Глава 12. Схема работы транслятора с языка Ассемблера

... выбирая транслятор Ассемблера, вы выбираете судьбу, изменить которую впоследствии, за здорово живешь, не удастся!

Крис Касперски ака мыщъх

Сейчас будет рассмотрено, как транслятор преобразует входной модуль на "чистом" языке Ассемблера (уже *без* макросредств, которые обработал Макропроцессор) в объектный модуль. Разумеется, будет изучена только общая схема этого достаточно сложного процесса. Наша цель – рассмотреть основные принципы работы транслятора с языка Ассемблера и ввести необходимую терминологию, а также снять тот покров таинственности с работы программы-переводчика, который может быть у некоторых читателей. Более подробно этот вопрос, который относится к большой теме "Формальные грамматики и методы компиляции", должен изучаться отдельно.

Итак, как известно, компилятор Ассемблера относится к классу системных программ, которые называются трансляторами[1] – переводчиками с одного алгоритмического языка на другой. Наш транслятор переводит модуль с языка Ассемблера на объектный язык. При трансляции выполняются следующие задачи.

- Анализ входного модуля на наличие синтаксических ошибок.
- Генерация объектного модуля.
- Выдача протокола работы компилятора – так называемого листинга, а также выдачу аварийных сообщений об ошибках. Выдачу листинга при желании, можно заблокировать.

Сначала будет разобрана первая задача – анализ программы на наличие ошибок. Ясно, что найти все ошибки можно, только *просмотрев* весь текст модуля, строка за строкой и символ за символом. Каждый просмотр текста программного модуля компилятором называется *проходом*. Многие компиляторы с Ассемблера просматривает программу дважды, т.е. совершают два прохода. Такие компиляторы называются двухпроходными. Двухпроходная схема трансляции наиболее простая для реализации, но можно, усложнив алгоритм, производить трансляцию и за один проход (например, так работает транслятор с языка Free Pascal). В основном это можно сделать потому, что в Паскале практически всегда имя пользователя описывается или объявляется *перед* его первым использованием в программе.

На первом проходе компилятор с Ассемблера, анализируя текст программы, находит многие (но не все) ошибки, и строит некоторые важные таблицы, данные из которых используются далее как на первом, так и на втором проходе. Вкратце алгоритм первого прохода состоит в следующем. Так как основной синтаксической единицей нашего языка является предложение Ассемблера, то рассмотрим, как происходит обработка предложений на первом проходе.

В программах на Ассемблере не всегда возможно описать имя до его использования, например, метка в команде `jmp L` может встречается ниже по тексту программы. Ситуация, когда в предложении Ассемблера используется имя, которое будет описано дальше, называется *ссылкой вперёд* (Forward Reference). В связи со ссылками вперёд возникают проблемы, например, на место метки `L` в команду `jmp L` надо подставить неизвестное на данный момент расстояние от текущего значения счетчика размещения (точнее, регистра `EIP`) до метки `L`. Для решения проблем ссылок вперёд как раз и могут использоваться два прохода компилятора по тексту программы.

Сначала каждое предложение Ассемблера обрабатывается специальной программой транслятора, которая называется *лексическим анализатором*. Она разбивает каждое предложение на *лексемы* – логически неделимые единицы языка. О лексемах Вы уже должны немного знать из изучения языка Паскаль. Как и в Паскале, в языке Ассемблера существуют следующие *классы* лексем.

---

[1] Термин транслятор обозначает программу-переводчика с одного (формального) языка на другой. Трансляторы делятся на компиляторы, которые переводят программу целиком, и интерпретаторы, которые выполняют непосредственное исполнение каждой строки программы (без её перевода на язык машины). Хорошим аналогом является письменный (всего текста) и устный (синхронный, по одному предложению) перевод с одного языка на другой. В английском языке для этих действий существуют два разных глагола: `translate` и `interpret`.

- Имена. Это ограниченные последовательности букв и цифр, начинающиеся с *буквы*. Как Вы помните, в языке Free Pascal к буквам относился также и символ подчёркивания, а в Ассемблере к буквам относятся и такие символы, как вопросительный знак, точка (правда, только в первой позиции и у служебных имен), символы `@` и `$`. В нашем Ассемблере длина имени ограничена 247 символами. Все имена Ассемблера делятся на *служебные* (ключевые слова) и имена пользователя. В языке Паскаль есть ещё и *стандартные* имен (напомним, что стандартное имя имело заранее определённый смысл, но эти имена программисту можно было *переопределить*). Как уже говорилось, все служебные имена в нашем Ассемблере программист может директивой **`option NOKEYWORD`** исключить из числа служебных и, при необходимости, использовать как имена пользователя. В зависимости от настроек Ассемблера, большие и маленькие буквы в *именах пользователя* могут различаться или не различаться, но в *служебных именах* они не различаются.
- Числа. Язык Ассемблера, как и язык Free Pascal, допускает запись целых чисел в различных системах счисления (десятичной, двоичной, шестнадцатеричной и некоторых других). Не надо забывать и о вещественных числах.
- Строки символов. Строки символов в Ассемблере ограничены либо апострофами, либо двойными кавычками. Исключениями являются параметр директивы эквивалентности, например, `X` **`equ`** `ABC++`. Напомним, что выполнения такой директивы **`equ`** требует замены во всех следующих предложениях имени `X` на строку символов `ABC++`. После такой замены обычно требуется повторное разбиение предложений на лексемы. Кроме того, надо ещё раз напомнить, что в этом упрощенном изложении алгоритма работы компилятора считается, что Макропроцессор, который рассматривал фактические параметры макрокоманд тоже как строки символов, ограниченных <>, запятыми или точкой с запятой, уже закончил свою работу.
- Разделители. Как и в Паскале, лексемы перечисленных выше классов не могут располагаться в тексте программы подряд, легко понять, что между ними обязательно должна находиться хотя бы одна лексема-разделитель. К разделителям относятся знаки арифметических операций, почти все знаки препинания, пробел и некоторые другие символы.
- Комментарии. Эти лексемы не влияют на выполнение алгоритма, заложенного в программу, они переносятся в листинг, а из анализируемого текста удаляются. Не являются исключением макрокомментарии, которые начинаются с двух символов `';;'`, такие комментарии не переносятся в макрорасширения и попадают в листинг программы в одном экземпляре (т.е. только внутри текста макроопределений).

В качестве примера ниже показано предложение Ассемблера,

```
Metka: mov eax,Mas+67[ebx]; Комментарий
```

Выделим каждую лексему в прямоугольник, обратите внимание на лексему-пробел между кодом операции **`mov`** и первым операндом `AX`:

`Metka` `:` **`mov`** ␣ `eax` `,` `Mas` `+` `67` `[` `ebx` `]` `; Комментарий`

Во время лексического анализа выявляются *лексические ошибки*, когда некоторая группа символов не может быть отнесена ни к одному из классов лексем. Примеры лексических ошибок:[1]

```
134X        'A'38       B"C
```

Все опознанные (правильные) лексемы записываются на первом проходе в специальную *таблицу лексем*. В этой таблице вместе с каждой лексемой указывается её класс и некоторые другие атрибуты. Это делается в основном для того, чтобы на втором проходе транслятор мог уже двигаться по программе *по лексемам*, а не по отдельным составляющим их символам, что позволяет существенно ускорить второй просмотр текста модуля. Для однопроходного транслятора таблица лексем не строится.

---

[1] Здесь надо сказать, что компилятор с Ассемблера не предполагает глубокого знания программистом теории компиляции, поэтому в листинге программы такие ошибки называются общим термином "синтаксические ошибки", хотя, как Вы узнаете немного позже, "настоящие" синтаксические ошибки определяются другой частью компилятора – синтаксическим анализатором.

После разбиения предложения Ассемблера на лексемы начинается второй этап обработки предложения – этап *синтаксического анализа* (на жаргоне программистов *парсинга*). Этот этап выполняет программа транслятора, которая называется *синтаксическим анализатором* (парсером). Алгоритмы синтаксического анализа могут быть весьма сложны, и здесь они не будут подробно рассматриваться, это, как уже упоминалось, тема отдельного курса по компиляторам. Обычно в результате синтаксического анализа строится так называемое дерево синтаксического разбора. Если говорить совсем коротко, то синтаксический анализатор пытается из лексем построить более сложные конструкции предложения Ассемблера.

Напомним, что в Ассемблере есть несколько видов предложений: комментарий, команда и макрокоманда, резервирование памяти и директива. Транслятор обрабатывает предложение в соответствии с его типом. Если не удаётся отнести анализируемое предложение текста ни к одному из данных видов фиксируется ошибка. Схема обработки разных видов предложений следующая.

1) Комментарий – пропускается.

2) Команда. Напомним синтаксис: [метка[:]] мнемокод [операнды] [;комментарий]

   a) Если перед командой стоит метка, транслятор заносит информацию об имени метки в таблицу (имя, адрес размещения в секции кода).

   b) Строится битовое представление машинной команды: заполняется поле кода операции и поля операндов.[1]

3) Директива – компилятор выполняет директиву: запоминает информацию, заполняет поля в объектном коде нужными данными.

4) Резервирование памяти – счетчик размещения в данной секции увеличивается на число зарезервированных байт, имя (если есть) заносится\ в таблицу имён.

Во время работы транслятор использует таблицу мнемокодов и таблицу директив. В таблице мнемокодов для каждого мнемокода указаны соответствующие код машинной операции, возможные виды операндов и возможный размер команды в байтах. В таблице директив каждому названию директивы сопоставлена процедура ее обработки. Если при анализе предложения лексема не находится в таблице мнемокодов и в таблице директив, фиксируется ошибка (напомним, что все макросредства уже обработаны).

Особое внимание на этом этапе уделяется в программе именам пользователя, они заносятся синтаксическим анализатором в специальную *таблицу имён*, её ещё называют таблицей символических имён или просто таблицей символов. Вместе с каждым именем в эту таблицу заносятся и *атрибуты* (свойства) имени. Всего в Ассемблере у имени пользователя различают три основных атрибута, перечисленные ниже (сразу надо отметить, что не у каждого имени есть все из них). Для удобства дальнейшего изложения этим атрибутам присвоены имена `Offset`, `Type` и `Value`.[2]

- Атрибут `Offset`, он имеет формат `i32` и задаёт смещение *расположения* имени от начала той секции, в котором оно описано. Атрибут `Offset` могут иметь только имена, определяющие области памяти, метки команд и имена процедур. В листинге программы этот атрибут обозначается как `XXXX XXXX R`, где `XXXXXXXX` – это (16-ричное) значение счетчика размещения для этого имени в его секции. Буква `R` (**R**elative) обозначает, что смещение является *перемещаемой* величиной, т.е. линейно зависит он адреса начала в памяти соответствующей секции. Для внешних имён, описанных в директиве **`extrn`**, смещение неизвестно, что обозначается в листинге как `0000 0000 E`.
- Атрибут типа `Type`. С этим атрибутом имени Вы уже знакомы, для имен переменных он равен длине переменной в байтах, а для меток и имён процедур равен **`near`**=**`proc`**=`-252`. Все остальные имена имеют тип ноль (в некоторых учебниках по Ассемблеру это трактуется как *отсутствие* типа, при этом говорится, что у таких имен атрибута `Type` нет, однако операция Ассемблера **`type`** выдает для таких имен значение ноль).
- Атрибут значения `Value`. Этот атрибут определен только для имен числовых макроконстант и числовых макропеременных и равен значению этой константы или переменной.

---

[1] Для обработки макрокоманд, как уже говорилось, вызывается Макропроцессор, который возвращает Ассемблеру построенное макрорасширение для повторного лексического и синтаксического анализа.

[2] Для вывода таблицы имён в листинга надо задать в программе директиву **`.CREF`** (**C**ross **REF**erence).

Приведем примеры описаний имен, которые имеют *первые два* из перечисленных выше атрибутов:

```
A    db  13;  Не имеет атрибута Value !
B    equ A;   Не имеет атрибута Value !
C:   mov B,1; Не имеет атрибута Value !
D    equ C;   Не имеет атрибута Value !
```

А теперь примеры имен, у которых есть только атрибут `Value` и атрибут `Type=0`:

```
N    equ 100; Value=100
M    equ N+1; Value=101
     K=13;    Value=13
P    equ K;   Value=13
```

Для вычисления адресов имён в своих секциях, компилятор использует особую переменную, называемую счетчиком размещения, в языке Ассемблера эта переменная имеет служебное имя `$`. Когда транслятор встречает первую директиву начала секции (данных или кода), значение этой переменной инициализируется нулём; по мере заполнения секции значение счетчика размещения увеличивается на длину размещённой команды или длину размещённых переменных. Директивы выравнивания **align** {2, 4, 8, 16} увеличивают счетчик размещения, чтобы он стал кратным параметру (т.е. выровнен в памяти на границу соответствующего числа байт). Директива Ассемблера **org** op1 присваивает счетчику адреса значение своего операнда, например, для секции данных

```
.data
    org 100h
  X dd ?
```

переменная `X` будет размещаться в секции данных, начиная с адреса `100h`.[1]

Рассмотрим теперь пример маленького, синтаксически правильного, но, вообще говоря, бессмысленного (не головного) программного модуля на Ассемблере, для этого модуля будет построена таблица имен пользователя.

```
.data
  A dd 19; счетчик размещения $=0
  B db ?;                  $=4
.code;      счетчик размещения $=0
    extrn X:proc
    mov   eax,A
    mov   ecx,R
    jmp   L
R   equ   -14
    call  X
L:  ret
    end
```

На рис. 12.1 показана таблица имен пользователя, которая построится синтаксическим анализатором при просмотре показанной выше программы до предложения

```
mov eax,A
```

---

[1] Для (сильно) продвинутых читателей. Использование директивы **org** позволяет делать в Ассемблере и "странные" вещи, например:

```
add ecx,1; формат r32,i8; ecx:=ecx+Longint(i8=1) длина 3 байта
add ecx,1000; формат r32,i32 длина 6 байт
org $-4; возврат счетчика размещения назад на 4 байта
dd  1;   формат r32,i32 ecx:=ecx+00000001h длина 6 байт
```

Таким образом команда **add** ecx, 1000 "превратилась" в команду **add** ecx, 1, но длиной не 3, а 6 байт. Обычно компиляторы с языков высокого уровня таким образом увеличивают длину команды (в последнем примере длина команды **add** стала 6 байт), не увеличивая время её выполнения. Это делается для выравнивания следующей команды на границу 16 байт для более эффективной работы кэш памяти и конвейера. Альтернативой является заполнение пустого места в программе командами **nop** или "пустыми" префиксами, но на выполнение каждого такого префикса будет тратиться дополнительный такт процессора.

включительно. Атрибуты, которые не могут быть у данного имени, отмечены прочерком. Учтите, что в отличие от целых чисел, хранящихся в памяти, значение счетчика размещения `Offset` указывается в листинге программы в прямом, а не в перевёрнутом виде.

Имя	Offset	Type	Value
A	0000 0000 R	4	———
B	0000 0004 R	1	———
X	0000 0000 E	-252	———

Рис. 12.1. Вид таблицы имён после предложения **`mov eax,A`**.

Таким образом, если у имени в таблице есть хоть один атрибут, то это имя уже описано или объявлено). А теперь рассмотрим ссылку вперёд, пусть теперь синтаксический анализатор рассматривает предложение

```
mov ecx,R
```

В этой команде *используется* имя `R`, однако этого имени ещё нет в таблице имён. В этом случае оно заносится туда (ниже приведена строка таблицы для этого имени) Этого имени не было в таблице потому, что оно ещё не описано и про него ничего неизвестно, поэтому и все поля атрибутов в таблице имеют пока неопределенные значения:

R	?	?	?

Соответственно, невозможно определить и точный код операции команды **`mov ecx,R`**, так как второй операнд этой команды, вообще говоря, может иметь любой из допустимых для этой команды форматов `r32,m32` или `i32`. Это уже упоминавшаяся проблема ссылки вперёд, когда некоторое имя используется в программе до того, как оно описано или объявлено.

На этом примере ярко видна цель *второго просмотра* текста программы: на втором просмотре, после получения информации об атрибутах имени `R`, возможно определить правильный формат этой команды (а значит, в частности, определить и *длину* этой команды в байтах). Некоторые языки (например, Паскаль) практически всюду запрещают ссылки вперёд, для таких языков естественна трансляция за один проход текста программы, однако в Ассемблере запретить ссылки вперёд невозможно.

После полного просмотра программы синтаксический анализатор построит таблицу имен, показанную на рис. 12.2.[1]

Имя	Offset	Type	Value
A	0000 0000 R	4	——
B	0000 0004 R	1	——
X	0000 0000 E	-252	——
R	——	0	-14
L	0000 00C3 R	-252	——

Рис. 12.2. Вид таблицы имён после анализа всего модуля.

На этапе синтаксического анализа выявляются все синтаксические ошибки в программе, например:

```
mov eax,bl; Несоответствие типов операндов
mov ax,A;   Несоответствие типов операндов
add A,A+2;  Нет такого формата команды
sub ecx;    Мало операндов
jel L;      Неизвестный код операции
```

и т.д. Используя таблицу имен, синтаксический анализатор легко обнаруживает ошибки следующего вида:

1. Неописанное имя. Имя не является внешним (**`extrn`**), но встречается только в позиции использования (в поле операндов) и ни разу не встречается в поле метки.
2. Дважды описанное имя. Одно и то же имя дважды (или большее число раз) встретилось в поле метки.[1]

---

[1] Для простоты изложения все имена помещены в одну таблицу. В то же время имена, являющиеся входными точками (**`public`**), а также внешние имена модуля (**`extrn`**), либо должны иметь соответствующие дополнительные атрибуты, либо помещаться в другую таблицу *внешних и входных* имен. Информация о таких именах, как уже говорилось, помещается компилятором в паспорт объектного модуля.

3. Описанное, но не используемое имя. Это *предупредительное* сообщение может выдаваться некоторыми "продвинутыми" Ассемблерами, если имя определено в поле метки, но ни разу не встретилось в поле операндов и отсутствует в параметрах директив **public** (видимо, программист что-то напутал).

Кроме того, синтаксический анализатор может обнаружить и ошибки, относящиеся к модулю в целом, например, превышение максимальной длины некоторой секции, использование метки команды (т.е. имени с двоеточием) в предложении резервирования памяти и т.д.

Итак, на втором проходе синтаксический анализатор сначала обнаруживает все ошибки в программном модуле и однозначно определяет формат каждой команды.[2] Затем он приступает к задаче генерации объектного модуля. Теперь все числа преобразуются во внутреннее машинное представление, выписывается битовый вид всех команд, оформляется паспорт объектного модуля. Далее полученный объектный модуль записывается в библиотеку объектных модулей (обычно куда-нибудь в дисковую память).

Как и компилятор с языка Free Pascal, компилятор Ассемблера MASM 6.14 является *однопроходным*, он просматривает исходный текст только один раз. При этом при генерации объектного модуля в полный рост встаёт проблема ссылок вперёд, когда надо точно знать длину и битовое представление каждой команды. Эта проблема решается путём повторного (и, возможно, многократного) просмотра, анализа и модификации полученной таблицы имён пользователя, в которую добавляются необходимые служебные поля). Теперь, когда уже есть полная информация об атрибутах каждого имени, в командах можно скорректировать тип и размер операндов.

Например, для команды `jmp L`, уже зная расположение метки `L`, можно выбрать для параметра перехода короткий формат `i8`, а не длинный формат `i32`. Для команды `mov ecx, R` уже однозначно определён формат `r32,i32` и т.д. Здесь надо особо подчеркнуть, что это не второй проход по тексту программного модуля, а только проходы по таблице имён. Именно это и позволяет считать компилятор однопроходным, он один раз просматривает исходный текст модуля

Напоминаем, что в объектном модуле поля `0000 0000 E` остаются незаполненными. Местонахождение всех таких полей, а также полей с относительными адресами `XXXX XXXX R` перечисляется в паспорте объектного модуля. В дальнейшем эти поля будут обрабатываться редактором внешних связей, а потом, для относительных адресов, возможно, и загрузчиком.

На этом завершим наше краткое знакомство со схемой трансляции исходного модуля с языка Ассемблера на объектный язык.

Вопросы и упражнения

1. Что означает, что компилятор с Ассемблера двухпроходный ?
2. Какую работу выполняет лексический анализатор транслятора ? Приведите примеры лексических ошибок в программе.
3. Что такое таблица лексем, когда она получается и для чего нужна ?
4. Как определить, когда в программе некоторое имя используется, а когда – описывается (объявляется) ?
5. Какие атрибуты есть у имён пользователя ?
6. Что такое таблица имён пользователя ?
7. Что такое внешнее имя, какие у него есть атрибуты ?
8. Что такое перемещаемое имя, какие у него есть атрибуты ?

---

[1] Исключением из этого правила являются имена процедур, которые должны встретиться в поле метки соответствующих директив начала и конца описания сегмента, имена локальных меток в процедурах, а также имена макроопределений (макроопределение, описанное позже, *переопределяет* одноимённое макроопределение, описанное ранее).

[2] Замечание для продвинутых читателей. Вспомните, что при написании макроопределений для контроля типов переданных параметров использовались средства условной генерации, и выдавалась необходимая диагностика об ошибках использования макрокоманды. Этим, по существу, дополнялись средства синтаксического анализатора Ассемблера. Таким образом, обычный пользователь-программист, используя макросредства, по существу способен расширить возможности служебной программы – компилятора.

9. В чём заключается проблема ссылок вперёд ?
10. Перечислите случаи, когда в Ассемблере MASM использование имени пользователя в позиции метки более одного раза не является ошибкой.
11. Приведите пример, когда конкретный формат команды сложения и её длина может быть определен на втором проходе компилятора.
12. Как решается проблема ссылок вперёд для однопроходного компилятора ?