Трансляторы языков программирования

Преобразование программы, написанной на одном языке программирования, в эквивалентную программу на другом языке, называется *трансляцией*. Язык, на котором представлена входная программа, называется *исходным языком*, а сама программа – *исходной программой* (*исходным кодом*). Выходной язык называется *целевым языком*, а программа – *целевой программой* (*целевым кодом*). Программа, выполняющая трансляцию, называется *транслятором*.

Транслятор, преобразующий исходную программу на язык низкого уровня (в машинный код или код, близкий к машинному, подобному ассемблеру, т. е. в язык процессора), называется *компилятором*. Полученную в результате компиляции программу часто называют *объектной программой* (*объектным кодом*).

Основные достоинства компилятора:

а) возможность эффективной оптимизации кода, сокращающей количество операций;

б) высокая скорость выполнения сгенерированного целевого кода;

в) отсутствует необходимость наличия компилятора на компьютере пользователя.

Недостатки компилятора:

а) генерируемый компилятором целевой код зависит от используемой операционной системы и ориентирован на работу с определённым типом процессора;

б) при внесении изменений требуется полная перекомпиляция исходного кода.

Другим типом транслятора является *интерпретатор*, который вместо получения целевой программы непосредственно выполняет операторы исходной программы (*чистая интерпретация*).

Достоинства интерпретатора:

а) Независимость от операционной системы (переносимость кода), поскольку использование виртуальной машины позволяет интерпретаторам эффективно работать на всех платформах.

б) при внесении изменений не требуется полная перекомпиляция исходного кода.

Недостатки интерпретатора:

а) интерпретируемая программа выполняется медленнее, поскольку промежуточный анализ исходного кода и планирование его выполнения требуют дополнительного времени в сравнении с непосредственным исполнением машинного кода, в который мог бы быть скомпилирован исходный код;

б) исходный код не может работать отдельно без наличия интерпретатора на компьютере пользователя.

Существуют также смешанные трансляторы, которые объединяют в себе как компиляцию, так и интерпретацию. Примером является языковый процессор Java, в котором исходная программа на Java сначала компилируется в промежуточный код, называемый байт-кодом. Затем байт-код интерпретируется виртуальной машиной.

Большие программы часто компилируются по частям, поэтому перемещаемый машинный код должен быть скомпонован совместно с другими перемещаемыми объектными и библиотечными файлами в единый код. Для этого предназначен *компоновщик* (*редактор связей*, *линкер*) – это инструментальная программа, которая из нескольких объектных модулей собирает исполняемый код, вычисляя соответствующие адреса для обеспечения доступа по соответствующим ссылкам из одного модуля к другому.

Полученный после компоновки машинный код загружается в память и запускается для выполнения (при этом относительные адреса заменяются на абсолютные). Для этого служит *загрузчик*, который обычно является частью операционной системы, но может быть и отдельной программой.

Фазы компиляции

Концептуально компиляция исходной программы в объектную программу выполняется в несколько этапов, которые называются *фазами компиляции*. В общем случае в процессе каждой фазы происходит преобразование исходной программы из одного представления в другое. Однако на практике некоторые фазы могут быть сгруппированы вместе и промежуточные представления программы внутри таких групп могут явно не строиться.

Выделяют следующие типичные фазы:

1. *Лексический анализ*. Реализуется частью компилятора, которая называется *лексическим анализатором* (или *сканер*, *лексер*). Сканер выполняет предварительную обработку текста исходной программы, группируя символы входного потока в лексические единицы (*лексемы*). Для каждой лексемы сканер формирует выходной *токен* вида <*код_токена*, *атрибут*> для последующих фаз компиляции. *Код_токена* идентифицирует класс лексемы (*лексический класс*). Для удобства записи *код_токена* обычно представляется абстрактным именем (или специальным обозначением). *Атрибут* токена обеспечивает доступ к дополнительной информации о лексеме, если лексическому классу соответствует множество лексем, например, для идентификатора атрибут токена указывает на запись в таблице символов.

Рассмотрим фрагмент исходной программы:

for *i* := 1 **to** 20 **do** *mas*[*i*] := 0**;**

Пусть в процессе формирования таблицы символов информация об идентификаторах *i* и *mas* оказалась в записях с номерами 3 и 7 соответственно, а о константах 1, 20 и 0 – в записях с номерами 2, 10 и 11 соответственно. Тогда сканер сформирует следующую последовательность токенов (символ Λ означает пустое значение атрибута):

<**for**, Λ>, <**id**, 3>, <**ass**, Λ>, <**num**, 2>, <**to**, Λ>, <**num**, 10>, <**do**, Λ>,

<**id**, 7>, <**[**, Λ>, <**id**, 3>, <**]**, Λ>, <**ass**, Λ>, <**num**, 11>, <**;**, Λ>.

Здесь имена токенов **for**, **to**, **do** обозначают соответствующие ключевые слова, **ass** – операцию присваивания, **id** – идентификатор, **num** – числовую константу, остальные токены обозначены соответствующими абстрактными символами **[,]**, **;**.

2. *Синтаксический анализ*. Реализуется частью компилятора, которая называется *синтаксическим анализатором* (или *парсером*). Парсер исследует последовательность токенов с целью проверки, соответствует ли она синтаксису языка, создавая промежуточное представление (в общем случае древовидное), описывающее грамматическую структуру потока токенов.

3. *Семантический анализ*. Реализуется частью компилятора, которая называется *семантическим анализатором*. В этой фазе выполняется проверка программы на соответствие семантическим соглашениям, сбор информации о типах и ее сохранение в таблице символов, проверку на соответствие типов, преобразование (приведение) типов (если это предусмотрено в языке).

4. *Генерация промежуточного кода*. Реализуется частью компилятора, которая называется *генератором промежуточного кода*. Данная фаза предназначена для преобразования исходной программы в промежуточное представление, которое можно рассматривать как программу для некоторой виртуальной машины. Промежуточный код должен легко генерироваться и легко транслироваться в целевой код. Наиболее распространенными формами представления промежуточной программы являются динамические структуры, представляющие ориентированный граф (в частности, синтаксическое дерево), трехадресный код, префиксная и постфиксная запись, байт-код Java, LLVM.

LLVM (ранее Low Level Virtual Machine) – проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. Состоит из набора компиляторов из языков высокого уровня (так называемых «фронтендов»), системы оптимизации, интерпретации и компиляции в машинный код. В основе инфраструктуры используется RISC-подобная платформонезависимая система кодирования машинных инструкций (байткод LLVM IR), которая представляет собой высокоуровневый ассемблер, с которым работают различные преобразования.

Написан на C++, обеспечивает оптимизации на этапах компиляции, компоновки и исполнения. Изначально в проекте были реализованы компиляторы для языков Си и C++ при помощи фронтенда Clang, позже появились фронтенды для множества языков, в том числе: ActionScript, Ада, C#, Common Lisp, Crystal, CUDA, D, Delphi, Dylan, Fortran, Graphical G Programming Language, Halide, Haskell, Java (байткод), JavaScript, Julia, Kotlin, Lua, Objective-C, OpenGL Shading Language, Ruby, Rust, Scala, Swift, Xojo, Zig.

LLVM может создавать машинный код для множества архитектур, в том числе ARM, x86, x86-64, PowerPC, MIPS, SPARC, RISC-V и других (включая GPU от Nvidia и AMD).

Некоторые проекты имеют собственные LLVM-компиляторы (например LLVM-версия GCC), другие используют инфраструктуру LLVM, например таков Glasgow Haskell Compiler.

LLVM родился как исследовательский проект Криса Латнера (тогда ещё студента-магистра в Университете штата Иллинойс в Урбана-Шампейн) и Викрама Адве (тогда и по сию пору профессора в том же университете). Целью проекта было создание промежуточного представления (intermediate representation, IR) программ, позволяющего проводить «агрессивную оптимизацию в течение всего времени жизни приложения» – что-то вроде Java байт-кода, только круче. Основная идея – сделать представление, одинаково хорошо подходящее как для статической компиляции (когда компилятор получает на вход программу, написанную на языке высокого уровня, например C++, переводит её в LLVM IR, оптимизирует, и получает на выходе быстрый машинный код), так и динамической (когда runtime система получает на вход машинный код вместе с LLVM IR, сохранённым в объектном файле во время статической компиляции, оптимизирует его – с учётом собранного к этому времени динамического профиля – и получает на выходе ещё более быстрый машинный код, для которого можно продолжать собирать профиль, оптимизировать, и так до бесконечности).

В настоящее время под словом «LLVM» понимается, что LLVM это не о виртуальных машинах, и вовсе даже не акроним, а просто название проекта.

Для присваивания a := *b* < *c* **and not** (*d* > *e* **or** *f* < *g*) может быть сгенерирован следующий трехадресный код:

```
    if b < c goto L3
    goto L2
L3: if d > e goto L2
    goto L4
L4: if f < g goto L2
    goto L1
L1: a:=true
    goto Snext
L2: a:=false.
```

5. *Машинно-независимая оптимизация кода*. Реализуется частью компилятора, которая называется *машинно-независимым оптимизатором кода*. Эта фаза предназначена для улучшения сгенерированного промежуточного кода, чтобы получить более качественный (уменьшить число операций, использовать меньшее количество ресурсов и т. п) целевой код.

Приведенный выше сгенерированный трехадресный код не оптимален. Можно без всяких последствий удалить команды безусловного перехода **goto** *L*4 и **goto** *L*1, поскольку они реализуют переход на непосредственно следующие за ними команды. Можно убрать также команду **goto** *L*2, если в первой команде заменить **if** на **ifFalse**, т. е. поменяв условие на обратное. Тогда улучшенный трехадресный код будет иметь следующий вид:

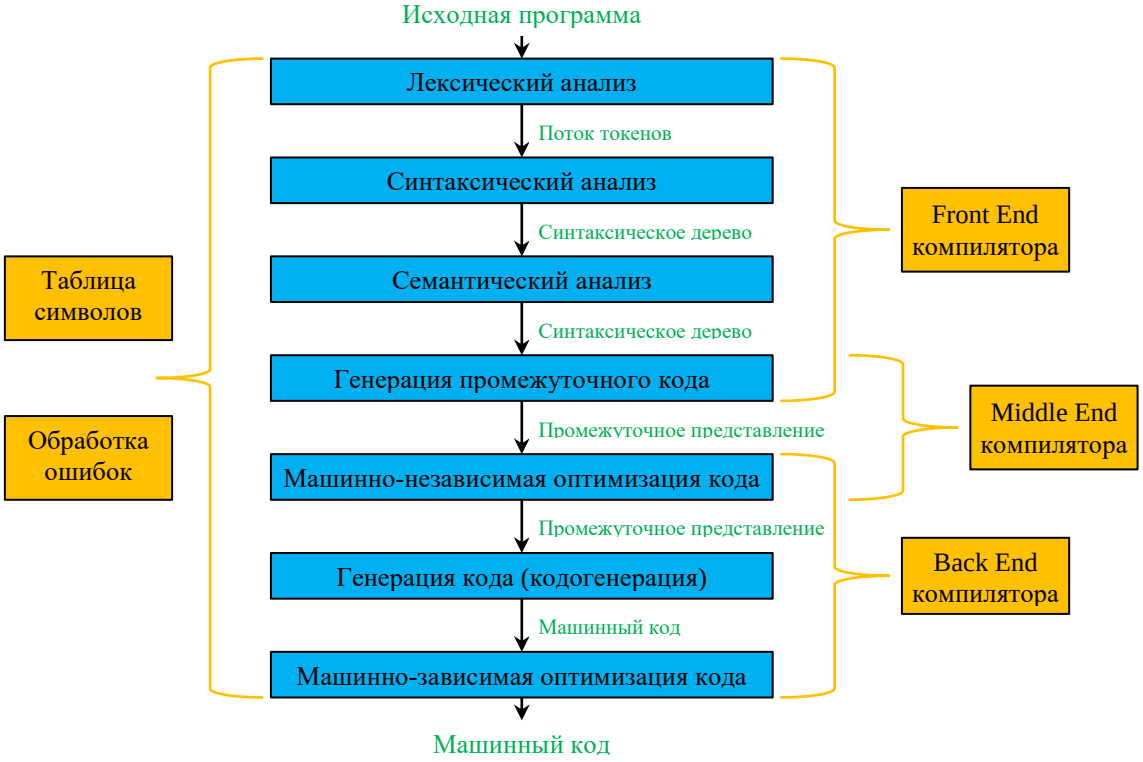
```
      ifFalse b < c goto L2
      if d > e goto L2
      if f < g goto L2
L1:  a:=true
      goto Snext
L2:  a:=false.
```

6. *Генерация кода* (*кодогенерация*). Реализуется частью компилятора, которая называется *генератором кода*. Выполняет преобразование промежуточного кода в целевой код. Если, например, целевой язык представляет собой машинный код, для каждой переменной выбираются соответствующие регистры и ячейки памяти, затем команды промежуточного языка транслируются в последовательность машинных команд.

7. *Машинно-зависимая оптимизация кода*. Реализуется частью компилятора, которая называется *машинно-зависимым оптимизатором кода*. В этой фазе производится попытка улучшения сгенерированного целевого кода с учетом специфики команд целевого языка.

Практически все фазы взаимодействуют с *таблицей символов* (на практике это может быть несколько таблиц, например, таблица идентификаторов, таблица констант и т. п.), в которой содержатся все необходимые данные об объектах (тип идентификатора или константы, число измерений массива, адрес памяти, число входных и выходных параметров процедуры и т. д.).

Типичное представление фаз компиляции приведено на рисунке.



Фазы компиляции

Укрупненно в компиляторе можно выделить две части: анализ и синтез. *Анализ* включает в себя первые три фазы. *Синтез* включает остальные фазы.

Можно выделить также машинно-независимую и машинно-зависимую части. *Машинно-независимая* часть объединяет те фазы компилятора (или часть фаз), которые зависят в первую очередь от исходного языка и практически не зависят от целевой машины. Обычно сюда входят первые пять фаз. *Машинно-зависимая* часть состоит из тех фаз компиляции, которые в первую очередь зависят от целевой машины, для которой выполняется компиляция, и, вообще говоря, не зависит от исходного языка, а только от промежуточного.

При проектировании компилятора важную роль играет определение числа проходов, которое необходимо выполнить для получения целевого кода. *Проход* – это последовательное чтение исходного текста программы или какого-либо его промежуточного представления. Имеется множество способов группировки фаз компиляции по проходам, их выполнение чередуется во время прохода.

Самыми простыми являются *однопроходные компиляторы*. Однако для большинства современных языков программирования создаются *многопроходные компиляторы*. В таких компиляторах отдельные фазы могут быть реализованы более простыми способами. Но приходится проектировать промежуточные языки для их обработки в последующих проходах. Часто оптимизация кода (как машинно-независимая, так и машинно-зависимая) выполняется отдельным проходом. Такие компиляторы структурно более сложные, чем однопроходные.

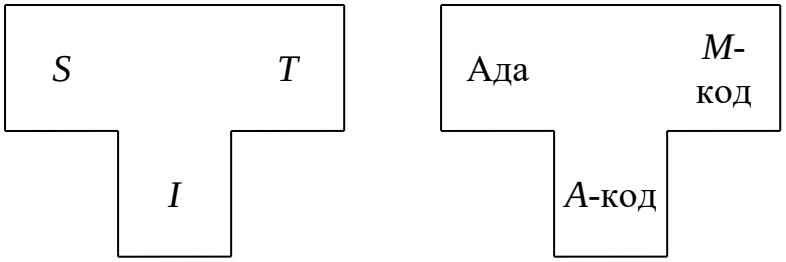
Для достаточно большого числа языков можно сгруппировать в отдельный проход лексический, синтаксический и семантический анализ, а также генерацию промежуточного кода.

Т-диаграммы

Удобным способом графического представления компиляторов являются *Т-диаграммы*, которые предложил Х. Брэтман (H. Bratman) в 1961 году. *Т*-диаграмма – это графическое изображение тройки языков, связанных с компилятором:

- входного (*компилируемого*) языка *S*;
- языка, на котором написан компилятор (*язык реализации*) *I*;
- выходного (*целевого*) языка *T*.

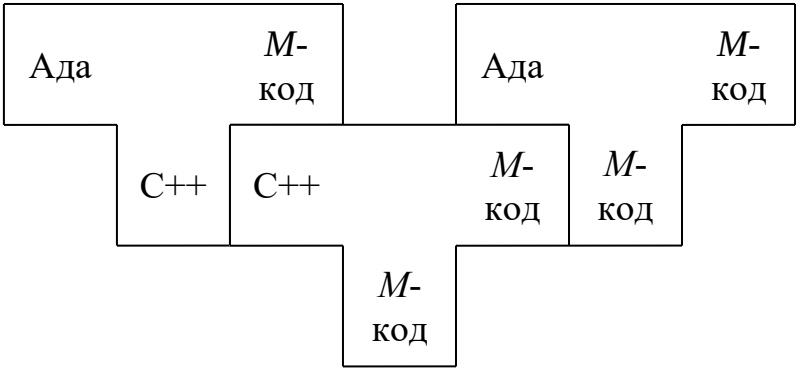
В тексте такой компилятор можно обозначить как S_IT, а соответствующая *Т*-диаграмма показана на рисунке (*а*). На рисунке (*б*) в качестве примера показан компилятор Ада$_{A\text{-код}}$*M*-код, написанный на ассемблере (*А*-код), для компиляции программ, написанных на языке Ада, в машинный код некоторого компьютера (*М*-код).



а *б*

Т-диаграммы можно соединять вместе в соответствии с простыми правилами:

- ветви среднего звена должны показывать те же языки, что и смежные с ним соседние звенья;
- в двух верхних звеньях в правых и левых углах должны быть записаны одинаковые языки.

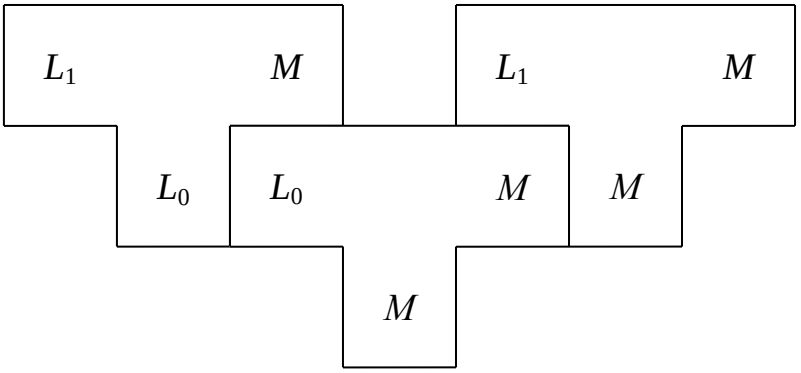


На рисунке показан компилятор с языка Ада в машинный язык *M*-код, построенный с помощью двух компиляторов. Если имеются компиляторы Ада $_{C++}$ *M*-код и C++ $_{M\text{-код}}$ *M*-код, то для получения компилятора с языка Ада с языком реализации *M*-код (Ада $_{M\text{-код}}$ *M*-код) достаточно пропустить компилятор Ада $_{C++}$ *M*-код через компилятор C++ $_{M\text{-код}}$ *M*-код. В результате получим Ада $_{M\text{-код}}$ *M*-код. Это можно записать как Ада $_{M\text{-код}}$ *M*-код = Ада $_{C++}$ *M*-код + C++ $_{M\text{-код}}$ *M*-код.

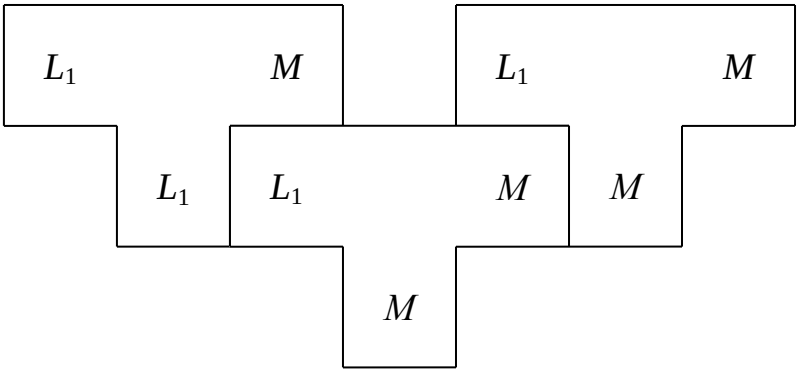
Интересна так называемая *технология раскрутки*. В основе технологии раскрутки лежит методологический прием, сущность которого состоит в использовании компилируемого языка для написания самого компилятора и последующей его трансляции в машинный код. Раскрутка – итерационная технология, состоящая в общем случае из нескольких стадий.

Пусть требуется получить компилятор для входного языка высокого уровня L в некоторый низкоуровневый целевой язык M. Предположим, что в качестве языка реализации хотим использовать сам язык L, который является более дружественным, чем M, т. е. требуемый компилятор – $L_L M$.

Сначала должен быть разработан небольшой компилятор для трансляции некоторого небольшого подмножества L_0 языка L в целевой код M, т. е. компилятор $L_{0\,M} M$. На первой стадии это подмножество L_0 используется для написания компилятора $L_{1\,L0} M$ для расширенного языка L_1 ($L_0 \subset L_1 \subset L$). После пропускания $L_{1\,L0} M$ через $L_{0\,M} M$ получим компилятор $L_{1\,M} M$ для более мощного (по сравнению с L_0) языка L_1.



Располагая этим компилятором, можно улучшить компилятор $L_{1\,L0}\,M$, переписав его на L_1 и получив версию $L_{1\,L1}\,M$. Пропустив $L_{1\,L1}\,M$ через $L_{1\,M}\,M$, получим новую улучшенную версию компилятора $L_{1\,M}\,M$. На этом завершается первая стадия.



На каждой новой стадии раскрутки язык L_{i-1} расширяется до L_i и вся процедура повторяется заново. Так продолжается до тех пор, пока на n-й стадии очередное расширение языка L_n не совпадет с языком L, т. е. пока не станет $L_n = L$. В результате ее выполнения будет получен требуемый компилятор $L_L M$.