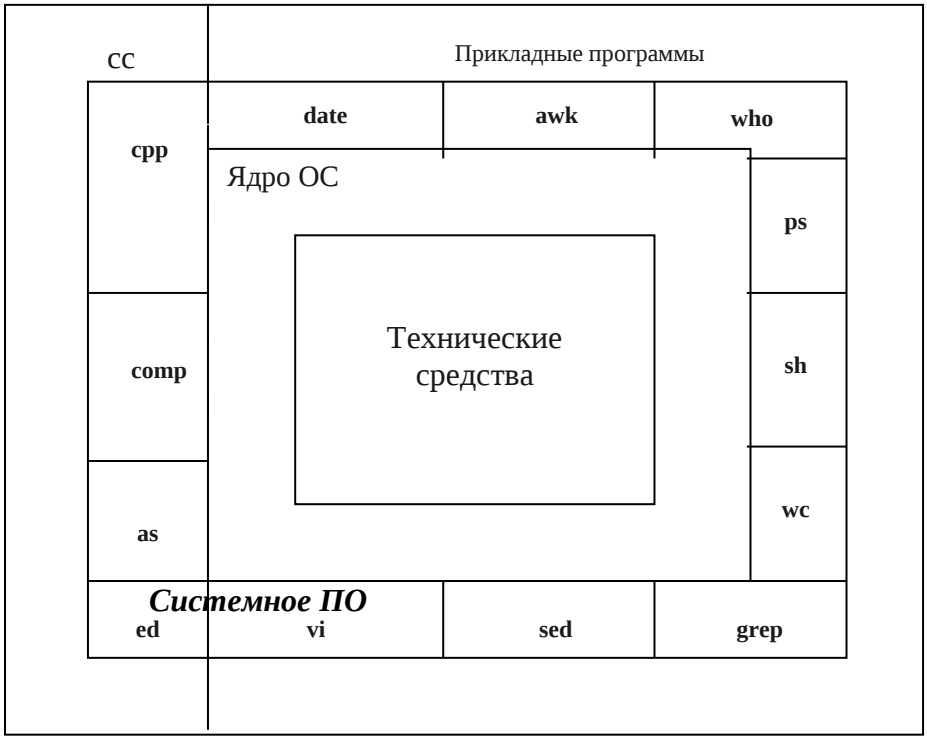
СОДЕРЖАНИЕ

Введение в Unix

За время, прошедшее с момента ее появления в 1969 году, система UNIX стала довольно популярной и получила распространение на машинах с различной мощностью обработки, от микропроцессоров до больших ЭВМ, обеспечивая на них общие условия выполнения программ.

Система делится на две части. Одну часть составляют программы и сервисные функции, то, что делает операционную среду UNIX такой популярной; эта часть легко доступна пользователям, она включает такие программы, как командный процессор, обмен сообщениями, пакеты обработки текстов и системы обработки исходных текстов программ. Другая часть включает в себя собственно операционную систему, поддерживающую эти программы и функции.

Архитектура UNIX



Технические средства, показанные в центре диаграммы, выполняют функции, обеспечивающие функционирование операционной системы.

Операционная система взаимодействует с аппаратурой, непосредственно обеспечивая обслуживание программ и их независимость от деталей аппаратной конфигурации. Если представить систему состоящей из пластов, в ней можно выделить системное ядро, изолированное от пользовательских программ. Поскольку программы не зависят от аппаратуры, их легко переносить из одной системы UNIX в другую, функционирующую на другом комплексе технических средств, если только в этих программах не подразумевается работа с конкретным оборудованием. Например, программы, рассчитанные на определенный размер машинного слова, гораздо труднее

переводить на другие машины по сравнению с программами, не требующими подобных установлений.

Программы, подобные командному процессору shell и редакторам (ed и vi) и показанные на внешнем по отношению к ядру слое, взаимодействуют с ядром при помощи хорошо определенного набора обращений к операционной системе. Обращения к операционной системе понуждают ядро к выполнению различных операций, которых требует вызывающая программа, и обеспечивают обмен данными между ядром и программой.

Некоторые из программ, приведенных на рисунке, в стандартных конфигурациях системы известны как команды, однако на одном уровне с ними могут располагаться и доступные пользователю программы, такие как программа a.out, стандартное имя для исполняемого файла, созданного компилятором с языка Си. Другие прикладные программы располагаются выше указанных программ, на верхнем уровне, как это показано на рисунке. Например, стандартный компилятор с языка Си ('cc') располагается на самом внешнем слое: он вызывает препроцессор для Си, ассемблер и загрузчик (компоновщик), т.е. отдельные программы предыдущего уровня.

Стандартизация

Стандартизация применима для:

1. Переноса приложений на широкий диапазон систем
2. Совместной работы приложений локальной или глобальной сети
3. Для взаимодействия с пользователем в стиле, облегчающим ему переход от одной системы к другой

Для обеспечения этого существуют открытые спецификации на интерфейс.

Термин “спецификация открыта” означает, что она общедоступна и находится под контролем общественности.

Открытая система, которой является UNIX, должна поддерживать открытые спецификации.

Открытая спецификация не зависит от программного и аппаратного обеспечения.

Файловая система

Файловая система UNIX в частности характеризуется:

- иерархической структурой
- согласованной обработкой массивов данных
- возможностью создания и удаления файлов
- динамическим расширением файлов
- защитой информации в файлах
- трактовкой периферийных устройств (таких как терминалы и ленточные устройства) как файлов.

Корневой каталог имеет вид:

```
/ - корневой каталог
|
|-bin      Двоичные коды наиболее важных команд
|-boot     Статические файлы загрузчика boot
|-dev      Файлы устройств
```

|-etc Файлы настройки конфигурации системы
|-home Домашние каталоги пользователей
|-lib Разделяемые библиотеки
|-mnt Точка монтирования временно подключаемых систем
|-proc Псевдо-файловая система с информацией о процессах
|-root Домашний каталог (пользователя) root
|-sbin Наиболее важные системные двоичные коды
|-tmp Временные файлы
|-var Переменные данные
|-usr / Вторая главная иерархия
|- bin
|- sbin
|- lib
|- local Программы, установленные в системе дополнительно, помимо основного дистрибутива

Каждый из этих, а также расположенных на других уровнях каталогов имеет строго определенное назначение, что обеспечивает удобство работы с файловой системой.

Unix – многопользовательская система , можно задавать несколько пользователей.

// Root – главный пользователь.

У каждого файла указывается идентификатор пользователя (UID) и идентификатор группы (GID).

Кроме того, файлу присваивают атрибуты доступа.

Три вида атрибутов.

1. Для владельца;
2. Для группы владельцев;
3. Для остальных.

Атрибуты:

1. Чтение -r
2. Запись -w
3. Исполнение -x

Пример:

$ /bin/ls

rwx r-x r-x

1. первая триада "rwx" разрешает владельцу каталога: r - читать, w - писать и x - выполнять (более точно, для файлов типа каталог w означает разрешение создавать файлы в каталоге и удалять их из него, а x разрешает доступ к файлам каталога);
2. вторая триада отражает права доступа членам группы, в которую входит владелец, которым разрешено только читать и выполнять (запрещено писать в этот файл, т.е. изменять содержимое каталога).
3. последняя триада отражает права доступа прочих пользователей, которым разрешено тоже что и в 2.

Моды доступа кодируются в восьмеричной системе счисления, например:

rwx 111 7
r-x 101 5
rw- r-- --- 640

Для их изменения используется команда chmod.

chmod 640 <файл> Установка мод доступа (файл должен быть разрешен на запись)

При установке мод доступа необходимо учесть, что в каталогах моды x обязательно должны быть установлены.

Аутентификация.

Список пользователей лежит в /etc/passwd и примитивная информация : имя пользователя, комментарий, домашний каталог, оболочка ,UID и GID .

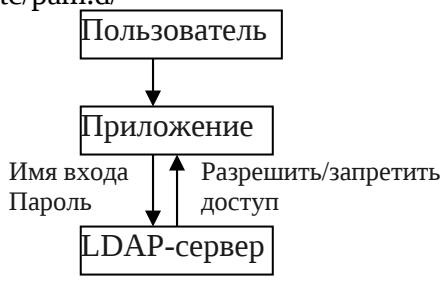
/etc/shadow хранится пароль в зашифрованном виде. Открыт для чтения для root.

/etc/group хранится: название группы, права, идентификатор, номера пользователей .

Аутентификация – это процедура, проверяющая, имеет ли пользователь с предъявленным идентификатором право на доступ к какому-либо ресурсу.

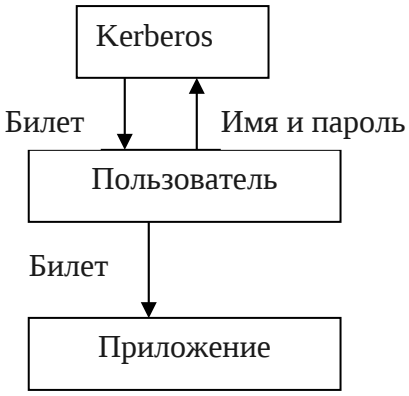
PAM Подключаемый модуль аутентификации

etc/pam.d/



Пользователь предоставляет приложению свои имя и пароль, после чего приложение отправляет эту информацию LDAP-серверу, который разрешет или запрещает доступ этого пользователя к приложению. В качестве LDAP-сервера используются Open-LDAP в паре с Samba (в Unix-подобных системах) либо Active Directory (в Windows).

Возникает проблема безопасности: под необходимое пользователю приложение может подстроиться вредоносное, получающее имена и пароли пользователей. Для предотвращения такой ситуации используют протокол Kerberos, который на имя и пароль выдает пользователю «билет» на доступ к приложению.



Shell

Shell - это одна из многих команд UNIX. То есть в набор команд оболочки (интерпретатора) 'shell' входит команда 'sh' - вызов интерпретатора 'shell'. Первый 'shell' вызывается автоматически при входе в систему и выдает на экран промтер. После этого можно вызывать на выполнение любые команды, в том числе и снова сам 'shell', который создаст новую оболочку внутри прежней. С помощью shell доступно полное манипулирование системой.

Простейшие средства Shell

Shell-макроязык, его команды находятся в каталоге /bin или /bin/usr и т.п.

- ; последовательное выполнение команд

$ leep 20;reboot

- & асинхронное (фоновое) выполнение предшествующей команды

$ cc pr.c &

- && выполнение команды (k2) при условии нормального завершения предыдущей (k2)

$ k1&&k2

- || выполнение последующей команды при условии ненормального завершения предыдущей

$ cp a b || echo “Error”

- {} для группировки команд в блок

$ k1 && {k2; k3}

Здесь обе команды ("k2" и "k3") будут выполнены только при успешном завершении "k1".

- () кроме выполнения функции группировки, выполняют и функцию вызова нового экземпляра интерпретатора shell

Пусть мы находились в начальном каталоге "/mnt/lab"

(cd ..; ls) ls выдаст сначала содержимое каталога "/mnt", а затем содержимое "/mnt/lab", т.к. при входе в скобки вызывается новый экземпляр shell, в рамках которого и осуществляется переход. При выходе из круглых скобок происходит возврат в старый shell и в старый каталог

- . для выполнения программы из файла в текущем Shell–е

$. $1.sh

Стандартный вход и выход. Перенаправление.

Пользователь имеет удобные средства перенаправления ввода и вывода на другие файлы (устройства).

- > перенаправление вывода.

$ ls >f1.test

команда "ls" сформирует список файлов текущего каталога и поместит его в файл "f1". Если файл "f1" до этого существовал, то он будет затерт новым.

- >> перенаправление вывода, но файл при этом не обнуляется.

$ pwd >>f1

команда pwd сформирует полное имя текущего каталога и поместит его в конец файла "f1", т.е. ">>" добавляет в файл, если он непустой.

- < и << перенаправление ввода

$ wc -l <f1

подсчитает и выдаст на экран число строк в файле f1.

$ ed f2 <<!

создаст с использованием редактора файл "f2", непосредственно с терминала. Окончание ввода определяется по символу, стоящему правее "<<" . То есть ввод будет закончен, когда первым в очередной строке будет "!".

- | Конвейер

Средство, объединяющее стандартный выход одной команды (k1) со стандартным входом другой(k2)

$ k1|k2

Классификация команд

1. команды для работы с файловой системой
2. команды для работы с текстовыми файлами
3. тестирующие команды

1.Для работы с файловой системой используются команды:

- cd смена директории
- pwd выдать текущую директорию
- mkdir создать директорию
- cp копирование файла
- mv перенос (переименование) файла
- rm удалить файл

  rm –r рекурсивное удаление

  rm-i спрашивает разрешение на удаление

  rm-f удалять не спрашивая

  $ rm-f *.b
- rmdir удалить директорию
- ls выдать содержимое директории, по умолчанию выдаётся содержимое текущего каталога, но каталог можно и задавать

  –R рекурсивный список

  -l выдать файлы в длинном формате

  -a выдача всех файлов, включая скрытые (т.е. начинающиеся с точки)

  cd переход в домашний каталог пользователя

  -t сортировка по времени создания

  -r соритровка в обратном порядке
- chmod изменение права доступа к файлу или директории
- find поиск файла по файловой системе

  -name файл - заставляет команду find искать указанный файл;

  -print - выводит имена найденных файлов.
- cpio копирование больших объемов файлов; создание архива и извлечение из него
- tar создание архива

  $ tar cvf ar.tar /tmp
- gzip

2.Команды для работы с текстовыми файлами

- cat выдать содержание файла на стандартный выход.
- split разбивает файлы на части
- less постраничная выдача текстового файла
- more постраничная выдача текстового файла (менее функционально)

- wc выводит число строк, слов и символов в файле

- c символы
- w слова
- l строки

$ ls –l | wc –l выдаёт сколько файлов в текущем каталоге

- grep ищет строку с заданной подстрокой во входном потоке
- vi Экранный текстовый редактор редактор

работает в 2-х режимах: режим набора текста и режим редоктирования

:wq выход с записью

:q выход без записи

- ed строчный текстовый редактор
- sed потоковый редактор
- diff сравнить текстовые файлы
- man выдача помощи по командам

3.Тестирующие команды и работы с переменными

- echo выдать строку на стандартный выход

$ echo “Hello”

-n не выдавать возврат коретки после строчки

-e разрешить служебные символы внутри строки

- date команда работы с датами, формат выдачи можно задавать
- cal календарь
- expr вычислить выражение

$ expr 2 + 3

- eval выполнить выражение (shell команду)

$ str=”cat f1”

$ eval $str

- set выдать переменные среды
- who выдаёт пользователей, работающих в системе
- ps показывает список выполняющихся процессов
- kill убить процесс

kill <№ процесса> убить навсегда

kill all <Название процесса> убить процесс по имени

- file выдаёт тип файла
- type Выдаёт тип

$ type ls

Ответ: /bin/ls

- test выполняет проверку условий

Вызов команд

<команда> [ключи] -- <аргументы >

Все аргументы разделяются пробелами. Шаблоны и переменные интерпретируются shell, команде передаются реальные значения.

$ ls|grep “.txt” выдают файлы txt в каталоге

$ \ переход на другую строку в конвейере

Для вызова собственных команд необходимо указать путь:

/home/stud/имякоманды

./имякоманды

Получение справки по команде.
команда --help
команда -?
man команда
man №раздела команда

Выполнение команды с помощью функций exec:
execl (путь, аргумент1, аргумент2, … , 0)
execlp (файл, аргумент1, аргумент2, … , 0)
execle (путь, аргументы, среда)
execv (путь, двумерный массив аргументов)
execvp (файл, двумерный массив аргументов)

Шаблоны файлов

Шаблон — это средство, которое позволяет задавать несколько имён файлов по определённому образцу. Шаблоны интерпретирует сам Shell.

set –f отключить работу шаблонов
set +f включить работу шаблонов

1. * обозначает любое число символов

rm * удалить все файлы

2. ? один символ

rm ?? удалить файлы в каталоге, которые содержат 2 символа
rm ??* удалить файлы в каталоге, которые содержат 2 и более символов

3. [] один символ из заключённого в них набора

rm [abcd] * удалить все файлы, которые начинаются на a,b,c,d

4. {}

cp t1.{exe,com} /tmp скопировать все файлы .com и .exe в /tmp
эквивалентно
cp t1.exe t1.com /tmp

5. # комментарий до конца строки

echo “Вася” # здесь был Вася =)

Чтобы все перечисленные символы не интерпретировались Shell’ом как шаблон, их необходимо заключать в кавычки - ‘*’ – или экранировать - *.

Переменные в Shell

Переменные в Shell имеют один тип - строковые. При обращении к shell-переменной необходимо перед именем ставить символ '$'.

$ a=hi
$ echo $a
Ответ: hi
$ echo ‘$a’ значение $a не интерпретируется Ответ: $a
Чтобы в переменную записать содержание файла: c=`cat f1`

Для того чтобы имя переменной не сливалось со строкой, следующей за именем переменной, используются фигурные скобки {}.

a=hi

a=${c}all

С помощью {} можно

- установить значение по умолчанию
- выделить подстроку в значении переменной
- произвести замены

man bash- см. инструкцию.

Переменные окружения

Каждый процесс имеет среду, в которой он выполняется. Shell использует ряд переменных этой среды. Если вы наберете команду 'set' без параметров, то на экран будет выдана информация о ряде стандартных переменных, созданных при входе в систему, а также переменных, созданных и экспортируемых вашими процессами.

Переменные окружения:

HOME=/home/kdb	имя домашнего каталога
PATH=/usr/local/bin:/usr/bin:	переменная, содержащая пути, в которых shell ищет заданные команды
LOGNAME	имя пользователя
UID	индификатор
PS1,PS2	вид промера(подсказки: $ у root; # у простых смертных)
LANG	язык (ru – русский, c – английский)
LS_COLORS	цвета при выводе в команде ls.

Создать переменную окружения можно командой export.

export a	создает переменную окружения a
export PATH= $PATH:/home/stud	добавляет в PATH домашний каталог
export PATH= $PATH:.	добавляет в PATH текущий каталог
export LS_COLORS=exe-red:sh-green	устанавливает цвета у файлов с указанным расширением

Командные файлы

В интерпретаторе shell существует два режима работы: выполнение команд из командной строки и последовательное выполнение всех команд из файла.

В начале этот файла должен быть указан транслятор. В нашем случае это - sh, находящийся в директории /bin

```
#!/bin/sh
```

Далее следует сама программа.

При выполнении sh с ключом –x файл будет выполняться с выдачей отладочной информации.

Пример программы:

```
#!/bin/sh
name=$1
echo “Привет, ${name}!”
```

Для того чтобы файл мог выполняться, ему надо сменить тип доступа.

```
$ chmod 755 prog
```

где 755 - тип доступа, prog - имя файла.
$ prog
$ ./prog

$ prog f1 5 k

$ echo $2
5
$ echo $* //$* выдать все аргументы
f1 5 k
$ echo $# //$* выдать число аргументов
3
shift — сдвигает аргумент, в этом случае, обращаясь к $1, можно организовать цикл.

Функции

имя_функции ()
{ тело функции }

########## Prog.sh #######
foo()
{
Echo “$1-hello”
}
foo Hi
##########################
$ a=`foo Vasya`
$ echo $a
Ответ Vasya Hello

Команда TEST

Команда 'test' проверяет выполнение некоторого условия. С использованием этой (встроенной) команды формируются операторы условия (if) и цикла (while, until) языка shell. Команда "test" дает значение "истина" (т.е. код завершения "0")

Два возможных формата команды:

test условие
или
[условие]

Условие может задаваться через

- числа
- строки
- имена файлов

Shell будет распознавать эту команду по открывающей скобке '[', соответствующей команде 'test'. Между скобками и содержащимся в них условием обязательно должны быть пробелы.

[$a] && echo $a если верна первая команда, то выполнить вторую
[‘$a’]

[-z ‘$a’] проверяет пустая ли строка
[“s” = ”Hi”] сравнение.

Условия сравнения чисел.

Переменные-числа в “” не записываются.

- x -eq y # x равно y,
- x -ne y # x неравно y,
- 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.

[$x = 5] && echo “Отлично”
[$x -lt 5] && echo “Меньше 7”

Условия проверки файлов

- -f имя файла Существует ли этот файл ?
- -d имя файла Является ли каталогом ?
- -c имя файла Специальный файл
- -r имя файла Файл доступен на чтение
- -w имя файла Файл доступен на запись
- -s имя файла Файл не пустой

[-c /tmp/f] Проверяет, существует ли файл с именем f

Сложные условия
усл.1 –a усл.2 = and «и»
усл.1 –o усл.2 = or «или»

Проверка условий:
&& позволяет выполнить 2-ю команду в случае успешного выполнения 1-й команды
[“$str” = “55”] && echo 55
k1 ||k2 команда k2 выполняется при условии неудачного выполнения k1
[-d /tmp/v] || mkdir /tmp/v

IF

Команда ветвления. В общем случае оператор 'if' имеет структуру

if команда

```
  then  команды
    [elif  команда
     then  команды]
  [else  команды]
fi
```

В зависимости от кода возврата выполняется та или иная ветвь.

```
if [ "$s"  =  "Привет" ]
 then
   echo "Hi"
 else
   echo "Не понимаю"
fi
```

```
if grep “слово” /tmp/f
        then     echo “нашли”
fi
```

CASE

Оператор выбора 'case' имеет структуру:

```
    case   <строка>  in
        <шаблон>)  <команды>;;
<шаблон>)  <команды>;;
    esac
```

Пример.

```
    echo "Сколько лет? "
    read z
    case $z in
      ?|1? )echo “Ребёнок”;;
      ?? ) echo “Молодёжь”;;
      3?|4?|5? ) echo “Взрослый”;;
    esac
```

FOR

Оператор цикла "for" имеет структуру:

```
  for <переменная> [in < значения >]
        do
команды
        done
```

Переменная приобретает поочередное значение из списка, а если список не задан , от аргументов программы.

Расчет сортировки трех файлов f1, f2, f3 будет выглядеть так:

```
for i in f1 f2 f3
    do
      sort $i > $i_sorted
    done


for i in *.txt
    do
      gzip $i
    done
```

WHILE

Структура оператора цикла с истинным условием 'while', обеспечивает выполнение расчетов, когда неизвестен заранее точный список значений параметров или этот список должен быть получен в результате вычислений в цикле.

```
    while команда
       do
          <команды>
       done


     n=0
    while [ $n -lt 10 ]    # пока n < 10
     do
       echo $i
       n=`expr $n + 1`
     done

cat f.txt|while read a b
do
  echo “ $a  съел  $b”
done
```

Список команд в теле цикла повторяется до тех пор, пока сохраняется ложность условия или цикл не будет прерван изнутри специальными командами ('break', 'continue' или 'exit').

UNTIL

Оператор цикла с ложным условием 'until' имеет структуру:

```
 until команда
       do
          список команд
       done
```

Пример 'Ожиданние 12:00'.

```
    until date | grep "12:00:"
```

```
do
   sleep 1
done
echo “Подъем”
```

Регулярные выражения

- это язык, описывающий шаблоны строк (дополнительную информацию искать в man 7 regex).

Существуют 2 типа регулярных выражений:
1) стандарта POSIX
2) принятый в языке Perl

- Одиночный символ, кроме / [\ { + * - $? ^
  $ grep “a” f.txt ищет строки, содержащие “a”
  Для использования метасимволов их экранируют.
  $ grep “ a\[i\] ” f.txt
- Мнимые символы
  ^ начало строки
  $ конец строки
  $ grep “ ^ups$ ” f.txt ищет строку, состоящую из слова “ups”
- Любой символ кроме конца и начала строки обозначается точкой (.).
  $ grep ‘ab.d’ f1.txt
- Классы символов []
  [abc] любой из символов 'a', 'b' и 'c'
  [a-zA-z] любая английская буква
- Альтернатива
  $ grep “ p|P” f.txt ищем строку, где встречается p или P
- Группировка ()
  ‘(hi)|(hello)’
- Квантификатор- показывает сколько раз символ должен повторяться
  + один и более раз
  * ноль и более раз
  ? 1 или 0 раз
  {n} n раз
  {n,} n раз и более
  {n,m} от n до m раз

Пример.
‘[a-zA-z]+@([a-z]+\ .)+[a-z]+’ шаблон на e-mail

Регулярные выражения используются в конфигурационных файлах, базах данных, командах grep, sed, awk, lex и т.д.

Потоковый редактор SED

Команда копирует файлы (по умолчанию со стандартного входа) на стандартный выход, редактирует их в соответствии со своими(!) командами, размещенными в "сценарии" (в командном файле или строке редактора [a не shell!]).

$ sed [-e] ‘сценарий’ [файл]

-f берет команды из файла сценария

-e script «скрипт»

-r будут использоваться регулярные выражения

Сценарий (скрипт) состоит из команд редактирования, по одной в строке, имеющих формат:

[адрес1 [, адрес2]] команда [аргументы]

"sed" циклически преобразует входные строки в выходные.

Адреса "[адрес1 [, адрес2]]" - это либо номера строк, либо последняя строка (символ "$"), либо регулярные выражения в стиле редактора "ed":

- "\" используется в многострочных командах для экранирования продолжения строки.
- "." совпадает с любым символом.
- Если адреса не указаны - просматриваются все входные строки.
- Если один адрес, то выбираются совпадающие строки.
- Если заданы два адреса, выбираются строки в заданном интервале.
- "!команда" выполняется команда "команда", для строк, которые не были выбраны по адресам.

1.a\

text добавляет text после указанной строки

Пример.

```
$ who | sed  '2a\
новая строка
'
```

Результат:

```
root     tty1         Mar 13 17:23
mas      tty2         Mar 13 18:50
новая строка
kdb      tty6         Mar 13 17:24
kdb      tty5         Mar 13 17:24
```

2.b\ label

Осуществляет переход к команде ("команда") "label:команда" Если метка ("label") отсутствует, то переход на конец командного файла.

3.c\

text Удаляет выбранные строки и заменяет их на "text'.

4.d\ Удаляет найденные строки

5. i\

text Вставляет "text" перед выбранной строкой.

(сравните с "a\")

Пример:

```
$ who | sed  '2i\
```

новая строка
'

Результат:
root tty1 Mar 13 17:23
новая строка
mas tty2 Mar 13 18:50
kdb tty6 Mar 13 17:24
kdb tty5 Mar 13 17:24

6.p\ Выводит найденные строки (используется с флагом "-n").
7.q\ Выходит из "sed".

Язык обработки шаблонов AWK

Используется для пакетной обработки текста на более детальном уровне, его средства позволяют выполнять работу с конкретными полями.AWK применим к текстовым файлам, которые имеют структуру таблицы

1. СТРУКТУРА awk-ПРОГРАММЫ
Программа состоит из операторов (правил), имеющих вид:
шаблон {действие}
шаблон {действие}
. . .
Частные случаи:
{действие} - когда действие выполняется для всех строк.
шаблон - когда выводятся строки с данным шаблоном.
Действие может состоять из последовательности операторов, разделяемой ";" или переводом строки или закрывающей скобкой.
Формат шаблона:
$ №столбца ~ /регулярное выражение/
Возможны комментарии (как в shell "#.....").
Пример:
Напечатать строки, которые начинаются на a.
$ AWK ‘/^a/ {print;}’ f-awk

Существует два оператора специального вида ("BEGIN"- начальные установки и "END" - "последействия"):
BEGIN {действие}
шаблон {действие}
шаблон {действие}
. . .
END {действие}

2. ВЫЗОВ awk
awk –f файл_с_программой
awk ‘программа’
Пример1.
Определить размер файлов в текущем каталоге, имя которого начинается на “e”.
$ ls –l | AWK ’BEGIN {a=0;} | $8` |^e.*|{a+=$5} END {print a;}’
Пример2.

```
$ ls –l | AWK ’BEGIN {a=0;}
{ if (NF>2)        # число полей в текущей строчке
{a++; M [a]=$8;} # записывает все файлы в массив
}
END { for (i=1; i<=a; i++ ) print i, M[i] }’
```

3. awk-ПЕРЕМЕННЫЕ И ВЫРАЖЕНИЯ

В языке awk выделяются две группы переменных: предопределенные и декларированные в программе. Исходные значения предопределенных переменных устанавливаются интерпретатором awk в процессе запуска и выполнения awk-программы.

К предопределенным относятся:		Умолчания:
NR	номер текущей строки;	
NF	число полей в текущей строке;	
RS	разделитель строк на вводе;	"\0"
FS	разделитель полей на вводе;	пробел и/или табуляция
ORS	разделитель строк на выводе;	RS
OFS	разделитель полей на выводе;	FS
OFMT	формат вывода чисел;	"%.6g"
FILENAME	имя входного файла.	

4. SPLIT

Разбивает строку заданным символом, добавляет поле.

Пример:

Найти все Дни Рождения

```
-----------------------------------
Валя  25/09/2006 |   20060925
-------------------------------
```

Добавить поле для сортировки по ДР

```
Awk ‘{D=split(“/” , $2);
Printf(“%s  %s  %d%d%d\n”,$1,$2,D[2], D[1], D[0]);}’
```

Компиляторы

CC Си Компилятор

```
$ cc t.c          файл с программой
$ ./a.out     результат вышенаписанного – созданный исполняемый файл

$ cc -o t t.c     t-имя исполняемого файла
$ ./t
```

-l<библиотека>	подключить библиотеку
-L<пусто>	путь, где хранится библиотека
-I<пусто>	указываются каталоги для поиска файлов, заданных #include <Ф.h>
-D name [=def]	задается макрос
-g	добавить отладочную информацию
	gdb отладчик
	strip чистит отладочную информацию
-c	сделать только объектный файл
-O	задать уровень оптимизации
-shared	подключает разделяемые библиотеки
-static	подключает статические библиотеки

-fPIC	генерируется позиционно-независимый код (для создания разделяемых библиотек)

Отладка программ

Необходима для разрешения ошибок алгоритмов, в частности, зацикливания, неправильного формирования индекса массива и значения указателя (утечка памяти), неверных значений переменных.

Отладка без отладчика

Если отладчик недоступен, используется выдача отладочной информации (значения переменных, этапы выполнения) в файл или стандартный выход. Для того, чтобы отладочную информацию впоследствии не пришлось очищать, используются инструкции препроцессора.

```
f(int x) {return x+2;}
main () {
int i, r;
#ifdef DEBUG                (1)
printf("Begin \n");
#endif
for (i=0; i<10; i++) {
    #if DEBUG > 1           (2)
    printf("i=%d\n", i);
    #endif
    r=f(i);
    #if DEBUG > 2           (3)
    printf("f(i)=%d\n", r);
    #endif
}
#ifdef DEBUG                (1)
printf("End \n");
#endif
}
```

При отладке определяем макрос:
#define DEBUG //будет исполняться (1)
Либо при компиляции:
cc-o ex ex.c –DEBUG=2

Отладчик

Функции отладчика:

- пошаговое выполнение программы
- выдача значений переменных
- запуск программы
- установка точек прерывания
- продолжение выполнения

Отладчик gdb

cc –g …
gdb <имя исполняемого кода>
Далее функционирует отладчик в режиме командной строки.

help - помощь
run – запуск программы
break – установка точек прерывания
break <№строки исх. кода>
break <имя функции>
break main отладка с самого начала
continue – продолжить выполнение до следующей точки прерывания или до конца
step – пошаговое выполнение с заходом в функции
next – пошаговое выполнение без захода в функции
print <имя переменной> - выдача значений переменнох

Отладка готового кода

strace – трассирует системные вызовы программ
strace <имя исполняемого кода>
strace –p <№ процесса>

Системный вызов ptrace (один из способов взаимодействия процессов) позволяет родительскому процессу управлять выполнением порожденных, а также исследовать и менять данные в образе процессов и значения регистров

ptrace(<действие>, <№процесса>, <адрес>, <данные>)

Действия:

PTRACE_ATTACH	подцепиться к процессу
PTRACE_DETACH	отцепиться
PTRACE_TRACEME	
PTRACE_KILL	уничтожение
PTRACE_PEEKTEXT	вернуть данные, записанные в адресе в области текста
PTRACE_PEEKDATA	вернуть данные, записанные в адресе в области данных
PTRACE_CONT	продолжить выполнение
PTRACE_SINGLSTEP	
PTRACE_SYSCALL	

Выполнение отладчика и отлаживаемой программы синхронизируется с помощью сигналов.

Профилирование программ

- это сбор статистической информации о времени выполнения отдельных частей программы

time <>
time cat f1.txt
gprof
cc –pg

Пример.

```
############
cycl5() {
int i
for (i=0; i<100000000; i++)
    cycl0();
}
cyci1() {
int i
```

```
for (i=0; i<100000000; i++)
}
cycl0() {
int i
for (i=0; i<5; i++)
}
Main() {
cycl1();
cycl5();
}
###########
```

После вызова gprof

% времени	Накопл. секунд	Своих секунд	Число вызовов	Общее время (мс/вызов)	Собств. время вызова	Имя
76,58	16,38	16,38	10^8	0	0	cycl0
15,61	19,72	3,34	1	19 720	3 340	cycl5
7,81	21,39	1,67	1	1 670	1 670	cycl1
0	21,39	0	1	21 390	0	main

Здесь:
Накопл. секунд – время, прошедшее с момента начала выполнения программы.
Своих секунд – время собственного выполнения
Общее время – время, прошедшее с вызовом функций
Собственное время вызова – время, прошедшее без вызова функции

Построение библиотек с использованием CC

```
##### Kv.c #######
Kv(int x)
{
Return(x*x);
}
################

##### T.c #######
Main()
{
Printf("%d",Kv(9));
}
################
```

Сначала создаётся статическая библиотека:

```
$ cc –o kv.o –c kv.c        компиляция
$ ar r libkv.a kv.o         создаём статическую библиотеку
$ ranlib libkv.a            подготавливаем библиотеку к использованию
$ cc –o t t.c –L. –lkv
```

Создание разделяемых библиотек:

```
$ cc –c –fPIC kv.c          PIC позиционно независимый код.
$ cc –shared –o libkv.so —fPIC kv.o
```

$ cc –o t t.c –L. –lkv
При запуске библиотека будет искаться в :
/lib
/usr/lib
/etc/ld.so.conf включается программа ldconfig
LD_LIBRARY_PATH переменная среды
ldd t выводит динамические библиотеки, которые используются в t

Команда Make и её Makefile

- задаёт правила компиляции.

Утилита 'make' предназначена для автоматического определения частей большой программы, которые необходимо перекомпилировать.

Обычно инструкция для работы make содержится в файле с именем 'Makefile'.

```
#!/bin/make
OBJ=    “t.o kv.o”          создаём переменную OBJ, указывая все объектные файлы
kv.o : kv.c
    cc –o kv.o –c kv.c
t.o : t.c
     cc –o t.o –c t.c
#Можно задать следующим образом
# %oO : %C                  %O цель     %C зависимость
#cc –o $@ -c $<
Делаем исполняемую программу
 t : ${OBJ}
 cc –o t ${OBJ}
Удаление временных файлов
Clear:
     rm *.o *.b
Вызов:
$ make t
```

Взаимодействие процессов и сокеты

Listen прослушивание порта
Accept Приём соединения. Создаёт для достижения соединения новый сокет.

```
##### Client #########
#include <sys/type.h>
#include <sys/socket.h>
#include <netinet/in.h>

main()
{
 int sd;
 struct sockaddr_in tss_addr;
 sd=socket(AF_INET,SOCK_STREAM,0);
```

```
 tss_addr.sin_family=AF_INET;
 tss_addr.sin_port=htons(45000); //номер порта
 tss_addr.sin_addr.s_addr=inet_addr("127.0.0.1");
 connect(sd,&tss_addr,sizeof(tss_addr));
 write(sd, ”hello” ,6 ) ;
}
###################

##### Server #########
main(){
int sd, ns;
int buf_ben;
char buff[256] ;
   struct sockaddr_in s;
   struct sockaddr_in c;
   int c_len;
   sd = socket(AF_INET, SOCK_STREAM, 0); // TCP-протокол
     soc.sin_family = AF_INET;
     soc.sin_port = htons( 45000 );
     soc.sin_addr.s_addr = inet_addr("127.0.0.1");
     bind(sd,&s,sizeof(s));
     listen(sd,1);
     for(;;)
     ns = accept(sd, &soc, &c_len)
        if ( fork()==0)
        {
             close(sd);
        read(ns, buff,256 );
        printf("%s",buff);
        exit(1);
        }

  close(ns);
    }
}

###################
```

Обслуживание множества запросов

- Обслуживание систем, которые работают со многими каналами
- Работа с устройствами без прерываний

Системный вызов позволяет обслужить несколько запросов select.

SELECT (**число на 1 больше Max**/*номера дескриптора во множествах*/, **множество дескрипторов для чтения** , **множество дескрипторов для записи** , **множество дескрипторов для ошибок исполнения** , тайм аут)

Тайм аут – сколько ждать события, если NULL- ждать до «победы».

Возвращает номер множества ,в котором сработал дескриптор. Во множестве остаются дескрипторы которые сработали.

Ожидаются строки из именованного канала

```
################
Int dset ;
Mknode tpip p;     #создание именованного канала
Int fd;
fd=open("tpip", O_RDONLY| O_NONBLOCK);
for (;;)
{
FD_ZERO(&dset);
FD_SET(fd,&dset);
FD_SET(sd,&dset);
if (select (5,&dset,NULL,NULL,NULL)!=1) continue ;
if (FD_SET(sd,&dset)) ns=accept(…);
else if  (FD_ISSET(fd,&dset)) {ns=dup(fd);
close(fd);
fd=open("tpip", O_RDONLY| O_NONBLOCK);

}
.
.
.
}
###################
```

Нити = threads, лёгкие процессы.

Что создаётся с помощью fork() – это тяжёлые процессы.

Минусы:

- Затрудненное совместное использование данных;
- Проблемы синхронизации ;
- Порождение процесса ресурсоёмкое ;
- Накладные расходы на переключение процессов.

Для решения проблем fork() используется нити. Они выполняются в том же адресном пространстве что и породивший процесс, имеют отдельные: программный сигнал, стек, маску обработки сигналов, errno – номер ошибки.

Нити выполняются в режиме корпоративной многозадачности, что решает проблему синхронизации.

Schedule – планировщик.

Pth – библиотека для нитей. Использует clone, она эмулирует API POSIX.1c.

Состояния нити:

Основные функции работы с нитями:
Pth_init() инициализирует библиотеку;
Pth_kill() убивает библиотеку;
Pth_ctrl(запрос, аргументы) информация о нитях : число нитей, приоритет, имя.
Pth_attr_init(атрибуты) инициализирует атрибуты
Pth_attr_set(атрибуты, поле, значение)

Нить = Pth_spaw (атрибуты, функции, аргументы) создание нити;

Pth_suspend() зарезервировать нить
Pth_resume() задействовать ранее зарезервированную нить
Pth_yield() передает управление нити или планировщику
Pth_exit(значение) завершение нити.

```
######## Threads #########
Static void *nit (void *n)
{
    int a=(int) n;
    for (int i=0;;i++) {printf("%d",a);
                        if(i%5==0)pth_yield(NULL)
                            }
}
    Main()
{
Pth_attr_t attr;
Pth_init();
Attr=pth_attr_new();
Pth_attr_set(attr, PTH,ATTR_NAME, "nit");
Pth_spawn(attr,nit,34);
Pth_spawn(attr,nit,75);
Pth_sleep(10);
}
########################
```

Элементы теории компиляции

Формальные грамматики

$G = <V_N, V_T, P , S >$
V_N -множество нетерминальных символов;
V_T - множество терминальных символов ;
P - множество правил вывода;
$S \in V_N$ - начальный нетерминальный символ.

Пример грамматик :
Метаязык
::= есть по определению
| или (исключающее)
[] необязательные символы
, перечисление
<нетерминальный символ>

<SELECT>::= select <IDS> from <IDS>
<IDS>::= <WRD> | <WRD> , <IDS> ; рекурсивное определение индефикаторов
<WRD>::=<LET>|<LET><WRD>
<LET>::= a|b|c…|z

Разберем строку:
for (i=0; i<10; i++)
< FOR >::=for(< инициализация >;< условие >;< изменение >)
< инициализация >::=< переменная >=< число >
< условие >::=< переменная >< знак сравнения >< число >
< изменение >::=< переменная >++ | < переменная >--
< переменная >::=< буква > | < буква >< переменная >
< буква >::=a|b|…|z
< число >::=< цифра > | < цифра >< число >
< цифра >::=1|2|…|9|0
< знак сравнения >::=< | >

Язык – множество $\{X \in V^*_T\}$, цепочек терминальных символов, таких, что они получаются из начальных нетерминальных символов.
$L(G) =\{ X \in V^*_T \mid S=>^*x\}$

Классификация языков по Хомскому

Язык – множество $\{X \in V^*_T\}$, цепочек терминальных символов, таких, что они получаются из начальных нетерминальных символов.
$L(G) =\{ X \in V^*_T \mid S=>^*x\}$
Классификация основывается по типу правил. Если в одном языке правила можно отнести к разным типам, то выбирается худший тип.

Типы:

0-тип: Не накладывает ограничения на правила, поэтому и не рассматривается.

1-тип: Контекстно-зависимые грамматики.

Правила имеют следующую форму:

vαw::=vβw

v,w∈ V* – цепочки любых символов(контекст)

α∈ V_N

β∈ V*

Пример:

φ->A	|		
[A]-> [C]	|		
(A)->(B)	|	=>	S=>(A)=>(B) =>(x)
B->x	|		
C->y	|		

2-тип: Контекстно-свободные грамматики.

Правила имеют следующую форму:

α::=β

α∈ V_N

β∈ V*

A->BcD

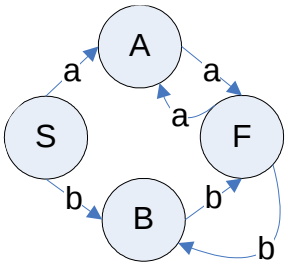
D->cA

3-тип: Автоматические грамматики (рекурсии).

- Левосторонние α::=wβ α::=w α, β ∈ V_N w∈ V_T
- Правосторонние α::=βw α::=w

aabbbbaaaabb┘

1. S->bB
2. S->aA
3. A->aF
4. B->bF
5. F->aA
6. F->bB
7. F->┘



Распознающие автоматы – автоматы Мура с множеством выделенных состояний - конечных. Этот автомат недетерминированный и частичный.

	S	A	B	F
q	A	F		A
b	B		F	B

====> переход к полному автомату

	S	A	B	F	Err
a	A	F	(E)	A	(E)
b	B	(E)	F	B	(E)

Переход от праволинейной грамматики к автоматам

S->select ==> S->s<elect>

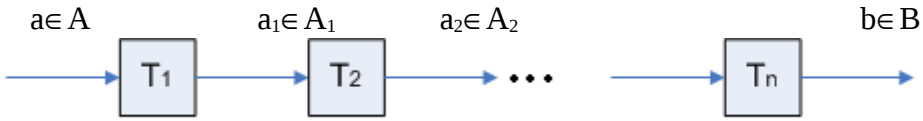
<elect> -> e<lect>
<lect> -> l<ect>
<ect> -> e<ct>
<ct> -> c<t>
<t> -> t

Трансляторы.

Трансляторы - программа или устройство, переводящее входную строку из одного языка в другой без потери списка.

$a \in A$ $b \in B$

Для упрощения процесс транслирования разбивают на шаги



Виды трансляторов:

- Интерпретаторы – перевод из одного языка в другой по шагам.
- Компиляторы - переводит целиком.

Претрансляция

-текстовая замена макроопределений.

Структура компилятора

Функции компиляторов.

1. Лексический анализ – выделяет лексемы в строчке и проверяет на правильность.
2. Синтаксический анализ – проверяет порядок лексем.
3. Семантический анализ- проверка на правильность присваивания.
4. Генерация выходного текста

Схема работы компилятора

Лексический анализ.

Функции лексического анализа:

1. Выделения численных констант
2. Выделение индификаторов
3. Выделение сложных символов: /* */ //
4. Определение ошибок ввода

Для лексического анализа используются автоматные (регулярные) грамматики. В средствах лексического анализа используются регулярные выражения.

< нетерм. символ > → [< нетерм. символ >]< терм. символ >

A → Bc

B → Cc

C → d

< нетерм. символ > → < терм. символ >[< нетерм. символ >]

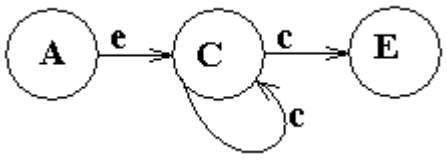
A → eC

C → cC

C → c

Зададим последнюю с помощью автомата

	A	C	E
a	-	C,E	-
e	C	-	-



Приведем к детерминированному

	A	C	E	{-}	{C,E}
c	{-}	{C,E}	{-}	{-}	{C,E}
e	C	{-}	{-}	{-}	{-}

Lex - команда Unix. Предназначена для ганерации программы на Си, которая будет проводить лексический анализ входного текста в

соответствии с правилами.

Формат файла на lex:

раздел деклараций : имя_значение.
%%
раздел правил : шаблон_действие (регулярное выражение).
%%
Код на Си

Раздел деклараций: %token лексемы
Раздел правил: нетерм.симв: | цепочка символов { код на Си }
;
Для обработки ошибок описываем функцию
yyerror()
{ printf (“Ошибка”); }

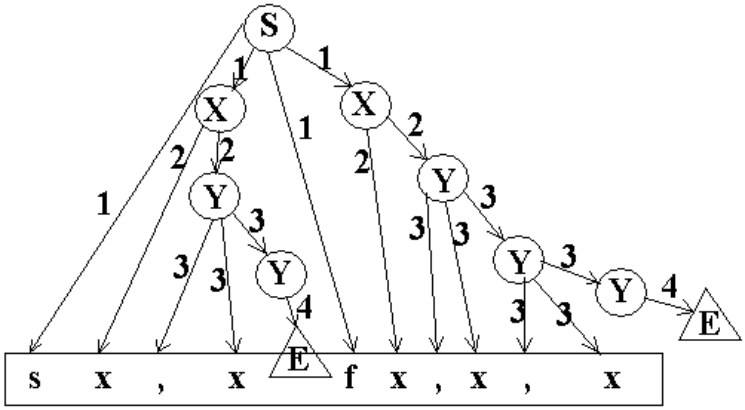
Подсчёт идентификаторов во входном потоке
################ P.l ################
```
digit [0-9]
letter [a-z]
%%
{letter}({letter}|{digit})*  { i++; }
%%
int i;
main()
{
  yylex();
  printf( “ %d", i );
}
yyerror (){}
```
##
yylex() - из раздела правил преобразует в Си
В командной строке пишем :
$ flex -oprogname.c progname.l
$ cc -o progname progname.c -lfl
$ ./progname [< filename]

Синтаксический анализ

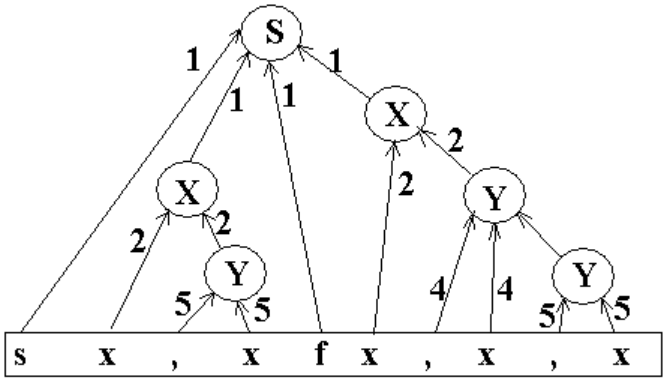
Выделяют 2 типа синтаксического анализа: сверху вниз и снизу вверх. Используются контекстно-свободные грамматики.

< нетерм. символ > → < цепочка символов >
A→bC
C→eA
A→dCCA
A→b
c→E
Слева должен быть нетерм. символ – без контекста
Пример:
select name1, name2 from tab1, tab2
S x,x,…x f x,x,…,x

Анализ сверху вниз
1) S→sXfX
2) X→xY
3) Y→,xY
4) Y→E
s x , x f x , x , x



Анализ снизу вверх
1) S→sXfX
2) X→xY
3) X→x
4) Y→,xY
5) Y→,x



S и q грамматики

-Контекстно-свободные грамматики , на которые наложены ряд ограничений :
правые части правил начинаются с терминальных символов, причем для одного и того же левого символа правые части начинаются с разных символов.

S -> aT
S -> TbS
T -> bT
=
T -> bS
=

q-грамматика отличается от s-грамматики наличием аннулирующего правила (в правой части есть пустой символ)
=> ε

Из-за аннулирующих правил для q-грамматики вводится понятие следующего символа. N(A) - множество терминальных следующих (Next) за A символов.

В данном случае за A могут следовать a или b - {a,b}.

LL(1)- грамматика

Правая часть может начинаться с нетерминального символа, но при этом должна сохраняться детерминальность в выборе правил, т.е. множества выборов из правил при одном и том же нетерминальном символе не должны пересекаться.

LL(1)=Left-left most- самый самый левый.

F(α) - множество терминальных символов, стоящих первыми (First) в цепочках, выводимых из строки α.

N(A) - множество терминальных символов, следующих (Next) в цепочках за данным нетерминальным символом A.

Множество выбора для каждого правила формируется с учетом множества первых и множества следующих символов.

1. S -> AbB	E(1) = F(AbB) = {a, b, c, e}
2. S -> d	E(2) = {d}
3. A -> CAb	E(3) = F(CAb) = {a, e}
4. A -> B	E(4) = F(B) N(A) = {c}U{b} = {c}U{b}
5. B -> cSd	E(5) = F(cSd) = {c}
6. B -> ε	E(6) = F(ε) U N(B) = { ε }U {b, d, ┤ }
7. C -> a	E(7) = {a}
8. C -> ed	E(8) = {e}

МП-автоматы (автоматы с магазинной памятью) – в стек помещается начальный нетерминальный символ

	x	Δ
(	↓x	↓x
)	↑x	-
┤	-	+

Преобразуем выражение (()())

(	(	)	(	)	)	┤
Δ	x	x	x	x	x	x
	Δ	x	Δ	x	Δ	
		Δ		Δ		

LL(1) грамматики распознаются с помощью метода рекурсивного спуска.

Пусть дана грамматика:

1.	I -> LP	{ab)
2.	P -> LP	{ab}
3.	P -> DP	{1,2}
4.	P -> E	N(P)د
5.	L -> a|b	
6.	D -> 1|2	

Метод рекурсивного спуска позволяет писать программы синтаксического анализа на языке, допускающем рекурсию, прямо по грамматическим правилам.

Метод рекурсивного спуска:

```
###########
int c
main()
{I();printf("+");}
I()
{c=getchar();
    Switch(c)       {
            case 'a':
            case 'b': L();P(); break;
            default printf ("-"); exit(0);
                    }
}
L() {}
D() {}
P()
{ c = getchar();
    Switch(c)       {
            case 'a':
            case 'b': L();P(); break;
            case '1':
            case '2': D();P(); break;
            case "\n" : break;
            default printf ("-"); exit(1);
                    }
}
####################################
```

LR-грамматика

Left-Right most- самый правые части для самых левых нетерминальных символов.

Грамматика с предшествованием .

Правила расстановки отношений между символами грамматики :

1. Если S_i и S_j входят в одну свёртку, то они равны $S_i = S_j$
   …S_i S_j …
   Свёртка –правая часть правила.
2. Если S_j в правой части свёртки, то $S_i < S_j$
   S_i S_j…
3. …S_i S_j => $S_i > S_j$
4. …S_i S_j … => $S_i > S_j$

Все отношения не являются симметричными.

1.	S -> sXfX	|	s ≐ X X ≐ f f ≐ X
2.	X -> x	|	s < x x > f f < x
3.	X -> xY	|	x ≐ Y Y > f
4.	Y -> ,x	|	x < , , ≐ x
5.	Y -> ,xY	|	

В грамматике с предшествованием между двумя одинаковыми символами не стоят разные отношения.

├ s x , x f x ┤ расставим знаки в цепочке символов.

=> ├ < s < x < , ≐ x > f < x > ┤ =>

Y X

=> ├ < s < x < (, ≐ x) > f (< x >) ┤ =>

X

=> ├ < s < (x ≐ Y) > f ≐ X > ┤ => ├ < s ≐ X ≐ f ≐ X > ┤ = > ├ < S > ┤

YACC – BISON

Yet **A**nother **C**ompilator of **C**ompilators

Предназначен для генерации программы на Си, которая бы проводила синтаксический разбор входной информации. Грамматика может быть неоднозначна, поэтому для преодоления этого нужно использовать правила предшествования.

Имеет следующую структуру:

```
%{
Раздел деклараций
}%
%token лексем
%%
Раздел правил     Интерпретирует правила типа:
Нетерм. символ : цепочка символов {код на Си} {или}|цепочка символов{код на Си};
%%
 Пользовательский код
```

Пример. Рассмотрим предыдущий пример.

s x, x f x

Программа будет состоять из двух частей: синтаксический разбор (на yacc) и лексический на lex).

В лексическом разборе мы будем определять буквы x, y, z и запятую. И передавать в yacc программу найденное.

1. На YYAC

```
##########prim1.y#################
%token S X F Z
%%
es: S iks F iks       {printf(“OK!”);}                  //первое правило
    ;
iks:X
    | X igrek
    ;
igrek:       Z X
    | Z X igrek
    ;
%%
yyerror() { printf(" Ошибка! "); }
```

```
main()  {
yyparse(); }                                  //функция, которая получается из раздела правил
#######################################
```

2. На Lex

```
########prim1.l#################
%{
#include "prim1_y.h"       //там лексемы будут определены как макросы
%}
%%
s               { return( S ); }
x               { return( X ); }
f               { return( F ); }
[,]             { return( Z ); }
.               { return( yytext[0] ); }
%%
#############################
```

Синтаксический разбор будет выглядеть так:

```
$ yacc -d  -o     prim1_y.c    prim1.y
  # по -d создается prim1_y.h, в котором описываются макросы X Y Z P
$ lex –o      prim1_l.c     prim1.l
$ cc -o      prim1     prim1_y.c     prim1_l.c
```

Семантический анализ

Используется для проверки типов данных при операциях и области действия переменных.

Атрибутная грамматика – это четверка $G = <V_N, V_T, P, S>$, в которой
V_N - ***нетерминальный словарь*** (множество нетерминальных символов);
V_T - ***терминальный словарь*** (множество терминальных символов) ;
P - ***множество грамматических правил***;
$S \in V_N$ - ***начальный нетерминальный символ***.

A(x) множество атрибутных символов
Атрибут определяется для каждого правила, входящего в грамматику .
$a_0<i_0>= fpa_0<l_0> fpa_1<l_1>\ldots a_j<i_j>$
$a_k<i_k>$ атрибут x_i
Атрибуты делятся на 2 вида :
1.Синтезируемые атрибуты
a<0>= fpa<0>(…)
2.Наследуемые атрибуты
$x_0 \rightarrow x_1 \ldots x_i \ldots x_{np}$
a<i>= fpa<i>(…)
Таким образом, атрибутная грамматика :
$АГ = <Г, A_s, A_i, R>$
Г- контекстная свободная грамматика
A_s – множество синтезируемых атрибутов
A_i – множество наследуемых атрибутов
R- семантические правила

Пример : Число из 2сс перевести в 10сс(систему исчисления) 1011

Грамматика

$P \rightarrow S$

$S \rightarrow B$

$S^1 \rightarrow BS^2$

$B \rightarrow 1|0$

$V(D)=V(S)$

$V(S)=V(B)*2^0$; $p(S)=1$

$V(S^1)=V(B)*2^{p(S\hat{}2)}+V(S^2)$; $p(S^1)=P(S^2)+1$

Значения атрибутов терминальных символов – это константы, т.е. они определены ,но нет семантичных правил.

Дерево разбора обходится сверху вниз слева направо.

Система семантического анализа атрибутных грамматик с использованием грамматик.

ELEGANT, FGC-2,OX,RIE или можно использовать Lex иYacc.

B2d.l

```
%{
#include <stdlib.h>
# include <b2d_y.h>
%}
bit [0-9]
%%
{bit}        {yylval.val=atoi(yytext);  /*формирование числового значения, yylval-передаваемый в  yacc значение */
      Return(BIT);}
.    {return(yytext[0]);} /* . любой символ*/
```

##############

B2d.y

```
%{
```

```
#include <math.h>
long double p=0;
%} //  для передачи используется union
%union {
int val;
}
%token <val> BIT // лексемы, которые будут обмениваться Lex и Yacc
% type <val> str
%%
dec:          . str {printf("%d",$1);}
     ;
str: BIT {$$=$<val>1;}
     |BIT str {$$=$ <val>1*pow(2L,p+$1); p++;}
     ;
%%
Yyerror() {}
Main()      {yyparse()}
```

```
###############

$ yacc –d –o bed_y.c b2d.y
$ lex –o b2d_l.c b2d..l
$ cc –o b2d  b2d_l.c  b2d_y.c –lfl –lm
-----
$ b2d <enter>
 1011 <enter>
```

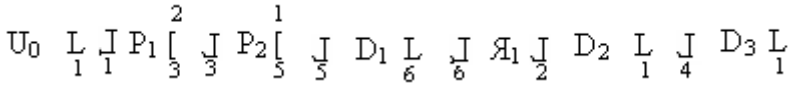
Оптимизация

1.Предварительные вычисления выражений.
x:=2 ; y:=3 ; z:=x+y+10; => z=15;
2. Исключение невыполнимых ветвей.
Switch
……..
Case : ……. Break; exit(0);
3. Выделение общих частей.
a:= (x+y)*z-35 |оптимизируем ==> w:= (x+y)*z; a:=w-35;
b:= (x+y)*z/a | b:=w/a;
4. Вынесение за цикл.
Выносятся значения, которые в цикле не вызываются.
… for (i=0; i<strlen(S); i++) printf("%c", S[i])…
… l=strlen(S); for (i=0; i<l; i++) …
5. Вычисление логических выражений.
if (i<strlen(str)&&str[i] !='x')
cand если не выполняется 1-ое условие, то не выполнять дальнейшие условия.
Лучше делать вложенные циклы.
6. Изменение последовательности команд с целью оптимизации межрегистровых передач и обращения к памяти.
/без примера/

Пример1. Язык логических схем.

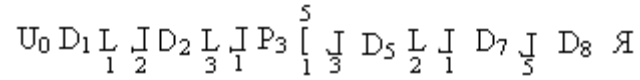
P – условие;
U_0 – начало

нет
]
да

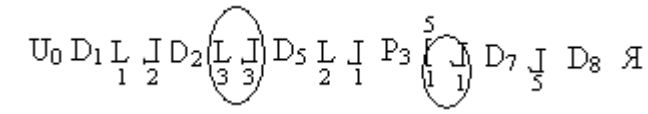


[P_1 : x > y; P_2 : y > x; D_1 : z:=x; D_2 : x:=x-y; D_3 : y:=y – x]

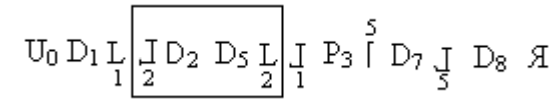
Пример2.



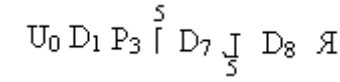
==>



==>



==>



Генерация выходного кода

При рассмотрении вопросов генерации выходного текста надо иметь в виду то, что реально выходной текст программы после трансляции - это, как правило, не выполняемый код, а некоторая промежуточная форма, поскольку программа в дальнейшем может быть загружена для выполнения в разные места памяти и т.д. С другой стороны, выполняемая программа (или программа в близком к такой форме виде) машинно-зависима, то есть использует конкретную систему команд и другие конкретные архитектурные особенности. Основная идея генерации выходного кода заключается в подстановке входных конструкций в готовые шаблоны.

Язык:
create id 0
create iid 0
create sch 3
create ssh 2
inc sch

```
loop sch
    inc id
    loop ssh
            inc iid
    pool
pool
print id
print iid

#### forc.l    #####
%{ …
# include “forc_y.h”                //файл с лексемами
%}
letter [a-z]
digit [0-9]
%%
inc                 {return (INC) ;}
create              {return (CREATE) ;}
print               {return (PRINT) ;}
loop                {return (LOOP) ;}
pool                {return (POOL) ;}
{letter}+   {strcpy(yylval.var, yytext); return(VAR);}
{digit}+    {sscanf(yytext,”%d”, &yylval.val); return(VAL);}
…
#############

### forc.y #####
%union
{int val; char var[16];}
%token <var> VAR
%token <val>  VAL
%token  CREATE INC LOOP POOL PRINT
%%
prog:       str             {printf ( “main() {%s}”,$1);}
    ;
str:        oper            {sprintf($$,”%s”,$1);}
    | oper str      {sprintf($$,”%s \n %s”,$1,$2);}
    ;
oper:       CREATE VAR VAL          {sprintf($$,”int %s=%d”,$<var>2, $<val>3);}
    |INC VAR                    {sprintf($$,”%s++”,$<var>2);}
    |LOOP VAR str POOL          {sprintf($$,”for(;%s>=0;%s--){%s}”,<var>2,<var>2, $3);}
    |PRINT VAR                  {sprintf($$,”printf(\”%s\”,%s)”, “%d”,$<var>2) ;}
%%
…
#############
```

На более низком уровне для подстановки используются таблицы решений генерации выходного кода.

ADD [регистр или память], регистр

A1+A2

A1	регистр	регистр	память	память

A2 регистр память регистр память
код ADD A1,A2 ADD A2,A1 ADD A1,A2 move A1,R
или ADD A2,R
ADD A2,A1

Загрузка программ

Загрузочный модуль, который получился в результате компиляции, может быть самодостаточным (в виде одного файла или программа перед загрузкой собирается из нескольких объектов).

Программа лежит на диске. Образ процесса загружен в оперативную память.

Виды загрузки :
1. Абсолютная загрузка
- Для однозадачной системы
- Для виртуальной адресации

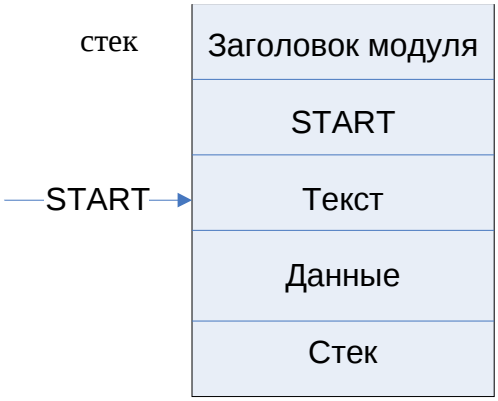
1.1. с разделами памяти , с физическими непересекающимися адресами.
2. Относительная загрузка
   Адреса в программе при загрузке пересчитываются

2.1. с базовой адресацией
   Программист задаёт адреса относительно какого-то числа
3. Загрузка с использованием позиционно-независимого кода
   Адресация относительно текущей команды
   + не надо перечитывать адреса
   - не все процессы поддерживают такую адресацию
4. Оверлей = перекрытие

<u>Абсолютная адресация</u>
Самый простой способ загрузки при котором программа загружается с одного и тоже адреса.
Допустима загрузка с случаях :
1. однозадачная система (RT11)
2. в системах трансляции виртуального адреса физическим.

Начальное содержимое образа процесса формируется простым копированием загрузочного модуля в память.

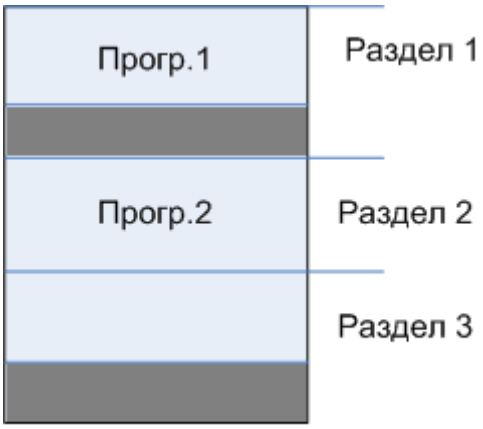
a.out



Формируются поля START => копируется текст, данные , => передаётся управление с позиции START

Абсолютная адресация с разделами памяти

Существует несколько допустимых стартовых адресов



Если заранее неизвестен раздел,то готовится несколько загрузочных модулей.

Относительная загрузка

Позволяет загружать программы каждый раз с нового адреса.

При загрузке пересчитываются адресные поля команд ,использующих абсолютную адресацию.

Таблица перемещений.

Относительная загрузка с базовой адресацией

Несколько регистров используется для хранения начала текста, данных, стека.

Эти регистры не используются для программистом (компилятором), адресация в программе используется относительно их.

Команда (базовый регистр + смещение)

Загрузка с использованием позиционно - независимого кода.

Относительная адресация, адрес которой получается сложением адреса команды и её адресного кода.

-f PIC

В начале функции адрес точки её входа помещается в регистр и вся адресация осуществляется относительно его . Используется при создании разделяемых библиотек UNIX.

Оверлей = перекрытие.

Используется для отображения большого количества объектов в ограниченном адресном пространстве.

Необходимо правильно разделять процедуры между оверлеями.Менеджер оверлея при обращении к процедуре смотрит есть ли подключённая процедура в текущем оверлее, если есть-подключает.