда выдрана неправильно ;;
esac
```

В данном примере пользователь, запустив командный файл, может выполнить одно из следующих действия: просмотреть весь справочник или только сведения об одном абоненте, удалить сведения об одном абоненте или добавить новые сведения. Добавление данных уже рассматривалось ранее, поэтому подробное описания этих действий приводить не будем. Для просмотра всех записей, содержащихся в справочнике, используется команда `cat`; для вывода сведений о конкретном человеке используется команда `grep`, причём в случае необ-

наружения данной записи, производится вывод соответствующего сообщения. Удаления записи о каком-либо абоненте производится в три этапа: копирование справочника во временный файл; выборка всех записей, не содержащих удаляемую запись, и перенос их в старый справочник с заменой его содержимого; удаление временного справочника.

Вопросы для самоконтроля

1. Напишите командный файл, который производит простейшие действия с файлами: копирование, переименование, удаление, вывод на экран. Выбор действия осуществлять с помощью оператора выбора `case`. В командном файле необходимо осуществлять проверку на то, чтобы файлы, указанные пользователем, были файлами, а не каталогами, и соответствующие действия над ними разрешены.

2. Модифицируйте программу предыдущего задания так, чтобы имена файлов, над которыми будут производиться действия, передавались с помощью позиционных параметров. Включите в командный файл проверку количества позиционных параметров и правильность используемого синтаксиса при вызове пользователем командного файла.

3.3.5 Циклическое управление

В командном интерпретаторе `shell` используются три оператора циклического управления [3, 7-11]:

- оператор цикла с перечислением `for`
- оператор цикла с истинным условием `while`
- оператор цикла с ложным условием `until`.

3.3.5.1 Оператор цикла с перечислением for

Оператор цикла `for` имеет структуру:

```
for имя [in список_значений]
do
    список_команд
done
```

где `for` – служебное слово, определяющее тип цикла, `do` и `done` - служебные слова, выделяющие тело цикла. Ключевые слова `for`, `do`, `done` пишутся с начала строки. Фрагмент `[in список_значений]` является необязательным и может отсутствовать. Если часть `[in список_значений]` опущена, то это означает `in "$@"` (то есть все позиционные параметры: `in $1 $2 ... $n`). `Имя` является параметром цикла и в теле цикла последовательно принимает значения, указанные в `список_значений`.

Например, командный файл, определяющий сумму последовательности чисел, будет иметь следующий вид:

```
for i in 1 2 3 4 5
do
    summa=`expr $summa + $i`
done
echo "Сумма равна $summa"
```

В этом примере имя `i` играет роль параметра цикла. Это имя можно рассматривать как `shell`-переменную, которой последовательно присваиваются перечисленные после служебного слова `in` значения (i=1, i=2, …, i=5)

Часто используется форма

```
for i in *
```

означающая "для всех файлов текущего каталога".

Использование данной формы оператора `for` можно проиллюстрировать следующим кодом, выводящем на экран список файлов – исходных текстов «паскалевских» программ – содержащих объявление типов данных запись:

```
for f in $ HOME/*.pas
do
    if [ -f  $f ]
    then
        if grep record $f > /dev/null
        then
            echo $f
        fi
    fi
done
```

В данном примере последовательно проверяются все файлы в домашнем каталоге пользователя, имеющие расширение `pas`. Если текущий файл является файлом и содержит слово `record`, то имя данного файла выводится на экран.

Как было указано ранее, при отсутствии в структуре оператора `for` фрагмента `[in список_значений]`, значения берутся из параметров вызывающей команды. Например, в качестве параметров вызова в следующей программе указывают список месяцев, и по именам этих месяцев выдаётся название сезонов, в которые они входят:

```
for i
do
    case $i in
        декабрь|январь|февраль) echo $i: ЗИМА ;;
        март|апрель|май) echo $i: ВЕСНА ;;
        июнь|июль|август) echo $i: ЛЕТО ;;
        сентябрь|октябрь|ноябрь) echo $i: ОСЕНЬ ;;
        *) echo $i: Название месяца не определено  ;;
    esac
done
```

Теперь, при вызове командного файла

```
$season сентябрь май июнь Март
```

на экране будет:

```
сентябрь: ОСЕНЬ
май: ВЕСНА
июнь: ЛЕТО
Март: Название месяца не определено
```

В качестве `список_значений` может выступать и результат выполнения каких-либо команд. Например, следующий командный файл производит широковещательную рассылку сообщения из файла, переданного в качестве позиционного параметра:

```
if test $# -eq 0
then
    echo Не указан файл с сообщением
    exit
fi
for i in `who|cut -b 1-17`
do
    echo Отправляю сообщение $i....
    cat $@|write $i
done
```

В данном примере оператором `if` проверяется количество позиционных параметров, переданных при вызове командного файла. Если параметров нет, то выдаётся соответствующее сообщение и прекращается дальнейшее выполнение файла. `Список_значений` циклического оператора получается посредством выполнения конвейера команд `who` и `cut -b 1-17`, первая команда возвращает список всех пользователей системы, конвейер передаёт их на вход второй команды, которая вырезает указанные подстроки из строк (в демонстрационном примере – это имена пользователей, которые указываются в первых 17 символах каждой строки вывода команды `who`). В теле цикла производится пересылка содержимого файла, указанного с помощью позиционного параметра, с помощью команды `write`. Конечно для успешной работы указанного командного файла необходимо чтобы пользователь, который получает сообщения, разрешил их приём с помощью команды `mesg`.

3.3.5.2 Оператор цикла с истинным условием while

Структура `while`, также обеспечивающая циклическое выполнение расчётов, чаще всего используется в том случае, когда неизвестен заранее точный список значений параметров или этот список должен быть получен в результате вычислений в цикле. Оператор цикла `while` имеет структуру:

```
while
```

```
        список_команд_1
    do
        список_команд_2
    done
```

где `while` – служебное слово, определяющее тип цикла с истинным условием.

Значением, проверяемым в цикле `while`, является код последней из списка команд, следующих за словом `while`. При каждой итерации выполняется `список_команд_1`. Список команд в теле цикла (между `do` и `done`) повторяется до тех пор, пока сохраняется истинность условия (то есть код завершения последней команды в теле цикла равен `0`) или цикл не будет прерван изнутри специальными командами (см. раздел 3.3.5.4). При первом входе в цикл условие должно выполняться.

Например, определение суммы последовательности чисел:

```
n=0
while [ $n -lt 5 ]    # пока n < 5
do
    n=`expr $n + 1`
    summa=`expr $summa + $n`
done
echo "Сумма равна $summa"
```

Как отмечалось выше, `список_команд_1` может состоять из нескольких операторов. Так следующий командный файл использует цикл `while` для ввода списка имён в файл:

```
    echo "Введите список имён, разделяя Enter"
    echo "Закончите ввод списка Ctrl-d"
    while
        echo –n "Следующее имя:"
        read x
    do
        echo $x >> xfile
    done
    echo "Файл содержит следующие имена:"
    cat xfile
```

Результаты выполнения программы:

```
    $ enter.name
    Введите список имён, разделяя Enter
    Закончите ввод списка Ctrl-d
    Следующее имя: Вася
    Следующее имя: Петя
    Следующее имя: <^d>
    Файл содержит следующие имена:
```

```
Вася
Петя
$
```

Если в качестве условия выполнения тела цикла установить оператор, всегда возвращающий код завершения 0, то цикл будет бесконечным. Например:

```
while true
do
    ...
done
```

или, используя пустой оператор:

```
while :
do
    ...
done
```

Бесконечные циклы могут найти своё применение при решении различных задач. Например, в разделе 3.3.4.2 был рассмотрен пример создания командного файла для работы с телефонным справочником с использованием интерактивного меню. Однако недостатком такого способа организации работы со справочником является тот факт, что за один запуск файла производится только одно действие со справочником. Используя бесконечный цикл, можно исправить данную ситуацию:

```
echo Работа с телефонным справочником:
while :
do
    echo "1) Просмотр всех данных."
    echo "2) Просмотр данных по указанному абоненту."
    echo "3) Добавление данных."
    echo "4) Удаление данных по указанному абоненту."
    echo "5) Выход."
    echo -n Ваш выбор:
    read Action
    Telephones=$HOME/telephones
    case $Action in
        1)  cat $Telephones ;;
        2)  echo -n Введите имя:
                read FIO
            if grep $FIO $Telephones
                then :
                else echo Сведений о таком абоненте нет
            fi
            ;;
        3)  echo -n Введите имя:
            read FIO
```

```
            echo -n "Введите пол (муж/жен): "
            read pol
            if [ $pol != муж –а $pol != жен ]
                then echo "Неправильно указан пол"
                    exit
            fi
            echo -n Введите телефон:
            read phone
            echo "$FIO    $pol   $phone" >> $Telephones
            ;;
        4)  echo -n Введите имя:
            read FIO
            cp $Telephones ${Telephones}.temp
            grep -v $FIO ${Telephones}.temp > $Telephones
            rm ${Telephones}.temp
            ;;
        5)  exit ;;
        *)  echo Команда указана неправильно ;;
    esac
done
```

Также оператор цикла с истинным условием удобно использовать и для построчного чтения файла от начала до конца:

```
i=1
while
    read line
do
    echo "Строка №$i: $line"
    i=`expr $i + 1`
done
```

Вызов данного командного файла производится следующим образом:

```
$ sh ReadFileToLine < OutFile
```

где `ReadFileToLine` – имя командного файла;

`OutFile` – имя файла, который необходимо прочитать построчно.

3.3.5.3 Оператор цикла с ложным условием until

Структура `until`, также обеспечивающая выполнение циклических расчётов, схожа с рассмотренным раннее оператором `while`. Оператор `until` используется тогда, когда неизвестен заранее точный список значений параметров или этот список должен быть получен в результате вычислений в цикле. Оператор цикла `until` имеет структуру:

```
until
    список_команд_1
do
```

```
        список_команд_2
    done
```

где until – служебное слово, определяющее тип цикла с ложным условием. Список команд список_команд_2 в теле цикла (между do и done) повторяется до тех пор, пока сохраняется ложность условия (условие продолжения или окончания выполнения итераций определяется результатом выполнения последней команды в список_команд_1, причём список_команд_1 выполняется в каждой итерации) или цикл не будет прерван изнутри специальными командами (см. раздел 3.3.5.4). При первом входе в цикл условие не должно выполняться.

Отличие от оператора while состоит в том, что условие цикла проверяется на ложность (на ненулевой код завершения последней команды тела цикла) проверяется после каждого (в том числе и первого) выполнения команд тела цикла.

Например, рассмотренный ранее пример определения суммы последовательности чисел при использовании оператора цикла until примет вид:

```
n=0
until [ $n -gt 5 ]    # пока n не станет больше 5
do
    n=`expr $n + 1`
    summa=`expr $summa + $n`
done
echo "Сумма равна $summa"
```

В следующем примере программа ждёт, пока не будет создан файл file1.

```
until [ -f file1]
do
    echo "Файл ещё не создан. Подожду..."
    sleep 300
done
```

Здесь условием завершения цикла является существование файла с именем file1. Каждая итерация цикла заключается в пятиминутном ожидании (300 секунд), после чего цикл повторяется снова. При этом предполагается, что какой-нибудь другой процесс в некоторый момент создаст этот файл.

Оператор цикла удобно использовать для перебора всех позиционных параметров командного файла. Следующий пример производит удаление всех файлов, имена которых переданы через позиционные параметры:

```
if [ $# -eq 0 ]          # проверяем наличие параметров
then
    echo "Нет параметров ..."
```

```
    exit
fi
until [ -z "$1" ]        # перебираем параметры
do
    if [ -f $1 ]         # если файл существует
    then
        echo "Производим удаление файла $1"
        rm –f $1
    else
        echo "Файла $1 не существует"
    fi
    shift                #сдвиг позиционных параметров
done
```

3.3.5.4 Специальные команды управления циклом

Для прерывания выполнения цикла используют специальные команды `break` и `continue`.

- Команда `break` имеет следующий синтаксис:

  ```
  break [n]
  ```

  Данная команда позволяет выходить из объемлющего цикла, если таковой существует. Необязательный параметр `n` указывает число вложенных циклов, из которых надо выйти, например,

  ```
  break 3
  ```

  означает выход из трёх вложенных циклов. Если параметр `n` отсутствует, то это эквивалентно

  ```
  break 1.
  ```

- Команда `continue` имеет аналогичный синтаксис:

  ```
  continue [n]
  ```

  Эта команда позволяет прекратить выполнение текущего цикла и возвращает управление на начало цикла. Необязательный параметр `n` указывает номер вложенного цикла, на начало которого надо перейти, например,

  ```
  continue 2
  ```

  означает выход на начало второго (если считать из глубины) вложенного цикла. Если параметр `n` отсутствует, то это эквивалентно

  ```
  continue 1.
  ```

Например:

```
while :
do
    if date|grep 12:00
    then
```

```
            echo "Время"
            break
        fi
        sleep 30
    done
```

Данная программа каждые 30 секунд проверяет текущее время: если оно равно 12:00, производится вывод фразы и выход из цикла. Данный пример можно использовать для выполнения каких-либо действий в назначенное время.

Наравне с предыдущими командами существует команда `exit`, синтаксис которой

```
exit [n]
```

Эта команда позволяет безусловно прекратить выполнение программы или процедуры и выйти из оболочки с кодом завершения `n` (признак конца файла также приведет к выходу из оболочки). Если необязательный параметр `n` не указан, то кодом завершения будет код завершения последней выполненной команды.

Методы использования команды `exit` были проиллюстрированы примерами предыдущих разделов.

Вопросы для самоконтроля

1. Напишите программу, которая просматривает телефонный справочник в файле `$HOME/telephones` на совпадение с параметрами командного файла и выводит на экран соответствующие записи. В качестве позиционных параметров передаются имена абонентов через пробел.

2. Написать командный файл, печатающий на экране список всех файлов, указанного каталога. Причём необходимо выводить пояснение о типе выводимого файла: обычный файл, специальный файл, каталог или именованный канал. Предусмотреть параметр вызова данного командного файла, при котором рекуррентно выводится информация не только текущего каталога, но и всех каталогов, входящих в текущий каталог.

3. В разделе 3.3.5.2 представлен пример использования бесконечного цикла `while` для организации работы с телефонным справочником. Модернизируйте программу, используя условие окончания работы с телефонным справочником как условие окончания цикла.

4. Командный файл для работы с телефонным справочником, созданный в ходе выполнения предыдущего задания, имеет один весомый недостаток – при удалении записей из справочника удаляются все строки, содержащие указанное имя абонента. Модернизируйте программу таким образом, чтобы удаление каждой конкретной записи из справочника сопровождалось подтверждением пользователя. Для

решения данной задачи удобно использовать построчное чтение телефонного справочника и нахождение интересующей записи, а при её нахождении – обращение к пользователю за подтверждением удаления записи.

5. Написать программу игры «Угадай число», в которой программа загадывает число в диапазоне от 1 до 99, а игрок должен за меньшее количество попыток угадать данное число. В программе предусмотреть ведения таблицы рекордов, представляющей собой отдельный файл, содержащий имя игрока, число попыток отгадывания числа и само загаданное число. Генерацию псевдослучайного числа можно осуществить следующим образом:

```
date > \dev\null
Random=`echo $$|cut -b 2-3`
```

Переменная `Random` будет содержать искомое число. При обновлении таблицы рекордов необходимо учитывать введённые ранее имена игроков: если игрок в таблице уже есть, то обновить его рекорд, если нет – добавить. Таблица должна быть отсортирована по возрастанию числа попыток угадывания числа. Для сортировки строк можно использовать команду `sort`, а для удаления информации из строк – команду `tr`.

3.3.6 Пользовательские функции

Аналогично любому другому языку программирования, в командном интерпретаторе `shell` можно использовать функции для группировки списка команд более логичным способом для последующего выполнения. [7-11]

Описание функции имеет вид:

```
имя()
{
    список_команд
}
```

Обращение к функции происходит по указанному в заголовке имени. При выполнении функции не создаётся нового процесса. Она выполняется в среде соответствующего процесса. Аргументы функции становятся её позиционными параметрами; имя функции – её нулевой параметр. Все переменные, используемые функцией, являются глобальными. Прервать выполнение функции можно оператором `return`, который имеет следующий синтаксис:

```
return [n]
```

где необязательный параметр `n` – это код возврата. Код возврата пользовательской функции доступен с помощью `$?`.

Пример использования функций. В программе создадим пользовательскую функцию, которая выводит диалоговое сообщение на экран и ждёт ответа.

```
message()
{
while true
echo –n "$1 (y-да, n-нет, c-отмена)"
read answer
do
    case $answer in
        y) return 0;;            # ответ - ДА
        n) return 1;;            # ответ - НЕТ
        c) return 2;;            # ответ - ОТМЕНА
    esac
done
}
message "Начать работу\?"
case $? in
    0) echo "Начнем, пожалуй...";;
    1|2) echo "Жаль..."
         exit;;
esac
for i in *
do
    if [ -f $i ]
    then
        message "Удалить файл $i\?"
        case $? in
            0) rm $i >> log.log;;
            1) continue;;
            2) break;;
        esac
    fi
done
```

Вопросы для самоконтроля

1. В командном файле, предназначенном для работы с телефонным справочником и созданном в рамках выполнения задания для самостоятельной работы предыдущего раздела, оформить все действия со справочником: удаление, добавление, изменение и вывод на экран – с помощью пользовательских функций.

2. Модернизируйте программу игры «Угадай число», созданную в рамках выполнения задания для самостоятельной работы предыдущего раздела, реализовав сравнение вводимого пользователем числа с загаданным программой с помощью пользовательской функции.

3.4 Инструменты отладки программы

3.4.1 Обработка прерываний trap

Командные процедуры обычно завершаются, если с терминала поступил сигнал прерывания. Однако часто бывает необходимо защитить выполнение программы от прерывания. Сигналы, поддерживающиеся ОС UNIX, приведены в таблице 18. [3, 6]

Таблица 18 – Сигналы прерываний UNIX.

Сигнал		Описание
Символьное представление	**Числовое представление**	
	0	Выход из командного интерпретатора
SIGHUP	1	Отбой (отключение удаленного абонента)
SIGINT	2	Прерывание от клавиши <Del>
SIGQUIT	3	Нестандартный выход
SIGILL	4	Неверная команда
SIGTRAP	5	Ловушка
SIGFPE	8	Исключительная ситуация при выполнении операций с плавающей запятой
SIGKILL	9	Уничтожение процесса (не перехватывается)
SIGBUS	10	Ошибка шины
SIGSEGV	11	Нарушение сегментации
SIGSYS	12	Неверный системный вызов
SIGPIPE	13	Запись в канал без чтения из него
SIGALRM	14	Будильник
SIGTERM	15	Программное завершение процесса (окончание выполнения)

Для защиты от прерываний существует команда trap, имеющая следующий синтаксис [7, 8]:

```
trap 'список_команд' сигналы
```

Если в операционной системе возникнут прерывания, чьи сигналы перечислены через пробел в параметре сигналы, то будет выполнен список_команд, после чего управление вернётся в точку прерывания и продолжится выполнение командного файла (если в списке команд не была указана команда exit). Команды выполняются по порядку номеров сигналов. Любая попытка установить сигнал, игнори-

руемый данным процессом, не обрабатывается. Попытка прерывания по сигналу 11 приводит к ошибке.

Если список_команд опущен, то все прерывания устанавливаются в их начальные значения. Если в качестве параметра список_команд указана пустая строка, то этот сигнал игнорируется командным интерпретатором и вызываемыми им программами. Если в качестве сигналы указан 0 (ноль), то список_команд выполняется при выходе из командного интерпретатора. Команда trap без аргументов выводит список команд, связанных с каждым сигналом.

Сигналы могут быть обработаны одним из трёх способов:

- их можно игнорировать (в таком случае сигнал никогда не передаётся в соответствующий процесс);
- их можно перехватывать (в этом случае процесс должен задать действия, которые необходимо выполнить при получении сигнала);
- можно не задавать реакции на сигнал (сигнал вызовет завершение процесса без выполнения каких-либо дальнейших действий).

Если к моменту входа в командную процедуру уже задано игнорирование некоторых сигналов, то команда trap для этих сигналов (как и сами эти сигналы) игнорируется.

Например, если перед прекращением по прерываниям выполнения командного файла необходимо удалить какой-то файл tempor, то это может быть выполнено командой trap:

```
trap 'rm -f tempor; exit 1' 0 1 2 3 15
```

В самой системе UNIX не использует сигнал 0; в оболочке он означает выход из командной процедуры.

Команда trap позволяет игнорировать прерывания, если список_команд пустой. Так, например, команда:

```
trap '' 1 2 3 15
```

позволит игнорировать возникновение перечисленных сигналов как самим командным файлом, так и командами, запускаемые им.

Реакцию на сигналы можно переустанавливать. Например, команда:

```
trap 2 3
```

восстановит для сигналов 2 и 3 стандартную реакцию. Перечень реакций, установленных в текущий момент, можно получить с помощью команды trap без параметров:

```
trap
```

3.4.2 Обработка ошибок

3.4.2.1 Средствами командной оболочки

Командный интерпретатор не имеет ни своего отладчика, ни отладочных команд или конструкций. Синтаксические ошибки или опе-

чатки часто вызывают сообщения об ошибках, которые практически никак не помогают при отладке. [3, 7, 8, 10]

Одним из инструментов, который может помочь при отладке неработающих сценариев, является запуск командной оболочки со специальными параметрами. Рассмотрим способы запуска командных файлов.

Проверка без выполнения

Данный способ запуска командного файла только проверит наличие синтаксических ошибок, не запуская сам файл на исполнение:

```
sh -n имя_программы
```

Того же эффекта можно добиться, вставив в сам командный файл команду

```
set -n
```

или

```
set -o noexec
```

Однако некоторые из синтаксических ошибок не могут быть выявлены таким способом.

Покомандный вывод

Данный способ запуска командного файла выводит каждую команду прежде, чем она будет выполнена:

```
sh -v имя_программы
```

Того же эффекта можно добиться, вставив в сценарий команду:

```
set -v
```

или

```
set -o verbose.
```

Ключи `-n` и `-v` могут употребляться совместно:

```
sh -nv имя_программы
```

Вывод результата каждой команды

Данный способ запуска командного файла выводит в краткой форме результат исполнения каждой команды:

```
sh -x имя_программы
```

Того же эффекта можно добиться, вставив в сценарий команду:

```
set -x
```

или

```
set -o xtrace
```

Другие параметры команды set

Однако обработка ошибок, обнаруженных командным интерпретатором, зависит от вида ошибки и от того, используется ли интерпретатор в интерактивном режиме. Интерпретатор считается интерактив-

ным, если его ввод и вывод связаны с некоторым терминалом. Интерпретатор, вызванный с флагом `-i`, также считается интерактивным.

Параметр `-e` вызывает выход из оболочки при обнаружении любой ошибки.

Отменяются любые параметры команды `set` с помощью указания снимаемого параметра со знаком `+` (плюс):

```
set +опция_без_-
```

Например, для отмены параметра проверки исходного кода без выполнения самого командного файла `-n` необходимо вызвать команду `set` следующим образом:

```
set +n
```

3.4.2.2 Отладка конвейеров

Использовать вышеперечисленные методы для отладки работы конвейеров затруднительно, так как при конвейерном выполнении команд стандартный вывод предыдущей команды перенаправляется в стандартный ввод следующей команды. В случае необходимости отладки конвейеров используется команда `tee`, которая сохраняет копию стандартного ввода в файл, имя которого дано как аргумент, в то же время не прерывая нормальной работы конвейера. [7, 8, 10]

Общий синтаксис команды `tee` следующий:

```
команда1 | tee промежуточный_файл | команда2
```

где `промежуточный_файл` – это файл, который сохраняет вывод команды для дальнейшего анализа.

Использование данной команды рассмотрим на следующем примере. Предположим, в командном файле необходимо определить идентификационные номера некоторых процессов (PID), запущенных текущим пользователем. Эту задачу можно решить с помощью такой конвейерной обработки:

```
ps -U $USER|grep sh|cut -b 1-6
```

где `ps -U $USER` – список процессов текущего пользователя;

`grep $1` – фильтр списка процессов текущего пользователя по какому-нибудь условию, задаваемому с помощью позиционного параметра при вызове командного файла;

`cut -b 1-6` – вырезание первых шести байтов – PID.

Можно воспользоваться командой `tee`, чтобы скопировать вывод каждой команды в отдельные файлы (`step1` и `step2`), не разрушая остальной конвейер:

```
ps -U $USER|tee step1|grep $1|tee step2|cut -b 1-6
```

Вызвав данный конвейер, указав вместо `$1`, например, `sh` (то есть определение PID всех процессов, где используется командный интерпретатор), получим следующие временные файлы:

```
$ ps -U $USER|tee step1|grep sh|tee step2|cut -b 1-6
```

```
 8772
 8865
$ cat step1
  PID  TT  STAT      TIME COMMAND
 8772  p0  S      0:00.37 -sh (sh)
 8863  p0  R+     0:00.07 ps -U student
 8864  p0  S+     0:00.03 tee step1
 8865  p0  S+     0:00.06 grep sh
 8866  p0  R+     0:00.04 tee step2
 8867  p0  R+     0:00.03 cut -b 1-6
$ cat step2
 8772  p0  S      0:00.37 -sh (sh)
 8865  p0  S+     0:00.06 grep sh
```

3.4.3 Необязательные параметры (ключи)

Необязательные параметры – это дополнительные ключи (опции), которые оказывают влияние на поведение командного файла и/или командного интерпретатора. [3, 7, 8, 10]

Команда `set` позволяет задавать дополнительные параметры прямо в исходном коде командного файла. В том месте программы, где необходимо, чтобы тот или иной параметр вступил в силу, используйте следующий синтаксис:

```
set -o имя_параметра
```

, или в более короткой форме:

```
set –сокращенное_имя_параметра
```

Эти две формы записи совершенно идентичны по своему действию.

Полный синтаксис команды `set`:

```
set [abefhkmnptuvxldCHP] [-o опция] [аргумент]
```

Описание всех параметров представлено в таблице 19.

Таблица 19 – Параметры команды set

Сокращённое имя параметра	Имя параметра	Описание
-a	allexport	Отмечает переменные, которые модифицированы или созданы для экспорта
-b	notify	Вызывает прекращение фоновых заданий, о котором сообщает перед выводом следующего приглашения командной строки
-e	errexit	Немедленный выход, если выходное состояние команды ненулевое
-f	noglob	выключает генерацию имени файла

Сокращённое имя параметра	Имя параметра	Описание
-h		Обнаруживает и запоминает команды как определенные функции до того, как функция будет выполнена
-k		В окружении команды располагаются все аргументы ключевых слов, не только те, которые предшествуют имени команды
-m	monitor	Разрешается управление заданиями
-n	noexec	Читает команды, но не выполняет их
-o имя_опции		Устанавливает флаг, соответствующий имени_опции
	braceexpand	Оболочка должна выполнить brace-расширение (фигурноскобочное расширение) – механизм, с помощью которого можно генерировать произвольные строки
	emacs	Использует интерфейс редактирования emacs
	ignoreeof	Оболочка не выходит при чтении EOF
	interactive-comments	Позволяет вызывать слово, начинающееся с #, и все оставшиеся символы на этой строке игнорировать в диалоговой оболочке
	posix	Изменяет режим интерпретатора в соответствии со стандартом Posix 1003.2 (предназначен для того, чтобы сделать режим строго подчиненным этому стандарту)
	vi	Использует интерфейс редактирования строки редактора vi
-p	privileged	Включает привилегированный режим. В этом режиме файл $ENV не выполняется, и функции оболочки не наследуются из среды. Это включается автоматически начальными действиями, если идентификатор эффективного пользователя (группы) не равен идентификатору реального пользователя (группы). Выключение этой опции присваивает

Сокращённое имя параметра	Имя параметра	Описание
		идентификатор эффективного пользователя (группы) идентификатору реального пользователя (группы).
-t		Выход после чтения и выполнения команды
-u	nounset	Во время замещения рассматривает незаданную переменную как ошибку
-v	verbose	Выдает строки ввода оболочки по мере их считывания
-x	xtrace	Выводит команды и их аргументы по мере выполнения команд
-l		Сохраняет и восстанавливает связывание имени в команде for
-d	nohash	Выключает хеширование команд, найденных для выполнения. Обычно команды запоминаются в хеш-таблице и, будучи однажды найденными, больше не ищутся
-C	noclobber	Не позволяет существующим файлам перенаправление вывода
-H	histexpand	Закрывает замену стиля истории. Этот параметр принимается по умолчанию
-P	physical	Если установлен, не следует символьному указателю при выполнении команды типа cd, которая изменяет текущий каталог. Вместо этого используется физический каталог
--		Если нет аргументов, следующих за этим параметром, то не задаются позиционные параметры. В противном случае позиционные параметры присваиваются аргументам, даже если некоторые из них начинаются с a-
-		Сигнал конца параметра, вызывающего присваивание оставшихся аргументов позиционным параметрам. Параметры -x и -v выключаются. Если здесь нет аргументов, позиционный параметр не изменяется

При использовании + вместо – осуществляется выключение этих параметров. Параметры также могут использоваться при вызове интерпретатора. Текущий набор параметров может быть получен пользователем с помощью специальной переменной $-.

В синтаксисе команды set также используется необязательный параметр аргумент, который представляет собой новые позиционные переменные $1, $2 и т.д. командного файла. Если все аргументы опущены, выводятся значения всех переменных.

Вопросы для самоконтроля

1. С помощью оператора trap выведите какое-либо сообщение при завершении выполнения командного файла.

2. С помощью оператора trap произведите автоматическую сортировку телефонного справочника (файл $HOME/telephones, полученный в ходе выполнения заданий для самостоятельной работы предыдущих разделов) при завершении выполнения командного файла, предназначенного для работы со справочником.

3. С помощью соответствующих параметров команды set задайте проверку синтаксиса и вывода каждой команды программы игры «Угадай число», созданной в рамках выполнения задания для самостоятельной работы предыдущего раздела. Прокомментируйте вывод программы.

Список литературы

1. Roman aka Docent. История UNIX // «Спецвыпуск: Хакер», №047, стр. 4
2. TanaT Вселенная UNIX. Эту историю должен знать каждый! // «Хакер», №49, стр. 78
3. Баурн С. Операционная система UNIX. – М.: Мир, 1986, 464 стр.
4. Дегтярев Е.К. Введение в UNIX. М.: – 1991, 156 с.
5. Кристиан К. Операционная система UNIX. – М.: Финансы и статистика, 1985, 320 стр.
6. Магда Ю.С. Администрирование UNIX. – СПб.: БХВ-Петербург, 2005. – 800 с.
7. А. Соловьев Программирование на shell (UNIX)
8. НПО "КЛОТО" Интерпретатор командного языка shell
9. А. Семенюченко Шелл для кодера // «Спецвыпуск: Хакер», №51, стр. 82
10. Mendel Cooper Искусство программирования на языке сценариев командной оболочки / Перевод с англ. Андрей Киселев
11. Гребенников Р. Кодим в Bash! // «Хакер», №55, стр. 55

Учебное издание

Ляховец Михаил Васильевич

ПРОГРАММИРОВАНИЕ В UNIX-СИСТЕМАХ

Учебное пособие

Редактор _____

Подписано в печать «___»_____ 20___г.
Формат бумаги 60×84 1/16. Бумага писчая. Печать офсетная.
Усл. печ. л. ___ Уч.-изд. л. ____ Тираж ___ экз. Заказ _____.

Сибирский государственный индустриальный университет,
654007, г. Новокузнецк, ул. Кирова, 42.
Типография СибГИУ