В.П. ТИХОМИРОВ
М.И. ДАВИДОВ

ОПЕРАЦИОННАЯ СИСТЕМА

ДЕМОС

В.П. ТИХОМИРОВ
М.И. ДАВИДОВ

ОПЕРАЦИОННАЯ СИСТЕМА ДЕМОС:

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПРОГРАММИРОВАНИЯ

МОСКВА
"ФИНАНСЫ И СТАТИСТИКА"
1988

ББК 32.973.2-01
Т 46
УДК 681.3.015+681.3.066

Рецензент канд. физ.- мат. наук В.А. Серебряков

Т $\frac{2405000000-057}{010(01)-88}$ 116-88

ISBN 5-279-00113-9

ПРЕДИСЛОВИЕ

В связи с ростом затрат на изготовление прикладных программ и их постановку на различные ЭВМ в последние годы широко ведутся работы по унификации операционных систем. В качестве основы для унификации используются концепции и интерфейс операционной системы **UNIX [4]**, разработанной фирмой **Bell Laboratories** (США). В нашей стране также проводятся работы в этом направлении (ДЕМОС) **[5]**. Любая ОС этого семейства совместима с ОС **UNIX**, поэтому все вышедшие к настоящему времени книги по операционным системам, следующим стандарту ОС **UNIX**, в значительной степени пригодны пользователям ОС ДЕМОС. Однако эти книги адресованы начинающим пользователям и не содержат достаточно полного описания ключевых компонент системы.

Цель этой книги - восполнить существующий пробел. В первой главе дана характеристика основных принципов работы ОС и построенных на них методов программирования.

Во второй главе читатель найдет подробное описание языка **C-shell**, в третьей - описан язык **Make**, в четвертой - язык обработки структурированных текстов **AWK**, в пятой - генераторы программ лексического и синтаксического анализов, построенных на основе языков **LEX** и **YACC**. Эти языки реализованы в ОС ДЕМОС в виде интерпретаторов и текстовых процессоров, входящих в список инструментальных средств программистов. Объем книги совершенно недостаточен, чтобы можно было бы представить здесь и другие не менее важные компоненты системы.

Содержание глав свидетельствует о том, что книга посвящена программированию на различных языках ОС ДЕМОС и адресована программистам.

Читателям, желающим более подробно ознакомиться с внутренними механизмами работы системы, рекомендуем книгу **[6]**.

Авторы благодарят разработчиков семейства ДЕМОС, чья работа послужила основанием для издания данной книги. Трудно перечислить здесь всех программистов, внесших вклад в создание системы, но нельзя не отметить Антонова В.Г., Аншукова С.А., Бардина В.В., Буркова Д.В., Володина Д.В., Егошина Л.А., Коротаева М.В., Машечкину И.Г., Паремского М.В., Руднева А.П., Сауха Н.М., Скукина А.В., Флерова М.Н., Чижикова С.Е., Школьникова Ю.В.

Активное участие в разработке плана книги приняли В.В.Бардин и М.В. Паремский. Целиком рукопись была прочитана Бардиным В.В., им был высказан ряд полезных замечаний.

Введение и первая глава книги написаны В.П. Тихомировым, остальные главы - М.И. Давидовым.

ВВЕДЕНИЕ

В шестидесятых и семидесятых годах в нашей стране и за рубежом операционные системы создавались как уникальные программные изделия для каждой ЭВМ. При их разработке стали отчетливо проявляться негативные явления в программном обеспечении, которые можно определить следующим образом:

- подмена разработки новых программных средств повторной реализацией существующих (например, известно о существовании сотен функционально близких баз данных, так называемых интегрированных пакетов);
- уровень координации и организации коллективной работы по созданию новых программных средств остается низким. Отсутствие средств координации и организации взаимодействия разработчиков приводит к затратам на организацию работы, соизмеримым со стоимостью самой работы;
- с увеличением количества совокупного программного продукта затраты времени на его освоение пользователями становятся соизмеримыми со временем эксплуатации.

Одним из путей устранения указанных недостатков является создание унифицированной системы модульных средств, позволяющей повысить коэффициент использования готовых компонент за счет применения ранее созданных программ в новых разработках и сделать затраты на освоение соразмерными новизне продукта. Унифицированная система модульных средств может стать полезной лишь в том случае, если будет общепризнанным набором инструментальных средств в единой машинно-независимой операционной среде.

Для формирования системы модульных средств среди множества существовавших к этому времени операционных систем наилучшим образом

подходила операционная система UNIX. Семейства BSD и SYSTEM V образуют различные ветви этой операционной системы и можно предполагать, что со временем все различия между ними будут устранены.

Система UNIX проектировалась в первую очередь как инструментальная. Подходы, положенные в ее основу, возможность функционирования на ЭВМ различной архитектуры привели к тому, что впервые в мировой практике сформировался стандарт операционной системы, которому без каких-либо официальных соглашений следуют программисты многих стран.

Мы, по-видимому, становимся свидетелями уникального процесса - рождения единой среды системного программирования. Она включает стандарт на средства коммуникации между программистами, системами программ, системами компьютеров на основе единого множества программных изделий, часть которых обладает свойствами модульности.

В течение ряда последних лет усилия разработчиков в нашей стране и за рубежом были направлены на создание семейства унифицированных операционных систем. В настоящее время несколько сотен различных типов ЭВМ (от супер ЭВМ до персональных компьютеров) оснащены системами семейства унифицированных ОС.

В нашей стране представителями такого семейства являются Диалоговая Единая Мобильная Операционная Система (ДЕМОС), а также Инструментальная Мобильная Операционная Система (ИНМОС) и Мобильная Операционная Система (МОС).

Глава 1. СЕМЕЙСТВО УНИФИЦИРОВАННЫХ ОПЕРАЦИОННЫХ СИСТЕМ ДЕМОС

Семейство унифицированных операционных систем ДЕМОС образуют операционные системы для серийных типов ЕС ЭВМ, СМ ЭВМ, ЭВМ "Электроника" и других. Все ОС семейства ДЕМОС совместимы по программным продуктам, способам управления процессами и взаимодействию с пользователями.

Эволюция операционных систем еще не завершилась. Пользователи, для которых работа на ЭВМ не является основным занятием, предъявляют новые требования к операционным системам. В первую очередь это относится к необходимости перехода от "свалки программ и пакетов" к коммутации программных модулей.

Примером операционной системы, которая предоставляет минимум возможностей пользователю при постоянном (уже неконтролируемом) росте числа прикладных программ и пакетов, является операционная система MS-DOS. Только текстовых экранных редакторов здесь насчитывается несколько десятков, практически одинаковых по своим возможностям, они отличаются системами команд, которые должен изучить пользователь. Еще разительнее то, что часть пакетов вообще не взаимодействует с операционной системой MS-DOS, стирая ее в момент загрузки. Это означает, что в пакете имеются свои программы управления устройствами ЭВМ, процессом выполнения. И какое-либо взаимодействие с другими пакетами программ без дополнительных затарат невозможно даже на уровне машинных носителей данных.

В условиях конкуренции на рынках западных стран появляется все больше пакетов (несколько тысяч), работающих под управлением MS-DOS. Наибольшее негативное воздействие оказывает тот факт, что эти пакеты являются "самостоятельными" изделиями, между которыми трудно осуществить какое-либо взаимодействие.

Унифицированные операционные системы используют принципиально отличающийся подход. Он заключается в том, что каждая новая программа, функционирующая под управлением ОС, возникает либо как комплекс взаимодействия различных компонент, образующих окружение, либо как новая компонента.

Основу такого подхода образуют несколько принципов, реализованных в ДЕМОСе.

Иерархический принцип построения.

Отдельные функции системы выделяются согласно их сложности, характерному масштабу времени выполнения. Ни один нижний уровень функций системы не зависит от организации высших уровней, и ни один высший уровень не имеет доступа к внутренним механизмам нижних уровней.

Управление процессами.

Основной объект операционной системы - процесс. Взаимодействие процесса с внешним миром обеспечивается ядром. Ядро операционной системы является подпрограммой (функцией) процесса и эксплуатируется им во время выполнения. Процесс может порождать новые процессы и (если необходимо) выполняться независимо от них. Пользователь может приостановить процесс и продолжить его выполнение либо в синхронном, либо в асинхронном режиме.

Унификация понятия файл.

Если на объекте можно выполнить операции: создать, уничтожить, открыть, закрыть, прочесть, записать, то это - файл. Даже если этот объект является устройством, например линия передачи данных, оперативная память ЭВМ, диск, магнитная лента или клавиатура дисплея, тем не менее он фигурирует в операционной системе как файл.

Совместимость представления данных.

В ОС повсеместно выполняется принцип совместимостей логических форматов представления данных на внешних машинных носителях различных типов ЭВМ.

Переадресация ввода и вывода данных.

В системе предусмотрена независимость операций ввода-вывода от устройств. Такая организация ввода-вывода позволяет осуществить принцип отлаженного связывания процесса и устройства на основе общесистемной буферизации потоков данных. Например, программа сортировки может получать входные данные из файла либо непосредственно с клавиатуры дисплея, а выходные данные посылать либо в файл, либо на экран дисплея, либо на печатающее устройство. Пассивные (файлы) и активные (процессы) элементы системы ввода-вывода могут использоваться в качестве источников и приемников данных процесса без каких-либо ограничений.

Связь с пользователем.

Программа связи между пользователем и операционной системой создает процесс пользователя. В качестве программы связи (оболочки) пользователь может использовать любую программу, в том числе собственной разработки. В процессе работы пользователь может заменять программы связи.

Персонализация интерфейса.

Пользователю предоставляется возможность определить программную среду и компоненты окружения, с которыми он желает работать, присвоить им свои имена.

Коммутация модулей.

Коммутация готовых модулей - основной принцип к которому целесообразно стремиться при работе в системе. Пользователям предлагается выполнять большую часть работ уже существующими программами окружения, а не посредством создания новых. Программы окружения являются модулями, которые могут работать с различными источниками и приемниками данных, их можно коммутировать межпроцессными каналами, формируя конвейеры процессов-фильтров потока данных.

Мобильность текстов программ.

Тексты программ окружения мобильны в пределах семейства ДЕМОС, имеют общий комплект документации и одинаковые область и способ применений. Программы окружения системы отвечают требованиям, предъявляемым компоненте системы: наличие исходного текста программы и описания применения, возможность автоматической компиляции, сборки и установки компоненты в любой ЭВМ.

Текстовые процессоры.

Текстовая обработка осуществляется на основе использования ряда текстовых процессоров, построенных с применением процедурных или непроцедурных языков высокого уровня. Совокупность текстовых процессоров доступна каждому пользователю на всех этапах разработки программ и манипуляций различными текстами.

Обеспечение коллективной работы.

Система позволяет координировать коллективную работу пользователей над проектом. Этой цели служат текстовые процессоры, обеспечивающие ведение версий файлов, учет работ по проекту и их

координацию; развитая система интерфейсов между пользователями (электронная почта) и различными ЭВМ (сетевые средства).

Программирование по аналогии.

В системе реализован принцип "смотри и повторяй". Как правило, ОС ДЕМОС поставляется с исходными текстами компонент. Это позволяет при необходимости либо модифицировать существующую, либо создать новую компоненту по аналогии. При этом всегда можно воспользоваться хотя бы частью имеющихся текстов.

Перечисленные выше принципы, реализованные в ДЕМОСе, образуют совокупность методов и программных средств, доступных в работе. В настоящее время окружение ДЕМОС образуют около двухсот компонент, доступных пользователю.

К ключевым компонентам системы относятся командные языки. ОС ДЕМОС оснащена интерпретаторами нескольких командных языков, из которых наиболее важными являются **Shell** и **C-shell**. Командные языки ОС ДЕМОС соответствуют требованиям, предъявляемым алгоритмическим языкам программирования. Имеются опытные реализации командного языка как языка функционального программирования.

Наибольшей популярностью у программистов ДЕМОС пользуется командный язык **C-shell**, что послужило серьезным поводом для подробного описания его в книге, тем более что в отечественной и зарубежной литературе он описан неполно. Командный язык **Shell [2]** также часто используется в системе. В настоящее время он широко реконструируется: отлаживается, например, аппарат функций, многооконный режим работы и т.д. Поэтому было бы, видимо, преждевременно помещать материал об этом языке в книгу.

К числу ключевых компонент ДЕМОС относится также интерпретатор **make [11]**. Достаточно сказать, что для каждой установленной в ДЕМОС компоненты имеется программа на языке **Make**, при выполнении которой осуществляются автоматическая компиляция, сборка и установка ее в системе. Этот язык программирования неполно описан в отечественной и зарубежной литературе. В книге дано описание языка **Make**, достаточное для самостоятельного изучения.

В ОС ДЕМОС широко используются программы обработки текстовых файлов. Особое место в их ряду занимает интерпретатор **awk [10]**. В процессе выполнения программы на языке программирования **AWK** можно выполнить ряд нетривиальных преобразований текста минимальными усилиями на программирование. Часто это преобразования, которые традиционно выполняются средствами баз данных, например для генерации отчетов. Язык **AWK** является представителем семейства текстовых процессоров ОС ДЕМОС и также относится к числу ключевых компонент.

Генераторы программ лексического и синтаксического анализов **[14,13]** использованы при создании компиляторов, например в компиляторе языка Паскаль, в ряде других компонент. Они позволяют на языке высокого уровня описывать лексику и синтаксис нового языка, что существенно упрощает создание как компиляторов и интерпретаторов, так и специализированных языков, ориентированных на применение пользователями, для которых программирование не является основной работой.

Перечисленные выше методы программирования в ОС ДЕМОС обеспечены рядом средств, часть которых нашла отражение в этой книге. Запланированный объем книги не позволил авторам включить описание ряда других компонент ОС ДЕМОС: средств оперативной полиграфии, средств коммуникаций, средств отладки программ и т.д.

В следующих главах описаны компоненты системы, профессиональное владение которыми является важным условием эффективной работы в ОС ДЕМОС. При этом предполагается, что читатель знаком с языком программирования Си и с общими принципами работы в системе.

Глава 2. КОМАНДНЫЙ ЯЗЫК C-shell

Взаимодействие пользователя и операционной системы осуществляет интерпретатор команд - программа связи между пользователем и операционной системой. Основная функция интерпретатора - создание процессов, выполняющих задание (одну и более команд) пользователя, сформулированное в виде предложений (командных строк) некоторого формализованного языка - языка взаимодействия с операционной системой. Такой язык называют командным.

Командный язык позволяет выполнять различные задания пользователя и управлять работой операционной системы. Пользователи ОС ДЕМОС используют несколько стандартных командных языков, однако авторы решили ограничиться подробным описанием наиболее развитого и популярного из них - командного языка **C-shell** [12].

C-shell - язык управления заданиями со свойствами универсального языка программирования. Совмещение свойств языка управления заданиями и универсального языка программирования делает **C-shell** во многом схожим как с универсальными алгоритмическими языками, так и с наиболее развитыми командными языками. Это отражено в самом названии языка **C-shell**: **C** - от имени универсального языка программирования Си и **shell** - язык взаимодействия пользователя с системой (буквально "оболочка"). В качестве программы связи между пользователем и операционной системой используется интерпретатор **csh**, предназначенный для разбора и выполнения предложений на языке **C-shell**. Интерпретатор **csh** работает в двух режимах: интерактивном и неинтерактивном.

В интерактивном режиме пользователь формулирует задание обычно в виде одной командной строки, после выполнения которого формулируется следующее, в виде другой командной строки. В этом режиме можно выполнять задание из нескольких командных строк, образующих фрагмент программы или всю программу целиком. Такая программа может содержать условные выражения и циклы.

В неинтерактивном режиме выполняется командный файл (программа на языке **C-shell**), в котором содержатся командные строки и управляющие конструкции (операторы языка **C-shell**).

2.1. Лексическая структура языка C-shell

Предложение на языке **C-shell** формулируется в виде командной строки, которая может содержать команду ОС ДЕМОС (например, **/bin/cat**), внутреннюю команду интерпретатора **csh** (например, **cd**), оператор языка программирования **C-shell** (например, оператор цикла **while**).

Командная строка состоит из списка слов и их разделителей. Слово может включать имя переменной, файла, метасимволы и конструкции из них. Интерпретация слова может привести к тому, что слово будет заменено списком слов, т.е. строкой. В число переменных могут входить переменные, определенные программистом, так называемые переменные окружения (с ними мы познакомимся ниже), и предопределенные переменные интерпретатора **csh**. В общем случае интерпретатор **csh** выделяет следующие лексемы в командной строке: слово, разделитель слов и метасимвол.

Слово - это завершенная конструкция, которую распознает интерпретатор **csh**. Разделителями слов в командной строке могут быть пробелы, табуляции и перечисленные ниже символы:

; ( ) < > & |

Если необходимо использовать эти символы в качестве части слова, а не разделителя, то применяется экранирование символом \. Например, если символу ";" предшествует символ \, он будет восприниматься не как разделитель группы команд, а как символ ";" слова, которому принадлежит. Некоторая часть символов образует класс так называемых метасимволов - символов, имеющих специальное значение. Каждый из перечисленных ниже символов имеет специальное значение в языке **C-shell**. Специальное значение символа определяется контекстом слова или командной строки

! # $ % : * , ?

[ ] { } @ ~ . ^

Символ \ отменяет специальное значение части указанных метасимволов.

После разбора командной строки и подстановки значений переменных слово может "превратиться" в строку или остаться словом, например именем файла. Интерпретатор **csh** позволяет оперировать строками, полученными в результате интерпретации слов в командной строке, осуществлять различные преобразования:

"строка" 'строка' `строка`

Кавычки используются для управления режимом грамматического разбора и интерпретации командной строки. Двойные и одинарные кавычки можно экранировать символом \. Если командная строка занимает более одной строки, то ее можно продолжить на следующей, поставив в конце символ \.

Строка, заключенная в двойные кавычки, интерпретируется **csh**, в ней используются специальные значения метасимволов и выполняются подстановки значений переменных.

Строка, заключенная в одинарные правые кавычки (апострофы), не интерпретируется. Все метасимволы и их последовательности теряют свое специальное значение. В некоторых случаях символы

? . * ! ~

сохраняют свое специальное значение и интерпретируются в такой строке.

Строка, заключенная в левые одинарные кавычки, интерпретируется как командная строка. Эта командная строка выполняется и заменяется результатом ее выполнения.

Ниже перечислены лексемы - имена операторов языка C-shell и внутренних команд интерпретатора **csh**:

alias	endsw	logout	suspend
alloc	eval	newgrp	switch
bg	exec	nice	time
break	exit	nohup	umask
breaksw	fg	notify	unalias
case	foreach	onintr	unhash
cd	glob	popd	unlimit
chdir	goto	pushd	unset
continue	hashstat	rehash	unsetenv
default	history	repeat	wait
dirs	if	set	while
echo	jobs	setenv	
else	kill	shift	
end	limit	source	
endif	login	stop	

Имена предопределенных внутренних переменных интерпретатора **csh**:

argv	history	nonomatch	status
cdpath	home	notify	time
checktime	ignoreeof	path	verbose

child	mail	prompt
cwd	noclobber	savehist
echo	noglob	shell

В некоторых случаях одна лексема определяет и имя переменной, и имя внутренней команды интерпретатора **csh**. Тип лексемы определяется по контексту. Например, команда **time** хронометрирует выполнение простой командной строки, а предопределенная переменная с именем **time** используется для указания интерпретатору, о каких заданиях выводить результаты хронометрирования.

2.2. Форматы командных строк, перемещения по файловой системе

Интерпретатор **csh** получает задание в виде командной строки или командного файла. Последовательность символов от приглашения до символа "перевод строки" (\n) является командной строкой. Командная строка может включать простую команду, последовательность команд, группу команд, конвейер. Задание может выполняться в синхронном или асинхронном режимах. В результате разбора командной строки интерпретатор **csh** запускает на выполнение один или более процессов.

Командой мы называем любой объектный или командный файл, который может быть выполнен под управлением ДЕМОС. Например, команда

```
pr -2 -w39 -l24 -t file
```

выведет на экран дисплея содержимое файла **file** в две колонки, строками по 39 символов и страницами по 24 строки. Команда **pr** (объектный выполняемый файл которой размещен в каталоге **/bin**) выполняет собственно форматирование перед выводом файла. Режимы работы команды **pr** задаются ключами, им в командной строке предшествуют знаки минус или плюс. Знак используется, чтобы можно было ключ отличить от имени файла. Кроме того, в командной строке указано имя файла **file**, который необходимо обработать команде **pr**. Прежде чем команда **pr** начнет выполняться, интерпретатор выполнит следующую работу:

проанализирует ошибки в командной строке;

найдет выполняемый файл в одном из каталогов (в данном случае - в **/bin**);

передаст его на выполнение операционной системе вместе с ключами и именем файла.

По завершению выполнения командной строки интерпретатор **csh** напечатает приглашение (форма приглашения устанавливается пользователем). Это значит, что можно вводить следующую командную строку. Командная строка объединяет несколько слов, разделенных пробелами, первое из которых - собственное имя команды.

Часто бывает необходимо выполнить последовательность команд, в этом случае можно использовать символ ";", например:

```
cat < file ; pr -2 -w39 -l24 -t file
```

Эта командная строка приводит к выполнению двух команд: сначала **file** будет выведен на дисплей таким, какой он есть, затем командой **pr** со всеми указанными преобразованиями.

Для управления последовательностями команд допускается использование логических связок **&&** и **||**, например:

```
cat < file && pr -2 -w39 -l24 -t file
```

В этом случае вторая команда выполнится, если успешно выполнится первая, т.е. если **file** существует и его разрешено читать.

```
cat < file1 || pr -2 -w79 -l24 -t file
```

И в этом случае вторая команда будет выполнена, даже если первая не выполнится, например если **file1** отсутствует. Слова "успешно выполнится" имеют определенный смысл - завершившийся процесс должен вернуть код завершения, равный нулю. В некоторых версиях команд код нормального завершения процесса не равен нулю.

Для группирования команд используются круглые скобки. Группа команд, заключенная в скобки, выполняется как самостоятельная командная строка и не влияет на внутренние переменные других частей командной строки, например:

```
ls -l; ( cat < f1 && cat < f2 ) && date
```

Сначала выдается листинг рабочего каталога, затем выполняется группа команд в скобках и, если оба файла существуют и разрешено их чтение, выполняется команда **date**. Для безошибочного разбора командной строки интерпретатором **csh** требуется, чтобы около скобок находились либо ";", либо **&&**, либо **|**, либо **||**, либо метасимволы перенаправления ввода-вывода.

Когда бывает необходимо организовать последовательную обработку потока данных, используются межпроцессные каналы. Один процесс выводит поток данных в канал, другой читает из него. Если необходимо расширить число взаимодействующих процессов, то образуется конвейер команд. Для обозначения в командной строке межпроцессного канала выделен символ **|**, например:

```
cat -n < file | pr -2 -w39 -l24 -t
```

Команда **cat** проставляет номера строк в **file** (ключ **n**), и ее вывод передается команде **pr** для форматирования. Результат выводится на экран дисплея.

```
sort file | cat -n | pr -2 -w39 -l24 >> file2
```

Команда **sort** сортирует файл; **cat** - проставляет номера строк; **pr** - форматирует вывод и дописывает его в **file2**. Команды в конвейере можно разделять с помощью логических связок **&&** и **||**, группировать круглыми скобками, разделять ";", например:

```
( cat < f1 && date ) && ( cat -n < f1 | sort ); date
```

Если имеется файл **f1**, он выводится на дисплей, затем выводится дата, а команда **cat**, пронумеровав строки, направляет файл на сортировку. На экран выводятся результат сортировки и дата. Если **f1** отсутствует или его нельзя читать, то выводится только дата.

Интерпретатор позволяет перейти к приему новой командной строки, не дожидаясь завершения предыдущей. Такой режим выполнения называют асинхронным или параллельным. Это дает возможность пользователю запустить на выполнение несколько командных строк и продолжать работу в интерактивном режиме. Символ **&** в конце командной строки используют, когда необходимо выполнить ее асинхронно, например:

```
cat -n < f1 | pr -2 -w39 -l24 -t > f2 &
```

Команда **cat** выводит строки с номерами, команда **pr** форматирует их, вывод направляется в **f2**.

Часто при наборе командной строки возникают ошибки, которые можно исправить простыми средствами.

Клавиша ЗБ (DEL) дисплея используется для удаления символа, около которого находится курсор.

Символ СУ/W (CNTRL/W) позволяет удалить последнее слово командной строки.

Символ СУ/U (CNTRL/U) позволяет удалить всю строку.

Перемещения по файловой системе выполняются командами **cd**, **popd** и **pushd**. Интерпретатор хранит путь от регистрационного к рабочему каталогу (его имя хранится в предопределенной переменной **cwd**), а также поддерживает стек каталогов, содержимое которого выводится по команде **dirs**. Команды **pushd** и **popd** используются для переходов по дереву каталогов файловой системы и модификации содержимого стека каталогов. Команда **cd** не меняет содержимого стека каталогов. Элементы стека нумеруются от 1, начиная от вершины стека.

Команда **popd** без аргументов равносильна команде **cd** имя_номер_2 стека имен каталогов, т.е. осуществляется переход в новый каталог, имя которого определяется автоматически. Имя_номер_1 из стека имен каталогов удаляется, остальные элементы стека сохраняются с новыми номерами. Команда **popd** +число удаляет имя_номер_(1+число) из стека, остальные элементы стека сохраняются с новыми номерами. При этом переход в другой каталог не осуществляется.

Команда **pushd** меняет порядок имен в стеке имен каталогов и увеличивает их число на 1. Команда **pushd** без аргументов равносильна команде **cd** имя_номер_2 стека. При этом имя_с_номером_2 ставится в вершину, а имя_с_номером_1 - на его место в стеке (остальные элементы стека остаются на своих местах). Команда **pushd** каталог равносильна команде **cd** каталог, при этом каталог записывается в вершину стека, остальные элементы стека сохраняются с новыми номерами. Команда **pushd** +число равносильна команде **cd** имя_с_номером_(1+число). При этом имя_с_номером_(1+число) ставится в вершину стека, а "число" имен переписывается в конец стека в том порядке, в котором они следовали от вершины стека. Другие элементы стека остаются без изменений.

2.3. Управление вводом и выводом

В ОС ДЕМОС используются так называемые стандартный ввод, стандартный вывод и стандартный вывод диагностических сообщений. Стандартный ввод определяет источник данных для команды, стандартный вывод - приемник данных, стандартный вывод диагностических сообщений - приемник сообщений об ошибках.

Существуют два режима управления вводом и выводом: первый - режим умолчания; второй - режим с явным указанием источника и/или приемника данных. В режиме умолчания в качестве стандартного ввода (источника) используется клавиатура дисплея, в качестве стандартного вывода и стандартного вывода ошибок (приемники) - экран дисплея. Интерпретатор

позволяет менять (переадресовывать) источник и приемники данных. Переадресация осуществляется с помощью разделителей специального вида. Для указания направления ввода (источника) используются следующие разделители:

<, <<, << "слово"

Если разделитель не указан, ввод осуществляется с клавиатуры дисплея (стандартный ввод).

Для указания направления вывода (приемника) используются следующие разделители:

>	>>	|
>&	>>&	|&
>!	>>!	
>&!	>>&!	

Если разделитель не указан, вывод осуществляется на экран дисплея (стандартный вывод и стандартный вывод ошибок).

Метасимвол & используется, когда необходимо сообщения об ошибках выводить вместе со стандартным выводом, а не на экран. Метасимвол "!" используется, когда необходимо временно отменить действие некоторых ключей.

Для управления режимами ввода-вывода используются значения ключей **noclobber**, **noglob** и **nonomatch**. Если ключи установлены, то выполняется особый режим выполнения операций ввода-вывода. Установку и отмену ключей выполняют с помощью команд **set** и **unset**. Например:

set noclobber или **unset noclobber**

Рассмотрим подробнее управление вводом:

< имя_файла
: открывается файл, который читается вместо чтения с клавиатуры дисплея;

<< слово
: в качестве ввода используется ввод с клавиатуры дисплея. Ввод прекращается, когда введенная строка будет идентична слову. Например:

```
cat > file << mmm
0123
3456
```

6789
mmm

Команда **cat** создает **file** и ждет ввода с клавиатуры дисплея. Каждая введенная строка сравнивается с **mmm**. Если она отличается от **mmm**, то записывается в **file**. Если она идентична **mmm**, ввод прекращается и **file** закрывается. Строка **mmm** в выходной файл не вводится. Аналогичную конструкцию можно использовать в командном файле.

Рассмотрим подробнее управление выводом:

> имя_файла

результат направляется в указанный файл.

>! имя_файла

восклицательный знак отменяет действие ключа **noclobber**. Ключ запрещает вывод в файл, если он к этому моменту существует и не является специальным файлом (например, **/dev/tty***). Допустим, существуют файлы с именем **file1** и **file2** и выполнена команда

```
set noclobber
```

Тогда команда

```
cat < file2 > file1
```

не выполнится, а команда

```
cat < file2 >! file1
```

выполнится. Предопределенная переменная **noclobber** используется как ключ, запрещающий случайное повреждение существующих файлов. Конструкции >!, >>!, >&! и >>&! отменяют действие этого ключа для указанного в командной строке файла.

>& имя_файла или >&! имя_файла

в первом случае диагностические сообщения направляются в файл, во втором - будет сделано то же, но с отменой действия ключа **noclobber**.

>> имя_файла или >>! имя_файла

вывод помещается в конец файла. Если файл отсутствует, то он создается, во втором - будет сделано то же, но с отменой действия ключа **noclobber**.

>>& имя_файла или >>&! имя_файла

в первом случае **csh** добавит диагностические сообщения в файл, во втором случае будет сделано то же, но с отменой действия ключа **noclobber**.

Можно запретить изменение расширения имени файла. Для этой цели используется ключ **noglob**. В общем случае имя файла имеет вид: основа_имени.суффикс. Если установлен ключ **noglob**, изменение суффиксов имен существующих файлов будет порождать состояние ошибки. Чтобы отменить действие этого ключа для конкретных файлов, в командных строках можно указывать те же конструкции, что и при использовании ключа **noclobber**.

Существует возможность перенаправления сообщений об ошибках в конвейере. С этой целью используется конструкция **|&**. Ее использование в конвейере приведет к тому, что все диагностические сообщения будут направлены не на экран дисплея (стандартный вывод ошибок), а вместе с остальным выводом. Например, сообщения об ошибках в конвейере

```
cat < file1 |& pr -w79 -l24 > file2
```

будут направлены не на стандартный вывод (экран дисплея), а через **pr** в **file2**.

2.4. Управление процессами

Процесс является основным объектом ОС ДЕМОС, может выполняться синхронно и асинхронно. Синхронный процесс - это процесс, который на все время выполнения прерывает связь между пользователем и интерпретатором, асинхронный процесс выполняется параллельно с **csh**. Командная строка может порождать несколько процессов, каждому из которых присваивается уникальный номер - идентификатор. Идентификатор используется для управления процессом. Существуют два типа идентификаторов процесса: системный и внутренний. Системный идентификатор процесса выводится командой **ps** и относится к каждому процессу. Каждому системному идентификатору процесса ставиться в соответствие уникальный внутренний идентификатор. Внутренний идентификатор процесса известен только пользователю, относится ко всем процессам, порожденным одной командной строкой, и используется для управления процессом в командах **fg**, **bg**, **stop**, **kill**, **notify**. Процесс может находиться в двух состояниях - выполняться или быть приостановленным. Механизм управления процессом включает следующие средства:

завершение выполнения синхронного (асинхронного) процесса;

приостановление (возобновление) выполнения синхронного (асинхронного) процесса;

изменение режима выполнения процесса с синхронного на асинхронный и наоборот;

вывод сообщения о состояниях асинхронного процесса после его завершения или в момент изменения;

управление вводом-выводом при выполнении процесса;

послать сигнал процессу;

управление ресурсами процесса.

Сведения о процессах хранятся интерпретатором в виде таблицы. Актуальное состояние таблицы можно получить, выполнив команду **jobs** или **jobs -l**. Во втором случае выводится более подробная информация. После выполнения команды **jobs** получим, например:

```
[1]     Остановлен    cc -c *.o
[2]   - Остановлен    make install
[3]   + Остановлен    red file
[4]     Выполняется   sort file > result &
[5]     Выполняется   mx -q -N -u -s *.m > out &
```

Каждая строка таблицы содержит сведения о конкретном процессе. В квадратных скобках указан внутренний идентификатор процесса. Существует несколько способов указания идентификатора в командной строке при ссылке на элемент таблицы процессов:

% или **%+**
: последний из приостановленных;

%-
: предпоследний из приостановленных;

%номер
: любой с таким внутренним идентификатором;

%символы
: любой с такими первыми символами строки;

%?шаблон**?**
: любой процесс с таким шаблоном в строке.

Запуск синхронного процесса осуществляется в результате выполнения командной строки, на конце которой нет символа **&**. Чтобы прекратить выполнение синхронного процесса, необходимо напечатать символ **СУ/С (CNTRL/C)**. Если нужно прекратить выполнение синхронного процесса с сохранением в файле образа памяти, то необходимо напечатать символ **СУ/\ (CNTRL/\)**. При этом в рабочем каталоге будет записан файл с именем **core**. Файл **core** является копией памяти, которую занимал в ОЗУ процесс в момент, когда был послан сигнал СУ/\.

Синхронный процесс может выводить информацию на экран дисплея. В этом случае, чтобы приостановить вывод, необходимо напечатать символ **СУ/S (CNTRL/S)**, чтобы продолжить выполнение, необходимо напечатать символ **СУ/Q (CNTRL/Q)**. Если синхронный процесс не выводит информацию на дисплей, чтобы его приостановить, необходимо напечатать символ **СУ/Z (CNTRL/Z)**, а чтобы продолжить - напечатать **fg** или **%**. Описанные способы управления синхронным процессом существенно различны. Первый используется только в случае вывода на дисплей, второй можно использовать в любом случае. Кроме того, при использовании **СУ/Z** синхронный процесс приостанавливается с возобновлением работы **csh**, а при использовании **СУ/S** этого не происходит. Существует возможность перевести процесс из синхронного в асинхронный режим выполнения. Для этого необходимо приостановить его, используя **СУ/Z**, затем напечатать **bg**. С этого момента и до своего завершения процесс будет выполняться асинхронно.

Запуск асинхронного процесса осуществляется в результате выполнения командной строки, в конце которой указан символ **&**. После запуска асинхронного процесса на экран дисплея выводится сообщение вида

```
[число]    число
```

Число в квадратных скобках - внутренний идентификатор процесса, число без скобок - системный идентификатор процесса. Чтобы приостановить выполнение асинхронного процесса, необходимо напечатать

stop %идентификатор_процесса

или

stop %строка

В формате **stop %**строка в качестве строки используется одна из форм ссылки на таблицу процессов. Допустим, имеется несколько асинхронно выполняемых процессов:

```
1  sort file > /tmp/result &
```

```
2  cc *.c >& errors &
3  lint *.c >& mymsg &
```

Первый можно остановить так: **stop %1** или **stop %sort**; второй - **stop %2** или **stop %c**; третий - **stop %3** или **stop %li**, или **stop %?mymsg?**. Чтобы возобновить выполнение приостановленного процесса, используется команда **fg** для запуска его как синхронного и **bg** для запуска его как асинхронного. Варианты запуска на асинхронное выполнение приостановленного процесса:

```
%идентификатор_процесса &
%строка &
bg %идентификатор_процесса
bg %строка
fg %идентификатор_процесса &
fg %строка &
```

Варианты запуска на синхронное выполнение приостановленного процесса:

```
%идентификатор_процесса
%строка
fg %идентификатор_процесса
fg %строка
```

Часто возникает необходимость принудительно (до нормального окончания) завершить работу асинхронного процесса. При этом не важно, выполняется он или приостановлен. В этом случае процессу необходимо послать сигнал принудительного завершения:

kill -KILL %идентификатор_процесса

или

kill -KILL %строка

или

kill -KILL системный_идентификатор_процесса

Для приостановки интерпретатора используется команда **suspend**, так как команду СУ/Z интерпретатор игнорирует.

Асинхронные процессы производят вывод стандартным образом. Вывод можно запретить, выполнив команду

stty tostop

Если установлен ключ терминала **tostop**, все асинхронные процессы будут останавливаться при попытке использовать стандартный вывод. При попытке чтения с терминала асинхронный процесс останавливается в любом случае. Пример:

```
stty tostop
cat < file &
[1] 285
jobs
[1]  + Ждет (вывод на терминал) cat < file
```

Чтобы возобновить вывод, необходимо выполнить команду

fg %1

или

fg %cat

Чтобы отменить ключ **tostop**, следует выполнить команду

stty -tostop

Команда **kill** используется для посылки сигнала процессу. Форматы команды **kill**:

kill -l
вывести список сигналов;

kill %номер или **kill %**строка
послать сигнал **TERM** (окончить) процессу;

kill системный_идентификатор_процесса
послать сигнал **TERM** процессу;

kill -SIG процесс
послать сигнал **SIG** процессу.

Сигналы задаются либо по их номерам, либо по их именам (как они заданы в **/usr/include/signal.h** без префикса **SIG**). Обычно все сообщения об изменении состояний процесса выводятся на экран дисплея после вывода приглашения (не в момент выполнения другого процесса). Переменная **notify**, если она установлена, обеспечивает вывод всех сообщений асинхронно. Можно установить режим **notify** для всех процессов или для конкретного процесса:

notify - установлен для всех процессов;

unset notify - отмена режима;

notify %номер - установлен для процесса с номером.

Для 32-битных машин возникает проблема ограничения ресурсов, выделенных процессу. Имеется возможность ограничивать следующие типы ресурсов:

cputime
максимальное число секунд центрального процессора, отводимое каждому процессу;

filesize
максимальный размер файла, который можно создать;

datasize
максимальное увеличение области (данные + стек) с помощью **sbrk(2)** за границу текста программы;

stacksize
максимальный размер автоматически расширяемой области стека;

coredumpsize
максимальный размер дампа, который будет создан.

Для определения размера ресурса используется команда **limit**, для отмены - команда **unlimit**. Для всех типов ресурсов, кроме **cputime**, используется размерность **k** - Килобайты или **m** - Мегабайты. Для **cputime** используются **ss** - секунды, **m** - минуты, **h** - часы. Для указания типа ресурсов используются приведенные выше ключевые слова. Примеры:

limit cputime 1m (1 минута)

limit cputime 05:30 (5минут **30** секунд)

limit filesize 2m (размер файла до **2** Мбайт)

2.5. Шаблоны имен файлов и каталогов

Если слово содержит один из символов *****, **?**, **[**, **]**, **{**, **}** или начинается с символа **~** (надчеркивание), то оно становится кандидатом на подстановку имени файла. При этом данное слово рассматривается как шаблон, который заменяется отсортированным в алфавитном порядке списком имен файлов, которые соответствуют шаблону. Если ни для одного возможного варианта подстановки имени по шаблону не находится, то порождается состояние ошибки. Ниже раскрыты значения символов в шаблонах имен файлов и каталогов.

*****
соответствует любой последовательности символов. Например, по команде **echo *** будут показаны имена всех файлов рабочего каталога, по команде **echo */*/f*** будут просмотрены входящие каталоги и выведен список всех файлов, имена которых начинаются на **f**.

?
соответствует одному любому символу. Пример:

```
% ls f.?
f.c
f.o
f.s
```

будут выведены имена всех файлов каталога, которые в качестве основы имени имеют символ **f**, а в качестве расширения имени - один любой символ.

[последовательность_символов**]**
соответствует любому символу указанной последовательности. Пример:

```
% ls f.[cso]
f.c
f.o
f.s
```

[символ-символ**]**
соответствует любому символу из лексикографически упорядоченного диапазона. Пример:

```
% ls f.[a-z]
f.c
f.o
f.s
```

Выводятся все файлы, расширение имени которых - любая строчная буква латинского алфавита.

~ (надчеркивание)
регистрационный каталог, например по команде

```
ls -l ~
```

будет выведено содержимое каталога, в который вы попадаете при входе в систему. Такой каталог принято называть регистрационным.

~регистрационное_имя_пользователя

регистрационный каталог пользователя. Пример:

```
% ls ~Иванов
```

будет выведено содержимое регистрационного каталога пользователя с именем Иванов.

{список_символов_через_запятую}

соответствует любому из перечисленных символов. Пример:

```
% ls f.{c,s,o}
f.c
f.o
f.s
```

слово1{список_слов_через_запятую}слово2

к слово1 добавляется первое слово из списка слов и добавляется слово2. Далее повторяется для второго слова из списка слов и так далее. Не допускаются пробелы. Слово1 и/или слово2 могут отсутствовать. Пример:

```
% ls aaa{ddd,ccc,bbb}eee
aaadddeee не найден
aaaccceee не найден
aaabbbeee не найден
```

При поиске файлов формировались их имена комбинациями указанных последовательностей. Сообщение "не найден" выводится, когда такой файл командой ls не обнаружен.

Описанная конструкция используется не только для подстановки имен файлов, но и для генерации различных слов, например:

```
% echo Та{ня,ню,не,тьяна,тьяну,тьяне}
Таня Таню Тане Татьяна Татьяну Татьяне
```

Возможны и более сложные применения, например:

```
%echo Та{ня,тьяна,нечка}{' Иванова',' Петрова'}
Таня Иванова Таня Петрова
```


Татьяна Иванова Татьяна Петрова
Танечка Иванова Танéчка Петрова

При создании шаблона имени файла особое место занимает символ точка в первой позиции имени. Если имя файла имеет вид ".имя", то точку необходимо указать явно.

В ряде случаев метасимволы шаблонов имен файлов и каталогов не экранируются символом \. Необходима определенная осторожность при использовании этих символов там, где по контексту не следует выполнять подстановки имен файлов и каталогов.

Выше было сказано, что если указанному шаблону не соответствует ни одно имя, то порождается состояние ошибки. В некоторых случаях это может привести к нежелательному прекращению выполнения задания. Существует ключ, связанный с шаблонами и именами файлов, - **nonomatch**. Если он установлен, интерпретатор **csh** сообщает, что указанному в командной строке шаблону не соответствует ни один файл в каталоге и этот шаблон выводится на стандартный вывод. Пусть, например, имеются файлы **f.1**, **f.2** и **f.3**. Выполним команду **echo f.[4-9]**, получим сообщение об ошибке:

```
echo: Нет таких имен.
```

Теперь выполним команды

```
set nonomatch; echo f.[4-9]
```

получим сообщение: **f.[4-9]**. В первом случае было выведено сообщение команды **echo** об ошибке. Во втором случае был выведен шаблон **f.[4-9]**, относительно которого не удалось подобрать имя файла. Принципиальное отличие второго случая от первого заключается в том, что не порождалось состояние ошибки - просто был выведен шаблон.

2.6. Подстановки значений переменных

В языке **C-shell** определены следующие типы переменных: слово, строка, массив слов, позиционная переменная.

В отличие от других языков программирования, когда указание имени переменной (например, в выражении) приводит к подстановке ее значения, в языке **C-shell** требуется явно указывать, когда собственно необходимо использовать значение переменной. Определить переменную и присвоить ей значение можно с помощью команды **set** или присвоив переменной значение выражения (например, **@ a = 5**). Подстановка значений переменных

выполняется после подстановки имен команд и файлов, если перед именем переменной указан без пробела символ **$**. Символы **?**, **#** используются для управления режимами подстановок. Имеются следующие типы подстановок:

подстановка собственного значения переменной;

подстановка собственного значения позиционной переменной;

подстановка информации о переменной, например определена она или нет.

Позиционные переменные инициализируются в командной строке при запуске командного файла на выполнение. Например,

comfile aaa bbb ccc ddd

каждое слово этой командной строки доступно внутри командного файла **comfile**. Чтобы получить значение слова, достаточно указать его номер в строке, например вместо **$0** подставится **comfile**, вместо **$3** - **ccc**. Рассмотрим примеры простых подстановок.

```
% set a = 5
% echo a
a
% echo $a
5
```

Действие символа **$** можно отменить, указав перед ним символ \. Например:

```
% set a = 5
% echo \$a
$a
% echo $a
5
```

В последовательности символов, взятых в двойные кавычки, всегда действует символ **$**, например:

```
% set a = 5
% echo \$a
$a
% echo "\$a"
```



```
\5
% set b = "Миша, Валера, $a."
% echo $b
Миша, Валера, 5.
%
% set b = "Миша, Валера, \$a."
% echo $b
Миша, Валера, \5.
```

В последовательности символов, взятых в одинарные правые кавычки, действие символа **$** отменяется, например:

```
% set a = 5
%set b = 'Миша, Валера, $a.'
% echo $b
Миша, Валера, $a.
%
% set b = 'Миша, Валера, \$a.'
% echo $b
Миша, Валера, \$a.
```

В последовательности символов, взятых в одинарные левые (обратные) кавычки, символ **$** приводит к подстановке переменной. Подстановка происходит непосредственно перед выполнением команды, взятой в левые кавычки. Пример: пусть имеется командный файл **f.csh**:

```
# !/bin/csh
set a = 1  b = ``echo $v``  c = 3
echo a = $a b = $b c = $c
set v = 100
@ b++
echo b = $b
@ d = ( $a + $b + $c )
echo d = $d
```

Первая строка начинается символом # - так в C-shell-программе обозначается строка-комментарий. Запись "!/bin/csh" указывает, что командный файл должен выполняться интерпретатором csh. В результате выполнения получим:

```
% f.csh
a = 1 b = `echo $v` c = 3
```

```
b = 101
d = 105
```

Здесь переменным **a**, **b**, **c** и **v** присваиваются различные значения, которые затем используются в вычислениях в строках, помеченных символом **@**. Все действия сопровождаются выводом значений переменных. В строке **set b = ``echo $v``** переменной **b** присваивается строка символов. Одиночные правые кавычки запрещают подстановку команде **set** и при выводе значения **b** имеем: **b = `echo $v`**. В строке **@ b++** выполняется команда **`echo $v`** (это значение переменной **b** к этому моменту), после чего значение переменной **b** увеличивается на **1**. Величина **b** равна теперь **101** и используется далее.

При выполнении подстановок особую роль выполняют фигурные скобки. Допустим, имеется необходимость осуществить конкатенацию значения переменной с некоторой последовательностью символов. Тогда, используя фигурные скобки, получим:

```
% set a = 'Опер'
% echo ${a}ционная система
Операционная система
% echo ${a}ция
Операция
% echo Исследование ${a}аций
Исследование Операций
```

Ниже перечислены все формы подстановок в различных режимах:

$**переменная или **${переменная**}**
: подставляется значение переменной; {} выделяют переменную от других символов, идущих следом без разделителя;

$переменная**[**номер**]** или **${**переменная**[**номер**]}**
: подставляется слово "номер" из массива слов;

$?**переменная или **${?переменная**}**
: подставляется 1, если переменная определена, иначе 0;

$$
: подставляется десятичный номер процесса **csh**;

$#**переменная или **${#переменная**}**
: подставляется количество слов, хранящихся в массиве;

$?0 или **${?0}**
: подставляется 1, если входной файл определен, иначе 0;

$0
: подставляется имя файла, который в данный момент времени выполняет интерпретатор;

$**число или **${число**}**
: эквивалентно **$argv[**число**]**;

$*
: эквивалентно **$argv[*]**, т.е. это строка параметров команды, которой был запущен **csh**.

Ниже приведены примеры различных вариантов использования подстановок в командном файле **f.csh**. Показаны **23** различных случая использования подстановок, каждый случай выделен строкой комментариев. За символом **#** в комментариях указаны номера вариантов. Чтобы легче было ориентироваться в выводе результатов, он организован таким образом, что сначала выводится номер варианта, потом результат его выполнения:

```
# !/bin/csh
      set m = ( w1 w2 w3 w4 w5 )
# 1 - 2
        if( $?m ) then
                echo 1 $m
        else
                 echo 2 0
        endif
# 3 - 4
        if( ${?m} ) then
                echo 3 $m
        else
                 echo 4 0
        endif
# 5 - 6
        if( $?0 ) then
                echo 5 1
        else
                echo 6 0
        endif
# 7 - 8
        if( ${?0} ) then
                echo 7 1
        else
                echo 8 0
        endif
# 9
        echo 9  $$
        @ p = ( $$ + 10 )
```

```
# 10
        echo 10 $p
        date > ofile$$
# 11
        echo  "11   `ls ofile$$`"
        cat ofile$$
        set p = 2
        @ p++
# 12 - 17
        echo  12 p = $p
        echo  13 ${p}aaa
        echo  14 $m[1]
        echo  15 $m[5]
        echo  16 $m[2-4]
        echo  17 ${m[3]}aaa
# 18 - 23
        set p = $#m
        echo  18 $p
        echo 19 ${#m}aaa
        echo  20 $0
        echo  21 $3
        echo  22 ${3}ddd
        echo  23 $*
```

Запустим на выполнение этот командный файл со следующим списком аргументов: 11 22 33 44 55 66. После выполнения получим:

```
% f.csh 11 22 33 44 55 66
1 w1 w2 w3 w4 w5
3 w1 w2 w3 w4 w5
5 1
7 1
9 525
10 535
11   ofile525
Вос Июл 10 22:57:20 MSK 1988
12 p = 3
13 3aaa
14 w1
15 w5
16 w2 w3 w4
17 w3aaa
```



```
18 5
19 5aaa
20 f.csh
21 33
22 33ddd
23 11 22 33 44 55 66
```

Результаты всех вариантов, кроме 11, легко объяснимы. Здесь номер процесса используется для создания файла с уникальным именем. Этот прием часто используют, когда необходимо получить такой файл. Варианты 20 - 23 демонстрируют использование переменной **argv**. Эта переменная получает свое значение из командной строки **f.csh 11 22 33 44 55 66**, которую мы ввели при запуске командного файла.

Конструкция $< используется для чтения строки с клавиатуры дисплея. Если в списке переменных указана конструкция $<, то интерпретатор ждет ввода с клавиатуры дисплея и заменяет $< на напечатанную строку

```
% set цвета = $<
красный синий зеленый
% echo $цвета
красный синий зеленый
% @ aaa = 1 + $< + 3
2
% echo $aaa
6
% echo $<
красный синий зеленый
красный синий зеленый
% set цвета = ( красный синий зеленый $< серый )
белый
% echo цвета: $цвета
цвета: красный синий зеленый белый серый
% echo $цвета[4]
белый
% set цвета[4] = $<
голубой
% echo $цвета[4]
голубой
```

В некоторых версиях интерпретатора **csh** конструкция $< не используется. В этом случае можно воспользоваться командой системы **rline**, например:

```
% set a = `rline`
ДЕМОС
% echo $a
ДЕМОС
```

2.7. **Модификаторы переменных**

Часто бывает необходимо изменить значение переменной и использовать в подстановке измененное значение. Для изменения значений переменных используются так называемые модификаторы, которые предназначены в основном для манипуляций именами файлов и каталогов. Как правило, модификаторы записываются в виде

модифицируемое_слово:модификатор

Ниже перечислены все формы использования модификаторов:

h удалить имя файла, сохранив компоненты пути (то есть удалить в слове текст справа до ближайшего символа /);

gh применить модификатор **h** глобально, ко всем словам;

r удалить расширение имени файла, указанное через точку, и саму точку;

gr выполнить модификатор **r** глобально, ко всем словам;

e удалить имя файла вместе с точкой, сохранив расширение имени;

ge выполнить модификатор **e** глобально, ко всем словам;

t сохранить имя файла, удалив компоненты пути (то есть удалить текст слева от самого правого символа / и сам этот символ);

gt применить модификатор **t** глобально, ко всем словам;

q запретить дальнейшую модификацию слова. Слово заключается в кавычки;

x разбить на слова по разделителям и запретить дальнейшую модификацию. Результат заключается в кавычки.

Бессмысленно применять модификаторы к следующим синтаксическим формам подстановок:

$?переменная
${?переменная**}**
$?0
$$

В каждой подстановке можно использовать только один модификатор. Перед модификатором должен стоять символ двоеточие. Если в подстановке используются символы {}, модификатор должен находиться внутри фигурных скобок. В командном файле **f.csh** показаны примеры использования модификаторов:

```
#!/bin/csh
        set a = /usr/bin/pr1
        set b = /dir1/dir11/dir111/file.c
        set c = ( /d/d1/d2/f.c /d/d1/c.c /d5/f.s )

# Вариант 1
        echo 'Вариант 1'
        echo $a:h
        echo $b:h
        echo $c[1]:h
        echo ${c[2]:h}
        echo $c[3]:h

# Вариант 2
        echo 'Вариант 2'
        echo $a:t
        echo $b:t
        echo $c[1]:t
        echo ${c[2]:t}
        echo $c[3]:t

# Вариант 3
        echo 'Вариант 3'
        echo $a:r
        echo $a:e
        echo $b:r
        echo $b:e
        echo $c[1]:r
        echo $c[1]:e
        echo ${c[2]:r}
        echo ${c[2]:e}
        echo $c[3]:r
        echo $c[3]:e

# Вариант 4
        echo 'Вариант 4'
```

```
        echo $c:gh
        echo $c:gt
        echo $c:gr

# Вариант 5
        echo 'Вариант 5'
        set m = "$c:x"
        echo $m
        echo $m[1]:t
        echo $m[1]:r

# Вариант 6
        echo 'Вариант 6'
        set m = "$c:q"
        echo $m
        echo $m[1]:h
        echo $m[1]:r
```

После выполнения командного файла получим:

```
% f.csh
Вариант 1
/usr/bin
/dir1/dir11/dir111
/d/d1/d2
/d/d1
/d5
Вариант 2
pr1
file.c
f.c
c.c
f.s
Вариант 3
/usr/bin/pr1
пустая строка
нет расширения имени у файла pr1
/dir1/dir11/dir111/file
c
/d/d1/d2/f
c
/d/d1/c
```



```
c
/d5/f
s
Вариант 4
/d/d1/d2 /d/d1 /d5
f.c c.c f.s
/d/d1/d2/f /d/d1/c /d5/f
Вариант 5
/d/d1/d2/f.c /d/d1/c.c /d5/f.s
f.s
/d/d1/d2/f.c /d/d1/c.c /d5/f
Вариант 6
/d/d1/d2/f.c /d/d1/c.c /d5/f.s
/d/d1/d2/f.c /d/d1/c.c /d5
/d/d1/d2/f.c /d/d1/c.c /d5/f
```

Вывод результатов организован таким образом, чтобы можно было отличить действия каждого варианта использования модификаторов. Варианты 5 и 6 демонстрируют действия модификаторов **x** и **q** . Переменная **c** содержит список слов. Этот список присваивается переменной **m** и к нему поочередно применяются модификаторы **x** и **q**. Из примера видно, что этот список превращается в одно слово, и модификаторы **h**, **t** и **r** работают с этим списком как с одним словом.

2.8. **Выражения**

Символ @ в начале строки означает, что все указанные далее слова образуют выражение, например:

```
% set a = 5  b = 7
% @ c = ( $a + $b )
% echo $c
 12
```

В языке **C-shell** числом считается любая символьная строка, которая может интерпретироваться как целое десятичное число. Если в выражении применяется строка, которая не может интерпретироваться как число, порождается состояние ошибки. В качестве логических значений используются числа (0 - ложь и 1 - истина). Как истина воспринимается любое число, отличное от нуля. Выражения и операции в **C-shell** в основном аналогичны операциям в языке Си. Выражения можно использовать также в операторах **if_then_else**, **while**, **exit**. В выражение в качестве простого

операнда можно включать команду системы, взятую в фигурные скобки. В процессе вычислений эта команда будет выполнена, и результат будет подставлен в выражение. В выражениях используются следующие операции:
операции сравнения строк

== или =~ равно?

!= или !~ не равно?

В качестве операндов в этих операциях применяются строки. В формах с символом ~ разрешается использовать в правой части шаблоны *, ? и [...]. Эта форма сравнения строк сокращает количество конструкций **switch** в командных файлах.

операции над числами

операнд_целое_число:

(два операнда):

||	или
&&	и
<=	меньше или равно?
>=	больше или равно?
>	больше?
<	меньше?
+	сложить
-	вычесть
*	умножить
/	делить
%	остаток деления

(один операнд):

++	инкремент, (постфиксная)
--	декремент, (постфиксная)

поразрядные_логические:

&	умножение
|	сложение
^	исключающее или
~	дополнение (унарная)
!	отрицание (унарная)
>>	сдвиг вправо
<<	сдвиг влево

операции опроса свойств файла:

они записываются в виде -код имя_файла, где код - одна из следующих букв, обозначающих операцию:

r (можно ли читать?)
w (можно ли писать?)
x (можно ли выполнять?)
e (файл существует?)
o (выполняющий владелец файла?)
z (размер файла = 0 ?)
f (файл зашифрован?)
d (файл - это каталог?)

Если файл обладает требуемым свойством, то возвращается значение истина, иначе ложь.

Строка с выражением может иметь следующие форматы:

символ @

распечатать значения всех переменных;

@ имя операция_присваивания выражение

присвоение переменной имя значения выражения;

@ имя[целое] операция_присваивания выражение

присвоение значения выражения элементу массива.

Операции присваивания аналогичны подобным операциям языка Си:

= += -= *= /= %=

Порядок выполнения операций определяется либо по умолчанию, с учетом старшинства, либо явным указанием круглых скобок. В показанном ниже ряду операций старшинство операций растет слева направо:

||, &&, | , ^ , & , ==, !=,

<=, >=, < , > , <<, >>, + ,

- , * , / , % , ! , ~ , ( )

Внутри указанных ниже групп операции имеют одинаковый приоритет:

[=~, !~, ==, !=]

```
[  <,  >, <=, >= ]
[        <<, >> ]
[         +,  - ]
[     *,  /,  % ]
```

Все знаки операций и знак присваивания должны отделяться от операндов пробелами. Части выражений, содержащие знаки операций, необходимо брать в круглые скобки. Перед круглыми скобками и за ними должен идти пробел. Количество пробелов, знаков табуляции не ограничивается, эти символы являются только разделителями.

Ниже приведен пример командного файла, содержащего строки с выражениями:

```
# !/bin/csh
# вариант 1
        set a = 5
       @ a++
        echo $a
        # результат: 6

# вариант 2
       @ a = 7
        echo $a
        # результат: 7

# вариант 3
        set a = 10  b = 15
       @ c = ( $a + $b )
        echo $c
        # результат:  25

# вариант 4
       @ c += 5
        echo $c
        # результат:  30

# вариант 5
       @ c = ( $c * 5 )
        echo $c
        # результат:  150

# вариант 6
```



```
       @ c *= 5
        echo $c
        # результат:  750

# вариант 7
       @ c *= $c
        echo $c
        # результат:  -27324

# вариант 8
        set  a = 5  b = 7  c = 9
        @ a = ( $a << 2 )
        @ b = ( $b >> 2 )
        @ c = ( $c + $a + $b )
        echo $a $b $c
        # результат:  20 1 30

# вариант 9
        set a = 5  b = 3
        @ c = ( $a | $b )
        echo $c
        # результат:  7

# вариант 10
        set a = 5 b = 3 c = 2
        @ d = ( ( $a + $b ) + ( $a + $b + $c ) )
        echo $d
        # результат:  18

# вариант 11
        set a = 5
        if( "$a" == "5" ) echo 'Строки идентичны'
        if( "$a" != "5" ) echo 'Строки различны '
        # результат:  Строки идентичны

# вариант 12
        date > file
        chmod 755 file
        if( -x file ) echo 'Выполняемый'
        # результат:  Выполняемый
```

Варианты 1 - 6 самообъяснимы. В варианте 7 получилось отрицательное

число, так как C-shell оперирует переменными типа **integer** (два байта на 16-разрядной ЭВМ).

В выражениях можно использовать операции запроса свойств файла:

```
% set c = 0
% @ c = -x  a.out  + 100
% echo $c
101
% @ c = 100 - -x a.out
% echo $c
99
```

Можно указать в фигурных скобках простую команду. В качестве значения будет использован код возврата выполненной команды

```
% @ c = 100 - { date }
сре фев 3 14:48:37 МСК 1988
% echo $c
99
% @ c = { date rrrjjj } - 100
Дата: плохой формат
% echo $c
-100
```

Имеются полезные различия в операциях сравнения строк:

```
==    !=
```

или

```
=~    !~
```

В первом случае в правых частях сравнения не интерпретируются шаблоны

```
?   .   *   [...]
```

во втором - интерпретируются. Пусть в рабочем каталоге имеются файлы и каталоги, тогда

```
% set c = "*"
% if( "$c" == "*" ) ____
```


или

```
% if( "$c" =~ "*" ) ____
```

В первом случае выражение всегда ложно - левая строка заменяется списком имен файлов и каталогов, а правая остается без изменений (символ *). Во втором случае строки всегда идентичны - они одинаково интерпретируются.

2.9. Операторы языка C-shell

Язык C-shell включает следующие операторы: **foreach**, **switch**, **while**, **if_then_else**, **goto**, **continue**, **break**, **shift**, **exit**. Операторы можно использовать в интерактивном режиме работы интерпретатора **csh** и в командном файле. Принципиальных различий выполнения операторов в интерактивном режиме и в командном файле нет. Рассмотрим работу операторов в интерактивном режиме. В процессе ввода оператора интерпретатор приглашает символом ? продолжать набор, пока не встретит ключевое слово, означающее конец ввода. Введенный текст можно рассматривать как временный командный файл, который интерпретируется и после выполнения уничтожается. Одно из ключевых слов **foreach**, **switch**, **while**, **if_then_else** или **goto** должно быть первым словом в строке.

Оператор цикла **foreach**

```
foreach имя (список слов)
        ...
end
```

Переменной "имя" последовательно присваиваются значения каждого члена "списка слов" и выполняется последовательность команд тела цикла **foreach**. **Foreach** и **end** должны находиться в отдельных строках

```
% foreach i  ( a b c d e f g h )
?         if( "$i" == "c" ) continue
?         glob  "$i "
?         if( "$i" == "f" ) break
? end
a b d e f %
```

Переменная цикла i последовательно принимает значения из списка, объявленного в предложении **foreach**. Внутри цикла стоят две

проверки. Если значение i равно c, то перейти к следующему шагу цикла, если значение i равно f, то прекратить выполнение цикла. В первом случае оператор **continue** требует перехода к новой итерации цикла, во втором - оператор **break** осуществляет выход за пределы цикла, и его действие прекращается. Команда **glob** работает аналогично команде **echo**, но после вывода курсор остается в той же строке, а не в начале следующей.

Оператор выбора **switch** имеет вид:

```
switch(входная_строка)
        case образец:
                ...
                breaksw
        ...
        default:
                ...
                ...
endsw
```

Образцы вариантов **case** последовательно сравниваются с указанной в **switch** входной строкой (в образцах можно использовать шаблоны имен файлов *, ? и [...]). Если в варианте **case** выявлено совпадение образца и входной строки, выполняются все строки до ближайшего **breaksw**, **default** или **endsw**. Если совпадение не обнаружено, выполнение продолжается после **default**. Если **default** отсутствует, выполнение продолжается после **endsw**. Слова **case** и **default** должны быть первыми в строке. Выполнение оператора **breaksw** приводит к тому, что управление передается на первую строку после **endsw**

```
% set j = 0
% foreach i ( aaa bbb ccc ddd eee )
?       @ j++
?       switch( $i )
?               case "aaa":
?                       glob "$i "
?                       breaksw
?               case "bbb":
?                       glob "$i "
?                       breaksw
?               case "ccc":
?                       glob "$i "
?                       breaksw
```



```
?               case "ddd":
?                       glob "$i "
?                       breaksw
?               default:
?                       breaksw
?
?       endsw
?       glob "$j "
?end
aaa 1 bbb 2 ccc 3 ddd 4 5 %
```

Переменной цикла i присваиваются значения из списка в предложении **foreach**. Внутри цикла работает **switch**. Если ни одно значение вариантов **case** не совпадает со значением переменной i, то выполняется вариант **default**. В данном случае это приводит к выходу за пределы переключателя **switch**, поэтому выводится порядковый номер итерации цикла и **foreach** выполняет следующую итерацию. В новой итерации цикла снова начинает действовать **switch**.

Оператор **if**

```
if(выр1) then
    ...
else if(выр2) then
    ...
else
    ...
endif
```

Если значение выр1 истинно (отлично от нуля), выполняются команды до первого **else**. Иначе, если значение выр2 истинно, выполняются команды до второго **else** и т.д. Возможно любое количество пар **else if**, **endif**,нужно только одно. Часть **else** необязательна. Слова **else** и **endif** должны быть первыми в строках, где они указаны. Оператор **if** должен находиться один в строке или после **else**

```
% foreach i ( a b c d e f )
?       if( "$i" == "a" ) then
?               glob "a "
?       else if( "$i" == "b" ) then
?               glob "b "
?       else if( "$i" == "c" ) then
?               glob "c "
```

```
?          endif
? end
a b c %
```

На каждой итерации цикла **foreach** осуществляется проверка текущего значения переменной цикла i с символами **a**, **b** и **c**. Если логическое условие проверки выполняется, то на экран выводится соответствующий символ, иначе осуществляется следующая итерация цикла.

Оператор цикла **while**

```
while(выражение)
        ...
end
```

Цикл выполняется, пока истинно значение выражения. Ключевые слова **while** и **end** должны находиться на отдельных строках. В цикле можно использовать команду **break** для выхода из цикла и команду **continue** для возобновления следующей итерации цикла без завершения текущей (все операторы цикла, следующие за командой **continue**, не будут выполняться)

```
% set argv = ( 1 2 3 1 2 3 0 1 )
% while( $#argv > 0 )
?          if( "$argv[1]" == "3" ) then
?                    shift
?                    continue
?          endif
?          if( "$argv[1]" == "0" ) then
?                    break
?          endif
?          glob " $argv[1]"
?          shift
? end
1 2 1 2 %
```

Здесь выполняется цикл, в котором выводятся значения **argv**. Если значение **argv[1]** есть символ "**3**", то оно не выводится и идет переход к следующей итерации цикла, если символ "**0**", то действие цикла прекращается. Оператор **shift** освобождает (выталкивает) вершину стека для следующего элемента. В данном случае под стеком понимается список слов массива **argv**. После каждого сдвига **argv[1]** приобретает значение следующего слова. Цикл **while** прекращает

работу, когда список слов массива **argv** станет пустым, т.е. **$#argv** станет равным нулю.

2.10. Командные файлы

Программы, написанные на языке **C-shell**, называют командными файлами. Каждая строка командного файла интерпретируется **csh**, в ней осуществляются подстановки, вычисляются, если необходимо, выражения. Командный файл может запускать на выполнение другие выполняемые файлы, в том числе командные файлы, написанные для интерпретаторов **csh** и **sh**. Кроме того, в командном файле доступна для выполнения любая команда системы.

Каждый командный файл на языке **C-shell** должен начинаться символом **#** в первой позиции первой строки, далее указывается системное имя интерпретатора. Допускаются пустые строки. Строка, начинающаяся **#**, является строкой комментариев. При создании командных файлов, используя переменную **argv**, можно организовать ввод и обработку аргументов командной строки, которые могут быть как именами файлов, так и любыми последовательностями символов. В качестве примера рассмотрим программу диалогового ввода содержания документа. Комментарии в программе достаточно полно раскрывают алгоритм работы

```
#!/bin/csh
# Программа в режиме  меню запрашивает
# сведения о загрузке ЭВМ.  Результат
# ввода дописывается в файл Result.
# Исправление ошибок ввода выполняется
# повторным вводом. Запись в  файл
# происходит по  команде Запомнить.
# Переменная  out  используется для
# указания направления вывода.  Когда
# out = /dev/tty, вывод дописывается на
# экран  дисплея,  когда  out = Result
# вывод дописывается в файл ./Result.

set ЭВМ НОМ ОРГ ПОД
set out = /dev/tty
set мес = ( 0 0 0 0 0 0 0 0 0 0 0 0 )
set сум = ( 0 0 0 0 0 )

# Имена месяцев для подтверждения
# режима ввода
```

```
set имя = ( Январь   Февраль Март     \
            Апрель   Май     Июнь     \
            Июль     Август  Сентябрь \
            Октябрь  Ноябрь  Декабрь  )

# Работа программы выполняется в
# бесконечном цикле while(1)

while ( 1 )
        glob ' Укажите режим работы > '
        set ответ = $<
        switch( "$ответ" )

                case '[КкKk]':
                        exit( 0 )

                case '[ЗзZz]':
                        set out = Result

                case '[ДдDd]':

# Вычисление показателей по кварталам
# и за год

@ сум[1] = $мес[1] + $мес[2] + $мес[3]
@ сум[2] = $мес[4] + $мес[5] + $мес[6]
@ сум[3] = $мес[7] + $мес[8] + $мес[9]
@ сум[4] = $мес[10] + $мес[11] + $мес[12]
@ сум[5] = $сум[1] + $сум[2] + $сум[3] + $сум[4]

# Очищать экран, если вывод не в файл

                if( "$out" != "Result" ) clear

# Команда echo выводит в файл out
# значения переменных

        echo "                  \
          Отчет о загрузке ЭВМ.\
              ЭВМ  $ЭВМ     \
  Заводской номер  $НОМ     \
     Организация   $ОРГ     \
```



```
   Подразделение   $ПОД     \
          Январь   $мес[1]  \
         Февраль   $мес[2]  \
           Март    $мес[3]  \
 Первый квартал    $сум[1]  \
         Апрель    $мес[4]  \
             Май   $мес[5]  \
           Июнь    $мес[6]  \
 Второй квартал    $сум[2]  \
            Июль   $мес[7]  \
          Август   $мес[8]  \
       Сентябрь    $мес[9]  \
 Третий квартал    $сум[3]  \
         Октябрь   $мес[10] \
          Ноябрь   $мес[11] \
         Декабрь   $мес[12] \
Четвертый квартал  $сум[4]  \
           Итого   $сум[5]  " >> $out
                continue

        case '[АаAa]':
                glob 'Тип ЭВМ: '
                set ЭВМ = $<
                continue

case '[БбBb]':
        glob 'Заводской номер ЭВМ: '
        set НОМ = $<
        continue

case '[ВвWw]':
        glob 'Организация: '
        set ОРГ = $<
        continue

case '[ГгGg]':
        glob 'Подразделение: '
        set ПОД = $<
        continue

case '[1-9]':
case "1[012]":
```

```
                glob $имя[$ответ]' : '
                set  мес[$ответ] = $<
                continue

        default:
# Вывод меню, если режим указан неправильно.
                echo 'Такого режима нет.'
                clear
        echo '                                  \
                                                \
                                                \
                                                \
                                                \
                                                \
                Режимы работы:                  \
                                                \
а  Ввод наименования ЭВМ.                       \
б  Ввод заводского номера ЭВМ.                  \
в  Ввод наименования организации.               \
г  Ввод наименования подразделения.             \
д  Вывод данных на экран.                       \
з  Запомнить.                                   \
к  Конец работы.                                \
   Вывод загрузки в часах по месяцам:           \
 1  Январь    2  Февраль    3  Март             \
 4  Апрель    5  Май        6  Июнь             \
 7  Июль      8  Август     9  Сентябрь         \
10  Октябрь  11  Ноябрь    12  Декабрь          \
                                                \

        endsw
end
```

Ниже показано содержимое файла **Result**, который формирует программа. Эта же информация выводится на экран, если указан режим Вывод данных

```
        Отчет о загрузке ЭВМ.
            ЭВМ  СМ 1420
Заводской номер  1673
   Организация   Поликлиника 124
 Подразделение   ИВЦ
       Январь    300
```



```
       Февраль    350
         Март     350
Первый квартал   1000
        Апрель    520
          Май     330
         Июнь     700
Второй квартал   1550
         Июль     200
       Август     150
     Сентябрь     250
Третий квартал   600
      Октябрь    300
       Ноябрь    310
      Декабрь    280
Четвертый квартал 890
        Итого    4040
```

Часто возникает необходимость принять с клавиатуры ответ в виде метасимвола, который не должен интерпретироваться. Этого можно достигнуть отменой специального значения символа. В отдельных случаях такая отмена затруднительна. Ниже приведен пример программы, в которой со стандартного ввода читаются метасимволы шаблонов имен файлов, но их специальное значение игнорируется.

```
# !/bin/csh
# программа демонстрирует  способ чтения
# со стандартного ввода символов  ? и *,
# которые обычно рассматриваются  как
# шаблоны имен файлов и интерпретируются
#
while( 1 )
        glob '=> '
        set ответ = "$<"

        switch( "$ответ" )

                case [?] :
                        echo 'Вопросительный знак'
                        breaksw

                case [*] :
                        echo 'Звездочка'
```

```
                breaksw

        case '{' :
        case '}' :
                echo 'Фигурная скобка'
                breaksw

        default :
                echo 'Другой символ'

    endsw
end
```

2.11. Протоколирование, средства работы с протоколом

Интерпретатор записывает во временный файл протокол работы пользователя в виде списка выполненных командных строк. Количество запоминаемых командных строк определяется переменной **history**, которая обычно определяется в файле **~/.cshrc** (о нем будет сказано ниже). Если установлена предопределенная переменная **savehist**, то по завершении сеанса работы пользователя указанное количество строк **history** будет сохранено в файле **~/.history**. При следующем входе в систему содержимое этого файла используется для занесения в протокол. Например, если выполнена команда

```
set savehist = 22
```

то последние **22** строки протокола будут сохранены в файле **~/.history** и восстановлены в начале следующего сеанса работы. Чтобы напечатать протокол, необходимо выполнить команду **history**. Каждая строка протокола имеет номер, что позволяет запустить ее на выполнение, указывая, например, соответствующий номер. Восклицательный знак в командной строке служит указанием для интерпретатора, что все указанное далее до разделителя относится к некоторой строке протокола. Восклицательный знак можно указать в начале или в любом другом месте командной строки. Пусть после выполнения команды **history** на экран дисплея выведен следующий протокол:

```
1  cat file1
2  pr -w39 -l24 -2 file1
3  cc program.c >& errors &
```



```
4  cat errors
5  ed program.c
6  history
```

тогда, используя восклицательный знак, можно выполнить ряд действий:

!2

выполнится вторая строка протокола;

!!

выполнится последняя строка протокола;

!-2

выполнится четвертая строка (вторая от последней);

!cat или **!c**

выполнится четвертая строка. Интерпретатор просматривает строки протокола снизу и выполняет первую, в которой найдена последовательность символов (**cat** или **c**), стоящая в начале строки;

!{cat}.a1

выполнится команда **cat errors.a1** - к найденной строке дописывается **.a1**;

!?gram?

выполнится пятая строка протокола. Интерпретатор выберет для выполнения эту строку, так как в ней будет найден шаблон **gram**. Здесь символы **?** выделяют шаблон, по которому осуществляется поиск;

cat !5* !1*

выполнится команда **cat program.c file1** - будут подставлены слова пятой и первой строк протокола, исключая первые слова этих строк.

Можно выбирать отдельные слова в строках протокола для включения их в командную строку. Слова командной строки нумеруются, начиная с **0**. Слово с номером **0** - обычно имя команды. Слово можно выделить с помощью определителя, перед которым необходимо указать символ двоеточие, например

```
cat !3:1
```

Из третьей командной строки протокола будет выбрано слово с номером **1**, получим

```
cat program.c
```

Рассмотрим подробнее определители слов в командных строках протокола:

***** или **n*** или **n-m**
выбрать все слова, начиная со слова с номером **1**, или выбрать все слова, начиная со слова с номером **n**, или выбрать все слова, начиная со слова с номером **n** и кончая словом с номером **m**;
n или **n-**, или **-n**
выбрать слово с номером **n**, или выбрать все слова, начиная со слова с номером **n**, не включая последнее, или выбрать все слова, начиная со слова с номером **0** до слова с номером **n**;
?шаблон?:%
выбрать слово, соответствующее шаблону;
^ или **$**
слово с номером **1** или последнее слово командной строки.

Разрешается не указывать двоеточие перед следующими определителями:

^ $ * - %

Рассмотрим пример. Пусть после выполнения команды **history** на экран дисплея выведен протокол:

```
1  cat file1 file2 file3
2  pr -w39 -l24 -2 file1 file5
3  cc -o program1.c program2.c >& errors &
4  cat errors
5  ed program2.c
6  history
```

тогда интерпретатор **csh** выполнит команды, осуществляя подстановки следующим образом:

!5:0 !1:3
из пятой строки выбирается слово с номером **0** (имя команды), из первой строки выбирается слово с номером **3**. Выполнится команда **ed file3**;

!5:0 !1$
из пятой строки выбирается слово с номером **0**, из первой строки - последнее слово. Выполнится команда **ed file3**;

!2:-3 !3:2-3
из строки **2** выбираются слова с номерами от **0** до **3** включительно, из строки **3** выбираются слова с номерами **2** и **3**. Выполнится команда:


```
pr -w39 -l24 -2 program1.c program2.c
```

!2-3 !?prog?%
выполнится команда:

```
pr -w39 -l24 -2 program2.c
```

После необязательного определителя могут указываться модификаторы слов, которые позволяют выполнить ряд преобразований над словом, и оно подставляется в командную строку модифицированным. Модификаторы переменных были рассмотрены выше. В действиях со строками протокола можно использовать и другие модификаторы:

p
распечатать новую команду, но не выполнять ее;

&
повторить предыдущую подстановку;

s/образец_1/образец_2/
заменить образец_1 на образец_2. Символ **/** можно заменить на любой, отсутствующий в образцах. Если образец_2 пустая строка, то образец_1 удаляется.

Перед каждым модификатором необходимо ставить двоеточие. Если имеется определитель слова, то модификатор должен следовать за ним. Пусть после выполнения команды **history** на экран дисплея выведено:

```
1  cat /usarc/gruppa/ivanov/file1.c
2  pr  /usarc/gruppa/ivanov/file1.c
3  cc pa1.c pa2.c pa3.c pa4.c >& errors &
4  cat errors
5  ed program.c
6  history
```

тогда интерпретатор выполнит команды, осуществляя подстановки и модификации, следующим образом:

!1:0 !1^:t:r
выбирает из строки с номером **1** слово с номером **0**, т.е. имя команды, в данном случае **cat**. Далее выбирает из первой строки слово с номером **1**, в данном случае это **/usarc/gruppa/ivanov/file1.c**. Модификатор **t** удалит из этого слова имена каталогов, в данном случае удаляется **/usarc/gruppa/ivanov**, и слово теперь будет именем файла **file1.c**.

Модификатор **r** удалит расширение имени файла. Таким образом, выполнится команда **cat file1**.

!1:0 !1^:h/document

по определителю ^ будет выбрано первое слово первой строки, модификатор **h** удалит из него имя файла, оставив имена каталогов, ведущих к нему, и выполнится команда

```
cat /usarc/gruppa/ivanov/document
```

!1:0 !1^:h:s?ivanov?sidorov?/document

из первого слова первой строки выбираются имена каталогов, ведущих к файлу, затем модификатор **s** заменит **ivanov** на **sidorov** и выполнится команда

```
cat /usarc/gruppa/sidorov/document
```

!1:0 !1^:h:s?ivanov?sidorov?/doc !1^:&:p

первые два слова командной строки действуют аналогично предыдущему примеру. Третье слово выбирает из 1 строки протокола слово с номером 1, в нем осуществляется замена (модификатор **&**), аналогичная предыдущей, т.е. выполняется замена **?ivanov?sidorov?**, сохранив все остальное в этом слове. Строка не выполняется, а только выводится на экран (модификатор **p**):

```
cat /usarc/gruppa/sidorov/doc \
/usarc/gruppa/sidorov/file1.c
```

!1:0 !3:1-4:gs?pa?ff?:p

имя команды выбирается из первой строки протокола, из **3** строки выбираются все слова с номерами от **1** до **4** включительно и в них глобально (модификатор **g**) делается замена **?pa?ff?** . Команда будет напечатана, но выполняться не будет (модификатор **p**):

```
cat ff1.c ff2.c ff3.c ff4.c
```

Существует также удобное средство редактирования последней строки протокола. Для этих целей используется конструкция ^шаблон^замена^. Допустим, последняя строка имеет вид **cat aaa bbb ccc ddd**, тогда после команды ^ccc^file.c^ будет выполнена замена: **cat aaa file.c ccc ddd**.

Имеется возможность ввести краткие обозначения для командных строк. Эти краткие обозначения называют псевдонимами команд. Если для какой-либо командной строки установлен псевдоним, то ее выполнение

теперь можно осуществлять, указывая псевдоним, а не всю строку. Допустим, имеется командная строка

```
alias sp "sort \!* | print"
```

тогда командные строки

```
sort file1 file2 | print
```

и

```
sp file1 file2
```

тождественны. Вместо **!*** в командную строку будут подставлены имена файлов или ключи команды **sort**, указанные за псевдонимом **sp**.

Интерпретатор команд ведет список псевдонимов, которые могут устанавливаться, отображаться и модифицироваться с помощью команд **alias** и **unalias**. Командная строка после просмотра разбивается на отдельные слова, каждое слово, интерпретируемое как имя команды, проверяется, чтобы выяснить, имеет ли оно псевдоним. Если да, это слово заменяется на значение псевдонима. Всюду в командных строках при создании псевдонимов символ **!** необходимо экранировать. В противном случае он будет интерпретироваться как обращение к протоколу. Псевдонимы можно устанавливать и на команды **csh**, например

```
alias a alias
```

устанавливает псевдоним на команду **alias**.

2.12. Переменные интерпретатора csh

Интерпретатор **csh** оперирует переменными двух видов: внутренними и внешними. Внутренние переменные устанавливают режим работы интерпретатора, а внешние в основном относятся к командным строкам, которые им интерпретируются. Обращение к переменным может быть простым (установлена переменная или нет) и сложным. Например, переменная **argv** представляет образ списка параметров командной строки, а переменная **verbose** является ключом и существенно лишь ее наличие или отсутствие. Особое место занимают так называемые переменные окружения. Интерпретатор считывает их значения при запуске. Значения переменных окружения становятся внутренними константами интерпретатора и их можно использовать как константы в командных строках и командных файлах. Каждая

внутренняя переменная имеет определенный смысл для интерпретатора. Часть внутренних переменных всегда устанавливается интерпретатором при инициализации либо при запуске-завершении процессов. После чего переменные не будут модифицироваться, если этого не сделает пользователь. К числу внутренних переменных относятся: **argv**, **cdpath**, **cwd**, **home**, **path**, **prompt**, **shell**. Переменные **child** и **status** устанавливаются при порождении процессов и сохраняют свое значение до появления новых. Значения переменных устанавливаются командой **set** или ключом при вызове **csh**. Исключить переменную из числа определенных можно командой **unset**. Ниже приводится список внутренних переменных и их назначение.

argv

представляет строку параметров. К ней применимы подстановки позиционных параметров.

cdpath

этой переменной присваивается список имен каталогов, к которым пользователь часто будет обращаться. Допустим, **cdpath** определена следующим образом: **set cdpath = (/usr/include /usr/lib)**, тогда команда **chdir sys** тождественна команде **chdir /usr/include/sys**.

checktime

если эта переменная установлена и если в течение указанного времени не выполнялось каких-либо действий, то выполняется **exit**. Допустим, выполнена команда **set checktime = 3**, тогда, если в течение **3** мин не выполнялись какие-либо действия, выполняется **exit**, и интерпретатор прекращает работу.

child

номер процесса. Выводится на экран дисплея при запуске параллельного процесса. Значение переменной **child** сбрасывается, когда этот процесс завершается.

cwd

значением этой переменной является строка - полное имя рабочего каталога. Это имя может не совпадать с истинным, если установлен **symlink**.

echo

вызывает печать каждой команды перед выполнением. Все подстановки выполняются перед выводом. Режим **echo** можно установить либо на все время работы, либо на период выполнения одного командного файла. Например, **csh -x comfile** установит режим **echo** на время выполнения командного файла **comfile**, а команда **set echo** - на все время работы интерпретатора.

history

численное значение этой переменной устанавливает количество строк, которое необходимо хранить в протоколе. Для слишком

большого числа строк может не хватить памяти. Оптимальное число - **22** строки.

home

регистрационный каталог пользователя. Его имя считывается при запуске **csh** из переменной окружения **HOME**.

ignoreeof

предотвращает случайное завершение работы интерпретатора по признаку "конец файла". Этот признак выглядит как **CУ/D** или **CNTRL/D** при вводе с клавиатуры дисплея. Признак конца файла можно заменить на другой командой системы **stty(1)**.

mail

имя файла, в который будет поступать почта.

noclobber

устанавливает защиту файлов от случайного разрушения.

noglob

запретить расширение имен файлов.

nonomatch

обычно, если указанному шаблону (например, **echo *.[2-5]**) не соответствует ни один образец, порождается состояние ошибки. Если установлена переменная **nonomatch**, состояние ошибки не возникает, а указанный шаблон возвращается программе.

notify

асинхронно выводить сообщения о состояниях выполняемых процессов. Если переменная **notify** не установлена, эти сообщения выводятся перед выводом нового приглашения.

path

определяет имена каталогов, в которых интерпретатор будет искать файлы команд, запускаемых на выполнение. При запуске интерпретатора создается хеш-таблица команд из каталогов, указанных в **path**. Хеширование существенно сокращает время поиска команды при ее запуске. Если после входа в систему, т.е. после хеширования каталогов, будет записана в один из них новая команда, то она будет отсутствовать в хеш-таблицах, и интерпретатор не будет ее обнаруживать. Для разрешения этой ситуации необходимо выполнить команду **rehash**. По команде **rehash** будут перестроены хеш-таблицы, и новые команды будут доступны. При запуске нового интерпретатора снова читается файл **~/.cshrc** и строится хеш-таблица. На эту операцию уходит достаточно много времени и, если есть необходимость более быстрого старта, при запуске необходимо использовать ключ **-f**

csh -f comfile

Перестройка хеш-таблиц осуществляется также всякий раз, когда с помощью команды **set** изменяется значение переменной **path**.

prompt

содержит строку символов, которая выводится в качестве приглашения. Если эта строка символов включает восклицательный знак, на его место подставляется текущий номер командной строки. Если переменная **prompt** не установлена, печатается приглашение **%** для рядового пользователя и **#** для суперпользователя.

shell

содержит имя интерпретатора, который запускается при входе пользователя в систему. Имя интерпретатора указано в переменной среды **SHELL** и считывается в начале сеанса.

status

принимает значение кода завершения команды, например:

```
% false ; echo $status
1
% true ; echo $status
0
```

Здесь команда **false** возвращает **1** - код неудачного завершения, команда **true** возвращает **0** - код удачного завершения.

time

хронометрирует выполнение командных строк. Если выполнение продолжалось дольше указанного времени, выводятся результаты хронометрирования. Например, в файле **~/.cshrc** выполнено назначение **set time = 6**, это значит, что интерпретатор будет выводить результаты хронометрирования, когда время выполнения командной строки превысит **6** с. Если теперь выполнить команду, например, **sort file**, то после ее завершения будет выведен результат хронометрирования:

```
1.6u 17.9s 0:26 74%
```

Здесь: **1.6u** - время пользовательской фазы процесса; **17.9s** - время системной фазы процесса; **0:26** - астрономическое время процесса; **74%** - отношение в процентах суммы пользовательской и системной фаз процесса к астрономическому времени.

verbose

устанавливает режим распечатки слов каждой команды с учетом подстановок. Этот режим можно установить, используя ключ **-v** при запуске интерпретатора на выполнение.

При запуске командных файлов можно устанавливать различные ключи. При этом командная строка выглядит следующим образом:

csh -список_ключей имя_файла ...

Если имя_файла не указано, то порождается новая интерактивная оболочка.

Ниже перечислены ключи интерпретатора и их значения:

-c считать команду из единственного параметра, указанного сразу после **-c**;

-e интерпретатор прекращает работу, если любая вызванная команда завершается ненормально (код возврата не **0**);

-f запретить чтение файла **~/.cshrc** для более быстрого старта;

-i запустить новую оболочку как интерактивную. Если вызов интерпретатора осуществляется с клавиатуры дисплея, этот ключ устанавливается по умолчанию;

-n осуществлять разбор командных строк, но не выполнять команды. Это режим отладки;

-s читать из стандартного ввода;

-t считывать и выполнять только одну строку. Эта строка может содержать в конце символ продолжения строки ****;

-v после подстановок из протокола распечатать команду перед ее выполнением;

-V перед разбором строк из файлов **~/.cshrc** и **~/.login** установить ключ **-v**. Это позволит увидеть на экране дисплея, как интерпретатор устанавливает назначения и выполняет командные строки при интерпретации этих файлов;

-x печатать на экране дисплея все команды перед выполнением;

-X установить ключ **-x** при интерпретации файлов **~/.cshrc** и **~./login**.

Внешние переменные - это такие переменные, которые устанавливаются и используются только пользователем. В отличие от внутренних переменных и переменных окружения внешние переменные имеют тот смысл, который придается им пользователем. Значения внешних переменных могут быть установлены и отображены командой **set** и отменены командой **unset**.

Система поддерживает массив переменных, который называют переменными среды или окружения. Переменные окружения используются системными и пользовательскими программами. Для установки значений переменных окружения используется команда **setenv**, для отмены - команда **unsetenv**. Имеется несколько стандартных имен переменных окружения, их значения зависят от соответствующих назначений командой **setenv**. Часть этих назначений происходит при открытии сеанса работы. Пользователь

имеет возможность переустановить значения существующим переменным, объявить и присвоить значения новым. Значения стандартных переменных окружения используются многими системными программами. Они доступны как константы и программам пользователя. Важно отметить, что значения переменных окружения являются внутренними константами интерпретатора. Перечисленные ниже имена зарезервированы как стандартные имена переменных окружения:

PATH

имена стандартных каталогов, разделенных двоеточием;

HOME

регистрационный каталог пользователя, установленный в файле **/etc/passwd**;

TERM

имя типа терминала;

TERMCAP

строка определения возможностей дисплея из файла **/etc/termcap**;

SHELL

имя интерпретатора командного языка, который инициализируется при входе пользователя в систему;

MSG

определяет, на каком языке будут выводиться сообщения пользователю при работе с системой (**MSG = r** - на русском, **MSG = l** - на английском);

USER

регистрационное имя пользователя.

Команда **unsetenv** удаляет добавленные во время работы переменные окружения. Команда **setenv** позволяет объявить и присвоить значение новой переменной окружения

```
% setenv NAME 15
% echo $NAME
15
% @ a = ( $NAME + 15 )
% echo $a
30
% setenv NAME "Jun Feb Mar Apr"
% echo $NAME
Jun Feb Mar Apr
```

Переменные окружения отличаются от переменных интерпретатора тем, что они не влияют на работу интерпретатора.

2.13. Специальные файлы

В регистрационном каталоге пользователя размещается несколько специальных файлов: **~/.hushlogin**, **~/.login**, **~/.cshrc**, **~/.logout** и **~/.history**.

Файл **~/.hushlogin** пустой и используется как ключ. Если он существует, при открытии сеанса работы не выводится на экран дисплея **/etc/motd** - файл с текстом сообщения администратора. Обычно в файле **/etc/motd** содержаться сведения о версии системы, "вывеска" организации и т.д.

При открытии сеанса работы интерпретатор читает файлы **~/.login** и **~/.cshrc**, а при завершении работы - файл **~/.logout**. При входе пользователя в систему первым читается файл **~/.cshrc**, потом **~/.login**. Если в регистрационном каталоге имеется файл **~/.history**, то он считывается в протокол. Все перечисленные файлы, кроме **~/.history**, являются обычными командными файлами, в которых программист определяет желательные для себя действия по входу и выходу из системы.

Файл **~/.login** определяет те действия, которые необходимо выполнить в начале сеанса работы пользователя. Ниже приведен пример такого файла:

```
set ignoreeof
set prompt = 'Иванов И.И._\!> '
if( $?MAIL ) set mail = $MAIL
msgs
setenv MSG r
```

В первой строке устанавливается ключ **ignoreeof**, который предотвращает случайное завершение работы интерпретатора при наборе на клавиатуре дисплея символа СУ/D (CNTRL/D).

Во второй строке устанавливается приглашение, которое будет выдаваться при готовности принять новую командную строку. Здесь вместо **\!** будут подставляться текущие номера строк **history**, например :

```
Иванов И.И._15>
```

В третьей строке указывается имя почтового файла. Если он будет не пуст, то на экране дисплея появится сообщение: "У Вас есть новая почта ". Эти определения существенно зависят от версии программы **mail**, установленной в системе.

В четвертой строке записано обращение к команде **msgs**, которая выдает новые информационные сообщения при входе в систему (эти

сообщения заносятся администратором, сопровождающим операционную систему).

В пятой строке определяется переменная **MSG**, которая определяет язык диагностик (в данном случае - русский, для английского указывется буква 1).

При завершении сеанса работы читается файл **~/.logout** и выполняются указанные в нем действия. Список таких действий зависит исключительно от фантазии программиста. В файле **~/.logout** можно разместить все команды, которые необходимо выполнить по выходу из системы. Важно отметить, что, несмотря на выход пользователя из системы, все процессы, запущенные им как асинхронные, будут продолжать выполняться.

При каждом вызове **csh** выполняет файл **~/.cshrc**. Ниже приведен пример файла **~/.cshrc**:

```
set path = ( . /bin /usr/bin /usr/ucb )
set history = 22
set savehist = 22
set checktime = 3
set prompt = 'ИВАНОВ И.И._\!> '
alias  h  history
alias  c /bin/cat
alias  l /bin/ls -l
```

set path = (. /bin /usr/bin /usr/ucb)

устанавливает те каталоги, где **csh** будет искать команды перед запуском их на выполнение.

set history = 22

устанавливает количество последних командных строк, которые должны сохраняться в протоколе.

set savehist = 22

устанавливает количество строк протокола, которое необходимо запомнить в файле **~/.history**. При входе в систему этот файл будет прочитан и записан в протокол.

set checktime = 3

устанавливает время (3 мин), в течение которого **csh** может "бездействовать". Если в течение указанного времени не будут выполняться какие-либо действия, то выполняется команда **exit** и **csh** прекращает работу. Такое завершение работы полезно, когда пользователю необходимо отлучиться.

set prompt = 'ИВАНОВ И.И._\!>'

устанавливает приглашение, которое будет выводить **csh** при порождении нового экземпляра интерпретатора. Символы этого

приглашения специально набраны заглавными буквами, чтобы у пользователя была возможность отличить основной экземпляр интерпретатора от порожденного нового.

Остальные команды демонстрируют возможность использования псевдонимов команд. Заметим, что файл **~/.login** выполняется только один раз - в начале сеанса работы в системе; файл **~/.cshrc** выполняется при запуске каждой новой оболочки.

2.14. Встроенные команды и операторы

Ниже перечислены имена встроенных команд, их синтаксис и действие.

alias
alias имя
alias имя список_слов

команда **alias** позволяет устанавливать псевдонимы команд и командных строк. При обращении к командам или командным строкам, для которых выполнена команда **alias**, можно использовать их псевдонимы. Первая форма распечатывает все псевдонимы, вторая - псевдоним для указанного имени, если он установлен, третья устанавливает псевдоним для заданного списка слов. В списке слов выполняются подстановки.

alloc.

показывает размер используемой динамической памяти, включая используемую и свободную память, а также адрес последней ячейки памяти. Используется для отладки интерпретатора.

bg
bg %внутренний_идентификатор_процесса
bg %шаблон_имени_процесса

переводит последний приостановленный или указанный процесс в асинхронный режим выполнения.

break

вызывает выход за **end** ближайшей внешней конструкции **foreach** или **while**. Можно выполнять переходы через несколько уровней посредством написания нескольких операторов **break** в одной строке, разделяя их символом ";".

breaksw

вызывает выход из оператора **switch** за пределы оператора **endsw**;

case метка:

метка - шаблон одного из вариантов оператора **switch**. В метках можно использовать шаблоны имен файлов и каталогов (*, ?, [...]). Двоеточие обязательно.

cd
cd имя
chdir
chdir имя

заменяет рабочий каталог на каталог имя. Если параметр отсутствует, осуществляется переход к регистрационному каталогу пользователя.

continue

продолжает выполнение ближайшего внешнего **while** или **foreach**. Все строки цикла до **end** пропускаются, и начинается новая итерация цикла.

default:

используется в **switch**. Если после всех проверок не нашлось варианта, совпавшего с вариантом в **case**, выполняется то, что указано в варианте **default**. Вариант **default** должен располагаться после всех **case**. Двоеточие обязательно.

dirs

напечатать содержимое стека имен каталогов. Занесение имен каталогов в стек выполняет команда **pushd**, удаление имен каталогов из стека выполняется командой **popd**. Элементы стека нумеруются от 1, начиная от вершины стека.

echo список_слов
echo -n список_слов

список слов записывается в стандартный вывод. Ключ **-n** запрещает переход на новую строку после вывода.

else
end
endif
endsw

см. ниже описание операторов **foreach**, **if**, **switch** и **while**.

eval арг ...

сначала производятся все подстановки, затем слово **eval** отбрасывается, и оставшиеся символы рассматриваются как командная строка, которая выполняется.

exec команда

команда запускается вместо интерпретатора. Работа интерпретатора завершается.

exit
exit(выражение)

происходит выход из интерпретатора (первая форма) либо со значением указанного выражения (вторая форма). Значение переменной **status** всегда равно коду возврата.

fg
fg %внутренний_идентификатор_процесса
fg %шаблон_имени_процесса

возвращает последний приостановленный (первая форма) или указанный процесс в синхронный режим выполнения.

foreach имя (список_слов)
...
end

переменной имя последовательно присваиваются значения каждого члена списка слов и выполняется последовательность команд между данной командой и соответствующим оператором **end** (**foreach** и **end** должны находиться в отдельных строках).

glob список_слов

аналогична команде **echo**. Слова выводятся без пробела и после последнего слова не выполняется переход на новую строку. Такой вывод полезен при манипуляциях с именами файлов, когда эти имена необходимо удлинять или создавать новые.

goto слово

оператор безусловного перехода на метку. Выполнение продолжается со строки, идущей после указанной метки. Метка должна завершаться символом :. Слово в операторе **goto** может быть строкой, содержащей команды, псевдонимы команд и расширения имен файлов. В этом случае метка формируется как результат интерпретации и выполнения этой строки.

hashstat

распечатывает строку статистики, определяющую, насколько эффективно внутренняя таблица хеширования размещает команды. Данная команда является отладочной.

history
history -r
history n
history -r n
history -h

выводит списки из протокола. По ключу **-r** строки выводятся в обратном порядке. Если указано число **n**, то выводятся только **n** строк протокола. С ключом **-h** выводятся строки протокола в формате файла **~/.history**.

if(выражение) команда

если выражение истинно (не равно нулю), то выполняется команда. Команда должна быть одна. Не допускается применение последовательности команд и/или конвейера. Интерпретатор вычисляет истинность выражения после подстановок как в команде, так и в выражении.

if(выражение_1) **then**
...
else if(выражение_2) **then**
...
else
...
endif

если значение выражения_1 истинно, выполняются команды до первого **else**. Иначе, если значение выражения_2 истинно, выполняются команды до второго **else** и т.д. Возможно любое количество пар **else-if**; **endif** нужен только один. Часть **else** является необязательной. Слова **else** и **endif** должны быть первыми в строках, где они указаны. **if** должен находиться один в строке или после **else**.

jobs
jobs -l

вывести таблицу процессов. Формат **jobs -l** выводит более полную

информацию. Интерпретатор обеспечивает работу с внутренними и системными идентификаторами процессов. Системные идентификаторы процессов выводятся командой **ps** или **jobs** с ключом **-l**, внутренние - командой **jobs**. Пользователю предоставляется возможность не обращаться к системным идентификаторам, а указывать в командах управления процессами внутренние идентификаторы. Внутренний идентификатор процесса печатается в квадратных скобках. Символом "+" помечается последний из приостановленных процессов; символом "-" предпоследний из приостановленных процессов.

kill %внутренний_идентификатор_процесса ...
kill -SIG %внутренний_идентификатор_процесса ...
kill %шаблон_имени_процесса ...
kill -SIG %шаблон_имени_процесса ...
kill системный_идентификатор_процесса ...
kill -SIG системный_идентификатор_процесса ...
kill -l

если **SIG** не указан, то процессам посылается сигнал **TERM** (окончить). Сигнал **SIG** указывается либо числом, либо именем без шаблона **SIG** (как это определено в **/usr/include/signal.h**). **Kill -l** выводит список имен сигналов.

limit тип_ресурса максимальный_размер
limit тип_ресурса
limit

для **32**-разрядных машин существует возможность ограничения ресурсов системы, выделяемых одному процессу. Ограничивается потребление текущим процессом и каждым, который создается, так, что ни один из этих процессов отдельно не превышает максимальный размер заданного ресурса. Если значение максимальный размер не задано, выводится значение текущего ограничения; если значение тип_ресурса не задано, выводятся все установленные ограничения. Текущее управление ресурсов включает время процессора **cputime** (максимальное число секунд центрального процессора, отводимое каждому процессу), размер файла **filesize** (максимальный размер одного файла, который можно создать), размер сегмента данных **datasize** (максимальное увеличение области данные_+_стек с помощью **sbrk(2)** за границу текста программы), размер стека **stacksize** (максимальный размер автоматически расширяемой области стека),

размер дампа **coredumpsize** (максимальный размер дампа, который будет создан). Максимальный размер ресурса может быть задан в виде числа (целого либо с точкой), за которым следует указатель размерности. Для задания имен типов ресурсов и указателей размерности достаточно задавать уникальные шаблоны имен.

login
login имя

по команде **login** завершается сеанс работы пользователя. Команду **login** можно выполнять без аргумента "имя" и с ним. Во втором случае сразу же будет запрошен входной пароль пользователя "имя".

logout

указывает интерпретатору о необходимости прекратить работу. Читается файл **~/.logout**, если он имеется, и выполняются все указанные в нем действия. Асинхронные процессы продолжают выполняться.

nice
nice +число
nice -число
nice команда
nice +число команда
nice -число команда

команда установки приоритета. В системе используется шкала приоритетов: **[-100; +100]**, приоритет **+100** самый низкий. Первая форма устанавливает приоритет для интерпретатора, равный **4**, вторая - приоритет, равный указанному числу, третья - приоритет для команды, равный **4**. Только администратор системы может установить отрицательный приоритет. Для выполнения запускается новый интерпретатор, который обеспечивает выполнение команды. Команда должна быть одна и не может содержать последовательность команд, конвейер или псевдоним.

nohup
nohup команда

запрещает реакцию на сигналы **SIGINT (CNTRL/C)**, **SIGQUIT (CNTRL/\)** и **SIGHUP (BREAK** - отключение удаленного терминала или ЭВМ по коммутируемой линии). Первая форма в командном файле устанавливает свое действие на все строки, указанные ниже. Вторая форма

приводит к тому, что при выполнении указанной команды эти сигналы игнорируются. Для всех асинхронных процессов **nohup** выполняется автоматически.

notify %внутренний_идентификатор_процесса
notify %шаблон_имени_процесса
notify

дает команду на асинхронный вывод сообщений об изменениях состояния процесса. Обычно эти сообщения выводятся после передачи на выполнение новой командной строки. Если команда **notify** выполнена без аргументов, то этот режим устанавливается для всех процессов.

onintr
onintr -
onintr метка

управляет реакцией на прерывания. Первая форма устанавливает стандартную реакцию на прерывания, которая заключается в завершении командного файла или возврате на уровень ввода терминальных команд. Вторая форма вызывает игнорирование всех прерываний. Последняя форма вызывает выполнение перехода на метку при получении прерывания или при завершении порожденного процесса из-за прерывания. В любом случае, если интерпретатор **csh** работает асинхронно, все формы команды **onintr** игнорируются.

popd
popd +число

выполняет команду **cd** имя_номер_2 стека имен каталогов. Имя_номер_1 из стека имен каталогов удаляется, остальные элементы стека сохраняются с новыми номерами. Форма **popd** +число удаляет имя_номер_(1+число) из стека, остальные элементы стека сохраняются с новыми номерами. При этом переход в другой каталог не осуществляется.

pushd
pushd имя_каталога
pushd +число

любая форма команды **pushd** меняет порядок в стеке имен каталогов. Форма **pushd** выполняет команду **cd** имя_номер_2 стека. При этом

имя_номер_2 ставится в вершину, а имя_номер_1 - на его место в стеке; остальные элементы стека остаются на своих местах. Форма **pushd** имя_каталога выполняет команду **cd** имя_каталога, при этом имя_каталога записывается в вершину стека, остальные элементы стека сохраняются с новыми номерами. Форма **pushd** +число выполняет команду **cd** имя_номер_(1+число). При этом имя_номер_(1+число) ставится в вершину стека, а "число" имен каталогов переписываются в конец стека в том порядке, в котором они следовали от вершины стека, другие элементы стека остаются без изменений.

rehash

обновить хеш-таблицу.

repeat число команда

команда **repeat** позволяет повторить выполнение команды указанное число раз. Команда должна быть одна в командной строке, она не должна быть последовательностью команд, псевдонимом или конвейером.

set
set имя
set имя = слово
set имя[индекс] = слово
set имя = (список_слов)
set список_присваиваний

первая форма команды **set** отображает значения всех переменных интерпретатора команд. Переменные, которые в качестве своих значений имеют не одно слово, отображаются как заключенный в скобки список слов. Вторая форма присваивает указанному имени пустую строку, третья - слово, четвертая - значение слова с номером индекс, пятая - список слов. Последняя форма используется для указания списка присваиваний - в одной командной строке несколько присваиваний. Во всех случаях происходят расширения псевдонимов командных строк и имен файлов. Подстановка переменных осуществляется перед выполнением присваиваний. Перед операциями над элементами массива его необходимо полностью определить. Не обрабатываются массивы с переменными или неопределенными границами.

setenv
setenv имя значение

первая форма выводит значение переменных окружения, вторая их устанавливает. Удалить переменную окружения можно командой **unsetenv**.

shift
shift переменная

осуществляет левый сдвиг списка слов переменной. Левый элемент списка исчезает. Попытка осуществить **shift** для пустого списка приводит к состоянию ошибки.

source файл

предназначена для выполнения командного файла без порождения нового интерпретатора. Команды **source** могут быть вложенными. Ошибка в команде **source** на любом уровне завершает выполнение. Вызов команды **source** без аргументов порождает состояние ошибки.

stop %внутренний_идентификатор_процесса
stop %шаблон_имени_процесса

останавливает выполнение асинхронного процесса.

suspend

останавливает выполнение интерпретатора.

```
switch( входная_строка )
        case образец_1:
                ...
                breaksw
        ...
        default:
                ...
endsw
```

в образцах вариантов **case** сначала выполняются подстановки. В образцах вариантов **case** можно использовать шаблоны имен файлов *, ? и [...]. Образцы вариантов **case** последовательно сравниваются с указанной в **switch** входной строкой. Если не выявлено совпадение образца со входной строкой, выполнение продолжается после **default**. Слова **case** и **default** должны стоять первыми в командной строке. Оператор **breaksw** передает управление на строку, следующую за **endsw**. Если в варианте **case** не указан оператор **breaksw**, то при совпадении с образцом выполняются все строки до первого **breaksw** или **default**. Если не обнаружено совпадение с образцом и

default отсутствует, выполнение продолжается после **endsw**.

time
time команда

при отсутствии параметров печатается итог времени, израсходованного интерпретатором и его потомками. В качестве команды нельзя использовать псевдонимы.

umask
umask маска

по умолчанию файлу устанавливается код доступа, который определяется маской. Файл будет иметь код доступа, в котором маскированы (равны 0) биты, установленные (равны 1) в маске. Пусть маска имеет вид 123. Первая цифра относится к маскированию битов доступа для владельца файла и администратора, вторая - к битам доступа группы, а третья - к битам доступа всех остальных пользователей. Значение маски указывается в восьмеричном коде. Обычно значением маски является 002, которое задает любой доступ для пользователей своей группы, доступ для чтения и выполнения другим пользователям, и 022, которое определяет любой доступ (за исключением записи) для пользователей своей группы и всех прочих. Чтобы узнать текущее значение маски, надо выполнить команду **umask** без аргумента.

unalias псевдоним ...

все псевдонимы, имена которых соответствуют указанным, отменяются. Следовательно, **unalias** * удаляет все псевдонимы. При запуске команды без аргументов порождается состояние ошибки.

unhash

запрещает использовать хеш-таблицу при поиске команды.

unlimit ресурс
unlimit

снимает ограничение на ресурс. Если ресурс не указан, снимаются ограничения на все ресурсы.

unset шаблон

удаляются все переменные, имена которых соответствуют указанному шаблону. Таким образом, "**unset** *" удаляет все переменные, установленные командой **set**.

unsetenv имя

удаляется переменная из окружения.

wait

ожидание всех выполняемых процессов. Пока выполняется команда **wait**, приглашение не печатается. Команда **wait** выполняется, пока не завершатся все запущенные на выполнение процессы. При прерывании выполнения команды **wait** (CNTRL/C или СУ/С) сообщаются имена и номера всех процессов, для которых выполнялось ожидание.

while(выражение)
...
end

цикл выполняется, пока истинно (не равно нулю) значение выражения. Ключевые слова **while** и **end** должны находиться на отдельных строках. В теле цикла можно использовать **break** для выхода из цикла и **continue** для возобновления следующей итерации цикла без завершения текущей.

Глава 3. ИНТЕРПРЕТАТОР make

Интерпретатор **make** [11], наиболее часто используемый программистами ДЕМОС, предоставляет уникальные возможности по управлению любыми видами работ в операционной системе. Всюду, где имеется необходимость учитывать зависимости файлов и времена их создания (модификации), **make** оказывается незаменимым инструментом. Интерпретатор реализует непроцедурный язык, который позволяет управлять группами командных строк системы. В основу такого управления положены зависимости между файлами с исходными данными и файлами, в которых содержатся результаты. При этом предполагается любой возможный список действий над исходными файлами: компиляция, макрообработка, редактирование, печать, упаковка или шифрование и т.д. Исходной информацией для интерпретатора является **Make**-программа, представляющая список определений макропеременных и список правил. Каждое правило включает формулировку цели и список действий для интерпретатора **shell**. При выполнении **Make**-программы интерпретатор **make** использует информацию о связях между целями и результатами и передает на выполнение **shell** списки действий, которые в данный момент необходимо выполнить для получения заданного результата. Таким образом, интерпретатор **make** позволяет записывать любой набор действий над исходными данными, благодаря чему он широко используется при решении прикладных и общесистемных задач. Очень важно и то, что **Make**-программа становится общесистемным стандартным описанием структуры задачи, алгоритма сборки и установки программного комплекса. Программист, владеющий средствами интерпретатора **make**, использует следующую технологию разработки программного комплекса, независимо от его сложности:

редактор -> **make** -> проверка -> редактор

При такой технологии существенно повышается производительность труда программиста,так как он освобождается от "ручной" сборки программ, сокращается загрузка ЭВМ - **make** "следит" за тем, чтобы при многократных компиляциях и отладках программ "не делалось то, что можно не

делать".

Важно отметить, что **make** является средством автоматизации процедур установки компонент ОС ДЕМОС. Например, компонента системы **cat** может включать следующие файлы:

выполняемый код - файл **/bin/cat**
текст программы - файл **./src/cmd/cat/cat.c**
документацию - файл **./man/man1/cat.1**
программу сборки - файл **./src/cmd/cat/Makefile**

Файл **./src/cmd/cat/Makefile** содержит всю необходимую для правильной компиляции и установки в ОС компоненты **cat** информацию. Особенно эффективен **make** для выполнения работ в программных проектах малой и средней (до **200** файлов) величин.

3.1. Принципы выполнения Make-программы

Интерпретатор **make** выполняет программу, которую будем называть **Make**-программой. **Make**-программа содержит структуру зависимостей файлов и действий над ними, оформленных в виде списка правил. Выполнение действий приводит к созданию требуемых файлов. Допустим, имеются файлы

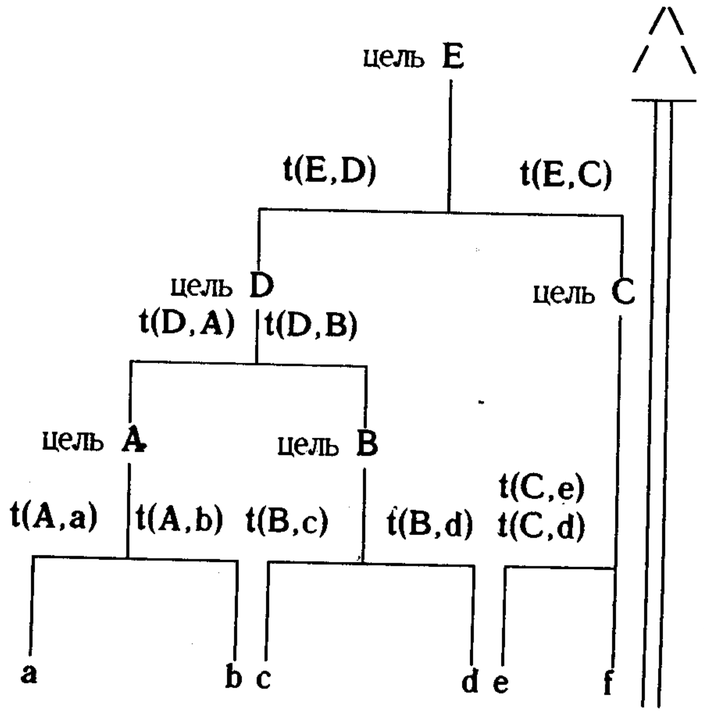
a b c d e f

из которых определенным образом необходимо получить файл **E**. Пусть далее известно, что над различными комбинациями исходных файлов выполняются некоторые действия, а результаты будут размещены в промежуточных файлах **A**, **B**, **C** и **D**. Рассмотрим граф, узлы которого - имена файлов. Дуги графа отражают зависимости файлов, стрелка указывает направление преобразований от исходных файлов к файлам, которые необходимо получить.

В **Make**-программе каждой паре **(x,y)** инцидентных узлов этого графа ставится в соответствие список действий, выполнение которых приведет к созданию **x**. Когда файл **x** существует, список действий не выполняется, но только в том случае, если файл **y** создан (модифицирован) раньше по времени, чем файл **x**. Каждой дуге графа можно поставить в соответствие значение функции **t(x,y)**. Функция **t(x,y)** возвращает результат в виде

x МОЛОЖЕ **y** - список действий не выполняется;

x СТАРЕЕ **y** - список действий выполняется.



Множество значений функции **t(x,y)** образует структуру динамических (зависящих от времени) связей между файлами. На этой основе интерпретатор **make** выделяет те разделы **Make**-программы, которые можно не выполнять.

Выше предполагалось, что каждый узел графа - это файл. Существует возможность записать в **Make**-программе список действий, выполнение которых не связано с созданием файла. Поэтому в общем случае узел графа правильнее называть целью. Пара инцидентных узлов графа образует цель и подцель. В примере узел **E** - цель, узлы **D** и **C** - подцели цели **E**. Аналогично узел **D** - цель, узлы **A** и **B** - подцели цели **D**. Наконец, узел **A** - цель, узлы **a** и **b** - подцели узла **A**. Перечисление вида ЦЕЛЬ - ПОДЦЕЛИ отражает обобщенную структуру алгоритма достижения целей.

Введем понятие "реконструкция файла-цели". Если файл-цель существует и "МОЛОЖЕ" всех файлов, от которых зависит, то он остается без изменений, иначе, если файл-цель существует и "СТАРЕЕ" какого-либо файла, от которого зависит, он реконструируется (изготавливается заново).

Приведем пример **Make**-программы, соответствующей приведенному выше графу. Программа выглядит как его линейная запись:

```
E : D C                 # E зависит от D и C
        cat D C > E     # действие правила 1
```



```
D : A B                 # D зависит от A и B
        cat A B > D     # действие правила 2

A : a b                 # A зависит от a, b
        cat a b > A     # действие правила 3

B : c d                 # B зависит от c, d
        cat c d > B     # действие правила 4

C : e f                 # C зависит от e, f
        cat e f > C     # действие правила 5

clean clear:
        -rm -f A B C D E
```

Здесь содержится **6** правил. Каждое правило включает строку зависимостей файлов и командную строку системы. Правило описывается просто: сначала указывается имя файла, который необходимо создать (цель), затем двоеточие, затем имена файлов, от которых зависит создаваемый файл (подцели), затем строки действий. Первую строку правила назовем "строкой зависимостей". Следует обратить внимание на шестое правило: в нем пустой список подцелей в строке зависимостей. Синтаксис строк действий соответствует синтаксису командных строк **shell**. Первым символом строки действия в **Make**-программе должен быть символ табуляции (или **8** пробелов) - это обязательное условие.

Все последовательности символов, начиная от символа # и до конца строки, являются комментарием. Пустые строки и лишние пробелы игнорируются. Если **Make**-программа размещена в файле **Makefile**, то имя файла с **Make**-программой в командной строке можно не указывать. Допустим, файл с **Make**-программой называется **Makefile**, в рабочем каталоге имеются файлы **a**, **b**, **c** и **d**. Файлы **A**, **B**, **C** и **D** отсутствуют, тогда по команде **make** мы получим результат - файл **E** и файлы **A**, **B**, **C** и **D**. Рассмотрим порядок выполнения **Make**-программы, когда эти файлы уже существуют, т.е. при повторном выполнении команды **make**.

Первый шаг выполнения **Make**-программы:

```
E : D C
        cat D C > E
```

Если файлы **D** и **C** существуют и не требуется их реконструкция, а файл **E** "МОЛОЖЕ", чем файлы **D** и **C**, то **make** прекратит выполнение программы - файл **E** готов. Если требуется реконструкция файлов **D** и/или **C**, то

осуществляется переход к выполнению подцелей **D** и/или **C**, затем возврат к этому правилу. Иначе, если требуется реконструкция файла **E**, то выполняется действие этого правила и **make** прекращает выполнение программы - готов реконструированный файл **E**. Иначе, если отсутствуют файлы **D** и/или **C**, будут выполнены подцели **D** и/или **C** в том порядке, в котором они указаны в списке зависимостей, затем выполняется действие этого правила и **make** прекратит выполнение программы - готов файл **E**.

Второй шаг выполнения **Make**-программы:

```
D : A B
        cat A B > D
```

Если файлы **A**, **B** и **D** существуют и не требуется их реконструкция, то выполняется переход к следующему шагу **Make**-программы (файл **D** уже готов). Если требуется реконструкция файлов **A** и/или **B**, происходит переход к выполнению подцелей **A** и/или **C**, затем возврат к этому правилу. Иначе, если требуется реконструкция файла **D**, выполняется действие этого правила и переход к следующему шагу выполнения **Make**-программы. Иначе, если отсутствуют файлы **A** и/или **B**, будут выполнены подцели **A** и/или **B** в том порядке, в котором они указаны в списке зависимостей, затем действие этого правила и переход к выполнению первого правила **Make**-программы.

Третий шаг выполнения **Make**-программы:

```
A : a b
        cat a b  > A
```

Проверяется наличие файлов **a** и **b** в рабочем каталоге. При отсутствии хотя бы одного из них выполнение программы прекращается. Затем проверяется наличие файла **A**, если его нет или требуется его реконструкция, выполняется действие этого правила. Иначе осуществляется переход к выполнению второго правила.

Четвертый шаг выполнения **Make**-программы:

```
B : c d
        cat c d  > B
```

Действия аналогичны описанным в третьем шаге, переход осуществляется к выполнению второго правила.

Пятый шаг выполнения **Make**-программ:

```
C : e f
        cat e f > C
```


Проверяется наличие файлов **e** и **f** в рабочем каталоге. При отсутствии хотя бы одного из них выполнение программы прекращается. Затем проверяется наличие файла **C**, если его нет или требуется его реконструкция, выполняется действие этого правила. Иначе осуществляется переход к выполнению первого правила.

При вызове интерпретатора **make** в командной строке можно указать имя цели. Если, например, необходимо получить файл **D**, то командная строка выглядела бы так

```
% make  D
```

или если необходимо получить файлы **C** и **D**, то командная строка выглядела бы

```
% make  C D
```

Таким образом, имя файла-цели в командной строке определяет вход в **Make**-программу. Если задано несколько входов, то **make** выполнит в указанном порядке все необходимые разделы **Make**-программы. Если же вход не указан, выполняется первое правило **Make**-программы.

В шестом правиле примера цель не является файлом, это важная особенность **make**. Программисту предоставляется возможность записать правило, цель и/или подцели которого не являются файлами. В таком случае цель - это имя входа в **Make**-программу (или метка правила). Шестое правило используется в программе для удаления файлов. Следует обратить внимание на то, что в этом правиле два имени цели (**clean** и **clear**), поэтому в командной строке можно указывать любое имя входа, например:

```
% make  clean
```

или

```
% make  clear
```

В результате выполнения будут удалены файлы **A**, **B**, **C**, **D** и **E**.

Все строки действий в правилах передаются на выполнение **shell** следующим образом:

```
sh -c "строка_действия"
```

и должны нормально выполняться интерпретатором **sh** (код возврата **0**), иначе (по получении другого кода завершения командной строки) **make**

прекратит выполнение программы. Существует способ обойти это условие. Обратите внимание на действие в **6**-м правиле: строка действий начинается с символа "-", что означает не прекращать работу при неудачном выполнении команды **rm**.

В **Make**-программе можно использовать макропеременные. Механизм макроопределений и подстановок макропеременных в **Make**-программе по смыслу аналогичен механизму подстановок в **shell**, хотя по синтаксису несколько отличается. Рассмотрим приведенный выше пример с использованием макропеременных. Теперь **Makefile** будет выглядеть так:

```
SRC1 = a b        # макроопределения
SRC2 = c d
SRC3 = e f
SRC4 = A B C D

E : D C
        cat D C > E

D : A B
        cat A B > D

A : $(SRC1)      # A зависит от SRC1
        cat $(SRC1) > A

B : $(SRC2)      # B зависит от SRC2
        cat $(SRC2) > B

C : $(SRC3)      # C зависит от SRC3
        cat $(SRC3) > C

clean clear:
        -rm -f $(SRC4)
```

Первые строки **Make**-программы - строки с макроопределениями, где каждой переменной **SRC** присваиваются значения. В правилах выполняется операция подстановки значения макропеременной, например **$(SRC1)**. Макропеременные позволяют манипулировать списками имен файлов при минимальных изменениях в тексте **Make**-программы.

Интерпретатор **make** реализует механизм обработки умолчаний зависимостей файлов со стандартными суффиксами имен. Например, файл с суффиксом имени ".o" можно получить из файлов с суффиксами имен ".c" (язык программирования Си) и ".s" (язык ассемблер). Рассмотрим

пример. Допустим, имеются файлы **a.c**, **b.c**, **c.c** и **d.s**, образующие программу, которую назовем **program**. Файлы **a.c**, **b.c** и **c.c** содержат строку

```
# include "program.h"
```

т.е. зависят от файла **program.h**. **Make**-программа для работы с этими файлами будет содержать **3** строки

```
program: a.o b.o c.o d.o
        cc a.o b.o c.o d.o -o program

a.o b.o c.o: program.h
```

По команде **make** будет создан файл **program**. При первом выполнении получим на экране дисплея:

```
cc  -c a.c
cc  -c b.c
cc  -c c.c
as - -o d.o d.s
cc a.o b.o c.o d.o -o program
```

Обратите внимание на то, что интерпретатор определил необходимые действия над исходными файлами с суффиксами имен ".c" и ".s", хотя имена этих файлов в **Make**-программе не указаны, и правильно осуществил сборку программы. Теперь, допустим, мы вызываем **make** на выполнение второй раз после редактирования файла **program.h**, при этом получим:

```
cc  -c a.c
cc  -c b.c
cc  -c c.c
cc a.o b.o c.o d.o -o program
```

Если выполнить **Make**-программу после редактирования файла **b.c**, то получим:

```
cc  -c b.c
cc a.o b.o c.o d.o -o program
```

Наконец, если, допустим, необходимо получить файл **c.o**, то можно выполнить команду

make c.o

Механизм умолчаний и обработки суффиксов спроектирован для автоматизации программирования Make-программ; он существенно сокращает размеры программ и количество ошибок в них.

3.2. Соглашения языка Make

Ниже в компактной форме приводятся основные синтаксические конструкции языка **Make**. В следующих параграфах по каждой конструкции будут даны описания и примеры.

идентификатор
: последовательность букв, цифр и символа "_", содержащая шаблоны имен файлов ([...], ?, *), символы "/" и "."

идентификатор = значение
: определение макропеременной. В правой части могут быть макроподстановки, а также вся правая часть может быть пустой строкой.

$(идентификатор)
${идентификатор}
$символ

: подстановка значения макропеременной. Специальное значение символа **$** можно отменить, указывая **$$**.

комментарий
: текст, следующий за символом # и до конца строки. Специальное значение символа # отменяется, если он указан в кавычках.

обратная наклонная черта
: символ продолжения строки. Специальное значение символа \ отменяется, если он указан дважды.

пустая строка, пробелы
: пробелы служат разделителями слов в командной строке, лишние пробелы и табуляции игнорируются. Пустая строка всюду игнорируется.

список_зависимостей
: список_действий

правило в общем виде. Список_действий может быть пустым.

.первый_суффикс.второй_суффикс:
: список_действий

правило с указанием зависимостей суффиксов. Если в **Make**-программе содержится хотя бы одно правило с суффиксами, отличными от предопределенных, в нее необходимо включить правило:

.SUFFIXES: список_суффиксов

список_целей :[:] список_подцелей
: список_зависимостей. Список_подцелей может быть пустым. Правила с одним и двумя символами двоеточия в списке_зависимостей отличаются порядком выполнения списка_подцелей и списка_действий.

имя_цели [имя_цели]
: список_целей. Имя_цели - идентификатор. Можно использовать символ / и точку. Имя_цели может быть именем файла или каталога, тогда включается выполнение механизма реконструкции. Имя_цели может не быть именем файла, тогда механизм реконструкции не включается, а имя_цели является меткой (именем правила, входа в **Make**-программу).

имя_подцели [имя_подцели] [#комментарий]
: список_подцелей. Имя_подцели - идентификатор. Можно использовать шаблоны имен файлов "*", "?", [...], символ / и точку. Имя_подцели может быть именем файла, для которого записано или не записано правило в **Make**-программе, в этих случаях включается механизм реконструкции. Имя_подцели может не быть именем файла, в этом случае механизм реконструкции не включается, а имя_цели является меткой (именем правила, входа в **Make**-программу).

строка_действия [#комментарий]
. . . . . .
строка_действия [#комментарий]

: список_действий. Любые командные строки ОС ДЕМОС и управляющие конструкции **Shell**.

'табуляция'командная_строка_**shell**
: строка_действия. Строка действия может быть указана в строке зависимостей через символ ";". Строку_действия можно указывать в следующих форматах:

'табуляция'командная__строка__**shell** 1

'табуляция'@командная_строка__**shell** 2

'табуляция'-командная_строка__**shell** 3

В первом формате командная строка выводится на печать; если код возврата после ее выполнения не **0**, **make** прекращает выполнение программы по ошибке. Во втором формате командная строка не выводится на печать; если код возврата после ее выполнения не **0**, **make** прекращает выполнение программы по ошибке. В третьем формате командная строка выводится на печать; если код возврата после ее выполнения не **0**, **make** игнорирует ошибку и выполнение программы продолжается. В четвертом формате командная строка не выводится на печать; если код возврата после ее выполнения не **0**, **make** игнорирует ошибку и выполнение программы продолжается. Простая командная строка (одна команда ДЕМОС с аргументами) выполняется без порождения оболочки. Другие командные строки выполняются **sh** следующим образом: **sh** -c командная_строка_shell

Для сокращения обозначений предусмотрены макропеременные с предопределенными именами. В правиле без суффиксов строка_действия может включать следующие предопределенные макропеременные:

@ имя цели;

? имена файлов из списка подцелей, которые МОЛОЖЕ файла_цели. Эти файлы участвуют в реконструкции цели.

В правиле с суффиксами строка_действия может включать следующие предопределенные макропеременные:

***** основа_имени_цели;

@ основа_имени_цели.второй_суффикс;

< основа_имени_цели.первый_суффикс.

3.3. Использование макропеременных

При выполнении **Make**-программы значения макропеременных устанавливаются в строках макроопределений и/или в командной строке при запуске **make** на исполнение. Кроме того, существуют макропеременные с предопределенными именами, значения которых устанавливаются при выполнении **Make**-программы. К ним относятся: макропеременная **@**, ее значение - имя_цели; макропеременная **?**, ее значение - имена тех файлов_подцелей, которые МОЛОЖЕ файла_цели.

Предопределенные макропеременные "**@**" и "**?**" используются только в списке действий правила и в каждом правиле имеют свои значения. Определять значения макропеременных можно в любом месте **Make**-программы, но не внутри правила. Интерпретатор **make** составляет список имен

макропеременных и присваивает им значения в процессе чтения **Make**-программы. Если значение макропеременной переопределяется, то ей присваивается новое значение; если используются макроопределения с вложенными макроподстановками, то значения устанавливаются с учетом всех присвоений в **Make**-программе. Если значение макропеременной задается в командной строке при запуске **Make**-программы на исполнение, то все определения этой макропеременной в **Make**-программе игнорируются и используется значение, взятое из командной строки. Состояние ошибки порождается, если используется рекурсия при присвоении значения макропеременным. Например,

```
A = $B
B = $A
```

приведет к аварийному завершению выполнения **Make**-программы.

В макроопределениях можно использовать метасимволы шаблонов имен файлов **shell**. Допустим, рабочий каталог имеет вид:

```
-rw-r--r-- 1 user     15658 Авг  6 16:03 1.m
-rw-r--r-- 1 user      2158 Авг  8 16:38 2.m
-rw-r--r-- 1 user      5185 Авг  9 17:38 3.m
-rw-r--r-- 1 user      4068 Июл 28 20:56 6.m
-rw-r--r-- 1 user       100 Авг  9 14:30 f2
-rw-r--r-- 1 user        66 Авг  9 17:42 f3
```

Пример **Make**-программы в **Makefile** :

```
A = *
B = f?
C = [0-9].*

aaa :
        echo $A
        echo $B
        echo $C
```

После выполнения команды **make** -s получим на экране дисплея:

```
1.m 2.m 3.m 6.m f2 f3
f2 f3
1.m 2.m 3.m 6.m
```

В **Make**-программе часто бывает необходимо манипулировать именами каталогов и файлов. Механизм макроподстановок предлагает удобные средства для этого. Пример:

```
A = *
B = [pg]*
C = f?r*
DIR1 = .
DIR2 = /etc
DIR3 = /usr/bin

aaa :
        echo ${DIR1}/$A
        echo ${DIR2}/$B
        echo ${DIR3}/$C
```

После выполнения получим:

```
./1.m ./2.m ./3.m ./6.m ./f2 ./f3
/etc/getty /etc/group /etc/passwd
/usr/bin/fgrep
```

Рассмотрим пример, в котором демонстрируются всевозможные способы использования макропеременных. Допустим, имеется **Makefile**

```
БИБЛИОТЕКА = ПОЛКА ${ДРУГОЕ}
ДРУГОЕ = Документы
Шкаф = ПОЛКА
папка = справки, копии.
СПРАВОЧНИКИ =
ЖУРНАЛЫ =
Словари = толковые, иностранных языков.
ТЕХНИЧЕСКАЯ = $(СПРАВОЧНИКИ) $(Словари)
ХУДОЖЕСТВЕННАЯ = проза, поэзия, драматургия.

t = Справка с места жительства.
x = Копия свидетельства о рождении.
файлы = d e

библиотека : ${БИБЛИОТЕКА}
        echo ´Действия правила´ $@
        echo ´$$? - список подцелей :´ $?
```



```
        echo ´$$@ - имя цели :´ $@
        echo ´Техническая : ´ $(ТЕХНИЧЕСКАЯ)
        echo ´Худ. : ´ $(ХУДОЖЕСТВЕННАЯ)
        echo ´ ´

${Шкаф} : b c
        echo ´Действия правила´ $@
        echo ´$$? - список подцелей :´ $?
        echo ´$$@ - имя цели :´ $@
        echo ´ ´

${ДРУГОЕ}: ${файлы}
        echo ´Действия правила´ $@
        echo ´$$? - список подцелей :´ $?
        echo ´$$@ - имя цели :´ $@
        echo ´Папка : ´ ${папка}
        echo $t
        echo $x
        echo ´ ´

b:
c:
d:
e:
```

Следует обратить внимание на то, что **$**буква используется для подстановки значения макропеременной, имя которой состоит из одной буквы, а **$**(идентификатор) и **$**{идентификатор} равноценны. Правила **b**, **c**, **d** и **e** включены исключительно для демонстрации значений макропеременной "?". Теперь, если выполнить команду **make -s**, получим на экране дисплея:

```
Действия правила ПОЛКА
$? - список подцелей : b c
$@ - имя цели : ПОЛКА

Действия правила Документы
$? - список подцелей : d e
$@ - имя цели : Документы
Папка :  справки, копии.
Справка с места жительства.
Копия свидетельства о рождении.
```

Действия правила библиотека
$? - список подцелей : ПОЛКА Документы
$@ - имя цели : библиотека
Техническая :толковые, иностранных языков.
Худ. : проза, поэзия, драматургия.

В командной строке можно присвоить значение макропеременной. После команды

make -s "Словари = английский, немецкий."

получим на экране дисплея:

Действия правила ПОЛКА
$? - список подцелей : **b c**
$@ - имя цели : ПОЛКА

Действия правила Документы
$? - список подцелей : **d e**
$@ - имя цели : Документы
Папка : справки, копии.
Справка с места жительства.
Копия свидетельства о рождении.

Действия правила библиотека
$? - список подцелей : ПОЛКА Документы
$@ - имя цели : библиотека
Техническая : английский, немецкий.
Худ. : проза, поэзия, драматургия.

3.4. Выполнение правил в Make-программе

Существует несколько разновидностей правил:
с одним и двумя двоеточиями;
с одинаковыми именами целей;
не содержащие списка действий;
не содержащие списка подцелей и/или списка действий;
правила, предопределенные в интерпретаторе **make**, которые программист может включать в **Make**-программу.

Кроме того, имеются некоторые различия в выполнении **Make**-программы, когда имя цели не является файлом. В общем случае правило

может содержать одно или два двоеточия в качестве разделителей списков целей и подцелей. Порядок выполнения в этих правилах одинаков, если в них указаны различные имена целей.

Правило выполняется следующим образом: сначала выполняются правила с именами целей из списка подцелей, затем список действий. Если в списке подцелей указано имя файла, для которого правило не определено, то время создания (модификации) файла используется для определения необходимости реконструкции цели. Если имя подцели не является файлом, оно должно быть меткой правила, иначе порождается состояние ошибки. Допустим, в **Make**-программе записано правило

monitor.c : monitor.h

Это означает, что файл **monitor.c** зависит от файла **monitor.h**. Если **monitor.h** действительно имеется в рабочем каталоге, то любые изменения в нем приведут к реконструкции файла **monitor.o**, если файл **monitor.h** отсутствует, **make** прекращает выполнять программу по ошибке. В **Make**-программе могут встречаться случаи, когда необходимо для одной цели записать несколько правил, тогда существенно важно, сколько двоеточий указано в правиле. Ниже приведены схемы, в которых цифрами показан порядок выполнения этих правил.

Порядок выполнения нескольких правил с одинаковым именем_цели и одним двоеточием: сначала выполняются правила, связанные со списками подцелей, затем список_действий одного из правил. Список действий разрешается указывать только в одном из таких правил:

```
имя_цели_A : список_подцелей    <—| 1 |

имя_цели_A : список_подцелей    <—| 2 |
             список_действий    <—| 4 |

имя_цели_A : список_подцелей    <—| 3 |
```

Порядок выполнения нескольких правил с одинаковым именем_цели и двумя двоеточиями. Список действий может быть в каждом правиле:

```
имя_цели_A :: список_подцелей   <—| 1 |
              список_действий   <—| 2 |

имя_цели_A :: список_подцелей   <—| 3 |
```

```
                список_действий   <—| 4 |

имя_цели_A :: список_подцелей   <—| 5 |
              список_действий     <—| 6 |
```

3.5. Режимы выполнения Make-программы

Интерпретатор **make** предоставляет ряд возможностей для управления выполнением **Make**-программы. Для этой цели используются следующие директивы:

.SILENT	- не печатать строки действий;
.IGNORE	- игнорировать ошибки действий;
.DEFAULT	- выполнить альтернативное действие;
.PRECIOUS	- удалить недостроенный файл.

Директивы .SILENT и .IGNORE можно указывать как в **Make**-программе, так и в командной строке ключами -s и -i, DEFAULT и PRECIOUS только в **Make**-программе. Допустим, имеется следующая **Make**-программа:

```
aa:     bb
        echo Действия правила $@

bb:
        echo Действия правила $@
        ppp     # Несуществующая команда
        sort e  # Нет файла e
```

После выполнения получим сообщения:

```
echo Действия правила bb
Действия правила bb
ppp     # Несуществующая команда
sh: ppp: не найден
*** Код ошибки 1
Конец.
```

Интерпретатор **make** прекратил работу по первой ошибке. Кроме того, на экран дисплея выводятся строки действий. Отменить вывод строк действий можно следующими способами:

указать ключ -s в командной строке при запуске **make**;
указать в **Make**-программе директиву **SILENT**;
начинать каждую строку действий символом **@**.

В первых двух случаях результат одинаков - не будут выводиться строки действий. В третьем случае не будут выводиться строки действий вида

'табуляция'@строка_действий

Пример использования директивы **SILENT**.

```
aa: bb
        echo Действия правила $@

bb:
        echo Действия правила $@
        ppp     # Несуществующая команда
        sort e # Нет файла e

.SILENT:
```

После выполнения программы получим:

```
Действия правила bb
sh: ppp: не найден
*** Код ошибки 1
Конец.
```

По любой ошибке **make** прекратит выполнение программы. Существуют следующие способы игнорирования ошибок (**Make**-программа продолжает выполняться):

указать ключ **-i** при запуске **make** на выполнение;
указать в **Make**-программе директиву **IGNORE**;
после символа табуляция указать символ "-".

В первом и втором случаях будут проигнорированы все ошибки при выполнении строк действий, в третьем игнорируются ошибки в тех строках действия, которые указаны следующим образом:

'табуляция'-строка_действий

Пример использования директив **SILENT** и **IGNORE**. Обработка ошибок при выполнении действий в правилах

```
aa: bb
        echo Действия правила $@

bb: a b c d
        echo Действия правила $@
        ppp     # Несуществующая команда
        sort e # Нет файла e

.SILENT:

.IGNORE:
```

После выполнения программы получим:

```
Действия правила bb
sh: ppp: не найден
*** Код ошибки 1 (игнорирован)
sort: не могу открыть e
*** Код ошибки 1 (игнорирован)
Действия правила aa
```

Правило **DEFAULT** используется для указания альтернативных действий по отсутствующему в данный момент файлу. Правило **DEFAULT** позволяет записать список действий, которые будут выполняться для всех отсутствующих файлов, поэтому требуется определенная осторожность, например:

```
aa: bb
        echo Действия правила $@
bb:  a b c d
        echo Действия правила $@
        ppp     # Несуществующая команда
        sort e # Нет файла e

.SILENT:

.IGNORE:

.DEFAULT:
        echo Действия правила .DEFAULT для $@
```


После выполнения программы получим:

```
Действия правила .DEFAULT для a
Действия правила .DEFAULT для b
Действия правила .DEFAULT для c
Действия правила .DEFAULT для d
Действия правила bb
sh: ppp: не найден
*** Код ошибки 1 (игнорирован)
sort: не могу открыть e
*** Код ошибки 1 (игнорирован)
Действия правила aa
```

Часто бывает необходимо прекратить выполнение **Make**-программы. Это не приведет к фатальным последствиям, так как сохраняется структура динамических (зависящих от времени) связей файлов. Однако, если создание некоторого файла не завершилось и он тем не менее образовался, его желательно удалить перед повторным запуском интерпретатора. Это можно сделать автоматически, используя директиву **PRECIOUS**, например:

```
aaa:    file
        sort file > $@

.PRECIOUS:
```

Если в момент синхронного исполнения **Make**-программы ввести сигнал **CNTRL/C**, файл **$@** будет удален.

3.6. Правила с суффиксами

В **Make**-программе можно записать одно правило для обработки различных файлов. В этом случае это одно правило многократно выполняется для различных файлов, что существенно сокращает размеры **Make**-программ, упрощает их разработку. Все полезные свойства **make** при этом сохраняются. Механизм выполнения таких правил строится на суффиксах имен файлов. Допустим, из файла с именем основа.суффикс_1 необходимо получить файл с именем основа.суффикс_2, тогда обычное правило будет выглядеть так:

основа.суффикс_2 : основа.суффикс_1

список действий

Понятно, что для группы файлов, основы имен которых различны, а первый и второй суффиксы имен одинаковы, желательно было бы записать одно правило обработки. Например, файлы "*.c" обычно преобразуются в файлы "*.o" одним списком действий, и, следовательно, эти правила желательно записывать в виде одного правила. Интерпретатор **make** предлагает эту возможность в правилах вида

.суффикс_1.суффикс_2:

список_действий

.SUFFIXES: .суффикс_1 .суффикс_2

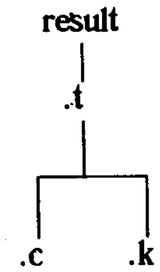
Если в **Make**-программе записаны эти правила, интерпретатор, получив в качестве аргумента имя файла с именем основа.суффикс_1, выполнит указанное выше правило, и мы получим результат - файл с именем основа.суффикс_2. Порядок выполнения правила с суффиксами такой же, как и в обычном правиле. Предопределенное правило **SUFFIXES** используется для указания списка суффиксов, который может содержать более двух суффиксов. Порядок суффиксов в списке роли не играет. Естественно, для каждой пары суффиксов в **Make**-программе должны быть записаны соответствующие правила. В правилах с суффиксами используются предопределенные макропеременные:

@ - имя_результата (основа.суффикс_2);

< - имя_аргумента (основа.суффикс_1);

***** - основа.

Рассмотрим пример. Допустим, имеются исходные файлы "*.c" и "*.k". Необходимо из них получить файлы "*.t", а из них - файл **result**. Допустим также, что файлы "*.c" зависят от файла **file.h** (содержат строку **# include** "file.h"). Граф преобразований файлов в этом случае выглядит так:



Программа, реализующая этот граф, может быть следующей:

```
result: d.t a.t b.t c.t
        echo ´Действия правила´ $@
        echo ´ Значение $$@ - ´ $@
        echo ´ Значение $$? - ´ $?
        touch $@

.k.t :
        echo ´Действия правила .k.t :´
        echo ´ Значение $$* - ´ $*
        echo ´ Значение $$@ - ´ $@
        echo ´ Значение $$? - ´ $?
        echo ´ Значение $$< - ´ $<
        touch $@

.c.t :
        echo ´Действия правила .c.t :´
        echo ´ Значение $$* - ´ $*
        echo ´ Значение $$@ - ´ $@
        echo ´ Значение $$? - ´ $?
        echo ´ Значение $$< - ´ $<
        touch $@

a.c b.c c.c : file.h

.SUFFIXES: .t .c .k
```

После выполнения команды **make -rs** получим:

```
 Действия правила .k.t :
  Значение $* - d
  Значение $@ - d.t
  Значение $? - d.k
  Значение $< - d.k
```

```
Действия правила .c.t :
 Значение $* -  a
 Значение $@ -  a.t
 Значение $? -  file.h a.c
 Значение $< -  a.c

Действия правила .c.t :
 Значение $* -  b
 Значение $@ -  b.t
 Значение $? -  file.h b.c
 Значение $< -  b.c

Действия правила .c.t :
 Значение $* -  c
 Значение $@ -  c.t
 Значение $? -  file.h c.c
 Значение $< -  c.c

Действия правила result
 Значение $@ -  result
 Значение $? -  d.t a.t b.t c.t
```

Заметим, что правило ".k.t" выполняется только один раз, правило ".c.t" - три раза, как и требовалось для списка исходных файлов, указанных в списке подцелей правила **result**. Смысл макропеременной "?" в правиле с суффиксом тот же, что и в обычном файле - список подцелей. Макропеременная "@" в обычном правиле - имя цели, а здесь имя результата - файл с именем основа.суффикс_2.

Интерпретатор **make** содержит список правил для стандартных суффиксов имен файлов и определения самих суффиксов. Этот список подключается к **Make**-программе пользователя, если **make** запускается на выполнение без ключа "-r". Программист полностью освобождается от указания правил преобразований файлов со стандартными суффиксами имен. Обрабатываются файлы, имена которых включают следующие суффиксы:

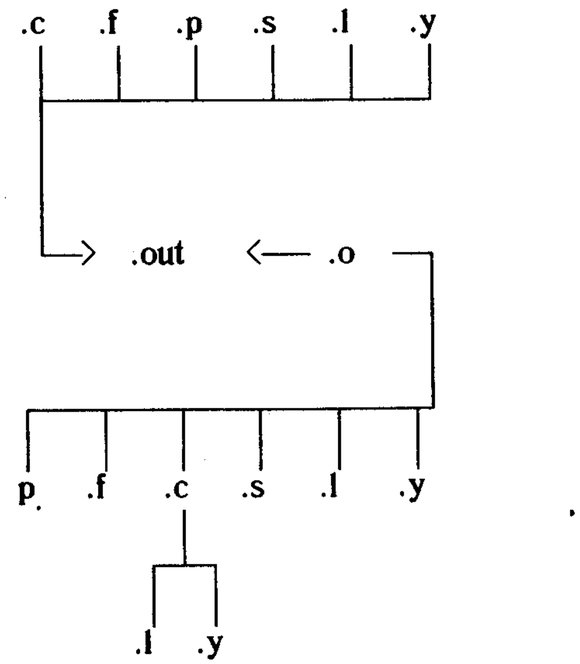
.out	-	файл в загрузочном формате;
.o	-	объектный файл;
.c	-	файл на языке Си;
.f	-	файл на языке Фортран;
.p	-	файл на языке Паскаль;
.s	-	файл на ассемблере;
.l	-	файл на **lex**;

.y	-	файл на **yacc**;

Кроме того, в **Make**-программу включаются предопределенные макропеременные:

```
LOADLIBES = # имена библиотек
AS = as -     # имя ассемблера
CC = cc       # имя Си-компилятора
CFLAGS =      # ключи Си-компилятора
PC = pc       # имя Паскаль-компилятора
PFLAGS =      # ключи Паскаль-компилятора
FF = f77      # имя f77-компилятора
FFLAGS =      # ключи f77-компилятора
LEX = lex     # имя генератора
LFLAGS =      # ключи lex
YACC = yacc   # имя генератора
YFLAGS =      # ключи yacc
```

Значения предопределенных макропеременных можно менять в **Make**-программе, например, если ввести строку "CFLAGS = -O", то предопределенная макропеременная **CFLAGS** будет иметь новое значение в **Make**-программе. Ниже приводятся граф зависимостей целей и список правил **Make**-программы, реализующей обработку файлов по суффиксам имен.



```
.l.out:
	$(LEX) $<
	$(CC) $(CFLAGS) lex.yy.c $(LOADLIBES) -ll -o $@
	rm lex.yy.c

.y.out:
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) y.tab.c $(LOADLIBES) -ly -o $@
	rm y.tab.c

.f.out:
	$(FF) $(FFLAGS) $< $(LOADLIBES) -o $@
	-rm $*.o

.o.out:
	$(CC) $(CFLAGS) $< $(LOADLIBES) -o $@

.c.out:
	$(CC) $(CFLAGS) $< $(LOADLIBES) -o $@

.p.out:
	$(PC) $(PFLAGS) $< $(LOADLIBES) -o $@

.s.out:
	$(CC) $(CFLAGS) $< $(LOADLIBES) -o $@

.l.c:
	$(LEX) $<
	mv lex.yy.c $@

.y.c:
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@

.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm lex.yy.c; mv lex.yy.o $@

.y.o:
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) -c y.tab.c
```



```
	rm y.tab.c; mv y.tab.o $@

.s.o:
	$(AS) -o $@ $<

.f.o:
	$(FF) $(FFLAGS) -c $<

.c.o:
	$(CC) $(CFLAGS) -c $<

.p.o:
	$(PC) $(PFLAGS) -c $<

.SUFFIXES: .out .o .c .f .p .y .l .s
```

Допустим, имеются файлы **f[1-6].c** с исходными текстами программы **result**. Файлы **f[1-3].c** содержат строки

```
# include "file1.h"
# include "file2.h"
```

и файлы **f[4-6].c** содержат строку

```
# include "file3.h"
```

Следующая программа управляет созданием программы **result**:

```
CFLAGS = -O
LDFLAGS = -s -n -o
OBJS = f1.o f2.o f3.o f4.o f5.o f6.o

result:  ${OBJS}
    ${CC} ${OBJS} ${LDFLAGS} $@

f1.o f2.o f3.o : file1.h  file2.h

f4.o f5.o f6.o : file3.h
```

Так выглядит протокол выполнения:

```
cc -O -c f1.c
```

```
cc -O -c f2.c
cc -O -c f3.c
cc -O -c f4.c
cc -O -c f5.c
cc -O -c f6.c
cc f1.o f2.o f3.o f4.o f5.o f6.o -s -n -o result
```

Теперь изменим время модификации файла командой

```
touch file3.h
```

и снова выполним программу, в результате получим:

```
cc -O -c f4.c
cc -O -c f5.c
cc -O -c f6.c
cc f1.o f2.o f3.o f4.o f5.o f6.o -s -n -o result
```

Теперь изменим время модификации файла командой **touch f3.c** и снова выполним программу

```
cc -O -c f3.c
cc f1.o f2.o f3.o f4.o f5.o f6.o -s -n -o result
```

Программист может отключить стандартные определения и список правил с суффиксами (ключ **-r**), может их использовать наряду с теми, что определены в **Make**-программе. Допустим, к имеющимся файлам предыдущего примера добавляются файлы **a1.t**, **a2.t** и **a3.t**, их необходимо обработать программой **sed**, выходом которой будут файлы **a1.c a2.c a3.c**. Теперь **Make**-программа выглядит так:

```
CFLAGS = -O
LDFLAGS = -o
OBJS = a1.o a2.o a3.o f1.o f2.o f3.o f4.o f5.o f6.o

result:  ${OBJS}
   ${CC} ${OBJS} ${LDFLAGS} $@

f1.o f2.o f3.o : file1.h  file2.h

f4.o f5.o f6.o : file3.h
```



```
a1.c: a1.t

a2.c: a2.t

a3.c: a3.t

.t.c:
        sed "s/aaa/bbb/" < $< > $*.c

.SUFFIXES: .t
```

Протокол выполнения программы:

```
sed "s/aaa/bbb/" < a1.t > a1.c
cc -O -c a1.c
sed "s/aaa/bbb/" < a2.t > a2.c
cc -O -c a2.c
sed "s/aaa/bbb/" < a3.t > a3.c
cc -O -c a3.c
cc -O -c f1.c
cc -O -c f2.c
cc -O -c f3.c
cc -O -c f4.c
cc -O -c f5.c
cc -O -c f6.c
cc a1.o a2.o a3.o f1.o f2.o f3.o f4.o f5.o f6.o -o result
```

Make допускает обработку суффиксов только одного уровня вложенности. В правиле **SUFFIXES** указан только суффикс **t**, суффикс **c** определен в подключаемом списке суффиксов.

3.7. Управление архивом в Make-программе

В ОС ДЕМОС существуют два типа архивов: архив текстовых файлов и архив объектных файлов (имеющий структуру библиотеки объектных модулей). Созданный архив является одним файлом, а файлы, из которых он собран, образуют его части. Управление архивом включает две задачи: создание файла_архива определенной структуры и его модификация (добавление, удаление и замена частей, изменение структуры, декомпозиция архива на файлы, из которых он был образован). Для решения таких задач **Make**-программа должна обеспечивать работу с целями двух типов: файл_архив и файл для включения в архив. При этом оценивается время

последней модификации включаемого в архив файла (а не время создания архива или записи файла_части в архив). Имя файла_архива может указываться в списке зависимостей правил как обычная цель:

имя_архивного_файла(имя_файла_для_включения_в_архив)

Кроме имен файлов, при работе с библиотекой объектных модулей можно указывать имена функций

имя_файла_библиотеки((_внешнее_имя_библ_функции))

Рассмотрим фрагмент **Make**-программы для построения библиотеки с именем **libP.a**:

```
L       =   libP.a
CFLAGS = -O

$(L)::
        ar r $(L)

$(L):: $(L)(la.o) $(L)(La.o) $(L)(Da.o)

# labs, Labs, Dabs - имена функций

$(L)(labs.o):       lib/la.c
        $(CC) $(CFLAGS) lib/la.c
        ar r $(L) la.o
        -rm -f la.o

$(L)(Labs.o):       lib/La.c
        $(CC) $(CFLAGS) lib/La.c
        ar r $(L) La.o
        -rm -f La.o

$(L)(Dabs.o):       lib/Da.c
        $(CC) $(CFLAGS) lib/Da.c
        ar r $(L) Da.o
        -rm -f Da.o
```

У такого способа работы с библиотекой есть недостаток - **Make**-программа должна содержать структуру библиотеки. Список имен библиотечных модулей во втором правиле "**$(L)::**" соответствует порядку их размещения в

библиотеке. Еще одно неудобство заключается в том, что если бы библиотека содержала большое количество модулей, например **100**, то потребовалась бы запись **100** правил в программе. Для построения однопроходных библиотек такой способ указания структуры имеет существенный недостаток, так как изменение исходного текста объектного модуля может привести к необходимости изменить структуру библиотеки, а это, в свою очередь, приведет к необходимости реконструировать **Make**-программу. Кроме того, при построении библиотеки необходимо, чтобы все объектные файлы были в рабочем каталоге.

Рассмотрим **Make**-программу, в которой эти недостатки устранены. Возьмем в качестве примера файлы с исходными текстами модулей объектной библиотеки **mylib**

m1.c m2.c m3.c m4.s m5.c m6.c m7.s

Допустим, структура однопроходной библиотеки с учетом вызовов модулей должна быть такой:

m4.o m2.o m1.o m5.o m6.o m7.o m3.o

Текст **Make**-программы:

```
CFLAGS = -O
SRC    = m1.c m2.c m3.c m4.s m5.c m6.c m7.s
LIB    = mylib

${LIB}: ${SRC}
      echo $? | sed s/\\.[cs]/\\.o/g > list
      make `cat list`
      ar cr $@ `cat list`
      lorder $@ | tsort > list
     -@if cmp list topology ; \
      then \
          rm -f `cat list` list;\
      else \
          ar x ${LIB}; rm $@;\
          mv list topology;\ .
          ar cr $@ `cat topology`;\
          rm -f `cat topology`;\
          echo Структура $@ изменилась.;\
      fi
      ranlib ${LIB}
```

```
        echo Библиотека $@ готова.

m1.c : x.h
        touch m1.c

m2.c : x.h y.h
        touch m2.c

m3.c : x.h
        touch m3.c

m5.c : y.h
        touch m5.c

m6.c : x.h
        touch m6.c
```

Рассмотрим простые случаи реконструкции уже существующей библиотеки. Допустим, изменился исходный текст одного из модулей. В этом случае достаточно на его место в библиотеке включить новую версию объектного файла этого модуля. Чтобы не компилировать другие файлы библиотеки, можно использовать предопределенную макропеременную "?" - список файлов, которые стали МОЛОЖЕ файла **mylib**. Как уже говорилось выше, в каталоге нет объектных файлов. Следовательно, в **Make**-программу необходимо включить предопределенные средства обработки суффиксов, а в качестве аргумента формировать имя цели с суффиксом **o**, тогда **make** автоматически построит новый объектный файл этого модуля. В примере файл **list** используется в качестве временного рабочего файла. Строка вида

```
echo $? | sed s/\\.[cs]/\\.o/g > list
```

записывает в файл **list** список целей с суффиксом **o**. Редактор **sed** используется в этой командной строке для замены суффиксов **c** и **s** на **o**. Таким образом, файл **list** содержит имена файлов-целей, которые необходимо создать и на этой основе реконструировать библиотеку. Это можно сделать, например, так:

```
make `cat list`
```

В результате выполнения будут созданы нужные объектные файлы. После этого можно включить объектные модули на свои места в библиотеке **mylib**


```
ar rc mylib `cat list`
```

Если же библиотека **mylib** отсутствует,то она создается, если модуль в библиотеке исходно отсутствовал, он записывается в конец библиотеки (так работает команда **ar** с ключами **cr**).

По команде "**make `cat list`**" выполняться будет **Makefile** (здесь срабатывает механизм умолчания имени файла с **Make**-программой). Таким образом, имеет место рекурсивный вызов **Make**-программы. При рекурсивном вызове в качестве имен целей передаются имена объектных файлов, которые необходимо получить и включить в библиотеку. В интерпретаторе **make** нет средств, позволяющих менять список целей в процессе выполнения **Make**-программы. Когда это необходимо, список целей создается в **Make**-программе и передается на выполнение как список аргументов при вызове подпрограммы. Рекурсия в примере понадобилась для того, чтобы не записывать полный список правил для всех файлов-целей, а вызвать **make** с актуальным списком аргументов.

Возможна такая реконструкция библиотеки, когда меняется ее структура. Для этого в **Make**-программе используются команды **lorder** и **tsort**. **Lorder** выводит список вызовов модулей существующей библиотеки, а **tsort** сортирует этот список таким образом, чтобы структура библиотеки была непротиворечивой, т.е. однопроходный редактор связей мог бы за одно чтение библиотечного файла найти все необходимые модули. В **Make**-программу включаются действия, в которых строится структура библиотеки во временном файле **list** и запоминается в сохраняемом файле **topology**. Возможны следующие случаи реконструкции библиотеки:

1) Реконструкция библиотеки не изменила ее структуры, в этом случае файл **topology** не отличается от файла **list**.
2) Реконструкция изменила структуру библиотеки. В этом случае файл **topology** отличается от файла **list** и требуется, во-первых, получить верную структуру и запомнить ее в файле **topology**, во-вторых, извлечь из библиотеки все объектные модули ("разобрать" библиотеку и удалить файл с библиотекой), затем собрать ее в соответствии с новой структурой. Разобрать библиотеку можно командой

```
ar x mylib
```

В рабочем каталоге будут созданы копии объектных модулей из библиотечного файла. Библиотечный файл при этом сохраняется. Разобранную библиотеку можно собрать заново, используя информацию о структуре библиотеки в файле **topology** и команду:

```
ar rc mylib `cat topology`
```

В **shell** допускается записать **if** одной строкой следующим образом:

```
if      список_команд;\
then \
        список_команд;\
else \
        список_команд;\
fi
```

В **Makefile** эта конструкция записана:

```
-@if cmp list topology ; \
  then \
      rm -f `cat list` list;\
  else \
      ar x $(LIB); rm $@;\
      mv list topology;\
      ar cr $@ `cat topology`;\
      rm -f `cat topology`;\
      echo Структура $@ изменилась.;\
  fi
```

Первая строка в ней выглядит так:

```
'табуляция'-@if cmp list topology ; \
```

Остальные строки имеют более 8 ведущих пробелов. Символ "-" указан, так как в отдельных версиях **shell** оператор **if** при нормальном завершении возвращает не 0. Символ "@" отменяет вывод этой строки перед выполнением.

Приведенная **Make**-программа позволяет работать с любыми объектными библиотеками. Для конкретной библиотеки (или архива) нужно изменить макропеременные **LIB**, **SRC** и записать зависимости от файлов включений. Если необходимо работать с текстовыми архивами, то достаточно удалить строку **ranlib $@**. Рассмотрим на примерах работу программы при различных исходных условиях.

1) Библиотеки нет, структура неизвестна

```
make -s
cc  -c m1.c
cc  -c m2.c
cc  -c m3.c
```



```
as - -o m4.o m4.s
cc  -c m5.c
cc  -c m6.c
as - -o m7.o m7.s
cmp: не могу открыть topology
Структура mylib изменилась.
Библиотека mylib готова.
```

2) Библиотека **mylib** имеется, структура остается без изменений, модифицируется файл **x.h**

```
touch x.h
make -s
cc  -c m1.c
cc  -c m2.c
cc  -c m3.c
cc  -c m6.c
Библиотека mylib готова.
```

3) Меняется содержимое библиотечного модуля **m5.c**. Меняется структура библиотеки: модуль **m5** вызывает теперь модуль **m1**

```
make -s
cc  -c m5.c
Структура mylib изменилась.
Библиотека mylib готова.
```

В файле **topology** теперь новая структура библиотеки

m4.o m2.o m5.o m1.o m6.o m7.o m3.o

4) Добавляется новый модуль в библиотеку. Придется изменить строку **SRC** в **Make**-программе. Имя модуля **m8.c** , вызывает он модуль **m5.c**

```
cc  -c m8.c
list topology различны: char 12, line 3
Структура mylib изменилась.
Библиотека mylib готова.
```

В файле **topology** теперь новая структура библиотеки

m4.o m2.o m8.o m5.o m1.o m6.o m7.o m3.o

5) Изменим модуль **m1**.c так, чтобы он вызывал модуль **m2**.c, а модуль **m2**.c вызывал **m1**.c, т.е. получается зацикленная взаимная зависимость библиотечных модулей **m1** и **m2**

```
make -s
cc  -c m1.c
tsort: зацикленная зависимость
tsort: m2.o
tsort: m1.o
Структура mylib изменилась.
Библиотека mylib готова.
```

Команда **tsort** вывела сообщение об ошибке в структуре библиотеки. Библиотека собрана, но пользоваться ею нельзя, необходимо исправить структуру модуля. Удалять файл **mylib** не нужно, так как он содержит все объектные модули, которые понадобятся для новой сборки.

3.8. Особенности программирования на языке Make

Всюду в примерах **Make**-программа размещалась в одном **Makefile**. Существует возможность разместить ее в файле с другим именем и при вызове интерпретатора **make** указать это имя

make -f имя_файла

Иногда возникает необходимость использовать несколько **Make**-файлов, образующих одну **Make**-программу, тогда при вызове **make** можно указать

make -f имя_файла1 **-f** имя_файла2 и т.д.

Указанные файлы составят текст одной **Make**-программы в том порядке, в котором они указаны в командной строке.

Внутри одной **Make**-программы можно вызывать **make** на выполнение другой **Make**-программы, например:

```
LLL: a b c d
     make -k -f имя_Make-файла $?
```

В том случае, если эта командная строка не может быть нормально выполнена, ключ **-k** указывает на необходимость продолжить выполнение других разделов **Make**-программы, которые не зависят от цели данного правила.

Если в этом примере не указать имя файла, в котором размещена **Make**-программа, то автоматически будет выполняться **Makefile** и будет иметь место рекурсивный вызов на выполнение одной программы.

Есть некоторые особенности при использовании макропеременных. Допустим, в **Make**-программе указана строка

```
SRC = a1.c a2.c a3.c      # комментарий
```

Между **a3**.c и символом **#** 9 пробелов, они будут передаваться всюду, где будет использовано значение макропеременной **SRC**.

Предопределенные макропеременные "**<**" и "*****" в правилах без суффиксов не определены, и их использование может привести к непредсказуемым результатам.

Все, что указано за символом табуляция в строке действий передается на выполнение **shell**. Однако идущие за символом табуляция символы "**-**" и "**@**" обрабатываются **make**. Интерпретатор **make** оптимизирует скорость выполнения действий. Если строка действий - простая команда системы, она выполняется без порождения процесса **shell**. По этой причине, например, такая строка вызовет состояние ошибки

```
'табуляция'#cat file
```

Действительно, как бы выполнялась строка "**exec(# cat file);**" в Си-программе?

Если в списке зависимостей отсутствуют имена подцелей, можно использовать сокращенную форму записи правила с одним действием. Оно имеет вид:

```
имя_цели:[:];  одна_строка_действия [# комментарий]
```

символ "**;**" обязателен.

Особую осторожность необходимо соблюдать при указании в **Make**-программе имени цели, которая не является файлом. В этом случае программист должен учитывать, что он сознательно исключает возможность использования этой цели при реконструкциях, так как она не связана соотношениями времен создания (модификации) с другими объектами **Make**-программы. Это препятствие можно обойти, создавая ЛОЖНЫЙ файл-цель, например:

```
print: f1 f2 f3
       print $?
       touch print
```

```
.DEFAULT:
        touch print
```

В рабочем каталоге создан пустой файл с именем **print**. Теперь выводиться на печать будут только те файлы, которые требуется распечатать как изменившиеся. Правило **DEFAULT** записано на тот случай, когда файл **print** отсутствует.

Команду **touch** можно использовать, когда необходимо разрушить динамическую структуру связей между файлами. Надо учитывать, что при этом **make** будет реконструировать все файлы заново.

Ниже перечислены все ключи интерпретатора **make** и их действие:

-d отладочный режим, в котором выводится дополнительная информация о выполнении **Make**-программы.

-f следующий параметр является именем **Make**-файла. По умолчанию ищется **Makefile** или **makefile**. Если имеются оба, то выполняется **Makefile**. В командной строке можно указать несколько ключей "**-f**" и параметров.

-i режим игнорирования кодов завершения команд не равных нулю. Эквивалентно директиве **IGNORE**.

-k если код завершения команды не равен нулю, прекратить выполнение текущего правила и перейти к выполнению других разделов, не зависящих от файла-цели этого правила.

-n вывести, но не выполнять строки действий **Make**-прсграммы.

-p вывести полную информацию о структуре **Make**-программы.

-q получить информацию о необходимости реконструкции цели. Если реконструкция указанной цели не требуется, возвращается **-1**, иначе **0**.

-r отменяет предопределенную обработку правил с суффиксами, предопределенные макропеременные и суффиксы.

-s отменить вывод выполняемых строк. Эквивалентно директиве **SILENT**.

-S прервать выполнение **Make**-программы при ошибочном завершении какой-либо команды.

-t уничтожить сложившуюся структуру динамических (зависящих от времени) связей между файлами.

3.9. Автоматизация программирования Make-программ

Для создания новой **Make**-программы можно иметь файл-шаблон, добавляя в него необходимые строки мы получим готовую к использованию программу. Такой принцип реализован в программе **mkmf**. Рассмотрим ее работу на примере. Пусть имеются исходные файлы **f.h**, **f1.c**, **f2.c** и **f3.c**, из которых необходимо получить файл **a.out**:


```
/*
** файл f.h
*/
# include       <stdio.h>
# include       <ctype.h>
# include       <time.h>

/*
** файл f1.c
*/
# include       "f.h"

main(ac, av)
int ac;
char **av;
{
        f1(); f2(); f3();

printf("Результат выполнения программы example.\n");
}
f1(){
        return;
}

/*
** файл f2.c
*/
# include       "f.h"

int f2(){
        return( a2());
}
# include       <stat.h>
char *a2(){
        return;
}

/*
** файл f3.c
*/
# include       "f.h"
```

```
int f3(){
        return( nn());
}

char *nn(){
        return;
}
```

Пусть все эти файлы размещены в одном каталоге. Выполним команду **mkmf**. В результате ее выполнения будет создан **Makefile** с программой сборки файла **a.out**:

```
DEST        =

EXTHDRS     = /usr/include/ctype.h \
              /usr/include/stat.h  \
              /usr/include/stdio.h \
              /usr/include/time.h

HDRS        = f.h

LDFLAGS     =

LIBS        =

LINKER      = cc

MAKEFILE    = Makefile

OBJS        = f1.o f2.o f3.o

PRINT       = pr

PROGRAM     = a.out

SRCS        = f1.c f2.c f3.c

all:        $(PROGRAM)

$(PROGRAM): $(OBJS) $(LIBS)
        @echo -n "Сборка $(PROGRAM) ..."
        @$(LINKER) $(LDFLAGS) $(OBJS) \
```



```
                $(LIBS) -o $(PROGRAM)
        @echo "готово."

clean:;     @rm -f $(OBJS)

depend:;    @mkmf -f $(MAKEFILE) \
    PROGRAM=$(PROGRAM) DEST=$(DEST)

index:;     @ctags -wx $(HDRS) $(SRCS)

install:    $(PROGRAM)
        @echo Установка $(PROGRAM) в $(DEST)
        @install -s $(PROGRAM) $(DEST)

print:;     @$(PRINT) $(HDRS) $(SRCS)

program:;   $(PROGRAM)

tags:       $(HDRS) $(SRCS)
        @ctags $(HDRS) $(SRCS)

update:     $(DEST)/$(PROGRAM)

$(DEST)/$(PROGRAM): $(SRCS) $(LIBS) \
                $(HDRS) $(EXTHDRS)
        @make -f $(MAKEFILE) \
                DEST=$(DEST) install

###
f1.o: f.h /usr/include/stdio.h \
          /usr/include/ctype.h \
          /usr/include/time.h

f2.o: f.h /usr/include/stdio.h \
          /usr/include/ctype.h \
          /usr/include/time.h  \
          /usr/include/stat.h

f3.o: f.h /usr/include/stdio.h \
          /usr/include/ctype.h \
          /usr/include/time.h
```

Программой **mkmf** в качестве исходного файла-шаблона использован стандартный файл **/usr/new/lib/p.Makefile**, но можно указать для использования и любой другой.

Программа **mkmf** работает следующим образом: сначала выбираются и вносятся в файл-шаблон имена всех исходных файлов рабочего каталога, далее определяется от каких **include**-файлов зависят исходные файлы, формируются правила и записываются в файл-шаблон. Для обозначения исходных файлов используются правила с суффиксами.

Makefile можно редактировать, изменять значения макропеременных. При этом, если повторить вызов программы **mkmf**, в нем появятся только те изменения, которые необходимы для сборки с учетом изменений в исходных текстах.

В **Makefile**, полученном из стандартного файла-шаблона, определены следующие макропеременные:

CFLAGS
: ключи Си-компилятора;

DEST
: каталог, в котором будет размещен результат;

EXTHDRS
: перечень полных имен **include**-файлов;

HDRS
: перечень имен **include**-файлов, размещенных в рабочем каталоге;

LIBS
: список объектных библиотек для сборки программы;

MAKEFILE
: имя файла с **Make**-программой;

OBJS
: список объектных файлов, участвующих в сборке программы;

PROGRAM
: имя программы, которую необходимо получить;

SRCS
: список имен всех файлов с исходными текстами;

Значения макропеременных **EXTHDRS**, **HDRS**, **OBJS**, **SRCS** устанавливаются программой **mkmf** и всегда имеют актуальные значения. Остальные макропеременные получают при создании **Makefile** значения по умолчанию, их можно изменять по своему усмотрению.

Рассмотрим правила **Make**-программы, которые можно использовать как самостоятельные входы:

all
: трансляция, сборка и запуск на выполнение полученной программы;

clean
: удаление ненужных файлов;

depend
: изменение структуры **Make**-программы с учетом существующего **Makefile**;

index
: печать индексов функций собираемой программы;

install
: трансляция, сборка и установка программы в указанный каталог;

print
: печать **include**-файлов и текстов программы;

tags
: создание файла **./tags** - ссылок программ, написанных на языках Си, Паскаль и Фортран;

update
: изменение **Makefile**, перегенерация, сборка и установка программы в указанный каталог. С учетом происшедших изменений в текстах исходных файлов будет выполнено только то, что необходимо в данный момент времени.

Пусть имеются файлы **f[123].c**, **f.h** и **Makefile**, заменим в нем значение макропеременной **DEST** на **/usr/tmp** и макропеременной **PROGRAM** - на **example**. Выполним следующую командную строку:

```
% make program install clean
```

получим на экране сообщение

```
cc  -c f1.c
cc  -c f2.c
cc  -c f3.c
Сборка example ... готово.
Результат выполнения программы example
Установка example в /usr/tmp
```

Выполним командную строку

```
% make index
```

получим имена функций и места их определений

```
a2          5 f2.c        char *a2(){
```

```
f1      11 f1.c     f1(){
f2       2 f2.c     int f2(){
f3       2 f3.c     int f3(){
main     5 f1.c      main(ac, av)
nn       6 f3.c      char *nn(){
```

Программа **mkmf** позволяет создавать **Makefile** для сборки библиотеки. Для этого используется файл-шаблон **/usr/new/lib/l.Makefile** и дополнительно вводятся макропеременная **LIBRARY** (имя библиотеки) и правила **extract** (извлечение из библиотеки всех частей в виде отдельных файлов), **library** (трансляция и загрузка библиотеки).

Программист может отказаться от стандартных файлов-шаблонов **/usr/new/lib/[lp].Makefile** и создать свои шаблоны в рабочем каталоге. Файлы-шаблоны должны иметь имена **l.Makefile** и **p.Makefile**.

Глава 4. ЯЗЫК ОБРАБОТКИ СТРУКТУРИРОВАННЫХ ТЕКСТОВ AWK

Язык **AWK** используется для комбинированной обработки символьных и числовых полей в записях. В результате генерируется отчет в запланированной программистом форме. Программы на языке **AWK** можно эффективно использовать как фильтры данных для преобразования вывода одной программы и передачи результата фильтрации на вход другой. В системе ДЕМОС установлен интерпретатор языка **AWK**, который получил название **awk**.

4.1. Принципы работы интерпретатора awk

Любой текст имеет некоторую структуру, в простейшем случае ее элементами являются строки и слова текста. В языке **AWK** [10] текст рассматривается как список записей и полей в них и на этой основе выполняется некоторый определенный программистом алгоритм обработки. Допустим, имеется следующий текст:

```
Сидоров Сидор   Сидорович      1957 г.р. 220 руб сл.
Петров  Петр    Иванович       1962 г.р. 200 руб сл.
Иванов  Михаил  Константинович 1965 г.р. 180 руб раб.
Волков  Леонид  Николаевич     1950 г.р. 280 руб раб.
Семенов Петр    Михайлович     1958 г.р. 210 руб раб.
```

Этот текст структурирован: записи - это строки, поля в строках - слова и числа. В каждой записи содержится по 8 полей, разделяющихся пробелами. Значащим (в качестве разделителя) является только один пробел между полями, остальные игнорируются. Рассмотрим несколько простых программ на языке **AWK**.

П р и м е р 1. **AWK**-программа выводит первые три поля из восьми, порядок полей в выводе изменен и перед каждой строкой печатается символ табуляции

```
{ print( "\t", $2, $3, $1 ); }
```

Оператор **print** выполняется для всех входных записей. После выполнения программы получим:

Сидор Сидорович Сидоров
Петр Иванович Петров
Михаил Константинович Иванов
Леонид Николаевич Волков
Владимир Михайлович Семенов

Как видно из программы, значения полей подставляются следующим образом:

$номер_поля_в_записи

Первому полю соответствует 1. В общем случае номером поля может быть значение выражения. Значением подстановки **$0** является вся запись.

П р и м е р **2**. **AWK**-программа выводит номера строк после табуляции

```
{ print( "\t", NR, $2, $3, $1 ); }
```

После выполнения программы получим:

1 Сидор Сидорович Сидоров
2 Петр Иванович Петров
3 Михаил Константинович Иванов
4 Леонид Николаевич Волков
5 Владимир Михайлович Семенов

Предопределенная переменная **NR** равна номеру обрабатываемой записи. Мы воспользовались ее значением для нумерации строк.

П р и м е р **3**. **AWK**-программа выводит полное число лет на **1988** год каждому лицу из списка

```
{ print("\t", NR, $2, $3, $1, "\t\t", 1988 - $4); }
```

После выполнения программы получим:

1 Сидор Сидорович Сидоров	31
2 Петр Иванович Петров	26
3 Михаил Константинович Иванов	23
4 Леонид Николаевич Волков	38

П р и м е р **4**. **AWK**-программа подсчитывает средний возраст и среднюю заработную плату перечисленных в списке лиц

```
{
    age += 1988   $4;
    pay += $6;
}

END    {
    print ("Средний возраст:\t",  age/NR );
    print ("Средняя зарплата:\t",  pay/NR );
}
```

После выполнения программы получим:

Средний возраст:	29.6
Средняя зарплата:	218

Когда необходимо обеспечить вывод результата по завершению списка записей, используется селектор **END**. Переменные **age** и **pay** определяются автоматически как числа в момент первого использования. Выражения вычисляются для всех входных записей.

П р и м е р **5**. **AWK**-программа подсчитывает средние возраст и заработную плату рабочих и служащих в списке. Для выделения строк со сведениями о рабочих используется шаблон /раб/, о служащих - шаблон /сл/. Шаблоны содержат образцы для поиска в полях записи. Данные выводятся после обработки всех записей

```
/раб/    {
    rage += 1988 - $4;
    rpay += $6;
    r++;
}

/сл/     {
    age += 1988 - $4;
    pay += $6;
    c++;
}
```

```
END    {
    print("\t\tСредний возраст  Средняя зарплата\n");
    print (" Рабочие:\t", rage/r, "\t",    rpay/r );
    print ("Служащие:\t",  age/c, "\t\t",  pay/c );
}
```

После выполнения программы получим:

	Средний возраст	Средняя зарплата
Рабочие:	30.3333	223.333
Служащие:	28.5	210

Программа выполняется следующим образом. Если запись в каком-либо из полей содержит образец, выполняется действие, записанное в фигурных скобках рядом с соответствующим шаблоном, иначе действие не выполняется. Действия, указанные после **END**, выполняются по концу списка записей. Шаблоны в примере используются как селекторы входных записей: если в четвертом примере действия были выполнены для всех входных записей, то в этом - только для отобранных по образцам, указанным в шаблонах. При этом **END** используется как селектор специального вида: он определяет список операторов **AWK**-программы, который должен выполниться после завершения входного потока записей.

Пример 6. **AWK**-программа вычисляет уровни заработной платы

```
BEGIN {
    Min = 1000;
    Max = 0;
}

{
    if ( $6 < Min ) {
        Min = $6;
        smin = $1 " " $2 " " $3;
    }

    if ( $6 > Max ) {
        Max = $6;
        smax = $1 " " $2 " " $3;
    }
}
```



```
END {
    print( "\t\tУровни зарплаты\n" );
    print( " Минимальный: ", Min, "   (",smin,")" );
    print( "Максимальный: ", Max, "   (",smax,")" );
}
```

После выполнения получим:

Уровни зарплаты

Минимальный: 180 (Иванов Михаил Константинович)

Максимальный: 280 (Волков Леонид Николаевич)

В этой программе три раздела. Первый раздел используется для установки начальных значений переменных **Max** и **Min** еще до чтения записей из списка. Специальный селектор **BEGIN** определяет список операторов **AWK**-программы, который должен выполниться до анализа первой записи из входного потока. Во втором разделе осуществляется собственно обработка записей. Операторы этого раздела программы выполняются для всех входных записей, так как селектор не указан. Третий раздел выполняется когда завершается список записей (селектор **END**). В строке

```
smin = $1 " " $2 " " $3;
```

переменной **smin** присваиваются значения первых трех полей записи, конкатенация которых вместе с пробелами, указанными в кавычках, образует строку. Таким образом значением переменной **smin** будет строка символов типа "фамилия Имя Отчество".

Существует несколько способов вызова интерпретатора **awk**. **AWK**-программа в файле:

awk -f имя_файла_с_**AWK**-программой входной_файл ...

По умолчанию разделителем записеи является символ новой строки, разделителем полей - символ пробел и/или табуляции. Символы-разделители можно явно определить в программе. Символ-разделитель записей можно определить и в командной строке. Вызов **awk** с указанием символа-разделителя:

awk -Fразделитель **-f** файл_**AWK**-программа входной_файл ...

Часто **AWK**-программы настолько коротки, что их целесообразно указывать непосредственно в командной строке, а не в файле. Вызов **awk** с программой в командной строке:

awk -Fразделитель **AWK**-программа´ входной_файл ...

awk ´**AWK**-программа´ входной_файл ...

Интерпретатор **awk**, как и большинство других программ системы, позволяет входной_файл заменить на стандартный ввод.

awk -**f** имя_файла_с_**AWK**-программой -

awk -Fразделитель ´**AWK**-программа´ -

awk ´**AWK**-программа´ -

Если не указано другое, результат выполнения **AWK**-программы печатается на экране дисплея.

4.2. Переменные, выражения и присваивания в AWK-программах

В языке **AWK** выделяют две группы переменных: предопределенные и декларированные в программе. Предопределенные переменные доступны для подстановок и изменений в программе, их исходные значения устанавливаются интерпретатором **awk** в процессе запуска и выполнения **AWK**-программы. К предопределенным переменным относятся:

NR номер текущей записи;
NF число полей в текущей записи;
RS разделитель записей на вводе (символ);
FS разделитель полей записи на вводе (символ);
ORS разделитель записей на выводе **AWK**-программы (символ);
OFS разделитель полей записи на выводе (символ);
OFMT формат вывода чисел;
FILENAME имя входного файла (строка).

По умолчанию имеют место следующие значения предопределенных переменных:

```
RS   =  "\0";
FS   =  ´пробел(ы) и/или табуляция´;
OFS  =  FS;
```



```
ORS  =  RS;
OFMT =  "%.6g";
```

Предопределенным переменным **RS**, **FS**, **ORS**, **OFS**, **OFMT** можно присваивать значения в **AWK**-программе.

В языке **AWK** отсутствуют декларация и явная инициализация переменной любого типа. Всякой переменной до ее первого использования присваивается значение "\0" - пустая строка. Применяются следующие типы переменных:

позиционная переменная;

число с плавающей точкой;

строка символов;

массив.

Позиционная переменная определяет поле записи, содержимое которого может быть отнесено к типам "строка" или "число_с_точкой" и используется в виде

$номер_поля_записи

$(выражение)

Номер_поля_записи может быть значением выражения. Значением позиционной переменной **$0** является вся запись.

Интерпретатор **awk** рассматривает переменную как строковую до того момента, когда необходимо выполнить некоторую операцию над ее значением. В зависимости от контекста тип значения переменной остается либо строковым, либо преобразуется к типу число_с_точкой. В двусмысленных случаях переменные рассматриваются как строковые. Строки, которые не могут быть интерпретированы как числа, в числовом контексте будут иметь числовое значение НОЛЬ. Устранить двусмысленность можно явным указанием типа переменной при присваивании ей значения, например:

```
name =  1 ;     # присвоено значение 1.0

name = "1";     # присвоено значение строки "1"
```

При интерпретации выражений существенную роль играет контекст,

например:

```
name = 3 + 2 ;

name = 3 + "2" ;

name = "3" + "2" ;

name = 3 + 2 + "яблоко груша апельсин";

name = "яблоко" + "груша";
```

В этом примере в первых четырех случаях **name** равно **5.0**, в пятом - **0**.

Массив не декларируется, он начинает существовать в момент первого использования. Индексы в массиве могут иметь любое ненулевое значение, включая нечисловые строки, это позволяет использовать ассоциативные массивы. Например, в приведенной ниже **AWK**-программе будет подсчитано число упоминаний об автомобилях различных марок во входном тексте:

```
/ЗИЛ/ { Автомобили["ЗИЛ"]++; }

/ГАЗ/ { Автомобили["ГАЗ"]++; }

/ВАЗ/ { Автомобили["ВАЗ"]++; }

END {
    print("ЗИЛ : ", Автомобили["ЗИЛ"]);
    print("ГАЗ : ", Автомобили["ГАЗ"]);
    print("ВАЗ : ", Автомобили["ВАЗ"]);
}
```

Массивы можно использовать для организации такого алгоритма обработки данных, в котором требуется многократный просмотр входного потока записей. Например, если не заботиться о размерах оперативной памяти, то можно весь входной файл записать в виде массива записей и по завершению входного потока приступить собственно к обработке:

```
{
    Массив_записей[NR] = $0
}
```



```
END {
    ...
    программа обработки массива
    ...
}
```

В качестве имени (значения) индекса массива можно использовать выражение, например:

```
name["2" * $3]
```

В языке **AWK** используются операторы присваивания

```
=   +=   -=   *=   /=   %=
```

и арифметические операции

```
+   -   *   /   %   ++   --
```

Они имеют тот же смысл, что и в языке программирования Си.

Имеются некоторые особенности выполнения операций сравнения

```
<   <=   ==   !=   >=   >
```

Если оба операнда интерпретируются как числа, то выполняется сравнение чисел. Если один из операндов является строкой символов, а другой - числом, то выполняется сравнение строк. Сравнение строк заключается в попарном сравнении внутренних кодов символов строк до первого неравенства кодов или до завершения одной из строк. Рассмотрим пример:

```
{
 if( $1 <  $2 )
   print(NR": $1 =", $1, "; $2 =", $2, "; $1 < $2");

 if( $1 == $2 )
   print(NR": $1 =", $1, "; $2 =", $2, "; $1 == $2");

 if( $1 >  $2 )
   print(NR": $1 =", $1, "; $2 =", $2, "; $1 > $2");
}
```

Допустим, имеется следующий входной текст:

```
2.01    2.02
2.01    abc
a       b
aa      b
aa      ab
aa      ba
abc     ab
ab      abc
ef      abc
```

В результате выполнения программы получим:

```
1: $1 = 2.01; $2 = 2.02; $1 <  $2
2: $1 = 2.01; $2 = abc ; $1 <  $2
3: $1 = a   ; $2 = b   ; $1 <  $2
4: $1 = aa  ; $2 = b   ; $1 <  $2
5: $1 = aa  ; $2 = ab  ; $1 <  $2
6: $1 = aa  ; $2 = ba  ; $1 <  $2
7: $1 = abc ; $2 = ab  ; $1 >  $2
8: $1 = ab  ; $2 = abc ; $1 == $2
9: $1 = ef  ; $2 = abc ; $1 >  $2
```

В AWK-программах можно использовать следующие логические операции:

! (не) || (или) && (и)

Как обычно, значением выражения, содержащего операции отношения и/или логические операции, являются: истина (не ноль) или ложь (ноль). Приоритеты операций в выражениях аналогичны установленным в языке Си. Для управления порядком выполнения операций в выражении используются круглые скобки.

В языке AWK имеется операция, не предусмотренная в Си, - это операция "пробел", которая используется для конкатенации переменных, значения которых интерпретируются как строковые

```
name = "яблоко " "и груша";
```

В этом случае значением переменной **name** будет строка вида

"яблоко и груша"

Вместо символа пробел можно использовать символ табуляции. При использовании операции "пробел" учитывается контекст, например:

```
$1 = "яблоко"

$2 = "и"

$3 = "груша"

name1 = $3 $2 $1;              # 1
name2 = $3" "$2" "$1;          # 2
name3 = "Красное " $1;         # 3
name4 = 1 2 3 4 5 6 7 8 9;     # 4
name5 = 123          789;      # 5
name6 = $3$2$1;                # 6
```

значением переменной **name1** будет строка:

"грушаяблоко"

Значением переменной **name2** будет строка:

"груша и яблоко"

Значением переменной **name3** будет строка:

"Красное яблоко"

Значением переменной **name4** будет строка:

"123456789"

Значением переменной **name5** будет строка:

"123789"

Из примера 5 видно, что в качестве знака операции "пробел" существенно наличие лишь одного пробела между операндами, остальные игнорируются. Значением переменной **name6** будет строка вида

"грушаяблоко"

Однако синтаксис, использованный в 6 строке примера, сомнителен и не стоит полагаться на "мудрость" интерпретатора **awk**.

Позиционные переменные можно использовать в выражениях любого вида, им можно присваивать новые значения. Рассмотрим несколько примеров:

```
$3  =  $1 " " $2;
$3 +=  $1;
$3  =  $3 $3 $3;
$3  =  "";
$0  =  $3;
```

В первом случае позиционной переменной **$3** присваивается строка, полученная в результате конкатенации значения позиционной переменной **$1**, пробела и значения позиционной переменной **$2**. В третьем случае выполняется конкатенация собственного значения переменной **$3**, в четвертом - переменной **$3** присваивается значениие пустой строки, в пятом - значение переменной **$0** (вся запись) заменяется значением поля **3**.

4.3. Структура AWK-программы

AWK-программа состоит из списка правил вида:

селектор1 { действие }
...
селекторN { действие }

Открывающая фигурная скобка должна стоять в той же строке, где селектор. В любом месте программы можно ввести комментарий, он печатается от символа # до конца строки.

Каждое правило выполняется для каждой записи из входного потока. Селектор используется для того, чтобы выделить запись, над которой будет выполнено действие соответствующего правила. Если запись не выделена ни одним из селекторов, она игнорируется и не выводится на стандартный вывод. Если запись выделена селектором, выполняется действие соответствующего правила.Если некоторую запись выделяют несколько селекторов, над ней выполняются действия соответствующих правил.

В правиле может отсутствовать селектор, тогда действие этого правила будет выполнено для всех без исключения входных записей. В правиле может отсутствовать действие, тогда все выделенные селектором записи будут направлены на стандартный вывод без изменений.

Определены два правила специального вида:

BEGIN { действие }
...
список других правил
...
END { действие }

Правило с селектором **BEGIN** выполняется до чтения первой входной записи, с селектором **END** - после чтения последней записи. Правило с селектором **BEGIN** должно быть первым в списке правил, с селектором **END** - последним. Возможно такое использование этих правил:

BEGIN { действие }
...
список других правил

или

список других правил
...
END { действие }

Действие в правиле может содержать список операторов и управляющих конструкций. Оператор должен заканчиваться символом ";" или символом новой строки, или закрывающей скобкой.

Переменную можно использовать в любом правиле **AWK**-программы, начиная с места, где она определена. Рассмотрим пример, в котором демонстрируются особенности выполнения правил **AWK**-программы и использования переменных:

```
# Программа  демонстрирует  работу
# правил  различного  вида  и область
# действия переменных

# Правило 1 выполняется
# для всех записей
```

```
    { print("Запись номер:", NR); }

# Правило 2 выполняется только для
# записей, где обнаружен образец aaa

/aaa/ {
    print("Правило 2:");
    print("      Вход:", $0);
    $1 = $1 $2;
    $2 = "***";
    A = $2;
    print("Результат:", $0, "A =", A);
}

# Правило 3 выполняется только для
# записей, где обнаружен образец ddd

/ddd/ {
    print("Правило 3:");
    print("      Вход:", $0);
    $1 = $1 $3;
    $2 = "&&&";
    A = $2;
    print("Результат:", $0, "A =", A);
}

# Правило 4 выполняется для всех записей

{
    print("Правило 4:", $0, "A =", A,"0);
}
```

Допустим, на вход этой программе передаются следующие три записи:

```
eee fff
ddd eee fff
aaa bbb ccc ddd eee
```

тогда после выполнения программы получим:

```
Запись номер: 1
Правило 4: eee fff A =
```



```
Запись номер: 2
Правило 3:
      Вход: ddd eee fff
Результат: dddfff &&& fff A = &&&
Правило 4: dddfff &&& fff A = &&&

Запись номер: 3
Правило 2:
      Вход: aaa bbb ccc ddd eee
Результат: aaabbb *** ccc ddd eee A = ***
Правило 3:
      Вход: aaabbb *** ccc ddd eee
Результат: aaabbbccc &&& ccc ddd eee A = &&&
Правило 4: aaabbbccc &&& ccc ddd eee A = &&&
```

4.4. Селекторы

Селектор указывается, чтобы определить, будет ли выполняться действие в правиле. В качестве селектора может быть использовано любое выражение, шаблон и произвольная их комбинация. Рассмотрим несколько примеров использования выражений в селекторах:

$1 != $2 || $1 > 128

выбрать запись, в которой либо первые два поля различны, либо содержимое первого поля больше 128;

$1 % $2 == 1

выбрать запись, в которой остаток от деления полей равен 1;

NF % 2 == 0 || name < 2.2

выбрать запись, либо содержащую четное число полей, либо если переменная **name** меньше 2.2;

$1 == "Иванов И.И."

выбрать запись, в которой первое поле относится к Иванову И.И.;

$1 >= "M" && $1 != "Москва"

выбрать запись, первое поле которой начинается с буквы М и далее по алфавиту, но не является словом Москва.

Шаблон используется для формирования одного или большего числа образцов в селекторе. При сканировании входной записи осуществляется поиск цепочки символов, тождественной образцу. В простейшем случае селектор с шаблоном выглядит следующим образом:

/образец/

В символах / указан образец, который будет использован для поиска. Существенно, что любой символ, в том числе пробел, указанный внутри пары символов /, является частью образца.

Если необходимо, чтобы соответствие образцу определялось в конкретном поле записи, используются операторы соответствия (~ и !~)

$номер_поля ~ шаблон

если при просмотре указанной позиционной переменной обнаруживается цепочка символов, тождественная образцу в шаблоне (оператор ~), выполняется действие правила.

$номер_поля !~ шаблон

если при просмотре указанной позиционной переменной не обнаруживается цепочка символов, тождественная образцу в шаблоне (оператор !~), выполняется действие правила.

В общем случае шаблон может формировать множество образцов и/или указывать, в каком месте записи необходимо искать соответствие входной цепочки символов образцу. При необходимости используются так называемые регулярные выражения, в этом случае шаблон выглядит следующим образом:

/регулярное_выражение/

В результате разбора регулярного выражения интерпретатором **awk** строится и выполняется алгоритм поиска одного или большего числа образцов во входной записи.

Регулярные выражения в шаблонах селекторов **AWK** аналогичны подобным в **Lex**, редакторе **ed** и в команде **grep**. Регулярное выражение формируется как композиция цепочек символов (и/или диапазонов символов) и операторов. Операторы в регулярных выражениях указываются в виде символов-операторов. Чтобы отнести действие символа-оператора к отдельному фрагменту регулярного выражения, используются круглые скобки. Чтобы отменить специальное значение символа-оператора, его экранируют символом \.

Для записи регулярных выражений употребляются следующие символы-операторы:

^ от начала;

$ на конце;

. любой символ;

символ

данный символ, если он не символ-оператор;

\символ

использовать символ-оператор как обычный символ;

[строка]

любой из символов данной строки;

[буква1-буква2]

любая буква из данного лексикографически упорядоченного диапазона букв;

[цифра1-цифра2]

любая цифра из данного диапазона цифр;

рег_выражение*

0 или более вхождений регулярного выражения;

рег_выражение+

1 или более вхождений регулярного выражения;

рег_выражение?

0 или 1 вхождение регулярного выражения;

рег_выражение1 рег_выражение2

последовательное вхождение рег_выражение1 и рег_выражение2;

рег_выражение1|рег_выражение2

вхождение рег_выражение1 или рег_выражение2;

Рассмотрим несколько примеров использования регулярных выражений:

/^Иванов/

выделить запись, начинающуюся цепочкой символов "Иванов". Таким образом, будут выделены случаи типа "Иванову", "Ивановой", ... ;

$3 ~ /^Иванов/

выделить запись, в которой третье поле начинается цепочкой символов "Иванов";

/([abc][ABC])$/

выделить запись, предпоследним символом которой является одна из букв abc и последним - одна из букв ABC;

/[0-9]+/

выделить запись, содержащую не менее одной цифры;

$3 !~ /(Сидор)|(Петр)/

не выделять запись, содержащую в третьем поле что-либо о Сидорах или Петрах;

Ниже приведен пример **AWK**-программы, печатающей имена регистрационных каталогов и имена всех пользователей системы, которыми не установлен пароль:

```
BEGIN {
        FS = ":";
```

```
        print("Имя\tКаталог");
}

$2 !~ /(([0-9])|([a-z])|([A-Z]))+/  {
        print( $1, "\t", $6);
}
```

В первом правиле (селектор **BEGIN**) меняется разделитель полей записи с пробела на двоеточие (такова структура записей в парольном файле **/etc/passwd** операционной системы ДЕМОС). Во втором поле записи парольного файла содержится зашифрованный пароль - обычно это комбинация цифр и букв. Если пароль не установлен, то второе поле записи пусто. Этот факт использован для формирования селектора - второе поле не должно содержать цифр и букв. Селектор выделяет второе поле записи и проверяет наличие не менее одного символа в этом поле. Если поле пусто, выполняется действие, которое заключается в печати имени пользователя (первое поле) и имени регистрационного каталога пользователя (шестое поле).

Иногда необходимо определить диапазон записей, для которых выполняется действие. Например, необходимо вывести на печать записи с номерами от **10** до **20** включительно. Или, допустим, вывести на печать поле номер **6** каждой записи, начиная с той, в которой второе поле "Петр", до той, в которой пятое поле "Сидор". Для определения диапазона записей в селекторах используется операция запятая. До запятой указывается селектор, выделяющий первую запись в диапазоне, после запятой - селектор, выделяющий последнюю запись в диапазоне. Таким образом, мы имеем дело с составным селектором. Для всех записей диапазона выполняется действие правила с составным селектором.

Рассмотрим пример. Допустим, имеется следующий файл:

```
sss    поле2 поле3 поле4  1
поле1  sss   поле3 поле4  2
поле1  поле2 sss   поле4  3
поле1  поле2 поле3 sss    4
ttt    поле2 поле3 поле4  5
поле1  ttt   поле3 поле4  6
поле1  поле2 ttt   поле4  7
поле1  поле2 поле3 ttt    8
```

Допустим, необходимо вывести на печать диапазон записей. Открывает этот диапазон запись, второе поле которой "sss", и закрывает запись, третье поле которой "ttt". Тогда программа выглядит следующим образом:


```
$2 ~ /sss/, $3 ~ /ttt/    {
        print( $0 );
}
```

В результате выполнения получим:

```
поле1 sss    поле3 поле4  2
поле1 поле2 sss    поле4  3
поле1 поле2 поле3 sss     4
ttt    поле2 поле3 поле4  5
поле1 ttt    поле3 поле4  6
поле1 поле2 ttt    поле4  7
```

В одной программе можно указать несколько правил с составными селекторами. При этом если выделенные диапазоны перекрываются, то каждая выделенная запись будет обрабатываться несколькими правилами. Например, для того же исходного файла используется следующая программа обработки:

```
$2 ~ /sss/, $3 ~ /ttt/ {
        print($0);
}

$1 ~ /sss/, NR == 5    {
        print($0, "*");
}

NR == 6, NR == 8       {
        print( $0, "<-" );
}
```

В результате выполнения получим:

```
sss    поле2 поле3 поле4  1 *
поле1 sss    поле3 поле4  2
поле1 sss    поле3 поле4  2 *
поле1 поле2 sss    поле4  3
поле1 поле2 sss    поле4  3 *
поле1 поле2 поле3 sss     4
поле1 поле2 поле3 sss     4 *
ttt    поле2 поле3 поле4  5
ttt    поле2 поле3 поле4  5 *
```

```
поле1 ttt    поле3 поле4  6
поле1 ttt    поле3 поле4  6 <-
поле1 поле2 ttt    поле4  7
поле1 поле2 ttt    поле4  7 <-
поле1 поле2 поле3 ttt     8 <-
```

Чтобы устранить эффект пересечения диапазонов выделенных записей, там, где это необходимо, можно использовать оператор **next**. Этот оператор прекращает обработку текущей записи, управление передается на начало программы и начинается разбор следующей записи. Теперь программа будет иметь вид:

```
$2 ~ /sss/, $3 ~ /ttt/ {
        print($0);
        next;
}

$1 ~ /sss/, NR == 5    {
        print($0, "*");
        next;
}

NR == 6, NR == 8       {
        print( $0, "<-" );
}
```

В результате выполнения программы получим:

```
sss   поле2 поле3 поле4  1 *
поле1 sss   поле3 поле4  2
поле1 поле2 sss   поле4  3
поле1 поле2 поле3 sss    4
ttt   поле2 поле3 поле4  5
поле1 ttt   поле3 поле4  6
поле1 поле2 ttt   поле4  7
поле1 поле2 поле3 ttt    8 *
```

Из примера видно, что в исходном списке не нашлось ни одной записи, которая была бы обработана всеми правилами и действие третьего правила программы не выполнялось вообще.

Если в результате выполнения правила с составным селектором выделено начало диапазона записей, но не выделен его конец, действие этого

правила выполняется для всех записей до конца ввода. Если же не обнаружена запись, открывающая диапазон записей, то действие правила с составным селектором не выполняется.

4.5. Действия

Действия в правилах **AWK**-программы определяют алгоритм обработки выделенных селектором записей. Для записи алгоритма используются присваивания, выражения, операторы управления, операторы вывода, встроенные функции.

Выше было показано, что действие в правиле записывается как блок (в смысле языка программирования Си). Фигурная скобка, открывающая блок, должна указываться в той же строке, что и селектор, закрывающая - по завершению блока. В общем случае блок может быть пустым, тогда, как это было сказано выше, все записи, выделенные селектором, передаются на стандартный вывод без преобразований.

К числу операторов управления относятся:

exit
: завершить выполнение программы;

next
: перейти к чтению следующей записи. Управление передается на первое правило **AWK**-программы (если имеется правило с селектором **BEGIN**, то на следующее за ним);

break
: прерывает выполнение охватывающего цикла. Управление передается на оператор, следующий за циклом;

continue
: переход к следующей итерации цикла;

if(выражение) { блок_1 } **else** { блок_2 }
: если значение выражения - истина, выполняются операторы блока_1, иначе операторы блока_2. Часть **else** можно опустить. Если блок_1 или блок_2 содержат по одному оператору, фигурные скобки можно не указывать;

while(выражение) { блок }
: операторы блока выполняются, пока значение выражения - истина. Если в блоке только один оператор, фигурные скобки можно не указывать;

for(выражение_1; выражение_2; выражение_3) { блок }
: если значение выражения_2 - истина, выполняются операторы блока. Выражение_1 вычисляется перед первой итерацией цикла, выражение_3 вычисляется на каждой итерации цикла. Если блок содержит один оператор, фигурные скобки можно не указывать.

for(индекс **in** имя_массива) { блок }

для каждого значения индекса массива выполняются операторы блока. Значение индекса формируется автоматически на каждой итерации цикла и равно значению, еще не использованному в цикле. Если используется ассоциативный массив, индекс формируется в лексикографическом порядке. Если в блоке происходит добавление элементов массива, результат выполнения цикла непредсказуем. Если в блоке изменяется значение индекса, результат выполнения цикла непредсказуем. Вместо индекса и/или имени массива можно указать выражение, значение которого интерпретируется как индекс и/или имя массива.

В качестве условных выражений можно использовать любые из описанных выше. В выражениях можно применять шаблоны, операторы ~ и !~. Рассмотрим пример:

```
/aaa/ {
        if( $3 !~ /fff/ )
                print( $0 );
}
```

В записи, выделенной по селектору **/aaa/**, проверяется соответствие содержимого поля **$3** шаблону **/fff/**. Если соответствие не обнаружено, печатается вся запись, иначе оператор **print** не выполняется.

Теперь рассмотрим пример использования цикла **for** по индексу в ассоциативном массиве. Допустим, имеется список записей

```
aaa aaa ddd ccc
ccc ddd
bbb ddd ddd
ccc
```

и пусть выполняется программа

```
/bbb/   { m["bbb"]++; }

/ccc/   { m["ccc"]++; }

/aaa/   { m["aaa"]++; }

/ddd/   { m["ddd"]++; }

END     { for( i in m )
                print("m["i"] =", m[i]);
}
```


В каждом из первых четырех правил селекторами выделяются записи и подсчитывается число таких записей в ассоциативном массиве с именем **m**. Цикл **for** выполняется по завершению списка входных записей. В результате выполнения программы получим:

```
m[aaa] = 1
m[bbb] = 1
m[ccc] = 3
m[ddd] = 3
```

Значением каждого элемента массива является число выделенных селекторами записей. В результате выполнения цикла по индексу в ассоциативном массиве получен вывод значений элементов массива в лексикографическом порядке значений индекса.

Ниже приведен пример программы, действия которой содержат примеры использования основных управляющих конструкций. Допустим, имеется следующий текст:

```
aaa, aaa, aaa  aaa  aaa.
aaa  aaa, aaa, aaa  aaa.
aaa  aaa  aaa, aaa  aaa.
aaa  aaa  aaa  aaa, aaa.
aaa; aaa  aaa  aaa: aaa.
aaa  aaa; aaa  aaa  aaa.
aaa  aaa  aaa; aaa; aaa.
aaa  aaa: aaa  aaa; aaa.
aaa: aaa  aaa  aaa  aaa.
aaa  aaa  aaa: aaa: aaa.
```

Требуется получить некоторую статистику о тексте:

```
# Программа вычисляет статистические
# характеристики текста.
# Разделитель записей точка.
# Разделитель полей пробел.
# Вывод результатов осуществляется
# после завершения входного текста.
```

```
BEGIN {
        # выделение и инициализация
        # переменных

        RS = ".";  # разделитель записей
        Nw = 0;    # число слов
        Nb = 0;    # длина строк
        Np = 0;    # число запятых
        Nd = 0;    # число двоеточий
        Nt = 0;    # число точек с запятой
}

{
 for( i = 1; i <= NF; i++ ){
        if( $i ~ /,$/ ) {
                Np++;
                Nb--;
        }

        # Nb--; не учитывать в длине
        # слова знак препинания

        if( $i ~ /:$/ ) {
                Nd++;
                Nb--;
        }

        if( $i ~ /;$/ ) {
                Nt++;
                Nb--;
        }

        Nb  += length( $i ); # длина слова
        Nw++; # увеличить число слов
 }
}

END  {
        print("Число запятых =", Np);
        print("Число двоеточий =", Nd);
        print("Число точек с запятой =", Np);
        print("Число слов =", Nw);
```



```
        print("Число символов в словах =", Nb);
        print("Число предложений =", NR );
        print("Средняя длина предл. =", Nw/NR,"(слов)");
        print("Средняя длина слова =", Nb/Nw);
}
```

Ниже показан результат работы программы:

```
Число запятых = 6
Число двоеточий = 5
Число точек с запятой = 6
Число слов = 50
Число символов в словах = 150
Число предложений = 10
Средняя длина предл. = 5 (слов)
Средняя длина слова = 3
```

4.6. Ввод и вывод данных в AWK-программах

Ввод данных в **AWK**-программу определяется именем входного файла в командной строке. Таких файлов может быть несколько, и обрабатываться **AWK**-программой они будут последовательно в том порядке, в котором указаны в командной строке, например:

```
awk  -f prog f1 f2 f3 f4
```

AWK-программа из файла **prog** будет выполняться над входным потоком записей из файлов **f1**, **f2**, **f3** и **f4**. Здесь необходимо отметить, что предопределенная переменная **NR** будет иметь значение, равное порядковому номеру записи (**NR** не обнуляется при переходе к чтению очередного файла). Пусть имеются четыре файла. Файл **f1**:

```
a[1][1]  a[1][2]  a[1][3]  a[1][4]
a[2][1]  a[2][2]  a[2][3]  a[2][4]
a[3][1]  a[3][2]  a[3][3]  a[3][4]
a[4][1]  a[4][2]  a[4][3]  a[4][4]
```

Файл **f2**:

```
b[1][1]  b[1][2]  b[1][3]  b[1][4]
b[2][1]  b[2][2]  b[2][3]  b[2][4]
```

```
b[3][1]  b[3][2]  b[3][3]  b[3][4]
b[4][1]  b[4][2]  b[4][3]  b[4][4]
```

Файл **f3**:

```
c[1][1]  c[1][2]  c[1][3]  c[1][4]
c[2][1]  c[2][2]  c[2][3]  c[2][4]
c[3][1]  c[3][2]  c[3][3]  c[3][4]
c[4][1]  c[4][2]  c[4][3]  c[4][4]
```

Файл **f4**:

```
d[1][1]  d[1][2]  d[1][3]  d[1][4]
d[2][1]  d[2][2]  d[2][3]  d[2][4]
d[3][1]  d[3][2]  d[3][3]  d[3][4]
d[4][1]  d[4][2]  d[4][3]  d[4][4]
```

Каждый из этих файлов включает по четыре записи (по четыре поля в каждой). Другими словами, каждый файл - матрица **(4*4)**. Допустим, необходимо получить новую матрицу с размерностью **(4*4)**, столбцы которой составлены из элементов диагоналей исходных матриц. Ниже приведен текст программы, в которой решается эта задача:

```
{
  if( FILENAME != Name ) {
            i = 0;
        Name = FILENAME;
    }

    i++;
    if( i == 1 ) {
          Dig1 = Dig1 " " $1;
           next;
    }
    if( i == 2 ) {
          Dig2 = Dig2 " " $2;
           next;
    }
    if( i == 3 ) {
          Dig3 = Dig3 " " $3;
           next;
    }
```



```
    if( i == 4 ) Dig4 = Dig4 " " $4;
  }

  END    {
          print( Dig1 );
          print( Dig2 );
          print( Dig3 );
          print( Dig4 );
  }
```

В программе два правила. Первое правило не содержит селектора, следовательно, выполняется для всех входных записей. Второе правило выполняется по завершению входного потока. Программа работает следующим образом: первоначально проверяется, изменилось ли имя входного файла (предопределенная переменная FILENAME), затем, если не изменилось, присваивается значение соответствующего поля записи к переменной **Dig** (используется операция конкатенации старого значения **Dig** со значением поля и присваивания **Dig** нового значения). Переменная **Name** предназначена для сохранения имени входного файла. Первоначально значения переменных **Name** и **Dig** равны пустым строкам. Важно, что мы знаем точно число записей, это позволяет не выделять по другому нужные поля в записях. Допустим, выполняется следующая командная строка:

```
awk -f prog  f1 f2 f3 f4 > Result
```

в файле **Result** будем иметь:

```
a[1][1] b[1][1] c[1][1] d[1][1]
a[2][2] b[2][2] c[2][2] d[2][2]
a[3][3] b[3][3] c[3][3] d[3][3]
a[4][4] b[4][4] c[4][4] d[4][4]
```

Результат работы программы существенно связан с порядком чтения входных файлов. Если выполнить командную строку

```
awk -f prog f4 f3 f2 f1 > Result
```

получим:

```
d[1][1] c[1][1] b[1][1] a[1][1]
d[2][2] c[2][2] b[2][2] a[2][2]
d[3][3] c[3][3] b[3][3] a[3][3]
```

```
d[4][4] c[4][4] b[4][4] a[4][4]
```

Когда возникает необходимость передать в **AWK**-программу значения некоторых переменных, можно воспользоваться возможностью указать их в файле. Допустим, заранее не известны образцы для выделения записей файла **f1**. В этом случае можно создать файл **f0** с описаниями образцов и, воспользовавшись значением переменной **FILENAME**, присвоить этим переменным нужные значения. Пусть файл **f0** имеет вид:

```
aaa bbb ccc
```

Пусть файл **f1** имеет вид:

```
aaa bbb ccc ddd eee
eee bbb ccc ddd aaa
aaa fff ccc ddd eee
aaa bbb ggg ttt eee
```

Программа на **AWK**:

```
FILENAME == "f0" {
        pat1 =  $1;
        pat2 =  $2;
        pat3 =  $3;
        next;
}

$1 == pat1 { print; next }
$2 == pat2 { print; next }
$3 == pat3 { print }
```

После выполнения командной строки

```
awk -f prog f0 f1
```

получим в файле **Result**:

```
aaa bbb ccc ddd eee
aaa fff ccc ddd eee
aaa bbb ggg ttt eee
```


Можно предусмотреть ввод переменных со стандартного ввода, воспользуемся тем, что переменная **FILENAME** для стандартного ввода определена как '-'. Пусть файл **f1** имеет вид:

```
aaa bbb ccc  ddd eee
eee bbb ooo ddd aaa
aaa fff  ccc  ddd eee
qqq bbb gggttt  eee
ooo fff  gggttt  eee
ccc bbb gggttt  eee
```

Приведенная ниже программа позволяет получить значения переменных с клавиатуры дисплея:

```
BEGIN   { print("Вводите значения полей:"); }

FILENAME == "-" {
        pat1 =  $1;
        pat2 =  $2;
        pat3 =  $3;
}

FILENAME == "f1" {
        if($1 == pat1) { print($0); next }
        if($2 == pat2) { print($0); next }
        if($3 == pat3) { print($0);}
}
```

После запуска на выполнение следующей командной строки

```
awk -f prog - f1
```

программа будет ждать ввода с клавиатуры дисплея (завершить ввод необходимо символом конец файла - **CNTRL/D**). Например:

```
Вводите значения полей:
qqq fff ooo
CNTRL/D
eee bbb ooo ddd aaa
aaa fff  ccc  ddd eee
qqq bbb gggttt  eee
ooo fff  gggttt  eee
```

Как уже говорилось раньше, вывод **AWK**-программы направляется на экран дисплея, если не было указано другое. Существует возможность направить вывод по нескольким каналам непосредственно из программы, для этого можно воспользоваться стандартными средствами системы из **AWK**-программы. Например:

```
print( $0 ) > "file";
```

запись будет направлена в файл с именем **./file**;

```
print( $0 ) >> "file";
```

запись будет дописана в **./file**;

```
print( $0 ) > $2;
```

запись будет направлена в файл с именем, равным содержимому ее второго поля.

Существует возможность из **AWK**-программы направить вывод в конвейер, например:

```
{
   print($0) | "tr ' ' '\n' | sort ";
}
```

Здесь запись будет направлена команде **tr**, которая заменит пробел символом '\n', затем отсортирована командой **sort**. Пусть выполнена следующая командная строка:

```
awk -f prog -
```

после ввода с клавиатуры нескольких записей

```
dfa nrk klm njf rty xvz
saa ass dcf vfr klm ttr
CNTRL/D
```

получим:

```
ass
dcf
dfa
```



```
klm
klm
njf
nrk
rty
saa
ttr
vfr
xvz
```

Вывод результата работы конвейера осуществляется по завершению чтения последней входной записи. Канал вывода в примере совпадает с каналом стандартного вывода, но его можно переопределить на любой файл.

В одной **AWK**-программе можно одновременно определить несколько каналов вывода, число которых зависит от числа файлов, разрешенных для одновременного использования. Это число устанавливается при генерации операционной системы ДЕМОС.

Для вывода данных в **AWK**-программе предназначен оператор **print**. До настоящего момента мы применяли лишь одну форму использования этого оператора:

```
print(список_фактических_параметров);
```

Круглые скобки использовались раньше для того, чтобы не отвлекать читателя, знакомого с языком программирования Си, - их можно не указывать. Существуют и другие формы использования этого оператора:

print;
выводится вся запись;

print $1, $2;
значения полей выводятся через пробел;

print $1 $2;
выводится конкатенация значений полей.

При необходимости управления форматом вывода можно использовать библиотечную функцию **printf**, синтаксис и результат работы которой такие же, как и в языке Си.

4.7. Использование встроенных функций

Интерпретатор **awk** включает набор встроенных функций, которые можно использовать в действиях правил. Существуют два способа вызова встроенных функций:

имя_функции(список_фактических_параметров)

имя_функции

Во втором случае в качестве фактического параметра применяется вся текущая запись. Как обычно, значение функции подставляется в выражение в том месте, где определен вызов.

Имеются следующие встроенные функции:

length(выражение)

значением выражения является строка. **Length** возвращает длину строки, например:

```
print( length($1 " " $2));
```

будет напечатана длина строки, полученной конкатенацией поля **$1**, пробела и поля **$2**. Форма без аргумента возвращает длину записи.

exp(выражение)

возвращает экспоненту от выражения.

log(выражение)

возвращает натуральный логарифм выражения.

sqrt(выражение)

возвращает значение квадратного корня от выражения.

int(выражение)

возвращает целую часть числа, равного значению выражения.

substr(S, M, N)

возвращает часть строки **S**, начинающуюся от позиции **M** и имеющую длину не более **N** символов. Символы в строке **S** нумеруются с **1**. Если аргумент **N** не указан, возвращаются все символы от **M** до конца строки.

```
string = substr( $0, 12, 20);
```

String будет включать **9** символов (с **12** по **20**) текущей записи.

index(As, Ps)

возвращает номер позиции, с которой строка **Ps** совпадает со строкой **As**. Если совпадения нет, возвращается **0**.

sprintf(формат, выражение, ...)

возвращает строку, выведенную по формату. Синтаксис функции и результат работы аналогичны функции **sprintf** в библиотеке языка программирования Си.

split(S, Name, разделитель)

строка **S** разбивается на поля, значения которых присваиваются

элементам массива **Name**. Значением первого элемента **Name[1]** будет содержимое первого выделенного поля, значением второго элемента **Name[2]** - второго выделенного поля и так далее. Если не указан разделитель полей, используется значение предопределенной переменной **FS**. Функция **split** возвращает число выделенных полей. Рассмотрим пример. Пусть имеется файл **f1**

```
aaa bbb ccc# ddd# eee fff# ggg
ttt# ggg eee# ccc ddd sss# yyy
```

и **AWK**-программа

```
{
  i = split( $0, Name, "#");
  for(j = 1; j <= i; j++)
      print( "Name["j"] =", Name[j]);
}
```

после выполнения командной строки

```
awk -f prog f1
```

получим:

```
Name[1] = aaa bbb ccc
Name[2] = ddd
Name[3] =  eee fff
Name[4] = ggg
Name[1] = ttt
Name[2] = ggg eee
Name[3] =  ccc ddd sss
Name[4] = yyy
```

Глава 5. СРЕДСТВА РАЗРАБОТКИ КОМПИЛЯТОРОВ И ИНТЕРПРЕТАТОРОВ

ОС ДЕМОС имеет в своем составе генератор программ лексического анализа (**lex**) [14] и генератор программ синтаксического анализа (**yacc**) [13], облегчающие разработку компиляторов. Генераторы по описанию, содержащемуся в файлах спецификаций, строят Си-программы анализаторов.

Оба генератора создают выходные программы на основе файлов-заготовок и файлов-спецификаций. Файл-заготовка для **lex** (**/usr/lib/lex/ncform**) содержит "каркас" будущей Си-программы лексического анализатора. В результате работы генератора **lex** создается файл с предопределенным именем **lex.yy.c**, содержащий текст Си-программы собственно лексического анализатора. Аналогично и для **yacc** имеется файл-заготовка (**/usr/lib/yaccpar**), а результат работы генератора - синтаксический анализатор на Си - записывается в файл **y.tab.c**.

Таким образом, **lex** и **yacc** представляют собой текстовые процессоры, строящие на основе файлов-заготовок, исходно существующих в системе, и файлов-спецификаций, подготовленных программистом, тексты соответствующих анализаторов на Си.

Файлы спецификаций оформляются программистом в виде набора правил специального вида. Например, правило для **lex** задает соответствие между лексемой, выделяемой из входного потока, и действием, которое необходимо выполнить, если эта лексема распознана. Действие обычно заключается в возвращении лексемы в том или ином виде в программу, вызвавшую лексический анализатор (обычно это синтаксический анализатор), и может быть задано в виде обращения к некоторой Си-функции. Лексема, которую необходимо распознать, чтобы данное действие выполнилось, задается в виде шаблонов (иногда их называют регулярными выражениями), которые позволяют эффективно распознавать лексемы во входном потоке символов.

Входной информацией для **yacc** является описание грамматики входного языка в виде набора правил, в форме, близкой к БНФ. С каждым правилом могут быть связаны действия, в которых возможно обращение к Си-функциям, описанным программистом, или входящим в набор стандартных функций **yacc**.

5.1. Принципы работы генераторов lex и yacc

Предполагается, что читатель знаком с основными теоретическими и методологическими аспектами конструирования компиляторов. В последующем изложении мы не претендуем на сколько-нибудь полное рассмотрение этих вопросов, а лишь напомним основное содержание терминов, используемых при описании **lex** и **yacc**.

Возьмем сильно упрощенный язык алгебраических выражений, имеющий словарь, состоящий из следующих символов (слов):

{ a , b , (,) , + , - , * , / }

Здесь 'a' и 'b' имеют смысл операндов выражения, смысл остальных символов очевиден. Рассмотрим грамматику (контекстно свободную по Хомскому) языка, заданную следующими правилами вывода:

S --> W + S
S --> W - S
S --> W
W --> W * W
W --> W / W
W --> (S)
W --> a
W --> b

Символы, составляющие словарь языка, называются терминальными символами грамматики, или просто терминалами, символы **S** и **W** называются соответственно нетерминальными символами, или нетерминалами грамматики. В отличие от терминалов, которые не должны появляться в левых частях правил, каждый нетерминальный символ должен стоять в левой части хотя бы одного правила. Один из нетерминалов, в данном случае **S**, фиксируется как начальный символ грамматики. Задание грамматики определяется как задание четверки, состоящей из

множества терминальных символов;

множества нетерминальных символов;

множества правил;

начального символа.

Каждое правило грамматики задает некоторую подстановку, которую можно применить для получения одной цепочки символов из другой. При этом левая часть правила определяет некоторый нетерминал, на место которого осуществляется подстановка, а правая часть определяет саму эту подстановку. Последовательность подстановок, с помощью которой из начального символа грамматики получается некоторая цепочка, называется выводом этой цепочки. Совокупность всех цепочек терминальных символов, которые можно вывести из начального символа, представляет собой язык, определяемый данной грамматикой. Например, вывод алгебраического выражения

```
( a + b * b ) - a / b
```

из начальнего символа **S** данной грамматики может иметь следующий вид:

```
S                  —>
W - S              —>
( S )- S           —>
( W + S )- S       —>
( a + S )- S       —>
( a + W )-         —>
( a + W * W )- S   —>
( a + b * W )- S   —>
( a + b * b )- S   —>
( a + b * b )- W   —>
( a + b * b )- W / W —>
( a + b * b )- a / W —>
( a + b * b )- a / b
```

Главной задачей синтаксического анализа является установление принадлежности последовательности символов входного текста языку, определяемому грамматикой. Для решения этой задачи синтаксическому анализатору необходимо восстановить вывод входной цепочки из начального символа или установить невозможность такого вывода. Методы анализа обычно делят на "нисходящие" и "восходящие", отличающиеся друг от друга порядком, в котором распознаются правила в выводе.

Синтаксический анализатор, построенный с помощью **yacc**, реализует так называемый **LALR(1)** анализ, относящийся к категории "восходящих" методов. Анализ заключается в переносе входных символов в стек с предварительной проверкой возможности свертывания нескольких верхних символов стека. Под свертыванием понимается замена этих символов одним нетерминалом, что соответствует распознаванию очередного правила в

выводе входной цепочки. Используя специальную управляющую таблицу, **LALR(1)**-анализатор по очередному входному символу и верхнему символу стека решает, какую надо выполнить операцию - "сдвиг" ("перенос") или "свертку". При выполнение операции "сдвиг" проверяется соответствие состояния анализатора очередному входному символу, и если это соответствие установлено, анализатор продолжает свою работу, в противном случае входная последовательность отвергается как содержащая ошибку. **Yacc** строит таблицы, в которых для каждого возможного состояния определена операция "сдвига" или "свертки".

Входной информацией для **lex** и **yacc** является файл спецификаций, имеющий следующую структуру:

```
раздел деклараций
%%
раздел правил
%%
раздел функций
```

Файл спецификаций для **yacc** будем называть **yacc**-программой, а для **lex** - **lex**-программой.

Пусть **f.y** - **yacc**-программа, а **f.l** - **lex**-программа, тогда готовые к использованию лексический и синтаксический анализаторы можно соответственно получить следующим образом:

```
lex f.l                    |   yacc f.y
cc lex.yy.c -o lan -ll     |   cc y.tab.c -o gram -ly
```

В файле **lex.yy.c** основной является функция **yylex()**, которая осуществляет анализ входного потока и выделяет из него лексемы. Последняя выделенная лексема хранится в предопределенной переменной **yytext**. Функция **yylex()** возвращает номер типа распознанной лексемы.

В файле **y.tab.c** основными являются таблицы, управляющие работой магазинного автомата, осуществляющего **LALR(1)** разбор. Функция **yyparse()** возвращает **0**, если входной текст не содержит ошибок, и **1** - в противном случае.

Программист имеет возможность включать в **lex** и **yacc**-программы написанные им Си-функции, строки, обрабатываемые Си-препроцессором, вызовы своих и библиотечных функций, и использовать их в определении действий, которые целесообразно выполнить в ходе лексического и синтаксического анализов.

Yacc и **lex** широко используются в ОС ДЕМОС для создания компиляторов, макрогенераторов, интерпретаторов, редакторов текстов и других компонент системы.

5.2. Шаблоны в правилах lex-программы

Шаблон может содержать символы латинского и русского алфавитов в верхнем и нижнем регистрах, другие символы (цифры, знаки препинания и т.д.) и символы-операторы. Операторы позволяют осуществлять различные действия над выделенной цепочкой символов. Операторы также обозначаются символами.

В шаблоне можно использовать любой символ. Его можно указывать в двойных кавычках, в этом случае - это всегда просто символ - его специальное значение отменяется. Некоторые символы имеют специальное значение, например:

. - любой символ, кроме символа 'перевод строки';

\n - символ 'перевод строки';

\t - символ табуляции;

\b - возврат курсора на один шаг назад;

\восьмеричный_код - указание символа кодом;

Символ пробела в выражении, если он не находится внутри квадратных скобок, необходимо заключать в двойные кавычки, так как пробел и табуляция используются Lex в качестве разделителя между определением и действием в правиле.

Операторы обозначаются символами-операторами, к ним относятся:

\ ^ ? * + | $ / %

[] {} () <>

Каждый из этих символов или пар скобок в шаблоне является оператором. Если необходимо отменить специальное значение символа, обозначающего оператор, перед ним нужно указать обратную наклонную черту или указать его в двойных кавычках, например:

abc+ - символ "+" - оператор;

abc\+ - символ "+";

abc"+" - символ "+"

Квадратные скобки

задают классы символов, которые в них заключены, например [abc] означает либо символ a, либо b, либо c. Знак "-" используется для указания любого символа из лексикографически упорядоченной последовательности. Например, [А-Я] - любая прописная русская буква, [+-0-9] - все цифры и знаки + и -.

Когда необходимо указать повторяемость вхождения символа в регулярном выражении, используют операторы-повторители "*" и "+".

Оператор *

любое (в том числе и 0) число вхождений символа или класса символов. Например: W* - любое число вхождений символа W; ABC* - любое число вхождений цепочки ABC; [A-z] - любое число вхождений любой латинской буквы.

Оператор +

означает одно и более вхождений. Например: F+ - одно или более вхождений F; [0-9]+ - цепочка цифр ненулевой длины; ABC+ - одно или более вхождений цепочки ABC.

Процессом выбора символов управляют операторы

? / | $ ^

Оператор /

ab/cd - 'ab' учитывается только тогда, когда за ним следует 'cd'.

Оператор |

ab|cd - означает или 'ab', или 'cd'.

Оператор ?

t? - необязательный символ t. Чтобы указать, что перед цепочкой любого количества латинских букв может быть необязательный знак подчеркивания, можно записать '_?[A-Za-z]*'.

Оператор $

x$ - выбрать символ x, если он является последним в строке (стоит перед символом перевод строки). ABC$ - выбрать цепочку 'ABC', если она завершает строку.

Оператор ^

x - выбрать символ 'x', если он является первым символом строки. ABC - выбрать цепочку символов 'ABC', если она начинает строку. [A-Z] - все символы, кроме прописных латинских букв. Когда символ

^ стоит перед выражением или внутри квадратных скобок, он выполняет операцию дополнение. Внутри квадратных скобок символ ^ должен обязательно стоять первым у открывающей скобки.

Оператор **x{n,m}**

здесь **n** и **m** натуральные, **m** > **n**. Означает от **n** до **m** вхождений **x**, например **x{2,7}** - от 2 до 7 вхождений **x**.

Оператор {имя}

вместо "{имя}" будет подставлено определение имени из раздела деклараций **Lex**-программы, например:

```
   БУКВА            [A-ZA-Яa-zа-я_]
   ЦИФРА             [0-9]
ИДЕНТИФИКАТОР  {БУКВА}({БУКВА}|{ЦИФРА})*

   %
{ИДЕНТИФИКАТОР} printf("\n%s", yytext);
        ...
        ...
```

В этом примере {ИДЕНТИФИКАТОР} будет заменен на {БУКВА}({БУКВА}|{ЦИФРА})*. Здесь **yytext** - это внешний массив символов программы **lex.yy.c**, которую строит генератор **lex**. **Yytext** формируется в процессе чтения входного файла и содержит текст, для которого установлено соответствие какому-либо шаблону.

5.3. Раздел деклараций lex-программы

Декларации помещаются перед первой парой символов **%%**. Любая строка этого раздела, не содержащаяся между %{ и %} и начинающаяся в первой колонке, является определением строки подстановки **Lex**. Раздел деклараций **lex**-программы может включать: начальные условия, определения, фрагменты программы пользователя, таблицы наборов символов, изменения размеров внутренних массивов и комментарии в формате языка Си.

Начальные условия задаются в форме

%START имя1 имя2 ...

Если начальные условия определены, то эта строка должна быть первой в **lex**-программе.

Определения задаются в форме

имя значение

Имя - любая последовательность букв и цифр, начинающаяся с буквы. В качестве разделителя между именем и значением используется один или более пробелов или табуляций.

Фрагменты программ пользователя указываются двумя способами:

шаблон предложение_языка_Си;

Например:

```
abc+      printf("%s", yytext);
```

выведет лексему, выбранную по шаблону **abc+**. Или

```
%{
 строки фрагмента
%}
```

такая форма включения фрагмента применяется для ввода, например, строк препроцессора Си, которые должны начинаться с первой колонки строки, например:

```
%{
# include      <stdio.h>

# define       VOID   int
extern         int    name;
%}
```

Все строки фрагмента пользовательской программы, размещенные в разделе деклараций, будут являться внешними для любой функции программы **lex.yy.c**

Таблица символов задается в виде

```
%T
 целое_число    строка_символов
          ...
          ...
```

```
целое_число     строка_символов
%T
```

Сгенерированная программа **lex.yy.c** осуществляет ввод-вывод символов посредством библиотечных функций **Lex** с именами **input, output, unput** (они будут описаны ниже). Таким образом, **Lex** помещает в **yytext** символы в представлении, используемом в этих библиотечных функциях. Символ представляется целым числом, значение которого образовано набором битов, представляющих символ в конкретной ЭВМ. Программисту предоставляется возможность менять представление символов с помощью таблицы наборов символов. Если таблица наборов присутствует в разделе деклараций, любой символ, появляющийся либо во входном потоке, либо в правилах, должен быть определен в ней. Символам нельзя устанавливать значение 0 и число, большее выделенного для внутреннего представления символов конкретной ЭВМ, например:

```
%T
 1      Aa
      ...
 26     Zz
 27     \n
 28     +
 29     -
 30     0
      ...
 39     9
%T
```

здесь латинские символы верхнего и нижнего регистров переводятся в числа **1-26**, символ новой строки - в **27**, "+" и "-" - в числа **28** и **29**, а цифры - в числа **30-39**.

Изменения размеров внутренних массивов задаются в форме

%x число

число - новый размер массива;
x - одна из букв:

p - позиции;
n - состояния;

e - узлы дерева;
a - упакованные переходы;
k - упакованные классы символов;
o - массив выходных элементов.

Генератор **lex** имеет внутренние таблицы, размеры которых ограничены. При построении программы лексического анализа может произойти переполнение любой из этих таблиц. Программисту предоставляется возможность изменить размеры таблиц (сокращая размеры одних и увеличивая размеры других) таким образом, чтобы они не переполнялись. Чтобы определить, каковы размеры таблиц и насколько они заняты, можно использовать ключ **-v** при вызове программы **lex**.

Комментарии в разделе деклараций задаются, как в Си, но указываются не с первой колонки строки.

5.4. Раздел правил и функции в lex-программе

Все, что указано после первой пары **%%** и до конца **lex**-программы или до второй пары **%%** (если она указана) относится к разделу правил, а после второй пары **%%** - к разделу функций. Раздел правил может включать правила и фрагменты программ. Фрагменты программ, содержащиеся в разделе правил, становятся частью функции **yylex**. Фрагмент программы указывается так же, как и в разделе деклараций, например:

```
%% .
%{
# include "file.h"
%}
    ...
```

Здесь строка <**#include "file.h"**> станет строкой функции **yylex()**. Раздел правил может включать список активных и неактивных (помеченных) правил, которые указываются в любом порядке. Активные правила выполняются всегда, неактивные - только по ссылке на них оператором **BEGIN**. Активное правило имеет вид:

ШАБЛОН ДЕЙСТВИЕ

Неактивное правило имеет вид:

<МЕТКА>ШАБЛОН ДЕЙСТВИЕ

или

<СПИСОК_МЕТОК>ШАБЛОН ДЕЙСТВИЕ

где СПИСОК_МЕТОК имеет вид:

МЕТКА_1,МЕТКА_2,...

В качестве первого правила **lex**-программы может быть правило вида **BEGIN** МЕТКА. В этом правиле отсутствует ШАБЛОН, и первым действием в разделе правил будет активизация помеченных меткой правил. Для возвращения в исходное состояние можно использовать действие **BEGIN 0**. Когда **lex**-программа содержит активные и неактивные правила, активные правила работают всегда. Оператор **BEGIN** МЕТКА просто расширяет список активных правил, активизируя помеченные меткой, а оператор **BEGIN 0** удаляет из списка активных правил все помеченные правила, которые до этого были активизированы. Кроме того, если из помеченного и активного в данный момент времени правила осуществляется действие **BEGIN** МЕТКА, то из помеченных правил активными останутся только те, которые помечены меткой.

Действие можно представлять либо как оператор **lex**, например **BEGIN** МЕТКА, либо как оператор Си. Если имеется необходимость выполнить достаточно большой набор преобразований, то действие оформляют как блок Си-программы (он начинается открывающей фигурной скобкой и завершается закрывающей фигурной скобкой), содержащий необходимые фрагменты. Действие в правиле указывается через не менее чем один пробел или табуляцию после шаблона (обязательно в той же строке), а его продолжение может быть указано в следующих строках только в том случае, если действие оформлено как блок Си-программы. Область действия переменных, объявленных внутри блока, распространяется только на этот блок. Внешними переменными для всех действий будут являться только те переменные, которые объявлены в разделе деклараций **lex**-программы.

Действия в правилах **lex**-программы выполняются, если правило активно и распознается лексема, соответствующая шаблону этого правила. Однако одно действие выполняется всегда - оно заключается в копировании входного потока символов в выходной. Это копирование осуществляется для всех входных строк, которые не соответствуют правилам, преобразующим эти строки. Комбинация символов, не учтенная в правилах и появившаяся на входе, будет напечатана на выходе. Можно сказать, что действие - это то, что делается вместо копирования входного потока символов на выход. Часто бывает необходимо просто не копировать на выход лексему, которая удовлетворяет некоторому шаблону. Для этой

цели используется пустой оператор Си, например:

```
[ \t\n]          ;
```

Это правило запрещает вывод пробелов, табуляций и символа "перевод строки". Запрет выражается в том, что появление этих символов вызывает действие ";" - пустой оператор Си, и они не копируются в выводной поток символов.

Существует возможность для нескольких шаблонов указывать одно действие. Для этого используется символ "|", который указывает, что действие данного правила совпадает с действием для следующего, например:

```
" "              |
\t               |
\n               ;
```

Когда необходимо вывести или преобразовать текст, соответствующий некоторому шаблону, используется внешний массив символов **yytext**, который формирует **lex**. Например:

```
[A-Z]+           printf("%s", yytext);
```

Здесь распознается слово, содержащее прописные латинские буквы, и выводится с помощью **printf**. Операция вывода распознанного выражения используется очень часто, поэтому имеется сокращенная форма записи этого действия:

```
[A-Z]+           ECHO;
```

В выходном файле **lex.yy.c ECHO** определено как макроподстановка:

```
# define  ECHO   fprintf(yyout, "%s", yytext)
```

Когда необходимо знать длину обнаруженной последовательности символов, используется счетчик числа найденных символов **yyleng**, который также доступен в действиях, например правило

```
[A-Z]+      printf("%c", yytext[yyleng-1]);
```

будет выводить последний символ слова, соответствующий шаблону **[A-Z]+**. В другом примере

```
[A-Z]+ {число_слов++; число_букв += yyleng;}
```

ведется подсчет числа распознанных слов и количества символов во всех словах. Рассмотрим еще один пример:

```
%Start COMMENT
КОММ_НАЧАЛО   "/*"
КОММ_КОНЕЦ    "*/"

%%
{КОММ_НАЧАЛО}      {ECHO; BEGIN COMMENT;};
 .                  ;
[ \t\n]*            ;
<COMMENT>[ ^*]*     ECHO;
<COMMENT>\*/[ ^/]   ECHO;
<COMMENT>{КОММ_КОНЕЦ} {ECHO; printf("\n"); BEGIN 0;};
```

Эта lex-программа выделяет комментарии в Си-программе и записывает их в стандартный файл вывода. Программа начинается с ключевого слова **Start**, за ним указана метка начального условия **COMMENT**. Оператор **BEGIN COMMENT** переводит анализатор в начальное условие **COMMENT**. После этого анализатор уже находится в новом состоянии и теперь разбор входного потока символов будет осуществляться и теми правилами, которые начинаются оператором <COMMENT>.

Lex-программа может содержать несколько помеченных начальных условий. Например, если она начинается строкой

```
%Start AA BB CC
```

то это означает, что она управляет тремя начальными состояниями анализатора. В каждое из этих начальных состояний можно перевести анализатор, используя оператор **BEGIN**. Количество помеченных правил не ограничивается. Кроме того, разрешается одно правило помечать несколькими метками, например:

```
<МЕТКА1,МЕТКА2,МЕТКА3>x ДЕЙСТВИЕ
```

Запятая - обязательный разделитель списка меток. Рассмотрим пример с несколькими начальными условиями:

```
%START AA BB CC
```



```
БУКВА          [A-ZA-Яa-za-я_]
ЦИФРА          [0-9]
ИДЕНТИФИКАТОР  {БУКВА}({БУКВА}|{ЦИФРА})*

%%
^#                          BEGIN AA;
^[ \t]*main                 BEGIN BB;
^[ \t]*{ИДЕНТИФИКАТОР} BEGIN CC;
 .                          ;
\t                          ;
\n                          BEGIN 0;

<AA>define                  printf("Определение.\n");
<AA>include                 printf("Включение.\n");
<AA>ifdef                   printf("Условная компиляция.\n");

<BB>[ ^\,]*","[ ^\,]*")"    printf("main с аргументами.\n");
<BB>[ ^\,]*")"              printf("main без аргументов.\n");

<CC>":"/[ \t]               printf("Метка.\n");
```

Лексический анализатор будет распознавать в Си-программе строки препроцессора, выделять функцию **main**, распознавая, с аргументами она или без них, распознавать метки. Программа не выводит ничего, кроме сообщений о выделенных лексемах.

В процессе распознавания символов входного потока может оказаться так, что одна цепочка символов будет удовлетворять нескольким шаблонам и, следовательно, возникает проблема: действие какого правила должно выполняться? Для разрешения этого противоречия можно использовать квантование (разбиение) шаблонов этих правил на такие, которые однозначно распознают лексемы. Однако, когда это не сделано, **Lex** использует определенный детерминированный механизм разрешения противоречия:

выбирается действие того правила, которое распознает наиболее длинную последовательность символов из входного потока;

если несколько правил распознают последовательности символов одной длины, то выполняется действие того правила, которое записано первым в списке раздела правил lex-программы, например:

```
...
[Мм][Аа][Йй]      ECHO;
[А-Яа-я]+         ECHO;
...
```

Слово "Май" распознают оба правила. Выполнится первое из них, так как и первое, и второе правило распознали лексему одинакового размера (3 символа). Если во входном потоке будет, допустим, слово "майский", то первые 3 символа удовлетворяют первому правилу, а все 7 символов - второму, следовательно, выполнится второе.

Комментарии можно указывать во всех разделах lex-программы. В каждом разделе lex-программы комментарии указываются по-разному. В разделе деклараций комментарии должны начинаться не с первой позиции строки. В разделе правил комментарии можно указывать только внутри блоков, принадлежащих действиям. В разделе функций комментарии указываются, как в языке Си. Пример 1.

```
%Start KOMMENT
    /*
     * Программа записывает в стандартный файл вывода
     * комментарии Си-программы.  Обратите внимание
     * на то, что  здесь  строки комментариев указаны
     * не с первой позиции строки.
     */
KOMM_НАЧАЛО   "/*"
KOMM_КОНЕЦ    "*/"

%%
{KOMM_НАЧАЛО}      {ECHO; BEGIN KOMMENT;}
 .                  ;
[\t\n]*             ;

<KOMMENT>[^*]*      ECHO;
<KOMMENT>\*/[^/]    ECHO;
<KOMMENT>{KOMM_КОНЕЦ} {ECHO; printf("\n");

/*
 * Здесь приведен пример использования  комментариев в
 * разделе правил lex-программы.  Обратите внимание на
 * то, что комментарий указан внутри блока, определяю-
 * щего действие правила.
 */
}
```



```
%%
/*
 * Пример комментариев в разделе функций.
 */
```

Пример 2.

```
%Start IC1 IC2 Normal
    /*
     * Фрагмент  lex-программы,  которая
     * строит лексический анализатор для
     * компилятора языка Паскаль.  Действие
     * return(...) возвращает тип лексемы в
     * вызывающую анализатор программу.
     * Все цепочки символов входного потока,
     * не  распознанные  в  правилах,
     * копируются в выходной поток символов.
     */

LETTER          [A-ZA-Яa-za-я_]
DIGIT           [0-9]
IDENT       {LETTER}({LETTER}|{DIGIT})*
INT             {DIGIT}+
FIXED           {INT}?\.{INT}
WHISP            [ \t\n]*

%%
                        BEGIN Normal;
<Normal>"{"             BEGIN IC1;
<IC1>[^}]               ;
<IC1>"}"                BEGIN Normal;
<Normal>"(*"            BEGIN IC2;
<IC2>[^*]|\*/[^)]       ;
<IC2>"*)"               BEGIN Normal;

<Normal>'([^']|'')*'    return( строка );
<Normal>"<"             return( меньше );
                    ...
                    ...
<Normal>"-"             return( минус );
<Normal>"/"             return( разделить );
<Normal>mod             return( t_mod );
```

```
                ...
                ...
<Normal>{FIXED}      x( "float" ) ;
<Normal>[ \n\t]      ;
%%

x( s )
char *s ;
{
    printf("%-15.15s > %s < \n", s, yytext ):
}
```

Lex строит программу - лексический анализатор на языке Си, - которая размещается в файле со стандартным именем **lex.yy.c**. Эта программа содержит две основные функции и несколько вспомогательных. Основные:

функция **yylex()**
: содержит разделы действий всех правил, которые определены пользователем;

функция **yylook()**
: реализует детерминированный конечный автомат, который осуществляет разбор входного потока символов в соответствии с шаблонами правил **lex**-программы.

Вспомогательные функции являются подпрограммами ввода-вывода. К ним относятся:

input()
: читает и возвращает символ из входного потока символов;.

unput(c)
: возвращает символ обратно во входной поток для повторного чтения;

output(c)
: выводит в выходной поток символ "**c**".

yywrap()
: возвращает 1 по завершении входного потока символов, иначе 0;

reject()
: возвращает символ во входной поток и вызывает повторный анализ;

yyless(n)
: выделяет n символов из строки **yytext**;

yymore()
: добавляет в **yytext** следующую распознанную лексему.

При сборке программы лексического анализа редактор связи **ld** по ключу **-ll** подключает к файлу **lex.yy.c** головную функцию **main**, если она не определена. Ниже приведен текст этой функции из библиотеки **/usr/lib/libl.a**:

```
# include <stdio.h>

main(){
        yylex(); exit(0);
}
```

В **lex**-программе невозможно записать правило, которое будет обнаруживать конец файла. Единственный способ это сделать - использовать функцию **yywrap**. Если **yywrap** возвращает 1, лексический анализатор прекращает работу. Если необходимо начать ввод данных из другого источника и продолжить работу, программист должен написать свою подпрограмму **yywrap**, которая организует новый входной поток и возвращает **0**, что служит сигналом к продолжению работы анализатора. Эта функция также удобна, когда необходимо выполнить какие-либо действия по завершении входного потока символов. По умолчанию **yywrap** всегда возвращает **1** при завершении входного потока символов. Пример:

```
%START AA BB CC
  /*
   * Строится  лексический анализатор,  который
   * распознает  наличие включений файлов  в
   * Си-программе,  условных  компиляций,
   * макроопределений, меток и головной функции
   * main.  Анализатор ничего не выводит, пока
   * осуществляется  чтение входного потока,  а
   * по его завершении выводит статистику.
   */

БУКВА          [A-ZA-Яa-za-я_]
ЦИФРА          [0-9]
ИДЕНТИФИКАТОР {БУКВА}({БУКВА}|{ЦИФРА})*
                int a1, a2, a3, b1, b2, c;

%%
        {a1 = a2 = a3 = b1 = b2 = c = 0;}
^#                          BEGIN AA;
^[ \t]*main                 BEGIN BB;
```

```
^[ \t]*{ИДЕНТИФИКАТОР}    BEGIN CC;
.                          ;
\t                         ;
\n                        BEGIN 0;

<AA>define                 { a1++; }
<AA>include                { a2++; }
<AA>ifdef                  { a3++; }

<BB>[ ^\,]*","[ ^\,]*")"   { b1++; }
<BB>[ ^\,]*")"             { b2++; }

<CC>":"/[ \t]              { c++;  }
%%
yywrap(){

  if( b1 == 0 && b2 == 0 )
    printf("В программе отсутствует  функция main");

  if( b1 >= 1 && b2 >= 1 ){
    printf("Многократное определение функции main");
  } else {

    if(b1 == 1.)
        printf("Функция main  с аргументами");

    if( b2 == 1 )
        printf("Функция main без аргументов");
  }
  printf("Включений файлов: %d.", a2);
  printf("Условных компиляций: %d.", a3);
  printf("Определений: %d.", a1);
  printf("Меток: %d.", c);
  return(1);
}
```

Оператор **return(1)** в функции **yywrap** указывает, что лексический анализатор должен завершить работу. Если необходимо продолжить работу анализатора для чтения данных из нового файла, нужно указать **return(0)**, предварительно осуществив операции закрытия и открытия файлов.

Каждый символ из входного потока рассматривается один и только один раз. Предположим, что мы хотим подсчитать все вхождения "она" и

"он" во входном тексте. Для этого мы могли бы записать следующие правила:

```
она     s++;
он      h++;
.       |
\n      ;
```

Так как "она" включает в себя "он", анализатор не распознает те вхождения "он", которые включены в "она". Иногда желательно переопределить этот выбор. Действие функции **REJECT** означает "выбрать следующую альтернативу". Это приводит к тому, что, каким бы ни было правило, после него необходимо выполнить второй выбор. Соответственно изменится и положение указателя во входном потоке:

```
она         { s++; REJECT; }
он          { h++; REJECT; }
.            |
\n           ;
```

Здесь после выполнения одного правила символы возвращаются назад во входной поток, и выполняется другое правило. Функция **REJECT** полезна в том случае, когда она применяется для определения всех вхождений какого-либо объекта, причем вхождения могут перекрываться или включать друг друга.

В обычной ситуации содержимое **yytext** обновляется всякий раз, когда на входе появляется следующая строка. Если возникает необходимость добавить к текущему содержимому **yytext** следующую распознанную цепочку символов, используется функция **yymore**. Формат вызова этой функции:

```
yymore();
```

В некоторых случаях возникает необходимость использовать не все символы распознанной последовательности в **yytext**, а только необходимое их число. Для этой цели предназначена функция **yyless**. Формат ее вызова:

```
yyless(n)
```

где **n** указывает, что в данный момент необходимы только **n** символов строки в **yytext**. Остальные найденные символы будут возвращены во входной поток.

5.5. Раздел деклараций yacc-программы

Входная информация для yacc задается в файле, который будем называть yacc-программой. В результате ее выполнения создается файл со стандартным именем y.tab.c. Этот файл содержит Си-программу грамматического анализатора. Основной в этой программе является функция **yyparse()** (это имя предопределено), которая возвращает 0 при удачном грамматическом разборе входного файла и 1 - в противном случае.

Кроме выходного файла y.tab.c, генератор yacc может дополнительно создавать следующие выходные файлы:

y.output,
содержащий описание состояний анализатора и сообщения о конфликтах;

y.tab.h,
содержащий описание лексем.

Для генерации этих файлов требуется задание соответствующих ключей при вызове генератора.

Раздел деклараций yacc-программы включает строки-директивы и строки исходного текста. Строки-директивы начинаются символом %. Группы строк исходного текста определяются так же, как и в lex-программе, они без изменений копируются в выходной файл y.tab.c, могут содержать директивы препроцессора Си-компилятора и описания Си-переменных. Область действия этих переменных распространяется на все разделы программы, размещенной в файле y.tab.c. Строки-директивы используются для декларации имен лексем, приоритетов, ассоциативности и декларации имени начального нетерминала. Декларация имен лексем осуществляется директивой **token** и имеет вид:

%token список_имен_лексем

При описании грамматики программист решает, какие лексемы целесообразнее непосредственно выделять из входного текста на этапе лексического анализа. Все виды лексем, кроме литералов, обозначаются некоторыми именами и под этими именами фигурируют в разделе правил. Декларация имен ни в коей мере не обеспечивает распознавания лексем, поэтому рекомендуется считать для себя директиву **%token** напоминанием о необходимости построения лексического анализатора. Пример декларации имен лексем:

%token IDENT CONST ЗНАК IF THEN GOTO

Заметим, что ключевые слова входного языка часто бывает удобно считать лексемами. Имена лексем могут совпадать с этими ключевыми словами, недопустимым является лишь совпадение имен лексем с зарезервированными словами языка Си. В примере во избежание недоразумений для имен лексем использованы прописные буквы. Приоритеты и ассоциативность лексем задаются в секции деклараций посредством директив **left**, **right**, **nonassoc**:

% left список_лексем
% right список_лексем
% nonassoc список_лексем

Каждая директива приписывает всем перечисленным в списке лексемам одинаковый приоритет. Директивы **left** и **right** одновременно задают соответственно левую или правую ассоциативность лексем. Директива **nonassoc** означает отсутствие у перечисленных лексем свойства ассоциативности. Устанавливаемый директивами приоритет не имеет численного выражения, а его относительное значение возрастает сверху вниз. Пример задания приоритетов лексем:

```
%token OR NOR XOR AND NAND
%right '='
%left  OR NOR XOR
%left  AND NAND
%left  '+' '-'
%left  '*' '/'.
```

Самый низкий приоритет имеет лексема '=', самый высокий - лексемы - '*' и '/'. Назначение декларации приоритетов и ассоциативности описывается ниже в связи с вопросом о разрешении конфликтов грамматического разбора. Использование лексемы само по себе не требует задания ее приоритета или ассоциативности. При первом появлении лексемы или литерала в секции деклараций за каждым из них может следовать натуральное число, рассматриваемое как НОМЕР ТИПА лексемы. По умолчанию номера типов всех лексем определяются следующим образом:

для литерала номером типа лексемы считается числовое значение символа, рассматриваемого как однобайтовое целое число;

лексемы, обозначенные именами, в соответствии с очередностью их объявления получают последовательные номера, начиная с **257**;

специальная лексема **error**, зарезервированная для обработки ошибок, получает номер типа **256**.

Для каждого имени лексемы независимо от того, переопределен ли ее номер пользователем, **уасс** генерирует в выходном файле **y.tab.c** строки препроцессора Си-компилятора вида

 # define имя_лексемы номер_типа

В случае переопределения номера типа литерала также формируется оператор **#define**. Например, директива

%left ´z´ 258

породит оператор

#define z 258

Заметим, что возможно переопределение номера лишь для литералов. Переопределив при необходимости ряд номеров типов лексем, программист должен проверить их уникальность.

Декларация имени начального нетерминала выполняется директивой **start**:

%start имя_начального_нетерминала

Директива отменяет действующий по умолчанию выбор в качестве начального нетерминала, указанного в первом правиле раздела программ.

5.6. Разделы правил и функций уасс-программы

В этом разделе **уасс**-программы с помощью набора правил должны быть определены все конструкции, из которых впоследствии будут строиться входные тексты, подлежащие разбору. Правила задаются в форме, близкой БНФ:

имя_нетерминального_символа: определение;

здесь ´:´ и ´;´ - специальные символы **уасс**. Правая часть правила - определение - представляет собой упорядоченную последовательность элементов (нетерминальных символов и лексем), составляющих описываемую конструкцию. При грамматическом разборе такая последовательность в

результате применения правила заменяется нетерминальным символом, имя которого указано в левой части. Такая процедура программы грамматического разбора получила название свертка. Нетерминальные символы в определении задаются именами, лексемы - именами или литералами. Запись имен совпадает с записью идентификаторов и символьных констант, принятой в языке Си. Например, правило

P_HEAD: NAME ´(´ P_LIST ´)´;

определяет нетерминал **P_HEAD** как последовательность **4** элементов: два элемента (скобки) заданы литерально, два других (**NAME** и **P_LIST**) - именами. По виду правила нельзя заключить, относятся эти имена к лексемам или нетерминалам. Считаются именами нетерминалов все имена, не объявленные в секции деклараций лексемами. Нетерминалы должны быть определены, т.е. имя каждого из них должно появиться в левой части хотя бы одного правила. Допустимо задание нескольких правил, определяющих один нетерминал, т.е. правил с одинаковой левой частью. Ниже приводятся три правила, определяющих виды операторов в некотором языке:

statement : assign_stat ;
statement : if_then_stat;
statement : goto_stat;

Правила с общей левой частью можно задавать в сокращенной записи, без повторения левой части, используя для разделения альтернативных определений символ ″|″. Предыдущие правила можно переписать в виде:

```
statement :    assign_stat
          |    if_then_stat
          |    goto_stat;
```

С любым правилом можно связать действие - набор операторов языка Си, которые будут выполняться при каждом распознавании конструкции во входном тексте. Действие не является обязательным элементом правил, заключается в фигурные скобки и помещается вслед за определением

имя_нетерминального_символа:

определение {действие};

Точка с запятой после правила с действием может опускаться. На операторы, входящие в действия, не накладывается никаких ограничений. В

частности, в действиях могут содержаться вызовы любых функций на языке Си. Отдельные операторы могут быть помечены, к ним возможен переход из других действий.

Существует возможность задания действий, которые будут выполняться по мере распознавания отдельных фрагментов правила. Действие в этом случае помещается после одного из элементов правой части так, чтобы положение действия соответствовало моменту разбора, в который данному действию будет передано управление. Например, в правиле

```
if_then_stat: IF '(' expression ')' { действие_1 }
                    THEN statement    ';' { действие_2 }
```

действия заданы с таким расчетом, чтобы при разборе строки вида

```
if( a > b ) then x = a;
```

действие_1 выполнялось при нахождении правой круглой скобки, а действие_2 - по окончании разбора. Как и в декларациях, в разделе правил можно разместить описания Си-переменных, которые используются в действиях правил и определены на весь раздел правил. В начале раздела правил Си-переменные описываются как в разделе деклараций.

Укажем еще на два вида объектов, использующихся в действиях. Это, во-первых, глобальные переменные, которые описываются в секции деклараций, и, во-вторых, специальные позиционные переменные, облегчающие взаимосвязь между действиями и связь их с лексическим анализатором. Использование позиционных переменных рассмотрим ниже.

Структура входной информации описывается в наборе правил иерархически, т.е. каждое правило соответствует определенному уровню структурного разбиения. Однако последовательность задания правил может не отражать этой иерархии и быть вполне произвольной. Единственно необходимой для **уасс** является информация о том, какой из нетерминалов задает вершину иерархии,т.е. соответствует конструкциям, определяющим входной текст в целом. Этот нетерминал принято называть начальным символом. Приведение входного текста к начальному символу посредством сверток является целью грамматического анализа. По умолчанию **уасс** считает начальным символом тот нетерминал, имя которого стоит в левой части первого из правил. Однако, если определять начальный символ в первом правиле пользователю неудобно, он может явно задать имя начального символа в секции деклараций.

Существуют два специфических вида правил, на которые полезно

обратить внимание, - это пустое и рекурсивное. Пустое правило имеет вид:

```
имя_нетерминального_символа: ;
```

Сочетание пустого правила с другими, определяющими тот же нетерминальный символ, является одним из способов указать на необязательность вхождения соответствующей конструкции. С пустым правилом может быть обычным образом связано действие

```
ИМЯ: { действие } ;
```

Правила часто описывают некоторую конструкцию рекурсивно, т.е. правая часть может включать определяемый нетерминал. Различают леворекурсивные правила вида

```
имя_нетерминала : имя_нетерминала
                  многократно повторяемый фрагмент;
```

и праворекурсивные вида

```
имя_нетерминала :
                  многократно повторяемый фрагмент
                  имя_нетерминала;
```

Генератор **уасс** допускает оба вида рекурсивных правил, однако при использовании правил с правой рекурсией объем анализатора увеличивается, так как во время разбора возможно переполнение внутреннего стека анализатора. Рекурсивные правила необходимы при задании последовательностей и списков. Следующие примеры иллюстрируют универсальный способ описания последовательностей:

```
последовательность: элемент
                  | последовательность элемент;
```

и списков

```
список: элемент
      | список ',' элемент;
```

В каждом из этих случаев первое правило (не содержащее рекурсии) будет применено только для первого элемента, а второе (рекурсивное) - для всех последующих. Нетерминальные символы, связанные с

последовательностями или списками разнородных элементов, могут описываться произвольным числом рекурсивных и нерекурсивных правил. Например, группа правил

```
идентификатор:    буква
        | '$'
        | идентификатор буква
        | идентификатор цифра
        | идентификатор '_'   ;
```

описывает идентификатор как последовательность букв, цифр и символов "_", начинающуюся с буквы или символа "$". Следует обратить внимание на то, что рекурсивные правила не имеют смысла, если для определяемого ими нетерминала не задано ни одного правила без рекурсии. Для обеспечения возможности задания пустых списков или последовательностей в качестве нерекурсивного правила используется пустое:

```
последовательность : |
        последовательность элемент ;
```

Сочетание пустых и рекурсивных правил является удобным способом представления грамматик и ведет к большей их общности. Некорректное использование пустых правил может вызывать конфликтные ситуации из-за неоднозначности выбора нетерминала, соответствующего пустой последовательности.

В разделе функций помещаются описания функций на языке Си, которые должны быть включены в генерируемую программу грамматического анализа. Эти функции можно разместить также в отдельных файлах, которые подключаются на этапе вызова Си-компилятора для трансляции файла **y.tab.c** и компоновки программы. Перечислим функции, которые одним из этих способов должны быть заданы:

лексический анализатор

функция с именем **yylex()** (это имя предопределено);

функции,

вызовы которых содержатся в действиях правил;

главная функция **main()**,

ее стандартный библиотечный вариант имеет вид:

```
main() {
        return (yyparse());
}
```


функция обработки ошибок **yyerror()**

библиотечный вариант функции имеет вид:

```
# include       <stdio.h>

yyerror(s)
char *s;
{
        fprintf(stderr, "%s", s);
}
```

Библиотечный вариант функции **yyerror()** можно заменить своим.

5.7. Позиционные переменные в yacc-программе

В процессе выполнения программы грамматического разбора осуществляется так называемая свертка нетерминала. В результате свертки происходит замещение нетерминала его значением, полученным в результате выполнения правой части правила. Полученное значение нетерминала хранит предопределенная переменная с именем **$$**. Для обеспечения связи между действиями, а также между действиями и лексическим анализатором создаваемые **yacc** грамматические анализаторы поддерживают специальный стек, в котором сохраняются значения лексем и нетерминальных символов. Значение лексемы автоматически попадает в стек после ее распознавания лексическим анализатором (им считается текущее значение переменной **yylval**). После каждой свертки вычисляется значение нетерминала, заместившего свернутую строку, и помещается в вершину стека. Значения элементов правой части примененного правила перед этим выталкиваются из стека. Таким образом, к моменту свертки любого правила все значения нетерминалов в правой части оказываются вычисленными в результате сверток. Описанный механизм не требует вмешательства пользователя и предоставляет ему следующие возможности:

использование в действиях, осуществляемых после свертки по правилу, значения любого элемента его правой части. Доступ к этим значениям обеспечивается набором так называемых позиционных переменных с именами **$1**, **$2**,..., где **$i** соответствует значению i-го элемента правой части правила. Элементы правой части правила нумеруются слева направо без различий для лексем и нетерминалов. Например, в правиле

```
P_Head:  P_name '(' P_list ')';
```

псевдопеременные относятся

$1 к **P_name**,

$2 к '(',

$3 к **P_list**,

$4 к ')'

формирование в действиях значения образованного в результате свертки нетерминала путем присвоения этого значения псевдопеременной с именем **$$**. Так, в правиле

```
expr: expr '+' expr { $$ = $1 + $3; }
```

значением нового нетерминала **expr** станет сумма ранее вычисленных значений двух других нетерминалов **expr**. Если в действии не определяется значение переменной **$$** (а также если действие отсутствует), значением нетерминала после свертки по умолчанию становится значение первого элемента правой части, т.е. неявно выполняется присваивание

```
$$ = $1;
```

Пример (вычисление значения целого числа):

```
%token ЦИФРА
%%
        ...
        ...
КОНСТАНТА:  ЦИФРА
        | КОНСТАНТА ЦИФРА { $$ = $1 * 10 + $2; }
        ...
        ...
%%

yylex() {
int c;
        if(( c = getchar() ) >= '0' && c <= '9'){
                yylval = c - '0';
```



```
                return(ЦИФРА);
        }
        ...
        ...
}
```

Здесь при свертке по первому правилу нетерминал КОНСТАНТА получает значение первой цифры, присвоенное в функции **yylex()** переменной **yylval**. При каждой свертке по второму правилу явно вычисляется значение нового нетерминала КОНСТАНТА.

Несколько иная ситуация в отношении использования позиционных переменных имеет место для правил, содержащих действия внутри правой части. На самом деле **yacc** интерпретирует правило вида

```
A: B C { действие_1 } D { действие_2 }
        K { действие_3 }
```

как

```
A: B C EMPTY1 D EMPTY2 K {действие 3}

EMPTY1: { действие_1 }

EMPTY2: { действие_2 }
```

И в точке, где вставлено действие, при разборе осуществляется свертка по пустому правилу, сопровождающаяся выполнением указанного действия. При этом независимо от характера действия очередной элемент в стеке значений отводится для хранения значения неявно присутствующего "пустого" нетерминала.

5.8. Конфликты грамматического разбора и обработка ошибок

Грамматика является неоднозначной, если существует входная строка, которая в соответствии с этой грамматикой может быть разобрана двумя или более различными способами. Рассмотрим, например, набор правил, описывающих константное арифметическое выражение:

```
expr:   CONST                   /* 1 */
      | expr '+' expr           /* 2 */
      | expr '-' expr           /* 3 */
```

```
        | expr '*' expr          /* 4 */
        | expr '/' expr;         /* 5 */
```

Описывая возможность построения выражения из двух выражений, соединенных знаком арифметической операции, правила неоднозначно определяют путь разбора некоторых входных строк. Так, строка

expr * expr - expr

допускает два пути разбора, приводящих к различным группировкам ее элементов:

expr * (expr - expr)

(expr * expr) - expr

Здесь имеет место неоднозначность выбора действия при вводе лексемы '-' в момент, когда разобранная часть строки приведена к виду **expr * expr**. Неоднозначность такого рода будем называть конфликтом сдвиг/свертка. Два возможных действия анализатора состоят в следующем:

можно ввести следующий символ и без применения правила подстановки перейти в новое состояние;

можно сразу применить к конструкции правило и без ввода нового символа перейти в очередное состояние. Такое действие было названо сверткой.

Возможен другой вид конфликта, состоящий в выборе между двумя возможными свертками, будем называть его конфликтом свертка/свертка. Для примера подобного конфликта приведем грамматику, задающую десятичную и шестнадцатеричную формы записи константы:

```
const:        const_10              /* 1 */
            | const_16 ;            /* 2 */
const_10:     dec_sequence;         /* 3 */
const_16:     hex_sequence 'x';     /* 4 */
dec_sequence: digit                 /* 5 */
```



```
            | dec_sequense digit;     /* 6 */
hec_sequense: digit                   /* 7 */
            | ABCDEF                  /* 8 */
            | hex_sequense digit      /* 9 */
            | hex_sequence ABCDEF;    /* 10*/
ABCDEF :    'A'|'B'|'C'|'D'|'E'|'F';
digit:      '0'|'1'|'2'|'3'|'4'|'5'|'6'
            |'7'|'8'|'9';
```

После замены первой десятичной цифры нетерминалом **digit** возникает конфликт между двумя возможными свертками: к нетерминалу **dec_sequence** в результате применения правила (5) и к нетерминалу **hex_sequence** с помощью правила (7). В отличие от грамматики в предыдущем примере здесь не удается корректно разобрать какую-либо строку двумя способами.

Каждая ситуация, которая при разборе способна вызвать конфликт сдвиг/свертка или свертка/свертка, выявляется генератором **yacc** уже на этапе построения грамматического анализатора. При этом выводится сообщение о числе выявленных конфликтных ситуаций, а в выходной файл **y.output** (если он формируется) помещается подробное описание. Используются два способа разрешения конфликтов:

в случае конфликта сдвиг/свертка по умолчанию делается сдвиг;

в случае конфликта свертка/свертка по умолчанию делается свертка по тому правилу, которое задано первым в **yacc**-программе.

В ряде ситуаций описанный способ разрешения конфликтов приводит к нужному результату. Например, рассмотрим фрагмент грамматики языка программирования, описывающий условный оператор **IF**:

```
оператор:  IF '(' условие ')' оператор      /* 1 */
         | IF '(' условие ')' оператор ELSE
           оператор;                        /* 2 */
```

Входная строка

IF(C1) IF(C2) S1 ELSE S2

вызвала бы при разборе конфликт сдвиг/свертка в момент просмотра символа **ELSE**. Введенная часть строки к этому времени имеет вид:

IF(условие) **IF**(условие) оператор

Если выполнить свертку второй части конструкции по правилу (1), то строка будет:

IF(условие) оператор

Заметим, что применить еще раз правило(1) мешает просмотренный заранее символ **ELSE**. После ввода конструкции **S2** и замены ее нетерминалом "оператор" к строке

IF(условие) оператор **ELSE** оператор

будет применено правило (2). Полученный разбор соответствует следующей интерпретации входной строки:

IF(c1) { IF(c2) S1 } ELSE S2

В случае применения сдвига в момент появления **ELSE** входная строка была бы введена полностью:

IF(условие) **IF**(условие) оператор

ELSE оператор

Ко второй части строки можно применить правило (2), а затем свернуть полученную конструкцию

IF(условие) оператор

по правилу (1). Такой разбор соответствует второй возможной интерпретации входной строки:

IF(c1) { IF(c2) S1 ELSE S2}

Как известно, в большинстве языков программирования принята именно эта интерпретация (каждый **ELSE** относится к ближайшему предшествующему **IF**). Значит, выбор сдвига, осуществляемый по умолчанию, для данной грамматики верен.

В качестве примеров устранения конфликтов путем изменения правил приведем перестроенные варианты рассматривавшихся выше грамматик. Перестроенная грамматика константного арифметического выражения имеет вид:

```
expr:   expr1
      | expr '+' expr1
      | expr '-' expr1;
expr1: CONST
     | expr1 '*' CONST
     | expr1 '/' CONST;
```

Перестроенная грамматика для задания константы:

```
const:    const_10
        | const_16;
const_10: dec_sequence    ;
const_16: hex_sequence 'x'
        | dec_sequense 'x';
dec_sequence: digit
        | dec_sequence digit;
hex_sequence: ABCDEF
        | dec_sequence ABCDEF
sp
        | hex_sequence ABCDEF
```

```
    | hex_sequence dec_sequence;

ABCDEF: ...

digit:  ...
```

Рассмотрим теперь способ разрешения конфликтов, базирующийся на задании программистом информации о приоритетах и ассоциативности. Приоритетное разрешение конфликтов сдвиг/свертка состоит в том, что с обоими действиями генератор **yacc** ассоциирует приоритеты (со сдвигом - приоритет лексемы, чтение которой вызывает данный конфликт, со сверткой - приоритет конкурирующего правила) и выбирает более приоритетное действие. В случае равенства приоритетов генератор **yacc** осуществляет выбор на основе свойств ассоциативности. Приоритеты и ассоциативность отдельных лексем (явно) и правил (явно и неявно) задаются программистом, все остальные приоритеты считаются неизвестными. **Yacc** использует для разрешения конфликта данный способ, если известны приоритеты обоих конкурирующих действий. Поэтому для разрешения ряда конфликтов на приоритетной основе необходимо установить приоритеты участвующих в них лексем и правил. Однако задание приоритетов не ведет к устранению конфликтов и не делает грамматику однозначной.

Приоритет правила автоматически определяется приоритетом последней лексемы в нем. Если в разделе деклараций для этой лексемы не задан приоритет или если правая часть правила вообще не содержит лексем, то приоритет правила не определен. Этот принцип можно отменить явным заданием приоритета правила, равным приоритету любой (имеющей приоритет) лексемы, с помощью директивы

%prec <лексема>

помещенной вслед за правой частью правила (перед точкой с запятой или действием). Например, правилу

```
expr: '-' expr %prec '*';
```

директива **%prec** придает приоритет лексемы '*'.

Сформулируем теперь используемые в генераторе **yacc** правила разрешения конфликтов сдвиг/свертка на основе информации о приоритетах и ассоциативности (напомним, что конфликты свертка/свертка разрешаются только по принципу умолчания):

если для входной лексемы и правила заданы приоритеты и эти приоритеты различны, выбирается действие с большим приоритетом.

Больший приоритет правила вызывает свертку по нему, больший приоритет лексемы - сдвиг;

если приоритеты заданы и совпадают, принимается во внимание заданная одновременно с приоритетом ассоциативность: в случае левой ассоциативности используется свертка, в случае правой - сдвиг. Отсутствие свойства ассоциативности (директива **nonassoc**) в данном случае указывает на ошибку во входном тексте, и анализатор воспримет вызвавшую данный конфликт лексему как ошибочную;

если не задан приоритет входной лексемы и/или приоритет правила, действует принцип разрешения конфликтов по умолчанию, в результате чего выбирается сдвиг.

Стандартной реакцией грамматического анализатора на ошибку является выдача сообщения "синтаксическая ошибка" и прекращение разбора. Эту реакцию можно изменить, например, сделав сообщение об ошибке несколько более информативным, задав собственную процедуру **yyerror()**. Однако желательно, чтобы анализатор в этом случае продолжал просмотр входного потока. Для этого программисту необходимо ввести дополнительные правила, указывающие, в каких конструкциях синтаксические ошибки являются допустимыми. Одновременно эти правила определяют путь дальнейшего разбора для ошибочных ситуаций. Для указания точек допустимых ошибок используется зарезервированное с этой целью имя лексемы **error**. Пример:

```
A: B C D ;          /* 1 */

A: B C error;       /* 2 */

D: D1 D2 D3;        /* 3 */
```

Второе правило указывает путь разбора в случае, если при распознавании нетерминала **A** встретится ошибка после выделения элементов **B** и **C**. Правила, содержащие лексему **error**, выполняются так же, как все остальные правила. Анализатор, встретив ошибку, пытается найти ближайшую точку во входном потоке, где разрешена лексема **error**. При этом сначала делается попытка возврата в рамках правила, по которому шел разбор в момент появления ошибочной лексемы, затем поиск распространяется на правила более высокого уровня. Рассмотрим порядок работы анализатора при появлении во входном потоке ошибочной лексемы (т.е. лексемы, ввод которой в данном состоянии вызывает действие **error**):

фиксируется состояние ошибки и вызывается функция **yyerror** для выдачи сообщения;

путем обратного просмотра пройденных состояний,начиная с данного, делается попытка найти состояние, в котором допустима лексема **error**. Отсутствие такого состояния говорит о невозможности восстановления, и разбор прекращается;

осуществляется возврат в найденное состояние (кроме случая, когда им является непосредственно то состояние, в котором встретилась ошибка);

выполняется действие, заданное в этом состоянии для лексемы **error**. Очередной входной лексемой становится лексема, вызвавшая ошибку;

разбор продолжается, но анализатор остается в состоянии ошибки до тех пор, пока не будут успешно прочитаны и обработаны три подряд идущие лексемы;

после обработки трех допустимых лексем считается, что восстановление произошло, и анализатор выходит из состояния ошибки.

С правилами, включающими лексему **error**, могут быть связаны действия. С их помощью программист может самостоятельно обработать ошибочную ситуацию. Кроме обычных операторов, здесь можно использовать специальные операторы **yyerror** и **yyclearin**. Оператор **yyerror** аннулирует состояние ошибки. Таким образом, можно отменить действие принципа "трех лексем". Оператор **yyclearin** стирает хранимую анализатором последнюю входную лексему. Приведем общую форму правила с восстановительным действием

```
оператор : error {
                resynch();
                yyclearin;
                yyerror;
        }
```

Предполагается, что функция **resynch()** просматривает входной поток до начала очередного оператора. Вызвавшая ошибку лексема, хранимая анализатором в качестве входной лексемы, стирается, после этого гасится состояние ошибки. При построении анализаторов, работающих в

интерактивном режиме, для обработки ошибок рекомендуются правила вида

```
входная_строка : error '\n' {
                        yyerrok;
                        printf("Повторите ввод строки \n");
                }
                входная_строка { $$ = $4; }
```

В действии, предусмотренном после ввода ошибочной строки, пользователю выводится приглашение, а состояние ошибки гасится. Значением нетерминала после свертки здесь становится значение повторно введенной строки.

5.9. Совместное использование lex и yacc

В yacc-программе лексический анализатор оформляется в виде функции **yylex()**, которая возвращает номер типа распознанной лексемы. Для нелитеральных лексем номером типа может служить объявленное в секции деклараций имя лексемы (с помощью препроцессора Си-компилятора в yacc-программе обеспечивается замена его нужным номером), в случае литералов номером типа является числовое значение символа. Пример функции **yylex()**, написанной на языке Си:

```
# include       <stdio.h>

yylex(){
register int c;

        while((c=getc(stdin)) == ' ' || c == '\n');

        if(c >= '0' && c <= '9'){

                while((c=getc(stdin)) >= '0'
                        && c <= '9');

                ungetc(c, stdin);

                return(CONST);
```

```
        }
        if (c >= 'a' && c <= 'z'){

                while((c=getc(stdin)) >= 'a'
                        && c <= 'z'
                        || c >= '0'
                        && c <= '9');

                ungetc(c, stdin);

                return (NAME);
        }
        return (c);
}
```

Сложность лексического анализатора зависит от того, какие структурные единицы взяты за основу при описании грамматических правил. Детализировав грамматику до отдельных символов, можно обойтись простейшим лексическим анализатором, осуществляющим только их ввод:

```
yylex(){
        return( getchar() );
}
```

В этом случае число правил в **yacc**-программе растет, а грамматический разбор оказывается менее эффективным. Поэтому программист обычно должен найти некоторый компромисс при выборе набора лексем. В процедуре лексического анализа кроме выделения лексем можно предусмотреть некоторую обработку лексем определенных типов, в частности запоминание конкретных значений лексем. Кроме того, эти значения обычно требуется передать грамматическому анализатору. С этой целью нужное значение должно быть присвоено внешней переменной целого типа с именем **yylval** (это имя предопределено). Если функция **yylex()** находится в отдельном файле, то она должна быть объявлена как **extern**. Уточним, что в дальнейшем значением лексемы мы будем называть значение, присвоенное при ее распознавании переменной **yylval**. Значение, возвращаемое функцией **yylex()**, является номером типа лексемы. Примером значения лексемы могут служить значение цифры, вычисленное значение константы, адрес идентификатора в таблице имен (построение таблицы имен удобно осуществлять лексическим анализатором). Заметим, что, хотя значение **yylval** устанавливается с целью использования его в действиях, непосредственное обращение к переменной **yylval** в действии не имеет смысла

(поскольку в **yylval** всегда находится значение последней выделенной лексемы).

Все виды лексем, кроме литералов, обозначаются некоторыми именами и под этими именами фигурируют в **yacc**-программе, где объявление имен лексем осуществляется директивой **token**, благодаря чему генератор **yacc** отличает имена лексем от имен нетерминальных символов.

Для каждого имени лексемы независимо от того, переопределен ли ее номер программистом, **yacc** генерирует в выходном файле **y.tab.c** оператор препроцессора:

 # define <имя лексемы> <номер типа>

Значение, возвращаемое функцией **yylex()**, является номером типа лексемы. Список лексем и номера их типов указываются в **yacc**-программе, а определения этих лексем в - **lex**-программе. Возникает проблема соответствия номеров типов лексем в файлах **y.tab.c** и **lex.yy.c**, которая разрешается следующим образом:

при вызове **yacc** с ключом **-d** последовательность операторов **define** помещается в файл **y.tab.h**.;

этот файл посредством оператора **include** включается в **lex**-программу.

В процедуре лексического анализа, кроме выделения лексем, можно предусмотреть некоторую обработку лексем определенных типов, в частности запоминание конкретных значений лексем. Если функция **yylex()** находится в отдельном файле, переменная **yylval** должна быть объявлена:

```
extern int yylval;
```

Допустим, мы располагаем **yacc**-программой в файле **source.y** и **lex**-программой в файле **source.l**, которые необходимо собрать в работающую программу. Существуют два способа сборки:

с созданием файла **y.tab.h**;

без создания файла **y.tab.h**.

Рассмотрим первый способ сборки. Ниже приведен пример **Makefile**, посредством которого **make** осуществляет последовательную обработку и сборку этих программ и размещает результат в файле **program**:

```
program:        y.tab.o lex.yy.o
        cc y.tab.o lex.yy.o -ly -ll -o program

y.tab.o:        y.tab.c
                cc -c -O y.tab.c

lex.yy.o:       lex.yy.c  y.tab.h
                cc -c -O lex.yy.c

y.tab.h:

y.tab.c:        source.y
                yacc -d source.y

lex.yy.c:       source.l
                lex -v source.l
```

Пример 1. В файле source.l размещена yacc-программа, реализующая небольшой настольный калькулятор. Эта программа заимствована из работы [13] и приводится с некоторыми изменениями. Калькулятор имеет 52 регистра, помеченных буквами от A до z, и разрешает использовать арифметические выражения, содержащие операции +, -, *, /, % (остаток от деления), & (побитовое и), | (побитовое или) и присваивание. Как и в языке Си, целые числа, начинающиеся с 0, считаются восьмеричными, все остальные - десятичными. Результат всегда выводится десятичными числами. Калькулятор работает в интерактивном режиме с построчным формированием выхода, может читать задание из файла и выводить результат в файл. Знак "=" используется для присваивания, а для вывода результата достаточно нажать клавишу <ВК>. Распознаются скобочные структуры, изменяющие порядок приоритетов в вычислениях. Калькулятор работает только с переменными типа integer:

```
%token DIGIT-LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS
%{
int base, regs[52];
%}
%%
```



```
list:
    | list stat '\n'
    | list error '\n' { yyerrok; }

stat:
       expr { printf( "%d\n",$1 ); }
    | LETTER '=' expr { regs[$1] = $3; }

expr: '(' expr ')'    { $$ = $2;  }
    |     expr '+' expr  { $$ = $1 + $3; }
    |     expr '-' expr  { $$ = $1 - $3; }
    |     expr '*' expr  { $$ = $1 * $3; }
    |     expr '/' expr  { $$ = $1 / $3; }
    |     expr '%' expr  { $$ = $1 % $3; }
    |     expr '&' expr  { $$ = $1 & $3; }
    |     expr '|' expr  { $$ = $1 | $3; }
    | '-' expr %prec UMINUS { $$ = -$2; }
    |    LETTER  { $$ = regs[$1]; }
    |    number;

number:
     DIGIT  { $$ = $1;
              base = 10;
              if($1 == 0) base = 8;  }
    | number DIGIT  { $$ = base * $1 + $2; }
```

В файле source.l размещена lex-программа анализатора для этого калькулятора:

```
%{
#include   "y.tab.h"
extern int yylval;
%}
%%
^\n                ;
[ \t]*             ;
[A-Za-z]     {
       yylval = yytext[yyleng-1] - 'a';
      return(LETTER);}

[0-9]                {
```

```
        yylval = yytext[yyleng-1] - '0';
        return(DIGIT);}
```

Рассмотрим **Make**-программу для сборки без использования файла **y.tab.h**:

```
program:        y.tab.c lex.yy.c
        cc -O y.tab.c -ly -ll -o program

y.tab.c:        source.y
                yacc  source.y

lex.yy.c:       source.l
                lex -v source.l
```

В файлах **source.y** и **source.l** произойдут следующие изменения. В разделе деклараций **yacc**-программы необходимо указать строку

```
    # include  "lex.yy.c"
```

из **lex**-программы необходимо убрать строку

```
    # include  "y.tab.h"
```

Теперь файл **source.y** выглядит следующим образом:

```
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS
%{
#include "lex.yy.c"
int base, regs[52];
%}
%%
```

раздел правил **yacc**-программы.

Файл **source.l** выглядит следующим образом:

```
%{
```



```
extern int yylval;
%}
%%
```

раздел правил **lex**-программы.

Пример **2**. Программа, преобразующая исходную систему условий задачи линейного программирования в каноническую форму (лексический анализатор написан на Си). Исходная информация для задачи линейного программирования представляется системой уравнений и функционалом:

```
a[1][1]x[1] + a[1][2]x[2] + ... + a[1][n]x[n]  =  b[1]
          ...
          ...
a[m][1]x[1] + a[m][2]x[2] + ... + a[m][n]x[n]  =  b[m]

c[1]x[1]    + c[2]x[2]    + ... + c[n]x[n] -> min(max)
```

и преобразуется в матричную:

```
a[1][1]     a[1][2]     ...   a[1][n]
          ...
          ...
a[m][1]     a[m][2]     ...   a[m][n]
b[1]        b[2]        ...   b[m]
c[1]        c[2]        ...   c[n]
```

При этом изменяются знаки коэффициентов при неизвестных в ограничениях с отрицательным свободным членом, а также знаки коэффициентов функционала в случае его максимизации. Предусмотрена возможность повторного ввода ошибочных строк:

```
%token число Xi оптим
%%
КАНФОРМ:   функционал '\n' система_ограничений
                                        { final(); }
         | система_ограничений функционал '\n'
                                        { final(); }

функционал:  линейная_функция { stf(); }
/*
**  По умолчанию - минимизация
*/
```

```
        | оптим '('линейная_функция')'
                { if($1) conv(); stf(); }

/*
В случае максимизации выполнить conv()
*/

        | линейная_функция '-''>' оптим
                { if($4) conv(); stf(); }

линейная_функция: элем1
        | линейная_функция элем ;

элем:       знак число Xi   {stc($1 * $2, $3); }
        | знак Xi '*' число { stc($1 * $4, $2); }
        | знак Xi           { stc($1, $2);}


/*
**  Формы записи коэффициентов
*/
элем1:      элем
        | число Xi      { stc($1, $2); }
        | Xi '*' число  { stc($3, $1); }
        | Xi            { stc( 1, $1); }

знак:       '+' { $$ =  1; }
        | '-' { $$ = -1; }

система_ограничений: ограничение '\n'
        | система_ограничений ограничение '\n'
        | система_ограничений error '\n'
          { aclear(); yyerrok;
          printf("повторите последнюю строку\n");}

/*
**  В случае ошибки: стирание информации о строке,
**  восстановление и выдача приглашения
*/

ограничение: линейная_функция '=' число  { stcb($3); }
        | линейная_функция '=' знак число
```



```
/*
**  Если bi < 0, выполнить conv()
*/

                { if($3 < 0) conv(); stcb($4); }
%%

/*
**  Лексический анализатор возвращает:
**  для целых чисел - ЧИСЛО,
**  (yylval равно значению числа);
**  для идентификаторов вида x[i], i = [1,XI]
**  (yylval равно его порядковому номеру);
**  для max/min - ОПТИМ,
**  (yylval  равно соответственно 1 или 0).
*/

#include "stdio.h"
# define MAXM 100   /* предельное число ограничений */
# define MAXN 100   /* и переменных                 */

int c[MAXN], b[MAXM], a[MAXM+1][MAXN];
int neqs, nx, x[MAXN];

yylex(){
char c, i;
    while((c=getc(stdin)) == ' ' );

    switch(c){

        case'0': case'1': case'2': case'3': case'4':
        case'5': case'6': case'7': case'8': case'9':
            yylval = c -'0';

            while((c=getc(stdin)) <= '9' && c >= '0')
                    yylval = yylval * 10 + c -'0';

            ungetc(c, stdin);
            return( число );

        case'm':
            if((c=getc(stdin)) == 'i')
```

```
            yylval = 0;
        else if(c == 'a')
                    yylval = 1;
             else
                    return('m');

        while((c=getc(stdin)) <= 'z' && c >= 'a');

        ungetc(c, stdin);
        return( оптим );

case'X': case'x':
        if((c=getc(stdin)) < '0' || c > '9')
             return('x');

        yylval = 0;

        while(c <= '9' && c >= '0'){
             yylval = yylval * 10 + c -'0';
             c = getc(stdin);
        }
        ungetc(c, stdin);

             for(i = 0; i < nx; i++)
                  if(x[i] == yylval){
                       yylval = i;
                       return(Xi);
                  }
             x[nx] = yylval;
             yylval = nx++;
             return(Xi);
        }
        return(c);
}

/*
**      Вывод элементов
*/

final(){
char i, j;
```



```
        printf("c:\n");
        for(i = 0; i < nx; i++) printf("%8d", c[i]);
        printf("\n a:\n");

        for(i = 0; i < neqs; i++){
             for(j = 0; j < nx; j++)
                       printf("%8d", a[i][j]);

             printf("\n");
        }
        printf("b:\n");
        for(i = 0; i < neqs; i++) printf("%8d", b[i]);
}

/*
**  Изменение знаков коэффициентов
*/
conv(){
char i;
        for(i = 0;i < nx; i++) a[neqs][i] *= (-1);
}

/*
**  Запоминание коэффициентов функционала
*/
stf(){
char i;
        for(i = 0 ; i < nx; i++) {
                 c[i] = a[neqs][i];
                 a[neqs][i] = 0;
        }
}

/*
**  Запоминание очередного коэффициента
*/
stc(nmb,adr)
int nmb, abr;
{
        a[neqs][adr] = nmb;
}
```

```
/*
**  Запоминание свободного члена
*/
stcb(nmb){
        b[neqs++] = nmb;
}

/*
**  Удаление ошибочной строки
*/
aclear(){
        char i;
        for(i = 0; i < nx; i++)
                a[neqs][i] = 0;
}
```

СПИСОК ЛИТЕРАТУРЫ

1. Банахан М., Раттер Э. Введение в операционную систему UNIX. - М.: Радио и связь, 1986.

2. Баурн С. Операционная система UNIX.- М.: Мир, 1986.

3. Браун П. Введение в операционную систему UNIX. - М.: Мир, 1987.

4. Готье Р. Руководство по операционной системе UNIX. - М.: Финансы и статистика, 1985.

5. Диалоговая единая мобильная операционная система ДЕМОС. - Калинин: ЦЕНТРПРОГРАММСИСТЕМ, 1985.

6. Инструментальная мобильная операционная система ИНМОС/ М.И. Беляков, А.Ю. Ливеровский, В.П. Семик и др. - М.: Финансы и статистика, 1985.

7. Керниган Б., Ритчи Д., Фьюер А. Язык программирования Си. Задачи по языку Си. - М.: Финансы и статистика, 1985.

8. Кристиан К. Введение в операционную систему UNIX. - М.: Финансы и статистика, 1985.

9. Хенкок Л., Кригер М. Введение в программирование на языке СИ. - М.: Радио и связь, 1986.

10. Aho A. V., Kernighan Brian V. W., Weinberger Peter J. AWK - a pattern scanning and processing language. Second edition. UNIX Programmers manual, 42 BSD, 1980. Bell Laboratries: Murray Hill, New Jersey, 1978.

11. Feldman S. I. Make - a program maitaining computer programms. Bell Laboratries: Murray Hill, New Jersey, 1978.

12. Joy W. N. An introduction the UNIX C-shell. UNIX Programmers manual, 42 BSD, 1980. Bell Laboratries: Murray Hill, New Jersey, 1978.

13. Johnson S. C. YACC - yet another compiler-compiler. Comp. Sci. tech. rep. N 32. Bell Laboratries: Murray Hill, New Jersey, 1975.

14. Lesk M. E. Lex - lexical analyzer generator. Comp. Sci. tech. rep. N 39. Bell Laboratries: Murray Hill, New Jersey, 1975.

СОДЕРЖАНИЕ

Тихомиров В.П., Давидов М.И.

Т46 Операционная система ДЕМОС: Инструментальные средства программиста.-М.:Финансы и статистика,1988.-206с.:ил.
ISBN5-279-00113-9.

Операционная система ДЕМОС принадлежит семейству унифицированных операционных систем, совместимых с ОС UNIX, и используется в СССР на серийных ЭВМ общего назначения.

Книга содержит описание ключевых инструментальных средств программиста.

Для пользователей ЭВМ, разработчиков программного обеспечения систем обработки информации, а также для аспирантов и студентов вузов.

Т $\frac{2405000000\text{-}057}{010(01)\text{-}88}$ 116-88 **ББК 32.973.2-01**

Практическое руководство

Владимир Павлович Тихомиров
Михаил Изгиявич Давидов

ОПЕРАЦИОННАЯ СИСТЕМА ДЕМОС: ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПРОГРАММИСТА

Зав. редакцией *И.Г. Дмитриева*
Редактор *Т.А. Петрова*
Худож. редактор *С.Л. Витте*
Техн. редактор *Г.А. Полякова*
Корректор *Т.М. Васильева*
Обложка художника *М.А. Закирова*

ИБ № 2235

Подписано в печать 16.04.88 А11184
Формат 60×88 1/16, Бум. офсет. № 1. Гарнитура «Литературная»
Печать офсетная. Усл.печ.л. 12,74. Усл.кр.-отт. 12,99.
Уч.-изд.л. 9,68. Тираж 15000 экз. Заказ 1254. Цена 50 коп.

Издательство "Финансы и статистика",
101000, Москва, ул. Чернышевского, 7

Набрано на персональной ЭВМ в системе "Астра-Н",
разработанной совместным отделом НИИЦЭВТ-ГИВЦ
Госкомиздата СССР

Отпечатано в типографии № 4 Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли,
129041, Москва, Б. Переяславская, 46

50 коп.