Глава 11. Макросредства языка Ассемблер

Афоризмы – это макросы, поскольку они выполняются в ходе чтения.

Алан Перлис

Лень – двигатель Макро.

Edmond/HI-TECH

Сейчас начнём изучение важной и достаточно сложной темы – *макросредств* в языках программирования. Знакомство с этим понятием будет производиться постепенно, используя примеры из макросредств языка Ассемблера.

Вообще говоря, макросредства могут включаться не только в язык Ассемблера, но и в языки *высокого* уровня. Стоит заметить, однако, что не случайно будут изучаться именно макросредства языка Ассемблера, а не макросредства каких-нибудь языков программирования высокого уровня (Паскаля, С и т.д.). Всё дело в том, что макросредства в языках высокого уровня чаще всего примитивны (не развиты), и не обеспечивают всех тех возможностей, которые должны быть изучены в рамках этой темы (потом будет сказано о причинах, почему так получилось). Говоря "по научному", макросредства языков программирования высокого уровня, в отличие от макросредств Ассемблера, не являются *алгоритмически полными*. Именно поэтому для уровня университетского образования приходится рассматривать достаточно развитые макросредства в языке Ассемблера.

Макросредства по своей сути являются *алгоритмическим языком*, встроенным в некоторый другой язык программирования,[1] который в этом случае будет называться *макроязыком*. Например, изучаемые нами макросредства встроены в язык Ассемблера, который поэтому называется Макроассемблером. Таким образом, программа на Макроассемблере в общем случае содержит в себе запись *двух* алгоритмов: один на макроязыке, а второй – собственно на языке Ассемблера.[2]

Как известно, у каждого алгоритма обязательно должен быть свой *исполнитель*. Например, исполнитель алгоритма на языке Паскаль называют Паскаль-машиной, это компьютер вместе с набором служебных программ (стоит вспомнить предыдущую главу "Понятие о системе программирования"). Исполнитель алгоритма на макроязыке называется *Макропроцессором*, а исполнителем алгоритма на Ассемблере является, в конечном счете, компьютер. Здесь важно, чтобы Вы не путали Макропроцессор с процессором компьютера: макропроцессор – это специальная программа (часть компилятора), а не часть аппаратуры ЭВМ.

Результатом работы Макропроцессора (будем пока считать, что этот исполнитель работает *первым*, а компилятор с Ассемблера – вторым) является программный модуль на "чистом" языке Ассемблера, уже *без* макросредств. В связи с этим заметим, что в некоторых языках программирования (например, в языке С), макропроцессор так и называется – *препроцессор*, чтобы подчеркнуть, что при обработке программы он выполняется первым.[i] [см. сноску в конце главы] Иногда говорят, что Макропроцессор, получив входной модуль на языке Макроассемблера, *генерирует* модуль на чистом Ассемблере, что хорошо отражает суть дела. Надо подчеркнуть, что компилятор Ассемблера относится к трансляторам (переводчикам) с языка Ассемблер на язык объектных модулей. В то же время Макропроцессор не стоит называть переводчиком с языка Макроассемблера на язык Ассемблера, лучше считать, что текст на языке Ассемблера получается не в процессе перевода, а путем *генерации* этого текста как выходных данных алгоритма, заданного макросредствами.

На рис. 11.1 показана упрощенная схема работы Макропроцессора и компилятора с Ассемблера (как уже говорилось, их часто называют общим именем – Макроассемблер). Программные модули пользователя на этом рисунке заключены в прямоугольники, а системные программы – в прямоугольники с закруглёнными углами.

---

[1] Макросредства могут быть встроены также и в формальные языки, не являющиеся языками программирования. Например, опытные пользователи компьютера должны уметь пользоваться, скажем, макросредствами современных текстовых редакторов, скажем, в широко распространённом текстовом процессоре Word.

[2] Для продвинутых читателей. Это частный случай так называемого *метапрограммирования*, когда целевая программа не пишется программистом полностью на своём языке программирования, а, частично или полностью, получается (генерируется) в результате выполнения другого алгоритма, встроенного в основную программу.

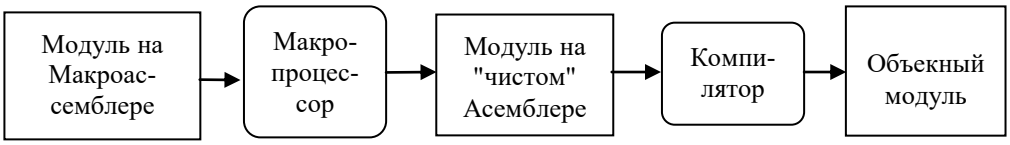

Рис. 11.1. Упрощённая схема работы Макроассемблера.

Работа Макропроцессора по обработке макросредств исходной программы называется *макропроцессированием*. Изучение макросредств языка Ассемблера будет начато с уже знакомых нам *макрокоманд* или макровызовов (macro instruction). До сих пор говорилось, что при компиляции программы на Ассемблере на место макрокоманды по определённым правилам подставляется некоторый набор предложений языка Ассемблер. Теперь пришло время подробно изучить, как это делается.

Каждое предложение Ассемблера, являющееся макрокомандой, имеет обязательное поле – код операции, который является *именем* макрокоманды. Именно по коду операции Макропроцессор будет определять, что это именно макрокоманда, а не какое-нибудь другое предложение языка Ассемблера. Коды операций макрокоманд являются *именами пользователя*, а все остальные коды операций – *служебными именами*.[1] Кроме того, у макрокоманды есть (возможно, пустой) список фактических параметров. Ниже приведён общий вид макрокоманды:

```
[<метка>:] <имя> [<список фактических параметров>]
```

Если у макрокоманды есть метка, то считается, что эта метка задаёт отдельное предложение Ассемблера, состоящее только из данной метки. Другими словами, макрокоманда с меткой:

```
<метка>: <имя> [<список фактических параметров>]
```

эквивалентна двум таким предложениям Ассемблера:

```
<метка>:
        <имя> [<список фактических параметров>]
```

Обратите внимание, что метка не может быть без двоеточия, которое, как известно, задает в нашем Ассемблере имя *команды*. Это совсем не означает, что макрокоманды не могут располагаться, например, в секции данных, просто тогда у них не может быть никакой метки. В случае, если макрокоманда располагается в секции данных, то на её место будет, вероятно, подставляться набор предложений Ассемблера, не являющихся командами (например, это могут быть предложения резервирования памяти или директивы). Пример такой макрокоманды будет рассмотрен в конце этой главы.

Итак, каждая макрокоманда обязательно имеет имя. Обработка макрокоманды Макропроцессором начинается с того, что он, просматривая текст модуля от данной макрокоманды по направлению к началу программы (*снизу вверх*), ищет специальную конструкцию Макроассемблера, которая называется *макроопределением* (macro definition). Отметим, что на жаргоне программистов часто и макрокоманда, и соответствующее ей макроопределение называют *макросом*.[2]

Каждое макроопределение тоже имеет имя, и поиск заканчивается, когда Макропроцессор находит макроопределение с тем же именем, что и у макрокоманды. Здесь надо сказать, что не считается ошибкой, если в программе будут несколько одноимённых макроопределений, Макропроцессор выбирает из них первое, встреченное при просмотре программы *снизу вверх* от данной макрокоманды. Если в программе есть несколько макроопределений с одинаковыми именами, то говорят, что новое макроопределение *переопределяет* одноимённое макроопределение, описанное ранее.[3]

---

[1] В примерах программ для удобства чтения имена макрокоманд выделялись жирным шрифтом (**`exit`**, **`inint`** и т.д.), хотя это, конечно, неправильно, так как это не служебные слова Макроассемблера, а имена пользователя.

[2] По гречески μακρός – длинный, большой, обширный; соответственно макрокоманда – "большая" команда, на её место подставляется целый набор предложений Ассемблера (хотя, возможно, и пустой 😕).

[3] Этот удобный механизм позволяет, например, заменять некоторое макроопределение, вставляемое в программу по директиве **`inclule`** `console.inc`. Так, если программисту не понравится, как работает, например, макроопределение **`inint`**, он может написать своё собственное макроопределение с таким же именем и вставить его в текст своей программы за директивой **`inclule`** `console.inc`. Заметим также, что это один из немногих случаев, когда в одном ассемблерном модуле могут встречаться описания одинаковых имён.

Макроопределение в нашем Макроассемблере имеет следующий синтаксис:

```
<имя> macro [<список формальных параметров>]
        Тело макро-
        определения
        endm
```

Первая строка является директивой – *заголовком* макроопределения, она определяет его имя и, возможно, список *формальных параметров*. Список формальных параметров – это (возможно пустая) последовательность *имён*, разделённых запятыми, причём каждое имя может содержать некоторый ключ, уточняющий использование этого параметра. Имена формальных параметров локализованы в теле макроопределения. Тело макроопределения – это набор (возможно пустой) предложений языка Ассемблера (среди них, в свою очередь, могут быть и предложения, относящиеся к макросредствам языка). Заканчивается макроопределение директивой **endm** (обратите внимание, что у этой директивы нет метки (имени макроопределения), как, скажем, у директивы конца описания процедуры). Попытка поставить это имя рассматривается как рекурсивный вызов этого макроопределения.

Макроопределение может находиться в любом месте программы до первой макрокоманды с таким же именем, но хорошим стилем программирования считается описание всех макроопределений в начале программного модуля. Более того, макроопределения, посвященные одной тематике можно объединить в один текстовый файл и хранить его *отдельно* от программных модулей. По аналогии с библиотеками объектных модулей, такие файлы часто называются библиотеками макроопределений. Например, для примеров программ из данной книги макроопределения объединены в библиотеку макроопределений, которая хранится в текстовом файле с именем `io.inc`. Для того, чтобы вставить текст этого файла в программный модуль и, таким образом, сделать содержащиеся в этом файле макроопределения доступными для макрокоманд ввода/вывода, используется директива Ассемблера **include**.[1]

Итак, каждой макрокоманде должно быть поставлено в соответствие макроопределение с таким же именем, иначе в программе фиксируется синтаксическая ошибка (неописанное имя). Макроассемблер допускает *вложенность* одного макроопределения внутрь другого, однако имя вложенного макроопределения становится для макропроцессора *видимым* только после первого вызова *внешнего* макроопределения, поэтому такая вложенность редко используется в практике программирования. Вполне допустимы и рекурсивные макроопределения.

Далее, в макрокоманде на месте поля операндов может задаваться список *фактических параметров*. Например, можно написать макрокоманду `outint X,10` с двумя фактическими параметрами. Как видно, здесь просматривается большое сходство с механизмом процедур, например, в языке Паскаль, где в *описании процедуры* мог задаваться список формальных параметров, а в *операторе процедуры* – список фактических параметров. Однако на этом внешнее сходство между процедурами в Паскале и макроопределениями в Макроассемблере заканчивается, и начинаются различия в виде, способах передачи и обработки параметров.

Фактические параметры, если их более одного, разделяются запятыми.[2] Каждый фактический параметр макрокоманды является *строкой символов* (возможно пустой). Эта строка должна быть сбалансирована по апострофам, кавычкам и угловым скобкам. Если фактический параметр расположен не в конце списка параметров и является пустой строкой, то его позиция просто выделяется запятыми, например:

```
Mymacro X,,Y; Три фактических параметра, второй пустой
Mymacro ,A,B; Три фактических параметра, первый пустой
```

---

[1] В самом включаемом файле могут находится и другие директивы **include**, например, в файле `concole.inc` находится директива **include** `io.inc`.

[2] Запятая, угловые скобки, точка с запятой (и некоторые другие символы) являются служебными и не могут просто так встречаться внутри фактического параметра. При необходимости вставить служебный символ внутрь параметра макрокоманды используются особые приёмы, о которых будет говориться далее. В ранних версиях Ассемблера MASM параметры макрокоманды могли разделяться как запятыми, так и пробелами, особых преимущество при программировании это не даёт. Можно включить такой старый режим работы, опцией компилятора **option** OLDMACROS.

Чтобы указать внутри фактического параметра служебные символы (запятую, точку с запятой) весь параметр заключают в угловые скобки, например:

```
Mymacro A,<A,B;C>; Два фактических параметра, остальные пустые
Mymacro <A,B,C,X>; Один фактический параметр, остальные пустые
```

Как видно, в отличие от Паскаля, все параметры макроопределения одного типа – это (возможно пустые) строки символов, другими словами, всегда есть *соответствие по типу* между фактическими и формальными параметрами. Далее, в Макроассемблере, в противоположность языку Паскаль, не должно соблюдаться соответствие в числе параметров: формальных параметров может быть как меньше, так и больше, чем фактических. Если число фактических и формальных параметров не совпадает, то Макропроцессор выходит из этого положения совсем просто. Когда фактических параметров *больше*, чем формальных, то лишние (последние) фактические параметры не используются (отбрасываются), при этом выдаётся *предупредительная* диагностика. В случае, когда фактических параметров не хватает, по последние недостающие фактические параметры считаются *пустыми* строками символов.

Рассмотрим теперь, как Макропроцессор обрабатывает (*выполняет*) макрокоманду. Сначала, как уже говорилось, он ищет соответствующее макроопределение, затем начинает передавать фактические параметры (строки символов, возможно пустые) на место формальных параметров (имён). В Паскале, как известно, существуют два способа передачи параметров – по значению и по ссылке. В Макроассемблере реализован другой (третий) способ передачи фактических параметров макрокоманды в макроопределение, его нет в Паскале. Этот способ называется передачей *по написанию* (иногда говорят – передачей *по имени* или *по подстановке*). При таком способе передачи параметров все *имена* формальных параметров в теле макроопределения заменяются соответствующими им фактическими параметрами (строками символов).[1] Возможна замена формального параметра на фактический даже внутри текстовых строк, правда, для этого имя формального параметра необходимо выделить специальным макроограничителем &, например:

```
P macro X
  db    "Параметр X=&X"
endm
```

При вызове

```
P ABC
```

Будет получено

```
    db "Параметр X=ABC"
```

После передачи параметров начинается просмотр тела макроопределения и поиск в нём предложений, содержащих макросредства, например, макрокоманд. Все предложения в макроопределении, содержащие макросредства, обрабатываются Макропроцессором так, что в результате получается набор предложений на "чистом" языке Ассемблера (уже без макросредств), такой набор предложений называется *макрорасширением* (macro expansion) Последним шагом в обработке макрокоманды является подстановка полученного макрорасширения на место макрокоманды, это действие называется *макроподстановкой* (macro substitution). Далее текст макрорасширения поступает на обработку уже компилятору Ассемблера. На рис. 11.2 показана схема обработки макрокоманды.

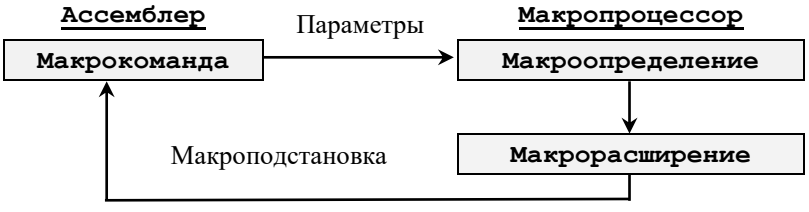

Рис. 11.2. Схема обработки макрокоманды.

Обратите внимание, что в некотором смысле Макропроцессор работает раньше компилятора. Более точно, проанализировав предложение на Ассемблере и определив, что там есть макросредства, компилятор вызывает Макропроцессор. Затем, получив от Макропроцессора результат (макрорасширение), начинает его обработку, например, для фрагмента:

---

[1] Это несколько упрощённое описание действий Макропроцессора при передаче параметров, позже в этом месте будут сделаны существенные уточнения.

```
Number equ 100
P   macro X
    echo X
    endm
    P Number
    outintln Number
```

Директива **echo** на этапе компиляции выводит следующую после неё и пробела часть строки на экран (в поток stdout), туда будет выведено Number, а вот макрокоманда **outintln** выводит 100. Так происходит потому, что компилятор Ассемблера подставляет вместо макроконстанты Number значение 100 уже *после* работы Макропроцессора. Немного позже будет рассказано, как заставить компилятор вычислить значение макроконстанты Number и передать в макроопределение не имя Number, а строку из трёх символов 100.

Из рассмотренного механизма обработки макрокоманд вытекает главное применение этого макросредства при программировании на Ассемблере. Как можно заметить, если необходимо выполнить в программе некоторое достаточно сложное действие, можно идти двумя путями. Во-первых, можно написать *процедуру* и вызывать её, передавая ей фактические параметры. Во-вторых, можно написать *макроопределение*, в теле которого реализовать нужное действие, и обращаться к этому макроопределению по соответствующей макрокоманде, также передавая необходимые параметры.

В дальнейшем эти два метода будут сравниваться, а пока надо отметить, что написание макроопределений – это хороший способ *повысить уровень* языка программирования. Действительно, макрокоманда по синтаксису практически ничем не отличается от обычных команд Ассемблера, но, в отличие от них, может задавать весьма сложное действие. Вспомним, например, макрокоманду **inint** для ввода целого значения. Соответствующее ей макроопределение по своим функциям похоже на процедуру Read языка Паскаль и реализует достаточно сложный алгоритм по преобразованию вводимых символов во внутреннее представление целого числа. С точки же зрения программиста в языке Ассемблера как бы появляется новая машинная *команда*, предназначенная для ввода целых чисел. Эта новая команда имеет свой формат и даже может вырабатывать флаги. Говорят, что при помощи макросредств можно *расширить* язык Ассемблера, как бы вводя в него *новые* команды, необходимые программисту. Таким образом, уровень языка Ассемблера, можно, по существу, приблизить к языкам высокого уровня. Отметим, что главным при этом является не столько уменьшение числа строк исходного текста программы, сколько обеспечение хорошего стиля программирования.

Теперь пришло время написать первое простое макроопределение и на его основе продолжить изучение работы Макропроцессора. Предположим, что во многих местах программы на Ассемблере находится оператор присваивания вида Z:=X+Y, где X, Y и Z – целочисленные операнды размером в двойное слово (**dword**). В общем случае (при произвольном задании операндов Z, X и Y – в памяти, на регистрах или в виде констант) для реализации этого оператора присваивания необходимы три команды Ассемблера, например, с использованием регистра EAX:

```
mov eax,X
add eax,Y
mov Z,eax
```

Естественно, что программисту было бы более удобно, если бы в языке Ассемблера существовала *трёхадресная* команда, которая реализовывала бы такой оператор присваивания. Пусть это будет, например, команда с кодом операции Sum:

```
Sum Z,X,Y; Z:=X+Y
```

Сначала надо уточнить *синтаксис* новой команды, т.е. зафиксировать допустимые форматы её операндов. Потребуем, чтобы первый операнд этой команды мог иметь форматы r32 и m32, а второй и третий операнды – форматы r32,m32 и i32 (в любой комбинации). Такой команды в машинном языке рассматриваемого компьютера нет, но можно создать новую *макрокоманду*, которая работала бы именно так.[1] Для этого можно написать, например, такое макроопределение:

---

[1] Любопытно отметить, что такой команды не было и в рассмотренной ранее учебной трёхадресной машине УМ-3, где допускался только прямой способ адресации, т.е. все операнды команды могли быть только адресами (в нашей терминологии m32). В то же время новая макрокоманда сложения допускает *смешанную* адре-

```
Sum macro Z,X,Y
    mov eax,X
    add eax,Y
    mov Z,eax
endm
```

Вот теперь, если в программе есть, скажем, описания переменных

```
A  dd ?
B  dd ?
C  dd ?
```

и надо выполнить присваивание `C:=A+B`, то программист может записать это действие в виде одного предложения Ассемблера – макрокоманды

```
Sum C,A,B
```

Увидев такую макрокоманду, Макропроцессор, просматривая текст программы снизу-вверх, найдёт соответствующее макроопределение с именем `Sum`. Затем, после замены формальных параметров на фактические, Макропроцессор построит следующее макрорасширение:

```
mov eax,A
add eax,B
mov C,eax
```

Это макрорасширение и будет подставлено в текст ассемблерной программы вместо макрокоманды `Sum C,A,B` (произойдёт *макроподстановка* полученного макрорасширения на место макрокоманды). Заметим, что эта макрокоманда, как и "настоящая" команда сложения, правильно устанавливает все флаги (`OF`, `CF`, `ZF` и `SF`).

Программист доволен: теперь *исходный* текст его программы значительно сократился, и программа стала более понятной. Таким образом, можно приблизить уровень языка Ассемблер (это язык *низкого* уровня) к языку высокого уровня (например, Паскалю). В этом, как уже говорилось, и состоит одно из назначений механизма макроопределений и макрокоманд – поднять уровень языка, в котором они используются. Кроме того, можно заметить, что с помощью макрокоманд как бы меняется *архитектура* нашего компьютера, например, появляется трёхадресная команда сложения, которой на самом деле в языке машины нет.

Далее, однако, программист может заметить, что некоторые вызовы макрокоманды `Sum` работают не совсем хорошо. Например, на место макрокоманды с *допустимым* форматом параметров

```
Sum C,eax,B; допустимый формат m32,r32,m32
```

будет подставлено макрорасширение

```
mov eax,eax
add eax,B
mov C,eax
```

Первая команда в этом макрорасширении, хотя и не влияет на правильность результата, но явно лишняя и портит всю картину. Естественно, так как истинный программист ценит в программах красоту и компактность 🙂, то хотелось бы убрать из макрорасширения первую команду, если второй операнд макрокоманды является регистром `EAX`. Другими словами, хотелось бы делать *условную макрогенерацию* и давать Макропроцессору указания вида "если выполняется такое-то условие, то вставляй в макрорасширение вот эти предложения Ассемблера, а иначе – не вставляй". На языке Паскаль такие указания записываются в виде *условных операторов*. Ясно, что и в макроязыке (а это тоже алгоритмический язык) также должны допускаться аналогичные *условные макрооператоры*.

В Макроассемблере условные макрооператоры принадлежат к средствам так называемой *условной компиляции*. Легко понять смысл этого названия, если вспомнить, что при выполнении таких макрооператоров меняется вид компилируемой программы на Ассемблере, в ней появляются (или не появляются) те или иные группы предложений. В этой книге будут изучены только самые употребительные макрооператоры, которые будут использоваться в наших примерах, для полного изучения этой темы необходимо обратиться, например, к учебникам [5-7].

сацию, когда второй и третий операнды могут быть регистровыми, прямыми или непосредственными (`r32`, `m32` или `i32`). Таким образом, с помощью макросредств можно в определённом смысле менять *архитектуру* ЭВМ, например, ввести в язык Ассемблера команды трёхадресной или безадресной (стековой) ЭВМ.

Итак, необходимо вставить в макроопределение условный макрооператор с таким смыслом:

"Если второй фактический параметр X не идентичен
(не совпадает) с именем регистра eax, то тогда необходимо
вставить в макрорасширение предложение **mov** eax, X".

На Макроассемблере наше макроопределение с именем Sum в этом случае будет иметь такой вид:

```
Sum macro Z,X,Y
  ifdif <X>,<eax>
      mov eax,X
  endif
      add eax,Y
      mov Z,eax
endm
```

Поясним работу условного макрооператора с именем **ifdif** (**if dif**ferent). Так как его аргументы – строки символов, т.е. он проверяет совпадение или несовпадение двух строк текста, то надо как-то задать эти строки. В качестве ограничителей строки в Макроассемблере выбраны угловые скобки, так что выражение <eax> эквивалентно записи 'eax' в Паскале. Таким образом, семантику нашего условного макрооператора на языке Паскаль можно записать как

```
if X<>'eax' then
```
    Вставить в макрорасширение **mov** eax, X

Теперь Макропроцессор для макрокоманды **Sum** C, eax, B не будет вставлять в макрорасширение ненужной команды **mov** eax, eax.

Изучая дальше это макроопределение можно заметить, что и на место, например, макрокоманды

```
Sum eax,eax,13
```

с допустимыми форматами операндов r32,r32,i32 подставится макрорасширение

```
add eax,13
mov eax,eax
```

с лишней последней строкой, что тоже, хотя формально и правильно, но некрасиво. Кроме того, макрокоманда всегда портит регистр EAX, что может быть недопустимо, если параметр Z отличается от регистра EAX, например, **Sum** ebx, ecx, 13. Чтобы это исправить, придётся снова изменить макроопределение, например, так:

```
Sum macro Z,X,Y
  ifdif <Z>,<eax>
      push eax;   запомнить EAX
  endif
  ifdif <X>,<eax>
      mov eax,X
  endif
      add eax,Y
  ifdif <Z>,<eax>
      mov Z,eax
      pop eax;    восстанов. EAX
  endif
endm
```

Вот теперь на место макрокоманды

```
Sum eax,eax,13
```

будет подставляться ну о-о-очень хорошее макрорасширение

```
add eax,13
```

Дальнейшее изучение работы написанного макроопределения, однако, выявит новую неприятность: макрокоманда

```
Sum eax,B,eax
```

с *допустимыми* форматами операндов r32,m32,r32 порождает *неправильное* макрорасширение

```
mov eax,B
add eax,eax
```

Легко понять, что это EAX:=B+B, вместо требуемого по смыслу EAX:=EAX+B. Как можно заметить, источник неприятности здесь состоит в том, что третий параметр макрокоманды *испортился* до того, как его использовали по назначению. С этой проблемой тоже можно справиться, снова усложнив макроопределение, например, так:

```
Sum macro Z,X,Y
  ifdif <Z>,<eax>
      push eax;   запомнить EAX
  endif
  ifidn <Y>,<eax>
      add eax,X
  else
    ifdif <eax>,<X>
      mov eax,X
    endif
      add eax,Y
  endif
  ifdif <Z>,<eax>
      mov Z,eax
      pop eax;    восстанов. EAX
  endif
endm
```

В новой версии макроопределения использовались *вложенные* условные макрооператоры. Первый из них с именем **ifidn** (**if id**entical) сравнивает свои аргументы-строки текста и вырабатывает истинное значение, если они идентичны (равны). Как и в условном операторе языка Паскаль, в условном макрооператоре может присутствовать ветвь **else**, которая выполняется, если при сравнении строк получается ложное значение.

Для последнего примера вызова макрокоманды `Sum eax,B,eax` сейчас получается правильное макрорасширение

```
add eax,B
```

Не нужно, конечно, думать, что теперь написано идеальное макроопределение и все проблемы решены. Первая неприятность, которая здесь подстерегает, связана с самим механизмом сравнения строк на равенство и неравенство в условных макрооператорах. Рассмотрим, например, что будет, если записать в программе макрокоманду

```
Sum EAX,X,Y
```

На её место будет подставлено макрорасширение

```
mov eax,X
add eax,Y
mov EAX,eax
```

с лишней последней строкой. Причина здесь в том, что Макропроцессор, естественно, считает строки текста <EAX> и <eax> *не идентичными*, так как полагает внутри строки текста большие и маленькие буквы различными, со всеми вытекающими отсюда последствиями. В то же время служебные имена регистров EAX и eax в языке Ассемблера считаются *идентичными*, и нет причин ограничивать пользователя в написании таких имён, как ему захочется. С этой проблемой можно справиться, приказав Макропроцессору сравнивать строки без учёта регистра (при этом большие и маленькие латинские буквы не различаются), например, для

```
ifidni <eax>,<EAX>
```

строки уже будут считаться одинаковыми (идентичными).

Другая трудность подстерегает программиста, если он попытается использовать в своей программе, например, макрокоманду с неверным синтаксисом (недопустимыми типами параметров)

```
Sum eax,ebx,dl
```

После обработки этой макрокоманды будет построено макрорасширение

```
mov eax,ebx
add eax,dl
```

Это макрорасширение Макропроцессор "со спокойной совестью" подставит на место нашей макрокоманды. Конечно, позже, когда компилятор Ассемблера будет анализировать синтаксическую правильность этих предложений, для команды `add eax,dl` зафиксируется ошибка – несоответст-

вие типов операндов. Это очень важный момент – ошибка зафиксирована не при обработке Макропроцессором *неправильной* макрокоманды `Sum eax,ebx,dl`, как происходит при проверке синтаксической правильности обычных команд Ассемблера, а позже, и уже при анализе не макрокоманды, а макрорасширения. В этом отношении наша макрокоманда *уступает* обычным командам Ассемблера, так как по аварийной диагностике будет труднее определить место и характер ошибки. Хотелось бы, конечно, чтобы диагностика об ошибке (и лучше на русском языке) выдавалась уже на этапе обработки макрокоманды Макропроцессором. Например, для неправильной макрокоманды

```
Sum eax,ebx,dl
```

Для программиста много лучше получить диагностику Макропроцессора

```
*** Macro Sum *** Bad Type of 3-rd Argument
```

чем такую диагностику Ассемблера:

```
add eax,dl
asm(46) : error A2070: invalid instruction operands
```

Итак, нужно уметь как-то определять, что у макрокоманды заданы плохие по типу параметры, скоро будет рассказано, как это сделать.

Теперь следует рассмотреть типичный ход мыслей программиста на Ассемблере. Когда у него может возникнуть необходимость в написании новой макрокоманды? Например, во многих программах необходимо вычислять абсолютное значение, в Паскале для этого используется стандартная функция `abs(x)`. Пусть программист решил реализовать новую команду, написать макроопределение `abs macro x`.

Сначала надо определить допустимые форматы операнда. Хорошим решением будет разрешить *знаковому* параметру `x` принимать значения в нескольких полезных для программиста форматах. Напомним, что в языке Free Pascal стандартная функция `abs(x)` является полиморфной, её параметр может быть значением любого (знакового) целого (`shortint`, `integer`, `longint`, `int64`) и вещественного (`single`, `real`, `double` (`int64`), `extended`) типов. Пусть, например, программист решил, что параметр макрокоманды может быть целым знаковым значением форматов `r8`, `r16`, `r32`, `m8`, `m16`, `m32` и `i32`, а результат возвращаться на регистре `EAX`. Ясно, что макроопределение **`abs`** должно быть полиморфным и *настраиваться* на тип переданного параметра.

Рассмотрим механизм настройки макроопределения на *типы* передаваемых ему фактических параметров. Здесь имеется в виду, что хотя с точки зрения Макропроцессора фактический параметр – это просто строка символов, но с точки зрения самого компилятора Ассемблера, если параметр содержит некоторое *имя*, то у этого параметра может быть *тип*. Например, фактическим параметром может быть *имя* длинной или короткой целой переменной, константы, регистра и т.д. и ясно, что для обработки операндов разных типов требуются и различные форматы команд. Другими словами, программисту может понадобиться внутри макроопределения выбрать (средствами условной генерации) один из нескольких форматов команды для обработки операндов разных типов, переданных в это макроопределение в качестве фактических параметров.

Надо также определить, что делать, если параметр макрокоманды отсутствует. Пусть, например, в этом случае во время компиляции выдаётся диагностика "No argument in macros `abs`!".[1] Приведём пример такого макроопределения, а потом обсудим его:

```
; EAX:=abs(x); x=r8, r16, r32, m8, m16, m32 и i32
abs macro x
    local L
ifb <x>
```

[1] К сожалению, выдать диагностику русскими буквами в этом случае будет затруднительно, так как программа на Ассемблере обычно набирается в каком-нибудь текстовом редакторе (например, NotePad) в кодировке Windows (CP1251), а консольное окно по умолчанию "считает", что текст диагностики Ассемблера выводится в кодировке DOS (CP866). Наши макрокоманды работы с консольным окном (из `io.inc`) сами перекодируют русский текст но только *на этапе счета* программы, однако директива **`echo`**, выводящая диагностику, выполняется *на этапе компиляции*. В этом случае единственный способ выдать диагностику русскими буквами, это набрать текст (только) этой диагностики в кодировке CP866, что возможно, но хлопотно. В принципе, возможно с помощью специальной консольной команды (временно) переключить кодировку консоли, но для программиста это малопригодно, так как он не знает, в какой момент работы программы это надо делать.

```
    echo No argument in macros abs!
    .err
    exitm
endif
if type x EQ 1 or type x EQ 2
    movsx eax,x; eax:=longint(x)
elseif type x EQ 4 or type x EQ 0
  ifdifi <eax>,<x>
    mov  eax,x
  endif
else
;; x не типов r8,r16,r32,m8,m16,m32,i32
;; т.е. type x <> {0,1,2,4}
    .err <Bad argument in macros abs!>
    exitm
endif
    cmp  eax,0
    jge  L
    neg  eax
L:
    endm
```

В этом макроопределении использовалось новое макросредство – директива

```
local L
```

Эта директива обязана стоять в *самом начале* тела макроопределения, она объявляет имя `L` *локальным* именем макроопределения. Как и локальные имена, например, в языке Паскаль, они не видны *извне* макроопределения, которое в этом смысле играет роль блока, следовательно, эти имена могут использоваться также и в других частях этого же программного модуля. Директива **`local`** для Макропроцессора имеет следующий смысл. При каждом входе в макроопределение локальные имена, перечисленные в этой директиве, получают новые уникальные значения. Обычно Макропроцессор выполняет это совсем просто: при первом входе в макроопределение заменяет локальное имя `L` на имя `??0001`, при втором входе – на имя `??0002` и т.д. Учтите, что в Ассемблере символ `?` относится к *буквам* и может входить в имена, хотя программисту и не рекомендуется использовать имена, начинающиеся с двух таких символов, чтобы не конфликтовать с именами, автоматически порождаемыми Макропроцессором.

Например, для макрокоманды `abs ax` (если она будет компилироваться в программе *первой*), будет построено макрорасширение

```
        movsx eax,ax
        cmp   eax,0
        jge   ??0001
        neg   eax
    ??0001:
```

Следующая (по тексту программы) макрокоманда `abs ebx` будет заменена на макрорасширение

```
        mov eax,ebx
        cmp eax,0
        jge ??0002
        neg eax
    ??0002:
```

Назначение директивы **`local`** становится понятным, если рассмотреть, что будет, если эту директиву убрать из макроопределения. В этом случае у двух макрорасширений макрокоманды **`abs`** будут внутри *одинаковые* метки `L`, что повлечёт за собой ошибку, которая будет зафиксирована на следующем этапе, когда компилятор Ассемблера станет переводить программу на объектный язык. Обязательно поймите, что при использовании директивы **`local`** внутри программного модуля на Ассемблере теперь не стало *одинаковых* имён.

Вслед за директивой **`local`** располагается условный макрооператор с именем **`ifb`** (**`if b`**lank), его условие считается истинным, если в угловых скобках ему задан *пустой* параметр (пустая строка символов). Это краткая (более удобная) форма условного макрооператора **`ifidn`** `<x>,<>`.

Директива Ассемблера **echo** предназначена для вывода во время компиляции диагностики об ошибке, текст диагностики программист располагает сразу вслед за первым пробелом после имени этой директивы. Таким образом, программист может задать *свою собственную* диагностику, которая будет выведена при обнаружении ошибки в макроопределении. В этом примере диагностика "No argument in macros `abs`!" выводится, если отсутствует аргумент у макрокоманды `abs`.[1]

После того, как макроопределение обнаружит ошибку в своих параметрах, у программиста есть две возможности дальнейших действий. Во-первых, можно считать выданную диагностику *предупредительной*, и продолжать компиляцию программы с последующим получением (или, как говорят, *генерацией*) объектного модуля. Во-вторых, можно считать обнаруженную ошибку *фатальной,* и запретить генерацию объектного модуля, в этом случае, естественно, будет невозможен и последующий выход программы на счет. Стоит, однако, заметить, что при возникновении фатальной ошибки компилятор Ассемблера, тем не менее, будет продолжать проверку остальной части входного модуля на наличие других синтаксических ошибок.

В этом макроопределении было принято второе решение и, чтобы зафиксировать фатальную ошибку, Макропроцессор выдаёт компилятору с Ассемблера директиву **.err**. Получив эту директиву, компилятор выдаст на экран, и вставит в протокол своей работы (листинг) диагностику о фатальной ошибке с номером `A2052`, обнаруженной в программе. Эта ошибка носит обобщенное название `forced error` (т.е. ошибка, "навязанная" компилятору Ассемблера Макропроцессором по указанию программиста). У директивы **.err** в угловых скобках можно задать вывод диагностики, например

.err <No argument in macros `abs`>

тогда предыдущая директива **echo** будет не нужна.

В отличие от диагностики, заданной директивой **echo**, диагностика из директивы **.err** выводится не только в поток `stderr` (по умолчанию на экран), но и в листинг программы, который выводится в поток `stdout` (если задана выдача листинга). Таким образом, использование этого способа выдачи диагностики предпочтительнее, если предполагается изучение листинга.

После возникновения фатальной ошибки и выполнения директивы **.err** дальнейшая обработка макроопределения не имеет уже никакого смысла, и эта обработка была прервана с помощью директивы **exitm**. Здесь эта директива прекращает процесс построения макрорасширения, и в нём остаются только те строки, которые попали туда до выполнения директивы **exitm**. В общем случае директива **exitm** производит выход вниз за ближайшую *незакрытую* директиву **endm**, которая, как вскоре будут показано, может задавать конец не только макроопределения, но и макроциклов. Таким образом, **exitm** прекращает обработку той части макроопределения, которая ограничена ближайшей при просмотре вниз директивой **endm**.

Вместо выдачи обобщённой диагностики об ошибке `forced error` можно попросить компилятор Ассемблера выдать конкретную диагностику об ошибку, например

.errb <x>, No argument in macros `abs!`

В этом случае генерируется диагностика с номером `A2057`. Так делать не рекомендуется, так как уже невозможно задать директиву **exitm**, так как **.errb** не стоит внутри условного оператора.

На этом примере показано, как программист может предусмотреть свою собственную реакцию и диагностику на ошибку в параметрах макрокоманды. Обязательно поймите, что реакция на ошибку в макрокоманде производится именно на этапе обработки самой макрокоманды, а не позже, когда компилятор Ассемблера будет анализировать полученное (неправильное) макрорасширение.

Далее изучим *логические* выражения Макропроцессора. Эти выражения весьма похожи на логические выражения Паскаля, только вместо логических констант **true** и **false** можно применять целые значения 1 и 0, а вместо знаков операций отношения используются мнемонические двухбуквенные имена, которые перечислены ниже:[2]

---

[1] Эта диагностика выводится в стандартный поток `stderr`, обычно он связан с экраном, в этот же поток компилятор выдаёт сообщения об ошибках.

[2] Похожие обозначения для операций отношения приняты и в некоторых других языках, например, в Фортране операции отношения обозначается как **.EQ.**, **.NE.**, **.GT.**, **.LT.** и т.д. В отличие от Паскаля, операции

EQ вместо **=** **NE** вместо **<>** **LT** вместо **<**
LE вместо **<=** **GT** вместо **>** **GE** вместо **>=**

Таким образом, ветвь нашего условного макрооператора

```
elseif type x EQ 4 or type x EQ 0
```

эквивалентна такой записи на языке Free Pascal

```
elseif (type X=4) or (type X=0)
```

Заметим, что в Паскале для этого примера нам необходимо использовать круглые скобки, так как операция отношения **=** имеет *меньший* приоритет, чем операция логического сложения **or**. В Макроассемблере же, наоборот, операции отношения (**EQ**, **GT** и т.д.) имеют более высокий приоритет, чем логические операции (**or**, **and**, **xor** и **not**), а так как оператор **type** имеет больший приоритет, чем оператор **EQ**, то круглые скобки вообще не нужны. К сожалению, в Ассемблере получается много уровней приоритета операторов, все они перечислены ниже в порядке убывания старшинства:

1. **()**, **[]**, **<>**, **length**, **size**, **width**, **mask**
2. **.** (точка)
3. **:** (переопределение сегмента по умолчанию)
4. **ptr**, **offset**, **seg**, **type**, **this**
5. **high**, **low**
6. Одноместные (унарные) **+** и **–**
7. *****, **/** (в смысле **div**), **mod**, **shl**, **shr**
8. Двухместные (бинарные) **+** и **–**
9. **EQ**, **NE**, **LT**, **LE**, **GT**, **GE**
10. **not**
11. **and**
12. **or**, **xor**
13. **short**, **.type**

Обратите также внимание на следующую типичную ошибку: выдачу диагностики при компиляции программы пытаются сделать с помощью макрокоманды **outstr**, например, так:

```
ifb <X>
  outstrln "Нет аргумента в abs!"
endif
```

При этом не понимают, что выдача такой диагностики будет производиться не на этапе *компиляции* программы, как необходимо, а только на этапе *счета* программы, до которого дело вообще не дойдёт, так как зафиксирована фатальная синтаксическая ошибка!

Как видно, в макроопределении программисту пришлось проверять отсутствие параметра. Можно, однако, потребовать, чтобы эту проверку выполнял сам Макропроцессор, для этого ему надо сказать, что этот параметр *нельзя* опускать. Это делается, описывая формальный параметр с ключевым словом **req**:

```
abs macro x:req
```

В этом случае Макропроцессор при отсутствии фактического параметра будет выдавать свою собственную (фатальную) диагностику "missing macro argument".

Как уже говорилось, для имен переменных оператор **type** возвращает длину соответствующей области памяти в байтах, это использовалось в последнем примере:

```
if type x EQ 1 or type x EQ 2
```

Здесь, однако, необходимо понять, что, когда Макропроцессор "спрашивает" у Ассемблера тип некоторого имени оператором **type** <имя>, то это имя должно быть описано в программе раньше (выше по тексту), чем вызов макрокоманды. Ясно, что в противном случае Ассемблер ещё ничего не знает об этом имени!

отношения Макропроцессора можно применять только к *целочисленным* операндам. Другими словами нельзя, например, написать сравнение на равенство текстовых строк

```
if <X> EQ <ax>; ОШИБКА ! Надо ifidn <X>,<ax>
```

Для сравнения строк, как рассказывалось ранее, используются специальные условные макрооператоры (**ifb**, **ifidn**, **ifdifi** и некоторые другие).

Заметим также, что в Ассемблере определены служебные имена констант **byte**=1 и **word**=2, поэтому можно было бы переписать последний условный макрооператор в виде

```
if type x EQ byte or type x EQ word
```

Здесь, однако, есть одна тонкость. Дело в том, что, например, переменную длиной в один байт можно описать в Ассемблере как в виде

```
X db ?; есть синоним X byte ?
```

так и в виде

```
Y sbyte ?
```

Так вот, несмотря на то, что служебное имя **sbyte** (но не **db** !) тоже является в Ассемблере константой, равной единице, для этих описаний в MASM 6.14 будет

```
type X = 1
type X = byte
type Y = 1
type Y = sbyte
type Y ≠ byte
```

Здесь надо ещё раз напомнить, что макросредства по существу являются алгоритмическим языком, поэтому полезно сравнить эти средства, например, с таким алгоритмическим языком, как Паскаль. В макроязыке сначала были изучены макроопределения и макрокоманды, которые являются аналогами соответственно описаний процедур и операторов процедур Паскаля. Затем изучены некоторые из условных макрооператоров, являющихся аналогами условных операторов Паскаля, а теперь пришла очередь заняться аналогами в макросредствах *операторов присваивания, переменных* и *циклов* языка Паскаль.[1]

Рассмотрим теперь следующую простую задачу. Известно, что для например, обнуления переменной X приходится использовать команду `mov X,0`, а для, например, обнуления регистра EBX предпочтительно использовать более короткую и быстро работающую команду `sub ebx,ebx`. Напишем макроопределение, которое само выбирает, как надо обнулять свой параметр:

```
Zero macro X:req; X:=0
;;  X форматов r8, r16, r32, m8, m16, m32, m64
    local K
    K=0
for reg,<al,ah,bl,bh,cl,ch,dl,dh          \
         ax,bx,cx,dx,si,di,bp,sp          \
         eax,ebx,ecx,edx,esi,edi,ebp,esp>
  ifidni <reg>,<X>
    K=1
    exitm
  endif
endm
if K EQ 1;  Параметр - регистр
    sub X,X
elseif type X EQ 1 or type X EQ 2 or type X EQ 4
    mov X,0
elseif type X EQ 8; X dq ? т.е. var X: int64;
    mov dword ptr X,0
    mov dword ptr X+4,0
else
    .err <Bad Argument in Macros Zero>
endif
    endm
```

Директива Макроассемблера из последнего примера

```
K=0
```

---

[1] В макроязыке есть и аналог оператора перехода **goto** <макрометка>, где макрометки задаются как "метки наоборот" в виде `:<метка>`, например, `goto M` . . . `:M`. Этот оператор может использоваться только внутри макроопределения. Однако, будучи ярыми сторонниками структурного программирования, макрооператоры **goto** мы использовать не будем.

является макрооператором *присваивания* и показывает использование нового важного понятия из макросредств Ассемблера – так называемых *переменных периода генерации*. Это достаточно сложное понятие, и сейчас будет разобрано, что это такое.

Название "переменные периода генерации" призвано подчеркнуть время *существования* этих переменных: они порождаются только на период обработки исходного программного модуля на Ассемблере и генерации объектного модуля. Следует понять, что когда компилятор Ассемблера завершает трансляцию программного модуля, числовые макропеременные уничтожаются (перестают существовать).

Числовые макропеременные могут принимать только *целочисленные* значения, в отличие от директивы **equ**, которая задаёт или числовые *макроконстанты* (их далее уже нельзя менять), или *текстовые макропеременные* (их далее можно менять), например:

```
V equ 100                 | W equ abc
V equ 200; ОШИБКА         | W equ def; НЕТ ОШИБКИ
V equ abc; ОШИБКА         | W equ 300; НЕТ ОШИБКИ
                          | W equ ijk; ОШИБКА
```

Здесь всё весьма запутано 😎.

Переменные периода генерации можно использовать в любом месте программы, например:

```
.data
  N = -1
  K db N and 111b; это K db 7
```

Как и переменные в Паскале, числовые макропеременные в Макроассемблере бывают *глобальные* и *локальные*. Глобальные переменные уничтожаются только после построения всего объектного модуля, а локальные – после выхода из того макросредства, в котором они порождены. В Макроассемблере различают локальные имена, заданные в директиве **local**, они уничтожаются после построения макрорасширения), и локальные переменные – параметры макроциклов **for** (они уничтожаются после выхода из этого цикла). В приведённом выше макроопределении локальной является числовая макропеременная с именем `K`, о чём объявлено в директиве **local**. Кроме того, локальным является и переменная периода генерации с именем `reg`, которая является параметром в макроцикле **for**, в отличие от числовых макропеременных, параметры цикла могут принимать только строковые значения, т.е. являются *строковыми* макропеременными.

В последнем макроопределении использована строка-комментарий, начинающаяся с двух символов (`;;`), это называется *макрокомментарием*. В отличие от обычных строк-комментариев, макрокомментарии не переносятся макропроцессором в макрорасширения, оставаясь, таким образом в листинге программы в *единственном* экземпляре (только в макроопределении).

К сожалению, в MASM 6.14 нет специальной директивы (аналога описания переменных **var** в Паскале), при выполнении которой *порождаются* макропеременные (т.е. им отводится место в памяти Макропроцессора). У нас макропеременные порождаются *автоматически*, при присваивании им первого значения (отметим, что также поступают и многие языки высокого уровня, например, модный Python). Так, в нашем макроопределении локальная числовая макропеременная с именем `K` порождается при выполнении макрооператора присваивания `K=0`, при этом ей, естественно, присваивается нулевое значение.

Следующая важная компонента макросредств Ассемблера – это макроциклы (которые, конечно, должны быть в макросредствах, как в любом "солидном" алгоритмическом языке высокого уровня).[1] В написанном макроопределении использовался один из видов макроциклов с именем **for**. Этот макроцикл называется циклом с параметром, он очень похож на цикл с параметром языка Паскаль и имеет такой синтаксис (параметр цикла назван, как обычно, именем `i`):

```
for i,<список макроцикла>
      тело макроцикла
endm
```

[1] Заметим, что в языках высокого уровня (Free Pascal, C и т.д.) есть далеко не все из рассматриваемых здесь макросредств.

Параметр цикла является локальной переменной, которая принимает *строковые* значения. Список цикла (он заключается в угловые скобки) является последовательностью (возможно пустой) текстовых строк, разделённых запятыми (напомним, что в Макропроцессоре строки могут и не заключаться в апострофы). В последнем макроопределении задан такой список макроцикла

```
<al,ah,bl,bh,cl,ch,dl,dh            \
 ax,bx,cx,dx,si,di,bp,sp            \
 eax,ebx,ecx,edx,esi,edi,ebp,esp>
```

Этот список содержит 24 текстовые строки длиной по 2 или 3 символа.

Выполнение макроцикла с именем **for** производится по следующему правилу. Сначала переменной цикла присваивается первое значение из списка цикла (первая строка текста, в нашем примере `i:='al'`), после чего выполняется тело цикла, при этом все вхождения в это тело имени параметра цикла заменяются на его текущее значение. После этого параметру цикла присваивается следующее значение из списка цикла и т.д. Для пустого списка тело цикла, естественно, не будет выполняться ни одного раза. В нашем примере тело цикла будет выполняться не более 24-х раз, при этом переменная будет последовательно принимать значения строк `al`, `ah`, `bl` и т.д.

+ Макроцикл считается *стандартным макроопределением*, поэтому в нём можно использовать многие средства из обычных макроопределений. Например, у параметра цикла, как и у параметров макрокоманды, можно задать ключ **`i:req`**, тогда будет фиксироваться ошибка при попытке присвоить параметру цикла *пустой* строки из списка параметров. Также можно указать параметр цикла в виде `i:=`<значение по умолчанию>, это значение будет присвоено параметру цикла, если очередной элемент из списка параметров окажется пустым. Сразу после заголовка макроцикла можно задать список локальных имён, например:

```
for i,<список макроцикла>
    local L
. . .
endm
```

При этом имя `L`, как и в обычном макроопределении, будет уникальным для *каждого* нового значения параметра цикла `i`.

Как можно заметить, целью выполнения макроцикла в нашем примере является присваивание числовой макропеременной `K` значения единицы, если параметр макрокоманды совпадает по написанию с именем одного из регистров, причём это имя может задаваться как большими, так и малыми буквами в любой комбинации. Это позволяет распознать имя регистра, как бы его ни записал программист, и присвоить переменной `K` значение единица, в противном случае переменная `K` сохраняет нулевое значение. После присваивания значения `K=1` дальнейшее выполнение цикла бессмысленно, и происходит выход из него по директиве **`exitm`** за ближайшую вниз директиву **`endm`**.

Следует обратить внимание, что макроцикл завершается такой же директивой **`endm`**, как и всё макроопределение. Как уже говорилось, это позволяет рассматривать макроцикл как некоторое *стандартное* макроопределение, только без заголовка (т.е. без имени и без параметров).

Далее в макроопределении расположен условный макрооператор нового вида, который, однако, очень похож на условный оператор языка Паскаль:

```
if <логическое выражение>
        ветвь then
elseif <логическое выражение>
        ветвь elseif
    . . .
else
        ветвь else
endif
```

При вычислении логических выражений не должно быть ссылок вперед, т.е. все имена должны быть определены выше по тексту программы (почему это так будет ясно из дальнейшего описания схемы работы Макроассемблера).

Таким образом, если в условном макрооператоре K=1, тогда параметр макрокоманды – это регистр, а когда **type** X равен 1, 2, 4, 8 или 0, то параметр имеет формат m8, m16, m32, m64 или i32. В остальных случаях мы попадаем на ветвь **else**, где выдаётся аварийная диагностика.[1]

Необходимо также заметить, что операции отношения **LT,GT,LE** и **GE**, как правило, рассматривают свои операнды как беззнаковые значения. Исключением является случай, когда Макропроцессор "видит", что некоторая числовая константа *явно* отрицательная (т.е. в константе есть явный знак минус). Например, рассмотрим следующий фрагмент программы:

```
        k = 0FFFFFFFFh
; Макропроцессор "видит" беззнаковое k=0FFFFFFFh
if k LT 0; Берётся k=0FFFFFFFFh > 0 ==> false !
        ...
        k = -1
; Макропроцессор "видит" знаковую макроконстанту k=-1
if k LT 0; Берётся K=-1 < 0          ==> true !
```

Отметим также, что при выполнении арифметических операций все выходы значения за диапазон **dword** игнорируются, например:

```
k = -1
k = k+1;; k = 0
k = 0FFFFFFFFh
k = k+1;; k = 0 !
```

Далее, Вам важно понять принципиальное отличие переменных языка Ассемблера и переменных периода генерации. Так, переменная Ассемблера с именем X может, например, определяться предложением резервирования памяти

```
X dw 13
```

В то время как числовая макропеременная с именем Y может порождаться макрооператором присваивания

```
Y = 13
```

Главное – это уяснить, что эти переменные имеют разные и непересекающиеся времена́ существования. Числовые макропеременные существуют только во время компиляции исходного модуля с языка Ассемблер на объектный язык, и заведомо уничтожаются до начала счета. В то же время описанные в секции данных переменные Ассемблера, наоборот, существуют только во время счета программы (от начала счета до выполнения макрокоманды **exit**), и на этапе компиляции *не существуют*. Некоторые учащиеся не понимают этого и пытаются использовать значение переменной Ассемблера на этапе компиляции, например, пишут такой неправильный условный макрооператор:

```
if X EQ 13
```

Это сразу показывает, что они не понимают суть дела, так как на этапе *компиляции* хотят анализировать *значение* переменной X, которая будет существовать только во время *счета* программы. Необходимо понять, что на этапе компиляции значение есть не у *переменной* с именем X, а только у *имени* переменной X (значение имени X равно адресу, т.е. **offset** X этой переменной в памяти). Кроме того на этапе компиляции почти все имена имеют *тип*, например, для нашего имени **type** X=2.

Как известно, в Паскале подзадачи можно реализовывать в виде процедур и функций. Применение функций оправдано, если подзадача вырабатывает один и небольшой по размеру результат, который можно непосредственно использовать в выражении или команде. Например, пусть необходимо реализовать оператор присваивания EBX:=abs(cl), с использованием макрокоманды **abs** это можно сделать *двумя* предложениями Ассемблера:

```
abs cl; eax:=Longword(abs(cl))
mov ebx,eax
```

---

[1] К сожалению, тип упакованного битового поля **record** и некоторых структур **struc** тоже может быть равен 1, 2 или 4 (это сумма длин их полей), в этом случае, вероятно, будет семантическая ошибка, так как компилятор Ассемблера её не обнаружит. С другой стороны, таким переменным всё равно присвоится ноль!

Описанное ранее макроопределение **abs** вообще говоря называется *макропроцедурой* (Macro Procedures). А вот если бы макроопределение было *макрофункцией* (Macro Functions), то можно было бы обойтись одним предложением

```
mov ebx,abs(cl)
```

Макроассемблер позволяет описывать макроопределения в виде *макрофункций*, для этого нужно в теле макроопределения использовать хотя бы одну директиву **exitm** и все они должны иметь вид

exitm <строка-возвращаемое значение>

В описании макроопределения **abs** для этого достаточно заменить все **exitm** на **exitm** <> и изменить самый конец описания:

```
        . . .
L:
        exitm <eax>
        endm
```

Обратите внимание, что наша макрофункция, в отличие от обычных функций, не возвращает результат на регистре EAX, она возвращает *строку символов* 'eax'. Как и для обычных функций строка-результат макрофункции заменяет вызов этой макрофункции. Эта строка не обязательно является записью имени регистра EAX, она может быть произвольной строкой, например

exitm <-3 **shr** 1>

Учтите, что ставить метку на *директиву*, например, L:**exitm** <eax> нельзя, т.к. это не команда. Параметры макрофункции заключаются в круглые скобки, если параметров нет, то необходимо указывать пустые скобки **()**. Как видно, макрофункция может возвращать и пустую строку символов **exitm** <>. Смешивание в одном макроопределение директив **exitm** <возвращаемое значение> и просто **exitm** приведёт к трудно переводимой диагностике Ассемблера :error A2126: EXITM used inconsistently.

Макрофункции Ассемблера, как и функции в языках высокого уровня, являются удобным и наглядным способом записи алгоритма. В качестве примера напишем макрофункцию min(x,y), которая возвращает минимальное из *знаковых* значений x и y формата двойного слова (**dd**), для простоты не будем делать анализ на наличие параметров и их правильность (например, на то, что y=eax):

```
min macro x,y
    mov   eax,x
    cmp   eax,y
    cmovg eax,y
    exitm <eax>
endm
```

Примеры использования этой макрофункции:

```
outintln min(z,ebx)
mov edx,min(ecx,min(z,-13))
```

Теперь рассмотрим пример, как макроопределение может обрабатывать макрокоманды с *переменным* числом фактических параметров. Задачи такого рода часто встают перед программистом. Пусть, например, в программе надо часто вычислять максимальное значение от нескольких *знаковых* целых величин в формате двойного слова (**dd**). Для решения этой задачи в рассматриваемом Макроассемблере можно написать макроопределение, у которого указан только *один* формальный параметр, на место которого будет, однако, передаваться *список* (возможно пустой) фактических параметров.[1] Такой список в Макроассемблере заключается в угловые скобки. Пусть, например, макроопределение должно вычислить и поместить на регистр EAX максимальное значение из величин ebx,X,-13,ecx, тогда нужно вызвать это макроопределение с помощью такой макрокоманды (дадим ей, например, имя maxn):

```
maxn <ebx,X,-13,ecx>
```

[1] Такая возможность есть и во многих современных языках высокого уровня, например, в языке Python.

Здесь у макрокоманды задан один фактический параметр (одна строка из 15 символов, включая и угловые скобки), который, однако, наше макроопределение будет трактовать как список, содержащий в угловых скобках четыре "внутренних" параметра.

Сделаем спецификацию этой макрокоманды. Пусть будет допустимо, чтобы некоторые параметры из списка опускались (т.е. задавались пустыми строками). При поиске максимума такие пустые параметры будут просто игнорироваться, и при этом не будет выдаваться никакой диагностики. Далее необходимо договориться, что будет делать макрокоманда, если список параметров вообще пуст. В этом случае можно, конечно, выдавать диагностику о фатальной ошибке и запрещать генерацию объектного модуля, но можно поступить и более "гуманно": в качестве результата будет выдаваться самое маленькое знаковое число `MinLongint` (`80000000h`). Макроопределение будет реализовано в виде макропроцедуры.

```
maxn macro X
;; переменное число аргументов
      mov   eax,80000000h; MinLongint
  for i,<X>
    ifnb <i>
      cmp   eax,i
      cmovl eax,i; if eax<i then eax:=i
    endif
  endm
     endm
```

Перед объяснением работы макроопределения `maxn`, как было обещано ранее, надо существенно уточнить правила передачи фактического параметра (строки символов) на место имени формального параметра. Дело в том, что некоторые символы, входящие в строку-фактический параметр, являются для Макропроцессора *служебными* и обрабатываются по-особому, такие символы называются *макрооператорами*.[1] Ниже приведено описание наиболее интересных макрооператоров, полностью их можно изучить по учебникам [5-7].

- Если фактический параметр заключён в угловые скобки, то эти скобки считаются макрооператорами *выделения*, они заставляют рассматривать заключённый в них текст как единое целое, даже если в нём встречаются служебные символы. Обработка макровыделения заключается в том, что парные угловые скобки *отбрасываются* при передаче фактического параметра на место формального. Обратите внимание, что отбрасывается только одна (внешняя) пара угловых скобок, если внутри фактического параметра есть ещё угловые скобки, то они сохраняются.
- Символ восклицательного знака (**!**) является макрооператором, он *удаляется* из фактического параметра, но при этом блокирует (иногда говорят, – *экранирует*) анализ следующего за ним символа на принадлежность к служебным символам (т.е. макрооператорам). Например, фактический параметр `<ab!!+!>>` преобразуется в строку `ab!+>`, именно эта строка и передаётся на место формального параметра. Так можно передать в фактическом параметре сами служебные символы.
- В том случае, если комментарий начинается с двух символов **;;**, то, как уже говорилось, это *макрокомментарий,* в отличие от обычного комментария (начинающегося одним символом **;**) он *не переносится* в макрорасширение.
- Символ **&** является макрооператором, он удаляется Макропроцессором из обрабатываемого предложения (заметим, что из двух и более следующих подряд символов **&** удаляется только один). Данный символ играет роль лексемы-разделителя, он позволяет выделять в тексте имена формальных параметров макроопределения и числовых макропеременных. Например, пусть в программе есть такой макроцикл

```
for i,<L,H>
   mov A&i,X&i
endm
```

---

[1] Здесь полезно вспомнить, что стандартная процедура `Read` языка Free Pascal при вводе из потока `input` тоже рассматривала некоторые символы (`Esc`, `Tab`, `CR` и др.) как служебные и обрабатывала по-особому.

После обработки этого макроцикла Макропроцессор подставит на его место в программу следующие строки:

```
mov AL,XL
mov AH,XH
```

Пример использования двух следующих подряд макрооператоров **&**:

```
M1 macro x
for i,<1,2,3>
  x&&i db i
endm
endm
```

Для макрокоманды `M1 var` после последовательной замены вхождений `x&` (на `var`) и `&i` (на 1, 2 и 3) будет получено макрорасширение

```
var1  db 1
var2  db 2
var3  db 3
```

- Символ **%** является макрооператором, он предписывает Макропроцессору вычислить следующее за ним *арифметическое макровыражение* и подставить строку-значение этого выражения вместо знака **%**. Например, после обработки предложений

```
P macro x
  echo x
endm
M equ (3*5+7)
P M
   OUT: M
P %M
   OUT: 22
```

Разберём теперь выполнение нашего макроопределения `maxn` на примере макрокоманды

```
maxn <-13,,ebx,Z>
```

При передаче фактического параметра-строки `<-13,,ebx,Z>` крайние угловые скобки будут отброшены, поэтому в макроцикле **for** на место формального параметра `X` (внутрь угловых скобок) будет подставлена строка символов `-13,,ebx,Z`. Таким образом, макроцикл принимает следующий вид:

```
for i,<-13,,ebx,Z>
  ifnb <i>
    cmp   eax,i
    cmovl eax,i
  endif
endm
```

В теле этого макроцикла располагается условный макрооператор с именем **ifnb** (**if n**ot **b**lank), он проверяет свой параметр и, если этот параметр *не является* пустой строкой символов, то он обрабатывает своё тело (все строки до ближайшего вниз незакрытого **endif**). Таким образом, получается, что в теле макроцикла выполняется условный макрооператор, который только для *непустых* элементов из списка цикла осуществляет поиск максимума.

Обратите внимание, что наше макроопределение будет *неправильно* работать для макрокоманд, содержащих в списке параметров регистр `EAX`, например

```
maxn <-13,eax,ebx,Z>
```

Однако, Вы уже должны представлять, как это можно исправить, усложнив макроопределение `maxn`. При вызове макропроцедуры `maxn` можно опустить угловые скобки, например

```
maxn -13,,ebx,Z
```

В этом случае, однако, надо изменить и заголовок макроопределения

```
maxn macro X:VarArg
```

Здесь ключевое слово **VarArg** (**Var**iable **Arg**ument Length) предписывает взять в качестве значения фактического параметра весь текст до конца строки или до начала комментария. Ясно, что параметр с таким ключевым словом может быть только один и *последним* в списке формальных параметров. Например, для макрокоманды

```
      maxn -13,,ebx,Z
```
будет получено следующее макрорасширение
```
      mov   eax,80000000h; MinLongint
      cmp   eax,-13
      cmovl eax,-13
      cmp   eax,ebx
      cmovl eax,ebx
      cmp   eax,Z
      cmovl eax,Z
```

Макроопределения с переменным числом параметров являются удобным средством при программировании многих задач. Аналогичный механизм (но с совершенно другой реализацией) есть, например, в языке высокого уровня C, который допускает написание функций пользователя с переменным числом фактических параметров. Правда, для функций языка C, как правило, фактических параметров должно быть не менее одного, и чаще всего требуется, чтобы значение этого *первого* параметра каким-то образом задавало *общее* число фактических параметров для этой функции.

Известно, что в языке Паскаль программист не может писать свои собственные процедуры и функции с переменным числом параметров, так как, по мнению автора этого языка Н. Вирта, это снижает надёжность программирования. В то же время совсем обойтись без такого очень удобного средства Паскаль не мог: в нём есть *стандартные* процедуры с переменным числом параметров, например, это процедуры ввода/вывода `Read` и `Write`.

В качестве ещё одного примера рассмотрим следующую задачу. Предположим, что в программе необходимо часто суммировать массивы коротких (**db**), длинных (**dw**) и сверхдлинных (**dd**) целых чисел. Известно, что для стандарта языка Паскаль у процедур и функций должно соблюдаться строгое соответствие между типами формальных и фактических параметров. На языках высокого уровня, которые не допускают написание полиморфных процедур и функций, хорошо решить эту задачу практически невозможно.[ii] [см. сноску в конце главы]

На Ассемблере для реализации такого суммирования можно написать макроопределение, например, с таким заголовком:
```
      SumMas macro X:req,N:=<100>
```
В качестве первого параметра `X` этого макроопределения можно задавать массивы коротких, длинных и сверхдлинных *знаковых* целых чисел, количество элементов в этом массиве указывается во втором параметре `N`. Другими словами, первый операнд макрокоманды `SumMas` может быть формата `m8`, `m16` или `m32`, а второй – формата `m32`, `r32` или `i32`. Сумма должна возвращаться на регистре `EAX`, который подставляется на место макрокоманды (т.е. будем писать макрофункцию).

Для второго параметра `N:=<100>` задано *значение по умолчанию*, это значение параметр принимает, когда он опущен (не задан) в макрокоманде, в этом случае Макропроцессор, считает его равным строке из трёх символов 100 (т.е. предполагается, что массивы такой длины наиболее часто встречаются в программе пользователя, и ему просто лень каждый раз задавать эту длину). Для простоты изложения макроопределение не будет сохранять, а затем восстанавливать используемые регистры, а переполнение при сложении будет игнорироваться без выдачи диагностики об ошибке. Кроме того, не будет проверяться допустимость типа второго параметра, например, передачу в качестве второго параметра короткого регистра `r8` или какой-нибудь метки программы (Вы должны уже представлять, как можно сделать такую проверку на допустимость параметра). Ниже показан возможный вариант такой макрофункции.
```
SumMas macro X:req,N:=<100>
      Local K,L
      K=type (X)
if K NE 1 and K NE 2 and K NE 4
      .err <***SumMas*** Bad Array Type>
      exitm <>; нет значения, т.к. счета не будет
endif
      mov   ecx,N;  длина массива
      lea   ebx,X;  адрес X[1]
      xor   eax,eax; сумма:=0
if K EQ 1
```

```
L:    movsx edx,byte ptr [ebx]; edx:=Longint(X[i])
      add   eax,edx
elseif K EQ 2
L:    movsx edx,word ptr [ebx]; edx:=Longint(X[i])
      add   eax,edx
else
L:    add   eax,[ebx]
endif
      add   ebx,K; i:=i+1
      loop  L
      exitm <eax>
      endm
```

Как видно, это макроопределение настраивается на тип переданного массива и оставляет в макрорасширении только команды, предназначенные для работы с элементами массива именно этого требуемого типа. Заметьте также, что вся эта работа по настройке на нужный тип параметров производится до начала счета (на этапе компиляции), то есть на машинном языке получается эффективная программа, настроенная на нужный тип данных и не содержащая никаких лишних команд.[1]

В качестве ещё одного примера рассмотрим написание макрофункции вычисления факториала от неотрицательного числа. Единственный параметр формата `i32` задаёт число `N` для вычисления `N!`

```
Fact  macro N:req
      local K,F
if type N NE 0
     .err <***Factorial*** Bad Parameter Type>
      exitm <>
endif
      K=1
      F=1
repeat N
      F=F*K
      K=K+1
endm
      exitm <F>
      endm
```

Параметр должен быть константой (**type** `N=0`). Для вычисления факториала используется макроцикл **repeat** (есть короткий синоним **rept**), который называется в Макроассемблере *блоком повторения*. Тело этого цикла выполняется `N` раз, а если `N<1`, то ни одного раза. Кроме того, в Ассемблере, конечно, есть и макроцикл **while**:[2]

```
      while <выражение>
            <тело цикла>
      endm
```

На Паскале выполнение этого макроцикла можно записать как

```
while <выражение> <> 0 do <тело цикла>
```

Директива `F=F*K` выполняет *беззнаковое* умножение, так как Макропроцессор не видит явного знака минус, а например, `F=F*(-K)` будет выполнять уже знаковое умножение. Перепишем теперь макрофункцию факториала в рекурсивной форме (но без проверки на плохой тип параметра):

```
Fact  macro N:req
      local F
      F=1
```

---

[1] Такая настройка оказалась возможна только потому, что компилятор Ассемблера и Макропроцессор работают "рука об руку", поочерёдно обрабатывая в программе предложение за предложением. Именно поэтому при выполнении оператора **type** `X` внутри макроопределения компилятору, а, следовательно, и макропроцессору, уже известен тип имени фактического параметра, описанного где-то выше в тексте программы. Более подробно как это делается, будет рассказано в главе, посвящённой схеме работы компилятора с Ассемблера.

[2] Макроциклы **repeat** и **while**, как и рассмотренный ранее макроцикл **for**, являются *стандартными макроопределениями*, поэтому в них, например, разрешена директива **local** для задания уникальных имён.

```
  if N GT 1
      F=(N)*Fact(N-1)
  endif
      exitm <F>
endm
```

Обратите внимание на сомножитель (N), он задан в скобках, так как на его место при рекурсивном вызове будут подставляться строки N-1, N-1-1, N-1-1-1 и т.д., что без скобок вызовет ошибку старшинства операций, например N-1*Fact(N-1). Скобки можно не ставить, если задать присваивание с макрооператором % в виде F=N*Fact(%N-1), что заставит Макропроцессор каждый раз *вычислять* фактический параметр макрофункции перед её вызовом.

Посмотрим теперь, как в макроопределении можно проконтролировать, что в качестве параметра передано *непустое имя* (переменной, процедуры и т.д.), например,

```
M macro Name
```

Проще всего проверить, что переданный параметр не пустой, как уже говорилось, это можно сделать, используя условный макрооператор

```
      ifb <Name>
        .err <Name is Blanc>
      endif
```

Кроме того, эту проверку (с выдачей фатальной ошибки) можно задать и короче:

```
      .errb <Name>,<Name is Blanc>
```

Несколько сложнее обстоит дело, если программист захочет проверить, что в качестве первого параметра передано именно *имя*, а не какая-нибудь другая строка символов, скажем X[EBP+6]. Поставленную задачу можно решить, например, с помощью такого фрагмента на Макроассемблере:

```
CheckName macro Name
      local Nom,Err
        Nom=1; Номер символа в имени
forc i,<Name>
        Err=1; Признак ошибки в символе имени
  if Nom EQ 1
    forc j,<abcdefghijklmnopqrstuvwxyz_?@$.>
      ifidni <i>,<j>
        Err=0
        exitm
      endif
    endm
  else
    forc j,<abcdefghijklmnopqrstuvwxyz_?@$0123456789>
      ifidni <i>,<j>
        Err=0
        exitm
      endif
    endm
  endif
  if Err EQ 1
       exitm; Выход из цикла forc
  endif
       Nom=Nom+1; Следующий номер символа в имени
endm; {forc}
if Err EQ 1
        .err <Bad character in Name>
        exitm; Выход из макроопределения
endif
   ...
```

Для анализа символов фактического параметра на принадлежность заданному множеству символов использовался макроцикл **forc**. Выполнение этого макроцикла очень похоже на выполнение макроцикла **for**, за исключением того, что параметру цикла каждый раз присваивается очередной один символ из строки-параметра, символы которой и являются списком цикла. В рассмотренном примере в списке цикла в первом операторе **forc** указаны все символы, с которых может *начинать-*

ся имя в Ассемблере (как и в Паскале, это латинские буквы и некоторые символы, приравнивающиеся к буквам). Во втором операторе **forc** к этим символам просто добавлено ещё и 10 цифр (но убрана точка).

Итак, пусть теперь в качестве первого параметра наше макроопределение получило *некоторое имя*. Но как проверить, что полученное имя является, например, *именем переменной*, а не процедуры, метки или директивы **equ** ? Чтобы выяснить это, в написанное макроопределение можно вставить проверку *типа* полученного имени, например, так:

```
if type Name LT 1
   .err < Not Name of a Variable! >
exitm
endif
```

Здесь предполагается, что тип переменной может быть любым положительным числом, однако, например, тип "имени типа" структуры, упакованного битового поля или объединения, тоже, к сожалению, будет иметь положительное значение. Например, раньше встречался тип структуры **type** Stud=25.

Некоторые дополнительные характеристики имени можно получить, применив к этому имени одноместный оператор Макроассемблера **.type** <операнд>. Результатом работы этого оператора является целое значение в формате байта (i8), при этом каждый бит в этом байте, если он установлен в "1", указывает на наличие некоторой *характеристики* операнда. Ниже приведены *номера* битов в байте, который этот оператор вырабатывает, будучи применённым к своему операнду (напомним, что биты в байте нумеруются справа налево, начиная с нуля):

№ бита	Характеристика операнда
0	операнд определяет команду (процедура или метка)
1	операнд определяет переменную (область данных)
2	операнд имеет абсолютное (неперемещаемое) значение
3	операнд имеет перемещаемое значение
4	регистровое выражение
5	операнд как-то определён
6	операнд определён в стеке
7	имя объявлено в **extrn**

Так, например, для имени, описанного в Ассемблере как `X dw ?` значение оператора **.type** X=00101010b, для метки `L:cwd` будет значение **.type** L=00100101b, а для имени типа структуры Stud будет значение **.type** Stud=00100100b (т.е. имя Stud как-то определено, но не является ни меткой, ни переменной).[1]

Теперь будет разобран пример, когда макрокоманда используется не в секции команд, а, например, в секции данных. Предположим, что программисту приходится много работать с целочисленными скалярными матрицами различной размерности, с разным типом и значением элемента на главной диагонали.[2] Каждая скалярная (а, следовательно, квадратная) матрица в Ассемблере полностью определяется следующими атрибутами:

- размерностью матрицы N>=1, этот параметр будет считаться *обязательным*;
- значением элемента на главной диагонали P (все они одинаковые), если он не задан, будет предполагаться единичная матрица;
- типом элемента матрицы Tip, в Ассемблере это **db**, **dw**, **dd** или **dq**, если параметр опущен, будет предполагаться матрица слов (**dw**);
- именем матрицы Name (точнее, это имя первого элемента матрицы), если имя не задано, то под матрицу будет зарезервирована *безымянная* область памяти. Поймите, почему имя Name нельзя сделать меткой, поставленной перед макрокомандой.

---

[1] Более общий оператор **OpAttr** <операнд> возвращает 16-битное значение, младший байт которого совпадает со значением оператора **.type**, а старший байт описывает язык, соглашения о связях которого используются в программе.

[2] Напомним, что квадратная матрица является скалярной, если все её элементы на главной диагонали совпадают, а вне главной диагонали равны нулю. Простейшим примером скалярной матрицы является единичная матрица.

Следовательно, для порождения, например, *единичной* матрицы размером 5x5 с именем My_Mat из двойных слов можно использовать макрокоманду

```
ScalMat 5,,dd,My_Mat
```

Соответствующее макроопределение может выглядеть таким образом:

```
ScalMat macro N:req,P:=<1>,Tip:=<dw>,Name:=<>
      local R
if type N NE 0 or type P NE 0
      .err <*** Bad Dimension or Matrix Element ***>
      exitm
endif
Name  Tip P,(N-1) dup (N dup (0),P)
      endm
```

Вот теперь в секции данных (как, впрочем, и в любой другой секции) можно резервировать область памяти под скалярную матрицу макрокомандой **ScalMat**, например:

```
.data
   X dw ?
     ScalMat 10,-1,db,Neg_10x10_Matrix
```

Заметим, что при N=0 Ассемблер MASM, встретив в строке (-1) **dup**, не считает это ошибкой и просто вообще не будет отводить память под матрицу, поэтому проверку на N>0 можно не проводить.

И, в заключение, рассмотрим стандартные *строковые* макрокоманды, встроенные в Ассемблер MASM. Важно понять, что эти макрокоманды работают только со строками символов, определёнными во время компиляции, следовательно, они не могут обрабатывать строки вида

```
T db "ABCDEF",0
```

которые *не существуют* во время компиляции, им будет отведена память и они будут существовать только во время счета программы. Разумеется, можно встроить *результаты* работы стандартных макрокоманд в конструкции (команды и предложения резервирования памяти) Ассемблера, чтобы позже работать с этими результатами на этапе счета. На этапе компиляции результаты работы текстовых макрокоманд можно вывести в поток stdout уже упоминавшейся ранее директивой **echo**. Каждая из строковых макрокоманд реализована как в виде макропроцедуры, так и в виде макрофункции. Для макропроцедуры каждый параметр заключается в угловые скобки, для макрофункции можно этого не делать, если внутрь строки-параметра не входят служебные символы. Имя строковых макрофункций начинается с символа @, а параметры, как всегда для функций, заключаются в круглые скобки.

Стандартная макрокоманда **CatStr** (con**Cat**ination **Str**ings) производит сцепление (склеивание) своих параметров-строк, например:

```
echo CatStr <A>,<+>,<B>
   OUT: CatStr <A>,<+>,<B>
```

echo видит просто текст CatStr <A>,<+>,<B> и выводит его.

```
T CatStr <A>,<+>,<B>
```

строковой макропеременной с именем T присваивается результат макропроцедуры CatStr т.е. строка A+B.

```
echo T
   OUT: T
```

echo видит просто имя T и выводит его.

```
%echo T
   OUT: A+B
```

%**echo** *вычисляет* свой параметр, т.е. значение переменной T, и выводит его.

```
%echo @CatStr(A,+,B)
   OUT: A+B
```

%**echo** вычисляет свой параметр, т.е. вызывает макрофункцию @CatStr, которая подставляет вместо себя результат своей работы, т.е. строку A+B, а %**echo** выводит эту строку.

```
.data
```

```
Txt db "Вот строка ",@CatStr(!",A,+,B,!"),0
.code
    outstr offset Txt
```

макрокоманда **outstr** выведет строку `Вот строка A+B` уже во время счета программы.

Стандартная макрокоманда

SubStr <строка>,<начальная позиция>[,<длина>]

выдаёт в качестве своего результата подстроку заданной длины, начиная с указанной позиции, а если длина не задана, то берётся текст до конца строки, например:

```
T SubStr <ABCDEF>,2,3
%echo T
   OUT: BCD
%echo @SubStr(ABCDEF,2,4)
   OUT: BCDE
%echo @SubStr(<ABC,DEF;>,3)
   OUT: C,DEF;
```

Стандартная макрокоманда

InStr <начальная позиция>,<строка>,<подстрока>

выдаёт в качестве своего результата номер первой позиции подстроки в строке, поиск подстроки начинается с указанной начальной позиции, например:

```
T InStr 2,<ADBCADEF>,<AD>
```

макрокоманда InStr выдаёт результат-строку 05, результат опознаётся как *целое число*, поэтому присваивается не строковой, а *числовой* макропеременной T, которая получает значение 5.

```
T=T+1
```

числовая макропеременная T получает значение 6.

```
%echo @CatStr(%T)
   OUT: 6
```

%T вычисляет (извлекает) целое значение из макропеременной T, а макрофункция @CatStr преобразует это число в строку символов для вывода, поэтому выводится 6, а не 06.

```
%echo @InStr(ADBCADEF,2,AD)
   OUT: 05
```

макрофункция InStr подставляет вместо себя результат-строку 05, который, так как он не присваивается никакой переменной, то остаётся просто строкой 05, которая и выводится директивой **%echo** (ну, как всё запутано ☹).

Макрофункции @Date и @Time возвращают текущие дату и время в виде строк символов, например:

```
T CatStr <Date=>,<@Date>,<, Time=>,<@Time>
```

строковая макропеременная T *вычисляется* (так как в правой части строка содержит макрофункции) и получает значение строки с датой и временем (в примере дата и время выбраны произвольно):

```
; Пусть Date=24/04/20, Time=10.41.52
%echo T
   OUT: Date=24/04/20, Time=10.41.52
```

строка T уже вычислена, поэтому надо ставить не %T, а просто T (поставьте %T и увидите, что будет ☺).

```
%echo @CatStr(Data=,@Date,<, Time=>,@Time)
   OUT: Date=24/04/20, Time=10.41.52
```

обратите внимание на параметр <, Time=>, в нём нужны угловые скобки, так как внутри запятая.

Макрофункция @Version возвращают числовое значение версии компилятора Ассемблера, в нашем случае 614, например:

```
%echo @CatStr(<Version MASM = >,          \
      %@Version/100,.,%@Version mod 100)
   OUT: Version MASM = 6.14
```

при вызове макрофункции @CatStr её числовые параметры вычисляются, так как используется макрооперация %, при этом будет %@Version/100=6 (здесь / ≡ **div**), %@Version **mod** 100=14, но, так как @CatStr рассматривает числовые значения как строки, то это и обеспечивает нужный

вывод. Угловые скобки в первом параметре нужны, так как внутри значащий конечный пробел. Можно сохранить числовое значение версии в *описании переменной* и использовать уже во время счета, например:

```
.data
   V   dw @CatStr(@Version)
. code
   mov ax,V
   mov bl,100
   div bl
   outword al,,"Наша версия MASM="
   outwordln ah,,"."
      OUT: Наша версия MASM=6.14
```

Макрофункция `@Line` возвращают номер строки программы на Ассемблере, в которой находится эта макрофункция, например:

```
%echo Current Program Line = @CatStr(%@Line)
   OUT: Current Program Line = 115
```

Макрофункция `@CurSeg` возвращают имя текущей секции программы на Ассемблере, например:

```
%echo Name of Current Section = @CurSeg
   OUT: Name of Current Section = _DATA [1]
```

Макрофункция `@SizeStr` возвращают числовую длину своего параметра-строки, например:

```
%echo @SizeStr(ABCDE)
   OUT: 05
```

Макрофункция `@FileCur` возвращают строку-имя текущего файла, например:

```
%echo @CatStr(<File Name = >,!",%@FileCur,!")
   OUT: "MyProg.asm"
```

Обратите внимание на использование макрооператора **!** для экранирования следующего за ним служебного символа – двойной кавычки. Попробуйте убрать макрооператоры **!** и посмотрите, что получится.

В Ассемблере MASM 6.14 есть и другие стандартные макрокоманды и макрофункции, которые в этой книге рассматриваться не будут. В качестве примера использования строковых макрокоманд рассмотрим задачу инвертирования списка строк, разделённых запятыми. Например, вместо строки

```
a,b,c,1,2,3
```

необходимо получить строку

```
3,2,1,c,b,a
```

Напишем макрофункцию, решающую эту задачу:

```
Revers macro x:VarArg
      Local T
echo Input List=x; echo параметра-КОНСТАНТЫ x
T     textequ <>;  T:='' - СТРОКОВАЯ МАКРОПЕРЕМЕННАЯ
   for i,<x>
T     CatStr  <i>,<,>,T; Это T:=i+","+T
   endm
T     SubStr  T,1,@SizeStr(%T)-1; убрать послед. запятую
%echo Output  List=T;  echo СТРОКОВОЙ МАКРОПЕРЕМЕННОЙ T
      exitm   <T>
endm
```

В этом примере использована новая директива с именем **textequ** для присваивания значения строковой макропеременной. В отличие от уже известной Вам директивы **equ**, новая директива обладает бóльшими возможностями, так как всегда *вычисляет* значение своего параметра в правой части непосредственно *перед* присваиванием этого значения строковой макропеременной в левой части. Фактически это (безымянная) однострочная макрофункция.

---

[1] Если использовать `@CurSeg` до директивы **include** `console.inc`, то выдаётся пустая строка (никакая секция пока не задана), а если после этой директивы, то выдаётся имя `_DATA` , так как секция **.data** объявляется в текущей версии включаемого файла `io.inc` *последней*.

Например, при вызове макрокоманды в секции данных:

```
.data
   S   db @CatStr(!",Revers(a,b,c,1,2,3),!"),0
. code
   outstrln "Исходный список=a,b,c,1,2,3"
   outstr   "Перевёрнутый список="
   outstrln offset S
```

На этапе компиляции будет вывод:

```
OUT: Input List=a,b,c,1,2,3
OUT: Output List=3,2,1,c,b,a
```

На этапе счета будет вывод:

```
OUT: Исходный список=a,b,c,1,2,3
OUT: Перевёрнутый список=3,2,1,c,b,a
```

В заключение рассмотрим пример, когда макроопределение изменяет себя во время компиляции. Ранее отмечалось, что при выполнении машинная программа может изменять себя (быть самомодифицирующейся), макроопределение тоже может быть самомодифицируюшимся, но уже *во время компиляции*. Например, рассмотрим макроопределение

```
SelfModify macro
   echo Old Macros SelfModify
   SelfModify macro
      echo New Macros SelfModify
      SelfModify macro
         echo Once More Macros SelfModify
      endm
   endm
endm
```

Для макрокоманд

```
SelfModify
SelfModify
SelfModify
```

на этапе компиляции будет вывод

```
Old Macros SelfModify
New Macros SelfModify
Once More Macros SelfModify
```

Разумеется, на самом деле это макроопределение не изменяет свой текст, а просто вставляет в программу новое макроопределение с таким же именем, при этом старое макроопределение, как уже говорилось, становится невидимым для макропроцессора. К сожалению, это макроопределение не допускает самомодификацию более двух раз.

На этом заканчивается описание возможностей Макропроцессора.

11.1. Сравнение процедур и макроопределений

Все познается в сравнении.

Фридрих Ницше

Как уже говорилось, на Ассемблере один и тот же алгоритм программист, как правило, может реализовать как в виде процедуры, так и в виде макроопределения. Процедура будет вызываться командой **call** с передачей параметров по стандартным или нестандартным соглашениям о связях, а макроопределение – макрокомандой, также с заданием соответствующих параметров. Следует сравнить эти два метода разработки программного обеспечения между собой, оценив достоинства и недостатки каждого из них.

Для изучения этого вопроса будет рассмотрен пример какого-нибудь простого алгоритма и реализация его двумя указанными выше способами. Пусть, например, надо реализовать оператор присваивания `EAX:=Max(X,Y)`, где `X` и `Y` – знаковые целые значения размером в двойное слово. Вот реализация этого оператора в виде функции со стандартными соглашениями о связях:

```
Max proc
   push   ebp
   mov    ebp,esp
   mov    eax,[ebp+8]; X
```

```
    cmp    eax,[ebp+12];Y
    cmovge eax,[ebp+12]
    ret  8
Max endp
```

Тело этой функции состоит из 8 команд, а каждый её вызов занимает 3 команды, например:

```
;EAX:=Max(A,B)
    push A
    push B
    call Max
```

А вот реализация этой же функции в виде макроопределения (не будем принимать во внимание, что это макроопределение будет неправильно работать для вызовов вида `Max Z,eax`, Вы уже знаете, как можно исправить такую ошибку с помощью средств условной компиляции):

```
Max macro X,Y
    mov   eax,X
    cmp   eax,Y
    cmovl eax,Y
    endm
```

Как видно, каждый вызов макроопределения будет порождать макрорасширение в три команды, а каждый вызов процедуры занимает 3 команды, кроме того, сама процедура имеет длину 6 команд, да ещё и использует стековый кадр. Таким образом, для коротких алгоритмов выгоднее реализовывать их в виде макроопределений.[1] Всё, конечно, меняется, если длина макрорасширения будет, хотя бы 10 команд. В этом случае, если, например, в программе содержится 20 макрокоманд, то в сумме во всех макрорасширениях будет `20*10=200` команд. В случае же реализации алгоритма в виде процедуры (пусть её длина будет не 10, а даже 20 команд) и 20-ти вызовов этой процедуры потребуется всего `20*3+20=80` команд. Получается, что для достаточно сложных алгоритмов реализация в виде процедуры более выгодна, что легко понять, если учесть, что процедура присутствует в памяти только в одном экземпляре, а каждая макрокоманда требует своего экземпляра макрорасширения, которое будет располагаться в программе на месте этой макрокоманды.

С другой стороны, однако, макроопределения предоставляют программисту такие уникальные возможности, как настройка алгоритма на типы передаваемых параметров, легкую реализацию переменного числа параметров, хорошую выдачу своих собственных диагностик об ошибочных параметрах. Такие возможности для гибкого и надежного программирования весьма трудно и неэффективно реализовываются с помощью процедур и функций.

Исходя из всего вышеизложенного, наиболее перспективным является *гибридный* метод. При этом надо реализовать алгоритм в виде макроопределения, в котором производится настройка на типы параметров и выдачу диагностики (такие действия не порождают дополнительных команд в полученной программе), а потом, если нужно, вызывается *процедура* для реализации основной части алгоритма. Именно так устроены достаточно сложные макроопределения., например, макроопределение **inint** в файле `io.inc`.

На этом заканчивается изучение макросредств языка Ассемблера, ещё раз напомним, что рекомендуется изучить эту тему в учебнике по языку Ассемблера.

Вопросы и упражнения

1. Что такое Макропроцессор ?
2. Как для макрокоманды ищется соответствующее ей макроопределение ?
3. Чем являются фактические параметры макрокоманды ?
4. Что делает макропроцессор, если фактических параметров в макрокоманде больше или меньше, чем формальных параметров в макроопределении ?
5. Что такое макрорасширение ?

---

[1] Тот факт, что короткие алгоритмы часто выгоднее реализовывать не в виде процедур и функций, а в виде макроопределений, нашел отражение и при разработке языков программирования высокого уровня. Так, в некоторых языках высокого уровня, например, в языке Free Pascal, существуют так называемые встраиваемые (**inline**) процедуры и функции, вызов которых во многом производится по тем же правилам, что и вызов макроопределений.

6. Как при помощи макросредств можно поднять уровень языка Ассемблер, приблизив его к языкам высокого уровня ?
7. Что такое средства условной компиляции и для чего они предназначены ?
8. Что такое локальные имена и для чего они нужны ?
9. Как программист может зафиксировать ошибки в параметрах макроопределения ?
10. Как программист может прекратить процесс построения макрорасширения, и когда это нужно делать ?
11. Чем отличаются макропроцедуры от макрофункций ?
12. Как задать значение параметра по умолчанию ?
13. Что такое числовые макропеременные, когда они порождаются и уничтожаются ?
14. Значения какого типа могут принимать числовые макропеременные в Макроассемблере ?
15. Как сделать макроопределение с переменным числом параметров ?
16. Что такое значение параметра макроопределения по умолчанию и как его задать ?
17. Чем задать в макроопределение фатальную ошибку и предупреждение ?
18. Что такое макрооператор? Приведите примера макрооператоров в Ассемблере.
19. Используя средства условной генерации (условные макрооператоры), исправьте описанное в этой главе макроопределение с именем **maxn** так, чтобы оно правильно обрабатывало регистр EAX в списке своих фактических параметров.
20. Когда можно считать, что у фактического параметра макроопределения есть тип и для чего может понадобиться настроить макроопределение на типы передаваемых ему фактических параметров ?
21. Как работает макроцикл **forc** ?
22. Обоснуйте, почему перед макрокомандой нельзя ставить метку без двоеточия (т.е. метку области памяти), хотя саму макрокоманду можно использовать, например, в сегменте данных. Для этого нужно вспомнить, как обрабатываются макрокоманды с метками.
23. Чем в Ассемблере отличаются циклы **while ... .endw** от **while ... endm** ?
24. Как работает оператор Ассемблера **.type** ?
25. Когда для реализации некоторой подзадачи следует использовать процедуру, а когда макроопределение ?
26. Объясните, для чего предназначено показанное ниже макроопределение и как к нему следует обращаться:

```
Prolog macro Name:Req,Reg,LocVar
 Name  proc
       push ebp
       mov  ebp,esp
 ifnb <LocVar>
       sub  esp,VocVar
 endif
 for i,<Reg>
       push i
 endm
        endm
```

---

[i] Замечание для продвинутых читателей. В отличие от препроцессора языка С, который действительно полностью отрабатывает перед компилятором, взаимодействие Макропроцессора и Ассемблера при обработке программного модуля более сложное. На самом деле Ассемблер и Макропроцессор обрабатывают программу попеременно, предложение за предложением, "рука об руку" (иногда говорят, что Ассемблер и Макропроцессор *сильно связаны*). Однако конечный этап компиляции – генерация объектного модуля – выполняется Ассемблером уже *после* полного завершения работы Макропроцессора. Пока это не будет приниматься во внимание, можно считать, что сейчас работа Макроассемблера рассматривается на *внешнем* уровне. Несколько более полно взаимодействие Макропроцессора и Ассемблера (уже на *концептуальном* уровне) будет рассмотрено в главе, посвященной схеме работы компилятора Ассемблера. Здесь следует только отметить, что такая совместная работа Макропроцессора и компилятора Ассемблера и порождают те развитые возможности макросредств,

которых нет в языках высокого уровня, где Макропроцессор в языках высокого уровня работает именно как препроцессор (до начала работы самого компилятора), поэтому тесное взаимодействие препроцессора и компилятора невозможно.

[ii] Замечание для продвинутых читателей. В стандарте Паскаля можно попытаться сделать элементы такого массива записями с вариантами, в языке Free Pascal – использовать так называемые безтиповые массивы. Однако во всех этих случаях, даже если не принимать во внимание отсутствие контроля типов и следующее отсюда значительное понижение надежности программы, придется передавать в функцию дополнительный параметр, задающий тип элемента массива, а внутри функции будет по существу находиться оператор выбора, разветвляющий вычисление по разным типам данных. Некоторым аналогом этого механизма Макроассемблера в объектно-ориентированных языках являются *шаблоны* и *прототипы* процедур и функций.