ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение
высшего профессионального образования
ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

О.В. Караваева

Ассемблеры и компиляторы

Учебное пособие

Киров 2011

Печатается по решению редакционно-издательского совета Вятского государственного университета

004.451(07)
К43

Караваева О.В. Ассемблеры и компиляторы: Изд-во ВятГУ, 2011. – 82 с.

Рецензент: кафедра «Информатики и математики» Московского гуманитарно-экономического института (Кировский филиал)

В учебном пособии описаны основные средства синхронизации процессов и потоков. Рассмотрено решение классических задач синхронизации и даны варианты заданий для выполнения лабораторных работ.

Пособие рассчитано на студентов, обучающихся по специальности 230101 (220100) «Вычислительные машины, комплексы, системы и сети» и изучающих дисциплины «Параллельное программирование» и «Операционные системы». Оно может быть полезным студентам других специальностей при знакомстве с использованием средств операционных систем для синхронизации процессов и потоков.

Редактор Н.Ю. Целищева

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время искусственные языки, использующие для описания предметной области текстовое представление, широко применяются не только в программировании, но и в других областях. С их помощью описывается структура всевозможных документов, трехмерных виртуальных миров, графических интерфейсов пользователя и многих других объектов, используемых в моделях и в реальном мире. Для того, чтобы эти текстовые описания были корректно составлены, а затем правильно распознаны и интерпретированы, используются специальные методы их анализа и преобразования. В основе методов лежит теория языков и формальных грамматик, а также теория автоматов. Программные системы, предназначенные для анализа и интерпретации текстов, называются трансляторами.

Транслятор - *обслуживающая программа, преобразующая исходную программу, предоставленную на входном языке программирования, в рабочую программу, представленную на объектном языке*.

Приведенное определение относится ко всем разновидностям транслирующих программ. Однако у каждой из таких программ могут быть свои особенности по организации процесса трансляции. В настоящее время трансляторы разделяются на три основные группы: ассемблеры, компиляторы и интерпретаторы.

Ассемблер - *системная обслуживающая программа, которая преобразует символические конструкции в команды машинного языка*.

Компилятор - *это обслуживающая программа, выполняющая трансляцию на машинный язык программы, записанной на исходном языке программирования*.

Ассемблеры отличаются от компиляторов с языков высокого уровня тем, что переводят в машинные коды машинно-ориентированные языки. Ассемблер обрабатывает программу, команды которой отображают внутреннюю структу-

ру процессора и памяти. Существует ряд основных функций, таких как: трансляция мнемонических кодов операций в их эквиваленты на машинном языке или присваивание машинных адресов символическим меткам, которые должны выполняться любым ассемблером. Однако, за пределами этого базового уровня возможности, предоставляемые ассемблерами, а также схемы их построения, сильно зависят как от входного, так и от машинного языков. Одним из аспектов этой машинной зависимости являются имеющиеся различия в форматах машинных команд и кодах операций. С другой стороны некоторые средства языка ассемблера не имеют прямой связи со структурой машины, их выбор является произвольным и определяется разработчиком языка ассемблера.

Компилятор обеспечивает преобразование программы с одного языка на другой (чаще всего в язык конкретного компьютера). Команды исходного языка значительно отличаются по организации и мощности от команд машинного языка. Существуют языки, в которых одна команда исходного языка транслируется в 7-10 машинных команд. Однако есть и такие языки, в которых каждой команде может соответствовать 100 и более машинных команд (например, Пролог). Кроме того, в исходных языках достаточно часто используется строгая типизация данных, осуществляемая через их предварительное описание. Программирование может опираться не на кодирование алгоритма, а на тщательное обдумывание структур данных или классов. Процесс трансляции с таких языков обычно называется компиляцией, а исходные языки обычно относятся к языкам программирования высокого уровня (или высокоуровневым языкам). Абстрагирование языка программирования от системы команд компьютера привело к независимому созданию самых разнообразных языков, ориентированных на решение конкретных задач. Появились языки для научных расчетов, экономических расчетов, доступа к базам данных и другие.

Интерпретатор - *программа или устройство, осуществляющее пооператорную трансляцию и выполнение исходной программы*. В отличие от ком-

пилятора, интерпретатор не порождает на выходе программу на машинном языке. Распознав команду исходного языка, он тут же выполняет ее. Как в компиляторах, так и в интерпретаторах используются одинаковые методы анализа исходного текста программы. Но интерпретатор позволяет начать обработку данных после написания даже одной команды. Это делает процесс разработки и отладки программ более гибким.

Практически во всех трансляторах (и в компиляторах, и в интерпретаторах) в том или ином виде присутствует большая часть перечисленных ниже процессов:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- генерация внутреннего представления программы;
- оптимизация;
- генерация объектной программы.

В конкретных компиляторах порядок этих процессов может быть несколько иным, некоторые из них могут объединяться в одну фазу, другие могут выполняться в течение всего процесса компиляции.

1 Структура исполняемых файлов

1.1 Синтаксические определения записи машинных команд

Команды центрального процессора рассматриваются как операционные ресурсы нижнего уровня, доступные программисту. Использование операционных ресурсов планируется программистом в виде последовательности машинных команд процессора, которые в языке Ассемблера отличаются названием операции и форматами операндов, что синтаксически определяется режимами адресации. Операнд команды в языке Ассемблера определяется названием регистра, содержимое которого принимает участие в операции, или выражением, обрабатываемым во время трансляции. Это выражение может включать операции: арифметические (+, -, *, /, MOD), логические (AND, OR, NOT, XOR), сдвига (SHL, SHR), отношений (EQ, NE, LT, LE, GE, GT) с традиционными приоритетами и включенными именами меток и областей памяти для хранения переменных и констант. Главной особенностью языка Ассемблера базового процессора по сравнению с предыдущими аналогичными языками является использование единого названия операций для всех семантически идентичных действий независимо от форматов и размеров данных. Форматы данных определяются типом именованных областей памяти или явными указателями в форме: *название типа данных* PTR *операнд из памяти.*

Для систематизации системы команд нужно выделить базовые способы адресации данных: сегментная регистровая ***s*** и регистровая ***r***, когда в команде указан 8-, 16- или 32-битовый регистр, где находятся данные. Кроме того, допускается непосредственная адресация ***i,*** когда данные есть часть машинной команды и группа способов адресации данных в памяти ***m***.

Последняя группа охватывает все варианты адресации памяти в пределах доступности сегментного регистра (64Кбайтов в реальном режиме или 4Гбайтов в защищенном режиме, начиная с процессора 80386). Сегмент может быть определен по умолчанию или задан явно, через имя используемого сег-

ментного регистра, отделенное от следующего адреса двоеточием, например, ES: *адрес*.

Различают пять типов адресов.

– Прямой 16-битовый адрес данного как часть команды, определяемый выражением, включающим в качестве базы название области памяти или числовой адрес, заключенный в квадратные скобки, например ARRAY+4 или [376H], в режиме 32-битовой адресации заменяется 32-битовым адресом.

– Регистровый косвенный адрес, по которому адрес данного находится в базовом или индексном регистре, выделенном в ассемблерной записи команды квадратными скобками, например [BX] или [DI]. В процессорах 80386 и выше допускается использовать любой 32-битовый регистр, кроме ESP.

– Регистровая относительная или индексная адресация, которую можно считать расширением регистровой косвенной через добавление 8- или 16-битового смещения к содержимому базового или индексного регистра, например DESTAR[DI], [Sl+5], SORCAR+5[BP]. В режиме 32-битового адреса вместо 16-битового смещения используется 32-битовое, и может задаваться любой 32-битовый регистр, кроме ESP.

– Регистровый базово-индексный адрес данного вычисляется как сумма содержимого базового и индексного регистров, определенных в команде, например SS:[BP][DI], где регистры BP и DI равноправны и хранят базовый адрес и индекс очередного объекта. В 32-битовом режиме ссылка на индексный регистр может сопровождаться масштабным множителем индекса 2, 4 или 8, что ускоряет индексацию данных основных типов. Например, GS:[EBX+EAX*8] вызовет обращение к двойному слову массива в сегменте GS с базой EBX, индекс или номер которого находится в регистре EAX.

– Относительный базово-индексный адрес можно считать расширением регистрового базово-индексного адреса, который вычисляется как сумма 8- или 16-битового смещения и содержимого базово-индексного регистра. Такой адрес дает возможность включения в команду постоянного смещения и ис-

пользования двойных индексов, например [BP][SI+5], DESTAR[BX][DI], SORCAR+5[BP][SI]. Адреса 32-битового режима задаются по аналогии с третьим и четвертым типом адресов.

1.2 Структура команды процессора

Машинная команда должна содержать следующую информацию:

– код операции. Помещается в первом байте команды. Код предполагает работу без операндов, с одним или двумя операндами.

– способ адресации и собственный адрес. В команде должно быть закодировано, содержатся ли операнды в регистрах, или это непосредственные данные, или операнд находится в памяти.

Форматы команд МП в языке Ассемблера строятся в соответствии двоичной внутренней машинной формой представления, по которой обобщенный формат использует базовую двухбайтную структуру для построения основной части команды, имеющей вид

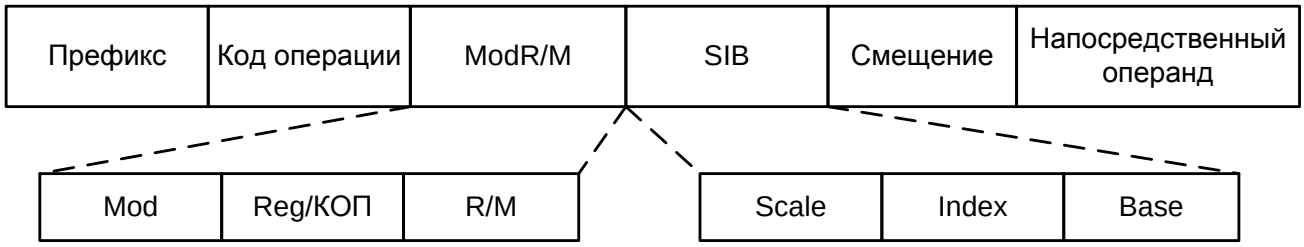


Рисунок 1.1 – Формат команды процессора x86 фирмы Intel

Префиксы – это необязательные элементы машинной команды, каждый из которых состоит из одного байта или может отсутствовать. В памяти префиксы предшествуют команде. Назначение префиксов — модифицировать операцию, выполняемую командой. Прикладная программа может использовать следующие типы префиксов:

– префикс замены сегмента. В явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных. Префикс отменяет выбор сегментного регистра по умолчанию. Префиксы замены сегмента имеют следующие значения:

2eh — замена сегмента cs;

36h — замена сегмента ss;

3eh — замена сегмента ds;

26h — замена сегмента es;

64h — замена сегмента fs;

65h — замена сегмента gs.

Если режим адресации использует регистр BP для формирования физического адреса, по умолчанию используется содержимое сегментного регистра SS, в других режимах адресации – DS. Если в команде используется префикс замены сегмента, то длина основной команды увеличивается на один байт префикса переключения сегментов, а время выполнения – на два цикла.

Аналогичная замена невозможна при вычислении адреса текущей команды, в этом случае всегда используется регистр CS. При выполнении основных манипуляций записи-чтения со стеком с использованием регистра SP всегда подключается сегментный регистр SS, а для адресации операнда-приемника в командах обработки строк, используется только сегментный регистр ES.

– префикс разрядности адреса уточняет разрядность адреса (32 или 16-разрядный). Каждой команде, в которой используется адресный операнд, ставится в соответствие разрядность адреса этого операнда. Этот адрес может иметь разрядность 16 или 32 бит. Если разрядность адреса для данной команды 16 бит, это означает, что команда содержит 16-разрядное смещение, оно соответствует 16-разрядному смещению адресного операнда относительно начала некоторого сегмента. Если разрядность адреса 32 бит, это означает, что команда содержит 32-разрядное смещение, оно соответствует 32-разрядному смещению адресного операнда относительно начала сегмента и по его значению формируется 32-битное смещение в сегменте. С помощью префикса разрядности адреса можно изменить действующее по умолчанию значение разрядности адреса. Это изменение будет касаться только той команды, которой предшествует префикс.

– префикс разрядности операнда аналогичен префиксу разрядности адреса, но указывает на разрядность операндов (32 или 16-разрядные), с которыми работает команда. В реальном режиме и режиме виртуального i8086 значения атрибутов — 16 бит. В защищенном режиме значения атрибутов зависят от состояния бита D в дескрипторах исполняемых сегментов. Если D = 0, то значения атрибутов, действующие по умолчанию, равны 16 бит; если D = 1, то 32 бит.

Значения префиксов разрядности операнда 66h и разрядности адреса 67h.

– префикс повторения используется с цепочечными командами (командами обработки строк). Этот префикс “зацикливает” команду для обработки всех элементов цепочки. Система команд поддерживает два типа префиксов:

– безусловные (rep — 0f3h), заставляющие повторяться цепочечную команду некоторое количество раз;

– условные (repe/repz — 0f3h, repne/repnz — 0f2h), которые при зацикливании проверяют некоторые флаги, и в результате проверки возможен досрочный выход из цикла.

Код операции - это обязательный элемент, описывающий операцию, выполняемую командой. Многим командам соответствует несколько кодов операций, каждый из которых определяет нюансы выполнения операции.

Последующие поля машинной команды определяют местоположение операндов, участвующих в операции, и особенности их использования. Само поле кода занимает 8 бит и имеет следующий формат

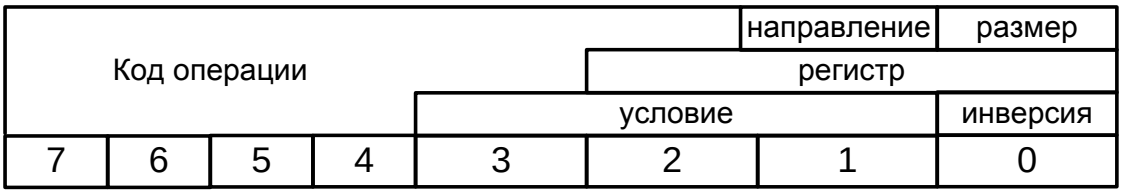


Рисунок 1.2 – Формат поля «Код операции»

Поле размера равно нулю, если операнды имеют размер 1 байт. Единич-

ное значение указывает на слово.

Направление обозначает операнд-приемник. При нулевом значении результат присваивается правому операнду, при единичном – левому.

Размер данных и количество следующих байтов команды определяется конкретным кодом операции и наличием в команде фиксированных байтов префиксов переключения разрядности данных и адресации.

Байт режима адресации modr/m (пост-байт). Значения этого байта определяет используемую форму адреса операндов. Операнды могут находиться в памяти в одном или двух регистрах. Если операнд находится в памяти, то байт modr/m определяет компоненты (смещение, базовый и индексный регистры), используемые для вычисления его эффективного адреса . В защищенном режиме для определения местоположения операнда в памяти может дополнительно использоваться байт sib (Scale-Index-Base — масштаб-индекс-база). Байт modr/m состоит из трех полей.

Поле mod

Определяет количество байт, занимаемых в команде адресом операнда. Поле mod используется совместно с полем r/m, которое указывает способ модификации адреса операнда смещение в команде.

К примеру, если mod = 00, это означает, что поле смещение в команде отсутствует, и адрес операнда определяется содержимым базового и (или) индексного регистра. Какие именно регистры будут использоваться для вычисления эффективного адреса, определяется значением этого байта.

Если mod = 01, это означает, что поле смещение в команде присутствует, занимает один байт и модифицируется содержимым базового и (или) индексного регистра.

Если mod = 10, это означает, что поле смещение в команде присутствует, занимает два или четыре байта (в зависимости от действующего по умолчанию или определяемого префиксом размера адреса) и модифицируется содержимым базового и (или) индексного регистра.

Если mod = 11, это означает, что операндов в памяти нет: они находятся в регистрах. Это же значение байта mod используется в случае, когда в команде применяется непосредственный операнд.

Поле reg/кoп

Определяет либо регистр, находящийся в команде на месте первого операнда, либо возможное расширение кода операции.

Поле r/m

Используется совместно с полем mod и определяет либо регистр, находящийся в команде на месте первого операнда (если mod = 11), либо используемые для вычисления эффективного адреса (совместно с полем смещение в команде) базовые и индексные регистры.

Допустимые варианты 16-битових адресов процессоров ix86 приведены в таблице 1.1 с явным указанием сегментных регистров, используемых по умолчанию, и количества дополнительных циклов, затрачиваемых на адресацию в процессоре 8086.

Таблица 1.1 – 16-битовые адреса процессоров ix86

r/m	*Mod=00*	*дц*	*mod=01/10*	*mod=11*		
				дц	*w=0*	*w=1*
000	DS:[BX][SI]	7	DS:d8/dl6[BX][SI]	11	AL	AX
001	DS:[BX][DI]	7	DS:d8/dl6[BX][DI]	11	CL	CX
010	SS:[BP][SI}	8	SS:d8/dl6[BP][SI]	12	DL	DX
011	SS:[BP][DI]	8	SS:d8/dl6[BP][DI]	12	BL	BX
100	DS:[SI]	5	DS:d8/dl6[SI]	9	AH	SP
101	DS:[DI]	5	DS:d8/dl6[DI)	9	CH	BP
110	DS:dl6	6	SS:d8/dl6[BP]	9	DH	SI
111	DS:[BX]	5	DS:d8/dl6[BX]	9	BH	Dl

В таблице [*регистр*] – обозначает содержимое указанного регистра, добавляемое при формировании исполнительного адреса в пределах 64 Кбайтов используемого сегмента, а имя сегментного регистра определяет использова-

ние сегментного регистра по умолчанию. Дополнительные циклы обращения (дц) приведены для процессоров Intel-8086/88. В 32-битовых процессорах при задании префикса 32-битовых данных вместо двухбайтных регистров используются четырехбайтные с дополнительной начальной буквой Е.

Байт масштаб-индекс-база (байт sib) используется для расширения возможностей адресации операндов.

На наличие байта sib в машинной команде указывает сочетание одного из значений 01 или 10 поля mod и значения поля r/m= 100. Байт sib состоит из трех полей:

- *поля масштаба ss.* В этом поле размещается масштабный множитель для индексного компонента index, занимающего следующие три бита байта sib. В поле ss может содержаться одно из следующих значений: 1, 2, 4, 8. При вычислении эффективного адреса на это значение будет умножаться содержимое индексного регистра;
- *поля index* используется для хранения номера индексного регистра, который применяется для вычисления эффективного адреса операнда;
- *поля base* используется для хранения номера базового регистра, который также применяется для вычисления эффективного адреса операнда. Напомню, что в качестве базового и индексного регистров могут использоваться практически все регистры общего назначения.

В машинной форме после основной части команды с указателями модификаций адреса в памяти может располагаться: двухбайтный адрес для прямой адресации, однобайтное или двухбайтное смещение, одно- или двухбайтные непосредственные данные или двухбайтное смещение и двухбайтный сегментный адрес для прямой межсегментной адресации. Длина команд ЦП без префиксов может достигать шести байтов. Некоторые группы команд имеют дополнительные сокращенные форматы, которые обрабатываются несколько быстрее.

1.3 Внутренние структуры данных процессоров

Транслятор с языка Ассемблера автоматически контролирует допустимость операнда и генерирует коды, включая префиксы, в соответствии с заданной моделью и режимом работы процессора для любого операнда, указывающего на память и заданного в обобщенной стандартной форме:

Имя_области_памяти_со_смещением[имя_регистра_базы]

*[имя_регистра_индекса * коэффициент]*

В этой записи может быть опущен любой из трех элементов или коэффициент. 32-разрядный эффективный адрес формируется в соответствии с таблицей 1.2.

Таблица 1.2 - 32-разрядный эффективный адрес

	Префикс 66h		
r/m	*mod=00*	*Mod=01/10*	*w=0, w=1 mod =11*
000	DS:[EAX]	DS:d8/d32[EAX]	AL AX EAX
001	DS:[ECX]	DS:d8/d32[ECX]	CL CX ECX
010	DS:[EDX]	DS:d8/d32[EDX]	DL DX EDX
01l	DS:[EBX]	DS:d8/d32[EBX]	BL BX EBX
100	Управление байтом SIB		AH SP ESP
101	DS:d32	SS:d8/d32[EBP]	CH BP EBP
110	DS:[ESI]	DS:d8/d32[ESI]	DH SI ESI
111	DS:[EdI]	DS:d8/d32 EDI]	BH Dl EDI

Таким образом, обобщенный формат команды ЦП 80386 может включать префиксы 4-байтных данных и адресов, а также байт SIB после постбайта. Байт SIB содержит 3-битовые поля INDEX и BASE, определяющие выбор регистров, используемых в качестве индексного и базового, и 2-битовое поле SCALE, задающее масштабный коэффициент для модификации значения индекса.

Формат байта SIB:

ss	*iii*	*bbb*

ss – масштаб *scale:*

00 => [iii]*1

10 => [iii]*2

01 => [iii]*4

11 => [iii]*8

iii – индексный регистр *index*

bbb – базовый регистр *base*

Если поле ***r/m*** пост-байта имеет значение 100, то используется байт SIB и формирование 32-разрядного эффективного адреса определяется по таблице.

Таблица 1.3 - формирование 32-разрядного эффективного адреса

	префикс 66h				
Base	*mod=00*	*mod=01/10*	*w=0*	*w=1*	*mod=11*
000	DS:[EAX+(I*S)]	DS:d8/d32[EAX+(l*S)]	AL	AX	EAX
001	DS:[ECX+(l*S)]	DS:d8/d32[ECX+(l*S)]	CL	CX	ECX
010	DS:[EDX+(I*S)]	DS:d8/d32[EDX+(l*S)]	DL	DX	EDX
011	DS:[EBX+(I*S)]	DS:d8/d32[EBX+(I*S)]	BL	BX	EBX
100	SS:[ESP+(1*S)]	SS:d8/d32[ESP+(l*S)]	AH	SP	ESP
101	DS:[d32+(I*S)]	DS:d8/d32[EBP+(I*S)]	CH	BP	EBP
110	DS:[ESI+(I*S)]	DS:d8/d32[ESI+(l*S)]	DH	SI	ESI
111	DS:[EDI+(l*S)]	DS:d8/d32[EDI+(I*S)]	BH	DI	EDI

Здесь I – имя индексного регистра, в качестве которого нельзя использовать только SP, а S – масштабный множитель для основных форматов данных. Кодировка индексных и базовых регистров в байте SIB соответствует стандартным номерам регистров. В этом случае длина команды может быть от 1 до 11 байт, а эффективный адрес вычисляется по формуле

$$EA=(base)+(index)*scale+disp.$$

Так в команде

ADD ECX, TB[EDI*8] – ***disp*** определется ассемблером, ***index*** = EDI, а ***scale****=3.*

Однако, так как вычисление адреса выполняется параллельно с другими действиями, дополнительное время не требуется, только при совместной обработке и BASE, и INDEX, и DISP время выполнения команды увеличивается на 1 такт.

Если MOD не равно 11, способ задания исполнительного адреса определяется полем R/M. Поле R/M кодирует все способы адресации, кроме прямой.

	R/M	EA		
	000	[BX] + [SI]	+disp	BASE + INDEX + DISP
	001	[BX] + [DI]	+disp	
	010	[BP] + [SI]	+disp	
	011	[BP] + [DI]	+disp	
	100	[SI]	+disp	INDEX + DISP
	101	[DI]	+disp	
*	110	[BP]	+disp	BASE + DISP
	111	[BX]	+disp	

Поле смещения DISP может отсутствовать, быть 8- или 16-разрядным. Число байтов поля смещения определяется битами MOD.

MOD	*Индексное смещение*
00	DISP=0, т.е. байты индексного смещения отсутствуют
01	-128 < DISP < 127, один байт индексного смещения
10	-32768 < DISP < 32767, два байта индексного смещения
11	R/M – поле регистра

Механизмом работы пост-байта не предусмотрена прямая адресация. Микропроцессор трактует случай * с нулевой длиной индексного смещения как прямую адресацию. При этом длина поля индексного смещения равна 2

байтам и в вычислении адреса не участвует ни один регистр. В связи с этим для доступа к ячейке памяти, на которую указывает регистр BP, необходимо наличие 1-байтового поля индексного смещения с нулевым значением. В этой же ситуации, но с использованием регистров BX, SI или DI байт смещения не требуется.

1.4 Выполнение лабораторной работы «Дизассемблирование»

Номер задания соответствует номеру компьютера, на котором работает студент. При запуске программной модели лабораторной работы в окне «Просмотр кода» находится исполнимый модуль программы в шестнадцатеричном виде, который необходимо декодировать. Окно просмотра доступно на протяжении всего времени выполнения лабораторной работы.

Выполнение лабораторной работы состоит из четырех шагов, причем переход к следующему шагу возможен лишь в том случае, если данный шаг выполнен правильно.

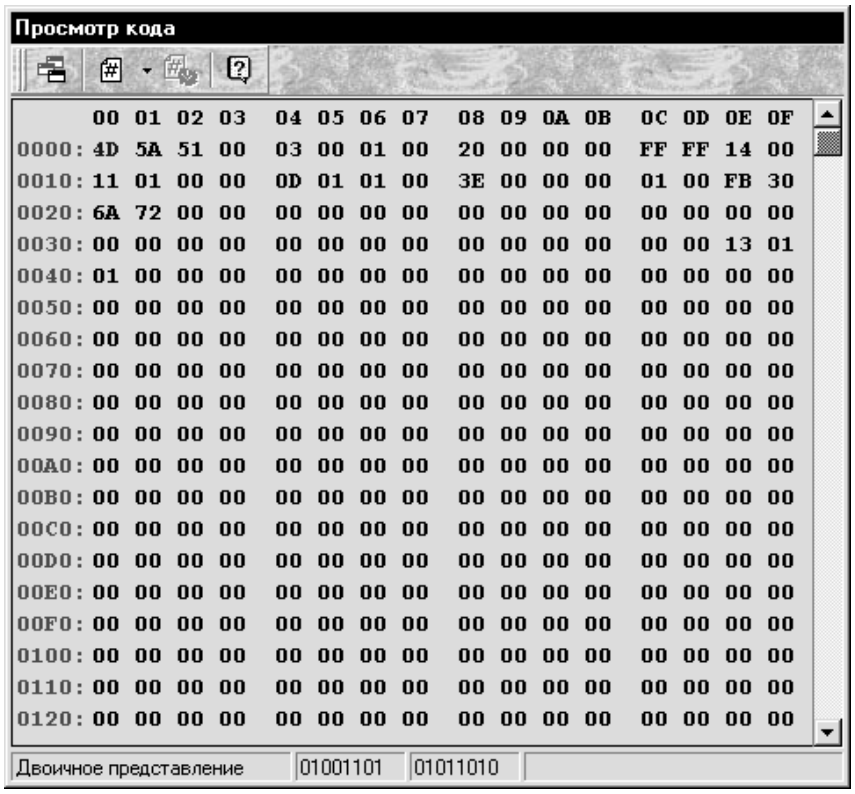


Рисунок 1.3 – Просмотр файла в шестнадцатеричном представлении

Первый шаг – заголовок EXE-файла

На первом шаге необходимо заполнить поля заголовка EXE-файла, то есть предлагается пройти тест на знание формата заголовка EXE-файла и умение пользоваться просмотром файла в шестнадцатеричном представлении для получения необходимой информации.

EXE-файлы создаются редактором связей (компоновщиком) и состоят из двух частей:

- заголовок;
- выполнимый модуль.

Заголовок состоит из управляющей информации и таблицы перемещаемых символов.

Таблица 1.4 - Управляющая информация в заголовке EXE-файла

Смещение	Название	Содержание
00H	Sign	4Dh, 5Ah – идентификатор EXE файла (“MZ”)
02H	PartPag	Число байтов в последнем блоке EXE файла
04H	PageCnt	Длина файла в блоках по 512 байтов
06H	ReloCnt	Количество элементов в таблице перемещаемых символов
08H	HdrSize	Размер заголовка в параграфах (по 16 байтов)
0AH	MinMem	Минимальное количество параграфов, необходимых после загруженной программы (обычно 0)
0CH	MaxMem	Максимальное количество параграфов, необходимых после загруженной программы (FFFFh) для программ загружаемых в младшие адреса памяти; 0000h для программ загружаемых в старшие адреса
0EH	ReloSS	Смещение сегмента стека в выполнимом модуле (в параграфах)
10H	ExeSP	Значение SP при получении управления выполнимым модулем
12H	ChkSum	Контрольная сумма файла
14H	ExeIP	Значение IP, когда выполнимый модуль получает управление
16H	ReloCS	Смещение сегмента кода в выполнимом модуле (в параграфах)
18H	TablOff	Начало таблицы перемещаемых символов (смещение с начала файла в байтах)
1AH		Номер перекрытия (0 для резидентной части програм-

Смещение	Название	Содержание
		мы)

Выполнимый модуль содержит инструкции и данные программы и находится непосредственно за заголовком. В выполнимом модуле можно иметь адресные константы, которые нужно настроить для конкретного адреса загрузки. Это адреса сегментов, значения которых в модуле вычислены относительно его начала. Во время выполнения программы адресные константы должны содержать абсолютные адреса. Преобразование относительных адресов в абсолютные выполняется системной программой загрузки, которая использует информацию из таблицы перемещаемых символов, где описано местоположение адресных констант в выполнимом модуле. Например, в командах

```
Mov AX,DataSeg
Mov DS,AX
… или …
Call MyFarProc
```

должны быть выполнены преобразования в соответствии с тем, в какие адреса загружена программа. Для этой цели служит таблица перемещений (Relocation Table). Таблица перемещаемых символов состоит из четырехбайтных записей в виде: *сегмент*: *смещение*.

При загрузке файла в память DOS производит следующие действия с таблицей перемещаемых символов:

a) считывает элемент таблицы (I_Off, I_Seg);

b) прибавляет к I_Seg начальный сегмент (StartSeg) загруженного файла: ReloSeg=StartSeg+I_Seg;

c) считывает слово по адресу ReloSeg : I_Off;

d) прибавляет Start_Seg к считанному слову;

e) записывает обратно полученное слово.

После преобразования адресов сегментов управление передается загруженной программе посредством длинного перехода на точку старта программы.

Когда программа получает управление, регистры DS и ES устанавливаются на PSP программы, а регистры CS, IP, SS и SP принимают значения, указанные в заголовке EXE – файла.

Второй шаг – декомпилирование EXE-файла

На втором шаге задания предлагается декомпилировать EXE-файл (только основную часть кодового сегмента – без учета возможных процедур)

Для начала декодирования надо определить точку входа в файле, то есть смещение, по которому находится первая выполняемая команда при передаче управления данному EXE-файлу. Затем, начиная с этого адреса, надо декодировать команды и вводить их в окно текста программы в формате языка Ассемблера.

Адрес точки входа в EXE-файле определяется по формуле

$$(HdrSize+ReloCS)*10h+ExeIP.$$

Длина загружаемой части EXE-файла определяется по формуле

$$((PageCnt*512)-(HdrSize*16)) – PartPag.$$

Для того чтобы правильно интерпретировать машинную команду необходимо перед выполнением лабораторной работы изучить теоретическую часть данных методических указаний, познакомиться с форматами команд и способами адресации. В приложении 1 даны коды инструкций, которые используются при выполнении лабораторной работы.

Рассмотрим пример формата, используемого в описании каждой команды на примере команды PUSH.

PUSH

Описание

Помещение операнда в стек

КОП	**Команда**	**Описание**
FF /6	PUSH m16	Помещение в стек слова памяти
50+rw	PUSH r16	Помещение в стек слова – регистра
0E	PUSH CS	Помещение в стек регистра CS
16	PUSH SS	Помещение в стек регистра SS
1E	PUSH DS	Помещение в стек регистра DS
06	PUSH ES	Помещение в стек регистра ES

В колонке «Код операции» содержится полный объектный код, генерируемый для каждой формы команды. Где это возможно, коды приводятся в шестнадцатеричной байтовой записи, в последовательности их расположения в памяти. Определения элементов, помимо шестнадцатеричных байтов, следующие:

/цифра (цифра от 0 до 7) указывает на то, что байт ModR/M команды использует только операнд r/m (регистр или память). Поле reg содержит цифру, представляющую собой расширение кода операции команды.

/r: указывает на то, что байт ModR/M команды содержит и операнд регистра, и операнд r/m.

Cb, cw: 1-байтовое (cb) или 2-байтовое (cw) значение, следующее за кодом операции, используемое для задания смещения кода и, возможно, нового значения регистра кодового сегмента.

Ib, iw: 1-байтовый (ib) или 2-байтовый (iw) непосредственный операнд команды, следующий за кодом операции, байтом ModR/M. Код операции

определяет, является ли данный операнд значением со знаком. Все слова и двойные слова представлены таким образом, что первым следует младший байт.

+rb, +rw: Код регистра, от 0 до 7, складываемый с шестнадцатеричным байтом, находящимся слева от знака “плюс”, образуя единый байт кода операции.

Колонка «Команда» дает синтаксис оператора команды в том виде, в котором этот оператор записывается в программе на языке Ассемблера. Ниже приводится список символических имен, используемых для представления операндов в операторах команды:

rel8 - относительный адрес в диапазоне 128 байтов от 128 байтов до конца команды до 127 байтов после конца команды.

rel16 - относительный адрес в пределах того же кодового сегмента, что и ассемблируемая команда. Rel16 применяется в командах с атрибутом размера операнда, равным 16 битам.

r8 - один из байтовых регистров AL,CL,DL,BL,AH,CH,DH или BH.

r16 - один из регистров размером в слово AX,CX,DX,BX,SP,BP,SI или DI.

imm8 - непосредственное значение байта. imm8 - это число со знаком в диапазоне от -127 до +127, включительно. Для команд, в которых imm8 комбинируется с операндом размером в слово, непосредственное значение расширяется по знаку, образуя слово. Старший байт слова заполняется самым старшим битом непосредственного значения.

imm16 - непосредственное значение размером в слово, используемое для команд, атрибут размера операнда которых равен 16 битам. Это число в диапазоне от –32768 до +32767 включительно.

R/m8 - однобайтовый операнд, представляющий собой либо содержимое байтового регистра (AL, BL, CL, DL, AH, BH, CH, DH), либо содержимое байта в памяти.

R/m16 - операнд регистра-слова или операнд памяти, используемый в командах, атрибут размера операнда которых равен 16 битам. Содержимое памяти находится по адресу, получаемому при вычислении исполнительного адреса.

Moffs8, moffs16 (смещение в памяти) – простая переменная памяти типа BYTE или WORD, используемая некоторыми вариантами команды MOV. Фактический адрес задается простым смещением относительно базы сегмента. В команде байт ModR/M не используется. Число, указанное в moffs, обозначает размер, определяемый атрибут размера адреса команды.

Sreg: сегментный регистр. Назначения битов сегментных регистров: ES=0, CS=1, SS=2, DS=3.

Колонка «Описание» содержит краткое описание разных форм команды.

Пример декодирования команды

Пусть есть следующая часть кода:

F0 26 FF 8B 34 12 C7 83 34 12 78 56 CD 21 …

1) Ищем в таблице кодов инструкций байт F0h.

2) Найден префикс блокировки шины LOCK.

3) Ищем в таблице кодов инструкций байт 26h.

4) Найден префикс замены сегмента ES.

5) Ищем в таблице кодов инструкций байт FFh.

6) Найдено несколько инструкций. Для выбора нужной требуется декодировать следующий байт (это байт ModR/M).

7) Переводим в двоичное представление байт 8Bh:

8Bh= 10 001 011

mod r r/m

8) Поле r равно 1. Это добавочный код операции. Опять обращаемся к таблице кодов инструкций и ищем инструкцию с кодом операции FFh и добавочным кодом 1 (т.е. КОП=FFh /1). Это инструкция DEC r/m16.

9) Так как поле mod <>11b, значит используется адресация к памяти.

10) По таблице адресаций получаем адресацию вида [BP+DI+d16]. Значит, следующие 2 байта являются 16-разрядным индексом, т.е. d16=1234h.

Итак, получена команда: Lock Dec Word Ptr ES:[BP][DI+1234h].

Для указания размера операнда (так как его размер явно в записи команды не указан) в команде ставится указатель типа Word Ptr.

Третий шаг – описание действий, выполняемых командами

В данном задании предлагается для каждой строки текста программы составить список действий, которые она выполняет.

Добавление в список действий осуществляется при помощи диалогового окна, описанного ниже. Окно представлено на рисунке 1.4.

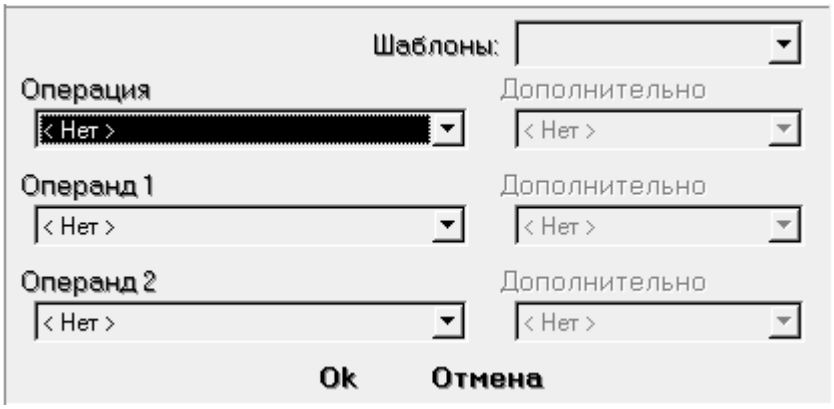


Рисунок 1.4 – Окно редактирования действий, выполняемых командой

Основные поля

В поле «Операция» указывается название микрооперации. В поле «Операнд 1» указывается первый операнд (приемник), а в поле «Операнд 2» указывается второй операнд (источник)

В списке «Шаблоны» занесены все ранее введенные описания действий для упрощения занесения повторяющихся действий.

Дополнительные поля

Для поля операции дополнительное поле активно только в том случае, когда в поле операции указан сдвиг. Для полей операндов дополнительные поля активизируются, когда в поле операнда указан регистр или память.

Пример:

Рассмотрим микрооперацию «исключающее или». В качестве операндов рассмотрим DX и число 1234h, т.е. DX:=DX xor 1234h. Для задания этой операции в поле операции заносим «Исключающее ИЛИ», в поле первого операнда заносим «Регистр общего назначения» (так как приемником в этой операции является DX – это РОН), в дополнительное поле первого операнда заносим «DX».

В поле второго операнда заносим «Непосредственный операнд», а в дополнительном поле второго операнда указываем непосредственное значение 1234h.

Таким же образом необходимо заполнить списки действий для каждой команды.

2 Основные функции компиляторов

Большинство языков высокого уровня построены так, чтобы быть независимыми от тех компьютеров, на которых они будут использоваться. Это означает, что процесс анализа синтаксиса программ, написанных на этих языках должен быть также машинно-независимым, но процесс генерации кода является машинно-зависимым, поскольку необходимо знать систему команд компьютера, для которого этот код генерируется. Компилятор должен выполнять анализ исходной программы, а затем синтез объектной программы. Сначала исходная программа разлагается на её составные части, затем из них строятся части эквивалентной объектной программы. Для этого на этапе анализа компилятор строит несколько таблиц, которые затем используются как при анализе, так и при синтезе.

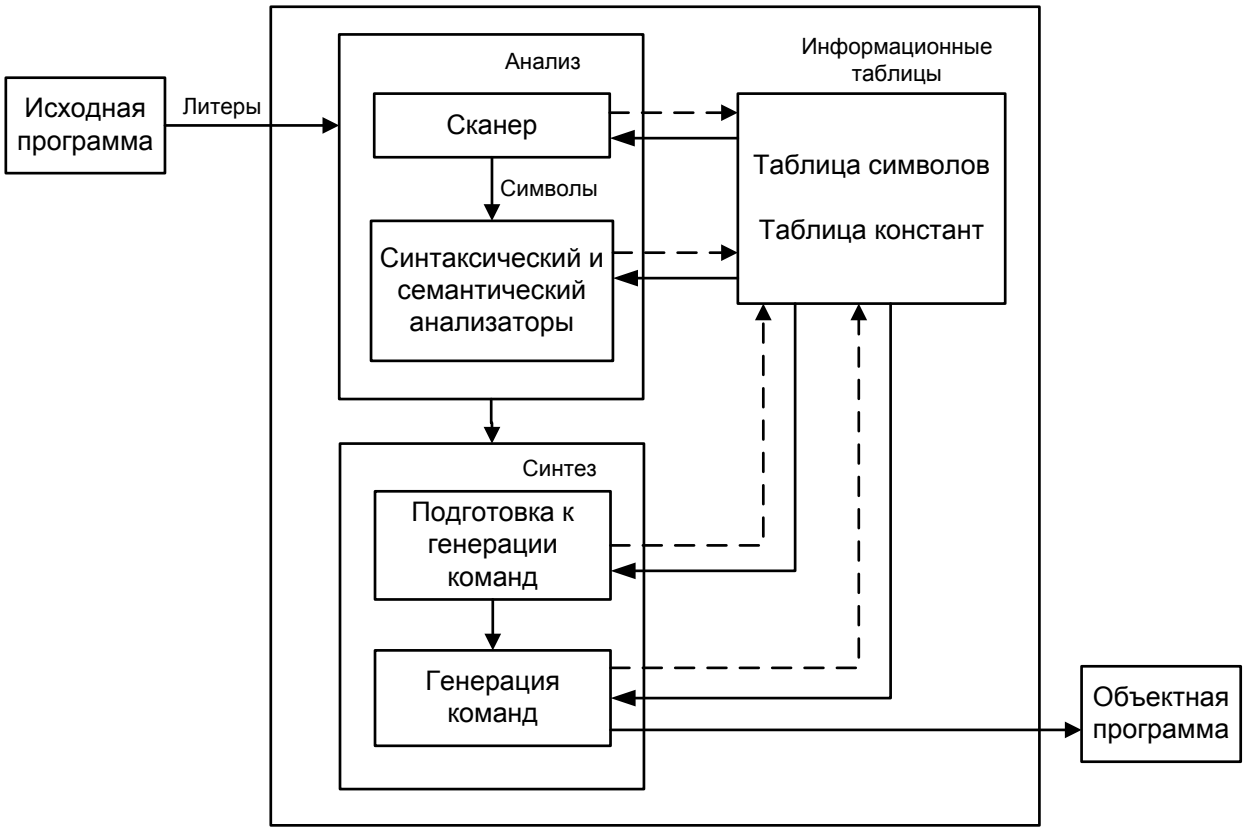


Рисунок 2.1 – Схема процесса компиляции

Для обеспечения построения компиляторов язык высокого уровня обычно описывается в терминах некоторой грамматики. Эта грамматика определяет форму, то есть синтаксис допустимых предложений языка. Проблема компиляции может быть сформулирована как проблема поиска соответствия написанных программистом предложений структурам, определенным грамматикой и генерацией соответствующего кода для каждого предложения. Наиболее существенная часть любого процесса компиляции – это введение в исходной программе элементарных составляющих, идентификаторов, ограничителей, операторов, чисел и т.д., которые называются *лексемами*. Предложения исходной программы удобнее представлять в виде последовательности лексем. Просмотр исходного текста, распознавание и классификация различных лексем называется *лексическим анализом*.

Как только лексемы выделены, каждое предложение программы может быть распознано как некоторые конструкции языка. Этот процесс называется *синтаксическим анализом*. Здесь лексемы, полученные на стадии лексического анализа, используются для идентификации более крупных программных структур, выражений, команд и т.д.

Последним шагом базовой схемы процесса трансляции является генерация объектного кода.

В данном лабораторном практикуме рассмотрены операции, необходимые для компиляции программ типичного языка высокого уровня. Лабораторная работа выполняется с помощью программной модели. Инструкции по использованию модели находятся в файле COMPILE.chm. В качестве примера используется программа на языке Паскаль, изображенная на рисунке 2.2.

```
1       PROGRAM stats;
2       VAR
3            sum, sumsq, i, value, mean, variance: INTEGER;
4       BEGIN
5         sum := 0;
6         sumsq := 0;
7         FOR i:=1 TO 100 DO
8                 BEGIN
9                       READ (value);
10                      sum := sum + value;
11                      sumsq := sumsq + value * value;
12                END;
13        mean := sum DIV 100;
14        variance := sumsq DIV 100 – mean * mean;
15        WRITE (mean, variance);
16      END.
```

Рисунок 2.2 - Пример программы на языке Паскаль

2.1 Грамматики

Грамматика языка программирования является формальным описанием его синтаксиса или формы, в которой записаны отдельные предложения программы или вся программа.

Описание языков программирования во многом опирается на теорию формальных языков. Эта теория является фундаментом для организации синтаксического анализа и перевода.

Существует два основных, тесно связанных способа определения языков:

– механизм порождения или генератор;

– механизм распознавания или распознаватель.

Первый обычно используется для описания языков, а второй для их реализации. Оба способа позволяют описать языки конечным образом, несмотря на бесконечное число порождаемых ими цепочек.

Неформально язык **L** - это множество цепочек конечной длины в алфавите **V**. Механизм порождения позволяет описать языки с помощью системы правил, называемой грамматикой. Цепочки (предложения) языка строятся в

соответствии с этими правилами. Достоинство определения языка с помощью грамматик в том, что операции, производимые в ходе синтаксического анализа и перевода, можно делать проще, если воспользоваться структурой, предписываемой цепочкам с помощью этих грамматик.

Синтаксис - *совокупность правил некоторого языка, определяющих формирование его элементов. Иначе говоря, это совокупность правил образования семантически значимых последовательностей символов в данном языке. Синтаксис задается с помощью правил, которые описывают понятия некоторого языка*. Примерами понятий являются: переменная, выражение, оператор, процедура. Последовательность понятий и их допустимое использование в правилах определяет синтаксически правильные структуры, образующие программы.

Одной из наиболее простых и широко используемых форм записи грамматик является нормальная форма Бэкуса-Наура (БНФ). Метаязык, предложенный Бэкусом и Науром, впервые использовался для описания синтаксиса реального языка программирования Алгол 60. Наряду с обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов. Это сделало описание языка нагляднее и позволило в дальнейшем широко использовать данную нотацию для описания реальных языков программирования. Были использованы следующие обозначения:

– символ "::=" отделяет левую часть правила от правой (символ "::=" можно читать как “является по определению”, иногда вместо "::=" используется символ "→");

– нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки "<" и ">" (нетерминалы являются именами конструкций, определенными внутри грамматики);

– терминалы - это символы, используемые в описываемом языке;

– каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты "|".

На рисунке 2.3 изображена одна из возможных грамматик БНФ для очень узкого подмножества языка Паскаль.

1	<prog>	::=	PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END.
2	<prog-name>	::=	**id**
3	<dec-list>	::=	<dec> | <dec-list> ; <dec>
4	<dec>	::=	<id-list> : <type>
5	<type>	::=	INTEGER
6	<id-list>	::=	**id** | <id-list> , **id**
7	<stmt-list>	::=	<stmt>
8	<stmt>	::=	<assign>
9	<assign>	::=	id :=<exp>
10	<exp>	::=	<term> | <exp> + <term> | <exp> – <term>
11	<term>	::=	<factor> | <term> *<factor> | <term> DIV <factor>
12	<factor>	::=	**id** | **int** | (<exp>)
13	<read>	::=	READ (<id-list>)
14	<write>	::=	WRITE (<id-list>)
15	<for>	::=	FOR <index-exp> DO <body>
16	<index-exp>	::=	id := <exp> TO <exp>
17	<body>	::=	<stmt> | BEGIN <stmt-list> END

Рисунок 2.3 - Упрощенная грамматика языка Паскаль

Результат анализа исходного предложения в терминах грамматических конструкций удобно представлять в виде дерева, которое называется деревом грамматического разбора или синтаксическим деревом.

Правило вывода 9 на рисунке 2.3 дает определение синтаксиса предложения присваивания: <assign> ::= id := <exp>

Это означает, что конструкция <assign> состоит из лексемы id, за которой следует лексема :=, за которой идет конструкция <exp>. Правило 10 дает определение конструкции <exp> как состоящей из любой последовательности конструкций <term>, соединенных операторами «плюс» или «минус». Аналогично правило 11 определяет конструкцию <term> как последовательность конструкций <factor>, разделенными знаками * или DIV. В соответствии с правилом 12 конструкция <factor> может состоять из идентификатора **id** или целого **int**, которое также распознается сканером, или из конструкции <exp>, заключенной в круглые скобки.

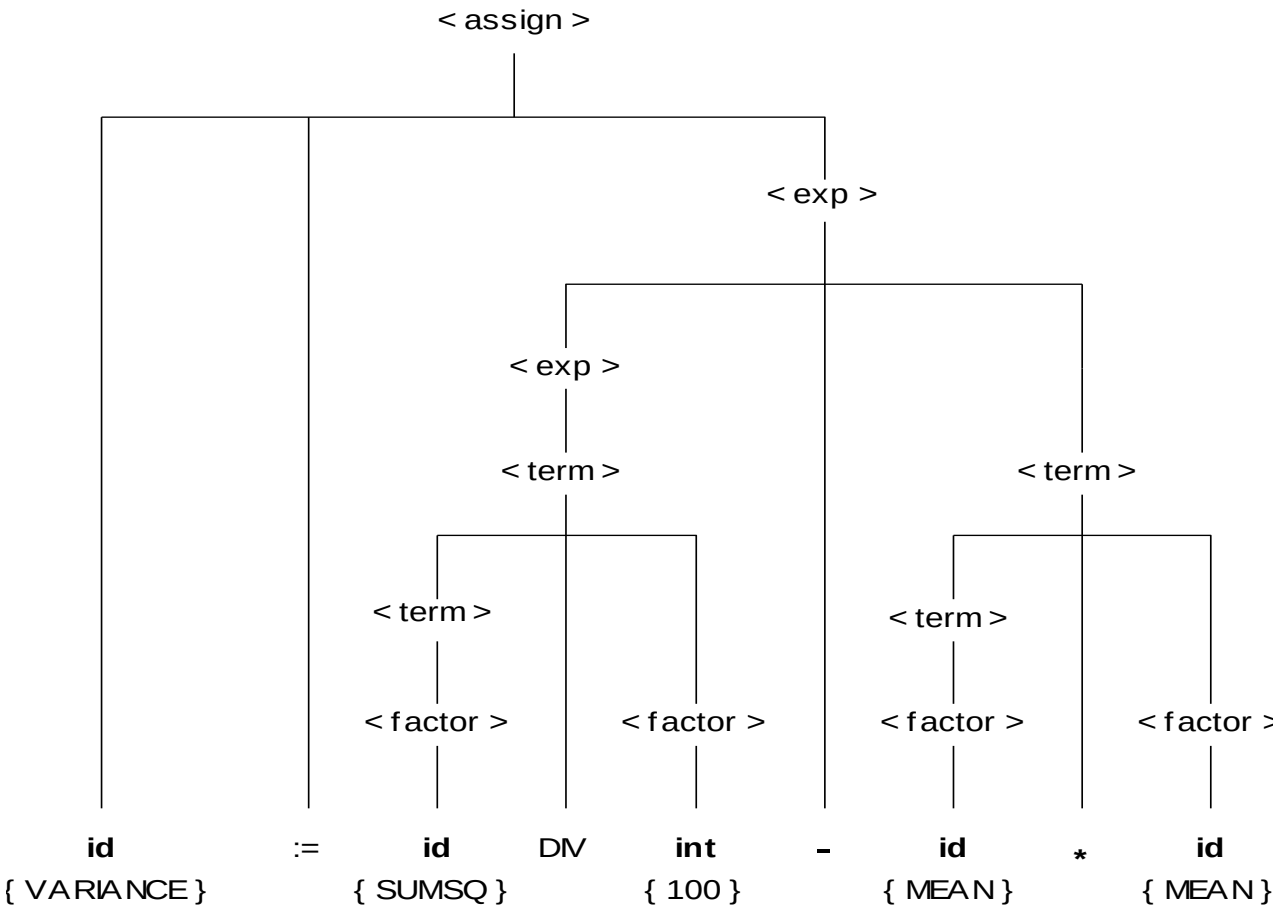


Рисунок 2.4 - Дерево грамматического разбора для предложения 14 программы на рисунке 2.2

На рисунке 2.4 изображено дерево грамматического разбора для предложения 14 на рисунке 2.2, основанное на только что рассмотренных правилах вывода. В соответствии с этим правилом умножение и деление производятся перед сложением и вычитанием. Прежде всего должны вычисляться термы SUMSQ DIV 100 и MEAN*MEAN, поскольку эти промежуточные результаты являются операндами (левым и правым поддеревом) операции вычитания. Операции умножения и деления имеют более высокий ранг, чем операции сложения и вычитания. **Такое ранжирование является следствием того, как записаны правила 10-12.** Дерево на рисунке 2.4 представляет собой единственно возможный результат анализа предложения 14 в терминах грамматики на рисунке 2.3. Для некоторых грамматик подобной единственности может не

существовать. Если для одного и того же предложения можно построить несколько различных деревьев грамматического разбора, то соответствующая грамматика называется неоднозначной. При разработке компиляторов предпочитают пользоваться однозначными грамматиками.

2.2 Лексический анализ

Лексический анализ включает в себя просмотр компилируемой программы и распознавание лексем, составляющих предложения исходного текста. Лексические анализаторы строятся таким образом, чтобы они могли распознавать ключевые слова, операторы и идентификаторы так же, как целые числа, числа с плавающей точкой, строки символов и другие аналогичные конструкции, встречающиеся в исходной программе. Точный перечень лексем, которые необходимо распознать, зависит от языка программирования, на который рассчитан компилятор, и от грамматики, используемой для описания этого языка.

Лексема	Код
PROGRAM	1
VAR	2
BEGIN	3
END	4
END.	5
INTEGER	6
FOR	7
READ	8
WRITE	9
TO	10
DO	11
;	12
:	13
.	14
:=	15
+	16
-	17
*	18
DIV	19
(	20
)	21
id	22

Рисунок 2.5 – Коды лексем для грамматики на рисунке 2.3

Лексический анализатор преобразует исходную программу в последовательность символов. При этом идентификаторы и константы произвольной длины заменяются символами фиксированной длины. Слова языка также заменяются каким-нибудь стандартным представлением. Например, слова языка могут заменяться целыми числами, идентификаторы – буквой I и следующим за ней целым числом, константы – буквой C и целым числом и т.д. Эти целые числа не могут быть произвольной

длины и поэтому число идентификаторов и констант ограниченно какой-либо, обычно очень большой, величиной. Пользователю практически невозможно увидеть данное ограничение.

Для повышения эффективности последующих действий каждая лексема обычно определяется кодом, например целым числом, а не в виде строки символов переменной длины. Если распознанная лексема является ключевым словом или оператором, такая схема кодирования дает всю необходимую информацию.

Коды, создаваемые для слов языка, не должны зависеть от программы, например: ключевое слово **begin** должно всегда иметь какой-либо конкретный код. В случае же идентификатора дополнительно необходимо конкретное имя распознаваемого идентификатора.

Получение кодов идентификаторов производиться последовательно, начиная с начала по тексту программы. Например, первый идентификатор будет иметь код I1, а второй - I2 и т.д. Каждый экземпляр определенного идентификатора (на любом уровне) заменяется одним и тем же кодом. Из этого следует необходимость создания таблицы идентификаторов, где будут храниться соответствия кодов и самих идентификаторов.

В таблице 2.1 показан результат обработки сканером программы, приведенной на рисунке 2.2 с использованием кодировки лексем, представленной на рисунке 2.5. Для лексем типа 22 (идентификаторы) и типа 23 (целые числа) должны быть определены спецификаторы для указания на таблицу символов и таблицу констант.

Таблица 2.1 – Результат лексического разбора программы на рисунке 2.2

Тип лексемы	Индекс	Символ	Спецификатор	Тип лексемы	Индекс	Символ	Спецификатор
T	1	PROGRAM		I	22	SUM	#SUM
I	22	STATS	#ST	T	15	:=	
T	2	VAR		I	22	SUM	#SUM

I	22	SUM	#SUM	T	16	+	
T	14	,		I	22	VALUE	#VALUE
I	22	SUMSQ	#SUMSQ	T	12	;	
T	14	,		I	22	SUMSQ	#SUMSQ
I	22	I	#I	T	15	:=	
T	14	,		I	22	SUMSQ	#SUMSQ
I	22	VALUE	#VALUE	T	16	+	
T	14	,		I	22	VALUE	#VALUE
I	22	MEAN	#MEAN	T	18	*	
T	14	,		I	22	VALUE	#VALUE
I	22	VARIANCE	#VARI	T	4	END	
T	13	:		T	12	;	
T	6	INTEGER		I	22	MEAN	#MEAN
T	3	BEGIN		T	15	:=	
I	22	SUM	#SUM	I	22	SUM	#SUM
T	15	:=		T	19	DIV	
C	23	0	^0	C	23	100	
T	12	;		T	12	;	
I	22	SUMSQ	#SUMSQ	I	22	VARIANCE	#VARI
T	15	:=		T	15	:=	
C	23	0	^0	I	22	SUMSQ	
T	12	;		T	19	DIV	
T	7	FOR		C	23	100	
I	22	I	#I	T	17	-	
T	15	:=		I	22	MEAN	#MEAN
C	23	1	^1	T	18	*	
T	10	TO		I	22	MEAN	#MEAN
C	23	100	^100	T	12	;	
T	11	DO		T	9	WRITE	
T	3	BEGIN		T	20	(	
T	8	READ		I	22	MEAN	#MEAN
T	20	(		T	15	,	
I	22	VALUE	#VALUE	I	22	VARIANCE	#VARI
T	21	)		T	21	)	
T	12	;		T	5	END.	

2.3 Синтаксический анализ

Синтаксический разбор - это основная часть компилятора на этапе анализа. Она выполняет выделение синтаксических конструкций в тексте исходной программы, обработанной лексическим анализатором. На этой же фазе компиляции проверяется синтаксическая правильность программы.

Синтаксический анализ используется для доказательства того, принадлежит ли анализируемая входная цепочка множеству цепочек, порождаемых грамматикой данного языка. Выполнение синтаксического разбора осуществляется распознавателями, являющимися автоматами. Поэтому данный процесс также называется распознаванием входной цепочки. Результатом работы распознавателя грамматики входного языка является последовательность правил грамматики, примененных для построения входной цепочки.

Выделяются два основных метода синтаксического разбора:

- восходящий разбор;
- нисходящий разбор.

Кроме этого можно использовать комбинированный разбор, сочетающий особенности двух предыдущих.

При **восходящем разборе** дерево начинает строиться от терминальных листьев путем подстановки правил, применимых к входной цепочке в произвольном порядке. На следующем шаге новые узлы полученных поддеревьев используются как листья во вновь применяемых правилах. Процесс построения дерева разбора завершается, когда все символы входной цепочки будут являться листьями дерева, корнем которого окажется начальный нетерминал. Если, в результате полного перебора всех возможных правил, невозможно построить требуемое дерево разбора, то рассматриваемая входная цепочка не принадлежит данному языку.

При использовании восходящего метода на множестве терминальных и нетерминальных символов необходимо ввести три отношения, называемые "отношения предшествования" и обозначаемые как $\lessdot$, $\gtrdot$ и $\doteq$.

Отношения $\lessdot$, $\gtrdot$ и $\doteq$ не похожи на обычные арифметические отношения $<$, $>$, $=$; отношение $\doteq$ не является отношением эквивалентности, а отношения $\lessdot$ и $\gtrdot$ не обязательно транзитивны. То есть для отношения предшествования не выполняются некоторые правила, привычные для отношения арифметического порядка. Например, ; $\gtrdot$ END но END $\gtrdot$;

Отношения предшествования удобно занести в матрицу, в которой строки и столбцы помечены терминалами грамматики.

Отношение $\doteq$ означает, что обе лексемы имеют одинаковый уровень предшествования и должны рассматриваться грамматическим процессором в качестве одной конструкции языка. Если для пар отношения предшествования не существует, это означает, что они не могут находиться рядом ни в каком грамматически правильном предложении. Если подобная комбинация лексем встретится в процессе грамматического разбора, то она должна рассматриваться как синтаксическая ошибка.

Существуют алгоритмы автоматического построения матриц предшествования на основе формального описания грамматики. Одна из таких матриц показана на рисунке 2.6.

Для применимости метода операторного предшествования необходимо, чтобы отношения предшествования были заданы однозначно.

Например, не должно быть одновременно отношений (; $\lessdot$ BEGIN) и (; $\gtrdot$ BEGIN). Это требование выполняется для используемой грамматики, однако некоторые из отношений грамматики Паскаля не являются однозначными и метод оперативного предшествования к ним не применим.

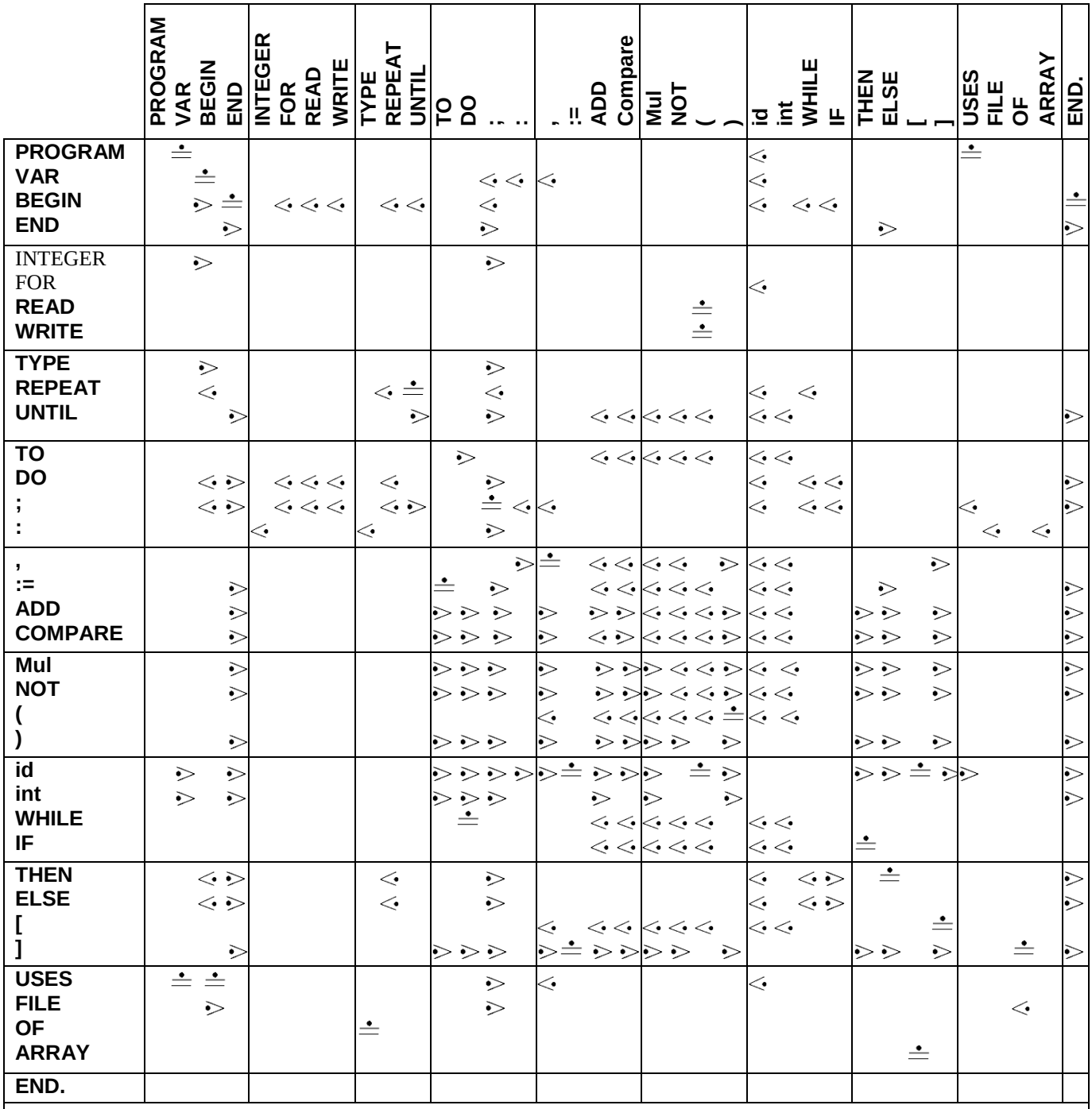
На рисунке 2.7 изображен пошаговый процесс грамматического разбора предложения присваивания в строке 14.

Процесс сканирования слева направо продолжается на каждом шаге грамматического разбора лишь до тех пор, пока не определился очередной фрагмент предложения для грамматического распознавания, то есть первый фрагмент, ограниченный отношениями $\lessdot$ и $\gtrdot$. Как только подобный фраг-

мент выделен, он интерпретируется как некоторый очередной нетерминальный символ в соответствии с каким-либо правилом грамматики.

Этот процесс продолжается до тех пор, пока предложение не будет распознано целиком. Следует обратить внимание на то, что каждый фрагмент дерева грамматического разбора строится, начиная с оконечных узлов вверх, в сторону корня дерева. Отсюда и возник термин «восходящий разбор».

Для метода операторного предшествования имена нетерминальных символов несущественны. Таким образом, вся информация о грамматике и синтаксических правилах языка содержится в матрице операторного предшествования.



Types (Тип) один из стандартных типов Паскаля (BYTE, INTEGER, SHORTINT, WORD, BOOLEAN, LONGINT, STRING, CHAR, POINTER, REAL, SINGLE, DOUBLE, EXTENDED, COMP, TEXT).

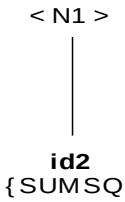
Add (Операция типа Сложение) одна из операций Паскаля равная по приоритету операции сложения (+, -, OR).

Mul (Операция типа Умножение) одна из операций Паскаля равная по приоритету операции умножения (*, /, DIV, MOD, AND, XOR, SHR, SHL).

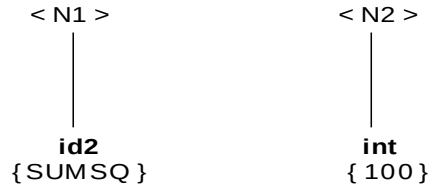
Compare (Операция типа Сравнение) одна из операций Паскаля равная по приоритету операции сравнения (<>, >=, <=, =, >, <).

Рисунок 2.6 - Матрица операторного предшествования

1. … id1 := id2 DIV
   < = < >

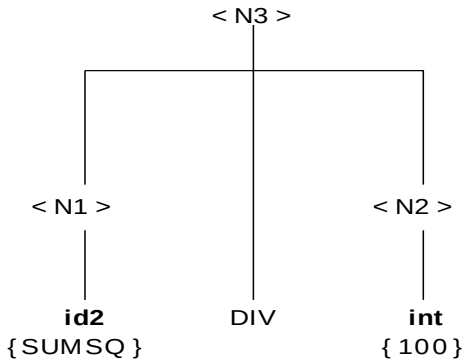


2. … id1 := < N1 > DIV int -
   < = < < >

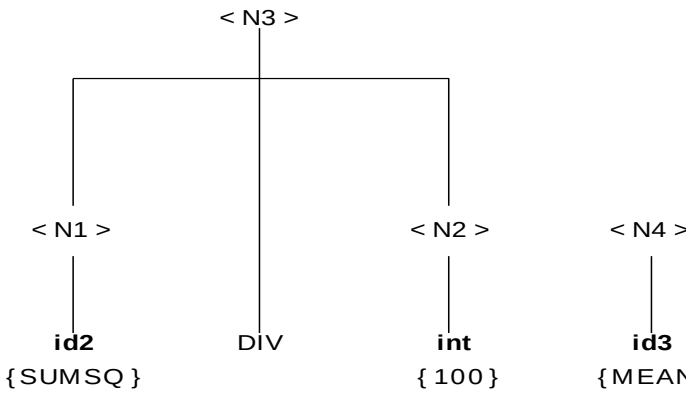


3. … id1 := < N1 > DIV < N2 > -
   < = < >

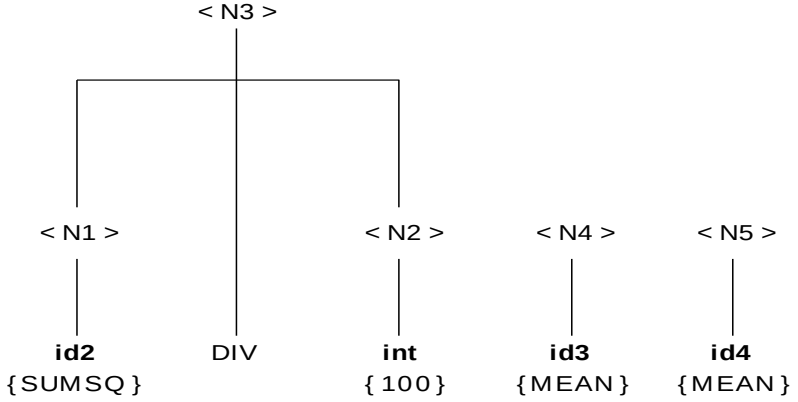
4. … id1 := < N3 > - id3 *
   < = < < >



5. … id1 := < N3 > - < N4 > * id4 ;
   < = < < < >

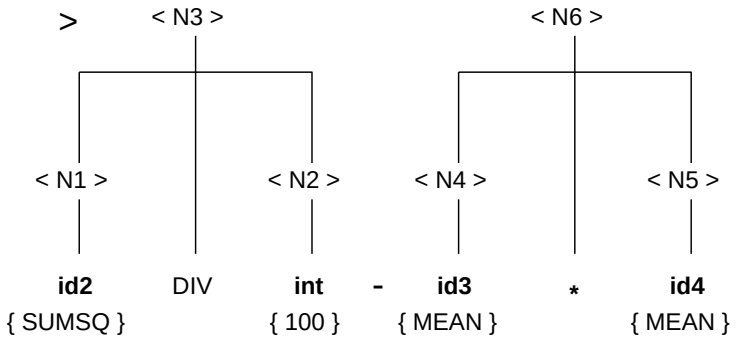


6. … id1 := < N3 > - < N4 > * <N5> ;
   < = < < >



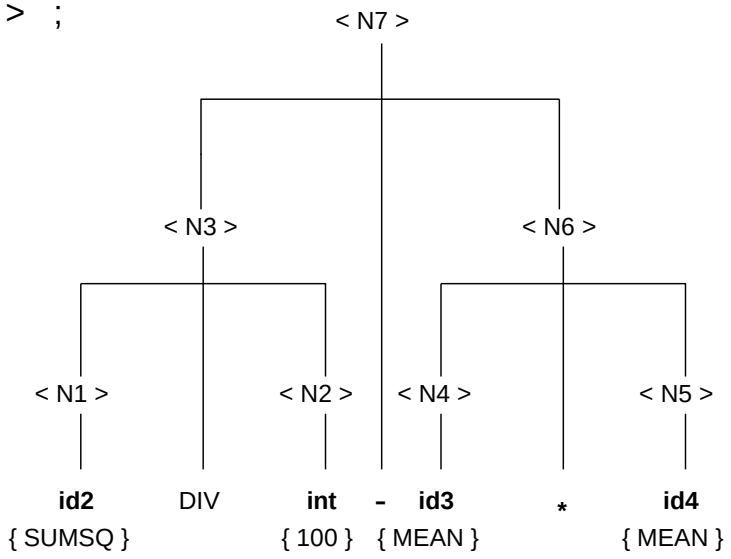
7. ... id1 := < N3 > - < N6 > ;

< = < >



8. ... id1 := < N7 > ;

< = >



9. ... < N8 > ;

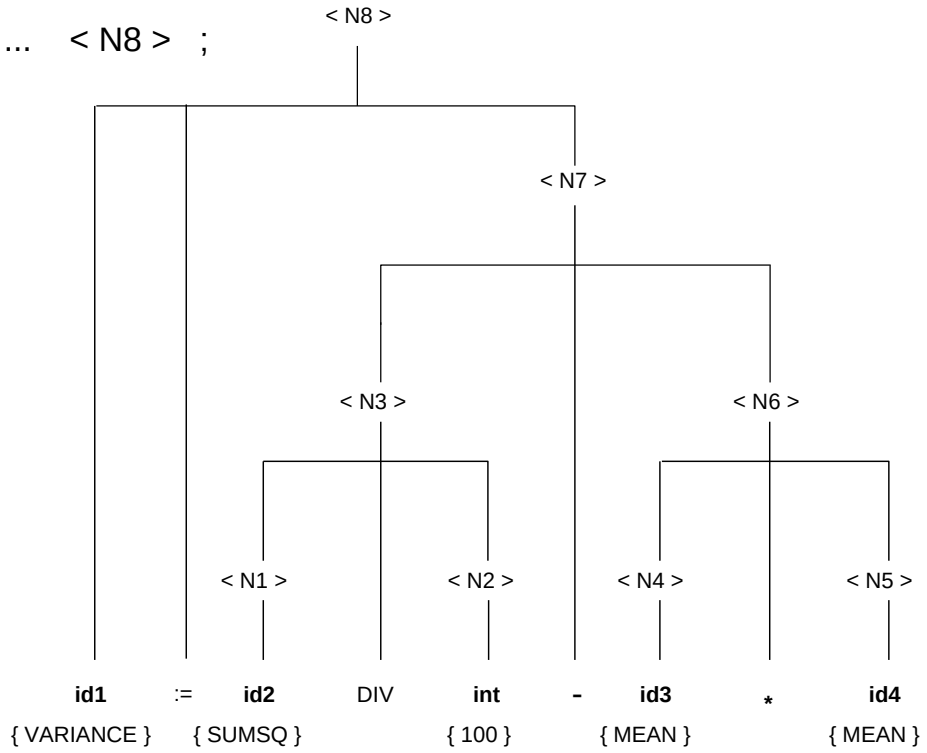
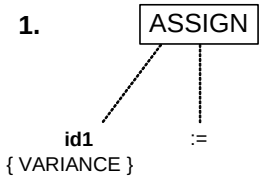


Рисунок 2.7 - Грамматический разбор предложения присваивания методом операторного предшествования

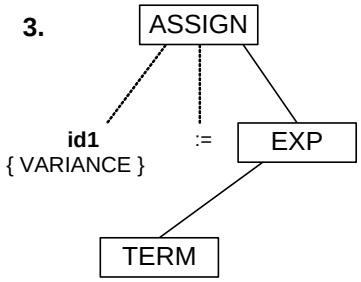
Другой метод грамматического разбора - **нисходящий метод**, называемый рекурсивным спуском. Процессор грамматического разбора, основанный на этом методе, состоит из отдельных процедур для каждого нетерминального символа, определенного в грамматике. Каждая такая процедура ищет во входном потоке подстроку, начинающуюся с текущей лексемы, которая может быть интерпретирована как нетерминальный символ, связанный с данной процедурой. В процессе своей работы она может вызывать другие подобные процедуры или даже рекурсивно саму себя для поиска других нетерминальных символов. Если эта процедура находит соответствующий нетерминальный символ, то она заканчивает свою работу, передает в вызвавшую ее программу признак успешного завершения и устанавливает указатель текущей лексемы на первую лексему после распознанной подстроки. Если же процедура не может найти подстроку, которая могла бы быть интерпретирована как требуемый нетерминальный символ, она заканчивается с признаком ошибки или вызывает процедуру выдачи диагностического сообщения и процедуру восстановления.

На рисунке 2.8 представлен разбор методом рекурсивного спуска оператора присваивания в строке 14 на рисунке 2.2.

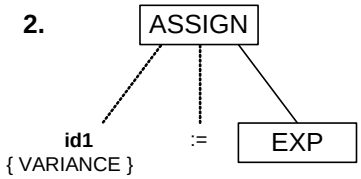
1.


ASSIGN
id1
{ VARIANCE }
:=

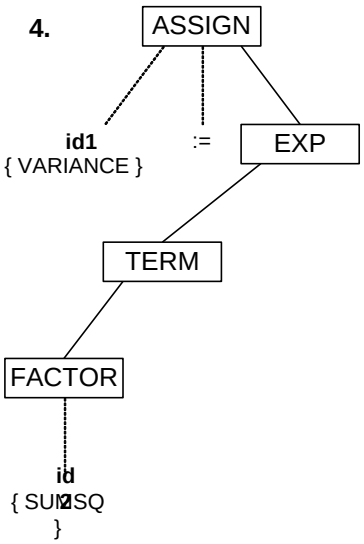
3.


ASSIGN
id1
{ VARIANCE }
:=
EXP
TERM

2.


ASSIGN
id1
{ VARIANCE }
:=
EXP

4.


ASSIGN
id1
{ VARIANCE }
:=
EXP
TERM
FACTOR
id
2
}

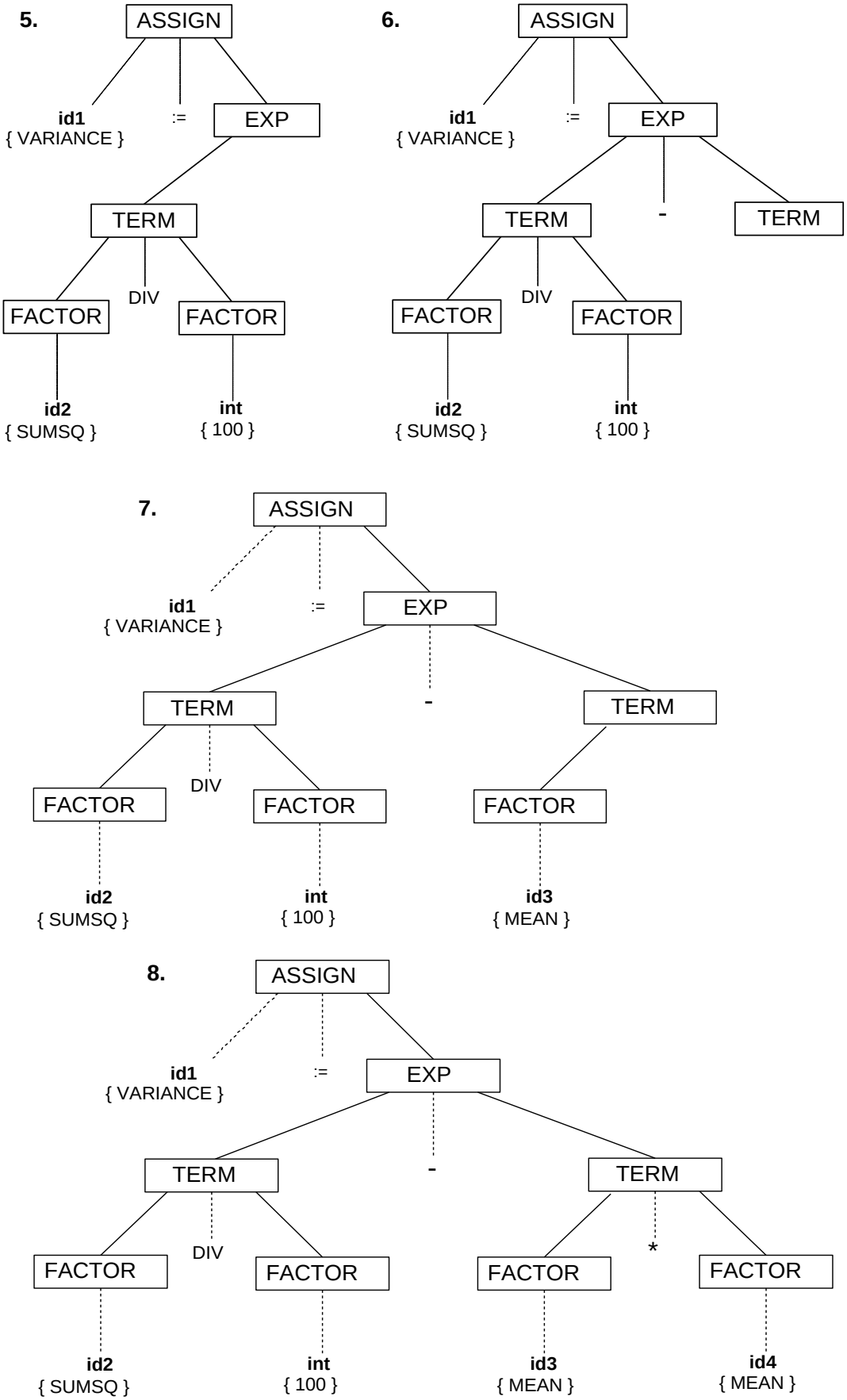


Рисунок 2.8 - Грамматический разбор предложения присваивания методом рекурсивного спуска

2.4 Генерация кода

Программы генерации кода предназначены для использования с грамматикой на рисунке 2.3. Ни один из методов грамматического разбора не распознает в точности те конструкции, которые описаны грамматикой. Метод операторного предшествования игнорирует некоторые нетерминальные символы, а метод рекурсивного спуска вынужден использовать несколько модифицированную грамматику

Основная работа при грамматическом разборе предложения присваивания состоит в анализе нетерминального символа <exp> в правой части оператора присваивания. В процессе грамматического разбора идентификатор SUMSQ распознается сначала как <factor>, потом как <term>. Далее распознается целое число 100 как <factor>, потом как <term>; затем фрагмент SUMSQ DIV 100 распознается как <term> и т.д. Порядок распознавания фрагментов этого предложения совпадает с порядком, в котором должны выполняться соответствующие вычисления; сначала вычисляются подвыражения SUMSQ DIV 100 и MEAN*MEAN, а затем второй результат вычитается из первого. Как только очередной фрагмент предложения распознан, вызывается соответствующая программа генерации кода. Например, код, соответствующий правилу $\langle term \rangle_1 := \langle term \rangle_2 * \langle factor \rangle$ будет генерироваться следующим образом.

```
MOV     AX,SUMSQ
MOV     DL,100
IDIV    DL
MOV     T1,AL
MOV     AL,MEAN
MOV     DL,MEAN
IMUL    DL
MOV     DL,T1
XOR     DH
SUB     DX,AX
MOV     VARIANCE,DX
```

Программы генерации кода выполняют арифметические операции с использованием регистра AX, и нужно заведомо сгенерировать в объектном коде операцию MUL. Результат этого умножения $\langle term \rangle_1$ после операции MUL со-

хранится в регистре AX. Если либо <term>$_2$, либо <factor> уже находятся в регистре AX(AL) (после выполнения предыдущих операций), нужно только сгенерировать инструкцию MUL. Иначе необходимо сгенерировать также инструкцию MOV, предшествующую инструкции MUL. В этом случае также надо предварительно сохранить значение регистра AX(AL), если это необходимо. Очевидно, что необходимо отслеживать значение, помещенное в регистр AX(AL), после каждого фрагмента генерируемого объектного кода. Один из вариантов генерации объектного кода приведен на рисунке 2.9.

Процесс генерации кода является машинно-зависимым, поскольку необходимо знать систему команд компьютера, для которого этот код генерируется. Существуют, однако, более сложные проблемы, вытекающие из машинной зависимости, которые возникают при распределении регистров и перестановки машинных инструкций для повышения эффективности. Такого рода оптимизация кода обычно осуществляется с использованием промежуточной формы представления компилируемой программы. В этой промежуточной форме синтаксис и семантика исходных предложений программы уже полностью проанализированы, но трансляция в машинные коды еще не осуществлена. Анализировать и модернизировать программу, представленную в такой промежуточной форме для оптимизации кода, существенно легче, чем для исходной программы на языке высокого уровня или для этой программы, представленной в машинных кодах.

Промежуточные коды можно проектировать на различных уровнях. Иногда промежуточный код получают, просто разбивая сложные структуры исходного языка на более удобные для обращения элементы. Однако чаще в качестве промежуточного кода используют какой-либо обобщенный машинный код. Существует несколько видов таких обобщенных кодов.

Тетрады представляют собой запись операций в форме из четырех составляющих: операция, два операнда и результат операции.

<операция> (<операнд1>,< операнд2>,<результат>)

Тетрады – это линейная последовательность команд, поэтому для них несложно написать алгоритм, который будет преобразовывать последовательность тетрад в последовательность команд результирующей программы либо в последовательность команд Ассемблера. Тетрады не зависят от архитектуры вычислительной системы, на которую ориентированы результирующая программа. Поэтому они представляют собой машинно-независимую форму внутреннего представления программы. Такой код часто называют трехадресным, т.е. два адреса для операндов (кроме случая унарных операций) и один – для результата. Например, для выражения (-a+b)*(c+d) можно представить тетрады следующим образом:

-a = 1

1+b=2

c+d=3

2*3=4

Триады представляют собой запись операций в форме из трех составляющих: операция и два операнда.

<операция> (<операнд1>,< операнд2>)

Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому триады при записи последовательно нумеруют для удобства указания ссылок одних триад на другие.

Так же, как и в случае с тетрадами, алгоритм преобразования последовательности триад в команды достаточно прост, но здесь требуется также и алгоритм, отвечающий за распределение памяти, необходимый для хранения промежуточных результатов вычислений, так как временные переменные для этой цели не используются.

Триады требуют меньше памяти для своего представления, чем тетрады, кроме того, они явно отражают взаимосвязь операций между собой, что делает

их применение удобным. Триады ближе к двухадресным машинам, чем тетрады. Выражение

a+b+c*d

можно представить в виде тетрад:

a+b=1

c*d=2

1+2=3

Это же выражение можно представить в виде триад:

a+b

c*d

1+2

Видно, что запись в виде триад является более компактной, но, если присутствует фаза оптимизации, их применение затруднительно. Наилучшее решение этой проблемы – косвенные триады, т.е. операнд, ссылавшийся на ранее вычисленную триаду, теперь ссылается на элемент таблицы указателей на триады, а не на саму триаду.

Как триады, так и тетрады можно распространить не только на арифметические выражения, но и на другие конструкции языка. Например, языковую конструкцию if – then – else можно рассматривать как выражение с тремя операндами, которому потребуются четыре адреса как тетраде и три – как триаде.

2.5 Оптимизация

Оптимизацией называется обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе в целях получения более эффективного объектного кода. Обычно выделяют машинно-зависимую и машинно-независимую оптимизацию. Под машинно-независимой оптимизацией понимается преобразование исходной программы в ее внутреннем представлении, что означает полную независимость от выходного языка, в отличие от машинно-зависимой, выполняемой на уровне объектной программы.

Среди машинно-независимых методов можно выделить самые основные:

- свертка, т.е. выполнение операций, операнды которых известны во время компиляции;
- исключение лишних операций за счет однократного программирования общих подвыражений;
- вынесение из цикла операций, операнды которых не изменяются внутри цикла.

Линейным участком является выполняемая по порядку последовательность операций с одним входом и одним выходом (первая и последняя операции соответственно). Например, последовательность операций представленных ниже образуют линейный участок:

```
I:=1+1;
I:=3;
B:=7+I;
```

Внутри линейного участка обычно проводят две оптимизации: свертку и устранение лишних операций.

Свертка – *это выполнение во время компиляции операций исходной программы, для которых значения операндов уже известны, и поэтому нет нужды их выполнять во время счета.* Например, внутреннее представление в виде триад линейного участка программы, представленного выше, изображено в следующем виде:

```
(1) + 1,1
(2) := I,(1)
(3) := I,3
(4) + 7,(3)
(5) := B,(4)
```

Видно, что первую триаду можно вычислить во время компиляции и заменить результирующей константой. Менее очевидно, что четвертую триаду также можно вычислить, так как к моменту ее обработки известно, что I равно 3. Полученный после свертки результат

(1) := I,2

(2) := I,3

(3) := B,10

Свертка, главным образом, применяется к арифметическим операциям, так как они наиболее часто встречаются в исходной программе. Кроме того, она применяется к операторам преобразования типа. Причем проблема упрощается, если эти преобразования заданы явно, а не подразумеваются по умолчанию.

Процесс свертки операторов, имеющих в качестве операндов константы, понятен и сводится к внутреннему их вычислению. Свертка операторов, значения которых могут быть определены в результате некоторого анализа, несколько сложнее. Обычно свертку осуществляют только в пределах линейного участка при помощи таблицы Т, вначале пустой, содержащей пары (К, А), где А – простая переменная для которой известно текущее значение К. Кроме того, каждая свертываемая триада заменяется новой триадой (С, К, 0), где С (константа) – новый оператор, для которого не надо генерировать команды, К – результирующее значение свернутой триады.

Алгоритм свертки последовательно просматривает триады линейного участка. Пример работы данного алгоритма приведен в таблице 2.2.

Таблица 2.2 - Последовательность шагов алгоритма свертки

	Шаг 1	*Шаг 2*	*Шаг 3*	*Шаг 4*	*Шаг 5*
1	+ 1,1	C 2	C 2	C 2	C 2
2	:= I,(1)	:= I,(1)	:= I,2	:= I,2	:= I,(1)
3	:= I,3	:= I,3	:= I,3	:= I,3	:= I,3
4	+ 7,(3)	+ 7,(3)	+ 7,(3)	C 10	C 10
5	:= B,(4)	:= B,(4)	:= B,(4)	:= B,(4)	:= B,10
T			(I,2)	(I,3)	(I,3)
T					(B,10)

С точки зрения работы компилятора, процесс свертки является дополнительным проходом по внутреннему представлению исходной программы, представленной триадами. Но свертку можно проводить и с тетрадами. Кроме

того, можно оптимизировать программу в семантических программах во время получения внутреннего представления, и при этом отпадает необходимость в дополнительном проходе. Например, для выражения A:=B+C при восходящем грамматическом разборе программа должна выполнить только одну дополнительную проверку семантик B и C. Если они константы или их значения известны, то программа их складывает и связывает результат с A. В данном случае можно использовать таблицу переменных с известными значениями, которая должна сбрасываться в местах генерации команд передачи управления.

Исключение лишних операций - *i-я операция линейного участка считается лишней, если существует более ранняя идентичная j-я операция и никакая переменная, от которой зависит эта операция, не изменяется третьей операцией лежащей между I-й и j-й операциями.* Например, на линейном участке

D:=D+C*B;

A:=D+C*B;

можно выделить лишние операции. Если представить линейный участок в виде триад, как это представлено ниже, то видно, что операция C*B во второй раз лишняя:

(1) * C,B

(2) + D,(1)

(3) := D,(2)

(4) * C,B

(5) + D,(4)

(6) := A,(5)

Лишней триада (4) является из-за того, что ни C ни B не изменяются после триады (1). В отличии от этого триада (5) лишней не является, так как после первого сложения D с C*B триада (3) изменяет значение D.

Алгоритм исключения лишних операций просматривает операции в порядке их появления. И если i-я триада лишняя, так как уже имеется идентичная ей j-я триада, то она заменяется триадой (SAME, j, 0), где операция SAME ничего не делает и не порождает никаких команд при генерации.

Для слежения за внутренней зависимостью переменных и триад можно поставить им в соответствие числа зависимостей (dependency numbers). При этом используются следующие правила:

- для переменной A число зависимости dep(A) выбирается равным нулю, так как ее значение не зависит ни от одной триады;
- после обработки i-й триады, где переменной A присваивается какое-либо значение число зависимости становиться равным i, так как ее новое значение зависит от i-й триады;
- при обработке i-й триады ее число зависимостей dep(i) становиться равным максимальному из чисел зависимостей ее операндов + 1.

Числа зависимостей используются следующим образом: если i-я триада идентична j-й триаде (j<i), то i-я триада является лишней тогда и только тогда, когда dep(i)=dep(j).

Таблица 2.3 – Пример исключения лишних операций

Обрабатываемая триада I	Dep (переменная) A B C D	Dep(i)	Триада, полученная в результате
(1) * C,B	0 0 0 0	1	(1) * C,B
(2) + D,(1)	0 0 0 0	2	(2) + D,(1)
(3) :=D,(2)	0 0 0 0	3	(3) :=D,(2)
(4) * C,B	0 0 0 3	1	(4) SAME 1
(5) + D,(4)	0 0 0 3	4	(5) + D,(4)
(6) :=A,(5)	0 0 0 3	5	(6) :=A,(5)
	6 0 0 3		

Вынесение инвариантных операций за тело цикла. *Инвариантными называются такие операции, при которых ни один из операндов не изменяется внутри цикла.*

В общем случае данная оптимизация выполняется двойным просмотром тела цикла с анализом операндов каждой из операций внутри цикла. В случае если они инвариантны, операция выносится назад за тело цикла, а внутри цикла выбрасывается. Для проверки инвариантности переменных используют специальную таблицу инвариантности.

Во время первого прохода цикла заполняется таблица инвариантных переменных, во время второго – выполняется собственно вынесение операций.

2.6 Выполнение лабораторной работы «Компиляторы»

Лабораторная работа выполняется с помощью программной модели. Целью данной лабораторной работы является закрепление теоретических знаний о процессе компиляции, полученных при изучении дисциплины "Системное программное обеспечение".

ВНИМАНИЕ!!!

1. Перед выполнением лабораторной работы необходимо прочитать теорию и пройти тест.

2. Перед выполнением какого-либо этапа работы необходимо прочитать особенности программной модели.

3. Нельзя работать методом подбора - все нажатия на кнопки проверки фиксируются и, если задание было выполнено неверно, считаются ошибками и запоминаются.

В работе используется упрощенный синтаксис языка Pascal.

Лабораторный практикум представляет собой программу исследования процесса компиляции, в которой можно выделить четыре основных шага, соответствующих этапам компиляции.

Данные шаги выполняются последовательно и включают в себя следующие задания.

Лексический анализ:

- выделение переменных и констант;
- расстановка приоритетов терминальных символов;
- разбор исходного текста на лексемы.

Синтаксический анализ:

- проведение синтаксического разбора восходящим методом, что включает в себя:
  - использование программы во внутреннем представлении в виде триад;
  - демонстрацию возможности использования тетрад;
- проведение синтаксического разбора нисходящим методом – построение дерева синтаксического разбора.

Оптимизация:

- проведение оптимизации методом свёртки;
- проведение оптимизации методом исключения лишних операций.

Генерация кода:

- генерация выходного кода на языке Ассемблера на основе построенной на предыдущем этапе таблицы триад.

Интерфейс программной модели показан на рисунке 2.10. Каждая вкладка соответствует одному из шагов компиляции. Вкладка становится доступной только после правильного выполнения предыдущего шага компиляции.

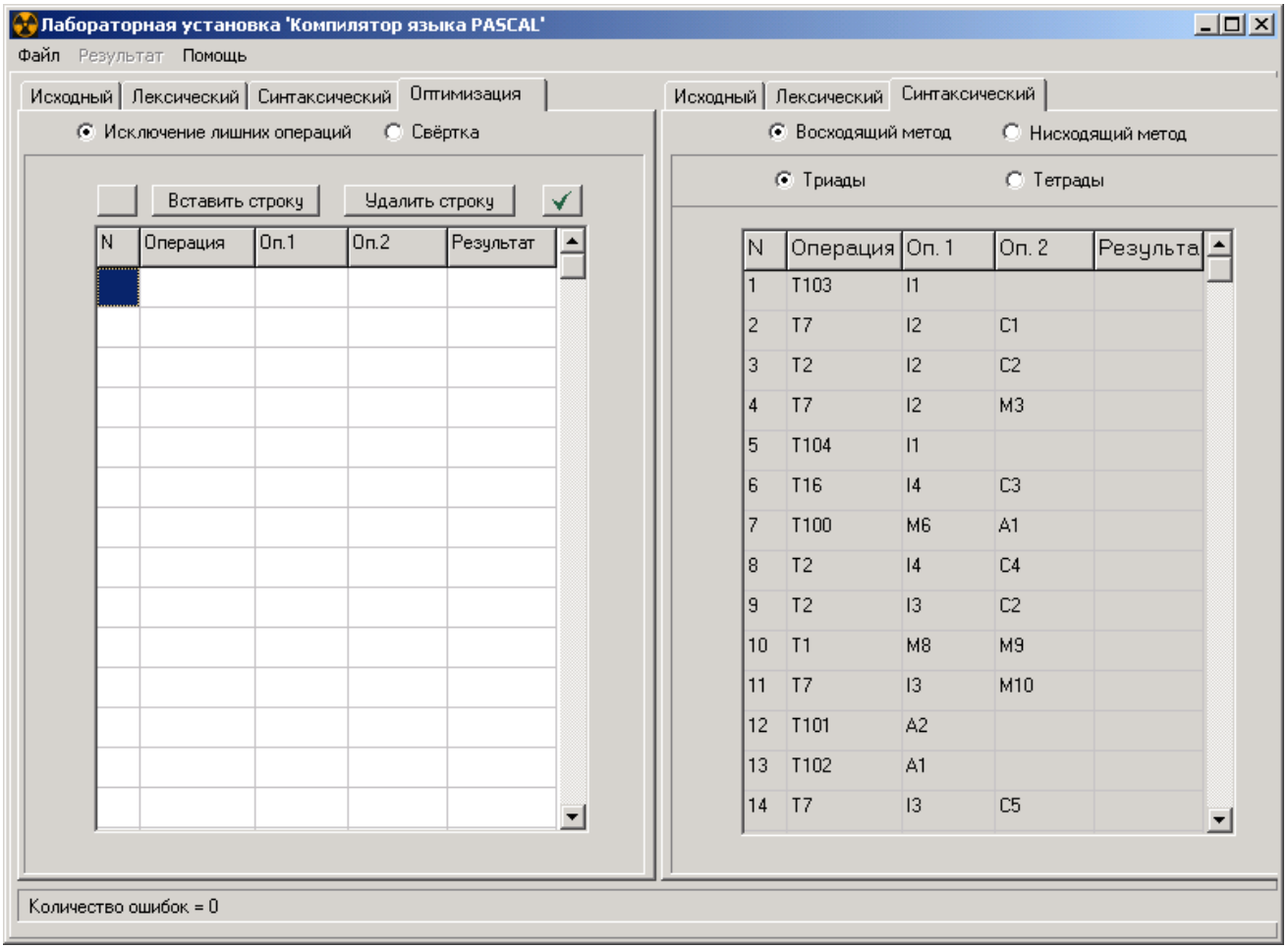


Рисунок 2.10 – Интерфейс программной модели

Выполнение лабораторной работы начинается с прохождения теста. **Студенты, не прошедшие тест, к лабораторной работе не допускаются.**

После прохождения теста загружается исходный текст программы на языке Pascal. Номер варианта лабораторной работы соответствует номеру компьютера, за которым выполняется работа.

После загрузки программы доступна вкладка «Лексический анализ».

На вкладке «Лексический анализ» расположены три внутренние служебные таблицы, которые предлагается заполнить.

После окончания работы с каждой из таблиц необходимо проверить правильность ее заполнения. Для этого необходимо нажать на кнопку "*Проверить*" (кнопка с зелёной галочкой), расположенную над соответствующей таблицей. При неверном заполнении какого-либо из элементов таблицы будет

выдано соответствующее сообщение с указанием места ошибки. Если таблица заполнена правильно, то произойдет изменение ее фона на более темный, и кнопка "*Проверить*" данной таблицы станет неактивной.

Таблица терминальных символов

В данной таблице поля "*символ*" и "*индекс*" заполнены программно и являются неизменными для всех вариантов заданий. Необходимо только правильно расставить приоритеты между операциями. При этом необходимо учитывать следующие особенности данной версии лабораторной установки:

- терминалы begin, end, end., ; и else имеют один и тот же приоритет;
- приоритет операций сравнения и закрывающей скобки один и тот же;
- приоритеты начинаются с нуля (низший приоритет) и заканчиваются шестью (высший приоритет).

Для расстановки приоритетов удобно воспользоваться следующей таблицей:

Приоритет	
0	Begin, End, End. , ;, Else
1	Then , Call
2	:= , If
3	= , > , < ,)
4	+ ,
5	* , /
6	(

Таблица констант

Необходимо заполнить все поля. При этом важно учитывать:

- индексы начинаются с единицы;
- константы заносятся по мере их появления в исходной программе.

Таблица переменных

Необходимо заполнить все поля. При этом должны соблюдаться следующие правила:

- индексы начинаются с единицы;
- первыми заносятся переменные процедур;

- поле "место" определяет местонахождение переменной:
  - 0 - основная программа;
  - 1, 2, 3... - процедуры по порядку их появления в теле программы;
- в поле "тип" для заголовка процедуры вводиться procname.

После заполнения всех таблиц автоматически произойдет переход к таблице лексем.

Таблица лексем

В данной таблице поле " *символ*" заполнять не обязательно - это поле является вспомогательным, введенным для удобства работы на следующих этапах. Поля "тип" и "индекс" выбираются из заполненных ранее внутренних таблиц. При этом для поля "тип" существуют следующие варианты заполнения:

T - терминальный символ;

I - переменная;

C - константа.

Заполнение таблицы идет последовательно по тексту программы.
При заполнении необходимо использовать терминал T103 - начало процедуры с символом PROCEDURE.

После правильного заполнения таблицы станет доступной вкладка «Синтаксический анализ».

Вкладка «Синтаксический анализ» предназначена для работы с таблицей триад и тетрад.

Таблица триад

Общие принципы работы с таблицей триад такие же как у таблицы лексем. При этом есть некоторые особенности, которые необходимо учитывать:

- не нужно заполнять поле "результат", так как оно используется только в случае применения тетрад, как формы внутреннего представления программы;
- формат полей "операция", "операнд 1", "операнд 2" имеет вид ТИ,

- где И - индекс символа в соответствующей таблице или порядковый номер, а Т - символ– идентификатор, принимающий следующие значения:

  М - строка матрицы;

  Т - таблица терминалов;

  С - таблица констант;

  I - таблица переменных;

  А - метка перехода.

Например, запись Т1 означает что используется терминальный символ имеющий в таблице терминалов индекс 1 (т. е. операция сложения);

- кроме терминальных символов из таблицы терминалов, представленной ранее, используются дополнительные терминальные символы, такие, как:

  Т100 - JF – Переход при невыполнении условия;

  Т101 - JMP – Безусловный переход;

  Т102 - Mark – Метка;

  Т103 - Procedure – Начало процедуры;

  Т104 - Procend – Конец процедуры;

- в случае отсутствия операнда поле соответствующего операнда остается незаполненным;

- нет необходимости обозначать конец и начало основной программы.

После правильного заполнения таблицы триад проводится демонстрация вида данного внутреннего представления для случая тетрад. При этом вводится новый символ **R,** обозначающий переменную памяти. Данный символ применяется вместо символа **М**. Кроме того, заполняется поле "*результат* ".

После окончания работы с таблицей триад и демонстрации таблицы тетрад необходимо переключиться на закладку «Нисходящий метод».

Вкладка «Нисходящий метод» предназначена для работы с деревом синтаксического разбора.

Дерево строится по основной программе; для процедур строятся только их вызовы.

Построение дерева

Главная ветвь дерева называется *idProgram* и изначально находится на форме построения дерева. *Все остальные ветви являются ее подветвями !!!*

Следующим уровнем являются секции: секция операций и секция переменных, причем секция переменных должна быть первой. В ней последовательно заносятся переменные и их типы.

Помните, что секция программ начинается с *Begin* и заканчивается *End.*

Ввод новых ветвей и редактирование старых производится нажатием на правую кнопку мыши на форме построения дерева, при этом появляется меню выбора операций.

При создании новой ветви дерева появляется форма "Добавление элемента". При этом в поле "*идентификатор*" необходимо правильно выбрать тип идентификатора. Существуют следующие типы:

idVariableSection - Секция переменных;

idOperation - SectionСекция операций;

idOperator - Оператор, т.е. строка заканчивающаяся точкой с запятой;

idExpression - Выражение, т.е. часть оператора, заключенная в скобки (сами скобки не относятся к выражению);

idKeyWord - Ключевое слово (например, VAR);

idIdent - Имя переменной;

idConst - Имя константы;

idSymbol - Символ (знаки операций, точка с запятой и т.д.);

idType - Тип переменной.

После построения правильного дерева синтаксического разбора становится доступной вкладка «Оптимизация».

Вкладка «Оптимизация» предназначена для оптимизации объектного модуля двумя методами. На этапе «Исключение лишних операций» рекомендуется четко следовать алгоритму, описанному в теории, даже если решение кажется очевидным, так как не все одинаковые операции можно исключать на

данном этапе. В таблицу необходимо вносить только окончательный вариант работы алгоритма.

После правильного заполнения таблицы необходимо перейти к этапу «Свертка».

На этапе «Свертка» предлагается повторно заполнить таблицу триад, но с оптимизированными триадами

Для ввода добавочных констант нужно воспользоваться кнопкой расположенной внизу таблицы.

После правильного заполнения данной таблицы необходимо перейти к этапу «Генерация».

Вкладка «Генерация» предназначена для построения выходного кода на языке Ассемблера.

Внимание!!! Выходной код строиться по строгим правилам перевода тетрад в ассемблерный код. При написании вместо пробелов использовать символы табуляции.

В общем случае структура на языке Ассемблера должна быть следующего вида:

```
.MODEL small
.DATA
.....
.CODE
PROC
.....
ENDP
Main:
.....
END Main
```

Секция переменных

- Переменные должны быть описаны в порядке их нахождения в таблице переменных, причем переменные процедур после имени должны иметь подчеркивание и индекс процедуры (Например, если в процедуре с индексом 2 есть переменная MMM, то она должна быть описана MMM_2).

- После описания переменных из таблицы переменных описываются дополнительные переменные, полученные на этапе построения таблицы тетрад (код строиться по ней).
- После переменных описываются константы из таблицы констант.
- Тип всех переменных и констант dw (для простоты).

Секция кода

Так как код является не полностью оптимизированным (в программе реализованы только два метода оптимизации), то любая триада переводиться в две, либо три строки ассемблерного кода. Например, триада присваивания сводиться к занесению в регистр AX второго операнда и занесению из регистра AX в ячейку, описанную в поле «результат» таблицы триад.

Оформление отчета

Отчет должен содержать дерево грамматического разбора, построенное восходящим методом, для фрагмента программы, указанного преподавателем.

Библиографический список

1. Григорьев В. Л. Микропроцессор i486. Архитектура и программирование — М.: ГРАНАЛ, 1993.
2. Гордеев А.В. Системное программное обеспечение /А.В. Гордеев, А.Ю.Молчанов. – СПб.:Питер, 2002. – 736с.:ил.
3. Молчанов А.Ю. Системное программное обеспечение: Учеб./ А.Ю.Молчанов. – СПб.:Питер, 2003. – 396с.:ил.
4. Пустоваров В.И. Язык Ассемблера в программировании информационных и управляющих систем /Пустоваров В.И. – Киев: М.: Век: Энтроп:Бином универсал, 1996. –304с.
5. Бек Л. Введение в системное программирование /Пер. с англ. /Л.Бек. − М.:Мир, 1988. − 448с.
6. Грис Д. Компиляторы для цифровых вычислительных машин: Пер. с англ./ Д.Грис.-М.:Мир, 1989.- 486с.
7. Григорьев В. Л. Микропроцессор i486. Архитектура и программирование: В 2 ч. / В. Л. Григорьев. – М.: ГРАНАЛ, 1993.
8. Абель П. Язык Ассемблера для IBM PC и программирования: Пер. с англ. / П. Абель. – М.: Высш. шк., 1992.
9. Фролов А.В. Аппаратное обеспечение IBM PC: В 2 ч. / А.В.Фролов, Г.В.Фролов.– М.: Диалог-МИФИ, 1993.
10. Юров В.И. Assembler: учебный курс / В.И. Юров. – СПб.:Питер, 1998.

Приложение 1

Набор команд для выполнения лабораторной работы «Дизассемблирование»

КОП	Команда	Описание
D4 0A	AAM	ASCII - коррекция регистра AX после умножения
3F	AAS	Коррекция AL после вычитания
14 ib	ADC AL,imm8	Сложение с CF непосредственного байта с AL
15 iw	ADC AX,imm16	Сложение с CF непосредственного слова с AX
80 /2 ib	ADC r/m8,imm8	Сложение с CF непосредственного байта с байтом r/m
81 /2 iw	ADC r/m16,imm16	Сложение с CF непосредственного слова со словом r/m
10 /r	ADC r/m8,r8	Сложение с CF байтного регистра с байтом r/m
11 /r	ADC r/m16,r16	Сложение с CF словного регистра со словом r/m
12 /r	ADC r8,r/m8	Сложение с CF байтного регистра с байтом r/m
13 /r	ADC r16,r/m16	Сложение с CF словного регистра со словом r/m
04 ib	ADD AL,imm8	Сложение непосредственного байта с AL
05 iw	ADD AX,imm16	Сложение непосредственного слова с AX
80 /0 ib	ADD r/m8,imm8	Сложение непосредственного байта с байтом r/m
81 /0 iw	ADD r/m16,imm16	Сложение непосредственного слова со словом r/m
00 /r	ADD r/m8,r8	Сложение байтного регистра с байтом r/m
01 /r	ADD r/m16,r16	Сложение словного регистра со словом r/m
02 /r	ADD r8,r/m8	Сложение байтного регистра с байтом r/m
03 /r	ADD r16,r/m16	Сложение словного регистра со словом r/m
24 ib	AND AL,imm8	Логическое И непосредственного байта с AL
25 iw	AND AX,imm16	Логическое И непосредственного слова с AX
80 /4 ib	AND r/m8,imm8	Логическое И непосредственного байта с байтом r/m
81 /4 iw	AND r/m16,imm16	Логическое И непосредственного слова со словом r/m
20 /r	AND r/m8,r8	Логическое И байтного регистра с байтом r/m
21 /r	AND r/m16,r16	Логическое И словного регистра со словом r/m
22 /r	AND r8,r/m8	Логическое И байтного регистра с байтом r/m
23 /r	AND r16,r/m16	Логическое И словного регистра со словом r/m
E8 cw	CALL rel16	Вызов близкий,смещение относительно следующей команды
FF /2	CALL r/m16	Вызов близкий, косвенный через r/m
98	CBW	AX <-- знаковое расширение AL
F8	CLC	CF <-- 0
FC	CLD	Сброс флажка направления DF <-- 0
FA	CLI	Сброс флажка прерывания IF <-- 0

F5	CMC	Дополнение (инвертирование) флага переноса CF <- NOT CF

КОП	**Команда**	**Описание**
3C ib	Cmp AL,imm8	Сравнение непосредственного байта с AL
3D iw	Cmp AX,imm16	Сравнение непосредственного слова с AX
80 /7 ib	Cmp r/m8,imm8	Сравнение непосредственного байта с байтом r/m
81 /7 iw	Cmp r/m16,imm16	Сравнение непосредственного слова со словом r/m
38 /r	Cmp r/m8,r8	Сравнение байтного регистра с байтом r/m
39 /r	Cmp r/m16,r16	Сравнение словного регистра со словом r/m
3A /r	Cmp r8,r/m8	Сравнение байтного регистра с байтом r/m
3B /r	Cmp r16,r/m16	Сравнение словного регистра со словом r/m
A6	CMPSB	Сравнение байта ES:[DI] (первый операнд) с байтом DS:[SI] (второй операнд)
A7	CMPSW	Сравнение слова ES:[DI] (первый операнд) с байтом DS:[SI] (второй операнд)
99	CWD	DX:AX <-- знаковое расширение AX
27	DAA	Десятичная коррекция AL после сложения
2F	DAS	Десятичная коррекция AL после вычитания
FE /1	DEC r/m8	Декремент байта r/m на 1
FF /1	DEC r/m16	Декремент слова r/m на 1
48+rw	DEC r16	Декремент словного регистра на 1
F6 /6	DIV r/m8	Деление AX на байт r/m (AL - частное, AH - остаток)
F7 /6	DIV r/m16	Деление DX:AX на слово r/m (AX - частное, DX - остаток)
F4	HLT	Останов процессора
F6 /7	IDIV r/m8	Знаковое деление AX на байт r/m (AL - частное, AH – остаток)
F7 /7	IDIV r/m16	Знаковое деление DX:AX на слово r/m (AX - частное, DX – остаток)
F6 /5	IMUL r/m8	Знаковое умножение AX <- AL * байт r/m
F7 /5	IMUL r/m16	Знаковое умножение DX:AX <- AX * слово r/m
E4 ib	IN AL,imm8	Ввод байта из непосредственного порта в AL
E5 ib	IN AX,imm8	Ввод слова из непосредственного порта в AX
EC	IN AL,DX	Ввод байта из порта DX в AL
ED	IN AX,DX	Ввод слова из порта DX в AX

FE /0	INC r/m8	Инкремент байта r/m на 1
FF /0	INC r/m16	Инкремент слова r/m на 1
40+rw	INC r16	Инкремент словного регистра на 1
CC	INT3	Прерывание 3-ловушка отладчика
CD	INT imm8	Номер прерывания в байте imm8

КОП	**Команда**	**Описание**
CE	INTO	Прерывание 4-если установлен флажок переполнения

Примечание: вместо команды **Int 3** рекомендуется использовать команду **Int3**

CF	IRET	Возврат из прерывания (далекий)
EB cb	JMP rel8	Переход короткий
E9 cw	JMP rel16	Переход близкий, смещение относительно следующей команды
FF /2	JMP r/m16	Переход близкий, косвенный через r/m

Jxx - Jump if condition is met (переход, если условие удовлетворяется)

77 cb	JA rel8	Если выше (CF=0 и ZF=0)
73 cb	JAE rel8	Если выше или равно (CF=0)
72 cb	JB rel8	Если ниже (CF=1)
76 cb	JBE rel8	Если ниже или равно (CF=1 или ZF=1)
72 cb	JC rel8	Если перенос (CF=1)
E3 cb	JCXZ rel8	Если регистр CX равен 0
74 cb	JE rel8	Если равно (ZF=1)
7F cb	JG rel8	Если больше (ZF=0 и SF=OF)
7D cb	JGE rel8	Если больше или равно (SF=OF)
7C cb	JL rel8	Если меньше (SF<>OF)
7E cb	JLE rel8	Если меньше или равно (ZF=1 или SF<>OF)
76 cb	JNA rel8	Если не выше (CF=1 и ZF=1)
72 cb	JNAE rel8	Если не выше или равно (CF=1)
73 cb	JNB rel8	Если не ниже (CF=0)
77 cb	JNBE rel8	Если не ниже или равно (CF=0 и ZF=0)
73 cb	JNC rel8	Если нет переноса (CF=0)
75 cb	JNE rel8	Если не равно (ZF=0)
7E cb	JNG rel8	Если не больше (ZF=1 или SF<>OF)
7C cb	JNGE rel8	Если не больше или равно (SF<>OF)
7D cb	JNL rel8	Если не меньше (SF=OF)
7F cb	JNLE rel8	Если не меньше или равно (ZF=0 и SF=OF)
71 cb	JNO rel8	Если нет переполнения (OF=0)
7B cb	JNP rel8	Если нет контроля четности (PF=0)
79 cb	JNS rel8	Если нет знака (SF=0)
75 cb	JNZ rel8	Если нет нуля (ZF=0)
70 cb	JO rel8	Если переполнение (OF=1)
7A cb	JP rel8	Если контроль четности (PF=1)
7A cb	JPE rel8	Если контроль четности (PF=1)
7B cb	JPO rel8	Если контроль нечетности (PF=0)|

78 cb	JS rel8	Если знак (SF=1)
74 cb J	Z rel8	Если 0 (ZF=1)
9F	LAHF	Загрузка: AH=флаги: SF ZF xx AF xx PF xx CF
C5 /r	LDS r16,m16:16	Загрузка в DS:r16 указателя памяти
C4 /r	LES r16,m16:16	Загрузка в ES:r16 указателя памяти

КОП	**Команда**	**Описание**
8D /r	LEA r16,m	Записать исполнительный адрес для m в регистр r16
AC	LODSB	Загрузка байта DS:[SI] в AL
AD	LODSW	Загрузка слова DS:[SI] в AX
88 /r	MOV r/m8,r8	Пересылка байтового регистра в байт r/m
89 /r	MOV r/m16,r16	Пересылка регистра-слова в слово r/m
8A /r	MOV r8,r/m8	Пересылка байта r/m в байтовый регистр
8B /r	MOV r16,r/m16	Пересылка слова r/m в регистр-слово
8C /r	MOV r/m16,Sreg	Пересылка сегментного регистра в слово r/m
8E /r	MOV Sreg,r/m16	Пересылка слова r/m в сегментный регистр
A0	MOV AL,moffs8	Пересылка байта в (сегмент:смещение) в AL
A1	MOV AX,moffs16	Пересылка слова в (сегмент:смещение) в AX
A2	MOV moffs8,AL	Пересылка AL в (сегмент:смещение)
A3	MOV moffs16,AX	Пересылка AX в (сегмент:смещение)
B0+rb	MOV reg8,imm8	Пересылка непосредственного байта в регистр
B8+rw	MOV reg16,imm16	Пересылка непосредственного слова в регистр
C6	MOV r/m8,imm8	Пересылка непосредственного байта в байт r/m
C7	MOV r/m16,imm16	Пересылка непосредственного слова в слово r/m
A4	MOVSB	Пересылка строки байтов DS:[SI] в ES:[DI]
A5	MOVSW	Пересылка строки слов DS:[SI] в ES:[DI]
F6 /4	MUL AL,r/m8	Умножение без знака (AX <- AL*байт r/m)
F7 /4	MUL AX,r/m16	Умножение без знака (DX:AX <- AX*слово r/m)
F6 /3	NEG r/m8	Отрицание с дополнением до двух байта r/m
F7 /3	NEG r/m16	Отрицание с дополнением до двух слова r/m
90	NOP	Нет операции
F6 /2	NOT r/m8	Изменение на противоположное значения каждого бита в байте r/m
F7 /2	NOT r/m16	Изменение на противоположное значения каждого бита в слове r/m
0C ib	OR AL,imm8	Операция логического ИЛИ непосредственного байта и AL

0D iw AX	OR AX,imm16	Операция логического ИЛИ непосредственного слова и
80 /1 ib	OR r/m8,imm8	Операция логического ИЛИ непосредственного байта и байта в r/m
81 /1 iw	OR r/m16,imm16	Операция логического ИЛИ непосредственного слова и слова в r/m
08 /r	OR r/m8,r8	Операция логического ИЛИ байтового регистра и байта в r/m
09 /r	OR r/m16,r16	Операция логического ИЛИ регистра-слова и слова в r/m
0A /r стра	OR r8,r/m8	Операция логического ИЛИ байта в r/m и байтового реги-
OB /r	OR r16,r/m16	Операция логического ИЛИ слова в r/m и регистра-слова
E6 ib та	OUT imm8,AL	Вывод байта AL в непосредственно заданный номер пор-

КОП	**Команда**	**Описание**
E7 ib та	OUT imm8,AX	Вывод слова AX в непосредственно заданный номер пор-
EE	OUT DX,AL	Вывод байта AL в порт, номер которого задан в DX
EF	OUT DX,AX	Вывод слова AX в порт, номер которого задан в DX
8F /0 ти	POP m16	Извлечение вершины стека и помещение ее в слово памя-
58+ rw регистр	POP r16	Извлечение вершины стека и помещение ее в слово-
1F	POP DS	Извлечение вершины стека и помещение ее в регистр DS
07	POP ES	Извлечение вершины стека и помещение ее в регистр ES
17	POP SS	Извлечение вершины стека и помещение ее в регистр SS
9D	POPF	Извлечение вершины стека в FLAGS
FF /6	PUSH m16	Помещение в стек слова памяти
50+rw	PUSH r16	Помещение в стек слова - регистра
0E	PUSH CS	Помещение в стек CS
16	PUSH SS	Помещение в стек SS
1E	PUSH DS	Помещение в стек DS
06	PUSH ES	Помещение в стек ES
9C	PUSHF	Помещение в стек FLAGS
D0 /2	RCL r/m8,1	Циклический сдвиг 9 битов (CF, байт r/m) влево 1 раз
D2 /2	RCL r/m8,CL	Циклический сдвиг 9 битов (CF, байт r/m) влево CL раз
D1 /2	RCL r/m16,1	Циклический сдвиг 17 битов (CF,слово r/m) влево 1 раз
D3 /2	RCL r/m16,CL	Циклический сдвиг 17 битов (CF,слово r/m) влево CL раз
D0 /3	RCR r/m8,1	Циклический сдвиг 9 битов (CF, байт r/m) вправо 1 раз
D2 /3	RCR r/m8,CL	Циклический сдвиг 9 битов (CF, байт r/m) вправо CL раз
D1 /3	RCR r/m16,1	Циклический сдвиг 17 битов (CF,слово r/m) вправо 1 раз
D3 /3 раз	RCR r/m16,CL	Циклический сдвиг 17 битов (CF,слово r/m) вправо CL

D0 /0	ROL r/m8,1	Циклический сдвиг 8 битов (байт r/m) влево 1 раз
D2 /0	ROL r/m8,CL	Циклический сдвиг 8 битов (байт r/m) влево CL раз
D1 /0	ROL r/m16,1	Циклический сдвиг 16 битов (слово r/m) влево 1 раз
D3 /0	ROL r/m16,CL	Циклический сдвиг 16 битов (слово r/m) влево CL раз
D0 /1	ROR r/m8,1	Циклический сдвиг 8 битов (байт r/m) вправо 1 раз
D2 /1	ROR r/m8,CL	Циклический сдвиг 8 битов (байт r/m) вправо CL раз
D1 /1	ROR r/m16,1	Циклический сдвиг 16 битов (слово r/m) вправо 1 раз
D3 /1	ROR r/m16,CL	Циклический сдвиг 16 битов (слово r/m) вправо CL раз
C3	RET	Возврат (ближний) в вызывающую процедуру
C3	RETN	Возврат (ближний) в вызывающую процедуру
CB	RETF	Возврат (дальний) в вызывающую процедуру
9E	SAHF	Запись AH во флаги: SF ZF xx AF xx PF xx CF
D0 /4	SAL r/m8,1	Умножение на 2 один раз байта r/m
D2 /4	SAL r/m8,CL	Умножение на 2 CL раз байта r/m
D1 /4	SAL r/m16,1	Умножение на 2 один раз слова r/m
D3 /4	SAL r/m16,CL	Умножение на 2 CL раз слова r/m
D0 /7	SAR r/m8,1	Деление со знаком на 2 один раз байта r/m

КОП	**Команда**	**Описание**
D2 /7	SAR r/m8,CL	Деление со знаком на 2 CL раз байта r/m
D1 /7	SAR r/m16,1	Деление со знаком на 2 один раз слова r/m
D3 /7	SAR r/m16,CL	Деление со знаком на 2 CL раз слова r/m
D0 /4	SHL r/m8,1	Умножение на 2 один раз байта r/m
D2 /4	SHL r/m8,CL	Умножение на 2 CL раз байта r/m
D1 /4	SHL r/m16,1	Умножение на 2 один раз слова r/m
D3 /4	SHL r/m16,CL	Умножение на 2 CL раз слова r/m
D0 /5	SHR r/m8,1	Деление без знака на 2 один раз байта r/m
D2 /5	SHR r/m8,CL	Деление без знака на 2 CL раз байта r/m
D1 /5	SHR r/m16,1	Деление без знака на 2 один раз слова r/m
D3 /5	SHR r/m16,CL	Деление без знака на 2 CL раз слова r/m
1C ib	Sbb AL,imm8	Вычитание с заемом непосредственного байта с AL
1D iw	Sbb AX,imm16	Вычитание с заемом непосредственного слова с AX
80 /3 ib r/m	Sbb r/m8,imm8	Вычитание с заемом непосредственного байта с байтом
81 /3 iw r/m	Sbb r/m16,imm16	Вычитание с заемом непосредственного слова со словом
18 /r	Sbb r/m8,r8	Вычитание с заемом байтного регистра с байтом r/m
19 /r	Sbb r/m16,r16	Вычитание с заемом словного регистра со словом r/m
1A /r	Sbb r8,r/m8	Вычитание с заемом байтного регистра с байтом r/m
1B /r	Sbb r16,r/m16	Вычитание с заемом словного регистра со словом r/m
AE	SCASB	Сравнение байтов AL и ES:[DI], обновление DI
AF	SCASW	Сравнение слов AX и ES:[DI], обновление DI
F9	STC	Установка флага переноса
FD	STD	Установка флага направления, что соответствует декременту SI или DI

FB	STI	Установка флага прерывания: прерывания разрешены, начиная с конца следующей команды
AA	STOSB	Сохранение AL в байте ES:[DI], обновление DI
AB	STOSW	Сохранение AX в слове ES:[DI], обновление DI
2C ib	Sub AL,imm8	Вычитание непосредственного байта с AL
2D iw	Sub AX,imm16	Вычитание непосредственного слова с AX
80 /5 ib	Sub r/m8,imm8	Вычитание непосредственного байта с байтом r/m
81 /5 iw	Sub r/m16,imm16	Вычитание непосредственного слова со словом r/m
28 /r	Sub r/m8,r8	Вычитание байтного регистра с байтом r/m
29 /r	Sub r/m16,r16	Вычитание словного регистра со словом r/m
2A /r	Sub r8,r/m8	Вычитание байтного регистра с байтом r/m
2B /r	Sub r16,r/m16	Вычитание словного регистра со словом r/m
A8 ib	TEST AL,imm8	Логическое И непосредственного байта с AL
A9 iw	TEST AX,imm16	Логическое И непосредственного слова с AX
F6 /0 ib	TEST r/m8,imm8	Логическое И непосредственного байта с байтом r/m
F7 /0 iw	TEST r/m16,imm16	Логическое И непосредственного слова со словом r/m
84 /r	TEST r/m8,r8	Логическое И байтного регистра с байтом r/m
85 /r	TEST r/m16,r16	Логическое И словного регистра со словом r/m
D7	XLAT	Установка AL равным байту памяти DS:[BX+AL без зна-
ка]		

КОП	**Команда**	**Описание**
D7	XLATB	Установка AL равным байту памяти DS:[BX+AL без зна-
ка]		
90 +rw	XCHG AX,r16	Обмен местами регистра-слова и AX
90 +rw	XCHG r16,AX	Обмен местами регистра-слова и AX
86 /r	XCHG r/m8,r8	Обмен местами регистра-байта и байта по исполнитель-
ному		адресу
86 /r	XCHG r8,r/m8	Обмен местами регистра-байта и байта по исполнитель-
ному		адресу
87 /r	XCHG r/m16,r16	Обмен местами регистра-слова и слова по исполнитель-
ному		адресу
87 /r	XCHG r16,r/m16	Обмен местами регистра-слова и слова по исполнитель-
ному		адресу
34 ib	XOR AL,imm8	Операция логического исключающего ИЛИ непосредственного байта и AL
35 iw	XOR AX,imm16	Операция логического исключающего ИЛИ непосредственного слова и AX
80 /6 ib	XOR r/m8,imm8	Операция логического исключающего ИЛИ

		непосредственного байта и байта в r/m
81 /6 iw	XOR r/m16,imm16	Операция логического исключающего ИЛИ непосредственного слова и слова в r/m
30 /r	XOR r/m8,r8	Операция логического исключающего ИЛИ байтового регистра и байта в r/m
31 /r	XOR r/m16,r16	Операция логического исключающего ИЛИ регистра-слова и слова в r/m
32 /r	XOR r8,r/m8	Операция логического исключающего ИЛИ байта в r/m и байтового регистра
33 /r	XOR r16,r/m16	Операция логического исключающего ИЛИ слова в r/m и регистра-слова