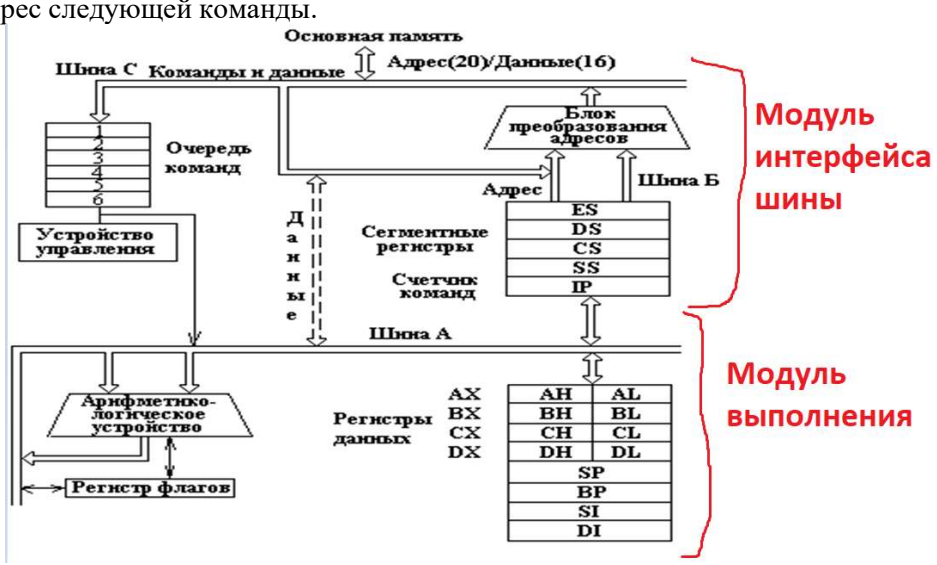
Варианты 1-го вопроса билета:

1. Процессор i8086. Структура, основные регистры и взаимодействие частей в процессе функционирования.

Он состоял из двух модулей – модуля выполнения и модуля интерфейса шины, их взаимодействие происходит через шину данных.

Модуль выполнения содержит АЛУ, предназначенный для арифметических операций. Также он содержит регистр флагов, хранивший состояния операций в аккумуляторе (6 флагов состояний и 3 системных) и 8 регистров общего назначения, которые используются для передачи данных через шину.

Модуль интерфейса отвечает за обработку всех данных, отправки инструкций в модуль выполнения, считывание и запись данных в память компьютера. Он способен хранить до 6 байт инструкций в буфере. Сегментные регистры отвечают за отправку данных в память компьютера. IP содержит адрес следующей команды.



Всего в состав микропроцессора i8086 входят четырнадцать 16-ти битовых регистров, каждый из которых может иметь специальное назначение, описанное далее:

а) четыре регистра общего назначения (называемые также регистрами данных):
- AX – регистр-аккумулятор,
- BX – базовый регистр,
- CX – счетчик,
- DX – регистр-расширитель аккумулятора;

б) три адресных регистра, которые должны использоваться для хранения частей адреса данных:
- SI – регистр индекса источника,
- DI – регистр индекса результата,
- BP – регистр-указатель базы;

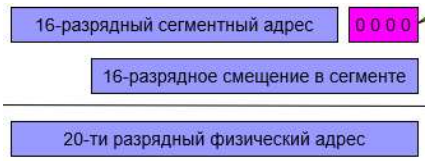
в) три управляющих регистра:
- SP – регистр-указатель стека,
- IP – регистр-счетчик команд,
- Flags – регистр флагов;

г) четыре сегментных регистра:
- CS – регистр сегмента кодов,
- DS – регистр сегмента данных,
- ES – регистр дополнительного сегмента данных,
- SS – регистр сегмента стека.

2. Процессор i8086. Адресация оперативной памяти i8086.

Этот процессор содержал 16-ти разрядную модель памяти. Адресация выглядела

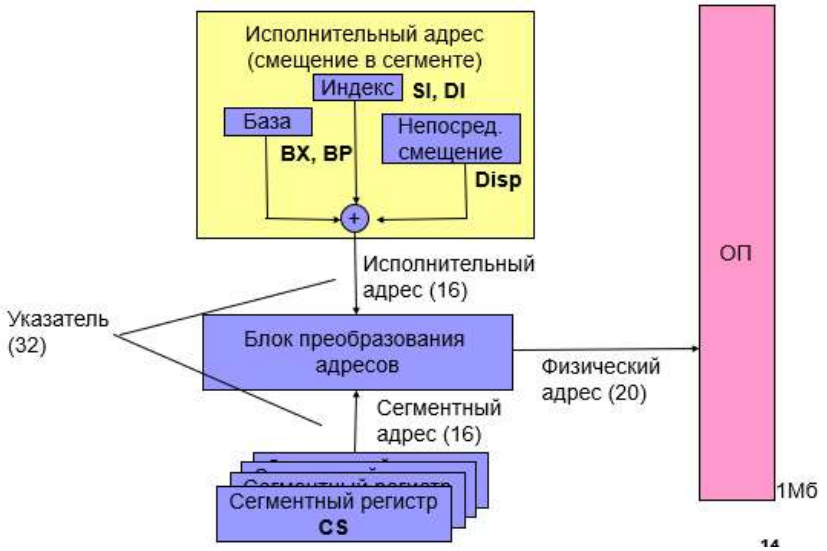
так: $A_{физический} = A_{базы} + A_{смещения}$, где A – адрес.



Физический адрес любых данных в памяти формируется из 16-ти битового смещения и 16-ти битового сегментного адреса по специальной схеме. Сначала к сегментному адресу аппаратно дописываются 4 двоичных нуля. В результате получается 20-ти битовый физический адрес начала сегмента. Затем выполняется сложение 20-ти битового базового адреса сегмента и 16-ти битового смещения. Откуда получается 20-ти битовый физический адрес данных.

Для размещения программ и данных в основной памяти выделяются специальные области – сегменты. Сегмент при 16-ти разрядной адресации – фрагмент памяти, начинающийся с адреса кратного 16 и имеющий размер от 1 байта до 64 Кб. Следовательно, базовый адрес сегмента всегда содержит в 4-х младших разрядах нули. Старшая часть адреса сегмента без последних четырех нулей называется сегментным адресом и хранится в одном из 4-х сегментных регистров. При этом каждый сегментный регистр используется для хранения адреса определенного сегмента:

- сегмент кода (CS:IP, где CS – сегмент адреса, а IP – смещение в сегменте)
- сегмент стека (SS:SP)
- основной и дополнительный сегменты данных (BX + <непосредственное смещение>, SI + <непосредственное смещение>, где BX – база, а SI – индекс).



3. Процессор i8086. Адресация сегментов различных типов.

Механизм управления памятью полностью аппаратный, т.е. программа сама не может сформировать физический адрес памяти на адресной шине.

В сегментированной модели память для программы делится на непрерывные области памяти, называемые сегментами. Программа может обращаться только к данным, которые находятся в этих сегментах. Сегмент представляет собой блок памяти.

К основным сегментам относятся:

Сегмент кодов (.CODE) – содержит машинные команды для выполнения. Обычно первая выполняемая команда находится в начале этого сегмента, и операционная система передает управление по адресу данного сегмента для выполнения программы. Регистр сегмента кодов (CS) адресует данный сегмент.

Сегмент данных (.DATA) – содержит определенные данные, константы и рабочие области, необходимые программе. Регистр сегмента данных (DS) адресует данный сегмент.

Сегмент стека (.STACK). Стек содержит адреса возврата как для программы (для возврата в операционную систему), так и для вызовов подпрограмм (для возврата в главную программу). Регистр сегмента стека (SS) адресует данный сегмент. Адрес текущей вершины стека задается регистрами SS:ESP.

Для адресации данных можно одновременно использовать два сегмента основной (адресуется DS) и дополнительный (адресуется ES).

4. Процессор i8086. Структура машинной команды. Примеры.

Префиксы	Код операции	1 байт адресации	2 байта смещения	2 байта данных	
Регистр/память <-> регистр	100010DW	Mod Reg R/M	Смещение млад. байт	Смещение стар. байт	Данные

Префиксы:

1) повторений - используется только для команд обработки строк и будет рассмотрен далее.

2) размер адреса (67h) - применяется для изменения размера смещения, например, если необходимо использовать смещение размером 16 бит при 32-х разрядной адресации

3) размер операнда (66h (AX)) - указывается, если вместо 32-х разрядного регистра для хранения операнда используется 16-ти разрядный

4) замены сегмента (26h) - используется при адресации данных любым сегментным регистром кроме DS

D – направление обработки, 1 – в регистр, 0 – из регистра.
W – размер операнда 1 – двойные слова, 0 – байты.

	W=1	W=0	Sr	
Reg	000 AX	000 AL	00	ES
	001 CX	001 CL	01	CS
	010 DX	010 DL	10	SS
	011 BX	011 BL	11	DS
	100 SP	100 AH		
	101 BP	101 CH		
	110 SI	110 DH		
	111 DI	111 BH		

Mod – режим: 00 – Disp = 0 – смещение в команде 0 байт
01 – Disp = 1 – смещение в команде 1 байт
10 – Disp = 2 – смещение в команде 2 байт
11 – операнды-регистры

Формат команд процессора i8086 позволяет указывать в команде только один операнд, размещенный в основной памяти, т.е. одной командой нельзя, например, сложить содержимое двух ячеек памяти.

Принципиально допускается 8 способов задания исполнительного адреса операндов, размещенных в основной памяти:

```
M = 000   EA=(BX)+(SI)+Disp
    001   EA=(BX)+(DI)+Disp
    010   EA=(BP)+(SI)+Disp
    011   EA=(BP)+(DI)+Disp
    100   EA=(SI) + Disp
    101   EA=(DI) + Disp
    110   EA=(BP) + Disp *
    111   EA=(BX) + Disp
```

Во всех случаях исполнительный адрес операнда определяется как сумма содержимого указанных регистров и смещения, заданного в команде и представляющего собой одно- или двухбайтовое число.

Примеры:

1) mov CX,AX

Отображение этой команды в отладчике показана на рисунке 16:

```
004010F9 | . 66:8BC8          MOV CX,AX
```

Рисунок 16 – команда mov CX,AX

0110 0110 100010 11 11 001 000

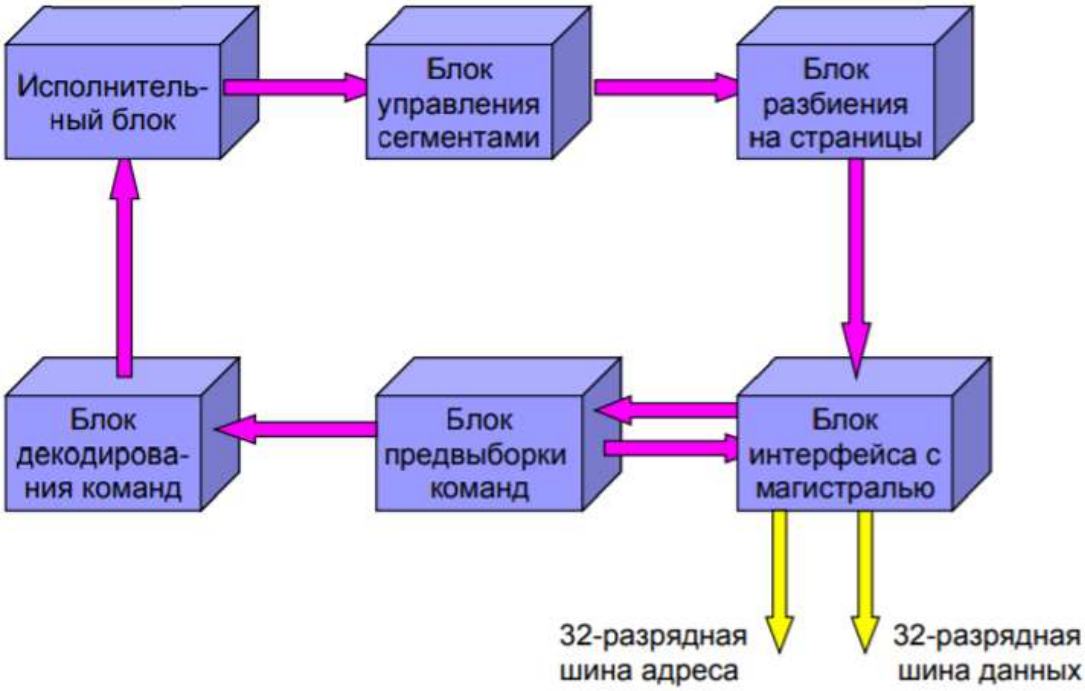
66h mov DW mod reg reg

D = 1 – в регистр, W = 1 – двойное слово

5. Процессор IA-32. Программная модель.

Структура процессора семейства IA-32 сложна, поскольку в них аппаратно реализована совокупность параллельных конвейеров. На рисунке процессор IA-32 представлен в виде набора основных блоков. Блок интерфейса с магистралью управляет передачей команд и данных из памяти в процессор и результатов – обратно в оперативную память. Блок предвыборки команд отвечает за чтение последующих команд из сегмента кодов. Блок декодирования команд осуществляет расшифровку команды и формирование последовательности управляющих сигналов для ее выполнения (аналог УУ). Исполнительный блок, согласно названию, выполняет команду (аналог АЛУ).

Блоки управления сегментами и страницами обеспечивают формирование физического адреса следующих команд и необходимых данных. При этом ограничение на нахождение не более одного операнда в оперативной памяти сохраняется



Он может функционировать в одном из трех режимов:

1) Реальной адресации – работает в режиме эмуляции 8086 с возможностью использования 32-битных расширений

2) Защищенном – работает с 32-х разрядными адресами, может использовать виртуальное пространство и обеспечивает защиту

3) Управления системой – предназначен для выполнения некоторых действий с возможностью их полной изоляции от прикладного программного обеспечения и даже от операционной системы. Микропроцессор переходит в этот режим только аппаратно

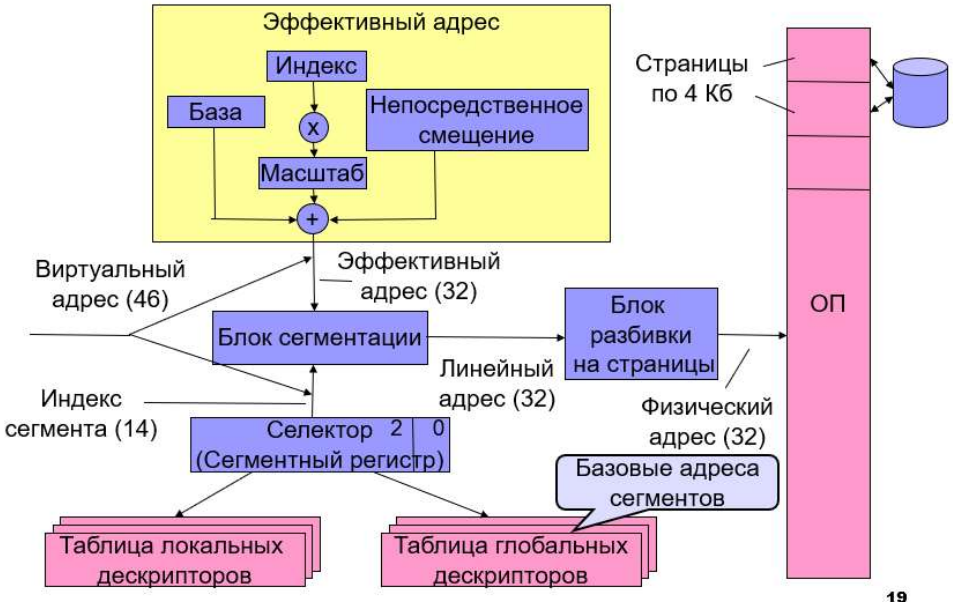
6. Процессор IA-32. Режимы адресации оперативной памяти. Схема адресации в защищенном режиме.

Режимы адресации ОП:

1) Реальной адресации – работает в режиме эмуляции 8086 с возможностью использования 32-битных расширений

2) Защищенном – работает с 32-х разрядными адресами, может использовать виртуальное пространство и обеспечивает защиту

3) Управления системой – предназначен для выполнения некоторых действий с возможностью их полной изоляции от прикладного программного обеспечения и даже от операционной системы. Микропроцессор переходит в этот режим только аппаратно



В этом режиме по-прежнему используется сегментная организация памяти, но размер сегмента уже не ограничивается 64 Кб, а теоретически может достигать 4 Гб. 32-х разрядный адрес базы сегмента хранится не в виде сегментного адреса в сегментном

32-х разрядный адрес базы сегмента хранится не в виде сегментного адреса в сегментном регистре, как при 16-ти разрядной адресации, а полностью в специальных внутренних регистрах процессора – дескрипторах. Номер дескриптора заносится в 14 бит сегментного регистра, который в этом режиме называется селектором. Один бит селектора из этих 14-ти отвечает за выбор таблицы локальных или глобальных дескрипторов.

Таблица локальных дескрипторов содержит дескрипторы сегментов приложения, а таблица глобальных – дескрипторы сегментов программ операционной системы. Оставшиеся два бита селектора содержат код уровня привилегий сегмента, который проверяется при обращениях из других программ. Таким образом, реализуется защита сегментов.

Эффективный адрес целиком получается из команды. В зависимости от того, как написаны операнды в команде могут использоваться все четыре части эффективного адреса: база, индекс, масштаб, непосредственное смещение. 32-х разрядный эффективный адрес складывается с базовым адресом сегмента.

Сумма 32-х разрядного базового адреса сегмента и 32-х разрядного эффективного адреса образует 32-х разрядный линейный адрес. Физический же адрес определяется по таблице страниц на основе линейного.

7. Процессор IA-32. Структура машинной команды. Байты изменения длины операнда и адреса. Примеры.

Размер машинной команды процессора IA-32 колеблется от 1 до 15 байт. Помимо обязательного кода операции (КОП), иногда состоящего из двух частей, команда может включать от 0 до 4 однобайтовых префиксов, а также возможно байты адресации, непосредственного смещения (смещение, указанное в команде) и непосредственного операнда.

<table>
<tr><td>Кол-во байт</td><td colspan="3">1 или 2</td><td colspan="3">0 или 1</td><td>0, 1, 2 или 4</td><td>0, 1, 2 или 4</td></tr>
<tr><td>Код операции …dw</td><td colspan="3">Байт mod r/m</td><td colspan="3">Байт sib</td><td>Смещение в команде</td><td>Непосредственный операнд</td></tr>
<tr><td></td><td>mod</td><td>Reg/КОП</td><td>r/m</td><td>ss</td><td>index</td><td>base</td><td></td><td></td></tr>
</table>

Префиксы:

1) повторений - используется только для команд обработки строк и будет рассмотрен далее: REP, REPE/REPZ, REPNE/REPNZ – используются с командами обработки цепочек символов

2) размер адреса (67h) - применяется для изменения размера смещения, например, если необходимо использовать смещение размером 16 бит при 32-х разрядной адресации

3) размер операнда (66h (AX)) - указывается, если вместо 32-х разрядного регистра для хранения операнда используется 16-ти разрядный

4) замены сегмента (26h) - используется при адресации данных любым сегментным регистром кроме DS

D – направление обработки, 1 – в регистр, 0 – из регистра.

W – размер операнда 1 – двойные слова, 0 – байты. Если 1, то EAX и т.д., если 0, то AL и т.д..

	W=1	W=0	Sr	
Reg	000	AX	000 AL	00 ES
	001	CX	001 CL	01 CS
	010	DX	010 DL	10 SS
	011	BX	011 BL	11 DS
	100	SP	100 AH	
	101	BP	101 CH	
	110	SI	110 DH	
	111	DI	111 BH	

Mod – режим: 00 – Disp = 0 – смещение в команде 0 байт
01 – Disp = 1 – смещение в команде 1 байт
10 – Disp = 2 – смещение в команде 2 байт
11 – операнды-регистры

Sib используется в тех случаях, если есть смещение. SS – масштаб, index – содержание индексного регистра, base – содержание базового регистра

Пример:

```
mov   ECX,6[EBX+EDI*4]
```

8B	4C	B B	0 6

100010DW Mod Reg Mem SS Ind Base См.мл.байт

10001011 01 001 100 10 111 011 00000110

Примеры:

2) mov BX,CX

префикс1	**100010DW**	**Mod**	**Reg**	**Reg**
01100110	10001001	11	001	011
6 6	8 9		C	B

3) mov ECX,DS:6[EBX]

100010DW	**Mod**	**Reg**	**Reg**	**См.мл.байт**
10001011	01	001	011	00000110
8 B		4	B	0 6

8. Структура программы на Masm32. Директивы объявления данных. Примеры.

В программе могут присутствовать предложения четырех типов:

- машинные команды ассемблера
- директивы ассемблера
- макрокоманды
- комментарии.

Программа состоит из нескольких сегментов:

1) сегмент данных

А) сегмент констант (постоянные) - объявление данных, которые в программе не меняются

Б) сегмент инициализированных данных (имеют значение) - объявление данных, которые в программе могут меняться

В) сегмент неинициализированных данных (выделяется память) - память под эти данные отводится во время загрузки программы на выполнение

2) сегмент кода (содержит команды ассемблера)

3) сегмент стека

В программе сегменты описываются полными или сокращенными директивами.

Сокращенные директивы описания сегмента кодируются следующим образом:

1. **.CODE [Имя сегмента]** – начало или продолжение сегмента кода;
2. **.MODEL Модель [Модификатор][,Язык][,Модификатор языка]**

где **Модель** – определяет набор и типы сегментов; при 32-х разрядной адресации используется единственная модель FLAT;

Модификатор – определяет тип адресации: use16, use32, dos;

Язык и Модификатор языка – определяют особенности передачи параметров при вызове подпрограмм на разных языках C, PASCAL, STDCALL;

3. **.DATA** – начало или продолжение сегмента инициализированных данных;
4. **.DATA?** – начало или продолжение сегмента неинициализированных данных;
5. **.CONST** – начало или продолжение сегмента неизменяемых данных;
6. **.STACK [Размер]** – начало или продолжение сегмента стека.

Директивы объявления данных:

1)(S)BYTE – 8-разрядное целое число

2)(S)WORD – 16-разрядное целое число

3)(S)DWORD – 32-разрядное целое число

Если S есть, то число со знаком, если нет, то без знака

4) FWORD - 48-разрядное целое число

5) QWORD - 64-разрядное целое число

6) TBYTE - 80-разрядное целое число

7) REAL4 – 32-ух разрядное короткое вещественное число

8) REAL8 – 64-ух разрядное длинное вещественное число

9) REAL10 – 80-ти разрядное расширенное вещественное число

Также могут использоваться:

1) DB – определить байт

2) DW – определить слово

3) DD – определить двойное слово (4 байта)

4) DQ – определить 8 байт

5) DT – определить 10 байт

однако при их применении знаковые и беззнаковые, целые и вещественные типы не различаются, поэтому директивы считаются устаревшими, хотя реальный контроль типов данных в ассемблере в настоящее время не реализован.

Примеры:

```
        .CONST
MsgExit DB    13,10,"Press Enter to Exit",0AH,0DH

        .DATA
Zapros1 DB    "Vvidite 30 znachenii: ",13,10,0
Zapros2 DB    "Ishodnaia matrica: ",13,10,0
Result  DB    "Result: ",13,10,0
probel  DB    " ",0
nextstr DB    13,10,0
N       DWORD 6
M       WORD  5
I       WORD  1
J       WORD  1

        .DATA?
inbuf   DB      49 DUP (?)
Buffer  DB      49 DUP (?)
B       SDWORD  30 Dup (?)
D       SDWORD  15 Dup (?)
ResStr  DB      49 dup (?)

        .CODE
Start:
```

9. Процедуры. Директивы описания процедур. Пример.

Процедура в языке ассемблера – это относительно самостоятельный фрагмент программы, к которому возможно обращение из разных ее частей. На языках высокого уровня такие фрагменты оформляют соответствующим образом и называют подпрограммами: функциями или процедурами.

В отличие от языков высокого уровня, ассемблер не требует специального оформления процедур. В любое место программы можно передать управление командой вызова процедуры, и оно вернется к вызвавшей процедуре, как только встретится команда возврата управления.

Процедура выглядит так: <имя процедуры> PROC [<Тип вызова>] [<Язык>]

<тело процедуры>

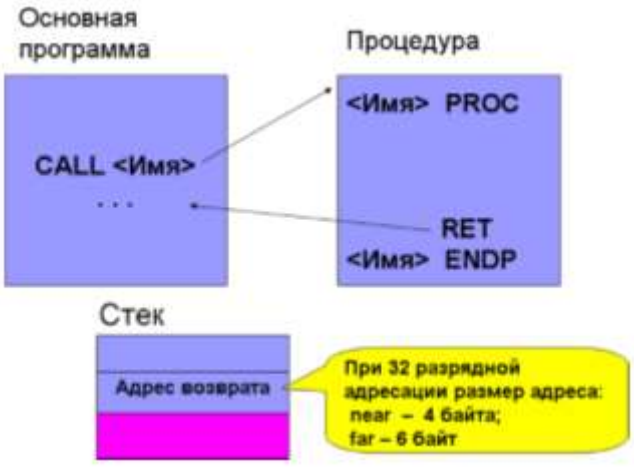
<имя процедуры> ENDP

[<>] – не обязательно, <> – обязательно

Тип вызова зависит от того, происходит ли вызов подпрограммы, находящейся в том же сегменте памяти или в другом. Вызов подпрограммы из того же сегмента называется ближним near, вызов подпрограммы из другого сегмента – дальним far

Указание языка нужно для тех случаев, когда процедура, написанная на ассемблере, вызывается из программы, написанной на другом языке.

Связь процедур ассемблера по управлению осуществляется с помощью специальных машинных команд: команды вызова процедуры CALL <имя или адрес процедуры> и команды возврата управления RET <число>, где число – удаления n байт из стека



Пример:

```
…
.CODE
Start:
XOR EAX,EAX
Call Proc1
Call Proc1
add EAX,1
jmp cont
Proc1 proc
      add EAX,2
         ret
Proc1 endp
cont:
Invoke ExitProcess,0
End    Start
```

10. Передача параметров в процедуру через глобальные переменные при совместной трансляции модулей. Пример.

Совместная трансляция основной программы и процедуры. При совместной трансляции вся программа представляет собой один исходный модуль, который транслируется за один вызов транслятора. В этом случае ассемблер формирует единое адресное пространство программы. Соответственно все имена данных, размещенных в сегменте данных, видимы и в программе, и в процедуре, расположенных в сегменте кода.

Основная программа вызывает процедуру, которая помещает значение A в B. При этом процедура для доступа к данным используют их символические имена. При необходимости так же будет обращаться к этим данным и основная программа.

Пример:

```
…
 .Data
A DWORD 1
.Data?
B DWORD ?
.CODE
Start:
Call Smesh
Invoke ExitProcess,0
Smesh proc
        XOR EAX,EAX
        Mov EAX, A
        Mov B, EAX
```

```
        add B, EAX
        Mov EAX, B
        Ret
Smesh endp
Invoke ExitProcess,0
End Start
```

11. Передача параметров в процедуру через глобальные переменные при раздельной трансляции модулей. Пример.

Раздельная трансляция процедур. При раздельной трансляции процедуры программы помещают в разные файлы, транслируют отдельно и объединяют в единую программу на этапе компоновки. Каждый файл в этом случае – отдельный модуль со своим адресным пространством. Поэтому необходимо указать компоновщику данные, по которым будет происходить «связывание» модулей. Такими данными являются:

• внутренние имена модуля, к которым будет происходить обращение из других модулей,

• внешние имена, которые определены в других модулях, но к которым есть обращение из данного модуля.

Для этого предусмотрены специальные директивы. Директива PUBLIC описывает внутренние имена, к которым возможно обращение извне:

PUBLIC [<Язык>] <Имя> [, <Язык>] <Имя>...

где <Язык> – параметр, определяющий конвенцию о связи, т.е. особенности формирования внутренних имен глобальных переменных и процедур

<Имя> – символическое имя, которое должно быть доступно в других модулях.

Директива EXTERN описывает внешние имена – имена, определенные в других исходных модулях, к которым есть обращение из данного модуля:

EXTERN [<Язык>] <Имя> [(<Псевдоним>)]: <Тип>

<Имя> – символическое имя, используемое в процедуре, но не описанное в ней;

<Тип> – определяется для различных типов имен следующим способом:

идентификатор (имя) данных: BYTE, WORD, DWORD;

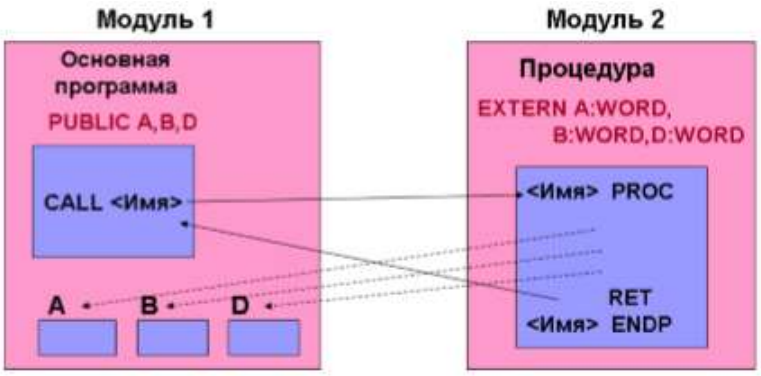
идентификатор (имя) процедуры, метка: NEAR, FAR

При этом, если в одной процедуре имя описано как EXTERN, то в другой оно должно быть описано как PUBLIC.

Универсальная директива EXTERNDEF описывает любое имя, которое описано в одном модуле, а используется в других:

EXTERNDEF [<Язык>] <Имя>:<Тип>[[<Язык>] <Имя>:<Тип>]...

В зависимости от обстоятельств может служить как Public или EXTERN.



Пример:

Основная программа:

…

```
.Data
A DWORD 1
.Data?
B DWORD ?
PUBLIC A,B
EXTERN Smesh:near
.CODE
Start:
Call Smesh
Invoke StdOut,ADDR MsgExit
Invoke StdIn,ADDR inbuf,LengthOf inbuf
End start
```

Модуль с процедурой:

```
.586
.MODEL  flat
.CODE
EXTERN A:DWORD,B:DWORD
Invoke ExitProcess,0
Smesh proc c
        push   EAX
        Mov EAX, A
        Mov B, EAX
        pop    EAX
        Ret
Smesh endp
End
```

12. Передача параметров в процедуру через таблицу. Пример.

При использовании данного способа в памяти вызывающей программы создается специальная таблица адресов параметров. В таблицу перед вызовом процедуры записывают адреса передаваемых данных. Затем адрес самой таблицы заносится в один из регистров (например, EBX) и управление передается вызываемой процедуре. Вызываемая процедура сохраняет в стеке содержимое всех регистров, которые собирается использовать, после чего выбирает адреса переданных данных из таблицы, выполняет требуемые действия и заносит результат по адресу, переданному в той же таблице.

Пример:

Основная программа:

```
…
.DATA
A     SWORD  1
.DATA?
inbuf    DB     100 DUP (?)
B     SWORD  ?
tabl     DWORD  1 dup(?)    ; таблица адресов параметров
EXTERN proc1:near
.CODE
Start: ; формирование таблицы адресов параметров
mov    tabl,offset A
mov    tabl+4,offset B
mov    EBX,offset tabl
call   proc1
Invoke ExitProcess,0
End    Start
```

Адрес A
Адрес B

Текст процедуры находится в другом модуле:

```
.586
.MODEL  flat, stdcall
```

```
OPTION CASEMAP:NONE
.CODE
proc1 proc   c
        push   ECX     ; сохранение регистров
        push   EDI
        push   ESI
        mov    ESI,[EBX]  ;загрузка адреса A
        mov    EDI,[EBX+4] ; загрузка адреса B
        xor    EAX,EAX
        mov  EAX, [ESI]
        mov    [EDI],EAX  ;загрузка результата по сохраненному адресу
        pop    ESI   ; восстановление регистров
        pop    EDI
        pop    ECX
        pop    AX
        ret
proc1 endp
END
```

13. Передача параметров в процедуру через стек. Пример.

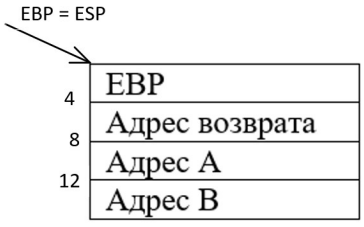
Наиболее распространенным способом передачи данных в практике программирования процессоров рассматриваемого типа является передача параметров в стеке. Именно этот способ принят в качестве базового и его большей частью используют языки высокого уровня.

В стек могут помещаться как копии значений параметров, так и их адреса. В первом случае мы говорим, что параметр передается «по значению», во втором – «по ссылке».

Копии значений параметров или их адреса помещают в стек командой PUSH, после чего управление передается вызываемой процедуре. Доступ к параметрам, хранящимся в стеке, из вызываемой процедуры осуществляют с использованием регистра EBP. В этот регистр помещают адрес вершины стека в момент начала работы процедуры, копируя его из регистра указателя стека ESP, а затем EBP используют как базовый регистр при адресации параметров.

Пример:

Основная программа:




```
…
.DATA
A     SWORD  1
.DATA?
inbuf    DB     100 DUP (?)
B     SWORD  ?
EXTERN proc1:near
.CODE
Start: ; формирование таблицы адресов параметров
push    offset A
push    offset B
call   proc1
Invoke ExitProcess,0
End    Start
```

Текст процедуры находится в другом модуле:

```
.586
.MODEL  flat, stdcall
OPTION CASEMAP:NONE
.CODE
proc1 proc   c
        push   EBP
        mov   EBP,ESP
        push   ECX         ; сохранение регистров
        push   EDI
```

```
        push   ESI
        mov    ESI,[EBP+8]  ;загрузка адреса A
        mov    EDI,[EBX+12] ; загрузка адреса B
        xor    EAX,EAX
        mov  EAX, [ESI]
        mov    [EDI],EAX  ;загрузка результата по сохраненному адресу
        pop    ESI   ; восстановление регистров
        pop    EDI
        pop    ECX
        pop    EAX
        ret
proc1 endp
END
```

14. Процедуры ввода-вывода консольного режима. Пример.

Библиотека MASM32.lib содержит специальные функции ввода, вывода и преобразования в консольном режиме, далее показаны некоторые из них:

1. Процедура ввода в консольном режиме - StdIn
   Invoke StdIn,ADDR Buffer,LengthOf Buffer, где Buffer – строковая переменная, а LengthOf – возвращает длину строки и ADDR - возвращает указатель на переменную
2. Процедура удаления символов конца строки при вводе - StripLF
   Invoke StripLF,ADDR Buffer, где Buffer – строковая переменная
3. Процедура вывода завершающейся нулем строки в окно консоли - StdOut
   Invoke StdOut,ADDR Result, где Result – строковая переменная
4. Функция позиционирования курсора - locate
   locate PROC x:DWORD, y:DWORD - местоположение курсора будет (0,0) – левый верхний угол
5. Процедура очистки окна консоли - ClearScreen
   ClearScreen PROC

```
1.          .586
2.          .MODEL  flat, stdcall
3.          OPTION CASEMAP:NONE
4.
5.  Include kernel32.inc
6.  Include masm32.inc
7.
8.  IncludeLib kernel32.lib
9.  IncludeLib masm32.lib
10.
11.         .CONST
12. MsgExit  DB    13,10,"Press Enter to Exit",0AH,0DH,0
13.
14.         .DATA
15. Zapros   DB    13,10,'Input A',13,10,0          ;Сообщение при запросе ввода
    переменной A
16. Result   DB    10,13,'Result='                  ;Сообщение при выводе
    результата сообщения
17. ResStr   DB    16 DUP (' '),0
18.         .DATA?
19. inbuf    DB    100 DUP (?)
20.         .CODE
21. Start:   Invoke StdOut,ADDR Zapros                ;Процедура вывода
    завершающейся нулем строки в окно консоли
22.         Invoke StdIn,ADDR ResStr,LengthOf ResStr ;Процедура ввода переменной
23.         Invoke StripLF,ADDR ResStr               ;Процедура замены символов
    конца строки нулем (буфер ввода)
24.         Invoke StdOut,ADDR Result
25.         XOR    EAX,EAX
26.         Invoke StdOut,ADDR MsgExit
27.         Invoke StdIn,ADDR inbuf,LengthOf inbuf
28.
```

```
29.         Invoke ExitProcess,0
30.         End    Start
```

15. Преобразования ввода-вывода. Пример.

Библиотека MASM32.lib содержит специальные функции ввода, вывода и преобразования в консольном режиме, далее показаны некоторые из них:

1. Функция преобразования завершающейся нулем строки в число - atol
   Invoke atol,ADDR Buffer, где Buffer – строковая переменная
2. Функция преобразования строки (зав. нулем) в беззнаковое число - ustr2dw
   ustr2dw proc pszString:DWORD *; результат – в EAX*
3. Функция преобразования строки в число - atodw
   Invoke atodw,ADDR Buffer, где Buffer – строка
4. Процедура преобразования числа в строку (16 байт) - ltoa
   Invoke ltoa, Buffer, ADDR ResStr, где Buffer – число, а ResStr – строковая переменная
5. Процедура преобразования числа в строку - dwtoa
   Invoke dwtoa, Buffer, ADDR ResStr, где Buffer – число, а ResStr – строковая переменная
6. Процедура преобразования беззнакового числа в строку - udw2str
   udw2str proc dwNumber:DWORD, pszString:DWORD

```
1.          .586
2.          .MODEL  flat, stdcall
3.          OPTION CASEMAP:NONE
4.
5.      Include kernel32.inc
6.      Include masm32.inc
7.
8.      IncludeLib kernel32.lib
9.      IncludeLib masm32.lib
10.
11.         .CONST
12.     MsgExit  DB    13,10,"Press Enter to Exit",0AH,0DH,0
13.
14.         .DATA
15.     Zapros   DB    13,10,'Input A',13,10,0        ;Сообщение при запросе ввода переменной A
16.     Result   DB    10,13,'Result='                ;Сообщение при выводе результата сообщения
17.     ResStr   DB    16 DUP (' '),0
18.         .DATA?
19.     inbuf    DB    100 DUP (?)
20.         .CODE
21.     Start:  Invoke StdOut,ADDR Zapros             ;Процедура вывода завершающейся нулем строки в окно консоли
22.         Invoke StdIn,ADDR ResStr,LengthOf ResStr ;Процедура ввода переменной
23.         Invoke StripLF,ADDR ResStr            ;Процедура замены символов конца строки нулем (буфер ввода)
24.         Invoke StdOut,ADDR Result
25.         XOR    EAX,EAX
26.         Invoke StdOut,ADDR MsgExit
27.         Invoke StdIn,ADDR inbuf,LengthOf inbuf
28.
29.         Invoke ExitProcess,0
30.         End    Start
```

16. Организация многомодульных программ на Masm32. Директивы. Пример.

Тоже самое что и в 11-ом?

17. Проблемы связи разноязыковых модулей. Пример.

1) Осуществление совместной компоновки модулей
2) Организация передачи и возврата управления
3) Передача параметров

А) с использованием глобальных переменных – прямой доступ к данным вызывающей программы

Б) с использованием стека (по значению и по ссылке) – косвенный доступ к данным вызывающей программы,

4) Обеспечение возврата результата функции
5) Обеспечение корректного использования регистров процессора

18. Связь Delphi Pascal – Masm32. Конвенция pascal. Пример.

В модуле на Delphi Pascal процедуры и функции, реализованные на ассемблере, должны быть объявлены и описаны как внешние external с указанием конвенции связи, например:

Модуль ассемблера предварительно ассемблируется и подключается с использованием директивы обычно – в секции реализации модуля Delphi Pascal: {$l <Имя объектного модуля>}

Параметры передаются через стек:

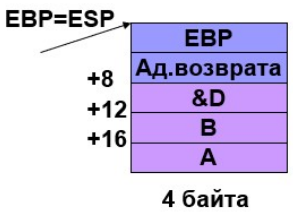
1) по значению – в стеке копия значения,
2) по ссылке – в стеке указатель на параметр;

Результаты функций возвращаются через регистры:

1) байт, слово – в **AX**,
2) двойное слово, указатель – в **EAX**,
3) строка – через указатель, помещенный в стек после параметров.

Пример:

```
        .586
        .model  flat
        .code
        public  ADD1
ADD1    proc
        push    EBP
        mov     EBP,ESP
        mov     EAX,[EBP+16]
        add     EAX,[EBP+12]
        mov     EDX,[EBP+8]
        mov     [EDX],EAX
        pop     EBP
        ret     12
ADD1    endp
        end
```



19. Связь Delphi Pascal – Masm32. Конвенция cdecl. Пример.

В модуле на Delphi Pascal процедуры и функции, реализованные на ассемблере, должны быть объявлены и описаны как внешние external с указанием конвенции связи, например:

Модуль ассемблера предварительно ассемблируется и подключается с использованием директивы обычно – в секции реализации модуля Delphi Pascal: {$l <Имя объектного модуля>}

Параметры передаются через стек:

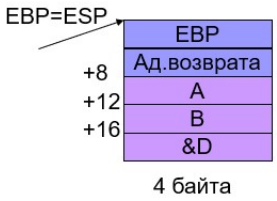
1) по значению – в стеке копия значения,
2) по ссылке – в стеке указатель на параметр;

Результаты функций возвращаются через регистры:

1) байт, слово – в **AX**,
2) двойное слово, указатель – в **EAX**,
3) строка – через указатель, помещенный в стек после параметров.

Пример:

```
        .586
        .model  flat
        .code
        public  ADD1
ADD1    proc
        push    EBP
        mov     EBP,ESP
        mov     EAX,[EBP+8]
        add     EAX,[EBP+12]
        mov     EDX,[EBP+16]
        mov     [EDX],EAX
        pop     EBP
        ret
ADD1    endp
        end
```



20. Связь Delphi Pascal – Masm32. Конвенция register. Пример.

В модуле на Delphi Pascal процедуры и функции, реализованные на ассемблере, должны быть объявлены и описаны как внешние external с указанием конвенции связи, например:

Модуль ассемблера предварительно ассемблируется и подключается с использованием директивы обычно – в секции реализации модуля Delphi Pascal: {$l <Имя объектного модуля>}

Параметры передаются через стек:

1) по значению – в стеке копия значения,
2) по ссылке – в стеке указатель на параметр;

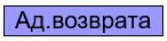
Результаты функций возвращаются через регистры:

1) байт, слово – в **AX**,
2) двойное слово, указатель – в **EAX**,
3) строка – через указатель, помещенный в стек после параметров.

Пример:

```
        .586
        .model  flat
        .code
        public  ADD1
ADD1    proc
        add     EDX,EAX
        mov     [ECX],EDX
        ret
ADD1    endp
        end
```

1-й параметр A в EAX;
2-й параметр B в EDX;
3-й параметр &C в ECX
остальные параметры
в обратном порядке в
стеке



21. Связь Delphi Pascal – Masm32. Создание локальных переменных. Пример.

Паскаль не позволяет создавать в подпрограммах глобальные переменные, поэтому в подпрограммах работают с локальными данными, размещаемыми в стеке. Для работы с локальными данными используют структуры.

Структура представляет собой шаблон с описаниями форматов данных, который можно накладывать на различные участки памяти, чтобы затем обращаться к полям этих участков памяти с помощью имен, определенных в описании структуры.

Формат описания структуры:

<Имя структуры> STRUCT
    <Описание полей>
<Имя структуры> ENDS

где <Описание полей> – объявление переменных

Пример:

```
.586
```

```
.MODEL flat
Per STRUCT
        A DWORD 23 DUP (?) ; локальная переменная
Per ENDS
.CODE
…
```

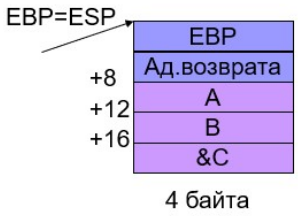
22. Связь C++ – Masm32. Конвенция cdecl. Пример.

В модуле на Visual C++ подключаемые процедуры и функции должны быть объявлены как внешние extern с указанием конвенции связи

Файл с расширением .obj надо подключить к проекту на C++

```
extern "C" void _cdecl add1(int a,int b,int *c);
```

```
        .586
        .model  flat
        .code
        public  _add1
_add1   proc
        push    EBP
        mov     EBP,ESP
        mov     EAX,[EBP+8]
        add     EAX,[EBP+12]
        mov     EDX,[EBP+16]
        mov     [EDX],EAX
        pop     EBP
        ret
_add1   endp
        end
```



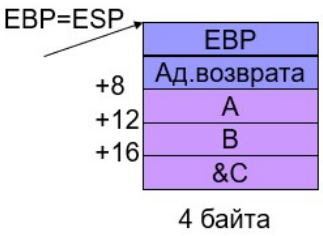
23. Связь C++ – Masm32. Конвенция stdcall. Пример.

В модуле на Visual C++ подключаемые процедуры и функции должны быть объявлены как внешние extern с указанием конвенции связи

Файл с расширением .obj надо подключить к проекту на C++

```
extern "C" void __stdcall ADD1(int a,int b,int *c);
```

```
        .386
        .model  flat
        .code
        public ?ADD1@@YGXHHPAH@Z
?ADD1@@YGXHHPAH@Z proc
        push EBP
        mov  EBP,ESP
        mov  ECX,[EBP+8]
        add  ECX,[EBP+12]
        mov  EAX,[EBP+16]
        mov  [EAX],ECX
        pop  EBP
        ret  12
?ADD1@@YGXHHPAH@Z endp
        end
```



24. Связь C++ – Masm32. Конвенция fastcall. Пример.

В модуле на Visual C++ подключаемые процедуры и функции должны быть объявлены как внешние extern с указанием конвенции связи

Файл с расширением .obj надо подключить к проекту на C++

```
extern "C" void __fastcall add1(int a,int b,int *c);

        .386
        .model  flat
        .code
        public  @ADD1@12
@ADD1@12 proc
        push    EBP
        mov     EBP,ESP
        add     ECX,EDX
        mov     EDX,[EBP+8]
        mov     [EDX],ECX
        pop     EBP
        ret     4       ; стек освобождает процедура
@ADD1@12    endp
        end
```

82

25. Связь C++ – Masm32. Создание локальных переменных. Пример.

Тоже самое что и в 21-ом?

Варианты 2-го вопроса билета:

1. Проблема построения компилирующих программ. Метод Рутисхаузера. Пример.

Первыми компилирующими программами были программы, которые переводили формальную запись выражений в машинный язык. В основе этого метода лежит представление выражение в виде последовательности “троек”, где каждая тройка включает два операнда и операцию. Пример: A + B => T_1

Основной проблемой при этом является необходимость учета приоритетов операций. Например: d = a + b * c => T_1 = b * c => d = a + T_1 => T_2 = a + T_1 => d = T_2

Решение этой проблемы предложил Рутисхаузер – чтобы выражение было записано в полной скобочной записи, а порядок выполнения операций указывается скобками. Для наглядности снизу подписываются номера выполнения. Справа выполняются действия, имеющие наибольшие число.

Пример: (((a + b) * c) / d) + f

1) (((a + b) * c) / d) + f => T_1 = a + b
   012 3 4 3 43 232121 0 1 0

2) ((T_1 * c) / d) + f => T_2 = T_1 * c
   012 3 2 321 21 0 1 0

3) (T_2 / d) + f => T_3 = T_2 / d
   01 2 1 21 0 1 0

4) T_3 + f => T_4 = T_3 + f
   0 1 0 1 0

Недостаток – требование полой скобочной структуры. Как правило, для этого используется специальная программа, которая вставляет скобки.

2. Типы компилирующих программ и их особенности.

1) Интерпретатор – принимает и выполняет программу, ПО используемое для запуска высокоуровневых программ и кодов. Анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора

2) Макрогенератор – обрабатывает исходную программу как текст, выполняя

замены символов на подстроки, при помощи задаваемых ей правил замены последовательностей символов, называемых правилами макроподстановки. Для компиляторов это процессор

3) Компилятор – программа, которая транслирует с языка высокого уровня на язык низкого уровня машинный или ассемблера

4) Транслятор – переводит программу, написанную на одном языке, эквивалентную ей, только на другом языке, чаще всего он используется при переводе программы на машинный код

5) Ассемблер – транслятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке

3. Структура компилирующей программы. Основные фазы процесса. Пример.

1) Лексический анализ - процесс аналитического разбора входной последовательности символов на распознанные группы - лексемы, из которых создается таблица токенов содержащая столбцы “лексема”, “тип”, “значение” и “ссылкой”

Пример: Лексема – if, Тип – Служебное слово, Значение – код “if”, Ссылки нет

Лексема – 5, Тип – литерал, Значение нет, Ссылки – адрес в таблице

Литералов (Таблица литералов - таблица, содержащая литералы (константы), их тип и другую информацию. Литералу выделяется ячейка памяти, в которую записывается константа. Далее все появления этого литерала заменяются обращениями по адресу этой ячейки)

2) Синтаксический анализ - это процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка, получаемых в результате лексического анализа, с набором правил, определяющие конструкции языка (форма)

Пример: “if (f == 5) then a = 3;” – верно, “if (f == 5) ; else then a = 3;” – ошибка

3) Семантический анализ - правила, определяющие соответствие между элементами (содержание).

Пример: “int a = 0; real b = 2,3; a = b;” – ошибка, “int a = 0; real b = 2,3; b = a;” - верно

4) Распределение памяти – выделение памяти (статической или временной) под переменные, используемые в программе. Оно может быть

А) статическое – в процессе компиляции

Б) автоматическое – во время выполнения программы при вызове подпрограмм

В) управляемое – при использовании динамических переменных

Г) базированное – по запросу программиста (getmem, freemem)

5) Генерация и оптимизация объектного кода

Генерация кодов – процесс замены распознанных конструкций на соответствующий заранее заготовленный фрагмент.

Пример «заготовки»:

```
Mov   AX, <Op1>
Add   AX, <Op2>
Mov   <Result>, AX
```

Оптимизация объектного кода – уменьшение либо время его выполнения, либо места памяти, либо и то, и другое путем преобразования кода.

4. Синтаксис и семантика языка программирования. Средства описания синтаксиса. Примеры.

Синтаксис – это набор правил, определяющие допустимые конструкции языка

Синтаксический анализ - это процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка, получаемых в результате лексического анализа, с его формальной грамматикой, то есть с набором правил, определяющие конструкции языка (форма)

Семантика – это совокупность правил, определяющих логическое соответствие\правильность между элементами, то есть содержание языка

Семантический анализ - правила, определяющие соответствие между элементами (содержание). Проверка типов является важной частью семантического анализа, где компилятор проверяет, что у каждого оператора есть совпадающие операнды.

Средства описания синтаксиса: Форма Бэкуса-Наура или синтаксические диаграммы

<Имя> - нетерминальный символ – конструкция
Имя – терминальный символ – символ алфавита
::= - “можно заменить на”
| - “или”

Пример:

Грамматика в форме Бекуса-Наура:

<Оператор> ::= <Оператор if> | <Присвоение>

<Оператор if> ::= if <Условие><Оператор> | if <Условие><Оператор> else <Оператор>

<Присвоение> ::= <Идентификатор> = <Выражение> | <Идентификатор> = <Идентификатор>

<Условие> ::= <Идентификатор>

<Идентификатор> ::= <Буква> | <Идентификатор><Буква> | <Идентификатор>< Цифра >

<Выражение> ::= <Число>

<Число> ::= <Цифра> | <Знак><Цифра> | <Знак><Цифра><Число> | <Цифра><Число>

<Цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

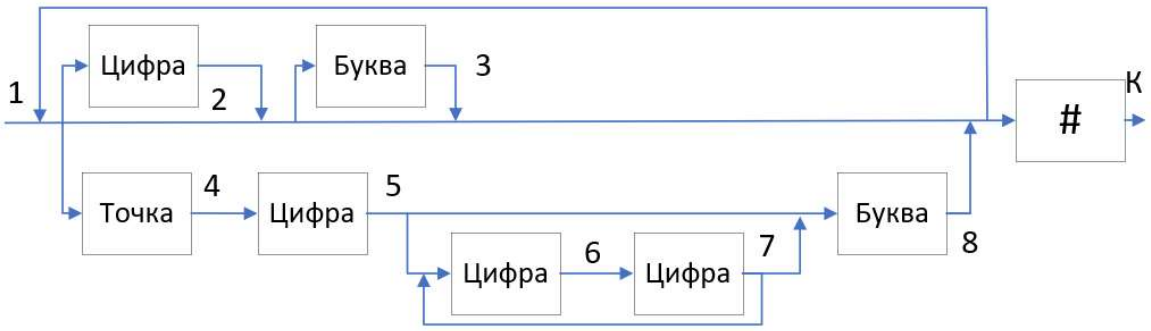
<Буква> ::= a | … | z | A | … | Z

<Знак> ::= - | + | * | /

Даны непустые строки, включающие цифры, буквы и точки. Признак завершения строки - #. Если в строке есть точки, то после каждой точки должно идти нечетное количество цифр и хотя бы одна буква. Например, допустимы строки:

а) .045nfgfj57656gjg.4b#
б) fh46546fh.69789fhe78w#
в) 5b57k67#

Построить синтаксическую диаграмму и таблицу конечного автомата “принимающего” только строки заданного вида.



	Буква	Цифра	Точка	#	Другие
1	3	2	4	error	error
2	3	2	4	Конец	error
3	3	2	4	Конец	error
4	error	5	error	error	error
5	8	6	error	error	error
6	error	7	error	error	error
7	8	6	error	error	error
8	3	2	4	error	error

5. Синтаксические диаграммы и особенности их построения. Примеры.

Любой язык программирования подчиняется правилам, описанным его грамматикой. Для описания синтаксиса языков с бесконечным количеством различных предложений используют рекурсию.

Для записи обычно используют более компактные и наглядные формы: форму Бэкуса-Наура

<Имя> - нетерминальный символ – конструкция
Имя – терминальный символ – символ алфавита
::= - “можно заменить на”
| - “или”

<целое> ::= <знак><целое без знака>|<целое без знака>,

<целое без знака> ::= <цифра><целое без знака>|<цифра>,

<цифра > ::= 0|1|2|3|4|5|6|7|8|9,

<знак> ::= +| - .

И синтаксические диаграммы – их (автоматов) графическое представление:

Он показывает последовательность действий автомата.

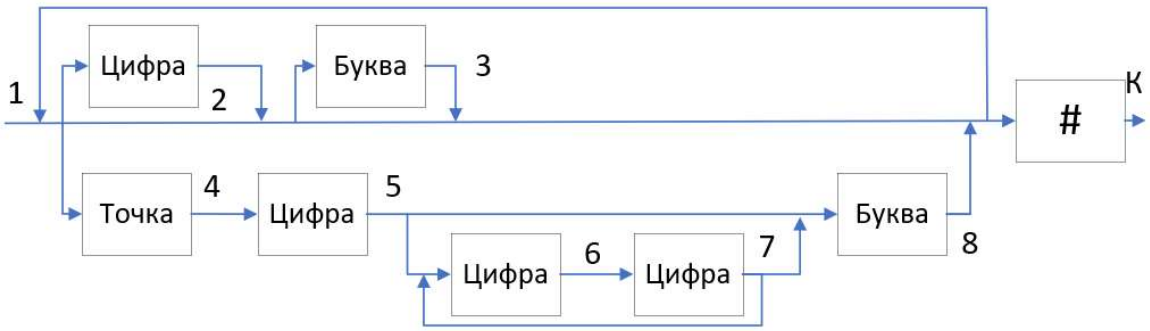
Правила изображения:

1) в кружок обозначаются терминальные символы
2) в прямоугольник нетерминальные символы
3) переход между действиями показывается дугами
4) если есть цикл, то дуга будет в обратную сторону, относительно основного пути графа
5) т.к. к ним еще рисуется таблица состояний, то каждое состояние на графе нумеруется

Даны непустые строки, включающие цифры, буквы и точки. Признак завершения строки - #. Если в строке есть точки, то после каждой точки должно идти нечетное количество цифр и хотя бы одна буква. Например, допустимы строки:

а) .045nfgfj57656gjg.4b#
б) fh46546fh.69789fhe78w#
в) 5b57k67#

Построить синтаксическую диаграмму и таблицу конечного автомата "принимающего" только строки заданного вида.



	Буква	Цифра	Точка	#	Другие
1	3	2	4	error	error
2	3	2	4	Конец	error
3	3	2	4	Конец	error
4	error	5	error	error	error
5	8	6	error	error	error
6	error	7	error	error	error
7	8	6	error	error	error
8	3	2	4	error	error

6. Формальный язык и формальная грамматика языка. Пример.

Алфавит – непустое конечное мн-во символов, используемых для записи предложений языка. Пример: A = {0,1,2,3,4,5,6,7,8,9,+,-}

Строка – любая последовательность символов алфавита

A* = мн-во строк, включая пустую, составленных из A

A^+ = мн-во строк, не включая пустую, составленных из A

Формальным языком L в алфавите A называют произвольное подмн-во A*

Формальная грамматика - G = (V_T, V_N, P, S), где V_T – алфавит языка или мн-во терминальных символов; V_N – мн-во нетерминальных символов – вспомогательный алфавит, символы которого допустимы в языке; P – мн-во правил, где каждое правило – это (α, β), где α ∈ V^+, β ∈ V*; S ∈ V_N – начальный символ.

Где V+ - это мн-во строк за исключением пустой, составленной из V; а V* - это мн-во строк включая пустую, составленной из V

Пример:

Грамматика записи десятичных чисел G_0

V_T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -};
V_N = {<целое>, <целое без знака>, <цифра>, <знак>};
P = {<целое> → <знак><целое без знака>,
<целое> → <целое без знака>,
<целое без знака> → <цифра><целое без знака>,
<целое без знака> → <цифра>,
<цифра> → 0,
<цифра> → 1,
<цифра> → 2,
<цифра> → 3,
<цифра> → 4,
<цифра> → 5,
<цифра> → 6,
<цифра> → 7,
<цифра> → 8,
<цифра> → 9,
<знак> → +,
<знак> → - };
S = <целое>.

Правосторонняя рекурсия

7. Классификация грамматик Хомского. Примеры грамматик 2-го и 3-го типов.

1) Тип 0 – грамматики фразовой структуры: α → β, где α ∈V+, β ∈ V* – допустимо наличие любых правил вывода, что свойственно грамматикам естественных языков;

2) Тип 1 – контекстно-зависимые грамматики: α → β, где α ∈V^+, β ∈ V*, | α | ≤ | β | – в этих грамматиках для правил вида α X β → α x β возможность подстановки строки x вместо символа X определяется присутствием подстрок α и β, т. е. контекста, что также свойственно грамматикам естественных языков;

3) Тип 2 – контекстно-свободные грамматики: A → β, где A∈V_N, β∈ V* – поскольку в левой части правила стоит нетерминал, подстановки не зависят от контекста;

4) Тип 3 – регулярные грамматики: A → α, A → αB, где A, B∈V_N, α ∈ V_T.

Где V+ - это мн-во строк за исключением пустой, составленной из V; а V* - это мн-во строк включая пустую, составленной из V

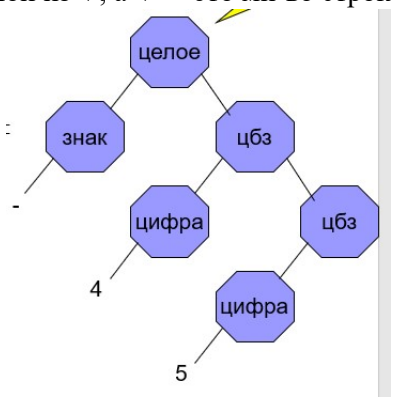
8. Грамматический разбор. Дерево грамматического разбора. Пример.

Грамматический разбор - процесс сопоставления линейной последовательности лексем (слов, токенов) языка с его формальной грамматикой.

Формальная грамматика - G = (V_T, V_N, P, S), где V_T – алфавит языка или мн-во терминальных символов; V_N – мн-во нетерминальных символов – вспомогательный алфавит, символы которого допустимы в языке; P – мн-во правил, где каждое правило – это (α, β), где α ∈ V^+, β ∈ V*; S ∈ V_N – начальный символ

Где V+ - это мн-во строк за исключением пустой, составленной из V; а V* - это мн-во строк включая пустую, составленной из V

Выводом называется последовательность подстановок.



Пример. Вывод строки «-45»:

<целое> => 1
=> 1<знак> <целое без знака> => 2
=> 2 <знак><цифра> <целое без знака> => 3
=> 3 <знак><цифра><цифра> => 4
=> 4 - <цифра><цифра> => 5
=> 5 - 4<цифра> => 6
=> 6- 45

9. Левосторонний нисходящий грамматический разбор для грамматик 3-го типа. Пример.

Идея метода:

При разборе строка просматривается слева направо и ищется часть, совпадающая с правой частью правила, – основа.

Основа – последовательность символов, сворачиваемая на следующем шаге разбора.

Найденная основа заменяется левой частью соответствующего правила и т.д.

Последовательность выбора основ определена алгоритмом.

Неверный выбор основы приводит к необходимости возврата.

При нисходящем разборе правила просматриваются сверху вниз. От понятия (понятия более высокого порядка, аксиомы) к символам или лексемам языка.

3-ий тип грамматики – это когда нетерминалы стоят как слева, так и справа, а терминалы могут стоять только справа

Пример:

Распознано	Строка	Правила	Правило
	–45◄	<Целое> ◄	?
	–45◄	<Знак><ЦБЗ>◄	1а – ?
	–45◄	+<ЦБЗ>◄ –<ЦБЗ>	4а – нет 4б – да
–	45◄	<ЦБЗ>◄	
–	45◄	<Цифра><ЦБЗ>◄	2а – ?
–	45◄	0..3<ЦБЗ>◄ 4<ЦБЗ>◄	3а..3г – нет 3е – да
–4	5◄	<ЦБЗ>◄	
–4	5◄	<Цифра><ЦБЗ>◄	2а – ?
–4	5◄	0..4<ЦБЗ>◄ 5<ЦБЗ>◄	3а..3е – нет 3ж – да
–45	◄	<ЦБЗ>◄	
–45	◄	<Цифра><ЦБЗ>◄	2а –?
–45	◄	0..9<ЦБЗ>◄	3а..3и – нет
	◄	<ЦБЗ>◄	2б –?
	◄	<Цифра>◄	3а..3и – нет
–4	5◄	<Цифра>◄	2б – ?
	5◄	1..4<ЦБЗ>◄ 5<ЦБЗ>◄	3а..3е – нет 3ж – да
–45	◄	◄	Конец

10. Левосторонний восходящий грамматический разбор для грамматик 3-го типа. Пример.

Идея метода:

При разборе строка просматривается слева направо и ищется часть, совпадающая с правой частью правила, – основа.

Основа – последовательность символов, сворачиваемая на следующем шаге разбора.

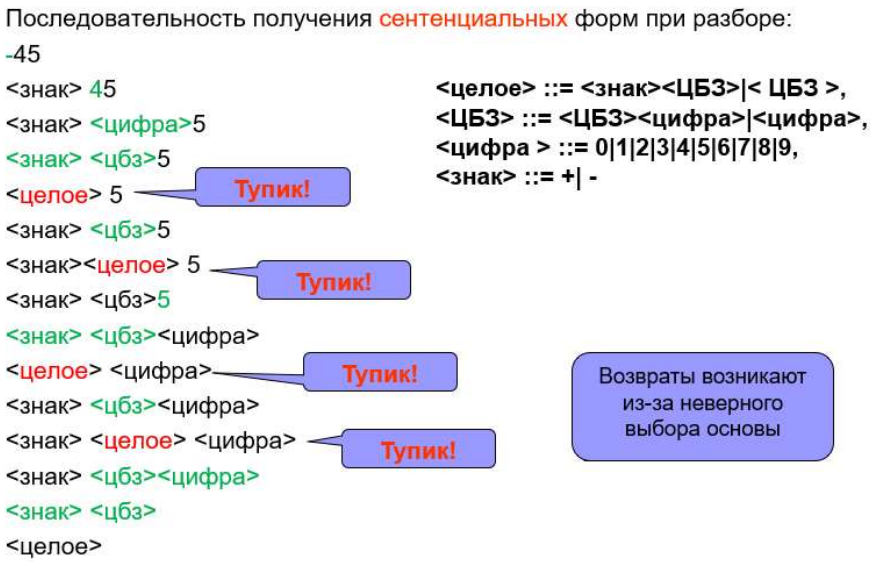
Найденная основа заменяется левой частью соответствующего правила и т.д.

Последовательность выбора основ определена алгоритмом. Неверный выбор основы приводит к необходимости возврата.

При восходящем разборе правила просматриваются сверх вниз. От понятий к символам или лексемам языка

3-ий тип грамматики – это когда нетерминалы стоят как слева, так и справа, а терминалы могут стоять только справа

Пример:



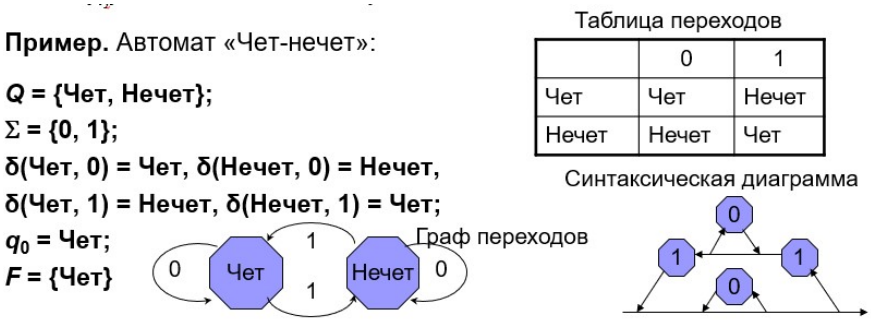
11. Конечный автомат. Пример описания и алгоритм программной реализации.

Конечный автомат — это некоторая абстрактная модель, содержащая конечное число состояний чего-либо. Используется для представления и управления потоком выполнения каких-либо команд.

Он представляется так M = (Q, Σ, δ, q_0, F), где Q – конечное мн-во состояний; Σ – конечное мн-во входных символов; δ(q_i, c_k) – функция переходов (начальное состояние); q_0 - начальное состояние; F = {q_i} – подмн-во допускающих состояний.

Автомат может быть представлен в виде “таблицы переходов”, “синтаксической диаграммы” и “графа переходов”

Пример:



12. Построение лексических анализаторов на конечных автоматах. Пример.

Конченый автомат – это множество M = (Q, ∑, δ, q_0, F),

где Q – конечное множество состояний;

∑ – конечное множество входных символов;

δ(q_i, c_k) – функция переходов (q_i – текущее состояние, c_k – очередной символ);

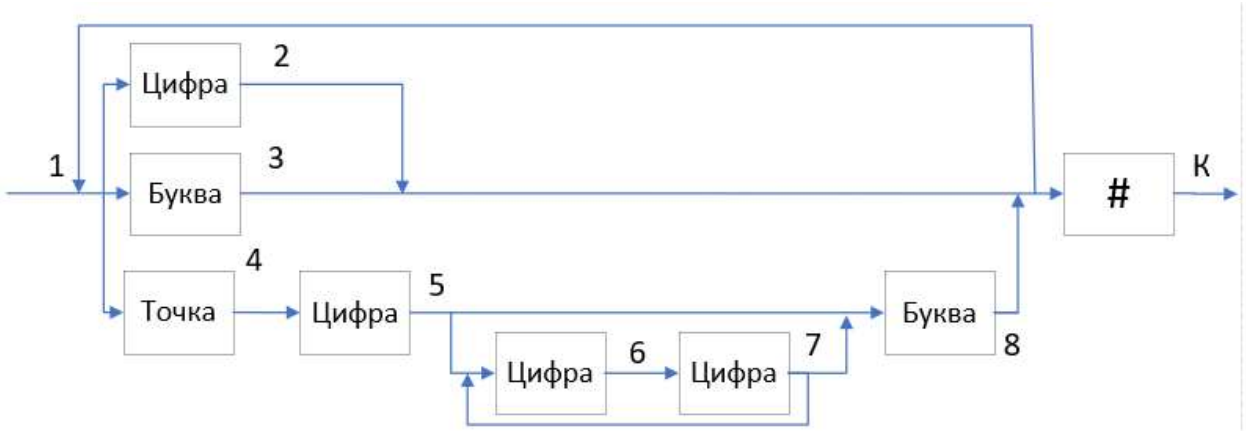
q_0 – начальное состояние;
$F = \{q_j\}$ – подмножество допускающих состояний

Пример:

Даны непустые строки, включающие цифры, буквы и точки. Признак завершения строки - #. Если в строке есть точки, то после каждой точки должно идти нечетное количество цифр и после них хотя бы одна буква. Например, допустимы строки:

а) .045nfgfj57656gjg.4b#
б) fh46546fh.69789fhe78w#
в) 5b57k67#

Построить синтаксическую диаграмму и таблицу конечного автомата "принимающего" только строки заданного вида.



	Буква	Цифра	Точка	#	Другие
1	3	2	4	error	error
2	3	2	4	Конец	error
3	3	2	4	Конец	error
4	error	5	error	error	error
5	8	6	error	error	error
6	error	7	error	error	error
7	8	6	error	error	error
8	3	2	4	Конец	error

13. Построение синтаксических анализаторов на конечных автоматах. Пример.

Конченый автомат – это множество $M = (Q, \sum, \delta, q_0, F)$,

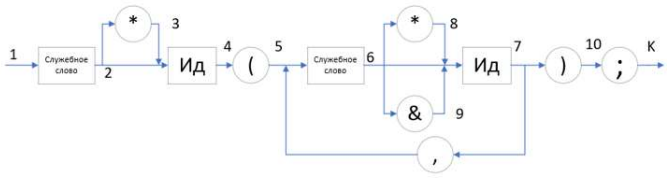
где Q – конечное множество состояний;
$\sum$ – конечное множество входных символов;
$\delta(q_i, c_k)$ – функция переходов (q_i – текущее состояние, c_k – очередной символ);
q_0 – начальное состояние;
$F = \{q_j\}$ – подмножество допускающих состояний

Пример:

Рассмотреть конструкцию "Объявление прототипа функции". Например:

```
int *acs(int a,float *j,short &y);
float xxx(char * r, int k);
```

1. Вручную выполнить лексический анализ заданных предложений языка и получить строки токенов для них.
2. Построить синтаксическую диаграмму и таблицу конечного автомата для синтаксического анализа указанных и аналогичных предложений языка.



	Сл_сл	*	Ид	(	&	)	;	,	Другие
1	2	Error	Error	Error	Error	Error	Error	Error	Error
2	Error	3	4	Error	Error	Error	Error	Error	Error
3	Error	Error	4	Error	Error	Error	Error	Error	Error
4	Error	Error	Error	5	Error	Error	Error	Error	Error
5	6	Error	Error	Error	Error	Error	Error	Error	Error
6	Error	8	7	Error	9	Error	Error	Error	Error
7	Error	Error	Error	Error	Error	10	Error	5	Error
8	Error	Error	7	Error	Error	Error	Error	Error	Error
9	Error	Error	7	Error	Error	Error	Error	Error	error
10	Error	Error	Error	Error	Error	Error	Конец	Error	error

14. Автомат с магазинной памятью.

Это конечный автомат, который использует стек для хранения состояний. В отличие обычных автоматов, этот является набором. Для чтения доступен только последний записанный в неё элемент.

$P_M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$,

где Q – конечное множество состояний автомата;

Σ – конечный входной алфавит;

Γ – конечное множество магазинных символов;

$\delta\ (q, ck, zj)$ – функция переходов;

$q_0 \in Q$ – начальное состояние автомата;

$z_0 \in \Gamma$ – символ, находящийся в магазине в начальный момент,

$F \subseteq Q$ – множество заключительных (допускающих) состояний.

15. Левосторонний нисходящий грамматический разбор для грамматик 2-го типа. Пример.

Идея метода:

Основа – последовательность символов, сворачиваемая на следующем шаге разбора.
Найденная основа заменяется левой частью соответствующего правила и т.д.
Последовательность выбора основ определена алгоритмом.
Неверный выбор основы приводит к необходимости возврата.

При нисходящем разборе правила просматриваются сверху вниз. От понятия (понятия более высокого порядка, аксиомы) к символам или лексемам языка.

Пример:

Дана грамматика, определяющая конструкцию "Логическое выражение":

<ЛВыражение> ::= <Пересечение > v < ЛВыражение >|<Пересечение>

<Пересечение> ::= <Операнд> ^ <Пересечение> |<Операнд>

<Операнд> ::= (<ЛВыражение>)|<Ид>

Используя правила грамматики, осуществите левосторонний нисходящий разбор сентенциальной формы (достаточно первых 20 шагов):

(<Ид> v <Ид>) ^ (<Ид> v <Ид>)

	№	Распознано	Текущая строка	Стек	Правило
1	1		(<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<ЛВ>◄	1а ?
2	2		(<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<П>v<ЛВ>◄	2а ?
3	3		(<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<О>Λ<П>v<ЛВ>◄	3а ?
4	4		(<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	(<ЛВ>)Λ<П>v<ЛВ>◄	Удалить символ
5	5	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<ЛВ>)Λ<П>v<ЛВ>◄	1а ?
6	6	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<П>v<ЛВ>)Λ<П>v<ЛВ>◄	2а ?
7	7	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<О>Λ<П>v<ЛВ>)Λ<П>v<ЛВ>	3а ?
8	8	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	(<ЛВ>)Λ<П>v<ЛВ>)Λ<П>v<ЛВ>◄	Тупик, возвращаемся к шагу 7
9	7	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<О>Λ<П>v<ЛВ>)Λ<П>v<ЛВ>◄	3б ?
10	8	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<Ид>Λ<П>v<ЛВ>)Λ<П>v<ЛВ>◄	Удалить символ
11	9	(<Ид>	v<Ид>)Λ(<Ид>v<Ид>)◄	Λ<П>v<ЛВ>)Λ<П>v<ЛВ>◄	Тупик, возвращаемся к шагу 6
12	6	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<П>v<ЛВ>)Λ<П>v<ЛВ>◄	2б ?
13	7	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<О>v<ЛВ>)Λ<П>v<ЛВ>◄	3а ?
14	8	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	(<ЛВ>)v<ЛВ>)Λ<П>v<ЛВ>◄	Тупик, возвращаемся к шагу 7
15	9	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<О>v<ЛВ>)Λ<П>v<ЛВ>◄	3б ?
16	10	(	<Ид>v<Ид>)Λ(<Ид>v<Ид>)◄	<ИД>v<ЛВ>)Λ<П>v<ЛВ>◄	Удалить символ
17	11	(<Ид>	v<Ид>)Λ(<Ид>v<Ид>)◄	v<ЛВ>)Λ<П>v<ЛВ>◄	Удалить символ
18	12	(<Ид>v	<Ид>)Λ(<Ид>v<Ид>)◄	<ЛВ>)Λ<П>v<ЛВ>◄	1а ?
19	13	(<Ид>v	<Ид>)Λ(<Ид>v<Ид>)◄	<П>v<ЛВ>)Λ<П>v<ЛВ>◄	2а ?
20	14	(<Ид>v	<Ид>)Λ(<Ид>v<Ид>)◄	<О>Λ<П>v<ЛВ>)Λ<П>v<ЛВ>◄	3а ?

16. Левосторонний восходящий грамматический разбор для грамматик 2-го типа. Пример.

Идея метода:

При разборе строка просматривается слева направо и ищется часть, совпадающая с правой частью правила, – основа.

Основа – последовательность символов, сворачиваемая на следующем шаге разбора.

Найденная основа заменяется левой частью соответствующего правила и т.д.

Последовательность выбора основ определена алгоритмом. Неверный выбор основы приводит к необходимости возврата.

При восходящем разборе правила просматриваются сверх вниз. От понятий к символам или лексемам языка

Пример:

Дана грамматика, определяющая конструкцию "Оператор if":

<Оператор> ::= <Оператор If>|<Выражение>|<Составной оператор>

<Оператор If> ::= if (<Условие>) <Оператор> else <Оператор>|

if (<Условие>) <Оператор>

<Составной оператор> ::= {<Операторы>}

<Операторы> ::= <Операторы><Оператор>|<Оператор>

Используя правила грамматики, осуществите левосторонний восходящий разбор сентенциальной формы (при длинной последовательности достаточно привести первые 20 строк):

if (<Условие>)

{ if (<Условие>) <Выражение> <Выражение> }

Уточните, к какому типу по Хомскому относится заданная грамматика. Почему?

1) if (<y>) {if (<y>)<в><в>}
2) if (<y>) {if (<y>)<оп-р><в>}
3) if (<y>) {if (<y>)<оп-ры><в>}
4) if (<y>) {if (<y>)<оп-ры><в>}

5) if (<y>) {if (<y>)<оп-ры><оп-р>}
6) if (<y>) {if (<y>)<оп-ры>} Тупик, возвращаемся к шагу 2
2) if (<y>) {if (<y>)<оп-р><в>}
3) if (<y>) {<оп-р if><в>}
4) if (<y>) {<оп-р if><в>}
5) if (<y>) {<оп-р><в>}
6) if (<y>) {<оп-ры><в>}
7) if (<y>) {<оп-ры><в>}
8) if (<y>) {<оп-ры><оп-р>}
9) if (<y>) {<оп-ры>}
10) if (<y>) {<с>}
11) if (<y>) {<оп-р>}
12) if (<y>) {<оп-ры>} Тупик, возвращаемся к шагу 11
13) if (<y>) {<оп-р>}
14) <оп-р if>

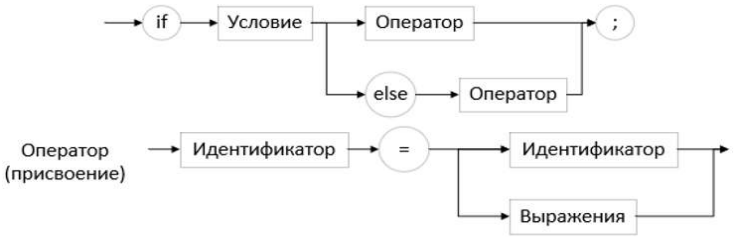
17. LL(k) грамматики. Метод рекурсивного спуска. Пример.

LL(K) грамматика – это класс КС-грамматик. Расшифровка аббревиатуры: L – левосторонний просмотр; L – левосторонний разбор; k – количество символов, просматриваемых для однозначного определения следующего правила. В этом классе отсутствует левосторонняя рекурсия.

Этот класс подразумевает нисходящий метод разбора.

Нисходящий метод: от понятия (понятия более высокого порядка, аксиомы) к символам или лексемам языка.

Метод рекурсивного спуска можно разобрать на примере синтаксической диаграммы. Если, к примеру, в схеме встречается “оператор”, который может быть либо уловным оператором, либо просто операцией присвоения. Если это операция присвоения, то это можно изобразить подробнее, то есть произойдет переход, а потом рекурсивное возращение к исходной диаграмме. Это же будет и происходить в программе, реализующей этот алгоритм.



18. LR(k) грамматики. Операторное предшествование. Стековый метод. Пример.

LR(K) грамматика – это класс КС-грамматик. Расшифровка аббревиатуры: L – левосторонний просмотр; R – правосторонний разбор; k – количество символов, просматриваемых для однозначного определения следующего правила. В этом классе отсутствует правосторонняя рекурсия.

К этому классу относят грамматики с операторным предшествованием, простым предшествованием и расширенным предшествованием. Что дает простые методы распознавания.

Автомат выполняет две функции – свертку и перенос. Свертка происходит, когда основа заканчивается и для её свертки есть соответствующие правила. Перенос происходит в процессе свертки и заключается в сохранении в стеке очередного распознаваемого символа сентенциальной формы.

Отношения предшествования:

Если два символа $\alpha, \beta \in V$ расположены рядом в сентенциальной форме, то между ними возможны следующие отношения, названные отношениями предшествования:

- α принадлежит основе, а β – нет, т. е. α – конец основы: $\alpha \cdot> \beta$;
- β принадлежит основе, а α – нет, т. е. β – начало основы: $\alpha <\cdot \beta$;
- α и β принадлежит одной основе, т. е. $\alpha =\cdot \beta$;
- α и β не могут находиться рядом в сентенциальной форме (ошибка).

Во время синтаксического разбора часто возникают возвраты из-за неверного

выбора основы, грамматика LR(k) обеспечивает же однозначный выбор основы.

Строится таблица предшествования:

╲	+	*	(	)	◄
►	<·	<·	<·	?	Выход
+	·>	<·	<·	·>	·>
*	·>	·>	<·	·>	·>
(	<·	<·	<·	=	?
)	·>	·>	?	·>	·>

где "<·" – начало основы, "·>" – конец основы, "=" – одна основа, "?" - ошибка

Стековый метод состоит из грамматики операторного предшествования, в которой существует однозначное отношение предшествования терминальных символов, которое не зависит от нетерминальных символов, находящихся между ними.

Операнды загружаются в стек операндов, когда встречается операция, берется два операнда из стека, производится операция и результат помещают в стек операндов.

Пример из РК4:

Дана грамматика, определяющая конструкцию "Оператор While":

<Оператор> ::= <Оператор W>|<Присваивание>|<Составной оператор>

<Оператор W> ::= while <Условие> do <Оператор>

<Составной оператор> ::= begin <Операторы> end

<Операторы> ::= <Операторы>;<Оператор>|<Оператор>

Используя стековый метод, разберите сентенциальную форму:

while <Условие>do

begin while < Условие> do <Присваивание>; <Присваивание> end

Уточните, к какому типу по Хомскому относится заданная грамматика. Почему?

Условие – у
Присваивание – пр
Оператор – о-р
Операторы – о-ры
Составной оператор – с
Оператор W – оп-р W

<· - начало основы;
·> - конец основы;
= - одна основа;
? - ошибка

	While	<y>	do	begin	<пр>	;	End	◄
►	<·	?	?	<·	<·	<·	?	Выход
While	?	=	?	?	?	?	?	?
<y>	?	?	=	?	?	?	?	?
do	<·	?	?	<·	<·	?	?	·>
begin	<·	?	?	<·	<·	?	·>	?
<пр>	?	?	?	<·	?	·>	·>	·>
;	<·	?	?	<·	<·	?	·>	·>
End	<·	?	?	<·	<·	?	·>	·>

While <y> do begin while <y> do <п>; <п> end

Содержание стека	Анализ-мые символы	Отн-ие	Операция	Тройка	Результат свертки
►	while	<·	Перенос		
►while	<y>	=	Перенос		
► while<y>	do	=	Перенос		
► while<y>do	begin	<·	Перенос		
► while<y>do begin	while	<·	Перенос		
► while<y>do begin while	<y>	=	Перенос		

► while<y>do begin while<y>	do	=	Перенос		
► while<y>do begin while<y>do	<пр>	<.	Перенос		
► while<y>do begin while<y>do<пр>	;	.>	Свертка	T_1 = <пр>	<о-р>
► while<y>do begin while<y>do T_1;	T_1	.>	Свертка	T_2 = <while<y>do T_1	<о-р W>
► while<y>do begin T_2 ;	<пр>	<.	Перенос		
► while<y>do begin T_2 ; <пр>	end	.>	Свертка	T_3 = <пр>	<о-р>
► while<y>do begin T_2 ; T_3 end	T_3	.>	Свертка	T_4 = T_2 ; T_3	<оп-р>
► while<y>do begin T_4 end	T_4	.>	Свертка	T_5 = begin T_4 end	<с>
► while<y>do	T_5	.>	Свертка	T_6 = while<y>do T_5	<оп-р W>
►	T_6	Конец			

19. LR(k) грамматики. Стековый метод. Пример.

LR(K) грамматика – это класс КС-грамматик. Расшифровка аббревиатуры: L – левосторонний просмотр; R – правосторонний разбор; k – количество символов, просматриваемых для однозначного определения следующего правила. В этом классе отсутствует правосторонняя рекурсия.

К этому классу относят грамматики с операторным предшествованием, простым предшествованием и расширенным предшествованием. Что дает простые методы распознавания.

Автомат выполняет две функции – свертку и перенос. Свертка происходит, когда основа заканчивается и для её свертки есть соответствующие правила. Перенос происходит в процессе свертки и заключается в сохранении в стеке очередного распознаваемого символа сентенциальной формы.

Отношения предшествования:

Если два символа $\alpha, \beta \in V$ расположены рядом в сентенциальной форме, то между ними возможны следующие отношения, названные отношениями предшествования:

- α принадлежит основе, а β – нет, т. е. α – конец основы: $\alpha \cdot> \beta$;
- β принадлежит основе, а α – нет, т. е. β – начало основы: $\alpha <\cdot \beta$;
- α и β принадлежит одной основе, т. е. $\alpha =\cdot \beta$;
- α и β не могут находиться рядом в сентенциальной форме (ошибка).

Во время синтаксического разбора часто возникают возвраты из-за неверного выбора основы, грамматика LR(k) обеспечивает же однозначный выбор основы.

Строится таблица предшествования:
где “<.” – начало основы, “.>” – конец основы, “=” – одна основа, “?” - ошибка

╲	+	*	(	)	◄
►	<.	<.	<.	?	Выход
+	.>	<.	<.	.>	.>
*	.>	.>	<.	.>	.>
(	<.	<.	<.	=	?
)	.>	.>	?	.>	.>

Стековый метод состоит из грамматики операторного предшествования, в которой существует однозначное отношение предшествования терминальных символов, которое не зависит от нетерминальных символов, находящихся между ними.

Операнды загружаются в стек операндов, когда встречается операция, берется два операнда из стека, производится операция и результат помещают в стек операндов.

Пример из РК4:

Дана грамматика, определяющая конструкцию "Оператор While":

<Оператор> ::= <Оператор W>|<Присваивание>|<Составной оператор>

<Оператор W> ::= while <Условие> do <Оператор>

<Составной оператор> ::= begin <Операторы> end

<Операторы> ::= <Операторы>;<Оператор>|<Оператор>

Используя стековый метод, разберите сентенциальную форму:

while <Условие>do

begin while < Условие> do <Присваивание>; <Присваивание> end

Уточните, к какому типу по Хомскому относится заданная грамматика. Почему?

Условие – у
Присваивание – пр
Оператор – о-р
Операторы – о-ры
Составной оператор – с
Оператор W – оп-р W

<· - начало основы;
·> - конец основы;
= - одна основа;
? - ошибка

	While	<y>	do	begin	<пр>	;	End	◄
►	<.	?	?	<.	<.	<.	?	Выход
While	?	=	?	?	?	?	?	?
<y>	?	?	=	?	?	?	?	?
do	<.	?	?	<.	<.	?	?	.>
begin	<.	?	?	<.	<.	?	.>	?
<пр>	?	?	?	<.	?	.>	.>	.>
;	<.	?	?	<.	<.	?	.>	.>
End	<.	?	?	<.	<.	?	.>	.>

While <y> do begin while <y> do <п>; <п> end

Содержание стека	Анализ-мые символы	Отн-ие	Операция	Тройка	Результат свертки
►	while	<.	Перенос		
►while	<y>	=	Перенос		
► while<y>	do	=	Перенос		
► while<y>do	begin	<.	Перенос		
► while<y>do begin	while	<.	Перенос		
► while<y>do begin while	<y>	=	Перенос		
► while<y>do begin while<y>	do	=	Перенос		
► while<y>do begin while<y>do	<пр>	<.	Перенос		
► while<y>do begin while<y>do<пр>	;	.>	Свертка	T_1 = <пр>	<о-р>
► while<y>do begin while<y>do T_1;	T_1	.>	Свертка	T_2 = <while<y>do T_1	<о-р W>
► while<y>do begin T_2 ;	<пр>	<.	Перенос		
► while<y>do begin T_2 ; <пр>	end	.>	Свертка	T_3 = <пр>	<о-р>
► while<y>do begin T_2 ; T_3 end	T_3	.>	Свертка	T_4 = T_2 ; T_3	<оп-р>
► while<y>do begin T_4 end	T_4	.>	Свертка	T_5 = begin T_4 end	<с>
► while<y>do	T_5	.>	Свертка	T_6 = while<y>do T_5	<оп-р W>
►	T_6	Конец			

20. LR(k) грамматики. Польская запись. Алгоритм Бауэра-Замельзона. Пример.

LR(K) грамматика – это класс КС-грамматик. Расшифровка аббревиатуры: L – левосторонний просмотр; R – правосторонний разбор; k – количество символов, просматриваемых для однозначного определения следующего правила. В этом классе отсутствует правосторонняя рекурсия.

К этому классу относят грамматики с операторным предшествованием, простым предшествованием и расширенным предшествованием. Что дает простые методы распознавания.

Автомат выполняет две функции – свертку и перенос. Свертка происходит, когда основа заканчивается и для её свертки есть соответствующие правила. Перенос происходит в процессе свертки и заключается в сохранении в стеке очередного распознаваемого символа сентенциальной

формы.

У метода есть проблема распознавании очередной основы.

Отношения предшествования:

Если два символа $\alpha, \beta \in V$ расположены рядом в сентенциальной форме, то между ними возможны следующие отношения, названные отношениями предшествования:

- α принадлежит основе, а β – нет, т. е. α – конец основы: $\alpha \cdot\!> \beta$;
- β принадлежит основе, а α – нет, т. е. β – начало основы: $\alpha <\!\cdot \beta$;
- α и β принадлежит одной основе, т. е. $\alpha =\cdot \beta$;
- α и β не могут находиться рядом в сентенциальной форме (ошибка).

Польская запись представляет собой последовательность команд двух типов:

K_I, где I – идентификатор операнда – выбрать число по имени I и заслать его в стек операндов;

K_ξ, где ξ – операция – выбрать два верхних числа из стека операндов, произвести над ними операцию ξ и занести результат в стек операндов.

Пример: A+B*C => $K_A K_B K_C K_* K_+$

Алгоритм Бауэра-Земельзона:

А) Если символ – операнд, то вырабатывается команда K_i

Б) Если символ – операнд, то выполняются действия согласно таблице:

η \ ξ	+	*	(	)	◄
►	<·	<·	<·	?	Выход
+	·>	<·	<·	·>	·>
*	·>	·>	<·	·>	·>
(	<·	<·	<·	=	?
)	·>	·>	?	·>	·>

η\ξ	+	*	(	)	←
→	I	I	I	?	Вых
+	II	I	I	IV	IV
*	IV	II	I	IV	IV
(	I	I	I	III	?

(слева таблица предшествования, справа таблица построения польской записи)

Операции:

1) I – заслать ξ в стек операций и читать следующий символ;

2) II – генерировать Kη (тройка), заслать ξ в стек операций и читать следующий символ;

3) III – удалить верхний символ из стека операций и читать следующий символ (не идет в стек);

4) IV – генерировать Kη и повторить с тем же входным символом, так как другой не помещается в стек.

Два этапа: 1) формирование польское записи; 2) построение польской записи

Пример:

Дана грамматика, определяющая конструкцию "Логическое выражение":

<ЛВыражение> ::= < ЛВыражение > v <Пересечение>|<Пересечение>

<Пересечение> ::= <Пересечение> ^ <Операнд> |<Операнд>

<Операнд> ::= (<ЛВыражение>)|<Ид>

Разберите логическое выражение, используя алгоритм Бауэра-Замельзона:

(<Ид> v <Ид>) ^ (<Ид> v <Ид>)

Уточните, к какому типу по Хомскому относится заданная грамматика. Почему?

Результат представить либо в виде файла Word .docx, либо в виде сканов или фотографий рукописного текста.

1) Построение польской записи

Стек операций	Символ	Действие	Команда
►	(	I	
►(	Ид		$K_{Ид}$
►(	v	I	
►(v	Ид		$K_{Ид}$

►(v	)	IV	$К_v$
►(	)	III	
►	Л	I	
►Л	(	I	
►Л(	Ид		$К_{Ид}$
►Л(	v	I	
►Л(v	Ид		$К_{Ид}$
►Л(v	)	IV	$К_v$
►Л(	)	III	
► Л	)	IV	$К_Л$
►	<-	Конец	

$К_{Ид}К_{Ид}К_vК_{Ид}К_vК_Л$

2)Выполнение операций

Стек операндов	Команда	Тройка
∅	$К_{Ид}$	
Ид	$К_{Ид}$	
Ид Ид	$К_v$	Т1 = Ид v Ид
Т1	$К_{Ид}$	
Т1 Ид	$К_{Ид}$	
Т1 Ид Ид	$К_v$	Т2 = Ид v Ид
Т1 Т2	$К_Л$	Т3 = Т1 Л Т2
Т3		

21. Способы распределения памяти под переменные. Достоинства и недостатки.

1) статическое – выполняется в процессе компиляции или загрузки в память (для сегмента неинициированных данных), используется для хранения глобальных переменных;

Достоинства: минимальная возможная фрагментация (процесс дробления чего-либо на множество мелких разрозненных фрагментов), так как на одну задачу может приходится по одной незаполненной странице.

Недостатки: межстраничные переходы осуществляются чаще, чем межсегментные, из-за чего трудно организовать разделение программных модулей между выполняющимися процессами и большой размер таблиц из-за большого количества страниц

2) автоматическое – выполняется при вызове подпрограмм, используется для локальных переменных, используемых только при запуске программы, размещаемых в стеке;

3) управляемое – выполняется по запросу программиста (new, delete), используется для динамических переменных, размещаемых в динамической памяти;

4) базированное – также выполняется по запросу программиста, но большими фрагментами (getmem, freemem), за размещение переменных отвечает программист.

Кроме того, различают локальную и глобальную память.

Локальная память предполагает ограниченный доступ. В универсальных языках программирования локальная память чаще всего отводится в стеке, т.е. для неё используется автоматическое распределение.

Глобальная память предполагает неограниченный доступ из любого места программы. Как правило это память распределяется статически.

22. Оптимизация кодов. Примеры.

Различают две группы методов:

Машинно-независимая оптимизация включает:

1) Удаление недостижимого кода – удалить то, что никогда не выполнится. Пример: if(1) s1; else s2; - else s2 никогда не выполнится

2) Оптимизация линейных участков

А) удаление бесполезных присваиваний. Пример: a = b; a = b => a = b

Б) Исключение избыточных вычислений. Пример: a = b * c; d = b * c;

=> t = b * c; a = d = t;

В) Свертка объектного кода. Пример: i= 1+1; j=1+i => i=2; j = 3

Г) Перестановка операций. Пример: a = f * 2 *c* 9 => a = (2 *9) * (b * c)

Д) Арифметические преобразования. Пример: a = b * c + b * d => a = b * (c + d)

3) подстановка кода функции вместо её вызова в объектный код – применяется к функциям\процедурам, вызываемых по адресу

4) Оптимизация циклов

А) Вынесение инвариантных вычислений из циклов. Пример: for (int I = 0; I < 5; i++) {a[i] = b*c*a[i]} => d = b*c => for (int I = 0; I <5; i++) {a[i] = d*a[i]}

Б) Замена операций с индуктивными переменными. Пример: for (i=1; i<=N; i++) a[i]=i*10; => t = 10; i = 1; => while (i<=N) {a[i]=t; t=t+10; i++;}

В) Слияние циклов. Пример: for (int i=0; i<=n; i++) for (int j= 0; j<= m; j++) a[i][j] = 0; => k = n * m => k = n * m; for (i=1; i<=k; i++) a[i] = 0;

Г) Развертывание цикла (только заранее известно сколько раз он будет проходить). Пример: for (int I = 0; I < 2; i++) {a[i] = 2} => a[1] = 2; a[2] = 2;

Машинно-зависимая оптимизация включает:

1) Распределение регистров процессора - использование регистров общего назначения и специальных для хранения значения операндов и результатов вычислений, что позволяет увеличить быстродействие программы.

2) Оптимизация передачи параметров в процедуры и функции - Обычно параметры процедур и функций передаются через стек. При этом всякий раз при вызове процедуры или функции компилятор создает объектный код для размещения ее фактических параметров в стеке, а при выходе из нее - код для освобождения соответствующей памяти.

Можно уменьшить код и время выполнения результирующей программы за счет оптимизации передачи параметров в процедуру или функцию, передавая их через регистры процессора.

3) Оптимизация кода для процессоров, допускающих распараллеливание вычислений - при возможности параллельного выполнения нескольких операций компилятор должен порождать объектный код таким образом, чтобы в нем было максимально возможное количество соседних операций, все операнды которых не зависят друг от друга. Пример: a + b + c + d => для одного потока: ((a + b) + c) + d и для двух: (a + b) + (c + d)