Зачёт №1
Утилиты и их синтаксис:

1. **cmp –** Позволяет осуществить побайтовое сравнение содержимого двух файлов
   Синтаксис: **cmp [*параметры*] *файл1 файл2***
2. **comm –** Читает файл1 и файл2, которые должны быть предварительно лексически отсортированы, и генерирует вывод, состоящий из трёх колонок текста: строки, найденные только в файле файл1; строки, найденные только в файле файл2; и строки, общие для обоих файлов
   Синтаксис: **comm [*параметры*] *файл1 файл2***
3. **wc –** Подсчёт количества слов, строк, символов и байтов
   Синтаксис: **wc [*параметры*] [*файл...*]**
4. **dd –** Аналог утилиты копирования файлов cp только для блочных данных. Утилита переносит по одному блоку данных указанного размера с одного места в другое
   Синтаксис: **dd if=[*источник_копирования*] of=[*место_назначения*] [*параметры*]**
5. **diff –** Сравнивает файлы, выводит разницу между ними
   Синтаксис: **diff [*параметры*] *файл1 файл2***
6. **grep –** Поиск строк по шаблонам
   Синтаксис: **grep [*параметры*] *выражение* [*файл...*]**
7. **join –** Объединение отсортированных файлов
   Синтаксис: **join [*параметры*] *файл1 файл2***
8. **sort –** Сортировка, объединение или проверка последовательности текстовых файлов
   Синтаксис: **sort [*параметры*] *файл..***
9. **tail –** Копирование последней части файла
   Синтаксис: **tail [*параметры*] *файл***
10. **tee –** Копирует стандартный поток ввода в стандартный поток вывода, при этом создается копия в файлах в количестве от нуля и выше. Вывод утилиты в буфер не записывается. Утилита tee обычно используется в конвейере для создания копии вывода утилиты
    Синтаксис: **tee [*параметры*] *файл***
11. **tr –** Преобразование символов
    Синтаксис: **tr [*параметры*] *строка1 строка2***
12. **uniq – uniq [*параметры*] [*входной_файл*] [*выходной_файл*]**
    Синтаксис: **uniq [*параметры*] [*входной_файл*] [*выходной_файл*]**
13. **od –** Дамп файла в различных форматах
    Синтаксис: **od [*параметры*] [*входной_файл*]**
14. **sum –** Вывод контрольной суммы и числа блоков
    Синтаксис: **sum [*параметры*] *файл...***
15. **cut –** Удаление выбранных полей в каждой строке файла
    Синтаксис: **cut [*список*] [*разделитель*] [*файл...*]**
16. **nroff –** Используется для форматирования документов для отображения или печати с фиксированной шириной. Используется для форматирования текста в man-страницах.
    Синтаксис: **nroff [*параметры*] *файл...***
17. **vim –** Текстовый редактор, обратно-совместимый с Vi
    Синтаксис: **vim [*параметры*] *файл...***
18. **mc –** Midnight Commander - программа, предназначенная для просмотра содержимого каталогов и выполнения основных функций управления файлами в UNIX-подобных операционных системах
    Синтаксис: **mc [*параметры*]**
19. **tar –** GNU версия утилиты архивирования, ввести можно только один ключ
    Синтаксис: **tar [*параметры*] *файл/директория...***
20. **gzip –** сжать или распаковать файлы
    Синтаксис: **gzip [*параметры*] *файл..***

21. **ed –** текстовый редактор
Синтаксис: **ed [*параметры*] *файл...***
22. **awk –** язык поиска и обработки шаблонов
Синтаксис: **awk [*параметры*] *файл..***
23. **sed –** потоковый редактор (Stream EDitor)
Синтаксис: **sed [*параметры*] *файл..***
24. **bzip2 –** сжимает и распаковывает файлы.
Синтаксис: **bzip2 [*параметры*] *файл..***
25. **head –** печатает первые 10 строк файла
Синтаксис: **head [*параметры*] *файл..***
26. **iconv –** преобразует кодировку указанных файлов
Синтаксис: **iconv [*параметры*] -f [*исходная_кодировка*] -t [*конечная_кодировка*] *файл..***
27. **patch –** применить изменения к текстовым файлам
Синтаксис: **patch [*параметры*] *файл..***
28. **md5 –** один из серии алгоритмов по построению дайджеста сообщения, разработанный профессором Рональдом Л. Ривестом из Массачусетского технологического института.
Синтаксис: **md5 [*параметры*] *файл..***
29. **du –** оценка места на диске, занимаемого файлом
Синтаксис: **du [*параметры*] *файл..***
30. **file –** уточнить тип файла
Синтаксис: **file [*параметры*] *файл..***
31. **touch –** изменяет временные штампы каждого заданного файла
Синтаксис: **touch [*параметры*] *файл..***
32. **find –** искать заданные файлы
Синтаксис: **find [*параметры*] *путь...* [*выражение_операнда*]**
33. **xargs –** Построение списка (списков) аргументов и вызов программы (POSIX)
Синтаксис: **xargs [*параметры*] [*программа* [*начальные_аргументы*]]**
34. **df** - отчёт об использовании дискового пространства
Синтаксис: **df [*параметры*] *файл..***
35. **paste –** объединить строки файлов
Синтаксис: **paste [*параметры*] *файл..***
36. **cpp –** препроцессор языка Си
Синтаксис: **cpp [*параметры*] [*входной_файл* [*выходной_файл*]]**
37. **indent –** Форматирование исходного текста на языке C
Синтаксис: **indent [*входной_файл* [*выходной_файл*]] [*параметры*]**
38. **split –** Разделение файлов на несколько частей (POSIX)
Синтаксис: **split [*параметры*] *файл..***
39. **mktemp –** создает уникальное временное имя файла
Синтаксис:
#include <stdlib.h>

char *mktemp(char **template*);

Вопросы

6.
1) Переменные -- это одна из основ любого языка программирования. Они учавствуют в арифметических операциях, в синтаксическом анализе строк и совершенно необходимы для абстрагирования каких либо величин с помощью символических имен. Физически

переменные представляют собой ни что иное как участки памяти, в которые записана некоторая информация.

2) Если variable1 -- это имя переменной, то $variable1 -- это ссылка на ее значение. "Чистые" имена переменных, без префикса $, могут использоваться только при объявлении переменный, при присваивании переменной некоторого значения, при удалении (сбросе), при экспорте и в особых случаях

Присваивание может производится с помощью символа = (например: var1=27), инструкцией read и в заголовке цикла (for var2 in 1 2 3).
Примечательно, что написание $variable фактически является упрощенной формой написания ${variable}. Более строгая форма записи ${variable} может с успехом использоваться в тех случаях, когда применение упрощенной формы записи порождает сообщения о синтаксических ошибках

```
#!/bin/bash

# Присваивание значений переменным и подстановка значений переменных

a=375
hello=$a

#-----------------------------------------------------------------------
# Использование пробельных символов
# с обеих сторон символа "=" присваивания недопустимо.

#  Если записать "VARIABLE =value",
#+ то интерпретатор попытается выполнить команду "VARIABLE" с параметром
"=value".

#  Если записать "VARIABLE= value",
#+ то интерпретатор попытается установить переменную окружения "VARIABLE" в ""
#+ и выполнить команду "value".
#-----------------------------------------------------------------------


echo hello    # Это не ссылка на переменную, выведет строку "hello".

echo $hello
echo ${hello} # Идентично предыдущей строке.

echo "$hello"
echo "${hello}"

echo

hello="A B  C   D"
echo $hello   # A B C D
echo "$hello" # A B  C   D
# Здесь вы сможете наблюдать различия в выводе echo $hello и echo "$hello".
# Заключение ссылки на переменную в кавычки сохраняет пробельные символы.

echo
```

```
echo '$hello'  # $hello
# Внутри одинарных кавычек не производится подстановка значений переменных,
#+ т.е. "$" интерпретируется как простой символ.

# Обратите внимание на различия, существующие между этими типами кавычек.


hello=    # Запись пустого значения в переменную.
echo "\$hello (пустое значение) = $hello"
#  Обратите внимание: запись пустого значения -- это не то же самое,
#+ что сброс переменной, хотя конечный результат -- тот же (см. ниже).

# --------------------------------------------------------------

#  Допускается присваивание нескольких переменных в одной строке,
#+ если они отделены пробельными символами.
#  Внимание! Это может снизить читабельность сценария и оказаться непереносимым.

var1=variable1  var2=variable2  var3=variable3
echo
echo "var1=$var1   var2=$var2  var3=$var3"

# Могут возникнуть проблемы с устаревшими версиями "sh".

# --------------------------------------------------------------

echo; echo

numbers="один два три"
other_numbers="1 2 3"
# Если в значениях переменных встречаются пробелы,
# то использование кавычек обязательно.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"   # other_numbers = 1 2 3
echo

echo "uninitialized_variable = $uninitialized_variable"
# Неинициализированная переменная содержит "пустое" значение.
uninitialized_variable=   #  Объявление неинициализированной переменной
                          #+ (то же, что и присваивание пустого значения, см. выше).
echo "uninitialized_variable = $uninitialized_variable"
                          # Переменная содержит "пустое" значение.

uninitialized_variable=23       # Присваивание.
unset uninitialized_variable    # Сброс.
echo "uninitialized_variable = $uninitialized_variable"
                            # Переменная содержит "пустое" значение.

echo

exit 0
```

3) *локальные переменные*
переменные, область видимости которых ограничена блоком кода или телом функции (см так же локальные переменные в функциях)

переменные окружения
переменные, которые затрагивают командную оболочку и порядок взаимодействия с пользователем

Note
В более общем контексте, каждый процесс имеет некоторое "окружение" (среду исполнения), т.е. набор переменных, к которым процесс может обращаться за получением определенной информации. В этом смысле командная оболочка подобна любому другому процессу.

Каждый раз, когда запускается командный интерпретатор, для него создаются переменные, соответствующие переменным окружения. Изменение переменных или добавление новых переменных окружения заставляет оболочку обновить свои переменные, и все дочерние процессы (и команды, исполняемые ею) наследуют это окружение.

Caution
Пространство, выделяемое под переменные окружения, ограничено. Создание слишком большого количества переменных окружения или одной переменной, которая занимает слишком большое пространство, может привести к возникновению определенных проблем.

bash$ eval "`seq 10000 | sed -e 's/.*/export var&=ZZZZZZZZZZZZZ/'`"

bash$ du
bash: /usr/bin/du: Argument list too long

7.

8.
1)
2)Присваивание значений переменным простое и замаскированное

```
#!/bin/bash

a=23            # Простейший случай
echo $a
b=$a
echo $b

# Теперь немного более сложный вариант (подстановка команд).

a=`echo Hello!`   # В переменную 'a' попадает результат работы команды 'echo'
echo $a
#  Обратите внимание на восклицательный знак (!) в подстанавливаемой команде
#+ этот вариант не будет работать при наборе в командной строке,
#+ поскольку здесь используется механизм "истории команд" BASH
#  Однако, в сценариях, механизм истории команд запрещен.

a=`ls -l`       # В переменную 'a' записывается результат работы команды 'ls -l'
echo $a         # Кавычки отсутствуют, удаляются лишние пробелы и пустые строки.
```

echo
echo "$a" # Переменная в кавычках, все пробелы и пустые строки сохраняются.
 # (См. главу "Кавычки.")

exit 0
Присваивание переменных с использованием $(...) (более современный метод, по сравнению с обратными кавычками)

 # Взято из /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
3)*Встроенный документ* (here document) является специальной формой перенаправления ввода/вывода, которая позволяет передать список команд интерактивной программе или команде, например ftp, telnet или ex. Конец встроенного документа выделяется "строкой-ограничителем", которая задается с помощью специальной последовательности символов <<. Эта последовательность -- есть перенаправление вывода из файла в программу, напоминает конструкцию interactive-program < command-file, где command-file содержит строки:

command #1
command #2
...

Сценарий, использующий "встроенный документ" для тех же целей, может выглядеть примерно так:

#!/bin/bash
interactive-program <<LimitString
command #1
command #2
...
LimitString

В качестве строки-ограничителя должна выбираться такая последовательность символов, которая не будет встречаться в теле "встроенного документа".

Обратите внимание: использование встроенных документов может иногда с успехом применяться и при работе с неинтерактивными командами и утилитами.

9. арифметические операторы

+
сложение

-
вычитание

*
умножение

/

деление

**
возведение в степень

В Bash, начиная с версии 2.02, был введен оператор возведения в степень -- "**".

```
let "z=5**3"
echo "z = $z"   # z = 125
```

%
модуль (деление по модулю), возвращает остаток от деления

```
bash$ echo `expr 5 % 3`
2
```

```
#!/bin/bash
# Базовая арифметика, использующая let

let a=5+4
echo $a # 9

let "a = 5 + 4"
echo $a # 9

let a++
echo $a # 10

let "a = 4 * 5"
echo $a # 20

let "a = $1 + 30"
echo $a # 30 + первый аргумент командной строки
```

10. 1)Практически любой язык программирования включает в себя условные операторы, предназначенные для проверки условий, чтобы выбрать тот или иной путь развития событий в зависимости от этих условий. В Bash, для проверки условий, имеется команда test, различного вида скобочные операторы и условный оператор if/then.

Оператор if/then проверяет -- является ли код завершения списка команд 0 (поскольку 0 означает "успех"), и если это так, то выполняет одну, или более, команд, следующие за словом then.

Существует специальная команда -- [(левая квадратная скобка). Она является синонимом команды test, и является встроенной командой (т.е. более эффективной, в смысле производительности). Эта команда воспринимает свои аргументы как выражение сравнения или как файловую проверку и возвращает код завершения в соответствии с результатами проверки (0 -- истина, 1 -- ложь).

Начиная с версии 2.02, Bash предоставляет в распоряжение программиста конструкцию [[...]] расширенный вариант команды test, которая выполняет сравнение способом более

знакомым программистам, пишущим на других языках программирования. Обратите внимание: [[-- это зарезервированное слово, а не команда.

Bash исполняет [[$a -lt $b]] как один элемент, который имеет код возврата.

Круглые скобки ((...)) и предложение let ... так же возвращают код 0, если результатом арифметического выражения является ненулевое значение. Таким образом, арифметические выражения могут учавствовать в операциях сравнения.

Предложение let "1<2" возвращает 0 (так как результат сравнения "1<2" -- "1", или "истина")
((0 && 1)) возвращает 1 (так как результат операции "0 && 1" -- "0", или "ложь")

Условный оператор if проверяет код завершения любой команды, а не только результат выражения, заключенного в квадратные скобки.

```
if cmp a b &> /dev/null  # Подавление вывода.
then echo "Файлы a и b идентичны."
else echo "Файлы a и b имеют различия."
fi

if grep -q Bash file
then echo "Файл содержит, как минимум, одно слово Bash."
fi

if COMMAND_WHOSE_EXIT_STATUS_IS_0_UNLESS_ERROR_OCCURRED
then echo "Команда выполнена успешно."
else echo "Обнаружена ошибка при выполнении команды."
fi
```

Оператор if/then допускает наличие вложенных проверок.

```
if echo "Следующий *if* находится внутри первого *if*."

  if [[ $comparison = "integer" ]]
    then (( a < b ))
  else
    [[ $a < $b ]]
  fi

then
  echo '$a меньше $b'
fi
```

2)Действие, когда какая либо команда или сама командная оболочка инициирует (порождает) новый подпроцесс, что бы выполнить какую либо работу, называется ветвлением (forking) процесса. Новый процесс называется "дочерним" (или "потомком"), а породивший его процесс -- "родительским" (или "предком"). В результате и потомок и предок продолжают исполняться одновременно -- параллельно друг другу.

В общем случае, встроенные команды Bash, при исполнении внутри сценария, не порождают новый подпроцесс, в то время как вызов внешних команд, как правило, приводит к созданию нового подпроцесса.
11.
Цикл -- это блок команд, который исполняется многократно до тех пор, пока не будет выполнено условие выхода из цикла.

1)циклы for

for (in)
Это одна из основных разновидностей циклов. И она значительно отличается от аналога в языке C.

```
for arg in [list]
do
 команда(ы)...
done
```

Note
На каждом проходе цикла, переменная-аргумент цикла arg последовательно, одно за другим, принимает значения из списка list.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# На первом проходе, $arg = $var1
# На втором проходе, $arg = $var2
# На третьем проходе, $arg = $var3
# ...
# На N-ном проходе, $arg = $varN

# Элементы списка заключены в кавычки для того, чтобы предотвратить возможное разбиение их на отдельные аргументы (слова).
```

Элементы списка могут включать в себя шаблонные символы.

Есл ключевое слово do находится в одной строке со словом for, то после списка аргументов (перед do) необходимо ставить точку с запятой.

```
for arg in [list] ; do
```

2)while
Оператор while проверяет условие перед началом каждой итерации и если условие истинно (если код возврата равен 0), то управление передается в тело цикла. В отличие от циклов for, циклы while используются в тех случаях, когда количество итераций заранее не известно.

```
while [condition]
do
 command...
done
```

Как и в случае с циклами for/in, при размещении ключевого слова do в одной строке с объявлением цикла, необходимо вставлять символ ";" перед do.

```
while [condition] ; do
```

Обратите внимание: в отдельных случаях, таких как использование конструкции getopts совместно с оператором while, синтаксис несколько отличается от приводимого здесь.

Пример 10-14. Простой цикл while

```
#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
do
  echo -n "$var0 "        # -n подавляет перевод строки.
  var0=`expr $var0 + 1`   # допускается var0=$(($var0+1)).
done

echo

exit 0
```

Пример 10-15. Другой пример цикла while

```
#!/bin/bash

echo

while [ "$var1" != "end" ]     # возможна замена на while test "$var1" != "end"
do
  echo "Введите значение переменной #1 (end - выход) "
  read var1                    # Конструкция 'read $var1' недопустима (почему?).
  echo "переменная #1 = $var1" # кавычки обязательны, потому что имеется символ "#".
  # Если введено слово 'end', то оно тоже выводится на экран.
  # потому, что проверка переменной выполняется в начале итерации (перед вводом).
  echo
done

exit 0
```

Оператор while может иметь несколько условий. Но только последнее из них определяет возможность продолжения цикла. В этом случае синтаксис оператора цикла должен быть несколько иным.

12.

1) Инструкции case и select технически не являются циклами, поскольку не предусматривают многократное исполнение блока кода. Однако, они, как и циклы, управляют ходом исполнения программы, в зависимости от начальных или конечных условий.

case (in) / esac
Конструкция case эквивалентна конструкции switch в языке C/C++. Она позволяет выполнять тот или иной участок кода, в зависимости от результатов проверки условий. Она является, своего рода, краткой формой записи большого количества операторов if/then/else и может быть неплохим инструментом при создании разного рода меню.

```
case "$variable" in

 "$condition1" )
 command...
 ;;

 "$condition2" )
 command...
 ;;

esac
```

Note
Заключать переменные в кавычки необязательно, поскольку здесь не производится разбиения на отдельные слова.

Каждая строка с условием должна завершаться правой (закрывающей) круглой скобкой).

Каждый блок команд, отрабатывающих по заданному условию, должен завершаться двумя символами точка-с-запятой ;;.

Блок case должен завершаться ключевым словом esac (case записанное в обратном порядке).

Пример 10-24. Использование case

```
#!/bin/bash

echo; echo "Нажмите клавишу и затем клавишу Return."
read Keypress

case "$Keypress" in
  [a-z]   ) echo "буква в нижнем регистре";;
  [A-Z]   ) echo "Буква в верхнем регистре";;
  [0-9]   ) echo "Цифра";;
  *       ) echo "Знак пунктуации, пробел или что-то другое";;
esac  # Допускается указыватль диапазоны символов в [квадратных скобках].
```

```
# Упражнение:
# --------
# Сейчас сценарий считывает нажатую клавишу и завершается.
# Измените его так, чтобы сценарий продолжал отвечать на нажатия клавиш,
# но завершался бы только после ввода символа "X".
# Подсказка: заключите все в цикл "while".

exit 0
```

2) select
Оператор select был заимствован из Korn Shell, и является еще одним инструментом, используемым при создании меню.

```
select variable [in list]
do
 command...
 break
done
```

Этот оператор предлагает пользователю выбрать один из представленных вариантов. Примечательно, что select по-умолчанию использует в качестве приглашения к вводу (prompt) -- PS3 (#?), который легко изменить.

Пример 10-29. Создание меню с помощью select

```
#!/bin/bash

PS3='Выберите ваш любимый овощ: ' # строка приглашения к вводу (prompt)

echo

select vegetable in "бобы" "морковь" "картофель" "лук" "брюква"
do
  echo
  echo "Вы предпочитаете $vegetable."
  echo ";-))"
  echo
  break  # если 'break' убрать, то получится бесконечный цикл.
done

exit 0
```

Если в операторе select список in list не задан, то в качестве списка будет использоваться список аргументов ($@), передаваемый сценарию или функции.

Сравните это с поведением оператора цикла

```
for variable [in list]
```

в котором не задан список аргументов.