Понимание fasm

(Текст «**Understanding the flat assembler» из пакета fasm**)

Этот текст представляет собой руководство для продвинутых пользователей, которое суммирует некоторые правила fasm и учит выбирать новые техники, комбинируя его различные возможности. Также мы ставим своей целью объяснить некоторые особенности, которые могут запутать и которые не соответствуют ожиданиям, пока не изучишь точно, как это все работает и взаимодействует на различных уровнях языка, используемого в fasm, который по некоторым аспектам отличается от других ассемблеров.

Это пособие, однако, не может заменить основное руководство по fasm. Мы полагаем, что вы уже получили базовые знания по языку fasm, и теперь можете их углубить.

Ассемблер как компилятор и ассемблер как интерпретатор

Реализации языков программирования могут относиться к одному из двух классов: компиляторы и интерпретаторы. Интерпретатор — принимает программу, написанную на каком-либо языке и исполняет ее. А компилятор просто транслирует программу, написанную на одном языке в программу на другом языке – чаще всего на машинном языке программирования, так, чтобы она могла выполняться процессором.

С этой точки зрения ассемблер представляет собой разновидность компилятора – он берет программу на языке ассемблера (исходный код) и транслирует ее в машинный язык. Однако, имеются некоторые различия. Когда компилятор выполняет трансляцию из одного языка в другой, то он, очевидно, создает программу на другом языке, которая, если ее запустить (с помощью интерпретатора или процессора), будет выполняться также и давать тот же результат. Если исключить такие детали как выбор возможных языковых конструкций, которые дают один и тот же результат, компилятор свободен в своих предпочтениях, хотя от него ждут наилучшего выбора. Различные компиляторы, которые осуществляют трансляцию между двумя одинаковыми языками, могут давать различные результаты, хотя программы будет выполняться одинаково.

С ассемблером все несколько иначе, поскольку имеется почти точное соответствие между командами языка ассемблера и командами машинного языка, в которые они транслируются. Фактически в большинстве случаев вы знаете точно в какие байты будет оттранслирована конкретная конструкция языка ассемблера. Это делает ассемблер немного похожим на интерпретатор.

Возьмем самую очевидную директиву:

```
db 90h
```

которая предписывает ассемблеру поместить один байт со значением 90h в текущую позицию в результирующем коде. Это более похоже на то, как если бы ассемблер был интерпретатором, и машинный язык, генерируемый им, был бы просто результатом работы интерпретатора. Даже команды, которые фактически представляют собой команды машинного языка в которые они транслируются, можно рассматривать как директивы, которые предписывают ассемблеру сгенерировать код данной команды и поместить его в текущую позицию на выходе.

Также можно не использовать мнемонических инструкций вообще в исходном коде и использовать только директивы DB, чтобы создать, например, какой-нибудь текст. В этом случае результат уже не программа вообще, так как она не содержит каких-либо машинных команд. Это делает ассемблер еще более похожим на интерпретатор.

Но когда кто-то пишет программу на языке ассемблера, он обычно мыслит так, как будто пишет программу на машинном языке. Ассемблер как раз и делает задачу создания программы на машинном языке проще, обеспечивая легкое запоминание имен команд (по этой причине называемых мнемониками), позволяя использовать метки для обозначения адреса в памяти, а также другие именованные величины и затем автоматически рассчитывая для них соответствующие адреса и смещения.

Когда мы пишем некоторую простую последовательность команд на языке ассемблера:

```
mov     ax,4C00h
int     21h
```

то обычно не думаем о них, как директивах для интерпретации, которые генерируют команды процессора. Мы в действительности думаем о них, как если бы они и были командами, которые они генерируют, мы думаем о программе, которую мы пишем с помощью языка ассемблера, а не о программе на языке ассемблера. Однако в действительности есть две программы, соединенные в одну, два уровня восприятия одного и того же кода. Это дает ассемблеру новое качество: инструмент, который является одновременно и компилятором, и интерпретатором.

Уровень исполнения и уровень интерпретации

Давайте взглянем на два простых примера на языке ассемблера, в которых регистр EAX складывается с самим собой, повторяя эту операцию, пять раз. Первый пример использует регистр ECX для подсчета повторений:

```
        mov     ecx,5
square: add     eax,eax
        loop    square
```

Из этого фрагмента генерируются три машинные команды, и когда процессор выполняет машинный код, полученный из этого фрагмента, пять раз выполняется операция сложения регистра EAX с самим собой. Это делается путем уменьшения содержимого ECX на единицу и перехода снова к команде сложения, если содержимое ECX не равно нулю. Второй пример выглядит проще:

```
repeat 5
        add     eax,eax
end repeat
```

На этот раз директива ассемблера используется, чтобы повторить инструкцию пять раз. Но здесь не используется команда перехода. То, что делает ассемблер, встретив данную инструкцию, в действительности тоже самое, как если бы мы написали следующие команды:

```
        add     eax,eax
        add     eax,eax
        add     eax,eax
        add     eax,eax
        add     eax,eax
```

Ассемблер генерирует пять копий одной и той же машинной команды. Если команда LOOP используется чтобы создать цикл времени исполнения – когда процессор исполняет машинный код, директива REPEAT создает цикл времени ассемблирования, она повторяет интерпретируемый блок, который находится между ней и директивой END. В нашем случае в блоке всего одна команда, но как мы сказали ранее, эта команда фактически директива, которая вызывает создание машинной команды. Таким образом, мы получаем пятикратное повторение интерпретируемой директивы ADD, которая каждый раз генерирует код команды сложения EAX с самим собой. Это один из хороших примеров уровня интерпретации языка ассемблера.

Тем не менее, здесь имеется также и уровень времени исполнения: то что мы реально получаем, это пять копий одной и той же машинной команды,

выполняемых одна за другой. Следующий пример – в большей степени язык интерпретации:

```
A = 1

repeat 5
  A = A + A
end repeat
```

Этот фрагмент на диалекте языка ассемблера fasm определяет переменную времени ассемблирования с именем A и затем пять раз складывает ее саму с собой. Все что здесь происходит это интерпретация, здесь нет машинного кода или чего-то другого, влияющего на генерацию вообще. Фрагмент мог бы повлиять на уровень времени исполнения, если величина A в дальнейшем была бы использована в какой-либо машинной команде.

Использование fasm, в качестве чистого интерпретатора

Как уже отмечалось на примере с директивой DB, результат работы ассемблера может и не быть программой вообще. В таком случае мы уже не рассматриваем ассемблер в качестве компилятора и ассемблирование становится эквивалентно работе обычного интерпретатора. Скопируйте код ниже в файл interp.asm, чтобы увидеть это на примере:

```
file 'interp.asm'
repeat $
  load A byte from %-1
  if A>='a' & A<='z'
    A = A-'a'+'A'
  end if
  store byte A at %-1
end repeat
```

Эта программа написана полностью с помощью интерпретирующего языка fasm, с использованием некоторых его расширенных возможностей. В начале он (fasm) начинает читать все содержимое файла interp.asm (где она сама и содержится) начиная с текущей позиции, которая всегда с момента начала работы ассемблера равна нулю и так повторяет для всех байтов файла. Поскольку $ всегда равна текущей позиции, то в конце загрузки $ становится равна длине файла. Затем берет каждый байт, преобразует его к верхнему регистру, на основе простого алгоритма, записывает обратно в текст программы. Так программа конвертирует все содержимое файла и заносит все в конечном итоге в выходной файл. Таким образом это пример программы преобразования текста, написанной на интерпретирующем языке

и, поскольку выходным файлом тоже является текст, не содержащий машинных команды, функция ассемблера как компилятора полностью здесь отсутствует.

Разрешение кода

Последние примеры продемонстрировали функционирование ассемблера в качестве интерпретатора, что полностью объясняет, что он делает. Но после того, как мы признаем (я надеюсь), что ассемблер можно рассматривать в качестве интерпретатора, самое время показать, где эта интерпретация терпит неудачу. Дело в том, что в действительности ассемблер является и компилятором, и интерпретатором одновременно, поэтому оба эти термина в отдельности не объясняют точно, что делает ассемблер fasm.

Одна из ключевых возможностей, которую должен иметь ассемблер, помечать различные места в ассемблерном коде или данные, с помощью произвольно выбранного имени и затем использовать эти имена вместо обычных адресов в команде. Это особенно важно для команды перехода, так чтобы можно было писать инструкцию перехода в какое-либо место в программе, используя имя метки этого места, и при этом программист не должен заботиться о том, какой это в действительности адрес. Однако для программиста нужны переходы как вперед, так и назад. При переходе назад ассемблер уже знает, что это за адрес, так как он встречал имя и интерпретировал его как метку. Однако, когда переход осуществляется вперед, ассемблер как интерпретатор, терпит неудачу. Нет способа узнать, что это может быть за адрес, если он еще не дошел до этой метки.

Но ассемблер все равно должен сделать это, и здесь он ведет себя снова как компилятор, а не интерпретатор: имея команды и метки на языке ассемблера, необходимо получить эквивалент команды и адресов на машинном языке. Существует не так много возможных способов достижения этого. В одном из них нужно в начале интерпретировать в начале весь исходный код, оставляя места в созданном машинном коде для величин, которые не известны в тот момент, когда это необходимо. Затем, когда такая обработка закончится, и все значения меток станут известны, все эти пустые места заполняются правильными значениями.

Однако есть и другая проблема, если ассемблер должен сгенерировать возможно более оптимальный код (что касается размера, он также влияет на скорость выполнения). И машинный язык данной процессорной архитектуры, которую поддерживает ассемблер (архитектура x86 – единственная, поддерживаемая fasm) имеет инструкции разной длины в зависимости от диапазона используемых адресов. Например, в архитектуре x86 имеется короткая форма команды перехода на адреса, которые находятся

не дальше 128 байтов вперед или назад, и длинная форма, аналогичного перехода, если нужно перейти на более далекие адреса. В таких случаях ассемблер может попытаться сгенерировать короткую форму, когда это возможно, но, если короткая форма используется, когда значение адреса перехода еще не известно, может оказаться, что такой адрес находится слишком далеко, и в данном месте следует использовать длинную команду перехода.

Таким образом, чтобы сделать возможной более эффективную оптимизацию, fasm использует другой подход. В начале он интерпретирует весь текст программы и выбирает каждый раз наименьшую форму команд, если значение адреса еще не известно, выбор им самой короткой формы команды основывается на предположении, что наилучшая оптимизация возможна. Но когда он заканчивает и узнает все значения меток, он не возвращается назад, чтобы проверить, можно ли только что созданный им код, заполнять правильными величинами. Он просто интерпретирует весь текст с начала, но на этот раз использует значения меток, собранных в предыдущем подходе, чтобы предсказать корректное значение адресов в местах, где они ранее не были известны. Если исходить из того, что наилучшая оптимизация возможна, ассемблирование на этот раз производится также, как в предыдущем случае, только соответствующие места заполняются правильными значениями.

Но также может случиться, что на этот раз некоторые команды изменят свою длину, по сравнению с предыдущим проходом. В таком случае все метки, определенные после такой инструкции, получат адреса со смещением, и предсказания, сделанные на основе адресов из предыдущего прохода, окажутся не точными. Но это не смущает ассемблер. Когда он понимает, что некоторые из адресов изменились во время второй интерпретации программы, он решает интерпретировать программу еще раз, взяв более точные значения меток и пытаясь на этот раз предсказать более правильные значения. Этот процесс может повторяться много раз, в надежде, что предсказанные величины наконец совпадут с их истинными значениями и только тогда ассемблер решит, что он закончил работу и запишет в выходной файл результат последнего прохода программы.

Весь процесс называется разрешением кода. Этот подход не только позволяет получить оптимальный код из данной последовательности команд и меток, но также позволяет интересным образом комбинировать уровень интерпретации и уровень компиляции. Посмотрите на следующий пример:

```
alpha:
   repeat gamma-beta
      inc     eax
```

```
   end repeat
beta:
      jmp     alpha
gamma:
```

Ассемблеру необходимо разрешить значения меток beta и gamma, так как они используются до их определения (такую ситуация обычно называют «ссылка вперед»). И эти значения определяют сколько раз команда INC будет повторяться. Уровень интерпретации здесь не очень заметен, так как все что нужно, это разрешение имен и ассемблер должен найти правильное решение самостоятельно. Но может оказаться, что решения вообще нет, например, в том случае, если из gamma мы будем вычитать alpha, а не beta. В этом случае очевидно разность будет каждый раз больше количества повторений, так что исходный код программы содержит противоречие и ассемблер не сможет найти решение.

В предыдущем примере уровень интерпретации как бы не виден на фоне проблемы разрешения. Но в следующем примере мы имеем случай, когда два уровня явно сосуществуют:

```
mov     eax,gamma
A = 1
repeat 5
   A = A*%
end repeat
label gamma at $+A-7
```

A – переменная времени ассемблирования и ее конечное значение вычисляется интерпретатором. Но затем эта величина используется, чтобы определить метку gamma, которая ссылается вперед и требует разрешения. Фактически мы могли бы даже использовать знак = для определения величины gamma, так как ассемблер будет рассматривать такое определение (когда определение gamma будет встречаться только один раз в ассемблируемом тексте) как определение глобальной константы, а не как переменной времени ассемблирования, поэтому она может ссылаться вперед. Это ведет нас к еще одному примеру разрешения кода:

```
dd      x,y

x = (y-2)*(y+1)/2-2*y
y = x+1
```

Здесь знак = используется, чтобы объявить числовые константы, на которые имеются ссылки вперед и ассемблер должен разрешить эти величины. Кроме

того, эти величины зависят друг от друга и довольно трудно сразу сказать есть ли решение, где определены все величины. Однако, если мы попробуем ассемблировать этот текст с помощью fasm, то он сможет найти решение за несколько проходов, в результате x=6 и y=7. Это действительно редкий случай, когда алгоритмы, предназначенные для оптимизации машинного кода, могут решить систему подобных уравнений, но это также дает нам чистый пример того, что из себя представляет разрешение кода.

Переместив директиву DD в последнюю строку представленного выше примера, мы получим, что только на y есть ссылка вперед, в то время как x оказывается добавочной переменной, необходимой для расчета независимой величины y. Мы можем даже разбить эти вычисления на несколько отдельных операций над x:

```
x = y-2
x = x*(y+1)
x = x/2-2*y
y = x+1
      dd     x,y
```

чтобы еще раз выделить уровень интерпретации. Но отметим, что мы не можем определить самостоятельную величину y без такой промежуточной переменной, так как fasm не позволяет ссылку вперед на переменную внутри ее собственного определения, так что определения, включающие предыдущие значения этой величины резервируются для переменных времени ассемблирования (как x в данном примере). Разрешение кода становится еще более сложным вопросом, когда мы рассматриваем директиву IF со всеми возможными сложными зависимостями, которые она создает. Вы можете найти некоторые примеры таких задач, в разделе, посвященном многопроходности в руководстве по fasm.

Конечно для данного набора зависимостей может существовать несколько корректных решений. В случае простых зависимостей между формой команды и значениями меток fasm пытается найти решение с возможно более короткими кодами команд, но это не означает, что ассемблер вообще всегда находит решение с самым коротким кодом, в некоторых случаях он не может найти решение, хотя оно и существует. Это потому, что предсказательные алгоритмы, которые он использует, ориентируются на получение качественного машинного кода, а не на решение сложных арифметических проблем, которые можно выразить с помощью языка. Но если fasm закончил свою работу без сообщений об ошибке, вы можете по крайней мере быть уверены, что все зависимости реализованы, а выходной код корректен по отношению к ним.

Препроцессор

Но у fasm есть еще один уровень, который делает всю картину более сложной. Это препроцессор. Основная особенность препроцессора заключается в том, что он обрабатывает весь исходный текст программы до того, как она попадает ассемблеру. То, что он обрабатывает исходный текст позволяет создавать с помощью простых операторов более сложные структуры, состоящие из ассемблерных команд. Имеется набор специальных директив, называемых директивами препроцессора, которые интерпретируются только препроцессором и удаляются из текста до передачи его ассемблеру. Все остальное, что имеется в тексте программы, передается препроцессором ассемблеру для обработки. Например, у нас есть такой текст:

```
mov     ax,bx
    include 'nop.inc'
mov     cx,dx
```

и содержимое файла NOP.INC только команда:

```
nop
```

Что делает с ним препроцессор? Он интерпретирует вещи, которые он распознает, типа директивы INCLUDE, которая указывает, что следует поместить все содержимое файла NIP.INC в место, где она стоит. Все, что препроцессором не распознается, остается нетронутым. Так что в конечном итоге ассемблеру передается:

```
mov     ax,bx
nop
mov     cx,dx
```

Отметим, что для самого ассемблера не существует такой директивы как INCLUDE. Препроцессор приготовил нужную последовательность инструкций для него. Если, например, есть директива IF перед директивой INCLUDE, а директива END IF внутри включаемого файла, то ни препроцессору, ни ассемблеру не на что жаловаться: для препроцессора их не существует, он просто оставит их ассемблеру, но и ассемблер не увидит каких-либо противоречий, так как после препроцессора останется непрерывная и корректная последовательность ассемблерных директив.

Когда препроцессор вставляет новые строки в текст программы, например, заменяя директиву INCLUDE всеми строками из указанного там файла, он обрабатывает все эти новые строки, прежде, чем идти дальше. Т.е. включаемый файл также может содержать директивы препроцессора, которые распознаются и соответствующим образом обрабатываются. Но и

строки, не содержащие препроцессорных директив, также могут измениться при их обработке препроцессором. Это происходит вследствие замещений, которые выполняет препроцессор. Такие замещения выполняются с так называемыми строковыми константами. Определяется строковая константа с помощью директивы EQU так:

```
A equ +
```

При таком определении имя A заменяется символом + всюду, где это имя будет обнаружено препроцессором после данного определения. Отметим, что поскольку препроцессор проходит один раз через весь текст программы (т.е. действует как чистый интерпретатор), замещение происходит только после определения A. Например:

```
      mov    eax,A
A equ ebx
      mov    eax,A
```

после препроцессинга:

```
      mov    eax,A
      mov    eax,ebx
```

Еще одна важная мысль касательно строковых констант – фактически они не константы, а переменные времени работы препроцессора, аналогичные переменным времени ассемблирования, которые задаются директивой =. Поэтому их можно переопределять и по-видимому, мы должны бы называть их строковыми переменными, хотя в руководстве они и называются константами (по чисто историческим причинам). Аналогично переменным времени ассемблирования мы можем переопределять такие переменные, с использованием их предыдущих значений, получая новые значения. Например:

```
A equ 2
A equ A+A
```

Определяет строковую константу со значением 2+2. Это работает потому что осуществляется замена строковых переменных их ранее определенными значениями в строке, содержащей EQU, но только после этого ключевого слова. Таким образом также:

```
A equ 2
B equ +
A equ A B A
```

определяет строковую переменную со значением 2+2. С точки зрения препроцессора пробел важен только там, где он используется чтобы отделить

имена, которые бы слились в одно имя, если бы их не разделять. Любые другие пробелы игнорируются и удаляются – для препроцессора важна только последовательность атомов (лексем), которые он и учитывает.

Давайте теперь подытожим в чем различие между переменными времени работы препроцессора и переменными времени ассемблирования. Одно различие очевидно. Переменные времени ассемблирования сугубо числовые и всегда равны какому-то числу или значению адреса, тогда как строковые переменные могут иметь любое значение (они могут быть даже пустыми, если после директивы EQU стоят только пробелы и комментарии). Тот факт, что строковые переменные предназначены для текстовой подстановки можно продемонстрировать следующим примером:

```
nA = 2+2
      mov     eax,nA*2
sA equ 2+2
      mov     eax,sA*2
```

Первая строка определяет константу nA, присваивая ей числовое значение 4. В следующей строке эта величина отправляется в регистр EAX, и предварительно умножается на 2. Таким образом инструкция, которая в дальнейшем будет переведена в машинный язык - MOV EAX,8. В третьей строке определяется строковая константа sA, которой присваивается текстовое значение 2+2, и таким образом инструкция в последней строке будет изменена препроцессором на MOV EAX,2+2*2, что будет воспринято ассемблером уже как MOV EAX,6. Этот пример демонстрирует, что мы должны быть аккуратны, имея дело со строковыми переменными и всегда думать, что мы получим в тексте программы, когда вместо переменной будет подставлено текстовое значение, которое ей было присвоено. Еще одна тонкость здесь заключается в том, что директивы EQU и = обрабатываются на различных уровнях. Все замены, которые вызывает EQU делаются до того, как весь текст программы передается ассемблеру. Давайте взглянем на особенности в следующем примере:

```
A = 0
X equ A = A+
X 4
dd A
```

После того, как программа будет «пережевана» препроцессором, вот что будет передано «на закуску» ассемблеру:

```
A = 0
A = A+4
dd A
```

Таким образом мы окончательно получим 32-битовое данное со значением равным 4. Этот пример показывает, как вы можете получить взаимодействие различных уровней. Однако такая задача требует, чтобы вы точно понимали, что какому уровню принадлежит. Мы еще обсудим далее вопрос смешения различных уровней.

Макроинструкции

Макроинструкции (часто для краткости называемые макросами) еще один инструмент препроцессора. Макрос – это инструкция для препроцессора и когда вы используете макрос с некоторым набором параметров, препроцессор использует эту инструкцию, чтобы создать новые строки программы, подставляя их в место, где макрос вызывается.
Определение макроса рассматривается препроцессором как одна большая директива (которая может охватывать несколько строк), и сама инструкция никак не обрабатывается препроцессором и не передается ассемблеру. Но когда вы вызываете макрос, однако, и препроцессор, используя инструкцию, создает новые строки, он также обрабатывает все эти строки, прежде чем идти дальше, подобно тому как он это делает в директиве INCLUDE. Давайте рассмотрим простой макрос:

```
macro Def name
{
   name equ 1
}
```

Так как все, что что стоит между скобками это содержимое макроса, препроцессор не обращает внимание на директиву EQU – все строки здесь это часть инструкции. Все, что препроцессор делает с этим, это отмечает для себя, что инструкция с именем Def есть и идет дальше, не передавая ничего ассемблеру. Но что происходит, когда вы используете макрос? Допустим, мы используем его так:

```
Def A
```

Препроцессор заменяет эту строку строками, генерируемыми на основе инструкции для макроса Def, в данном случае это будет только строка:

```
A equ 1
```

Эта новая строка интерпретируется обычным образом – препроцессор распознает директиву EQU и применяет ее к константе A.

Это также означает, что любая строка, генерируемая макросом, может содержать вызов другого макроса. Отметим, однако, что нет возможности в макросе генерировать самого себя, так как во время генерации строк макроса, сам макрос отключен и используется его предыдущее значение (подобным образом работает директива purge, единственное отличие в том, что макрос становится доступным снова, когда процесс вызова завершается). Это делает трудным использование рекуррентных макросов (но возможным, мы это обсуждаем), однако смысл такого поведения заключается в том, что можно слишком усложнить определение макроса, как об этом сказано в руководстве.

Имеются специальные операторы и директивы, которые можно использовать внутри определения макроса – они управляют тем, как препроцессор выполняет инструкцию генерации новых строк. Так что очевидно, что, когда генерируются новые строки, эти директивы должны быть выполнены до того, как препроцессор закончит обрабатывать эти строки. Например:

```
macro Inc f
{
   include `f#'.inc'
}

Inc win32a
```

Имеем таким образом инструкция препроцессору преобразовать первое слово параметра f и затем объединить со строкой '.inc'. Если параметр состоит из одного слова, то результат такой операции будет строка и вызов макроса Inc сгенерирует такой результат:

```
   include 'win32a.inc'
```

данный результат затем будет обычным образом обработан препроцессором, так что будет загружен файл win32a.inc.

Обратный слеш (символ эскейп-последовательности или экранирования) может быть полезен в том случае, если вы хотите вставить в строку, которую генерирует макрос, имена, которые в противном случае (без использования слеш) будут сразу интерпретироваться при выполнении инструкции макроса. Например,

```
macro Defx x
{
   \x db x
}
```

Здесь x параметр макроса, так что когда препроцессор выполняет инструкцию, то всюду, где встречает этот параметр, подставляет его значение. Однако в этом макросе определяется байт (переменная) с именем x, причем само значение переменной определяется параметром. Конфликт имен можно разрешить просто, дав параметру макроса другое имя, но здесь мы можете увидеть, как можно добиться желаемого результата используя escape-символ - экранирование. Когда генерируется строка препроцессор удаляет первый обратный слеш в начале каждого атома, никак не интерпретируя то что идет далее.

Наиболее частое использование экранирования встречается, когда один макрос определяется через другой. Поскольку строки, порождаемые препроцессором из макроса, также обрабатываются препроцессором, они сами могут содержать определение макроса. Но при этом необходимо избежать того, чтобы операторы внутри дочернего макроса интерпретировались бы в процессе развертывания родительского макроса. Давайте рассмотрим теперь несколько более сложный пример:

```
macro Parent [name]
{
 common
   macro Child [\name]
   \{
   \common
 forward
     name dd ?
 common
   \forward
     \name dd ?
   \}
}
```

и посмотрим, что происходит, когда мы вызываем макрос следующим образом:

```
Parent x,y
```

Не экранированные директивы макроса и имена параметров, интерпретируются при развертывании родительского макроса, тогда как экранированные имена (лексемы) остаются в определении дочернего макроса. В результате:

```
macro Child [name]
{
 common
```

```
   x dd ?
   y dd ?
 forward
   name dd ?
}
```

Рекомендуем подробнее рассмотреть данный пример, чтобы точно понять, что здесь происходит.
Ясно почему мы должны различать разные уровни, используя экранирование, как в выше представленном примере. Но иногда задают вопрос, зачем мы должны также экранировать закрывающие фигурные скобки.

Такое экранирование, очевидно, помогает отслеживать (особенно если есть несколько уровней вложения макросов) имена, которые принадлежат конкретным макросам и которые должны иметь столько же обратных слешей в начале, сколько имеют скобки, ограничивающие данный макрос. А в руководстве объясняется, что экранирование закрывающих фигурных скобок необходимо, потому что первая закрывающая скобка, интерпретируется препроцессором как конец макроопределения, так что, если вы не отметите обратным слешем закрывающую скобку дочернего макроса, оно будет принят за конец макроса родительского.

Но почему препроцессор не может сосчитать все скобки, чтобы определить какая из них закрывает какой блок? Ответ скрыт в том, что уже сказано. При разворачивании макроса препроцессор добавляет строку начала другого макроса, но не его конец, как это показано в примере альтернативного определения синтаксиса макроса в руководстве. Во время процесса генерации строк макроса препроцессор генерирует и определение другого вложенного макроса и добавляет строки из определения, пока не встретит закрывающую скобку (отметим, что препроцессор не обрабатывает генерируемые строки вложенного макроса и таким образом единственный способ закрыть такое определение – среагировать на закрывающую фигурную скобку; используя директиву FIX можно сгенерировать закрывающую скобку из другого идентификатора, как показано в руководстве).

Могут возникнуть и другие проблемы при генерации блока макроса, такие, например, как открывание скобок, не связанных с блоком другого макроопределения или при генерации блоков повторения. В целом все эти факторы являются причиной того, что препроцессор всегда трактует закрывающие скобки, как конец определения макроса, и таким образом следует экранировать все фигурные скобки, которые должны быть сгенерированы внутри макроса. Конечно, вы можете не экранировать

открывающие скобки, но это не рекомендуется делать, чтобы сохранить понятность программного кода.

Непосредственные макроинструкции

Есть ряд директив, которые определяют блок без имени, которые реализуются непосредственно в том месте, где они определены. Это директивы REPT, IRP, IRPS и также MATCH (этой директиве будет посвящен отдельный раздел), имеющие свои блоки, и мы называем их «непосредственными» микроинструкциями, чтобы подчеркнуть, что они реализуются (выполняются) непосредственно в том месте, где определены.

Они также отличаются от обычных макросов тем, как в них используются параметры, но их блоки представляют собой такую же инструкцию и генерируют новые строки также как обычные макросы. Это также означает, что, когда вы помещаете непосредственные макроинструкции внутрь другой макроинструкции, вы должны экранировать их соответствующим образом.

Чтобы показать, как непосредственные макроинструкции связаны с обычными макросами рассмотрим четыре эквивалентные конструкции

```
rept 4 i
{
  ; ...
}

irp i, 1,2,3,4
{
  ; ...
}

irps i, 1 2 3 4
{
  ; ...
}

macro Inst [i]
{
  ; ...
}
Inst 1,2,3,4
```

Если поместите одну и ту же последовательность директив в каждый из четырех представленных выше блоков, результат будет идентичен. В

частности, директивы FORWARD, REVERSE и COMMON дадут один и тот эффект во всех случаях.

Из трех продемонстрированных выше непосредственных макроинструкций, IRP является единственной, где значения параметров в целом передаются также как в именованной макроинструкции – они отделяются друг от друга запятыми и т.о. могут иметь пустое значение (если только с помощью символа * после имени параметра не указано препроцессору, что параметр не может быть пустым). Также вы можете заключить значение параметра в угловые скобки (< и >), если необходимо, чтобы значение параметра содержало запятую.

Директива REPT сама по себе генерирует все возможные значения параметра – счетчика, можно задать базовое значение счетчика, или не использовать счетчик вообще. Однако, счетчик ведет себя также, как параметры макроса, получающие список возможных значений.
Что касается подробного объяснения директивы IRPC, то мы в начале рассмотрим несколько деталей по поводу того, как препроцессор воспринимает исходный текст.

Лексемы и генератор строк

Препроцессор не обрабатывает текст в той форме, как он хранится в файле. Он выделяет из каждой строки значимое для него содержание, игнорируя лишние пробелы и комментарии. Он, фактически, разбивает каждую строку источника на последовательность простых лексем, и потом строит весь процесс на основе этих лексем (в руководстве лексемы называются атомами, по историческим причинам, здесь мы будем использовать оба этих термина).

Первый класс лексем – символы. В руководстве утверждается, что это:
+-/*=<>()[]{}:,|&~#`
Каждый из этих символов, когда он используется в исходном тексте, имеет самостоятельную сущность и становится отдельной лексемой. Есть еще специальные символы такие как пробел и табуляция. Они не являются самостоятельными лексемами, но могут использоваться для разделения лексем. К подобным же типам символов относятся символы, указывающие на переход к следующей строке. Также следует указать на точку с запятой, которая указывает на начало комментария в строке, который игнорируется fasm. Также кавычки и обратный слеш, о которых мы скажем ниже.

Любая последовательность символов, не являющихся специальными, например, последовательность букв и цифр, становится именованной лексемой. Такая последовательность может быть разбита на отдельные

лексемы при помощи пробелов или других специальных символов. Например, в строке

```
mov     ax,2+1
```

содержится шесть лексем: первая именованная лексема MOV, затем именованная лексема AX (они разделены пробелами), запятая – специальная лексема, именованная лексема 2, символ плюс – специальная лексема и, наконец, именованная лексема 1. Добавление дополнительных пробелов в данную строку не повлияет на то, как это будет воспринимать препроцессор. Однако удаление пробелов между MOV и AX создаст новую именованную лексему.

Но есть еще один тип лексем – строки в кавычках. Когда первый символ лексемы одинарная или двойная кавычка, это интерпретируется, как строка в кавычках и все последующие символы, отличные от символов перевода строки, считаются принадлежащими одной лексеме, пока не встретится закрывающая кавычка (но, следуя принятому во многих ассемблерах положению - две кавычки подряд не закрывают строку, а трактуются как один символ). Так что в данной строке:

```
db '2+2'
```

имеются две лексемы – за именованной лексемой следует строка в кавычках.

Тем не менее, если кавычка не является первым символом лексемы, а стоит где-то внутри, она не имеет какого-то особого значения (кавычки не являются сами по себе специальными символами). Таким образом:

```
Jule's:
```

Только обычная именованная лексема, за которой следует символ «двоеточие».

Обратный слеш представляет собой специальный символ, который может в зависимости от позиции, иметь два значения. Если за слеш следует лексема, то он интегрируется в эту лексему. Слеш может использоваться рекурсивно, так что если за ним стоит еще один слеш, а за тем лексема, то оба интегрируются в эту лексему. Это свойство существенно только для того, чтобы экранировать имена в макроинструкциях.

Если за обратным слешем не следует лексемы, то он интерпретируется как символ продолжения строки, и лексемы на следующей строке присоединяются к лексемам текущей строки. Таким способом множество строк исходного текста могут сформировать для препроцессора одну строку.

Таким образом мы теперь понимаем, что строки, которые обрабатывает препроцессор, в действительности представляют собой последовательность лексем, а не просто текст. И такая последовательность появляется двумя путями – или из исходного программного текста или генерируется из макроинструкции. Т.е. мы имеем два вида генератора строк: генератор, читающий исходный текст (считыватель) и генератор, создающий строки на основе макроинструкции (обработчик макроинструкций).

Как мы видели ранее имеется группа специальных команд, таких, например, как оператор конкатенации или директива COMMON, которые могут выполняться в то время, как генерируются строки макроинструкции. Подобным же образом имеются специальные команды, которые понимает и которым подчиняется генератор, читающий исходный текст. Примерами являются обратный слеш, упомянутый ранее, а также директива FIX.

Директива FIX обеспечивает текстовую замену (когда лексема может заменяться некоторой последовательностью лексем), подобно директивам EQU или DEFINE, однако эти определения и замены осуществляются на стадии чтения исходного текста, когда строки готовятся для обработки препроцессором. Таким образом вы можете сделать определенные замены, до того, как начнется работа препроцессора.

Поскольку замена, определяемая директивой FIX осуществляется на стадии чтения, то она может реально пригодится для некоторых синтаксических настроек, в том случае, когда некоторые текстовые конструкции, используемые программистом в исходном файле, обрабатываются препроцессором уже как нечто совсем другое. В официальном руководстве содержится пример использования FIX для определения другого способа выделения макроинструкций – использование лексемы ENDM вместо закрывающей скобки. Это один из не многих примеров, когда такая возможность может реально пригодится.

Обработчик макроинструкций имеет больше специальных команд и операторов, чем считыватель. Он не только заменяет параметры макроинструкции их значениями и обрабатывает некоторые специальные команды, такие как COMMON или FORWARD, но и осуществляет конкатенацию лексем, соединенных оператором # или конвертирует лексему в строку, при наличии оператора `. Эти операторы не распознаются считывателем и, таким образом, они работают только внутри макросов. Что касается обратного слеша, то он удаляется из начала лексемы, когда обработчик макроинструкции помещает ее во вновь генерируемую строку. Если в начале лексемы стоит несколько обратных слеша, то удаляется только первый, таким образом лексема будет обрабатываться много раз при

обработке макроинструкции, пока наконец не будут удалены все обратные слеши и ни откроется другой символ.

Еще одна команда, предназначенная обработчику макроинструкций – директива LOCAL. Она сообщает, что определяется новые макропараметры в дополнение к обычным параметрам, и что величина, которая присваивается каждому из них, должна быть уникальной лексемой, генерируемой каждый раз, когда обрабатывается макроинструкция. Затем они замещаются этими величинами в каждом месте текста макроинструкции, где они встречаются, подобно тому, как это осуществляется с обычными параметрами. Таким образом, например, обработчик макроинструкций будет генерировать новые уникальные значения меток, каждый раз, когда они используются. Если имя локального параметра совпадает с именем обычного параметра, то он замещается локальным параметром.

```
macro Testing a
{
  db `a,13,10
  local a
  db `a,13,10
}

Testing one
```

Ассемблируя представленный выше пример и взглянув сгенерированный текст, вы увидите, какие значения будут присвоены параметру, определенному с помощью директивы LOCAL.

Есть еще одна важная деталь, демонстрируемая данным примером: обработчик макроинструкции выполняет преобразование лексемы только после того, как заменит параметры их значениями. Тоже правило применимо операции конкатенации над лексемами.

С другой стороны, как уже говорилось, оператор преобразования действует только на единичную лексему. Если параметр замещается последовательностью более, чем в одно слово, оператор ` воздействует только на первое из них. Это возможно трудно для понимания, например, выше представленный макрос не будет работать корректно, если параметр будет представлять собой последовательность отдельных слов, поскольку только первое из них будет преобразовано в строку, и таким образом, содержимое для директивы DB будет не корректным. Но это как раз тот случай, когда можно использовать директиву IRPS.

Обработка индивидуальных лексем

Директива IRPS это пример непосредственной макроинструкции. Она подобна директиве IRP, когда параметр принимает значение из списка величин. Однако в данном случае это не список, состоящий из элементов, разделенных запятой, а просто последовательность лексем (лексем любого типа), и для величины параметра берется на каждом шаге одна лексема, одна за другой.

Таким образом любая последовательность величин, которая передается в макрос, может быть разобрана и обработана по частям (лексемам). Например, таким образом можно обойти проблему, когда оператор преобразования лексемы может конвертировать за раз только одну лексему. Этот макрос обрабатывает все лексемы, передаваемые через параметр и выводит их при ассемблировании:

```
macro Tokens sequence
{
 irps token, sequence
 \{
   display \`token,32
 \}
}

Tokens eax+ebx*2+1
```

Отметим, что оператор преобразования, должен быть экранирован, иначе он будет обрабатываться при развертывании внешнего макроса и IRPS не «увидит» его (вместо этого в строке просто появится слово «token»).

Представленный выше пример не будет работать, если последовательность в последней строке модифицировать, добавив запятую – просто потому, что запятая будет рассматриваться как разделитель, отделяющий различные значения для параметров. Конечно, можно выделять параметр, который содержит запятую заключая величину в угловые скобки (< и >), но возможно также модифицировать макрос, чтобы он работал даже когда запятые используются в качестве разделителей:

```
macro Tokens [sequence]
{
 common irps token, sequence
 \{
   display \`token,32
 \}
```

```
}
```

Tokens mov ax,2+1

Директива COMMON берет все отдельные величины последовательности, переданные через параметр, и соединяет их со всеми разделителями в одну цепочку, которая в этом случае точно соответствует оригинальной строке. Но есть еще одно слабое место в макросах. Оператор преобразования ничего не делает, когда лексема является строкой в кавычках. Так что при выводе в данном макросе нет способа передать, является данная лексема единой строкой (в кавычках) или нет. Это можно решить, используя условное ассемблирование.

```
macro Tokens [sequence]
{
 common irps token, sequence
 \{
   if " eqtype token
     display """,token,""",32
   else
     display \`token,32
   end if
 \}
}
```

Но такое решение немного похоже на смесь яблок с апельсинами. Оператор EQTYPE выполняется на стадии ассемблирования и позволяет различать несколько классов синтаксических структур, которые распознаются ассемблером. Например, оператор может отделить вещественные числа от имен регистров. В частности, оператор различает строку в кавычках, потому что это отдельный класс синтаксических элементов в fasm – и по этой причине EQTYPE может быть использован здесь, чтобы определить, когда лексема представляет собой строку в кавычках.

Однако можно это узнать и во время работы препроцессора. Для того, чтобы добиться этой цели необходимо иметь возможность выполнять некоторые команды препроцессора, в зависимости от того, какую величину передают через параметр – для этого имеется директива MATCH с нужной функциональностью.

Условный препроцессинг

Простейшее описание директивы MATCH заключается в том, что эта непосредственная макроинструкция выполняется при условии, что указанная

в ней величина будет соответствовать заданному шаблону. Но имеется много специфических деталей, которые нужно знать для того, чтобы корректно применять эту директиву.

Синтаксис MATCH предусматривает, что первым указывается шаблон, а затем, после запятой значение, которое необходимо сравнивать с шаблоном. Значение идет последним по той причине, что в нем должна присутствовать последовательность лексем, включая запятые и другие специальные символы (исключая, конечно, открывающуюся фигурную скобку, которая для всех макроинструкций означает начало ее тела и таким образом указывает для MATCH, что сравниваемая с шаблоном величина заканчивается). Таким образом в определении MATCH имеется только одна служебная запятая, которая указывает на конец шаблона и начало сравниваемой величины, и все что следует за запятой до появления открывающей скобки есть сравниваемая величина, которая может быть, чем угодно.

Шаблон с другой стороны также подчиняется строгим правилам. В нем содержаться элементы, которые определяют какие лексемы или группы лексем должны быть во втором параметре. Эти элементы делятся на два вида: элементы, которые требуют точного соответствия и элементы типа «шаблон поиска».

Любой символ (исключая =) и любая строка в кавычках являются элементами точного соответствия. Также перед любой лексемой (даже лексемой, которая должна совпасть буквально в любом случае) может стоять знак =, чтобы указать на буквальное соответствие элемента. В частности, == и =, могут быть использованы для сравнения знака равно и запятой, соответственно, так как их нельзя использовать напрямую, поскольку они используются шаблоне в качестве служебных символов.

Все примеры ниже используют шаблоны, составленные из элементов для буквального соответствия, чтобы получить позитивный результат по отношению к величине, которая стоит после запятой.

```
match +,+
{ display "special character is matched as-is",13,10 }

match 'a','a'
{ display "the same with quoted string",13,10 }

match =a,a
{ display "the name token must be preceded with =",13,10 }

match =a=+='a' , a+'a'
```

```
{ display "and = may be actually used with any token",13,10 }
```

Что касается именованных лексем, причина по которой им необходим префикс = заключается в том, чтобы получить точное совпадение, в противном случае они рассматриваются как шаблоны поиска, которые могут соответствовать любой последовательности лексем, если есть по крайней мере одна лексема. Кроме того, параметр макроса, имеющий такое же имя инициализируется величиной, в точности равной последовательности лексем, с которыми он сравнивается.

В простейшем случае, если шаблон представляет собой некоторую именованную величину, он будет соответствовать величине, которая идет за запятой, если в ней есть хотя бы одна лексема. Следовательно, этот факт можно использовать для выполнения блока MATCH, при условии, что величина не является пустой.

```
macro check value
{
  match any,value
  \{ display "the value is not empty" \}
}
```

В этом примере директива MATCH встраивается в другой макрос (поэтому ее скобки должны быть экранированы) и может, таким образом, проверить что передается в качестве параметра, поскольку параметры заменяются их значениями во время разворачивания макроса препроцессором. Так что, когда строки, полученные при развертывании внешнего макроса, обрабатываются препроцессором, начинается интерпретация директивы MATCH, которая получает это самое значение от макроса, помещаемое после запятой.

Это не означает, что директива MATCH полезна только, когда она внутри макроса. Директива MATCH может быть удобна, и сама по себе, благодаря тому, что строковые константы в заголовке директивы, заменяются их значениями, перед тем как директива будет выполнена.

```
define X

match any,X
{ not preprocessed }

define X +

match any,X
```

```
{ display `any }
```

В представленном выше примере первая директива MATCH сравнивает величину, которая в действительности пуста, потому что символическая константа получает пустое значение до того, как выполнится MATCH. Поскольку шаблон должен сравниваться с чем-либо и условие таким образом не выполняется - блок MATCH не выполняется. Прежде чем выполнится вторая директива MATCH, символической константе присваивается значение, и в этом случае величина не пуста, таким образом шаблон соответствует значению X и становится равным этому же значению.

Единственная возможность получить положительный результат, если значение после запятой пусто, это использовать пустой шаблон, поскольку элементу шаблона должно быть поставлено не пустое значение из величины после запятой. С другой стороны, любая последовательность элементов шаблона стремится найти соответствие в наибольшем количестве лексем. Если шаблон состоит из двух или более элементов, то каждый элемент, кроме последнего ставится в соответствие одной лексеме из выражения справа (и положительный результат, таким образом возможен только в том случае, если выражение справа от запятой состоит по крайней мере из такого же количества лексем, что и шаблон). Рассмотрим пример:

```
match car cdr, 1+2+3
{
  db car
  db cdr
}
```

Здесь мы имеем в шаблоне два элемента (лексемы) и они оба теоретически могут получить соответствующее парное значение из последовательности справа. Однако, согласно алгоритму, которому придерживается fasm первый элемент шаблона получит одну лексемы, оставив все остальное второму элементу. Таким образом директива DB в первом случае получит “1”, тогда как вторая директива DB получит “+2+3”.

Добавляя некоторые элементы, требующие точного совпадения, между лексемами, можно добавить дополнительные условия в процесс выполнения директивы MATCH, но общих правил вполне достаточно, чтобы узнать результат, как в данном примере:

```
match first:rest, 1+2:3+4:5+6
{
  db first
  dd rest
```

```
}
```

Здесь первому элементу шаблона ставится в соответствие все справа от запятой до двоеточия (которое сравнивается буквально), поскольку это максимально-возможный вариант соответствия. Следовательно, для директивы DB мы получаем “1+2”, тогда как для директивы DD получается весь остаток справа от первого двоеточия, т.е. “3+4:5+6” (отметим, что синтаксис DD это допускает, как сегментный адрес и смещение, поэтому такое выражение ассемблируется корректно).

Буквальное сравнение строк в кавычках, хотя и менее полезно, чем другие случаи, но может быть использовано для специфической цели определения, является величина параметра строкой или нет. Макрос из предыдущей главы может быть переписан для использования в условном препроцессинге, как это показано ниже.

Отметим возможность не обычного использования директивы LOCAL (см. ниже) (не важно, относится она к внешнему макросу или внутреннему макроопределению - IRPS) - поскольку она не экранирована, она обрабатывается при развертывании внешнего макроса и не выдает результат в генерируемый внешний файл внутренней макрокомандой, таким образом не нарушает синтаксис:

```
macro Tokens [sequence]
{
 common irps token, sequence
 local output
 \{
   output equ \`token,32
   match \`token,token
   \\{ output equ """,token,""",32 \\}
   display output
 \}
}
```

Поскольку оператор преобразования не действует на строку в кавычках, конвертируемая лексема, т.о., будет равна своему оригиналу тогда и только тогда, когда оригинал представляет собой строку в кавычках. Чтобы проверить это условие в директиве MATCH в качестве шаблона берется преобразуемая лексема, что дает гарантию того, что каждый раз будет осуществляться буквальное сравнение (поскольку шаблон всегда будет строкой в кавычках). Таким образом содержимое блока директивы MATCH (который должен быть дважды экранирован, потому что находится внутри

двух макроопределений) будет обрабатываться только в том случае, если лексемы представляют собой строки в кавычках.

Обработка синтаксических конструкций пользователя

Благодаря шаблонам поиска, директива MATCH может делать гораздо больше, нежели просто проверять простые условия. Можно использовать ее для разбора некоторых, определяемых пользователями конструкций и, следовательно, определять макроинструкции с более гибкой структурой аргументов.

Допустим мы хотим определить макрос, который позволял бы писать "let al=4" вместо "mov al,4". Обычно макроинструкции принимают параметры, разделенные запятой, так что макрос "let" определен так, что "al=4" будет восприниматься как один параметр. Следовательно, нам необходимо встроить директиву MATCH в макрос, чтобы распознать подобную структуру в параметре и разбить ее на две части:

```
macro let param
{
  match dest==src,param
  \{
     mov dest,src
  \}
}
```

Поскольку символ = имеет специальное значение в шаблоне MATCH, конструкция == должна быть использована, чтобы найти буквальное соответствие для =. Таким образом MATCH даст позитивный результат при условии, что параметр макроса имеет структуру, когда перед знаком = и после него что-то стоит. В противном случае макрос не сработает. Но мы можем добавить еще синтаксиса, в виде дополнительной директивы MATCH следующим образом:

```
macro let param
{
  match dest==src,param \{ mov dest,src \}
  match dest++,param \{ inc dest \}
}
```

Эта версия не только позволяет генерировать "mov al,4" вызовом "let al=4", но также позволяет писать "let al++", получая команду "inc al". Но теперь рассмотрим возможность дополнительно писать "let al+=4", чтобы в

результате получить "add al,4". Мы могли бы просто добавить еще одну строку к макросу, представленному выше:

```
match dest+==src,param \{ add dest,src \}
```

Но несмотря на то, что данная строка обеспечивает ADD инструкцию, если мы напишем «let al+=4", шаблон "dest==src" сработает для этого случая с присвоением "dest" значения "al+" и мы получим ошибочную команду "mov al+,4", которая будет сгенерирована.

Чтобы решить данную проблему нам необходимо учесть информацию, что данная структура уже воспринята и предотвратить, чтобы другие директивы MATCH сработали в этом случае. Каждая директива MATCH независима, так что для того, чтобы учесть такой случай нужно задействовать строковую переменную. Первое решение, которое приходит на ум, представлено ниже. Отметим, что нам надо распознать шаблон "dest+==src" в начале, чтобы "al+=4" воспринималось со структурой +=, а не с единичным символом =.

```
macro let param
{
 local status
 define status 0
 match dest+==src,param
 \{
   add dest,src
   define status 1
 \}
 match =0,status
 \{
   match dest==src,param
   \\{
     mov dest,src
     define status 1
   \\}
 \}
}
```

Но это можно сделать проще. Вложенные директивы MATCH можно заменить одной:

```
 match =0 dest==src , status param
 \{
   mov dest,src
   define status 1
```

```
 \}
```

Запятая, которая отделяет шаблон от сравниваемого текста, обрамлена с двух сторон пробелами, чтобы было видно, что реально сравнивается. Так как первая лексема в шаблоне, которая сравнивается непосредственно, равно 0, а первая лексема в выражении справа равна или 0 или 1 – единственное возможное значение, то блок MATCH может выполниться только в том случае, если лексема status будет равна 0. Тогда оставшийся текст, который зависит от параметров макроса, сравнивается с "dest==src".
Полный текст макроса может иметь, т.о. вид:

```
macro let param*
{
 local status
 define status 0
 match =0 dest+==src , status param
 \{
   add dest,src
   define status 1
 \}
 match =0 dest==src , status param
 \{
   mov dest,src
   define status 1
 \}
 match =0 dest++ , status param
 \{
   inc dest
   define status 1
 \}
 match =0 any , status param
 \{
   err "SYNTAX ERROR"
 \}
}
```

Каждый из шаблонов содержит =0 для сравнения с переменной status справа от запятой, даже в первой директиве MATCH, что излишне, однако создает единообразие в представленном фрагменте. Последняя директива MATCH осуществляет проверку на тот случай, если все предыдущие директивы MATCH не распознали синтаксис, и сообщает об ошибке. Проверка на случай пустого параметра не нужна, так как после имени параметра в заголовке макроса указан знак *. В противном случае нам пришлось бы

добавить еще одну директивы (на этот раз последнюю) MATCH, для проверки случая пустого параметра.

Использования директивы ERR требует некоторых пояснений. Эта команда обрабатывается на стадии ассемблирования (так что она может и не сработать, если будет находиться в блоке IF), но вызывает мгновенное прекращение процесса обработки, как только она встречается ассемблеру, даже если это происходит в какой-то промежуточной стадии процесса разрешения кода. Следовательно, цель этой директивы сообщить об неустранимой ошибке; в данном случае синтаксической. Директива ERR не имеет дополнительных правил, касающихся ее аргументов – она прерывает ассемблирование с сообщением об ошибке не обращая внимание на наличие дополнительного текста в строке. В выше представленном примере в строке вставлена дополнительная текстовая константа, чтобы прояснить о какой проблеме идет речь – fasm всегда выводит строку, вызвавшую ошибку, так что это сообщение будет представлено на консоли.

Перевод, 2015, сентябрь