3. Основы компиляторов

- Основные понятия. Компиляторы и интерпретаторы.
- Входной язык, целевой язык, язык реализации. Т-диаграммы.
- Прямой компилятор. Раскрутка. Кросс-трансляторы. Виртуальные машины. Компиляция "на лету".

Лекция 3. Основы компиляторов

В этой лекции рассматриваются следующие вопросы:

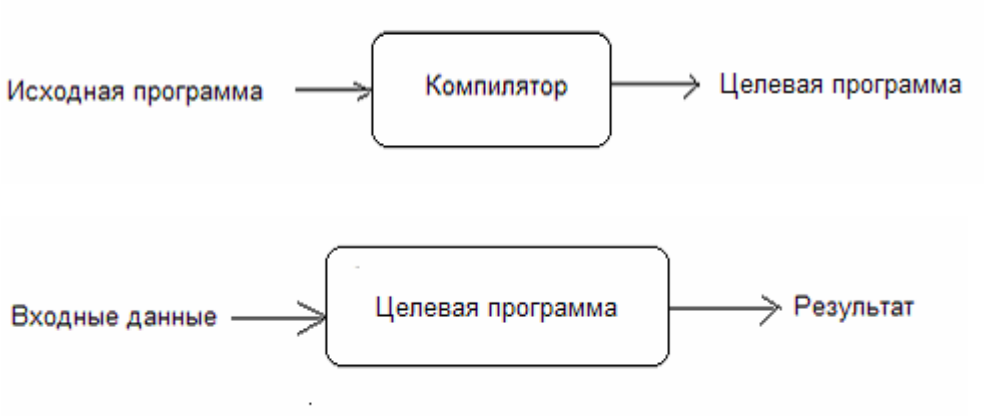
- Компиляторы и интерпретаторы
- Основные понятия компиляции
- Понятия входного языка, целевого языка и языка реализации
- Т-диаграммы
- Прямая компиляция, кросс-компиляция, раскрутка
- Виртуальные машины
- Компиляция "на лету"

Основные задачи компиляторов

- Необходимость в компиляторах появилась одновременно с появлением первых языков высокого уровня (Фортран, Кобол)
- Процесс компиляции обычно состоит из анализа исходной программы и синтеза объектной программы

Основные задачи компиляторов

Задача транслятора (*translator*) сделать программу, написанную на некотором языке программирования, понятной компьютеру. Этого можно добиться одним из двух способов: компиляцией (*compilation*) или интерпретацией (*interpretation*). Программу, являющуюся входными данными транслятора, будем называть *исходной программой* (*source program*). Обычно язык, на котором написана исходная программа, - это *язык высокого уровня* (*high-level language*).
Компилятор (*compiler*) переводит исходную программу в эквивалентную программу на языке, понятном компьютеру, то есть на машинном языке. Мы будем называть программу, получающуюся в результате работы компилятора, *целевой программой* (*target program*). Процесс компиляции и последующего выполнения программы можно изобразить следующим образом:



Сам компилятор также является программой, причем программой на машинном языке. Хотя, как мы увидим в дальнейшем, первоначально компилятор может быть написан на каком-нибудь языке более высокого уровня, чем машинный язык, а затем откомпилирован в машинный язык. Язык, на котором написан компилятор, будем называть *языком реализации*.

Если мы обозначим язык, на котором написана исходная программа, то есть *исходный язык* (*source language*), L_1, язык целевой программы (*целевой язык – target language*) L_2, а язык реализации (*implementation language*) L_3, то мы можем представить компилятор, как отображение множества L_1 в множество L_2 , т.е. $K_{L_3} : L_1 \to L_2$.

Процесс компиляции состоит из двух частей: анализа (*analysis*) и синтеза (*synthesis*). Анализирующая часть компилятора разбивает исходную программу на составляющие ее элементы (конструкции языка – *language constructions*) и создает промежуточное представление исходной программы. Синтезирующая часть из промежуточного представления создает новую, целевую, программу.

Отметим, что однажды полученная в результате компиляции целевая программа может в дальнейшем выполняться много раз с различными входными данными.

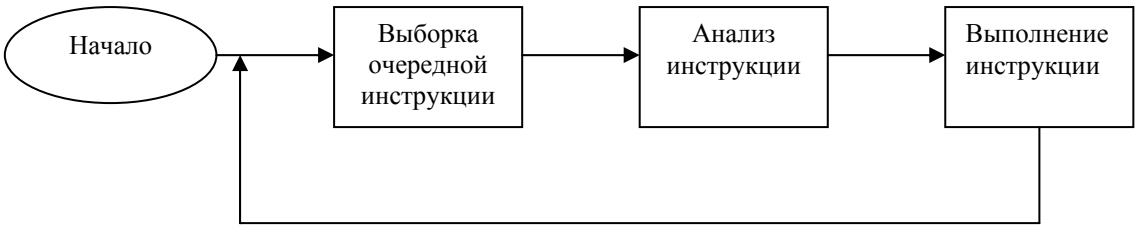
Далеко не всегда исходные программы корректны с точки зрения исходного языка. Более того, некорректные программы подаются на вход компилятору значительно чаще, чем корректные – таков уж современный процесс разработки программ. Поэтому крайне важной частью процесса компиляции является точная диагностика ошибок, допущенных во входной программе. Впрочем, это замечание верно не тоько для компиляторов, но и для интерпретаторов.

Интерпретатор

В отличие от компилятора интерпретатор не создает никакой новой программы. Входными данными интерпретатора является не только исходная программа, но и входные данные самой исходной программы.

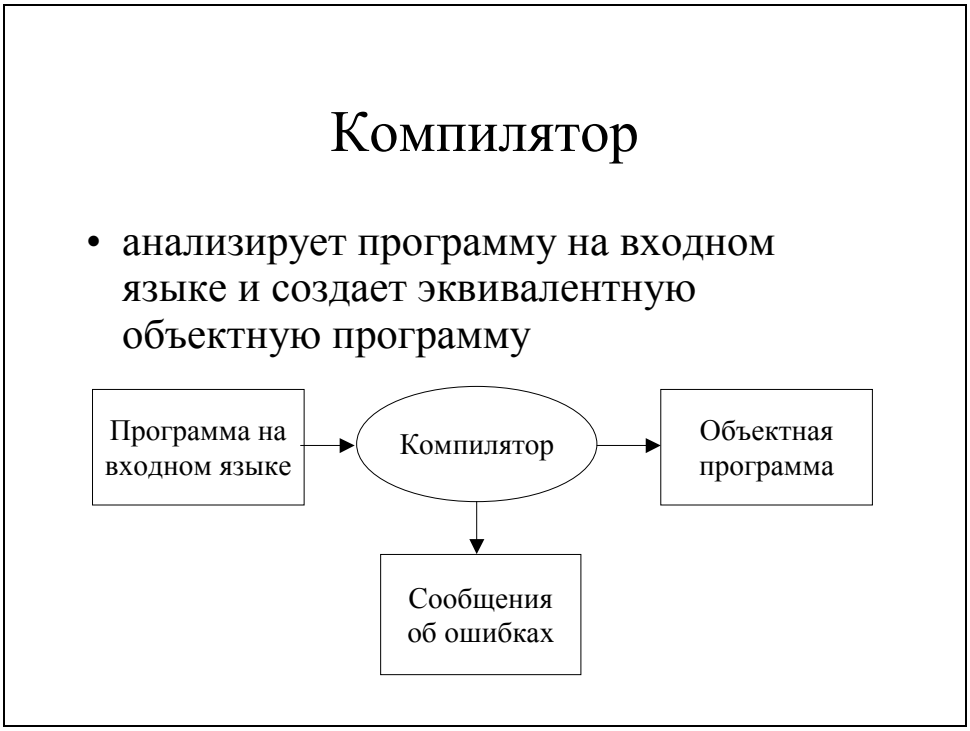
Интерпретатор, так же, как и компилятор, анализирует программу на входном языке, создает промежуточное представление, а затем выполняет операции, содержащиеся в тексте этой программы. Например, интерпретатор может построить дерево разбора, а затем выполнить операции, которыми помечены узлы этого дерева.

В том случае, если исходный язык достаточно прост (например, если это язык ассемблера или Basic), то никакое промежуточное представление не нужно, и тогда интерпретатор – это простой цикл. Он выбирает очередную инструкцию языка из входного потока, анализирует и выполняет ее. Затем выбирается следующая инструкция. Этот процесс продолжается до тех пор, пока не будут выполнены все инструкции, либо пока не встретится инструкция, означающая окончание процесса интерпретации.



Понятно, что при повторном запуске программы она должна интерпретироваться с самого начала.
Интерпретация приводит к более гибкой и лучшей диагностике ошибок, чем компиляция. Поскольку исходная программа исполняется непосредственно, интерпретатор может включать хороший *отладчик* (*debugger*). Кроме того, интерпретатор может легко справиться с языками, позволяюшими создавать программы, некоторые характеристики которых (например, размеры и типы переменных) могут зависеть от входных данных. Некоторые черты языков программирования таковы, что они не могут быть реализованы иначе, чем с использованием интерпретации.

Современная ситуация такова, что в трансляторах часто используются как элементы компиляции, так и интерпретации.

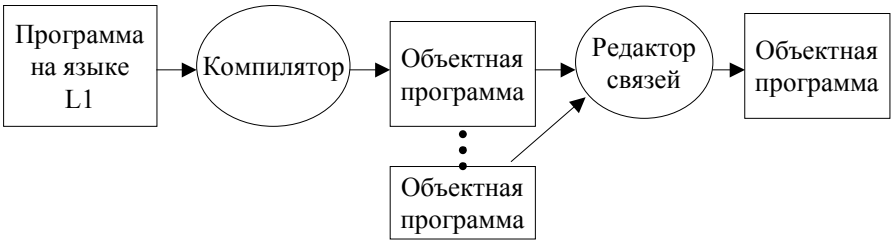


Компилятор

При обсуждении процесса компиляции необходимо рассмотретьСуществует огромное количество различных языков программирования, начиная с таких традиционных языков программирования как Fortran и Pascal и кончая современными объектно-ориентированными языками такими, как C# и Java. Практически каждый язык программирования имеет какие-то особенности с точки зрения создателя транслятора. Однако мы начнем с рассмотрения разнообразных целевых языков компиляторов.

Объектная программа

- Последовательность абсолютных команд машинных команд
- Последовательность перемещаемых машинных команд
- Программа на языке ассемблера
- Программа на некотором другом языке



Целевая программа

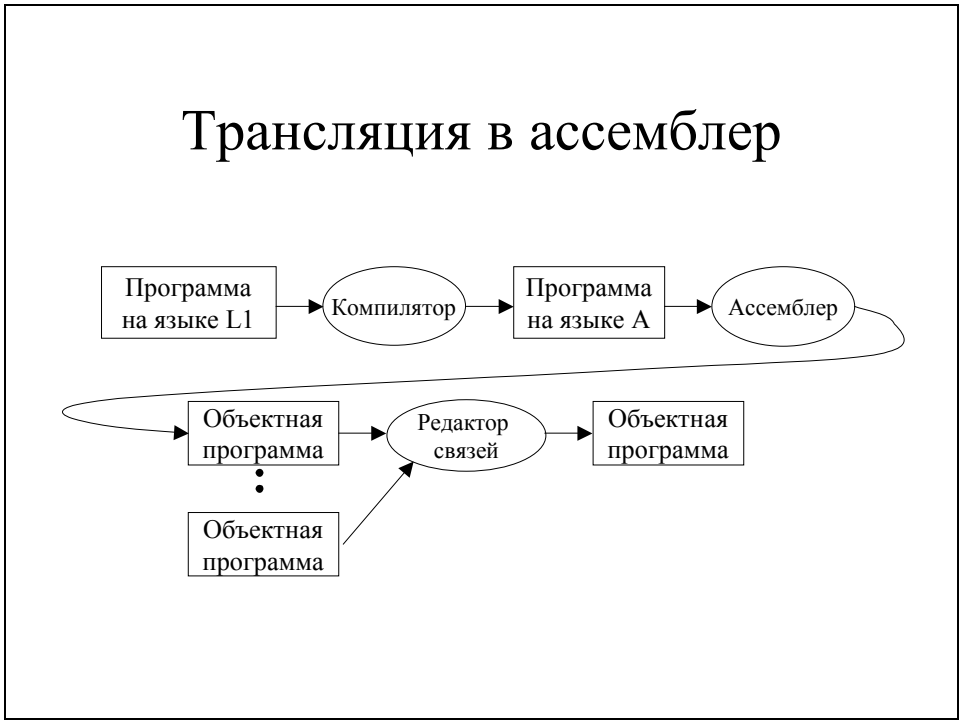
Целевая программа может быть
- последовательностью абсолютных машинных команд
- последовательностью перемещаемых машинных команд
- программой на языке ассемблера
- программой на некотором другом языке

Наиболее эффективным методом с точки зрения скорости компиляции является отображение исходной программы в *абсолютную программу* на машинном языке, пригодную для непосредственного исполнения. Такой тип компиляции больше всего подходит для небольших программ, не использующих независимо компилируемых подпрограмм.

Однако для более сложных программ необходимо создавать объектные программы в форме последовательности *перемещаемых машинных команд*. Перемещаемая машинная команда представляет собой команду, в которой адресация ячеек памяти производится относительно некоторого подвижного начала. Объектную программу называют также перемещаемым объектным сегментом. Этот сегмент может быть связан с другими сегментами, такими, как независимо компилируемые подпрограммы пользователя, подпрограммы ввода-вывода и библиотечные функции.

Такое *связывание* (*binding*), то есть создание единого перемещаемого объектного сегмента из набора различных сегментов, осуществляется программой, которая называется *редактором связей* (*linkage editor*). Далее единый перемещаемый объектный сегмент или модуль загрузки размещается в памяти программой, называемой *загрузчиком* (*loader*), которая превращает перемещаемые адреса в абсолютные. После этого программа готова к выполнению. Приведенная схема связывания называется *статическим связыванием* (*static binding*). В настоящее время часто испольузуется также *динамическое связывание* (*dynamic binding*). Это означает, что объектный сегмент не включается в единый перемещаемый объектный сегмент до тех пор, пока не произойдет обращение к этому сегменту. Выбор статического или динамического связывания

зависит от операционной системы. Заметим, что динамическое связывание часто используется для того, чтобы разрешить нескольким выполняемым программам разделять одну копию подпрограммы. Более подробно со статическим и динамическим связыванием вы можете познакомиться в курсе, посвященном операционным системам.



Трансляция в ассемблер

Трансляция программы в ассемблер несколько упрощает конструирование компилятора. Однако, такой подход удлиняет технологическую цепочку выполнения программы, так как объектная программа, порожденная компилятором, должна быть впоследствии обработана ассемблером, а также редактором связей и загрузчиком.

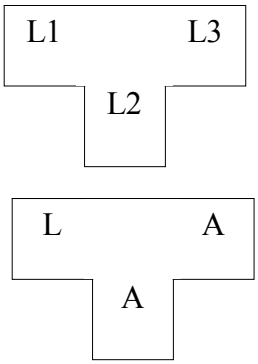
У трансляции в ассемблер есть несколько ощутимых преимуществ:

- уровень ассемблера все-таки выше, чем у машинного кода; поэтому при трансляции в ассемблер программисту не приходится возиться с некоторыми утомительными техническими деталями, например, ассемблер берет на себя разрешение операторов безусловного перехода на еще неопределенные метки (переходы вперед), распределение памяти под данные, расчет длин переходов и т.п.
- использование ассемблера позволяет отследить целый ряд ошибок, которые могут возникнуть при генерации кода (например, неправильная мнемоника команды); при генерации в машинные коды отловить такие ошибки значительно труднее
- порождаемый текст на ассемблере значительно читабельней, чем машинный код; это может помочь при отладке компилятора.

При генерации кода для платформы .NET у нас, в сущности, и нет никаких других возможностей, так как MSIL представляет собой высокоуровневый ассемблер, максимально абстрагированный от конкретных целевых платформ.

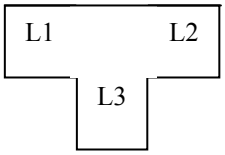
Т-диаграммы

Т-диаграммы – это удобный способ графического представления компиляторов

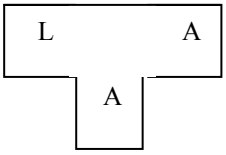


Т-диаграммы

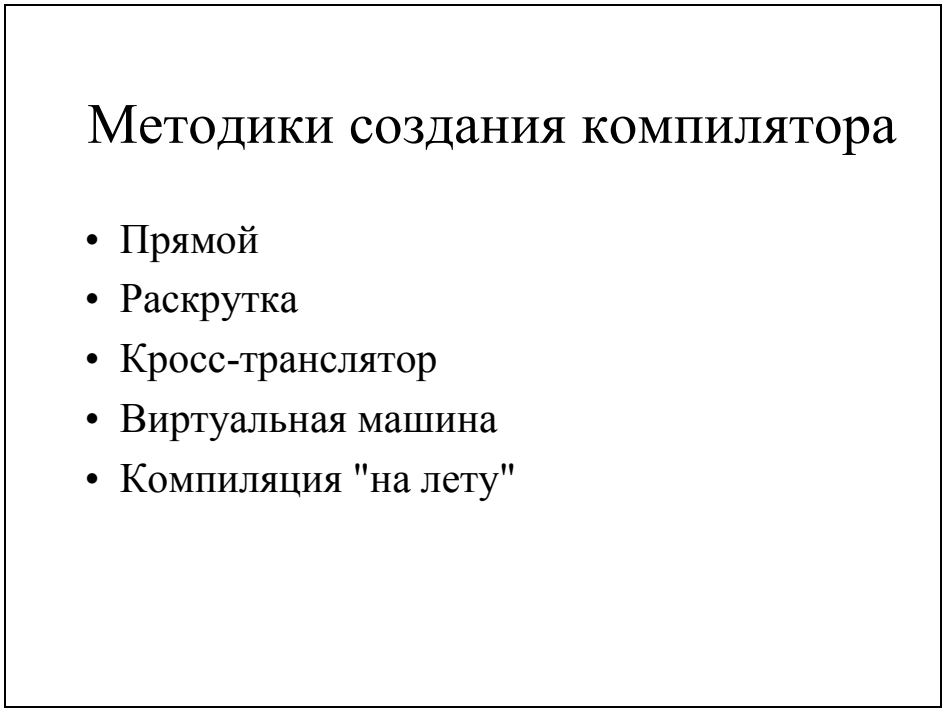
Мы обозначили компилятор, переводящий цепочки языка L_1 в цепочки языка L_2 и написанный на языке L_3, следующим образом: $K_{L_3}: L_1 \rightarrow L_3$. Для представления компилятора мы можем использовать также так называемые Т-диаграммы:



Компилятор, переводящий язык L в язык ассемблера A и написанный на языке A, можно представить следующими способами: $K_A: L \rightarrow A$ или



Т-диаграммы достаточно удобны при обсуждении различных способов получения компиляторов.



Методики создания компилятора

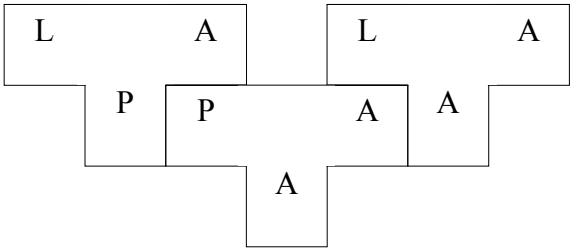
Написание компилятора может потребоваться в самых разнообразных условиях – для различных языков, целевых платформ, с различными требованиями к работе компилятора и т.д. Например, одна из типичных проблем написания компилятора связана с тем, что на целевой платформе могут отсутствовать подходящие средства для разработки. Именно так обычно обстоят дела при создании новой компьютерной архитектуры: на начальном этапе жизни компьютера системные программисты не имеют никаких средств разработки, кроме системы команд самого компьютера (на этом этапе даже ассемблер может отсутствовать). Бывают и такие случаи, когда компилятор создается для еще не существующей целевой платформы.

Таким образом, цели и условия написания компиляторов могут очень сильно варьироваться от одного проекта к другому. Поэтому существует целый ряд методик разработки компиляторов; мы остановимся на наиболее распространенных из них:

- прямой, при котором целевым языком является язык ассемблера
- метод раскрутки
- использование кросс-трансляторов
- использование виртуальных машин
- компиляция "на лету"

Метод раскрутки

Как избежать написания компилятора на ассемблере?



Метод раскрутки

Компилятор – это весьма большая и сложная программа; на написание и отладку компилятора на языке ассемблера можно истратить слишком много времени. Для того, чтобы как-то справиться с этой проблемой, был придуман метод раскрутки, суть которого заключается в следующем.

Пусть есть компилятор K_A: $P \rightarrow A$, где P – некоторый язык более высокого уровня, чем язык ассемблера. Тогда напишем K_P: $L \rightarrow A$, а затем применим компилятор K_A к компилятору K_P, т.е. получим $K_A = K_A(K_P)$: $L \rightarrow A$. Такая схема проиллюстрирована с помощью Т-диаграмм на слайде и называется *раскруткой* (*bootstrapping*[1]).

Описанная схема может быть использована при написании компилятора некоторого языка на нем самом. Пусть у нас есть компилятор некоторого подмножества S языка L в язык A, написанный на языке A, K_A: $S \rightarrow A$. Тогда мы можем написать K_L: $L \rightarrow A$ и получим новый компилятор $K_A = K_A(K_L)$. Мы используем это подмножество S для того, чтобы написать компилятор языка L в язык A, K_S: $L \rightarrow A$. Если теперь мы применим компилятор K_A к программе K_S, то получим $K_A = K_A(K_S)$: $L \rightarrow A$.

Впервые такая схема была применена в 1960 году при реализации языка Neliac. В 1971 году Вирт написал с использованием раскрутки транслятор языка Pascal, причем самый первый компилятор был оттранслирован вручную. Количество шагов раскрутки было больше 1, т.е. была построена последовательность языков $S_1 \subset S_2 \subset ... S_n = L$ и построена последовательность компиляторов: K_A: $S_1 \rightarrow A$, $K_A^1 = K_A(K_{S1}$: $S_2 \rightarrow A)$, ...

Раскрутку можно использовать и в следующей ситуации. Пусть у нас есть недостаточно эффективный компилятор K_A: $L \rightarrow A$. Можно написать более эффективный компилятор K_L: $L \rightarrow A$, а затем применить раскрутку.

---

[1] Название метода раскрутки произошло от фразы “to pull oneself up by one’s bootstrap”, т.е. "вытянуть себя за шнурки", аналогично легендарному методу барона Мюнхгаузена

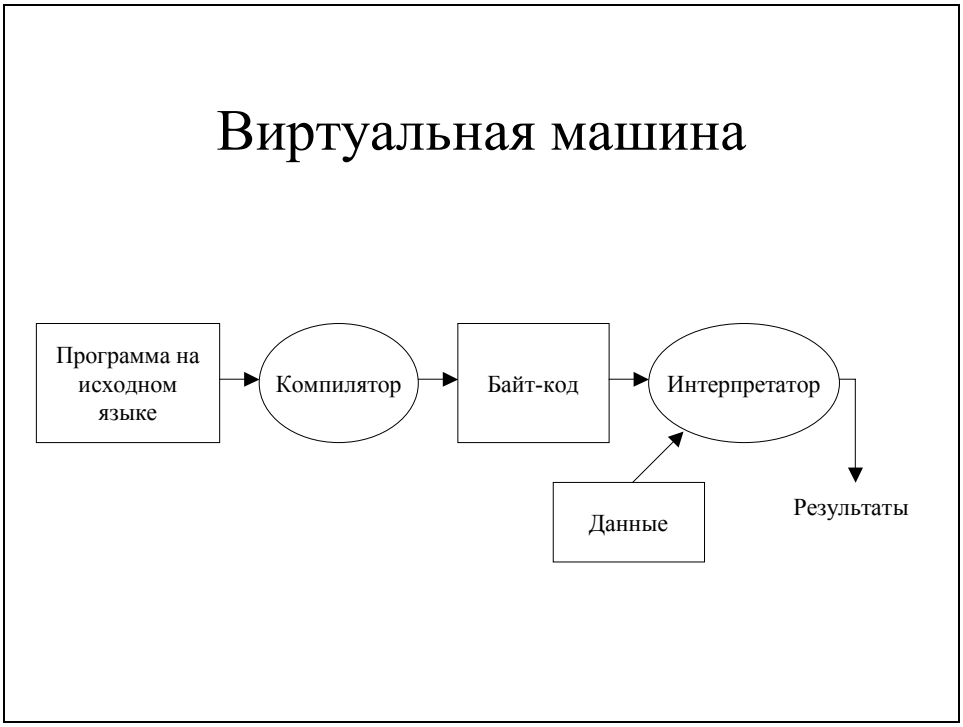
Кросс-транслятор

- Компилятор, который работает на одной платформе и создает код для другой платформы
- Более общий вопрос – создание переносимых компиляторов

Кросс-транслятор

Пусть у нас есть два компьютера: компьютер M с языком ассемблера A и компьютер M_1 с языком ассемблера A_1. Кроме того, предположим, что имеется компилятор K_{A1}: $P \rightarrow A_1$, а сам компьютер M по каким-то причинам не доступен либо пока еще не существует компилятор K_A: $P \rightarrow A$. Нас интересует компилятор K_A: $L \rightarrow A$. В такой ситуации мы можем использовать M_1 в качестве *инструментальной* машины и написать компилятор K_P: $L \rightarrow A$, который принято называть *кросс-транслятором* (*cross-compiler*). Как только машина M станет доступной, мы сможем перенести K_P на M и "раскрутить" его с помощью K_A. Понятно, что это решение достаточно трудоемко, поскольку могут возникнуть проблемы при переносе, например, из-за различий операционных систем.

Под переносимой (*portable)* программой понимается программа, которая может без перетрансляции выполняться на нескольких (по меньшей мере, на двух) платформах. В связи с этим возникает вопрос о переносимости объектных программ, создаваемых компилятором. Компиляторы, созданные по методикам, рассмотренным выше, порождают *непереносимые* (*non-portable*) объектные программы. Поскольку компилятор, в конечном итоге, является программой, то мы можем говорить и о переносимых компиляторах. Одним из способов получения переносимых объектных программ является генерация объектной программы на языке более высокого уровня, чем язык ассемблера. Такие компиляторы иногда называют *конвертерами (converter)*.

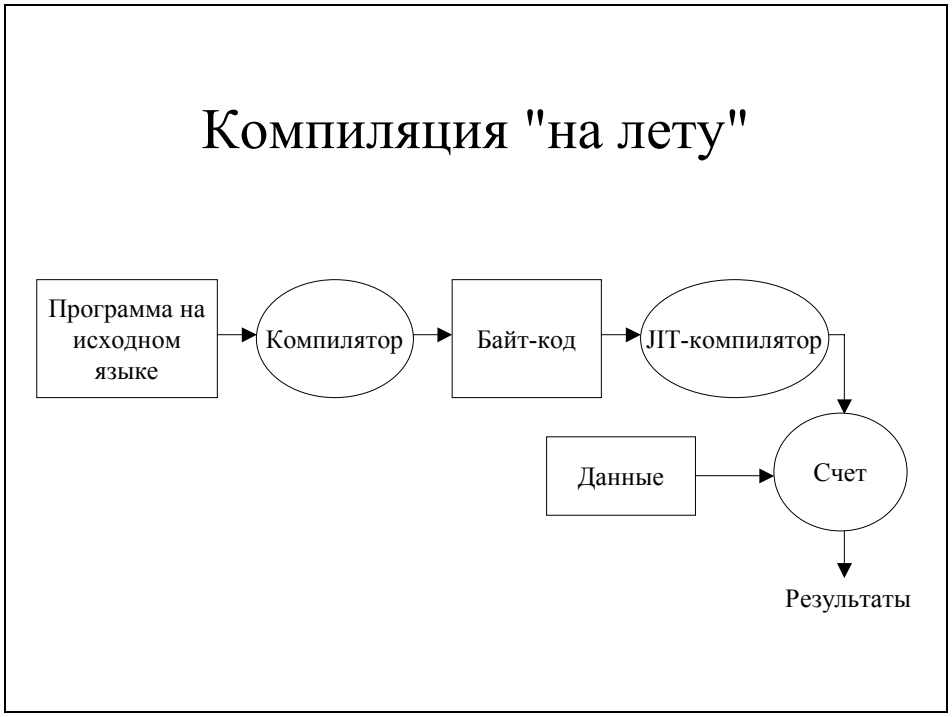


Виртуальная машина

Другой способ получения *переносимой* (*portable*) объектной программы связан с использованием *виртуальных машин* (*virtual machine*). При таком подходе исходный язык транслируется в коды некоторой специально разработанной машины, которую никто не собирается реализовывать "в железе". Затем для каждой целевой платформы пишется интерпретатор виртуальной машины.

Понятно, что архитектура виртуальной машины должна быть разработана таким образом, чтобы конструкции исходного языка удобно отображались в систему команд и сама система команд не была слишком сложной. При выполнении этих условий можно достаточно быстро написать интерпретатор виртуальной машины.

Одна из первых широко известных виртуальных машин была разработана в 70-х годах Н. Виртом при написании компилятора Pascal-P. Этот компилятор генерировал специальный код, названный P-кодом и представляющий собой последовательность инструкций гипотетической стековой машины. Сегодня идея виртуальных машин приобрела широкую известность благодаря языку Java, компиляторы которого обычно генерируют так называемый *байт-код*, т.е. объектный код, который представляет собой последовательность команд виртуальной Java-машины.

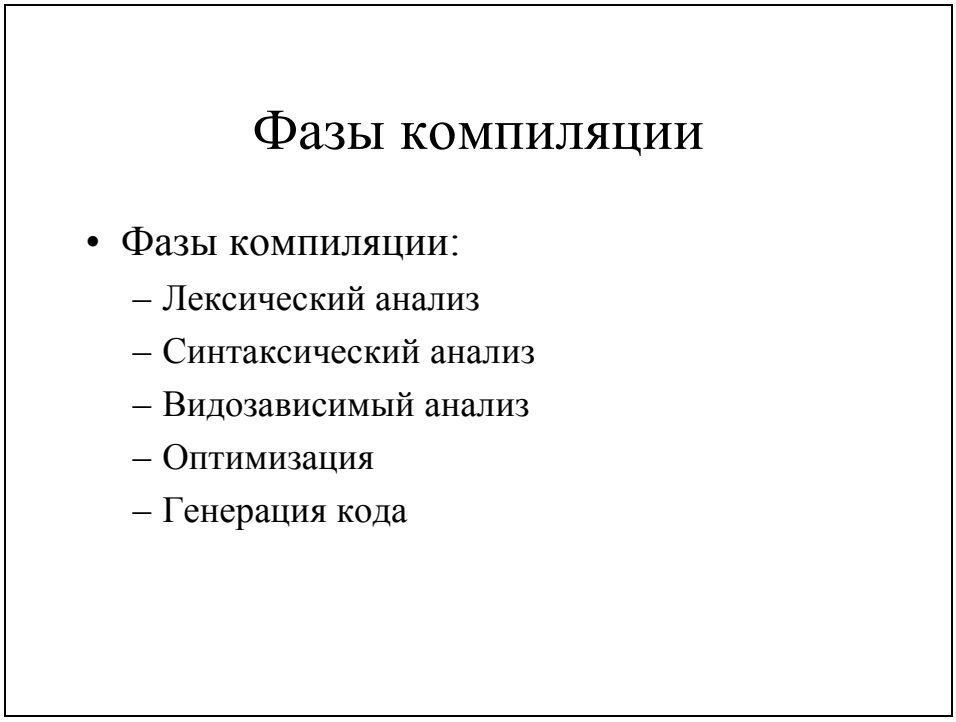


Компиляция "на лету"

Основная неприятность, связанная с использованием виртуальных машин, заключается в том, что обычно время выполнения интерпретируемой программы значительно больше, чем время работы программы, оттранслированной в "родной" машинный код. Для того, чтобы увеличить скорость работы приложений, была разработана технология *компиляции "на лету"* (*Just-In-Time compiling*; иногда такой подход называют также динамической компиляцией). Идея заключается в том, что JIT-компилятор генерирует машинный код прямо в оперативной памяти, не сохраняя его. Это приводит к значительному увеличению скорости выполнения приложения. Как мы уже говорили в лекции 1, именно так и устроена платформа .NET.

Часто JIT-компилятор используется вместе с интерпретатором виртуальной машины. Организовывается это следующим образом. Вначале сгенерированный байт-код поступает на вход интерпретатору виртуальной машины, которая его интерпретирует. Одновременно с интерпретатором работает программа, которая вычисляет время интерпретации какого-то куска байт-кода, например, процедуры. Если оказывается, что время интерпретации некоторого куска кода достаточно большое, то вызывается JIT-компилятор, который транслирует его в "родные" машинные коды. Когда при выполнении приложения произойдет повторное обращение к этому куску кода, то он уже не будет интерпретироваться, а будет выполняться сгенерированный фрагмент машинного кода.

Использование связки "компилятор+интерпретатор+JIT-компилятор" позволяет заметно повысить скорость выполнения исходной программы, причем переносимость кода, создаваемого компилятором, естественно, сохраняется.

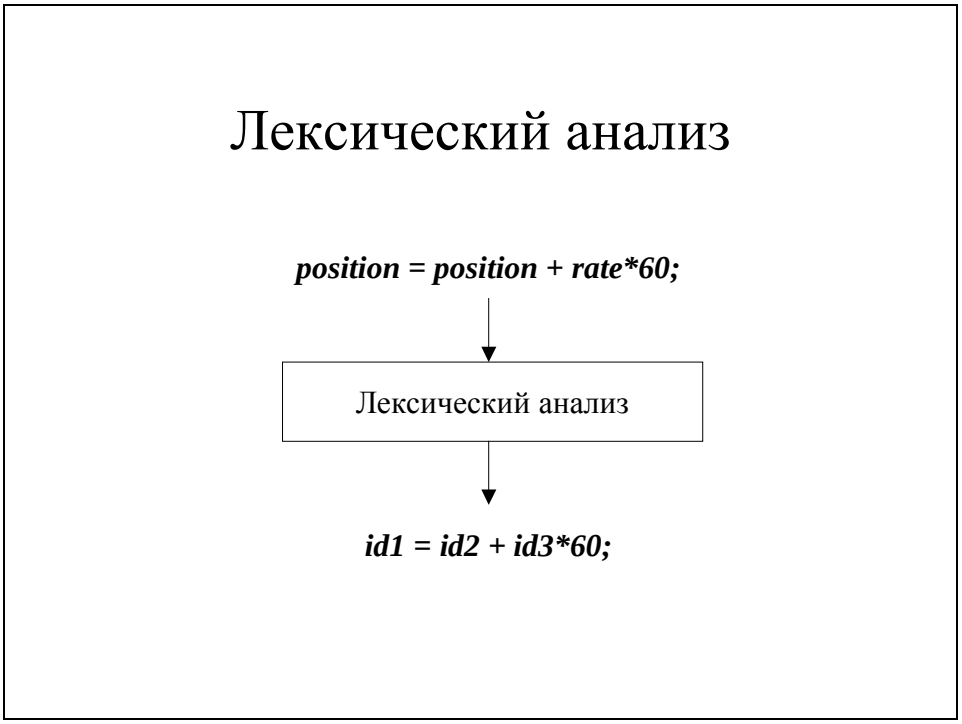


Фазы компиляции

Процесс создания компилятора можно свести к решению нескольких задач, которые принято называть *фазами компиляции* (*compilation phases*). Обычно компилятор состоит из следующих фаз:

- лексический анализ
- синтаксический анализ
- видозависимый анализ
- оптимизация
- генерация кода.

Сформулируем основные цели каждой из фаз компиляции. Мы продемонстрируем преобразования, которым подвергается исходная программа на перечисленных фазах компиляции, на небольшом примере — мы рассмотрим оператор присваивания *position = initial + rate * 60*, причем предположим, что все переменные вещественные.



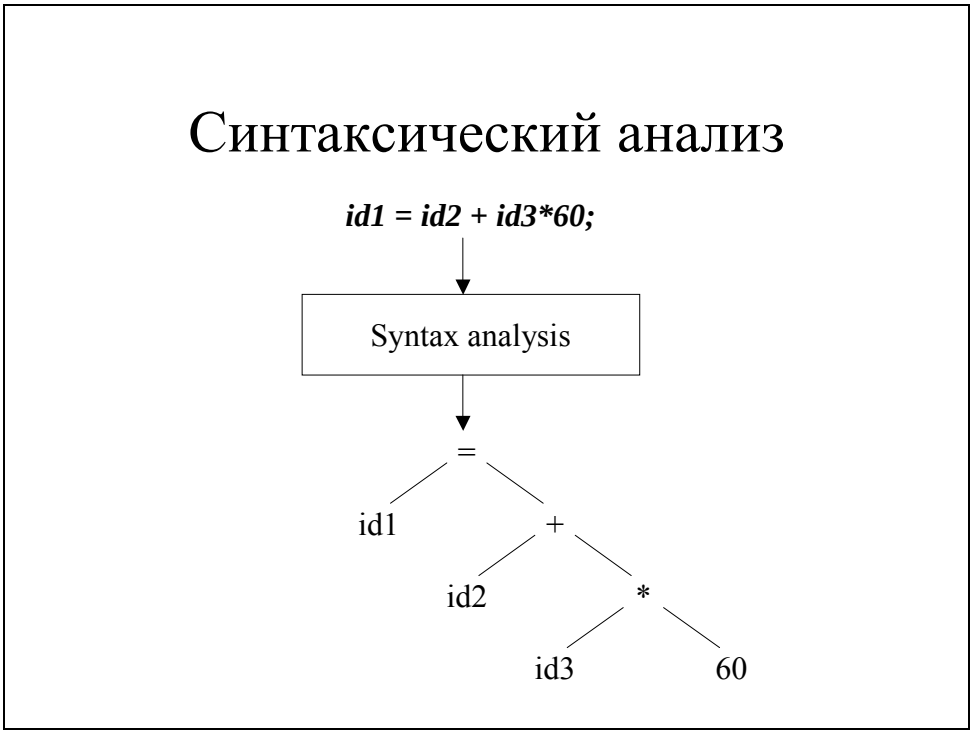
Лексический анализ

Входом компилятора служит программа на исходном языке программирования. С точки зрения компилятора это просто последовательность символов. Задача первой фазы компиляции, *лексического анализатора (lexical analysis)*, заключается в выделении некоторых более "крупных" единиц, *лексем*. Примерами лексем являются ключевые слова, идентификаторы, константные значения (числа, строки, логические) и т.п.

На этапе лексического анализа обычно также выполняются такие действия, как удаление комментариев и обработка директив условной компиляции.

Для отображения некоторых лексем достаточно всего одного числа, то есть лексического класса, в то время как для записи других лексем может потребоваться пара, состоящая из номера лексического класса и ссылки в таблицу внешних представлений. Хорошая модель лексического анализатора – конечный преобразователь.

Лексический анализ будет подробно рассмотрен в лекции 5.

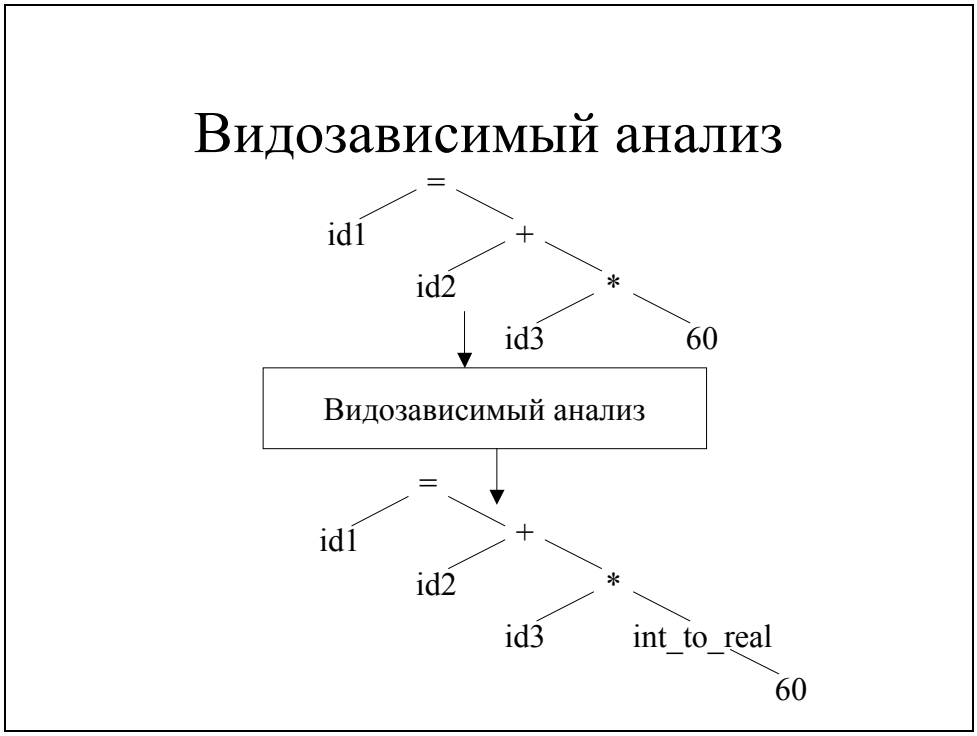


Синтаксический анализ

Синтаксический анализатор (syntax analyzer, parser) получает на вход результат работы лексического анализатора и разбирает его в соответствии с некоторой грамматикой. Эта грамматика аналогична грамматике, используемой при описании входного языка. Однако грамматика входного языка обычно не уточняет, какие конструкции следует считать лексемами.

Синтаксический анализ является одной из наиболее формализованных и хорошо изученных фаз компиляции. Лекция 4 посвящена математическому аппарату, используемому при описании языков и создании синтаксических анализаторов. Различные методы построения синтаксических анализаторов будут рассмотрены в лекциях 6–8.

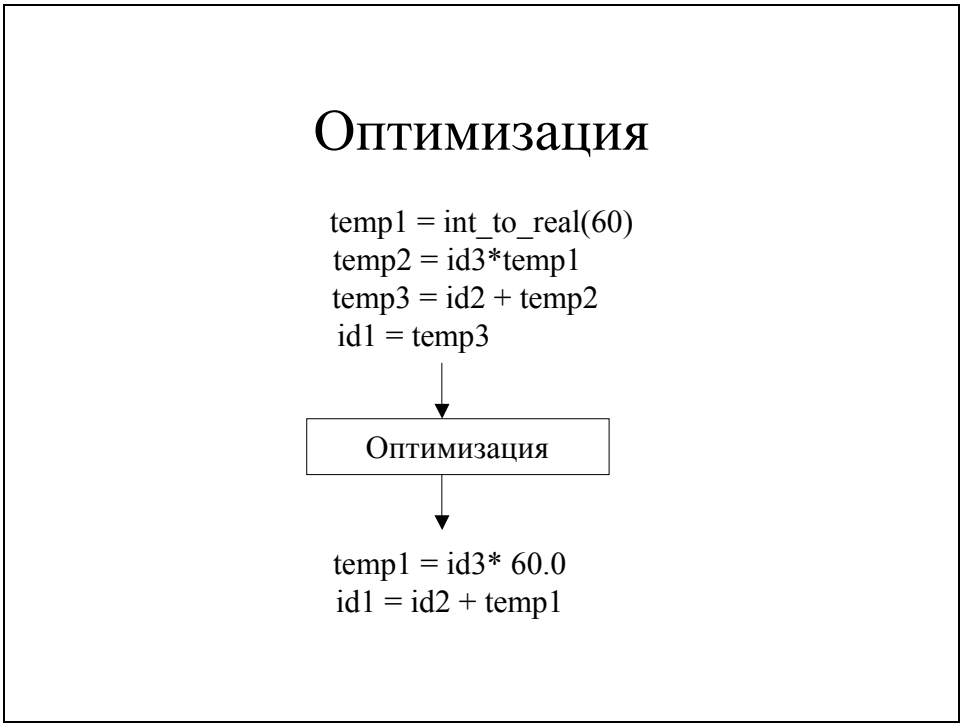
После синтаксического анализа можно считать, что исходная программа преобразована в некоторое промежуточное представление. Некоторые распространенные формы промежуточного представления программы будут рассмотрены в лекции 9. Пока же мы остановимся на одной формой промежуточного представления, которая будет использована в нашем курсе, – на дереве разбора программы (иногда его также называют синтаксическим деревом). В дереве разбора программы внутренние узлы соответствуют операциям, а листья представляют операнды.



Видозависимый анализ

Видозависимый анализ (type checking), иногда также называемый *семантическим анализом (semantic analysis)*, обычно заключается в проверке правильности типов данных, используемых в программе. Кроме того, на этом этапе компилятор должен также проверить, соблюдаются ли определенные контекстные условия входного языка. В современных языках программирования одним из примеров контекстных условий может служить обязательность описания переменных: для каждого использующего вхождения идентификатора должно существовать единственное определяющее вхождение. Другой пример контекстного условия: число и атрибуты фактических параметров вызова процедуры должны быть согласованы с определением этой процедуры.

Такие контекстные условия не всегда могут быть проверены во время синтаксического анализа и потому обычно выделяются в отдельную фазу. Эта фаза компиляции подробно обсуждается в лекции 9.

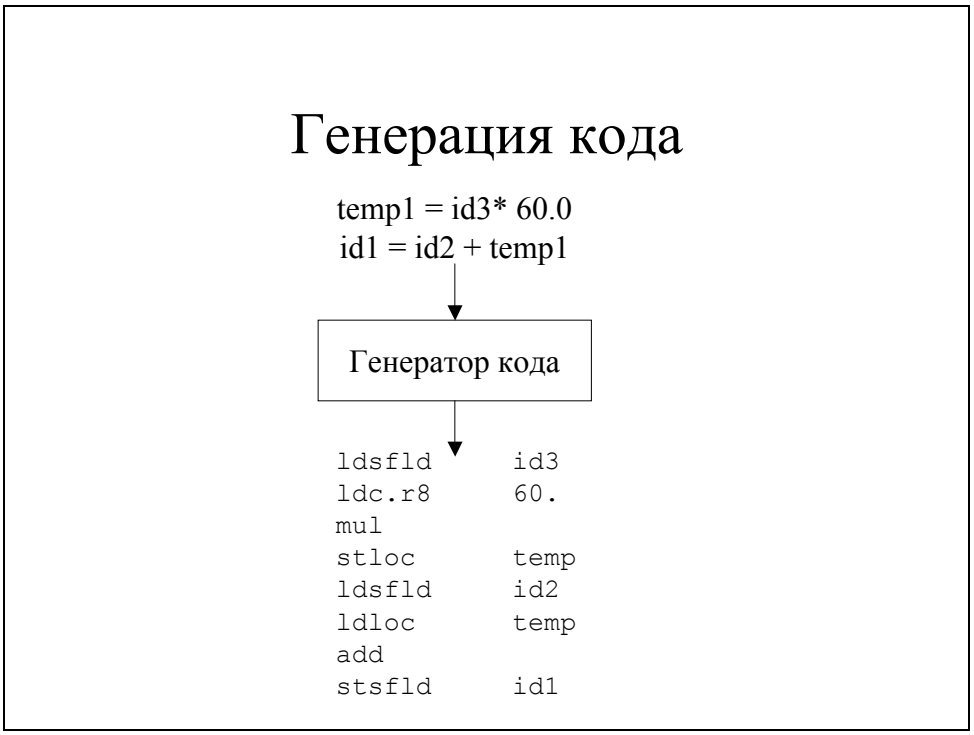


Оптимизация кода

Основная цель *фазы оптимизации (code optimization)* заключается в преобразовании промежуточного представления программы в целях повышения эффективности результирующей объектной программы. Отметим, что существует различные критерии эффкетивности, например, скорость исполнения или объем памяти, требуемый программе. Очевидно, что все преобразования, осуществляемые на фазе оптимизации, должны приводить к программе, эквивалентной исходной.

Некоторые оптимизации тривиальны, другие требуют достаточно сложного анализа программы. В лекциях 11 и 12 мы рассмотрим анализ потоков управления программы и анализ потоков данных. Затем в лекции 13 мы рассмотрим сам этап оптимизации программ и рассмотрим наиболее распространенные оптимизации:

- константные вычисления
- уменьшение силы операций
- выделение общих подвыражений
- чистка циклов и т.д.



Генерация кода

Наконец, по оптимизированной версии промежуточного представления генерируется объектная программа. Эту задачу решает фаза *генерации кода (code generator)*. В лекции 14 мы изучим основные свойства целевой платформы .NET и рассмотрим процесс генерации кода для этой платформы. В лекции 15 мы рассмотрим более сложные алгоритмы генерации, использующие методику восходящего переписывания деревьев (BURS). Во многих случаях такой подход может значительно улучшить качество порождаемого кода.

Помимо собственно генерации кода, на этом этапе необходимо решить множество сопутствующих проблем, например:

- распределение памяти, т.е. отображение имен исходной программы в адреса памяти
- распределение регистров, т.е. определение для каждой точки программы множества переменных, которые должны быть размещены в регистрах
- выбор такой последовательности записи значений в регистры, которая избавила бы от необходимости частой выгрузки значений из регистров, а затем повторной загрузки

Практически все эти задачи решаются окружением времени исполнения .NET и потому остались за пределами данного курса. Тем не менее, для полноты картины мы рассмотрим проблемы управления памятью в лекции 12.

Внешний и внутренний интерфейсы

- Фазы компиляции, в большей степени зависящие от входного языка и в меньшей степени от целевого, образуют внешний интерфейс компилятора.
- Фазы, в меньшей степени зависящие от входного языка и в большей степени от целевого, образуют внутренний интерфейс компилятора.

Внешний и внутренний интерфейсы

Нетрудно заметить, что одни фазы компиляции в большей степени зависят от входного языка и в меньшей степени от целевого. Другие фазы, наоборот, в меньшей степени зависят от входного языка и в большей степени от целевого. Например, лексический, синтаксический, видозависимый анализ и некоторые оптимизации не слишком значительно зависят от целевого языка. Эти фазы иногда объединяют вместе под названием *внешний интерфейс* или *front-end*. Front-end подразумевает также обработку ошибок, которые могут встретиться на перечисленных фазах (вопросы обработки ошибок обсуждаются в лекции 9). Остальные фазы, несущие на себе отпечаток целевого языка, называются *внутренним интерфейсом (back-end)*.

Таким образом, если мы хотим иметь компиляторы некоторого языка на разных платформах, то наша задача сводится к написанию нескольких back-end'ов без изменения front-end'а. С другой стороны, для создания многоязыкового компилятора достаточно написать несколько front-end'ов.

Просмотры

- Под просмотром (или проходом) компилятора понимается процесс обработки **всего**, возможно, уже преобразованного, текста исходной программы.

Просмотры

Обсудим еще один важный термин, а именно, *просмотр (passes)*. Под просмотром (или проходом) компилятора понимается процесс обработки **всего**, возможно, уже преобразованного, текста исходной программы.

Одна или несколько фаз компиляции могут выполняться на одном просмотре. Например, лексический анализ и синтаксический анализ часто выполняются на одном просмотре, т.е. синтаксический анализатор может обращается к лексическому анализатору за очередной лексемой лишь по мере необходимости. С другой стороны, некоторые оптимизации могут выполняться на разных просмотрах.

Передача информации между просмотрами происходит в терминах так называемых промежуточных языков. Таким образом, если компилятор состоит из *N* просмотров, то должно быть определено *N–1* промежуточных языков. Каков этот промежуточный язык, зависит от разработчиков компилятора. Обычно программа на промежуточном языке представляет собой синтаксическое дерево и, возможно, какие-то внутренние таблицы компилятора.

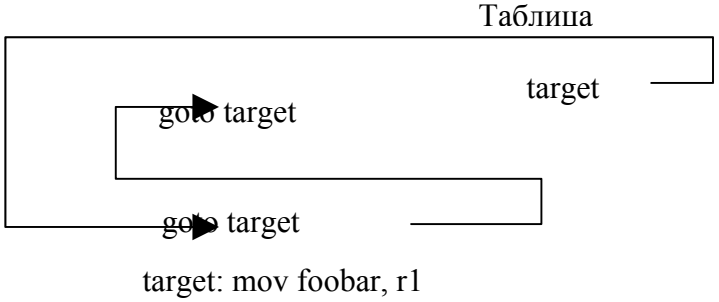
Желательно иметь относительно мало просмотров, т.к. чтение программы на одном промежуточном языке и записи ее на другом промежуточном языке может занимать довольно много времени. С другой стороны, объединяя несколько фаз в один просмотр, мы должны помнить о том, что иногда мы не можем выполнить некоторую фазу, не получив информацию из предыдущих фаз. Например, C# позволяет использовать имя метода до того, как он был описан. Понятно, что мы не можем выполнить видозависимый анализ до тех пор, пока мы не будем знать имена и типы всех методов объектов. Таким образом, эти задачи должны решаться на разных просмотрах.

Техника "заплат"

Аналогично, если в языке есть оператор go to, то мы можем использовать его не только для перехода назад, но и для перехода вперед. Таким образом, мы не всегда можем определить операнд команды перехода, по крайней мере до тех пор, пока не доберемся до конца программы. Однако в простых случаях мы можем использовать *технику "заплат"* (*backpatching*). Рассмотрим фрагмент программы на языке ассемблера:

```
            ...
            goto target;
            ...
            goto target;
            ...
target:     mov foobar, r1
```

Транслятор может при генерации кода сгенерировать вначале скелет команды перехода, запомнив ее адрес в строке некоторой таблицы, соответствующей идентификатору *target*. Когда адрес этого идентификатора будет определен, мы сможем завершить формирование команд перехода, пробежав по имеющемуся у нас списку.



Литература к лекции

- А. Ахо, Р. Сети, Дж. Ульман "Компиляторы: принципы, технологии и инструменты", М.: "Вильямс", 2001, 768 стр.

Литература к лекции

- А. Ахо, Р. Сети, Дж. Ульман "Компиляторы: принципы, технологии и инструменты", М.: "Вильямс", 2001, 768 стр.