Введение.
Обзор языков программирования

Любая вычислительная система — от мобильного телефона и до суперЭВМ сверхвысокой производительности — обладает своим собственным языком — набором инструкций ее процессора. Этот язык принято называть *машинным* языком. Его алфавит состоит всего из двух символов — 0 и 1. Разрабатывать программы на таком языке — крайне тяжелый и малопроизводительный труд, однако на заре развития вычислительной техники приходилось поступать именно так.

В дальнейшем для программирования стали использовать более удобный язык, в котором каждой инструкции сопоставлялось ее мнемоническое (символическое) обозначение. Символические обозначения стали использоваться и для указания адресов операндов. Такой символический язык получил название *языка ассемблера.*

Естественно, что для компьютеров с различными архитектурами существовали свои ассемблеры. Для перевода (трансляции) исходной программы на языке ассемблера в машинный язык использовалась специальная программа, которая тоже называлась *ассемблером.* Таким образом, язык ассемблера стал первым языком программирования, а ассемблеры – первыми специальными программами — *трансляторами.*

Ассемблеры существенно повысили производительность труда программистов, однако решаемые задачи усложнялись и потребовались более совершенные средства для разработки программ. Такими средствами стали языки программирования высокого уровня. Первоначально они использовались в основном для решения различных прикладных задач, однако сфера их применения очень быстро расширялась. В настоящее время они эффективно используются для решения задач, которые до сих пор считалось необходимым программировать на языке ассемблера.

Так, например, операционная система UNIX на 90 % написана на языке высокого уровня С. Первым языком программирова-

ния высокого уровня, получившим широкое распространение, по праву считается язык Algol, разработанный в 60-х гг. XX столетия. Свое название он получил как аббревиатура английских слов *Algo*rithmic *L*anguage. Он использовался как язык программирования и как язык публикации алгоритмов. Его особенностями были блочная структура, объявления переменных, определение способа передачи параметров и т.п.

Большинство современных языков программирования основывается на идеях, заложенных в Algol. Однако в последующие годы Algol был вытеснен так называемыми проблемно-ориентированными языками, получившими условное наименование языков II поколения. К их числу принадлежат: FORTRAN (*For*mula *Tran*slator) — язык для выполнения научных и инженерно-технических расчетов, COBOL (*C*ommon *B*usiness-*O*riented *L*anguage) — язык для решения коммерческих задач, BASIC (*B*eginner's *A*ll *P*urpose *S*ymbolic *I*nstruction *C*ode) — язык для обучения программированию и т.п.

Каждый из этих языков имел свои характерные особенности. Так, например, в языке FORTRAN имелся развитый аппарат подпрограмм (процедур и функций), язык COBOL отличался достаточно мощными средствами работы с файлами, десятичной арифметикой и формой записи, приближенной к английскому языку. Язык BASIC был очень прост в освоении и ориентирован на диалоговую работу. На смену перечисленным языкам в 80-е гг. пришли так называемые языки программирования III поколения, к числу которых, прежде всего, можно отнести языки Pascal и C. Они предназначались для решения различных задач. Общими чертами этих языков являются понятие типа данных и возможность конструирования новых типов данных любой сложности, базирующихся на элементарных типах данных, встроенных в язык. Кроме того, оба они основаны на принципах структурного программирования.

Усложнение возлагаемых на компьютеры задач и увеличение аппаратных ресурсов вычислительной техники привели к разработке языков программирования IV поколения, среди которых наибольшее распространение получили языки C++, Object-Pascal, Modula-2 и Ada. Последний назван в честь Августы Ады Лавелейс — первой в истории женщины-программиста. Все эти языки являются объ-

ектно-ориентированными, а Modula-2 и Ada имеют встроенные средства для реализации параллельных вычислений, такие как средства синхронизации, передача управления между процессами, обмен данными и т.п. Следует отметить, что программное обеспечение космической программы Shuttle написано на языке Ada.

Развитие сетевых технологий, начавшееся в 80-х гг. XX в., привело к широкому распространению распределенных вычислений, а также к необходимости разработки программного обеспечения, способного работать без перекомпиляции в рамках неоднородных (гетерогенных) вычислительных сетей, т.е., к переносимому (мобильному) программному обеспечению. Все перечисленные выше языки программирования являются *языками компилирующего типа.* В этих языках исходный текст программы с помощью компилятора (транслятора) переводится сразу в объектный код компьютера конкретной архитектуры, работающего под управлением конкретной операционной системы. В дальнейшем этот код может быть выполнен на различных компьютерах этой же архитектуры, но работающих под управлением одной и той же операционной системы. Таким образом, программы, подготовленные на перечисленных выше языках программирования, являются переносимыми только на уровне исходных текстов. Для их выполнения на компьютерах какой-то другой архитектуры или даже в рамках другой операционной системы необходимо их перекомпилировать.

Однако гетерогенные вычислительные сети содержат множество компьютеров различной архитектуры, работающих под управлением различных операционных систем. В этой ситуации актуальной является разработка программного обеспечения, которое могло бы работать на различных компьютерах без перекомпиляции, так как разработчики программ в настоящее время могут и не знать заранее, в какой вычислительной среде будут работать их программы. Для решения этой задачи были разработаны и внедрены *языки интерпретирующего типа.* При их использовании исходная программа транслируется не в объектный код, как во всех вышеупомянутых языках, а в некоторый промежуточный код, называемый *Р-кодом.* Последний можно рассматривать как «машинный язык» некоторой виртуальной машины (псевдомашины). В дальнейшем Р-код

может быть выполнен в режиме интерпретации на любом компьютере и в любой операционной среде, где такая виртуальная машина установлена. В настоящее время эта идея реализована в распространенном языке программирования Java.

Особое место среди языков программирования занимают *языки логического программирования*. Все традиционные языки требуют алгоритмического подхода: они реализуют алгоритм, т.е. детерминированные формальные вычисления директивного типа. Однако первоначальная формулировка некоторой задачи имеет, как правило, описательный характер. Неформальный этап построения подходящего алгоритма и запись его на некотором формальном языке требуют не только значительных затрат времени, но и достаточно высокой квалификации в соответствующей предметной области. Этот недостаток стимулировал поиск других возможностей.

В общем случае вычисление представляет собой частный случай логического вывода, а алгоритм — аксиоматическое задание функции. Отсюда возникла идея предоставить компьютеру в качестве программы не алгоритм, а описание предметной области и решаемой задачи (функции) в виде аксиоматической системы, и тогда построение решения задачи в виде вывода в этой системе можно поручать самой машине.

Таким образом, программист уже не должен строить алгоритм для решения задачи, поскольку нужный алгоритм создает сама система программирования. Отсюда следует, что основная задача программиста — описать предметную область в виде системы логических формул и такого множества отношений на ней, которые с достаточной степенью полноты представляют задачу. Такой подход был реализован в виде экспертных систем, одним из примеров которых является PROLOG (*Pro*gramming in *Log*ic).

Программа на языке PROLOG представляет собой совокупность утверждений и правил. Утверждения состоят из предикатов, логических связок и констант и образуют базу данных. Правила имеют вид «A если B_1 и B_2 и.....B_k», где A и B_j — предикаты, содержащие переменные. Выполнение программы на PROLOG инициируется запросом, состоящим из предикатов, логических связок, констант и переменных.

Кроме того, во многих случаях находят применение различные специальные языки, например, JPSS — язык для моделирования различных систем, язык обработки списков LISP (*Lis*ts *P*rocessing) и др. LISP применяется в теории игр, теории искусственного интеллекта, в автоматизации доказательства теорем, обработке естественных языков и т.п. В LISP представления программ и данных эквивалентны, что позволяет вычислять структуры данных как программы и модифицировать программы как данные. Основная управляющая структура в этом языке — не цикл, а рекурсия.

Следует отметить специальный язык для автоматизации математических расчетов MATLAB, построенный на расширенном представлении и применении матричных операций (MATrix LABoratory — матричная лаборатория). В нем реализованы такие мощные типы данных, как многомерные массивы, массивы ячеек, массивы структур, массивы Java и разреженные матрицы, что позволяет применять его при создании и отладке новых алгоритмов матричных и основанных на них параллельных вычислений и крупных баз данных.

Совершенствование и развитие операционных систем и системного программного обеспечения стимулирует разработку новых языков программирования, учитывающих все достижения своих предшественников и предоставляющих программистам все новые возможности. Удачным примером такого языка является разработанный сравнительно недавно язык программирования C#, учитывающий достижения многих других языков программирования: C++, Delphi – Pascal, Visual Basic, Java. В результате был получен действительно простой, удобный и современный язык программирования, по мощности не уступающий C++, но существенно повысивший продуктивность программных разработок.

Итак, исходная программа, написанная на языке компилирующего типа (как на языке высокого уровня, так и на языке ассемблера), с помощью транслятора (компилятора) преобразуется в программу на машинном языке, называемую *объектным кодом*. Для программ, написанных на языках интерпретирующего типа, результатом трансляции исходного текста является промежуточный P-код, который в дальнейшем выполняется (интерпретируется) на виртуальной машине. Таким образом, очевидно, что общим для

обоих подходов является обработка исходного текста программы (трансляция). Генерация объектного кода зависит, в первую очередь, от архитектуры аппаратных средств конкретного компьютера. В случае интерпретирующих языков генерация промежуточного Р-кода выполняется стандартно и не зависит от архитектуры аппаратных средств.

В настоящем пособии основное внимание уделяется вопросам обработки исходных текстов программ. Генерация объектных кодов рассматривается в общих чертах и носит иллюстративный характер.

Глава 1. МАШИННЫЙ ЯЗЫК И ЯЗЫК АССЕМБЛЕРА

Почти все традиционные современные ЭВМ используют концепцию запоминаемой программы, которую связывают с именем известного математика Дж. фон Неймана. Общая структура центрального процессора типичной машины фон Неймана имеет вид, представленный на рис. 1.1.

Центральный процессор состоит из интерпретатора команд, счетчика адреса, регистра команд и ряда рабочих и общих регистров.

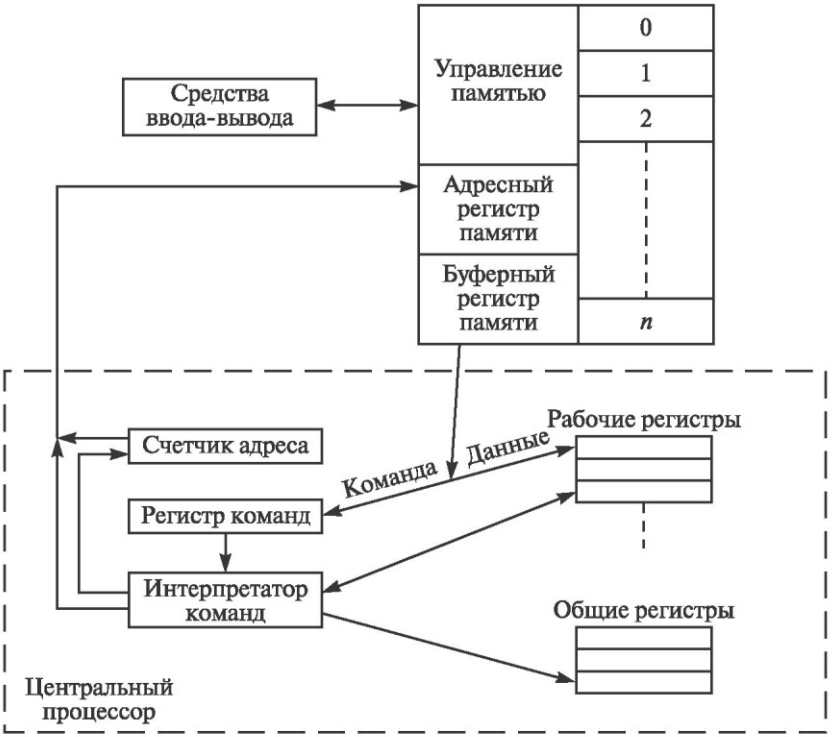

Рис. 1.1. Архитектура процессора фон Неймана

Интерпретатор команд осуществляет действия, определяемые смыслом выбираемой из памяти команды.

Счетчик адреса (LC), называемый также программным счетчиком PC или счетчиком команд (Instruction Counter — IC), — запоминающее устройство, которое указывает адрес выполняемой команды. Копия текущей команды сохраняется в регистре команд IR. Рабочие регистры представляют собой запоминающие устройства, которые служат промежуточной памятью для интерпретатора команд. Общие регистры используются программистом как временная память и для вычисления специальных функций.

Адресный регистр памяти содержит адрес ячейки памяти, по которому должна вычисляться операция чтения — записи. Буферный регистр памяти содержит копию ячейки памяти, адрес которой указан в адресном регистре, а перед записью — новое содержимое этой ячейки. Устройство управления памятью обеспечивает передачу информации между ячейками памяти, адрес которых указан в адресном регистре, и буферным регистром памяти.

Все современные ЭВМ в той или иной мере имеют данную архитектуру. Общими элементами для любой аппаратной платформы являются следующие:

1. *Память.* Для построения объектного кода очень важны способы адресации, применяемые в процессоре, и объем оперативной памяти. В подавляющем большинстве современных компьютеров минимально адресуемая единица памяти равна байту. Два байта составляют слово. Два слова составляют двойное слово. Такая структура позволяет выбирать команды и данные длиной от одного до нескольких байтов. Очень важными для построения объектного кода являются способы адресации, используемые в процессоре.

Основные схемы адресации современных процессоров показаны на рис. 1.2. В случае прямой адресации в адресной части находится адрес данных. Относительный адрес указывается относительно начала некоторого блока памяти, абсолютный адрес представляет собой непосредственный номер ячейки памяти. В случае косвенной адресации в адресной части находится не адрес данных, а адрес адреса данных. Такая схема облегчает использование указателей в языках высокого уровня.

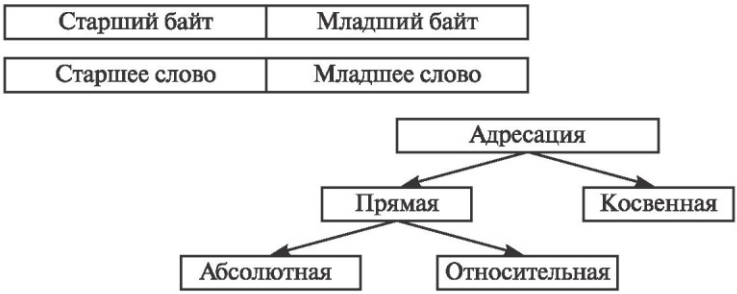

Рис. 1.2. Способы адресации в современных компьютерах

Объем оперативной памяти зависит от типа процессора. Так, в системе Intel-386 допускается подключение до 2 Мбайт оперативной памяти. Минимальная адресуемая единица — 1 байт. В системе z/OS корпорации IBM объем оперативной памяти достигает 2^{31} Мбайт. Минимальная адресуемая единица — 1 байт.

2. *Регистры.* Очень важным для построения объектной программы является число программно доступных регистров, их функциональное назначение и разрядность. В архитектуре Intel-386 предусмотрены регистры общего назначения, символически обозначаемые ax, bx, cx, dx, sp, bp, si, di, и специальные регистры es, ds, ss, cs, ip. В системе z/OS имеется 16 регистров общего назначения и 4 регистра для выполнения команд с плавающей точкой. Общие регистры могут использоваться для различных операций и в качестве базисных регистров.

3. *Типы данных*, обрабатываемые процессором. В архитектуре Intel-386 обрабатываемые данные могут иметь переменную длину, от 1 до 4 байт (формат с плавающей точкой). Кроме того, предусмотрена обработка последовательностей символов. В системе z/OS, кроме общепринятых типов (целые, вещественные, символьные), данные могут быть представлены в двоично-десятичном и упакованном форматах.

4. *Команды.* Для обработки различных типов данных в процессорах используются различные классы команд. При этом адресная часть каждой команды формируется в зависимости от типа обрабатываемых данных. При наличии регистров общего назначения, все

процессоры, как правило, реализуют следующие основные способы адресации операндов:

- регистр—регистр;
- регистр—память;
- память—регистр;
- память—память.

В архитектуре Intel-386 команды имеют различные форматы. Их длина варьируется от 1 до 6 байт. Это сделано для того, чтобы можно было коротко представлять простые команды. Адрес представляется номером сегмента и смещением. Допускается как прямая, так и косвенная адресация. Кроме того, данные могут быть представлены в виде литералов. Литерал представляет собой непосредственное задание операнда в команде.

В системе z/OS адрес формируется как сумма смещения, содержимого базисного регистра и содержимого индексного регистра. Команды, ссылающиеся на операнды, расположенные в оперативной памяти, должны использовать относительную адресацию с базированием. Механизм косвенной адресации в этой системе не реализован.

Рассмотренные примеры показывают, что большое разнообразие аппаратных платформ оказывает существенное влияние на генерацию объектного кода, поэтому в любом трансляторе можно выделить аппаратно-зависимую и аппаратно-независимую составляющие. Первая — это системы команд и форматы представления данных, способы адресации и число различных регистров общего и специального назначения. Вторая — это обработка исходного текста программы.

1.1. Ассемблер

Выше говорилось, что первым языком программирования стал язык ассемблера. В общем случае язык ассемблера состоит из отдельных операторов. Каждый оператор представляет собой набор термов и выражений. В выражениях используются в основном арифметические операции, строящиеся по стандартным правилам с использованием «+», «–», «*», «/». Кроме операторов, в языке ассемблера, как правило, присутствуют директивы ассемблера. Разумеется, язык ассемблера имеет алфавит. В общем виде структуру оператора ассемблера можно представить следующим образом:

[<метка>]<разд><коп><разд><адрес>,

где <разд> — символ-разделитель («_», «:» и т.п.);

метка, коп — термы, адрес — терм или выражения.

Выражения — это термы, связанные арифметическими операциями.

Термы могут быть относительными и абсолютными. Константа — абсолютный терм. Метки и термы в адресной части — относительные термы.

Директивы ассемблера имеют следующий вид:

[<метка>]<разд><имя директивы><разд>[<метка>].

Имя директивы — это терм; метка слева — терм; метка справа — терм или выражение. Термы в директивах могут быть как абсолютными, так и относительными.

Некоторые термы называются специальными, например термы, ссылающиеся на текущее значение счетчика команд («*» или «.»).

Относительный терм обычно задает некоторую точку внутри программы. Обычно исходный текст программы на языке ассемблера оформляется в виде текстового файла на диске. Каждая запись (запись = предложение) в таком файле представляет собой оператор или директиву ассемблера. В начале программы обычно находится директива, определяющая имя и начальный адрес программы. В конце исходного текста программы обычно находится специальная директива, указывающая на конец ассемблирования и задающая первую исполняемую команду программы. Транслятор с языка ассемблера исторически также называется *ассемблером*, или во избежание путаницы *программой-ассемблер.*

Таким образом, ассемблер — это программа, на вход которой поступает программа на языке ассемблера, а на выходе – получается эквивалентная ей программа на машинном языке. Кроме того, ассемблер должен формировать необходимую для загрузчика и/или редактора связей информацию. Например, должны быть отмечены и переданы загрузчику базовые регистры, а редактору связей — внешние символы. Ассемблер не знает адресов (значений) этих символов и на редактор связей возлагается задача найти программы, содержащие ссылки на эти символы, подключить их к вызывающей программе и поместить значения этих символов в вызывающую программу. Кроме того, ассемблер должен обеспечивать выдачу удобного для использования листинга с индикацией ошибок

ассемблирования, если они есть. Программа-ассемблер для перевода исходного текста программы в обычный код в простейшем случае должна выполнять следующие действия:

1) преобразовывать символические коды операций в их эквиваленты на машинном языке;

2) преобразовывать символические адреса (операнды) в эквивалентные им машинные адреса;

3) построить машинные команды в соответствующем формате;

4) преобразовывать константы, заданные в исходной программе, во внутренние машинные представления;

5) записать на диск объектную программу и выдать листинг;

6) отметить ошибки в исходной программе.

Все указанные действия могут быть легко выполнены простой построчной обработкой исходной программы, за исключением действия 2. Например, в операторе

```
mov ax, DATA + 2
```

заранее неизвестно, какой адрес будет присвоен метке DATA. Поэтому большинство ассемблеров выполняют два просмотра исходной программы. Основной задачей первого просмотра является поиск символических имен и назначение им адресов. Фактическая трансляция (действия 1—5) выполняется во время второго просмотра. Ассемблер, наряду с трансляцией исходной программы, должен также обрабатывать и директивы ассемблера. Эти директивы непосредственно не переводятся в машинные команды, а управляют работой самого ассемблера.

Принцип работы программы-ассемблер рассматривается на следующем примере программы, написанной на языке ассемблера 80386:

```
//В начале программы помещаются директивы,
Title Prog. ASM //определяющие имя программы,
Prog Segment // объявляющие программный сегмент
assume cs: Prog, ds: Prog и сегментные регистры,
Begin       proc //а также начальный адрес программы.
            mov ax, prog;  // далее (1)
            mov ds, ax;    // следуют (2)
            mov al, x;     // операторы (3)
            mov bx, y;     // исходной программы (4)
x           db      // и директивы (5)
```

```
y           dw      // резервирования памяти (6)
//В конец программы помещаются специальные директивы,
Begin       endp // задающие первую исполняемую команду
                 // программы
Prog        ends // и указывающие на конец
            end // ассемблирования.
```

В рассматриваемом примере символические адреса x и y объявляются после их использования в операторах, отмеченных символами (3) и (4). Поэтому во время первого просмотра вычисляются их относительные адреса, как это показано на рис. 1.3.

		отн.адрес	коп	адр
Title Prog. ASM		0		
Prog Segment		0		
assume cs: prog, ds: prog		0		
Begin	prog	0		
	mov ax, prog; // длина 3 б	0	mov	ax, ?
	mov ds, ax; // длина 2 б	3	mov	ds, ax
	mov al, x; // длина 4 б	5	mov	ax, ?
	mov bx, y; // длина 4 б	9	mov	bx, ?
x	db 1 б	D 00		
y	dw 2 б	E 0000		
Begin	endp	10		
Prog	ends	10		
	end	10		

Рис. 1.3. Первый просмотр программы-ассемблер

Директивы ассемблера не влияют на значение счетчика адреса, следовательно, относительный адрес остается неизменным (равным 0) до оператора (1), который имеет длину 3 байт. Поэтому относительный адрес оператора (2) будет равен 3 и т.д. Директивы резервирования памяти (5) и (6) выделяют один и два байта в оперативной памяти и, следовательно, тоже влияют на счетчик адреса. После того как вычислены адреса меток x и y, появляется возможность подставить их значения (D и E соответственно) в операторы (3) и (4). Эта подстановка выполняется на втором просмотре программы-ассемблер (рис. 1.4).

			отн.адрес	коп	адр
Title Prog. ASM			0		
Prog Segment			0		
assume cs: prog, ds: prog			0		
Begin	prog		0		
	mov ax, prog;	// .... 3 б	0	mov	ax, - - -R
	mov ds, ax;	// .... 2 б	3	mov	ds, ax
	mov al, x;	// .... 4 б	5	mov	ax, [D]R
	mov bx, y;	// .... 4 б	9	mov	bx, [E]R
x	db 1 б		D 00		
y	dw 2 б		E 0000		
Begin	endp		10		
Prog	ends		10		
	end		10		

Рис. 1.4. Второй просмотр программы-ассемблер

Символ R означает, что помеченные им адреса являются перемещаемыми. Символ --- означает, что он не определен на этапе ассемблирования, его значение будет определено при загрузке программы в оперативную память. Естественно, что при выполнении второго просмотра символические имена операций заменяются их двоичными эквивалентами (операторами машинного языка).

Таким образом, цель первого просмотра заключается в присвоении адреса каждой команде или псевдокоманде определения данных, т.е. в определении значений символов, появляющихся в поле метки исходной программы.

Для этого необходимо:

1) определить длину машинных команд;

2) следить за значениями счетчика адреса;

3) запоминать значения символов для второго просмотра;

4) обработать некоторые псевдокоманды (в частности, резервирование памяти);

5) запомнить литералы.

Структурами данных, которыми оперирует ассемблер на первом просмотре, являются:

1. Исходная программа в виде текстового файла.

2. Счетчик адреса, используемый для слежения за адресом каждой команды.

3. Таблица машинных операций (MOT), содержащая для каждой команды ее мнемонику, длину двоичного машинного кода и формат команды (MOT).

4. Таблица директив (POT), содержащая мнемонику и информацию о действиях, которые должны выполняться для каждой псевдо-команды во время первого просмотра.

5. Таблица символов (ST), используемая для запоминания каждой метки и ее значения.

6. Таблица литералов (LT), используемая для запоминания каждого встреченного литерала и присвоения ему адреса.

В общем виде на первом просмотре ассемблера происходит следующее:

1. Анализируется поле кода операции очередного предложения исходной программы. Если в поле КОП находится директива, то определяется ее тип. Если это директива резервирования памяти, то анализируется поле операнда, чтобы определить необходимое число байтов памяти. Директивы определения базисных регистров никаких действий не вызывают при первом просмотре.

2. Если директива не найдена, то в таблице MOT отыскивается элемент, соответствующий коду операции исходного предложения. Найденный элемент таблицы MOT указывает длину текущей команды. Поле операнда просматривается на наличие литерала. Если найден новый литерал, то он заносится в таблицу литералов для дальнейшей обработки (повтор литералов не допускается). Затем проверяется поле метки. Если метка имеется, то она заносится в таблицу символов ST вместе с текущим значением счетчика адреса. После этого значение счетчика адреса увеличивается на длину команды.

3. Директива END определяет конец. В этом случае литералам присваиваются адреса — операция, похожая на обработку псевдокоманд резервирования памяти.

После того как определены все символы, можно завершить ассемблирование, определив значения кодов операций и полей операндов. В процессе второго просмотра сгенерированные коды должны быть представлены в соответствующем формате для последующей обработки загрузчиком или редактором связей. Кроме того, должен быть напечатан листинг. Для этого необходимо:

1) найти значения символов;

2) сгенерировать команды;

3) сгенерировать данные для директив типа резервирования памяти и литералов;

4) обработать остальные директивы.

Структурами данных, которыми оперирует ассемблер на втором просмотре, являются:

1. Копия исходной программы.

2. Счетчик адреса.

3. Таблица машинных операций, содержащая для каждой команды ее мнемонику, длину двоичного машинного кода и формат команды (MOT).

4. Таблица директив (POT), содержащая для каждой директивы мнемонику и информацию о действиях, которые должны выполняться при втором просмотре.

5. Таблица символов (ST), подготовленная в процессе первого просмотра и содержащая все метки и их значения.

6. Таблица базисных регистров, указывающая, какие регистры в настоящее время определены в качестве базовых и каким должно быть содержимое этих регистров (BT).

7. Рабочее поле INST, необходимое для конкатенации команд из отдельных частей (двоичного кода операции полей регистров, поля смещения и т.п.).

8. Рабочее поле PRINT_LINE, используемое для печати листинга.

9. Выходной файл объектной программы в формате, пригодном для загрузки или редактирования.

При втором просмотре происходит следующее:

1. Как и в первом просмотре, анализируется поле кода операции. Вначале код операции отыскивается в таблице POT, а если код директивы не найден, — то в таблице MOT. Найденный элемент таблицы MOT определяет длину, двоичный код операции и тип формата команды.

Проще всего обрабатываются команды формата «регистр—регистр». В поле операндов заносятся просто номера регистров. Для команд формата «регистр—память» для операнда, адресующего память, вычисляется действительный адрес относительно счетчика команд или базисного регистра. В любом случае ассемблер должен

вычислить смещение, которое является составной частью обычной команды. Оно вычисляется так, чтобы после сложения с регистром счетчика команд (LC) или базисным регистром (B) получить требуемый целевой адрес TA. При этом должно выполняться следующее условие:

$$|D| = TA - ([LC] \cup [B]) < 2^n, \quad (1.1)$$

где D — смещение;
TA — целевой адрес;
LC — счетчик команд;
B — базисный регистр;
n — число разрядов в команде для адресации памяти.

При адресации относительно базисного регистра должно быть

$$0 <= D <= 2^{n-1}. \quad (1.2)$$

В случае адресации относительно счетчика команд:

$$-2^{n-1} <= D <= 2^{n-1} - 1. \quad (1.3)$$

Следует отметить, что директивы указания базисного регистра не переводятся в выполняемые машинные команды. Для загрузки базисного регистра требуемым значением необходимо в программе предусмотреть соответствующие команды.

Смещение для относительных способов адресации вычисляется следующим образом.

В большинстве компьютеров во время вычисления команд счетчик продвигается вперед после выбора очередной команды, но до ее выполнения (рис. 1.5). Поэтому смещение вычисляется по формуле

$$D = \langle\text{адрес метки из ST}\rangle - (LC + L),$$

где L — длина текущей команды.

Значение смещения D, вычисленного таким способом, заносится в поле операнда, адресующего память. После вычисления смещения проверяется условие (1.3). Если оно не выполняется, то используется режим полной адресации (замечание: отрицательное смещение представляется в дополнительном коде).

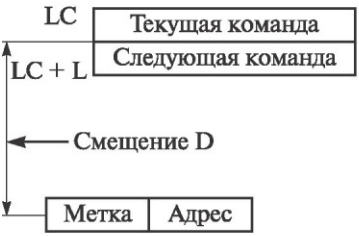

Рис. 1.5. Схема вычисления смещения

При адресации относительно базисного регистра известно его содер-

жимое. Для вычисления смещения используется следующая формула:

D = <адрес метки из ST> – <содержимое базисного регистра>.

В базисном регистре должно, в свою очередь, находиться некоторое значение: либо начало программной акции (относительно нулевого значения), либо начало какой-либо области памяти. После вычисления проверяется условие (1.2). Если оно не выполняется, необходимо использовать расширенный режим адресации (указание полного адреса).

2. Директивы резервирования памяти обрабатываются так же, как и при первом просмотре. Для директив определения констант генерируется действительный код в зависимости от указанных ти-

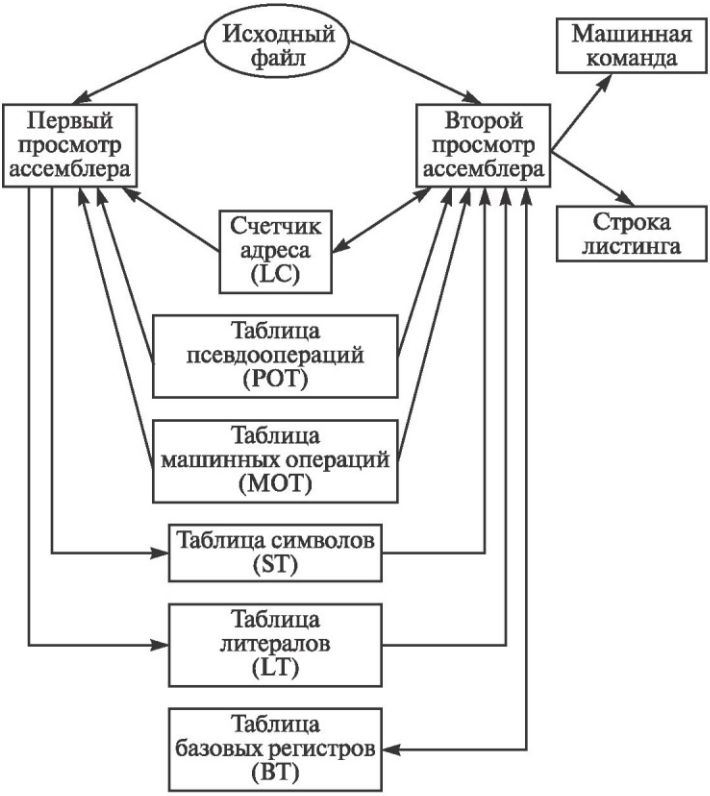

Рис. 1.6. Взаимосвязь первого и второго просмотров ассемблера

пов данных. Для этого требуется вычислить различные преобразования (плавающая точка — двоичное представление и т.п.) и значения символов, например, для адресных констант.

3. После ассемблирования команды и директивы определения констант преобразуются в формат, необходимый для последующей обработки загрузчиком. Обычно несколько команд помещаются в одну запись в выходной объектный файл.

Взаимосвязь первого и второго просмотров показана на рис. 1.6. Таблицы МОТ И РОТ являются фиксированными таблицами. Их содержимое не изменяется в процессе работы ассемблера. Формат этих таблиц определяется архитектурой соответствующего процессора. Более подробные сведения о структурах первого и второго просмотра программы — ассемблер можно найти, например, в работе [D1].

1.2. Макроязык и макропроцессор

Программист, пишущий на ассемблере, часто встречается с необходимостью повторять некоторые последовательности команд в программе несколько раз. Такая последовательность, например, может состоять из команд, служащих для сохранения или изменения содержимого групп регистров, установление ими модульных связей, вычисление некоторых арифметических операций и т.п. В подобных случаях можно воспользоваться аппаратом макрокоманд.

Макрокомандой называется однострочное сокращение для групп команд. Используя макрокоманду, программист по существу определяет одну команду для представления последовательности команд. Вместо каждого вхождения этой однострочной макрокоманды макропроцессор подставит в программу всю последовательность команд.

Макрокоманды обычно рассматриваются как расширение языка ассемблера, а макропроцессор — как расширение основного алгоритма ассемблера. Однако, как особая форма языков программирования, макроязыки существенно отличаются от языков ассемблера и компилируемых языков.

Процесс замены макрокоманд соответствующими последовательностями команд ассемблера, осуществляемый макропроцессором, называется *макрорасширением* или макрогенерацией.

Основной функцией макропроцессора является замена одних групп символов или строк на другие. За исключением некоторых специальных случаев, макропроцессор не анализирует смысл обрабатываемого им текста. На структуру и возможности процессора может повлиять форма операторов используемого языка. Смысл этих предложений и вопросы их трансляции в машинные коды не имеют непосредственного отношения к процессу макрогенерации. Поэтому механизм работы макропроцессора практически не связан с архитектурой аппаратных средств, на которой он должен работать.

1.3. Макрокоманды

В своей простейшей форме макрокоманда представляет собой сокращение для обозначения последовательности операций. Использование макрокоманды позволяет часть программы, которая должна была бы записываться в несколько строк, представить одной строкой. Например, если в исходном тексте программы на ассемблере в разных местах несколько раз встретилась следующая последовательность операторов, то с помощью аппарата макрокоманд можно присвоить ей имя и использовать это имя вместо нее:

```
add ax, 258
mov bx, ax
xor ax, ax
------------
add ax, 360
mov bx, ax
xor ax, ax
------------
add ax, 100
mov bx, ax
xor ax, ax
```

Имя последовательности команд присваивается с помощью определения макрокоманды, имеющего формат:

```
имя   macro [параметры]
      последовательность
      команд
      ассемблера
endm
```

В рассматриваемом примере можно объявить следующую макрокоманду:

```
M_1  macro x
     add     ax, x
     mov     bx, ax
     xor     ax, ax
endm
```

Тогда вместо последовательностей операторов ассемблера можно трижды написать обращение к макрокоманде M_1 с соответствующими параметрами:

```
M_1  258
M_1  360
M_1  100
```

При обработке исходной программы строка M_1 258 будет преобразована в последовательность ассемблерных операторов

```
add   ax, 258
mov   bx, ax
xor   ax, ax
```

Таким образом, несколько строк ассемблерной программы записываются одной строкой.

Процесс преобразования M_1 x в соответствующие строки называется макрорасширением или макрогенерацией. Макрокоманды фактически расширяют базовый язык ассемблера. Язык ассемблера, в котором допускается использование макрокоманд, называется *макроассемблером.*

Если сравнить макрокоманду с подпрограммой и если их рассматривать как способ представления группы команд в компактном виде, макрокоманда и подпрограмма эквивалентны. Если же их рассматривать с точки зрения использования этой группы одинаковых команд в разных местах программы, то в случае подпрограммы эта группа будет располагаться в одном месте, в результате чего длина программы уменьшится. В случае использования макрокоманды группа команд будет переписываться при каждом обращении к ней. Поэтому длина программы будет увеличиваться. Однако при использовании подпрограммы требуется время на ее вызов и последующее возвращение в основную программу. При использовании макрокоманды такие потери отсутствуют.

Макрокоманды, как и подпрограммы, могут иметь один или несколько параметров или не иметь их. Макрокоманды могут храниться в макробиблиотеках, имеющих вид исходных программ на ассемблере. Так, ассемблер i386 является типичным макроассемб-

лером (masm). Ассемблер VAX — тоже макроассемблер (mac). Процессор обработки макрокоманд фактически представляет собой отдельный языковой (пре) процессор со своим собственным языком (макропроцессор).

1.4. Функции макропроцессора

Любой процессор макрокоманд должен решать следующие четыре основные задачи:

1. Распознавать макроопределения, выделяемые псевдокомандами MACRO и ENDM (MEND). Эта задача может усложняться, когда макроопределения встречаются внутри других макроопределений. Когда макроопределения являются вложенными, макропроцессор должен распознать вложения и правильно сопоставить последнюю (внешнюю) команду ENDM с первой командой MACRO. Весь промежуточный текст, включая вложенные MACRO и ENDM, определяют отдельную макрокоманду.

2. Запоминать определения. Макропроцессор должен запомнить определения макрокоманд, которые он будет использовать для расширения макровызовов.

3. Распознавать вызовы. Макропроцессор должен распознавать макровызовы, представленные в форме мнемонического кода операции. Это предполагает, что имена макрокоманд обрабатываются как один из типов кода операции.

4. Выполнять расширения макрокоманд и подстановку фактических параметров. Вместо формальных параметров макроопределения макропроцессор должен подставить соответствующие операнды макрокоманды: результирующий текст на языке ассемблера затем подставляется вместо макрокоманды. Этот текст, в свою очередь, может содержать макрокоманды и макроопределения.

Таким образом, макропроцессор должен распознавать и обрабатывать макроопределения и макрокоманды.

Самым простым способом построения макропроцессора является двухпросмотровый алгоритм. Пусть макропроцессор функционально не зависит от ассемблера и его выходной текст будет передаваться ассемблеру. Пусть макровызовы или макроопределения запрещены внутри макроопределений.

Макропроцессор, как и ассемблер, просматривает и обрабатывает строки текста. В языке ассемблера строки связаны адресаци-

ей: одна строка может ссылаться на другую с помощью ее адреса или имени, которое должно быть известно ассемблеру. Более того, адрес, присваиваемый каждой строке, зависит от предшествующих строк, от их адресов и, возможно, от их содержимого. Если рассматривать макроопределения как единые объекты, то можно сказать, что строки макроязыка не так тесно взаимосвязаны. Макроопределения не могут ссылаться на объекты, находящиеся вне этого макроопределения, а макровызовы ссылаются только на макроопределения.

Двухпросмотровый макропроцессор на первом просмотре отыскивает макроопределения, а на втором — макрокоманды. Так же как ассемблер не может обрабатывать ссылку на символ до ее определения, так и макропроцессор не может выполнять расширенные макрокоманды до того, как будут найдены и заполнены соответствующие макроопределения. Отсюда и следует необходимость двух просмотров исходного текста — один для обработки макроопределений, а второй — для обработки макрокоманд. Во время первого просмотра проверяется каждый код операции, все макроопределения запоминаются в таблице макроопределений MLT, а копии исходного текста без макроопределений запоминаются в рабочем файле для использования во втором просмотре. Кроме того, во время первого просмотра формируется таблица имен MNT.

Во время второго просмотра проверяется каждый мнемонический код операции, и каждая макрокоманда заменяется соответствующим текстом из макроопределения.

Во время первого и второго просмотров макропроцессор использует следующие структуры данных:

1) текст исходной программы в виде файла;

2) таблицу макроопределений MDT, необходимую для хранения тела макроопределений;

3) таблицу имен (MNT), необходимую для хранения имен макрокоманд, для которых имеются макроопределения;

4) счетчик таблицы макроопределений MDTC, указывающий на следующий сводный элемент в MDT;

5) счетчик таблицы имен MNTC, указывающий на следующий сводный элемент в MNT;

6) массив списка параметров ALA, используемый для подстановки индексных маркеров вместо формальных параметров перед запоминанием определения.

Массив списка параметров ALA служит для упрощения дальнейшей подстановки фактических параметров во время макрорасширения. Формальные параметры в макроопределении заменяются позиционными индикаторами: *i*-й формальный параметр в имени макро представляется в теле макроопределения символом индексного маркера #i, где # — символ, зарезервированный макропроцессором (т.е. запрещенный в качестве использования в символических именах). Эти символы используются вместе со списком параметров, который готовится перед расширением макрокоманды.

Символические формальные параметры сохраняются в имени макро для того, чтобы макропроцессор мог выполнять замену параметров по имени, а не по позиции.

Контрольные вопросы

1. Какие особенности языка ассемблера приводят к необходимости построения двухпросмотрового ассемблера?

2. Для нижеприведенной программы на ассемблере напишите эквивалентную программу на машинном языке Intel-386:

```
SWAPW Proc
      les di,PAR1
      mov dl,byte ptr es:[di]
      mov al,dl
      les di,PAR2
      mov dl,byte ptr es:[di]
      stosb
      mov al,dl
      les di,PAR1
      stosb
      ret
SWAPW endp
CODE ends
      end
```

3. В языке ассемблера часто удобно иметь возможность предоставлять первому просмотру программы-ассемблер символы из таблицы символов до того, как завершится сам просмотр. Например, в директиве A EQU B необходимо знать значение B для того, чтобы получить значение A. Предложите метод запоминания данных в таблице символов, чтобы удовлетворить этому требованию.

Глава 2. ЯЗЫКИ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

Для построения транслятора язык высокого уровня обычно описывается в терминах некоторой грамматики. Эти грамматики определяют форму (синтаксис) допустимых предложений языка. Например, оператор присваивания может быть определен в грамматике как имя переменной, за которой следует оператор присваивания (=, или : =), за которым следует выражение. Проблема компиляции может быть сформулирована как проблема поиска соответствия написанных программистом предложений структурам, определенным грамматикой, и генерации соответствующего кода для каждого предложения. Сказанное позволяет определить основные функции и структуру компилятора (транслятора с языка высокого уровня).

Предложения исходного языка удобно представлять в виде последовательности лексем, а не как строку символов. *Лексемы* можно понимать как фундаментальные элементы, из которых состоит язык. Например, лексемой может быть ключевое слово, имя переменной, арифметический оператор и т.п. Просмотр исходного текста, распознавание и классификация различных лексем называется *лексическим анализом*. Каждая лексема состоит из двух частей: класса и значения. Первая часть означает, что лексема принадлежит одному из конечного множества классов, и указывает характер информации, включенной в значение лексемы.

Так, конкретная выделенная переменная принадлежит классу «переменные» и имеет значение, которое служит указателем на элемент таблицы имен (идентификаторов). Такой указатель на таблицу имен является, по существу, внутренним именем выделенной переменной. *Литерал*, или константа, принадлежит классу «константы» и имеет в качестве значения набор битов, представляющих выделенную константу в памяти. Знаки операций относятся к классу «операции», а их значениями могут быть, например, старшинство операций. Некоторые лексемы, например, принадлежащие клас-

су «ключевые слова», не требуют информации о значении. Часть транслятора, выполняющая эту задачу, называется *сканером.*

Как только лексемы выделены, каждое предложение программы может быть распознано как некоторая конструкция языка, например, как декларативные операторы или оператор присваивания, описанные с помощью грамматики. Процесс, называемый синтаксическим анализом или синтаксическим разбором, осуществляется частью транслятора, которая называется *синтаксическим анализатором* (parser). Этот блок фактически переводит последовательность лексем, построенную лексическим блоком, в другую последовательность, которая непосредственно отражает порядок, в котором, по замыслу программиста, должны выполняться операторы в программе. Например, если программист написал оператор

A = A + B*C,

то он подразумевает, что числа, представленные идентификаторами B и C, должны быть перемножены, и результат умножения будет добавлен к числу, представленному идентификатором A. Таким образом, последовательность из 7 лексем, выданная лексическим анализатором, преобразуется в последовательность действий над некоторыми новыми единицами, называемыми *атомами.* Для рассматриваемого примера схематически можно определить следующий порядок действий:

Умножить B на C;

Запомнить Результат;

Сложить Результат с A и записать в A.

Таким образом, выходом синтаксического блока в рассматрваемом случае будет последовательность из трех атомов. В общем случае выходом синтаксического блока является последовательность атомов, отображающая ход выполнения программы. Так, в рассматриваемом примере знак умножения находится после знака сложения, но синтаксический блок вначале помещает действие умножения. Выполняя необходимые преобразования, синтаксический блок должен учитывать структуру языка, которая задается его грамматикой.

Последним шагом базовой схемы процесса трансляции является генерация объектного кода. Последняя заключается в преобразовании последовательности атомов, построенной синтаксическим блоком, в последовательность операторов машинного языка (пос-

ледовательность машинных команд). Первые трансляторы с языков высокого уровня выполняли преобразование последовательности атомов в операторы соответствующего ассемблера, а затем осуществляли вызов программы-ассемблер. Однако в настоящее время большинство трансляторов генерирует непосредственно объектный код, а не программу на языке ассемблера, предназначенную для последующей трансляции в машинные коды ассемблером.

Кроме синтаксиса, большое значение имеет семантика предложений, определяющая их смысл. Рассмотрим следующий пример:

float x,y, int x, y,
x = x + y, x = x + y.

Видно, что в данном случае синтаксис одинаков, но в результате трансляции создается различный код (т.е. семантика).

Таким образом, генератор выполняет некоторую работу, связанную со смыслом (семантикой) лексем, или семантическую обработку. В данном случае семантикой идентификатора является его тип.

Приведем пример грамматического разбора. Пусть задано предложение:

if (a>3) then a:=7,

где if — оператор языка;
then — ключевое слово;
a>3 — условное выражение;
a:=7 — выражение, которое вычисляется, если условное выражение истинно.

С учетом жесткого разделения грамматики и семантики общая схема трансляции представлена на рис. 2.1. Возможны два основных типа взаимодействия между блоками компилятора.

Один тип предполагает, что каждый раз, когда лексический блок выдает лексему, управление передается синтаксическому блоку для обработки лексемы. Когда возникает необходимость в следующей лексеме, управление возвращается в лексический блок.

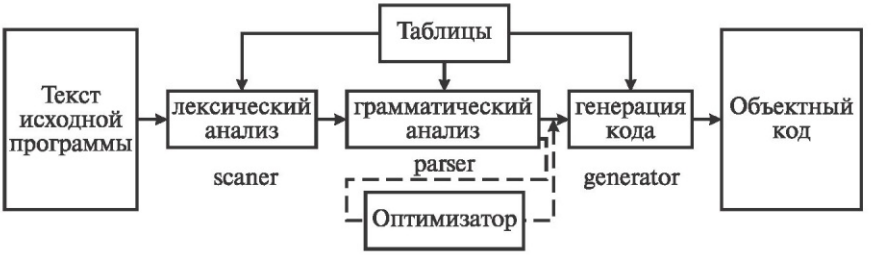

Рис. 2.1. Общая схема трансляции

При другом типе взаимодействия лексический блок выделяет всю цепочку лексем до того, как управление передается синтаксическому блоку. В этом случае говорят, что работа лексического блока образуют отдельный проход.

В схеме трансляции проходы можно организовать четырьмя различными способами. Если транслятор организован как однопросмотровый, управление передается из блока в блок всякий раз, когда лексический блок выдает лексему. Схема может быть организована как трехпроходный компилятор. В этом случае лексический блок подготавливает всю последовательность лексем, которая затем используется синтаксическим блоком при порождении всей последовательности атомов. Последняя, в свою очередь, используется генератором при построении машинного кода.

Возможны также два типа двухпроходной организации. В одном случае лексический и синтаксический блоки работают одновременно в течение одного прохода, подготавливая полную последовательность атомов для генерации кода. В другом случае синтаксический блок и генератор кода объединяются в один проход.

Чем больше блоков в компиляторе, тем больше возможностей для его многопроходной реализации. Разбиение на проходы может привести к дополнительным затратам памяти, так как каждое взаимодействие между проходами требует, чтобы сохранялась вся цепочка выходных символов. Однако разбиение на проходы имеет и преимущества.

1. *Логика языка.* Иногда сам исходный язык наводит на мысль о том, что компилятор должен иметь не меньше двух проходов. Такая потребность возникает, если в какой-то момент компилятору нужна информация из еще непросмотренной части программы. Например, если описание идентификатора или переменной может появиться в тексте программы после их использования, то может оказаться, что код нельзя построить до тех пор, пока не будет обработана вся информация. В этом случае для генерации кода требуется отдельный проход.

2. *Оптимизация кода.* Иногда объектный код получается более эффективным, если генератору кода доступна информация обо всей программе. Например, согласно некоторым методам оптимизации, нужно знать все те места программы, где используются переменные и где могут изменяться значения. Поэтому, прежде чем

начать оптимизацию, необходимо просмотреть всю программу до конца.

3. *Экономия памяти.* Обычно многопроходные компиляторы занимают в памяти меньше места, чем компиляторы с одним проходом, так как код каждого прохода может вновь использовать память, занимаемую кодом предыдущего прохода.

Каждый проход компилятора можно организовать в виде одного блока или комбинации нескольких блоков. С точки зрения теории построения компиляторов блок — это просто часть компилятора, которая строится как одно целое. С этой же точки зрения не имеет значения, будет ли блок реализован как отдельный проход или как часть некоторого прохода. Так, работу лексического анализатора, заключающуюся в порождении лексем, можно рассматривать независимо от того, помещаются ли эти лексемы в промежуточный файл или сразу передаются в синтаксический блок.