МОДУЛЬ 1

Раздел 1. Языки и метаязыки

Мета- (с греч. μετά- — между, после, через), часть сложных слов, обозначающая абстрагированность, обобщённость, следование за чем-либо. В древнегреческом языке предлог μετά (metá) и приставка μετα- имеет значения: «после», «следующее», «за».

С префиксом МЕТА-

- **Метаданные** — данные о данных (источник, формат, дата создания, объем и т. п.).
- **Метатеория** — теория, изучающая свойства другой теории. Любая дисциплина имеет метатеорию, которая является теоретическими соображениями о её основах и методах.

Метаязык - язык, предназначенный для описания другого языка (национального, другой семиотической системы и т.д.).

Впервые различение языка-объекта и метаязыка проведено **Давидом Гильбертом** (нем. David Hilbert; 1862-1943, немецкий математик-универсал), применительно к различению “математики” и “метаматематики” и без использования соответствующей терминологии.

Понятия «язык-объект» и «метаязык» были введены **Альфредом Тарским** (польск. *Alfred Tarski)* и Рудольфом Карнапом (нем. Rudolf Carnap)в середине 1930-х гг. Это различение стало активно использоваться в исследованиях проблем математической логики и оснований математики.

Позже понятие "метаязык" стали применять в лингвистике, семиотике, в философии и методологии науки.

Понятие метаязыка используется:

- в лингвистике, при описании естественных языков;
- в математике и логике — при исследовании формальных языков исчислений;
- в информатике — метаданные, служащие для описания имеющихся;

В каждой из этих дисциплин применение термина “метаязык” при сохранении его общего значения приобретает некоторые специфические черты.

Так, в логике и математике метаязык обычно понимается как средство изучения формализованных языков — логических и математических исчислений, или (в несколько иной формулировке) как формализованный или неформализованный язык, на котором формулируются утверждения метаматематики; в лингвистике метаязык рассматривается как средство построения металингвистики и т. д. При всем различии таких трактовок общепризнано, что метаязык должен быть богаче соответствующего языка-

объекта, так как он должен содержать не только обозначения для всех имен и выражений последнего, но и фиксировать с помощью своих специфических средств их свойства и устанавливать различного рода отношения и связи между ними.

С лестницей метаязыков Тарского тесно связана **Теория типов и Теория дескрипций** (англ. *Theory of descriptions*).

Общие особенности языков программирования и их трансляции

Любой естественный язык возникал как средство общения людей. Именно по этой причине ему присущи такие особенности как:

- изменчивость, которая состоит в непостоянстве словарного состава языка;
- неоднозначность трактовки фраз различными людьми;
- избыточность.

Перечисленные особенности не позволяют применить естественный язык для записи алгоритма, поскольку одним из свойств алгоритма является его детерминированность, т.е. однозначность выполнения шагов любым исполнителем. Наиболее простой путь преодоления нежелательных свойств естественных языков – построение языков искусственных со строгим синтаксисом и полной смысловой определенностью – такие языки получили название формальных.

Метаязык Хомского имеет следующую систему обозначений:

- символ “®” отделяет левую часть правила от правой (читается как "порождает“ и "это есть");
- символ “®” отделяет левую часть правила от правой (читается как "порождает“ и "это есть");
- нетерминалы обозначаются буквой **A** с индексом, указывающим на его номер;
- терминалы - это символы используемые в описываемом языке;
- каждое правило определяет порождение одной новой цепочки, причем один и тот же нетерминал может встречаться в нескольких правилах слева.

Описание идентификатора на метаязыке Хомского будет выглядеть следующим образом:

1. A_1 ® A	**23. A_1 ® W**	**45. A_2 ® s**
2. A_1 ® B	24. A_1 ® X	46. A_2 ® t
3. A_1 ® C	25. A_1 ® Y	47. A_2 ® u
….	….	…..

20. $A_1 \to T$ 42. $A_2 \to p$ 64. $A_3 \to A_3A_1$

21. $A_1 \to U$ 43. $A_2 \to q$ 65. $A_4 \to A_3A_2$

Язык программирования - это знаковая система для планирования поведения компьютеров.

Языки программирования достаточно сильно отличаются друг от друга по назначению, структуре, семантической сложности, методам реализации. Это накладывает свои специфические особенности на разработку конкретных трансляторов.

Языки программирования являются инструментами для решения задач в разных предметных областях, что определяет специфику их организации и различия по назначению. Например, **Фортран** ориентирован на научные расчеты и сейчас эффективен при работе с кластерами, **С** предназначен для системного программирования, **Пролог**, эффективно описывающий задачи логического вывода, **Лисп**, используемый для рекурсивной обработки списков. Такие примеры можно продолжать и продолжать.

Каждая из предметных областей предъявляет свои требования к организации самого языка. Поэтому можно отметить разнообразие форм представления операторов и выражений, различие в наборе базовых операций, снижение эффективности программирования при решении задач, не связанных с предметной областью. Языковые различия отражаются и в структуре трансляции. *Лисп* и *Пролог* чаще всего выполняются в режиме интерпретации из-за того, что используют динамическое формирование типов данных в ходе вычислений. Для трансляторов с языка *Фортран* характерна агрессивная оптимизация результирующего машинного кода, которая становится возможной благодаря относительно простой семантике конструкций языка - в частности, благодаря отсутствию механизмов альтернативного именования переменных через указатели или ссылки. Наличие же указателей в *языке С* предъявляет специфические требования к динамическому распределению памяти.

Структура языка характеризует иерархические отношения между его понятиями, которые описываются **синтаксическими правилами**. Языки программирования могут сильно отличаться друг от друга по организации отдельных понятий и по отношениям между ними. Язык программирования *PL/1* допускает произвольное вложение процедур и функций, тогда как в *С* все функции должны находиться на внешнем уровне вложенности. Язык *С++* допускает описание переменных в любой точке программы перед первым ее использованием, а в Паскале переменные должны быть определены в специальной области описания. Еще дальше в этом вопросе идет PL/1, который допускает описание переменной после ее использования. Или описание можно вообще опустить и руководствоваться правилами, принятыми по умолчанию. В зависимости от принятого решения, транслятор может анализировать

программу за один или несколько проходов, что влияет на скорость трансляции.

Семантика языков программирования изменяется в очень широких пределах. Они отличаются не только по особенностям реализации отдельных операций, но и по парадигмам программирования, определяющим принципиальные различия в методах разработки программ. Специфика реализации операций может касаться как структуры обрабатываемых данных, так и правил обработки одних и тех же типов данных. Такие языки, как PL/1 и APL (A Programming Language) поддерживают выполнение матричных и векторных операций. Большинство же языков работают в основном со скалярами, предоставляя для обработки массивов процедуры и функции, написанные программистами. Но даже при выполнении операции сложения двух целых чисел такие языки, как С и Фортран могут вести себя по-разному. Даже один и тот же язык может быть реализован нескольким способами. Это связано с тем, что **теория формальных грамматик** допускает различные методы разбора одних и тех же предложений. В соответствии с этим трансляторы разными способами могут получать один и тот же результат (объектную программу) по первоначальному исходному тексту.

Вместе с тем, все языки программирования обладают рядом общих характеристик и параметров. Эта общность определяет и схожие для всех языков принципы трансляции.

Языки программирования предназначены для облегчения программирования. Поэтому их операторы и структуры данных более мощные, чем в машинных языках.

Для повышения наглядности программ вместо числовых кодов используются символические или графические представления конструкций языка, более удобные для их восприятия человеком.

Для любого языка определяется:

- Множество символов, которые можно использовать для записи правильных программ (алфавит), основные элементы;
- Множество правильных конструкций программ (синтаксис).
- "Смысл" правильного блока конструкций программы (семантика).

Независимо от специфики языка **процесс трансляции можно считать функциональным преобразователем F, обеспечивающим однозначное отображение X в Y, где X - программа на исходном языке, Y - программа на выходном языке**. Поэтому сам процесс трансляции формально можно представить достаточно просто и понятно:

$$Y = F(X)$$

Формально каждая правильная программа X - это цепочка символов из некоторого алфавита **V**, преобразуемая в соответствующую ей цепочку Y, составленную из символов алфавита $\mathbf{V_1}$.

Язык программирования, как и любая сложная система, определяется через иерархию понятий, задающую взаимосвязи между его элементами. Эти понятия связаны между собой в соответствии с синтаксическими правилами. Каждая из программ, построенная по этим правилам, имеет соответствующую иерархическую структуру.

В связи с этим для всех языков и их программ можно дополнительно выделить следующие общие черты: ***каждый язык должен содержать правила, позволяющие порождать программы, соответствующие этому языку или распознавать соответствие между написанными программами и заданным языком***.

1.1 Парадигмы языков программирования

Итак, наряду с традиционным процедурным программированием, называемым также императивным, существуют такие парадигмы как функциональное программирование, логическое программирование и объектно-ориентированное программирование. Структура понятий и объектов языков сильно зависит от избранной парадигмы, что также влияет на реализацию транслятора.

На сегодняшний день имеются четыре основные парадигмы языков программирования, отражающие вычислительные модели, с помощью которых описывается большинство существующих методов программирования:

□ императивная;
□ функциональная;
□ декларативная;
□ объектно-ориентированная.

Императивные языки

Императивные (процедурные) языки – это языки программирования, управляемые командами, или операторами языка. Основной концепцией императивного языка является состояние компьютера – множество всех значений всех ячеек (слов) памяти компьютера. Данная модель вытекает из особенностей аппаратной части компьютера стандартной архитектуры, названной "фон-неймановской" в честь одного из ее авторов – американского математика Джона фон Неймана. В таком компьютере данные, подлежащие обработке, и программы хранятся в одной памяти, называемой *оперативной.* Центральный процессор получает из оперативной памяти очередную команду на входном языке, декодирует ее, выбирает из памяти указанные в качестве операндов входные данные, выполняет команду и возвращает в память результат.

Программа на императивном языке представляет собой последовательность команд (операторов), которые выполняются в порядке их написания. Выполнение каждой команды приводит к изменению состояния компьютера. Основными элементами императивных языков программирования,

ориентированных на фон-неймановскую архитектуру, являются **переменные**, моделирующие ячейки памяти компьютера, и **операторы присваивания**, осуществляющие пересылку данных. Выполнение оператора присваивания может быть представлено как последовательность обращений к ячейкам памяти за операндами выражения из правой части оператора присваивания, передача их процессору, вычисление выражения и возвращение результата вычисления в ячейку памяти, представляющую собой переменную из левой части оператора присваивания.

Императивные языки поддерживают, как правило, один или несколько итеративных циклов, различающихся синтаксисом. Большинство императивных языков включает в себя конструкции, позволяющие программировать рекурсивные алгоритмы, но их реализация на компьютерах с фон-неймановской архитектурой не эффективна, что связано с необходимостью программного моделирования стековой памяти.

К императивным языкам относятся такие распространенные языки программирования, как ALGOL-60, BASIC, FORTRAN, PL/1, Ada, Pascal, C, C++, Java.

Успех языка FORTRAN определен в частности благодаря большой поддержке его различными производителями вычислительной техники, которые потратили много усилий на его реализацию и подробное описание.

Языки функционального программирования

В ***языках функционального программирования (аппликативных языках)*** вычисления в основном производятся путем применения функций к заданному набору данных. Разработка программ заключается в создании из простых функций более сложных, которые последовательно применяются к начальным данным до тех пор, пока не получится конечный результат. Типичная программа, написанная на функциональном языке, имеет следующий вид:

$функция_n$ *(…* $функция_2$ *(*$функция_1$ *(данные))* ...).

На практике наибольшее распространение получили язык функционального программирования LISP и два его диалекта: язык Common LISP и язык Scheme. Основной структурой данных языка LISP являются связные списки, элементами которых могут быть либо атомы (идентификаторы или числовые константы), либо другие связные списки.

Языки функционального программирования не получили широкого распространения из-за невысокой эффективности реализации их на компьютерах с фон-неймановской архитектурой. На компьютерах с параллельной архитектурой трансляторы для функциональных языков реализуются более эффективно, однако они еще не конкурентоспособны по сравнению с реализациями императивных языков.

Кроме языка LISP, основной областью применения которого являются системы искусственного интеллекта, известны и другие языки функционального программирования: ML (Meta Language) – семейство строгих языков функциональногопрограммирования), Miranda (созданный в

1985 году Дэвидом Тёрнером) и Haskell (в честь математика Хаскелла Карри ученика Гильберта).

Программирование, как на императивных, так и на функциональных языках является *процедурным*. Это означает, что программы на этих языках содержат указания, *как* нужно выполнять вычисления.

Декларативные языки

Декларативные языки программирования – это языки программирования, в которых операторы представляют собой объявления или высказывания в символьной логике. Типичным примером таких языков являются *языки логического программирования* (языки, основанные на системе правил).

В программах на языках логического программирования соответствующие действия выполняются только при наличии необходимого разрешающего условия. Программа на языке логического программирования схематично выглядит следующим образом:

разрешающее условие 1 → последовательность операторов 1
разрешающее условие 2 → последовательность операторов 2
...
разрешающее условие n → последовательность операторов n

В отличие от императивных языков, операторы программы на языке логического программирования выполняются не в том порядке, как они записаны в программе. Порядок выполнения операторов определяется системой реализации правил.

Характерной особенностью декларативных языков является их ***декларативная семантика***. Основная концепция декларативной семантики заключается в том, что смысл каждого оператора не зависит от того, как этот оператор используется в программе. Так, смысл заданного высказывания в языке логического программирования можно точно определить по самому оператору. Декларативная семантика намного проще семантики императивных языков, что может рассматриваться как преимущество декларативных языков над императивными.

Наиболее распространенным языком логического программирования является язык Prolog. Основными областями применения языка Prolog являются экспертные системы, системы обработки текстов на естественных языках и системы управления реляционными базами данных. К языкам программирования, основанным на системе правил, можно отнести языки синтаксического разбора (например, YACC – Yet Another Compiler Compiler), в которых синтаксис анализируемой программы рассматривается в качестве разрешающего условия.

Объектно-ориентированные языки

Концепция объектно-ориентированного программирования складывается из трех ключевых понятий: *абстракция данных, наследование* и *полиморфизм.*

Абстракция данных позволяет *инкапсулировать* множество объектов данных (члены класса) и набор абстрактных операций над этими объектами данных (методы класса), ограничивая доступ к данным только через определенные абстрактные операции. Инкапсуляция позволяет изменять реализацию класса без плохо контролируемых последствий для программы в целом.

Наследование – это свойство классов создавать из базовых классов производные, которые наследуют свойства базовых классов и могут содержать новые элементы данных и методы. Наследование позволяет создавать иерархии классов и является эффективным средством внесения изменений и дополнений в программы.

Полиморфизм означает возможность одной операции или имени функции ссылаться на любое количество определений функций, зависящих от типа данных параметров и результатов. Это свойство объектно-ориентированных языков программирования обеспечивается динамическим связыванием сообщений (вызовов методов) с определениями методов.

В основе объектно-ориентированного программирования лежит объектно-ориентированная декомпозиция. Разработка объектно-ориентированных программ заключается в построении иерархии классов, описывающих отношения между объектами, и в определении классов. Вычисления в объектно-ориентированной программе задаются сообщениями, передаваемыми от одного объекта к другому.

Объектно-ориентированная парадигма программирования является попыткой объединить лучшие свойства других вычислительных моделей. Наиболее полно объектно-ориентированная концепция реализована в языке Smalltalk. Поддержка объектно-ориентированной парадигмы в настоящее время включена в такие популярные императивные языки программирования, как Ada 2012, Java и C#.

Рассмотренная выше классификации языков программирования не единственная. Например, в отдельный класс по типу применения выделяют **сценарные** или **скри́птовые** языки**;**

Scripting language (**язык сценариев**) — высокоуровневый язык программирования для написания *сценариев* — кратких описаний выполняемых системой действий. Разница между программами и сценариями довольно размыта. Сценарий — это программа, имеющая дело с готовыми программными компонентами.

Сценарии обычно **интерпретируются**, а не **компилируются**, хотя сценарные языки программирования один за другим обзаводятся JIT-компиляторами. JIT-компиляция (*англ.* Just-in-time compilation, компиляция «на лету»), динамическая компиляция (*англ.* dynamic translation) — технология увеличения производительности программных систем, использующих *байт-*

код, путём компиляции байт-кода в *машинный код* непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом (сравнимая с компилируемыми языками) за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. JIT базируется на двух более ранних идеях, касающихся среды исполнения: компиляции байт-кода и динамической компиляции.

В более узком смысле под скриптовым языком может пониматься специализированный язык для расширения возможностей **командной оболочки** или текстового редактора и *средств администрирования* операционных систем.

1.2 Возможности формального описания языка программирования

Метаязык Хомского

Метаязык Хомского имеет следующую систему обозначений:

- символ “®” отделяет левую часть правила от правой (читается как "порождает" и "это есть");
- нетерминалы обозначаются буквой **A** с индексом, указывающим на его номер;
- терминалы - это символы используемые в описываемом языке;
- каждое правило определяет порождение одной новой цепочки, причем один и тот же нетерминал может встречаться в нескольких правилах слева.

Описание идентификатора на метаязыке Хомского будет выглядеть следующим образом:

1. A_1 ® A	23. A_1 ® W	45. A_2 ® s
2. A_1 ® B	24. A_1 ® X	46. A_2 ® t
3. A_1 ® C	25. A_1 ® Y	47. A_2 ® u
....	....	.....
20. A_1 ® T	42. A_2 ® p	64. A_3 ® A_3A_1
21. A_1 ® U	43. A_2 ® q	65. A_4 ® A_3A_2

Приведенный пример описания идентификатора показывает громоздкость метаязыка Хомского, что позволяет эффективно использовать его только для описания небольших абстрактных языков.

Метаязык Хомского-Щутценберже

Более компактное описание возможно с применением метаязыка Хомского-Щутценберже.

Метаязык Хомского-Щутценберже

- символ “=” отделяет левую часть правила от правой (вместо символа “®”);
- нетерминалы обозначаются буквой A с индексом, указывающим на его номер;
- терминалы - это символы используемые в описываемом языке;
- каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом “+”, что позволяет, при желании, использовать в левой части только разные нетерминалы.

Введение возможности альтернативного перечисления позволило сократить описание языков. Описание идентификатора будет выглядеть следующим образом:

1. A_1=A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P+Q+R+S+T+U+V+W+X+Y+Z+a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z
2. A_2=0+1+2+4+5+6+7+8+9
3. A_3=A_1+A_3A_1+A_3A_2 ...

Бэкуса-Наура формы (БНФ)

Метаязыки Хомского и Хомского-Щутценберже использовались в математической литературе при описании простых абстрактных языков.

Метаязык, предложенный **Джоном Бэкусом** и **Петером Науром (Backus J.W., Naur P.)** впервые разработан для описания синтаксиса языка программирования Алгола-60, однако, в дальнейшем он использован для многих других языков.

При записи грамматики в форме Бэкуса-Наура используются два типа объектов:

- основные символы (или терминальные символы, в частности, ключевые слова)

- основные символы (или терминальные символы, в частности, ключевые слова)

- металингвистические переменные (или нетерминальные символы), значениями которых являются цепочки основных символов описываемого языка. Металингвистические переменные изображаются словами (русскими или английскими), заключенными в угловые скобки (<...>)
- металингвистические связки (::=, |)

Наряду с новыми обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов.

Это сделало описание языка нагляднее и позволило в дальнейшем широко использовать данную нотацию для описания реальных языков программирования.

Были использованы следующие обозначения:

Бэкуса-Наура формы (БНФ)

- металингвистическая связка "::=" отделяет левую часть правила от правой;
- металингвистические переменные обозначаются произвольной символьной строкой, заключенной в угловые скобки "<" и ">";
- терминальные символы (терминалы) - это символы, используемые в описываемом языке, в частности, ключевые слова;
- каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга металингвистической связкой - символом вертикальной черты "|".

Пример описания идентификатора с использованием БНФ:

Правила описания идентификатора с использованием БНФ:

- <буква> :: = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
- <цифра> :: = 0|1|2|3|4|5|6|7|8|9
- <идентификатор> ::= <буква> | <идентификатор><буква> |<идентификатор><цифра>

Правила можно задавать и раздельно:

- <идентификатор> :: = <буква>
- <идентификатор> :: = <идентификатор> <буква>
- <идентификатор> :: = <идентификатор> <цифра>

Формы Бекуса-Наура используются с двумя основными целями:

- они являются металингвистическим языком, стандартным для описания языков программирования;
- они описывают правила построения текстов или конструкций.

Расширенные Бэкуса-Наура формы (РБНФ)

Метаязыки, представленные выше, позволяют описывать любой синтаксис.

Однако, для повышения удобства и компактности описания, целесообразно ввести в язык дополнительные конструкции.

ABNF (Augmented Backus-Naur Form, пополненная нормальная форма Бэкуса-Наура) - расширение нормальной формы Бэкуса-Наура (BNF) {а’гмент}

В частности, специальные метасимволы были разработаны для описания необязательных цепочек, повторяющихся цепочек, обязательных альтернативных цепочек. Существуют различные расширенные формы метаязыков, незначительно отличающиеся друг от друга.

Их разнообразие зачастую объясняется желанием разработчиков языков программирования по-своему описать создаваемый язык.

К примерам таких широко известных метаязыков можно отнести: метаязык PL/I, метаязык Вирта, используемый при описании Модулы-2, метаязык Кернигана-Ритчи, описывающий Си.

Зачастую описание таких языков называются расширенными формами Бэкуса-Наура (РБНФ).

РБНФ, используемые Виртом, имеют следующие особенности:

- Квадратные скобки "[" и "]" означают, что заключенная в них синтаксическая конструкция может отсутствовать;
- фигурные скобки "{" и "}" означают ее повторение (возможно, 0 раз);
- круглые скобки "(" и ")" используются для ограничения альтернативных конструкций;
- сочетание фигурных скобок и косой черты "{/" и "/}" используется для обозначения повторения один и более раз. Нетерминальные символы изображаются словами, выражающими их интуитивный смысл и написанными на русском языке.

1.3 Синтаксические диаграммы Вирта

Наряду с текстовыми способами описания синтаксиса языков широко используются и графические метаязыки, среди которых наиболее широкую известность получил язык диаграмм Вирта.

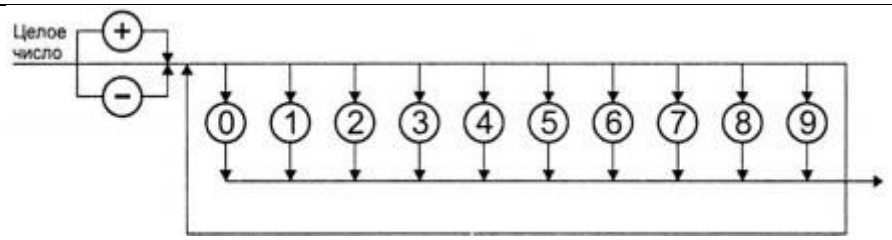
Синтаксические диаграммы были предложены Никлаусом Виртом для описания синтаксиса языка Pascal и являются удобной графической формой представления РБНФ.

Элементами синтаксических диаграмм Вирта являются: прямоугольники, кружки или овалы, стрелки. В прямоугольниках записываются имена металингвистических переменных, в кружках или овалах – основные символы языка, а стрелки определяют порядок сочетания металингвистических переменных и основных символов языка для образования определяемой синтаксической конструкции.

Каждой синтаксической конструкции соответствует одна диаграмма Вирта, в которой каждому сочетанию металингвистических переменных соответствует своя графическая диаграмма, на которой они посредством дуг. Альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием. Имя определяемой синтаксической конструкции записывается над стрелкой,

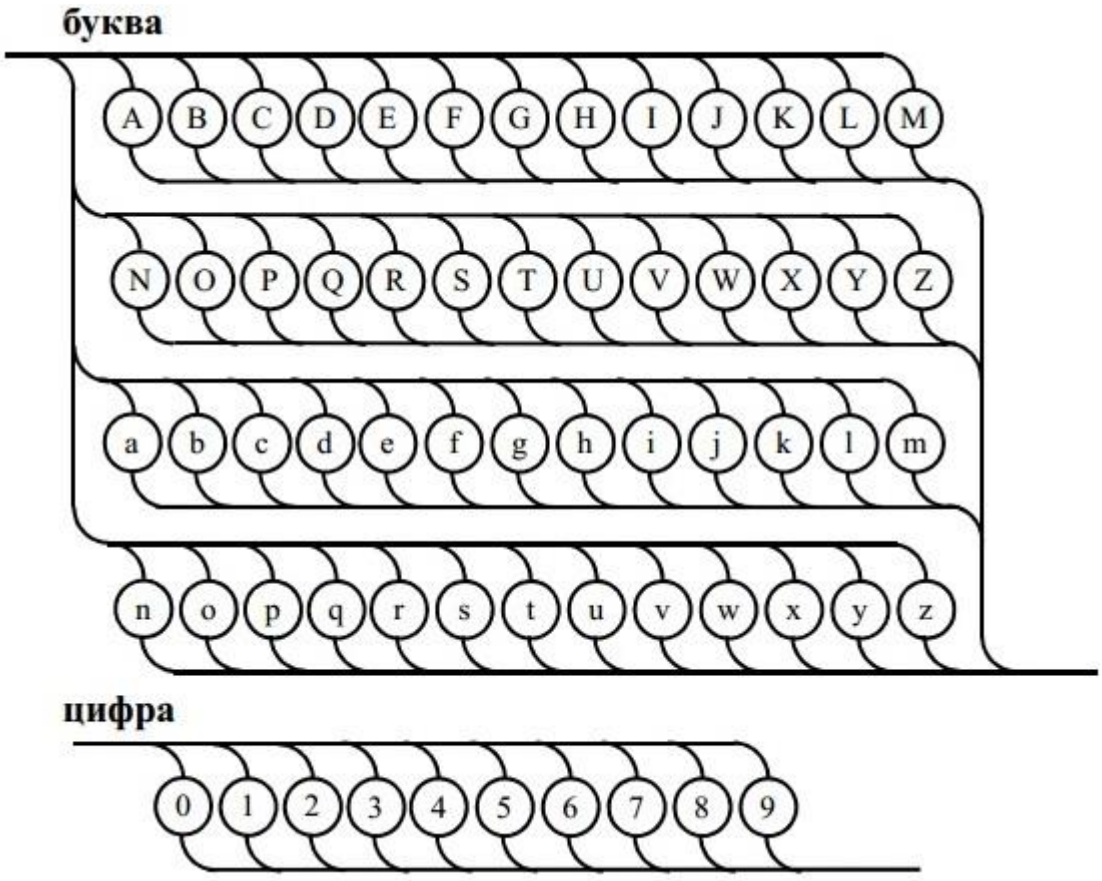
входящей в диаграмму (точка входа в диаграмму), которая, как правило, располагается в левом верхнем углу.

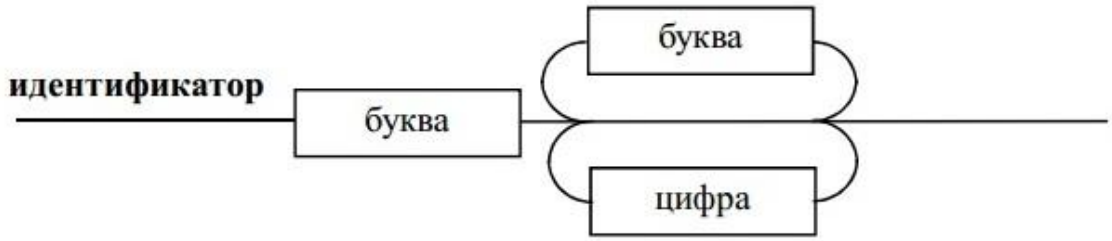
Любой путь от точки входа в синтаксическую диаграмму к выходу (исходящая из диаграммы стрелка) представляет собой цепочку металингвистических переменных и основных символов языка, соответствующую одному из вариантов правой части РБНФ. На рис. 1.2 приведены синтаксические диаграммы, определяющие множество целых чисел.



Синтаксические диаграммы для определения множества целых чисел

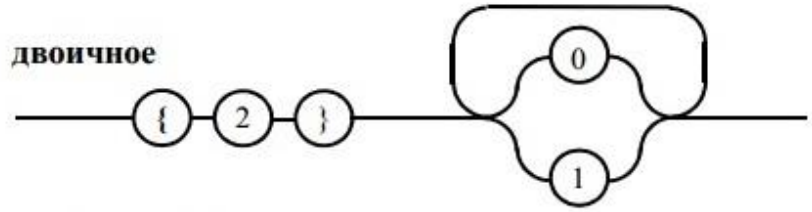
Элементарные конструкции. Использование диаграмм Вирта

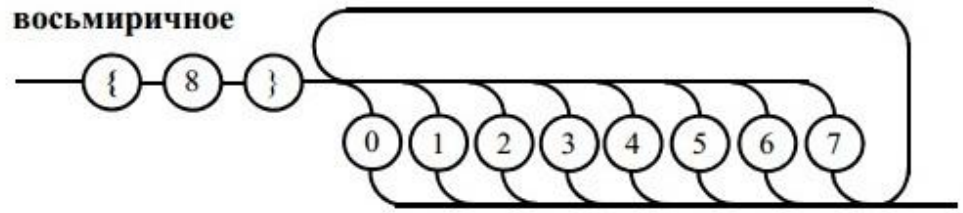


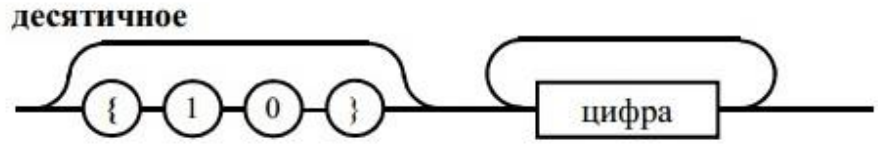

идентификатор
буква
буква
цифра

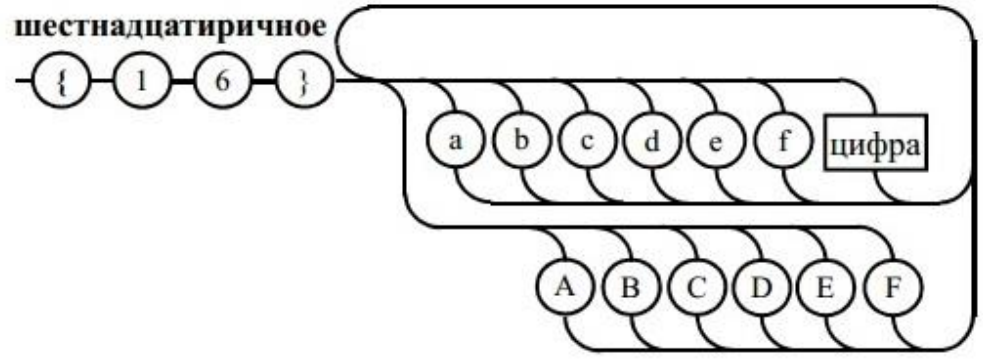

число
целое
действительное


целое
двоичное
восьмиричное
десятичное
шестнадцатиричное

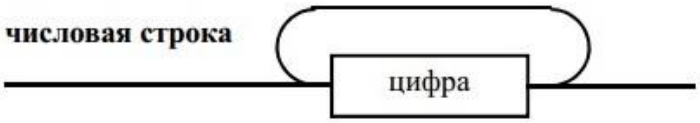

двоичное
{
2
}
0
1

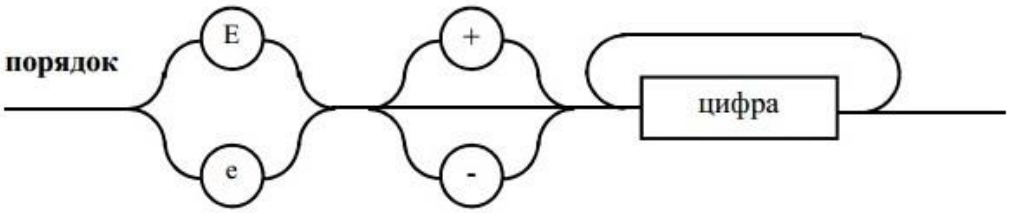

восьмиричное
{
8
}
0
1
2
3
4
5
6
7


десятичное
{
1
0
}
цифра

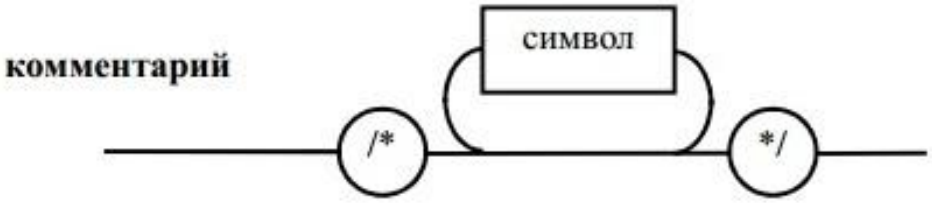

шестнадцатиричное
{
1
6
}
a
b
c
d
e
f
цифра
A
B
C
D
E
F

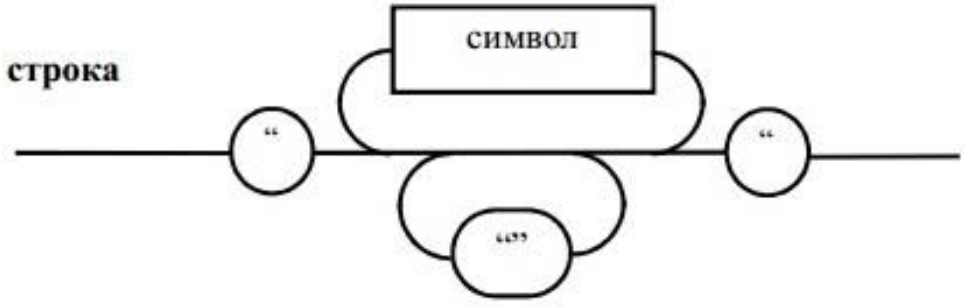

действительное
числовая строка
.
числовая строка
порядок
.

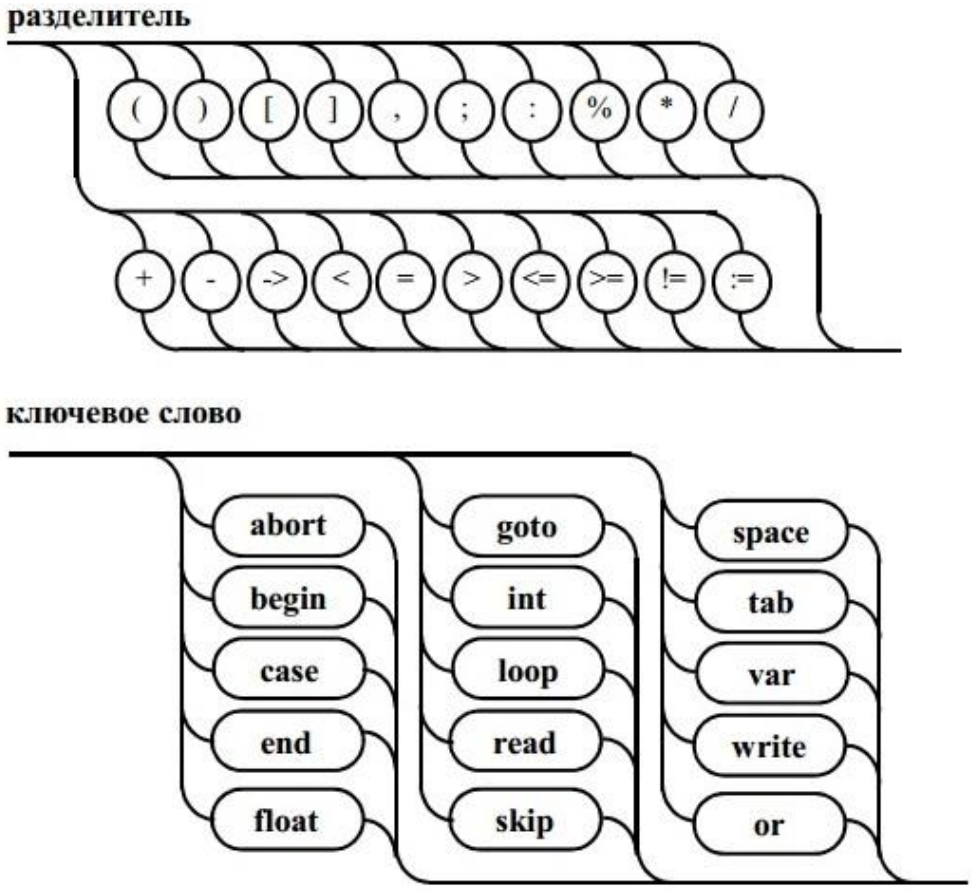

числовая строка
цифра


порядок
E
e
+
-
цифра


Пробельный символ
пробел
табуляция
перевод строки
перевод формата
комментарий


комментарий
символ
/*
*/


строка
символ
“
“
“”



БНФ, РБНФ и синтаксические диаграммы Вирта дают возможность косвенно включать в формальное описание синтаксиса языков программирования элементы семантики, т. к. в них входят металингвистические переменные, являющиеся осмысленными названиями описываемых конструкций. При программном анализе языков элементы семантики, заложенные в эти формальные модели, теряют смысл, поэтому в теории и практике проектирования языковых процессоров используются формальные грамматики.

Обычно стрелки на дугах диаграмм не ставятся, а направления связей отслеживаются движением от начальной дуги в соответствии с плавными изгибами промежуточных дуг и ветвлений. Таким же образом определяются входы и выходы терминалов и нетерминалов. Диаграммы Вирта позволяют задавать альтернативы, рекурсии, итерации и по изобразительной мощности эквивалентны РБНФ. Но графическое отображение правил более наглядно. Кроме этого допускается произвольное проведение дуг, что уменьшает количество элементов в правиле за счет его неструктурированности. Диаграммы Вирта являются удобным исходным документом для построения лексического и синтаксического анализаторов.

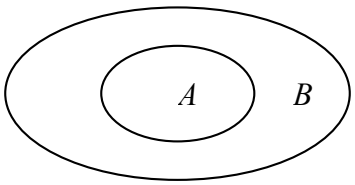
Множества и операции с ними

сли А - множество, то его *элементы* – это есть объекты a, для которых $a \in A$. Знак $\in$ означает принадлежность множеству А. Итак, $A=\{a_1, a_2, \ldots a_n, \}$ – множество, $a_i \in A$ *элемент* множество. *Отрицание этого утверждения записывается* $a \notin A$. *Если* А содержит конечное число элементов, то А называется *конечным* множеством. Символ $\varnothing$ обозначает пустое множество, т.е. множество, в котором нет элементов.

Один из способов определения множества – определение с помощью *предиката*. Предикат – это утверждение, состоящее из нескольких переменных и принимающее значение 0 или 1 («*ложь*» или «*истина*»). Множество, определяемое с помощью предиката, состоит из тех элементов, для которых предикат истинен.

Говорят, что множество А *содержится* во множестве В, и пишут $A \in B$, если каждый элемент из А является элементом из В. Если В содержит элемент, не принадлежащий А и $A \subseteq B$, говорят, что А, *собственно* содержится в В (диаграмма Венна для включения множеств).

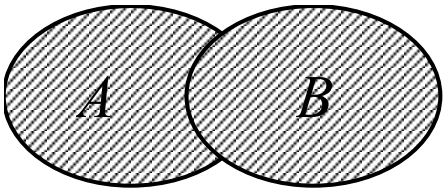
Диаграмма Венна



Объединение множеств

$A \cup B = \{x \mid x \in A \text{ или } x \in B\}$ -

это множество, содержащее все элементы А и В (Диаграмма Венна для объединения множеств):



Пересечение множеств

$A \cap B = \{x \mid x \in A \text{ и } x \in B\}$ -

это множество, состоящее из всех тех элементов, которые принадлежат обоим множествам *A* и *B* (рис. 1.5).

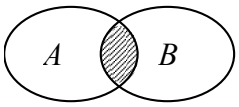


Рис. 1.5. Диаграмма Венна для пересечений множеств

Разность множеств

A – B -

это множество элементов A, не принадлежащих B (рис. 1.6).

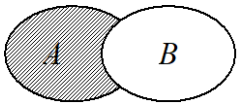


Рис. 1.6. Диаграмма Венна для разности множеств

Декартово произведение A и *B*

$A \times B = \{(a,b) \mid a \in A \text{ и } b \in B\}$

Пример. Если $A=\{1,2\}$, $B=\{2,3,4\}$, то $A \times B=\{(1,2), (1,3), (1,4), (2,2), (2,3), (2,4)\}$

Отношения

Пусть *A* и *B* – множества. **Отношением из *A* в *B* называется любое подмножество множеств *A*×*B*.**

Если *A=B*, то отношение задано (определено) на *A*.

Если *R* отношение из *A* в *B* и $(a, b) \in R$, то пишут *aRb*.

Множество *A* называют областью определения, *B* - множеством значений.

Пример.

A – множество целых чисел. Отношение *L* представляет множество $\{(a, b) \mid a < b\}$ *aLb.*

Определение.

Отношение $\{(b, a) \mid (a, b) \in R\}$ называют обратным к отношению *R*, т.е. R^{-1}.

Определение.

Пусть *A* – множество, *R* – отношение на *A*.

Тогда *R* называют:

1) *рефлексивным*, если *aRa* для всех пар из *A;*
2) *симметричным*, если *aRb* влечет *bRa* для всех *a* и *b* из *A;*
3) *транзитивным*, если *aRb* и *bRc* влекут *aRc* для *a, b, c* из *A.*

Рефлексивное, симметричное и транзитивное отношение называют отношением *эквивалентности*.

Отношение эквивалентности, определенное на A, заключается в том, что оно разбивает множество A на непересекающиеся подмножества, называемые классами эквивалентности.

Пример.

Рассмотрим отношение сравнения по модулю *N*, определенное на множестве неотрицательных чисел.

a сравнимо с *b* по модулю *N*

a=b(modN), т.е. $a-b=kN$ (*k* - целое).

Пусть *N*=3, тогда множество {0, 3, 6,…, 3*n*,…} будет одним из классов эквивалентности, т.к. $3n=3m(mod3)$ для целых *n* и *m*. Обозначим его через [0].

[0]={0, 3, 6,…, 3*n*,…}

Другие два:

[1]={1, 4, 7,…, 3*n*+1,…};

[2]={2, 5, 8,…, 3*n*+2,…}.

Объединение этих трех множеств дает множество неотрицательных целых чисел (рис. 1.7).

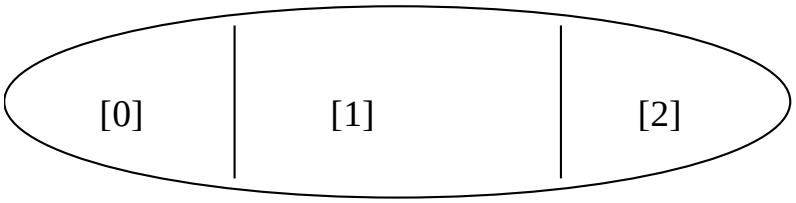


Рис. 1.7. Классы эквивалентности отношения сравнения по модулю 3

Число классов, на которые разбивается множество отношением эквивалентности, называется индексом этого отношения.

Замыкание отношений

Задача.

Для данного отношения *R* найти другое *R′,* обладающее дополнительными свойствами (например, транзитивностью). Более того, желательно, чтобы *R′* было как можно «меньше», т.е., чтобы оно было подмножеством *R*.

Задача в общем случае не разрешима, однако для частных случаев решение имеется.

Определение.

k – степень отношения *R* на *A* (R^k) определяется:

1) aR^1b тогда и только тогда, когда *aRb;*

2) aR^ib для $i>1$ тогда и только тогда, когда существует такое $c \in A$, что *aRc* и cR^{i-1}

Транзитивное замыкание отношения множества *R* на *A* (R^+) определяется так: aR^+b тогда и только тогда, когда aR^ib для некоторого $i \geq 1$.

Расшифровка понятия:

aR^+b, если существует последовательность c_1, c_2,…, c_n, состоящая из 0 или более элементов принадлежащих *A*, такая, что aRc_1, aRc_2,… aRc_{n-1}, aRc_n, c_nRb. Если $n=0$, то *aRb.*

Рефлексивное и транзитивное замыкание отношения *R* (R^*) на множестве *A* определяется следующим образом:

1) aR^*a для всех $a \in A$;

2) aR^*b, если aR^+b;

3) в R^* нет ничего другого, кроме того, что содержится в 1) и 2).

Если определить R^0, сказав, что aR^0b тогда и только тогда, когда $a=b$, то aR^*b тогда и только тогда, когда aR^ib для некоторого $i \geq 0$.

Единственное различие между R^+ и R^* состоит в том, что aR^*a истинно для всех $a \in A$, но aR^+a может быть, а может и не быть истинным.

Отношения порядка

Отношения порядка играют важную роль при изучении алгоритмов, особенно специальный вид порядков – частичный порядок.

Определение.

Частичным порядком на множестве *A* называют отношение *R* на *A* такое, что:

1) *R* – транзитивно;

2) для всех $a \in A$ утверждение *aRa* ложно, т.е. отношение *R* иррефлексивно.

Пример.

S= {e_1, e_2, …, e_n}, - множество, состоящее из *n* элементов, и пусть *A=P*(*S*). Положим *aRb* для любых *a* и *b* из *A* тогда и только тогда, когда $a \not\subset b$. Отношение *R* называется частичным порядком.

Для случая *S*={0, 1, 2} имеем (рис. 1.8).

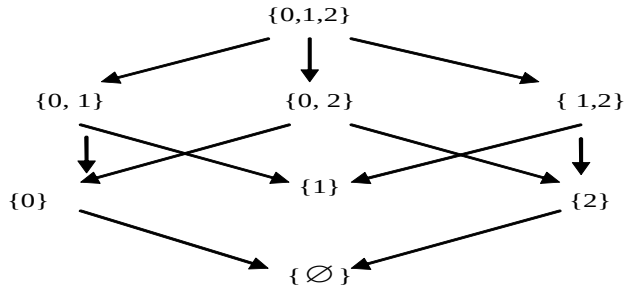


Рис. 1.8. Частичный порядок

Определение.

Рефлексивным частичным порядком называется отношение *R,* когда

1) *R* – транзитивно;
2) *R* – рефлексивно;
3) если *aRb*, то *a=b.*

Последнее свойство называется *антисимметричностью.*

Каждый частичный порядок можно графически представить в виде ориентированного ациклического графа.

Линейный порядок *R* на множестве *A* – это такой частичный порядок, что, если *a* и $b \in A$, то либо *aRb*, либо *bRa*, либо *a=b*. Удобно это понять из следующего.

Пусть *A* представлено в виде последовательности $a_1, a_2, \ldots, a_n$, для которых $a_i R a_j$ тогда и только тогда, когда $i<j$.

Аналогично определяется рефлексивный линейный порядок.

Из традиционных систем отношение < (меньше) на множестве неотрицательных целых чисел – это линейный порядок, отношение ≤ - рефлексивный линейный порядок.

Раздел 2. Общая модель компилятора. Фазы компиляции.

2.1 Общая модель компилятора

2.1.1 Основные задачи компиляторов

Трансляторы бывают двух типов: **компиляторы (compiler) и интерпретаторы (interpreter)**. **Компилятор** (от англ. Compile - собирать вместе, составлять) - системная программа, выполняющая преобразование программы, написанной на одном алгоритмическом языке, в программу на языке, близком к машинному, и в определенном смысле эквивалентную первой.

Процесс компиляции можно представить как **анализ (analysis) и синтез (synthesis).** Анализирующая часть компилятора разбивает исходную программу на составляющие ее элементы (конструкции языка) и создает промежуточное представление исходной программы. Синтезирующая часть из промежуточного представления создает новую программу, которую компьютер в состоянии понять. Такая программа называется объектной программой. Объектная программа может в дальнейшем выполняться без перетрансляции.

В качестве промежуточного представления часто используются деревья, например, так называемые деревья разбора. В дереве разбора каждый узел соответствует некоторой операции, а ветви этого узла – операндам.

В том случае, если исходный язык достаточно прост (например, языки Basic, FoxPro), то никакое промежуточное представление не нужно, и тогда вместо компиляции используется интерпретация – это простой цикл обработки конструкций языка. **Интерпретатор** 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.

2.1.2. Компилятор

Первые компиляторы появились в начале 1950-х гг. Сегодня сложно определить, когда появился первый компилятор, поскольку в те годы проводилось множество экспериментов и разработок различными независимыми группами. В основном, целью разработки первых компиляторов было преобразование в машинный код арифметических формул.

Годом рождения теории компиляторов можно считать 1957, когда появился первый компилятор языка Фортран, созданный Бэкусом и дающий

достаточно эффективный объектный код. Он работал на платформах IBM 7040, IBM 360 и DEC PDP-11. В 1980 г. была разработана новая версия для IBM 360 и IBM PC, которая поддерживала стандарт FORTRAN 77, на сегодня под Win7 работает Silverfrost FTN95. Через год была образована фирма Watcom, которая в 1988 г. представила компилятор C. Он сразу получил широкую популярность среди программистов, так как генерировал самый быстрый код среди компиляторов того времени.

Джон Бэкус (англ. *John Backus)*

Компилятор переводит программы с одного языка на другой. Входом компилятора служит цепочка символов, составляющая исходную программу на языке программирования L_1. Выход компилятора (объектная программа) также представляет собой цепочку символов, но принадлежащую другому языку L_2, например, языку некоторого компьютера. При этом сам компилятор написан на языке L_3, возможно, отличающемся от первых двух. Будем называть язык L_1 исходным языком, язык L_2 – целевым языком, а язык L_3 – языком реализации. Таким образом, можно говорить о компиляторе K как об отображении множества L_1 в множество L_2, т.е. $L_1 \Rightarrow_K L_2$ (см. рис. 2.1).

Программа на входном языке - цепочка символов, составляющая исходную программу на языке **программирования L_1.**

Объектная программа – код программы на **целевом языке L_2.**

Компилятор K отображает множество L_1 в множество L_2, используя язык реализации L_3 – т.е. $L_1 \Rightarrow_K L_2$

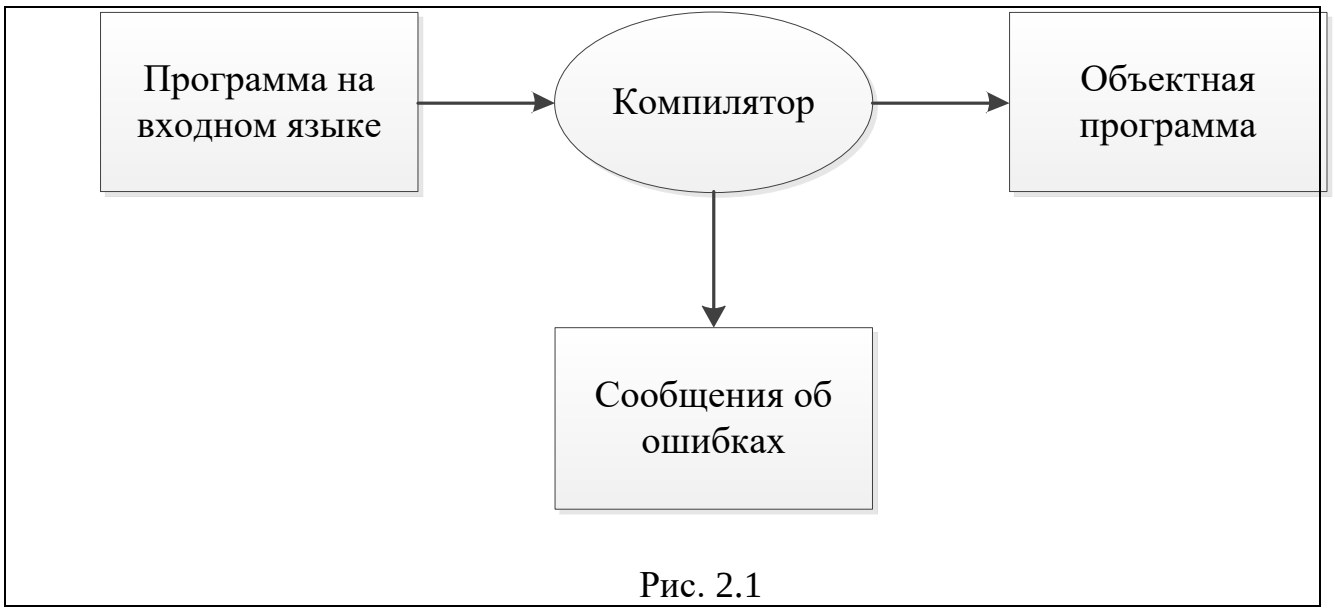


Рис. 2.1

Отметим, что 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 важной частью процесса трансляции является точная диагностика ошибок, допущенных во входной программе.

Я уже отмечал, что существует великое множество языков программирования, начиная с таких традиционных языков программирования как C и Pascal и заканчивая клонами современных объектно-ориентированных языков C# и Java. Практически каждый язык программирования имеет какие-то особенности с точки зрения создателя транслятора.

Важной исторической особенностью компилятора являлось то, что он мог производить и компоновку (то есть содержал две части - транслятор и компоновщик). Это связано с тем, что раздельная компиляция и компоновка как отдельная стадия сборки выделились значительно позже появления компиляторов, и многие популярные компиляторы (например, GCC- основной компилятор проекта GNU) до сих пор физически объединены со своими компоновщиками. В связи с этим, вместо термина "компилятор" иногда используют термин "транслятор" как его синоним: либо в старой литературе, либо когда хотят подчеркнуть его способность переводить программу в машинный код (и наоборот, используют термин "компилятор" для подчеркивания способности собирать из многих файлов один).

Т.о. большая часть компиляторов переводят программу с некоторого высокоуровневого языка программирования в машинный код, который может быть непосредственно выполнен центральным процессором. Как правило, этот код также должен выполняться в среде конкретной операционной системы, поскольку использует предоставляемые ей возможности (системные вызовы, библиотеки функций). Архитектура (набор программно-аппаратных средств), для которой производится компиляция, называется **целевой машиной**.

Одной из составных частей компилятра может быть препроцессор. Его задача – обработать как встроенные в язык, так и пользовательские макросы в

2.1.3. 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 (см. рис. 2.2).

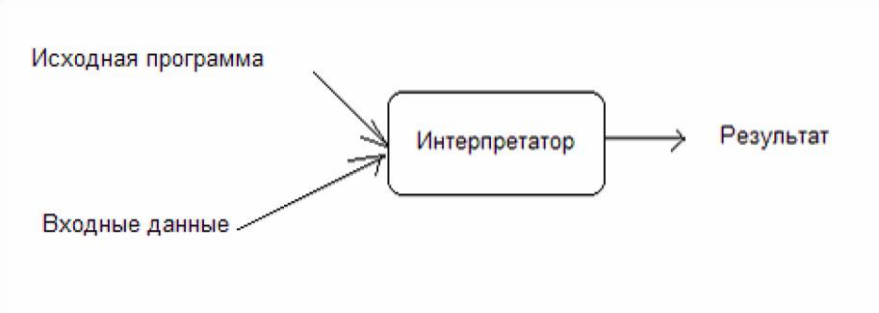


Рис. 2.2

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), 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.

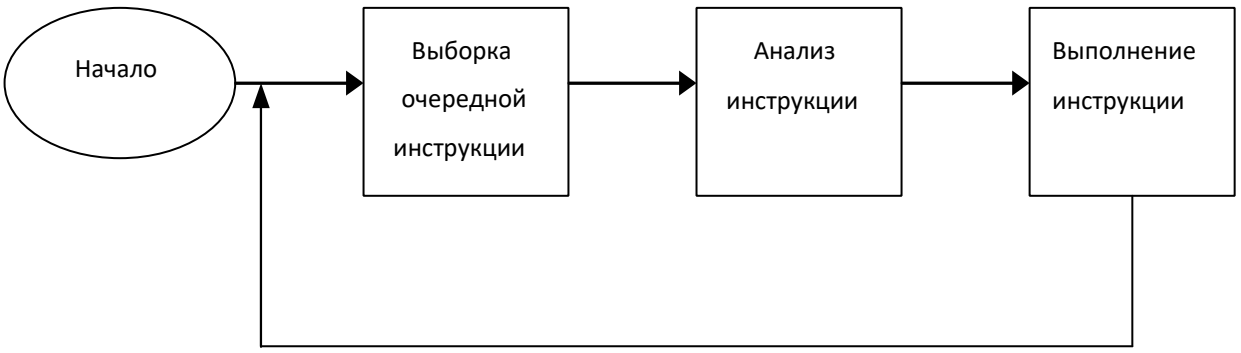


Рис. 2.3

Понятно, что при повторном запуске программы она должна интерпретироваться с самого начала.

Интерпретация позволяет выполнить более гибкую и лучшую диагностику ошибок, чем компиляция. Поскольку исходная программа исполняется непосредственно, интерпретатор может включать хороший *отладчик* (*debugger*). Кроме того, интерпретатор может легко справиться с языками, позволяющими 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.

В современных трансляторах часто используются как элементы компиляции, так и интерпретации.

2.1.4. Объектная программа

Объектная 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 (создание единого перемещаемого объектного сегмента из набора различных сегментов) осуществляется программой, которая называется *редактором связей*. Далее единый перемещаемый объектный сегмент или модуль загрузки размещается в памяти программой, называемой загрузчиком, которая превращает перемещаемые адреса в абсолютные. После этого программа готова к выполнению (см. рис. 2.4).

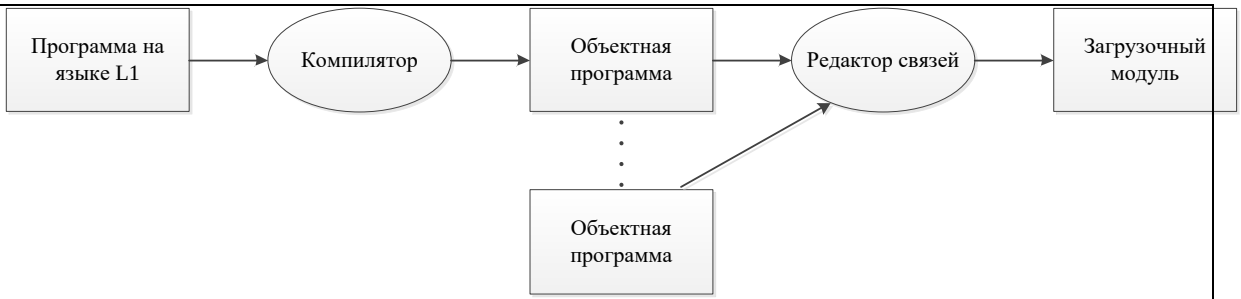


Рис. 2.4

2.1.5. 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 (см. рис. 2.5).

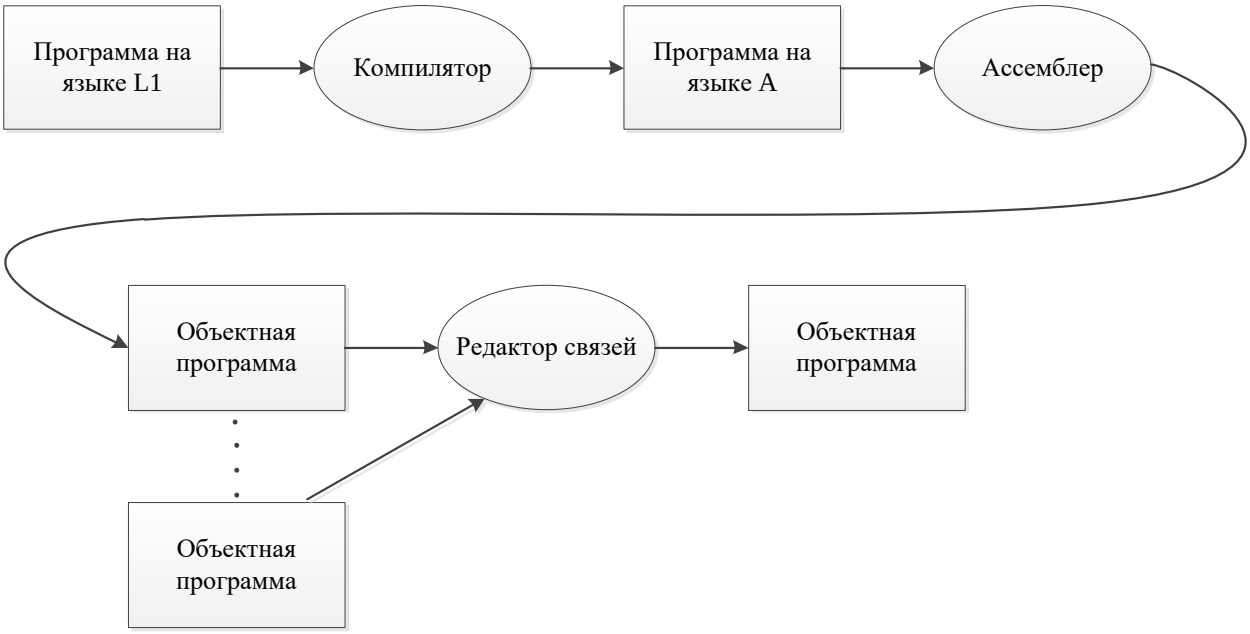


Рис. 2.5

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 используется виртуальная машина MSIL, представляет собой высокоуровневый ассемблер, максимально абстрагированный от конкретных целевых платформ.

2.1.6. Кросс-транслятор

Кросс-транслятор - это вид транслятора, который переводит программу, записанную в нотации одного языка программирования и выполняющуюся в одной инструментальной среде, на одной компьютере, который характеризуется своим операционным окружением и/или архитектурой, в код вычислительной системы другой среды другого компьютера.

Та вычислительная система, для которой генерируется код, называется объектной ЭВМ и соответственно полученный код называется объектным кодом (не путать с объектным модулем).

Например, для бортового компьютера, который управляет полетом беспилотника, крылатой ракеты совершенно не нужно все то, что обеспечивает работу программиста по разработке программ для него. Ему не нужны средства редактирования текста, у него совершенно другие функции: контроль полета и т.п. На таком компьютере будет работать какая система? Ответ: система реального времени. Так вот, для создания программ для таких компьютеров используются кросс-трансляторы и системы кросс-программирования.

Кросс-транслятор:

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

Для каждой целевой машины (IBM, Apple и т.д.) и каждой операционной системы или семейства операционных систем, работающих на целевой машине, требуется написание своего компилятора. Существуют также так называемые кросс-компиляторы, позволяющие на одной машине и в среде одной ОС получать код, предназначенный для выполнения на другой целевой машине и/или в среде другой ОС. Кроме того, компиляторы могут быть оптимизированы под разные типы процессоров из одного семейства (путём использования специфичных для этих процессоров инструкций). Например, код, скомпилированный под процессоры семейства i686, может использовать специфичные для этих процессоров наборы инструкций - MMX, SSE, SSE2.

В общем случае рассмотрим вариант, когда у нас есть два компьютера: компьютер *M* с языком ассемблера *A* и компьютер *K* с языком ассемблера *A1*. В наличии имеется компилятор P языка ассемблер для компьютера *K* ($P \Rightarrow_K A1$), а

сам компьютер *M* по каким-то причинам не доступен либо пока еще не существует компилятор P языка ассемблер для компьютера *M* ($P \Rightarrow_M A$). Нам необходимо создать компилятор *L* для компьютера *M* ($L \Rightarrow_K A$). В такой ситуации мы можем использовать *K* в качестве *инструментальной* машины и написать компилятор *L* для компьютера *M* ($L \Rightarrow_K A$), который принято называть *кросс-транслятором* (*cross-compiler*). На основе его можно разрабатывать программы, вплоть до операционной системы для целевого компьютера *M*. Как только машина *M* станет доступной, он и разработанное на его основе ПО можно перенести на *M*. Понятно, что это решение достаточно трудоемко, поскольку могут возникнуть проблемы при переносе, например, из-за различий операционных систем.

Существуют программы, которые решают обратную задачу - перевод программы с низкоуровневого языка на высокоуровневый. Этот процесс называют декомпиляцией, а программы - декомпиляторами. Но, поскольку компиляция - это процесс с потерями, точно восстановить исходный код, скажем, на C++ в общем случае невозможно. Более эффективно декомпилируются программы в байт-кодах.

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)*.

2.1.7. 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 (см. рис. 2.6). При выполнении этих условий можно достаточно быстро написать интерпретатор виртуальной шины. можно достаточно быстро написать интерпретатор виртуальной машины.

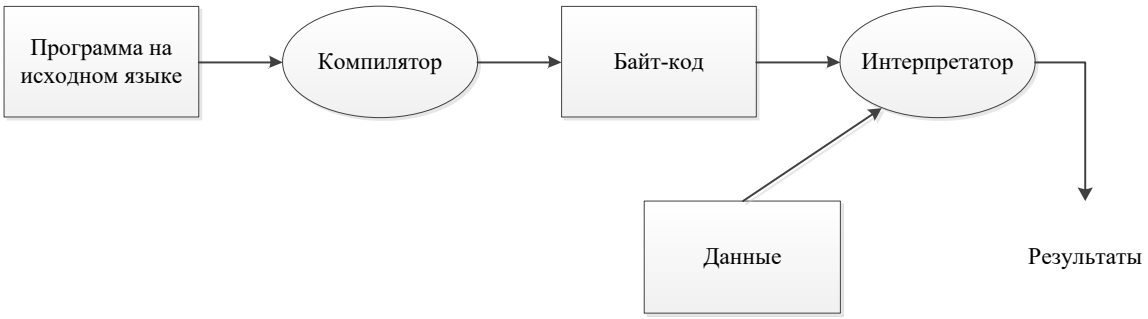


Рис. 2.6

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.

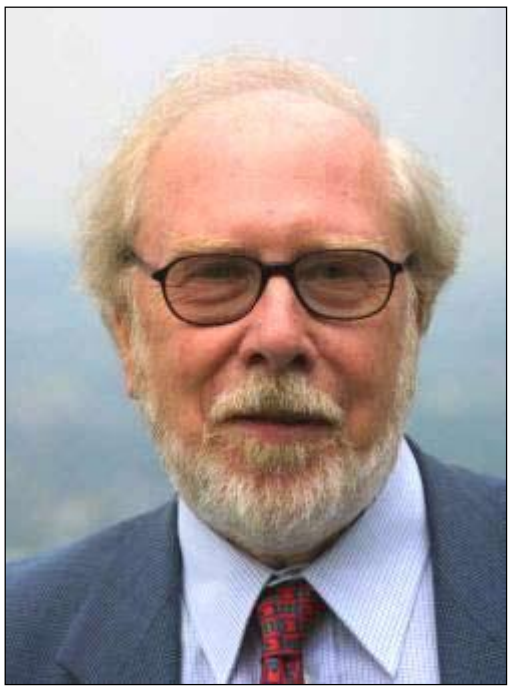


Рис. 2.7 Никлаус Вирт (нем. *Niklaus Wirth)*

Никлаус Вирт - известнейший теоретик в области разработки языков программирования, профессор Швейцарской высшей технической школы Цюриха, создал языков программирования Паскаль, Модула-2, Оберон (совм. с Юргом Гуткнехтом). Вирт часто критикует «американский подход» к разработке средств программирования, в котором маркетинговые соображения превалируют над требованиями математической стройности и гарантированной надёжности. В частности, говоря об **ООП**, Вирт неоднократно отмечал, что оно является достаточно тривиальным расширением того же структурного подхода,

сдобренным новой терминологией, и вряд ли может претендовать на звание «революционной методологии программирования».

2.1.7. 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, именно так и устроена платформа .NET (см. рис. 2.8).

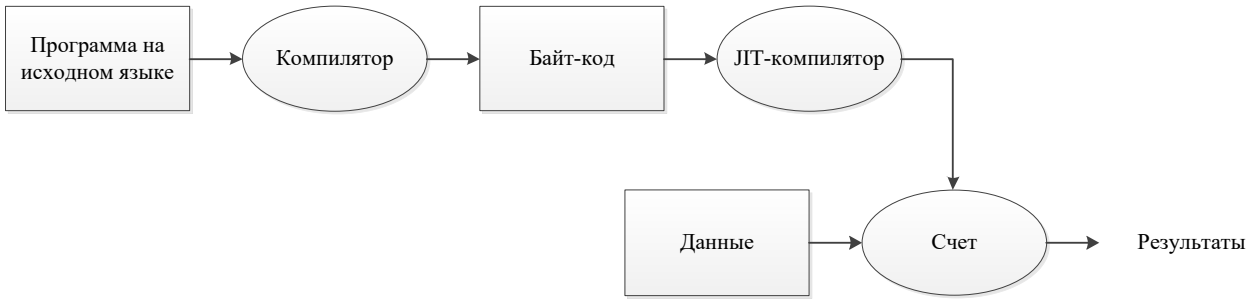


Рис. 2.8

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.

Таким образом компилятор Java переводит программу не в машинный код, а в программу на некотором специально созданном низкоуровневом языке. Такой язык - байт-код - также можно считать языком машинных команд, поскольку он подлежит интерпретации виртуальной машиной. Например, для

языка Java это **Java Virtual Machine** (сокращенно **Java VM**, **JVM**) — виртуальная машина Java — основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). JRE исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java. JVM (язык виртуальной машины Java), или так называемый байт-код Java (вслед за ним все промежуточные низкоуровневые языки стали называть байт-кодами). Для языков программирования на платформе .NET Framework (C#, C++, VisualBasic.NET и другие) это MSIL (Microsoft Intermediate Language, "Промежуточный язык фирмы Майкрософт").

Программа на байт-коде подлежит интерпретации виртуальной машиной, либо ещё одной компиляции уже в машинный код непосредственно перед исполнением. MSIL-код компилируется в код целевой машины также Just-In-Time-компилятором, а библиотеки .NET Framework компилируются заранее.

2.2 Фазы компиляции

Общие свойства и закономерности присущи как различным языкам программирования, так и трансляторам с этих языков. В них протекают схожие процессы преобразования исходного текста. Несмотря на то, что взаимодействие этих процессов может быть организовано различным путем, можно выделить функции, выполнение которых приводит к одинаковым результатам. Назовем такие функции фазами процесса трансляции (compilation phases).

Т.о. процесс создания компилятора можно свести к решению нескольких задач, которые распределяются между фазами компиляции. Обычно компилятор состоит из следующих фаз:

- лексический анализ
- синтаксический анализ
- семантический (видозависимый) анализ
- оптимизация
- генерация кода.

Дополнительно могут быть использованы фазы перевода в промежуточное представление, семантического анализа компонент промежуточного представления, анализа корректности и оптимизации промежуточного представления.

Интерпретатор отличается тем, что фаза генерации кода обычно заменяется фазой эмуляции элементов промежуточного представления или объектной модели языка. Кроме того, в интерпретаторе обычно не проводится оптимизация промежуточного представления, а сразу же осуществляется его эмуляция.

Рассмотрим основные цели каждой из фаз компиляции.

Мы продемонстрируем преобразования, которым подвергается исходная программа на перечисленных фазах компиляции, на небольшом примере оператора присваивания

*position = position + rate * 60*,

причем предположим, что все переменные вещественные.

2.2.1 Лексический анализ

Лексический анализ. Лексический анализатор выполняет распознавание лексем языка и замену их соответствующими кодами.

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 разборе входной цепочки и выделении некоторых более "крупных" единиц, *лексем* (токенов[1]), которые удобнее для последующего разбора.

Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами. Для каждой лексемы анализатор строит выходной токен (token) вида:

<имя_токена, значение_атрибута>

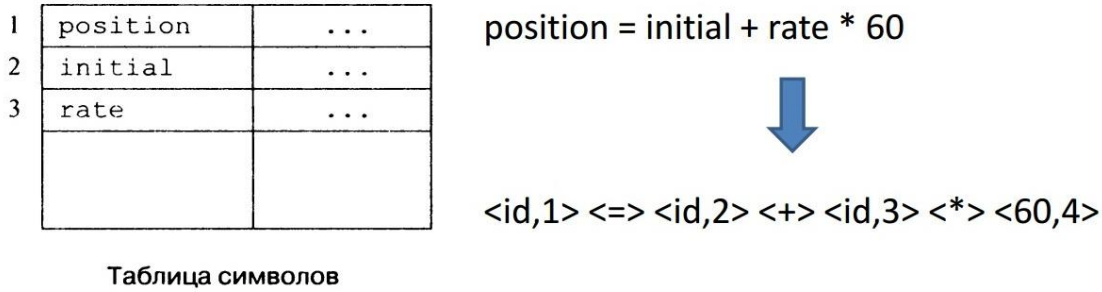


Рис. 2.9 Создание токенов и таблицы лексем

Анализатор использует первые компоненты токенов, полученных при лексическом анализе, для создания синтаксического дерева, которое описывает грамматическую структуру потока токенов.

[1] Токен в лексическом анализе - это объект, создающийся из лексемы в процессе лексического разбора Часто понятия токен и лексема полагают одинаковыми (Токен - частный денежный знак, как правило, металлический)

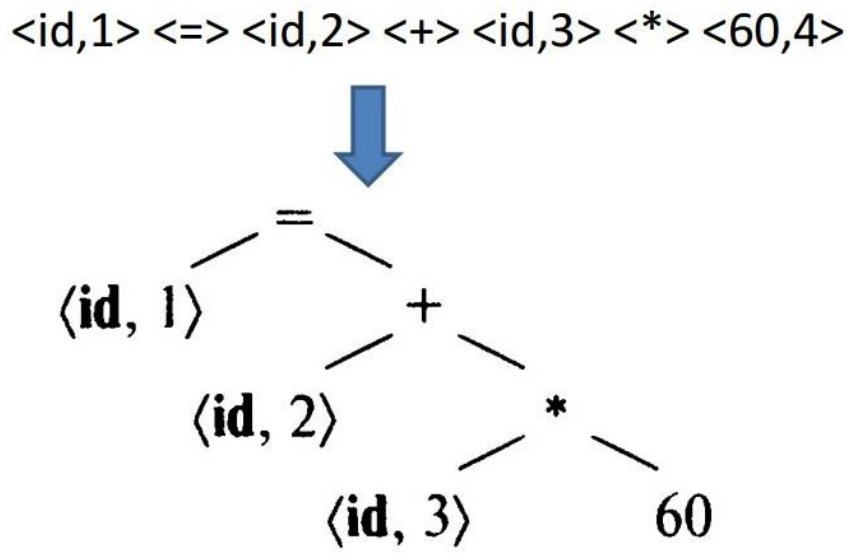


Рис. 2.10 Синтаксическое дерево грамматической структуры потока токенов

Лексический анализатор выполняет распознавание лексем языка и замену их соответствующими кодами.

Входом компилятора служит программа на исходном языке программирования. С точки зрения компилятора это просто последовательность символов. Задача первой фазы компиляции, *лексического анализатора (lexical analysis)*, {лэ'ксикол эна'лэзиз} заключается в разборе входной цепочки и выделении некоторых более "крупных" единиц, *лексем* (токенов), которые удобнее для последующего разбора (см. рис. 2.9). Под лексемами понимаются элементарные единицы, входящие в структуру предложения языка, такие как ключевые слова, идентификаторы, константные значения (числа, строки, логические) и т.п. Правильность задания структуры предложения языка на фазе лексического анализа не выполняется. Результатом является поток лексем (кодов - ссылок на таблицы), эквивалентный исходному тексту.

Рис. 2.11

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 (это может быть, например, номер ключевого слова согласно внутренней нумерации компилятора), 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 анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами. Для каждой лексемы анализатор строит выходной токен (token) вида:

<имя_токена, значение_атрибута>

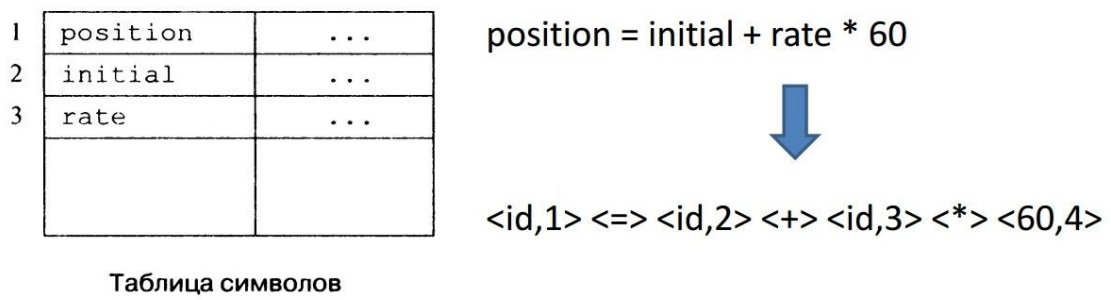


Рис. 2.12 Создание токенов и таблицы лексем

2.2.2 Синтаксический анализ

Синтаксический анализатор (syntax analyzer, parser) необходим для того, чтобы выяснить, удовлетворяют ли предложения, из которых состоит исходная программа, правилам грамматики этого языка. Он получает на вход результат работы лексического анализатора и разбирает его в соответствии с некоторой грамматикой. Эта грамматика аналогична грамматике, используемой при описании входного языка. Грамматики могут использоваться как для порождения так и для распознавания предложений языка. Порождение начинается с начального понятия (или аксиомы грамматики). При распознавании с помощью грамматических правил порождается предложение, которое затем сравнивается с входной строкой. При этом применение правил подстановки для порождения очередного символа предложения зависит от результатов сравнения предыдущих символов с соответствующими символами входной строки. Однако грамматика входного языка обычно не уточняет, какие конструкции следует считать лексемами (см. рис. 2.13).

Синтаксический анализ является одной из наиболее формализованных и хорошо изученных фаз компиляции. Методы построения синтаксических анализаторов будут рассмотрены в дальнейшем.

После синтаксического анализа можно считать, что исходная программа преобразована в некоторое промежуточное представление. Результат анализа исходного предложения в терминах грамматических конструкций удобно представлять в виде дерева. Поэтому первые компоненты токенов, полученных при лексическом анализе, преобразуются дерево грамматического разбора (синтаксическое дерево), которое описывает грамматическую структуру потока токенов.

В дереве разбора программы внутренние узлы соответствуют операциям, а листья представляют операнды.

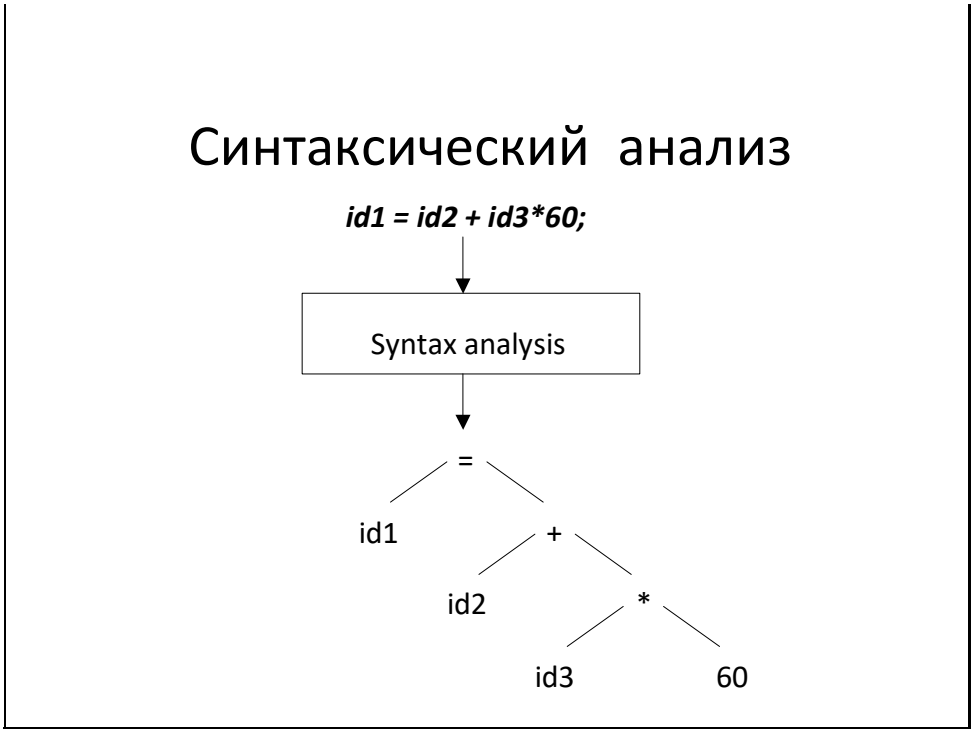


Рис. 2.13

2.2.3. Семантический (видозависимый) анализ

Слайд 14

Видозависимый анализ (type checking), также называемый *семантическим анализом (semantic analysis)*, {семэ'нтик энлалэ'зиз} обычно заключается в проверке правильности типа и вида всех идентификаторов и данных, используемых в программе (см. рис. 2.14).

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице идентификаторов для последующего использования в процессе генерации промежуточного кода.

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.

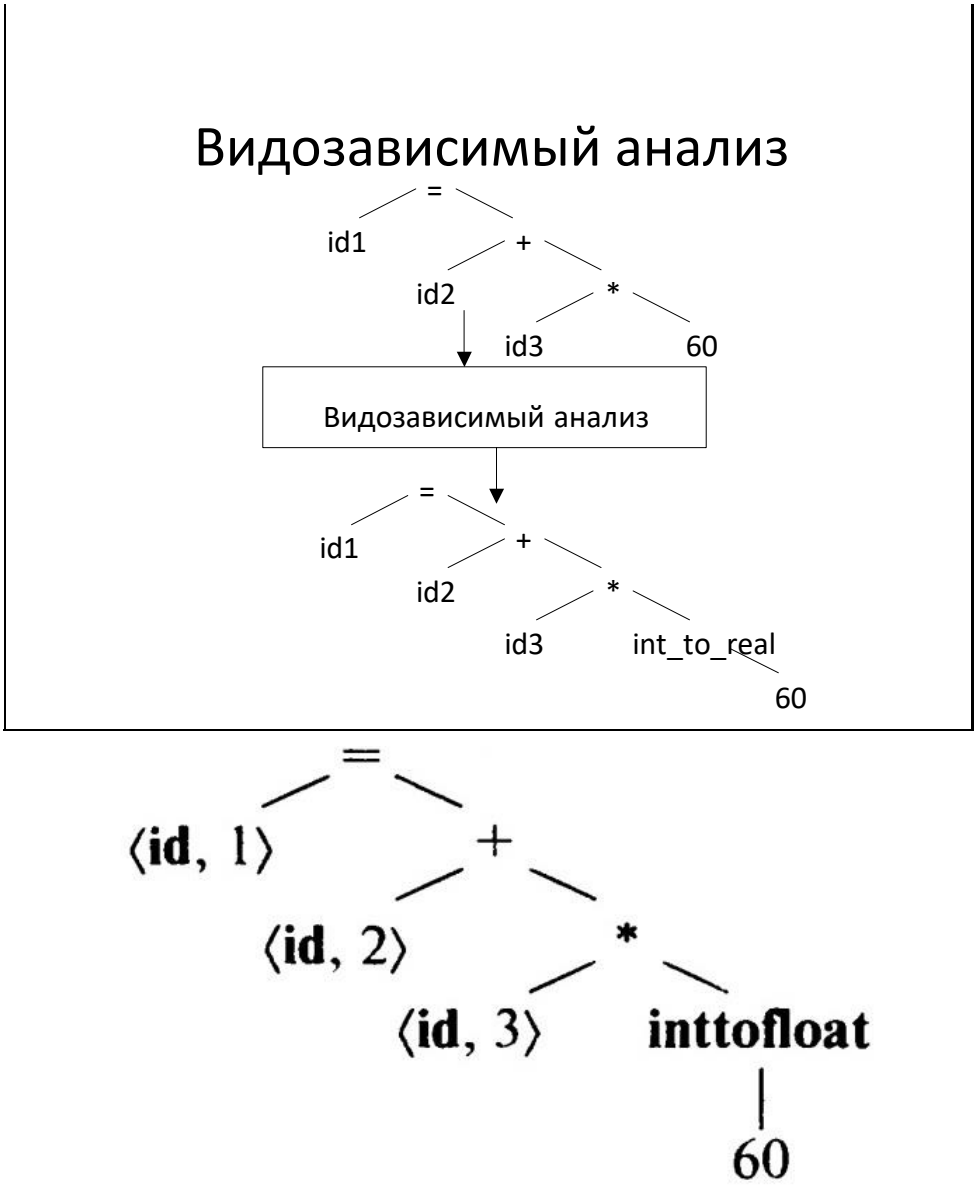


Рис. 2.14

Такие контекстные условия не всегда могут быть проверены во время синтаксического анализа и потому обычно выделяются в отдельную фазу.

2.2.4. Оптимизация кода

Основная цель *фазы оптимизации (code optimization)* заключается в преобразовании промежуточного представления программы в целях повышения эффективности результирующей объектной программы (см. рис. 2.15).

Решаются проблемы уменьшения избыточности программы по затратам времени и памяти. В зависимости от критериев проектирования транслятора данная фаза обработки программы может исключаться из цикла обработки программы. При оптимизации происходит преобразование исходной программы в промежуточную (например, польскую) форму записи. Оптимизация промежуточного кода - выделение общих подвыражений и вычисление константных подвыражений.

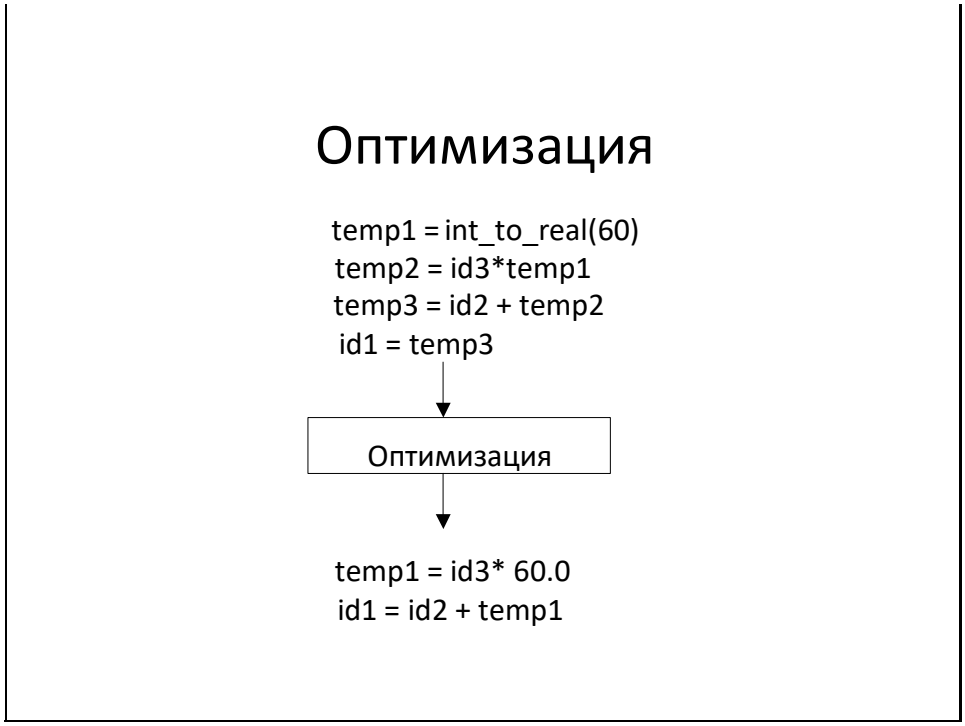


Рис. 2.15

Отметим, что существует различные критерии эффективности, 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. Наиболее распространенные методами оптимизации являются:

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

2.2.5. Генерация кода

Наконец, по оптимизированной версии промежуточного представления генерируется объектная программа. Эту задачу решает фаза *генерации кода (code generator).* Генератор кода получает в качестве входных данных промежуточное представление исходной программы и отображает его в целевой язык (см. рис. 2.16).

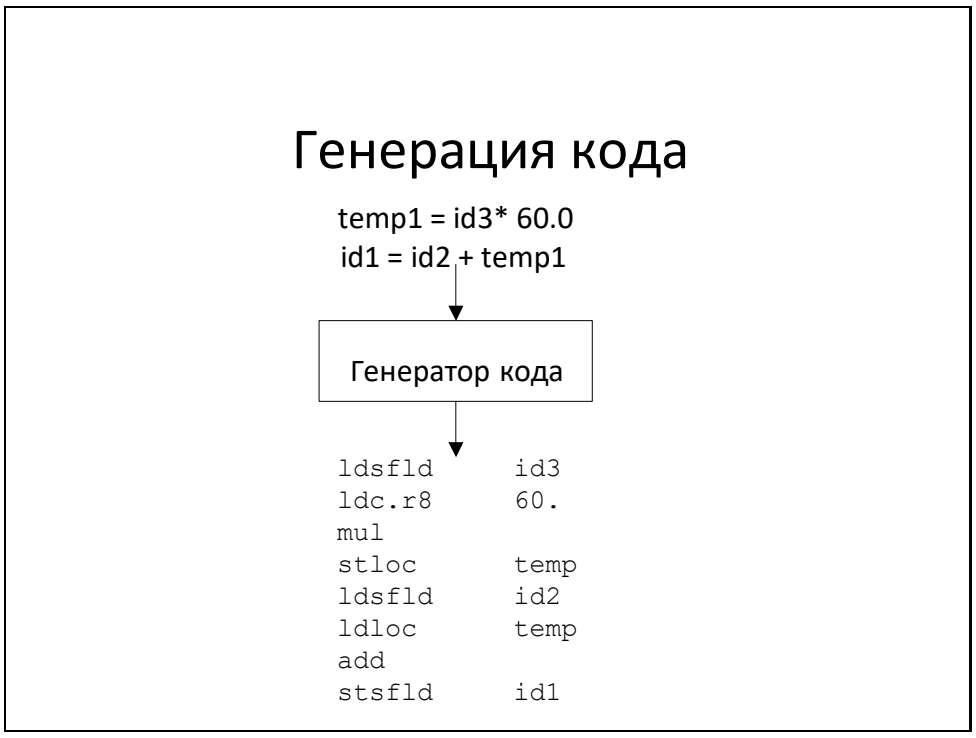


Рис. 2.16

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

- распределение памяти, т.е. *отображение имен исходной программы в адреса памяти*
- генерация объектного (ассемблерного) кода - выполняет подстановку кодовых образцов на выходном языке, соответствующих промежуточным кодам программы. Генератору кода могут не требоваться шаблоны, он весь может быть реализован в процедурном виде.
- машинно-зависимая компиляция - распределение регистров, т.е. определение для каждой точки программы множества переменных, которые должны быть размещены в регистрах, а также выбор такой последовательности записи значений в регистры, которая избавила бы от необходимости частой выгрузки значений из регистров, а затем повторной загрузки

2.2.6. Просмотры

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 нескольких 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 добиться как можно меньше просмотров, т.к. чтение программы на одном промежуточном языке и запись 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 семантический анализ до тех пор, пока мы не будем знать имена и типы всех методов объектов. Таким образом, эти задачи должны решаться на разных просмотрах (см. рис. 2.14).

Решение задачи:

- сколько фаз, возможно и более 80 фаз компиляции ПЛ/1
- какие лексемы: if, +, /*, -<имя>, тип
- на основании чего проверять синтаксис => метаязык (для описания языка)
- как описать семантику: нет формализма, но правила известны
- какой генератор кода на целевую машину

Один просмотр, а точнее полтора:

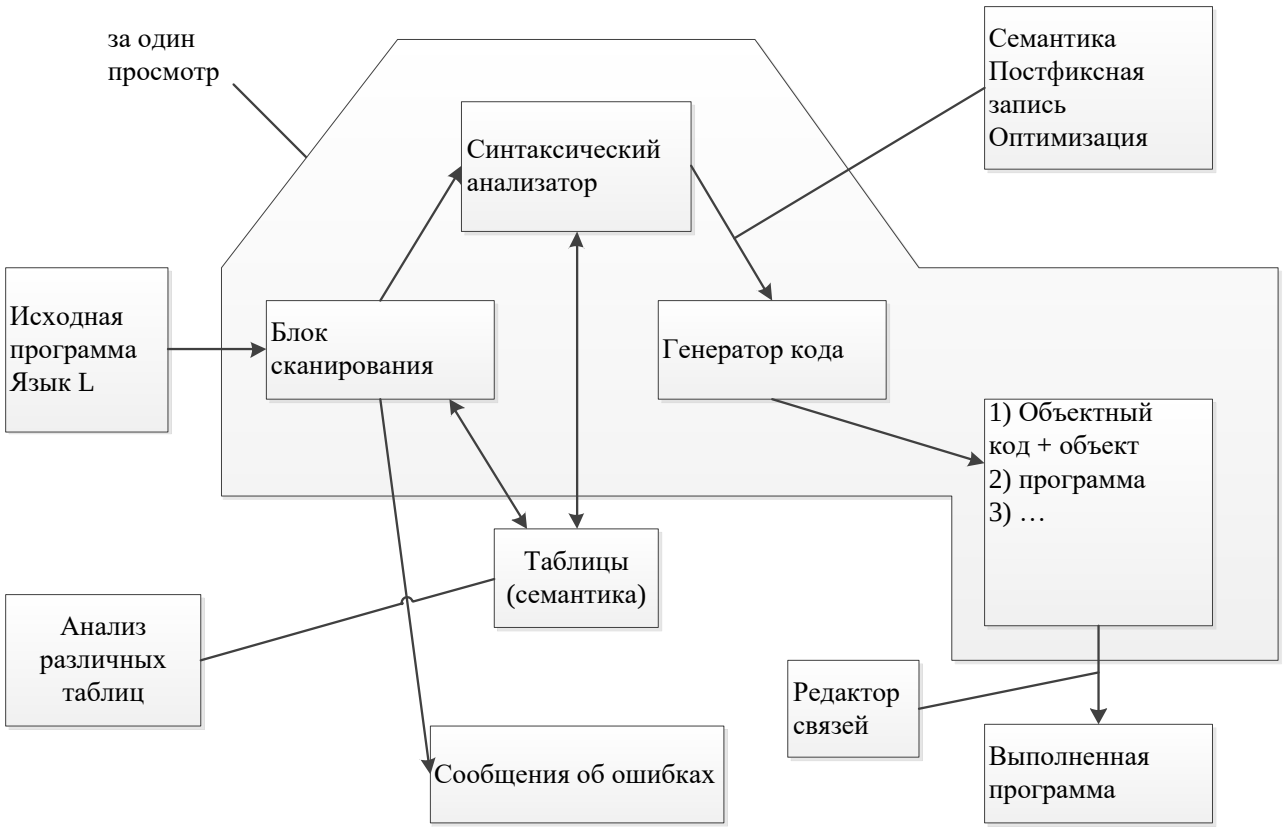


Рис. 2.17

2.2.7. Обобщенная структура транслятора

Учитывая схожесть компилятора и интерпретатора, рассмотрим фазы, существующие в компиляторе. В нем выделяются:

Фаза лексического анализа.

Фаза синтаксического анализа, состоящая из:

- распознавания синтаксической структуры;

- семантического разбора, в процессе которого осуществляется работа с таблицами, порождение промежуточного семантического представления или объектной модели языка.

Фаза генерации кода, осуществляющая:

семантический анализ компонент промежуточного представления или объектной модели языка;

перевод промежуточного представления или объектной модели в ***объектный код***.

Наряду с основными фазами процесса трансляции возможны также дополнительные фазы:

Фаза исследования и оптимизации промежуточного представления, состоящая из:

- анализа корректности промежуточного представления;

- оптимизации промежуточного представления.

Фаза оптимизации объектного кода.

Существуют и различные варианты взаимодействия блоков транслятора

Многопроходная организация, при которой каждая из фаз является независимым процессом, передающим управление следующей фазе только после полного окончания предыдущей

Интерпретатор отличается тем, что фаза генерации кода обычно заменяется фазой эмуляции элементов промежуточного представления или объектной модели языка. Кроме того, в интерпретаторе обычно не проводится оптимизация промежуточного представления, а сразу же осуществляется его эмуляция.

Кроме этого можно выделить единый для всех фаз процесс анализа и исправление ошибок, существующих в обрабатываемом исходном тексте программы.

Обобщенная структура компилятора, учитывающая существующие в нем фазы, представлена на рис. 2.18.

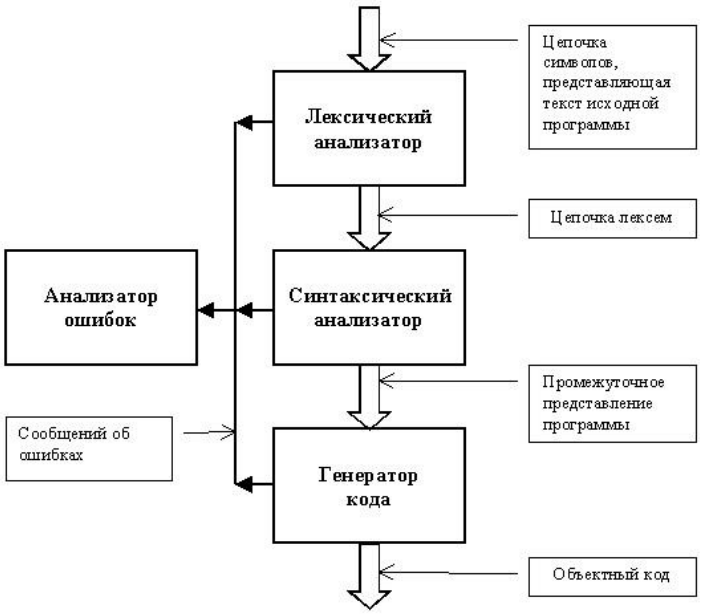


Рис. 2.18 Обобщенная структура компилятора

Он состоит из лексического анализатора, синтаксического анализатора, генератора кода, анализатора ошибок.

В интерпретаторе вместо генератора кода используется эмулятор (рис. 2.19), в который, кроме элементов промежуточного представления, передаются исходные данные. На выход эмулятора выдается результат вычислений.

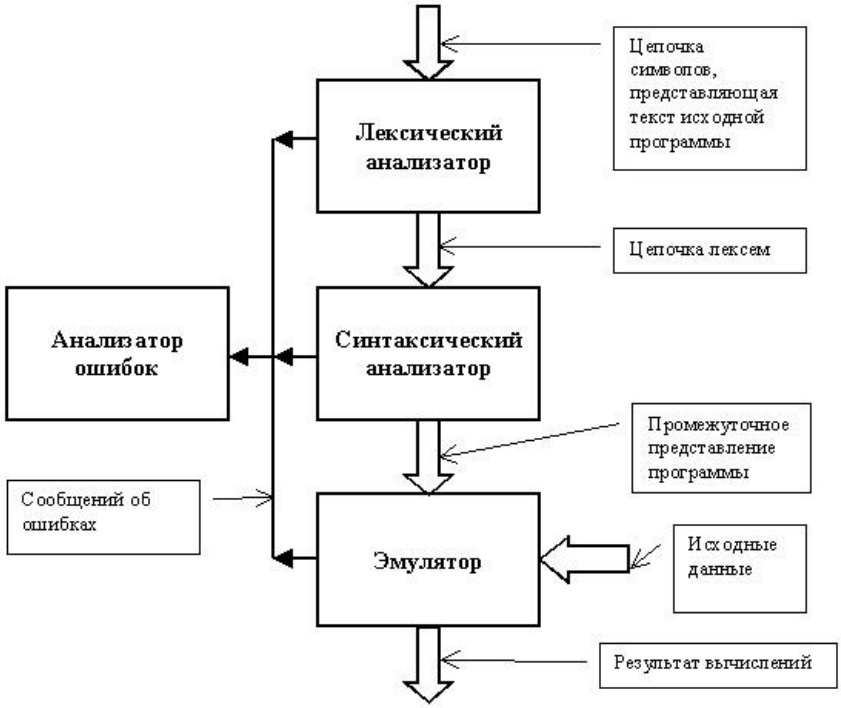


Рис. 2.19 Обобщенная структура интерпретатора

Лексический анализатор (известен также как сканер) осуществляет чтение входной цепочки символов и их группировку в элементарные конструкции, называемые лексемами. Каждая лексема имеет класс и значение. Обычно претендентами на роль лексем выступают элементарные конструкции языка, например, идентификатор, действительное число, комментарий. Полученные лексемы передаются синтаксическому анализатору. Сканер не является обязательной частью транслятора. Однако, он позволяет повысить эффективность процесса трансляции. Подробнее лексический анализ рассмотрен в теме: "Организация лексического анализа".

Синтаксический анализатор осуществляет разбор исходной программы, используя поступающие лексемы, построение синтаксической структуры программы и семантический анализ с формированием объектной модели языка. Объектная модель представляет синтаксическую структуру, дополненную семантическими связями между существующими понятиями. Этими связями могут быть:

ссылки на переменные, типы данных и имена процедур, размещаемые в таблицах имен;

связи, определяющие последовательность выполнения команд;

связи, определяющие вложенность элементов объектной модели языка и другие.

Таким образом, синтаксический анализатор является достаточно сложным блоком транслятора. Поэтому его можно разбить на следующие составляющие:

- распознаватель;

- блок семантического анализа;

- объектную модель, или промежуточное представление, состоящие из таблицы имен и синтаксической структуры.

Обобщенная структура синтаксического анализатора приведена на рис. 2.20.

Рис. 2.20 Обобщенная схема синтаксического анализатора

Распознаватель получает цепочку лексем и на ее основе осуществляет разбор в соответствии с используемыми правилами. Лексемы, при успешном разборе правил, передаются семантическому анализатору, который строит таблицу имен и фиксирует фрагменты синтаксической структуры. Кроме этого, между таблицей имен и синтаксической структурой фиксируются дополнительные семантические связи. В результате формируется объектная модель программы, освобожденная от привязки к синтаксису языка программирования. Достаточно часто вместо синтаксической структуры, полностью копирующей иерархию объектов языка, создается ее упрощенный аналог, который называется промежуточным представлением.

Анализатор ошибок получает информацию об ошибках, возникающих в различных блоках транслятора. Используя полученную информацию, он формирует сообщения пользователю. Кроме этого, данный блок может попытаться исправить ошибку, чтобы продолжить разбор дальше. На него также возлагаются действия, связанные с корректным завершением программы в случае, когда дальнейшую трансляцию продолжать невозможно.

Генератор кода строит код объектной машины на основе анализа объектной модели или промежуточного представления. Построение кода сопровождается дополнительным семантическим анализом, связанным с необходимостью преобразования обобщенных команд в коды конкретной

вычислительной машины. На этапе такого анализа окончательно определяется возможность преобразования, и выбираются эффективные варианты. Сама генерация кода является перекодировкой одних команд в другие.

Варианты взаимодействия блоков транслятора

Организация процессов трансляции, определяющая реализацию основных фаз, может осуществляться различным образом. Это определяется различными вариантами взаимодействия блоков транслятора: лексического анализатора, синтаксического анализатора и генератора кода. Несмотря на одинаковый конечный результат, различные варианты взаимодействия блоков транслятора обеспечивают различные варианты хранения промежуточных данных. Можно выделить два основных варианта взаимодействия блоков транслятора:

- многопроходную организацию, при которой каждая из фаз является независимым процессом, передающим управление следующей фазе только после окончания полной обработки своих данных;
- однопроходную организацию, при которой все фазы представляют единый процесс и передают друг другу данные небольшими фрагментами.

На основе двух основных вариантов можно также создавать их разнообразные сочетания.

Многопроходная организация взаимодействия блоков транслятора

Данный вариант взаимодействия блоков, на примере компилятора, представлен на рис 2.21.

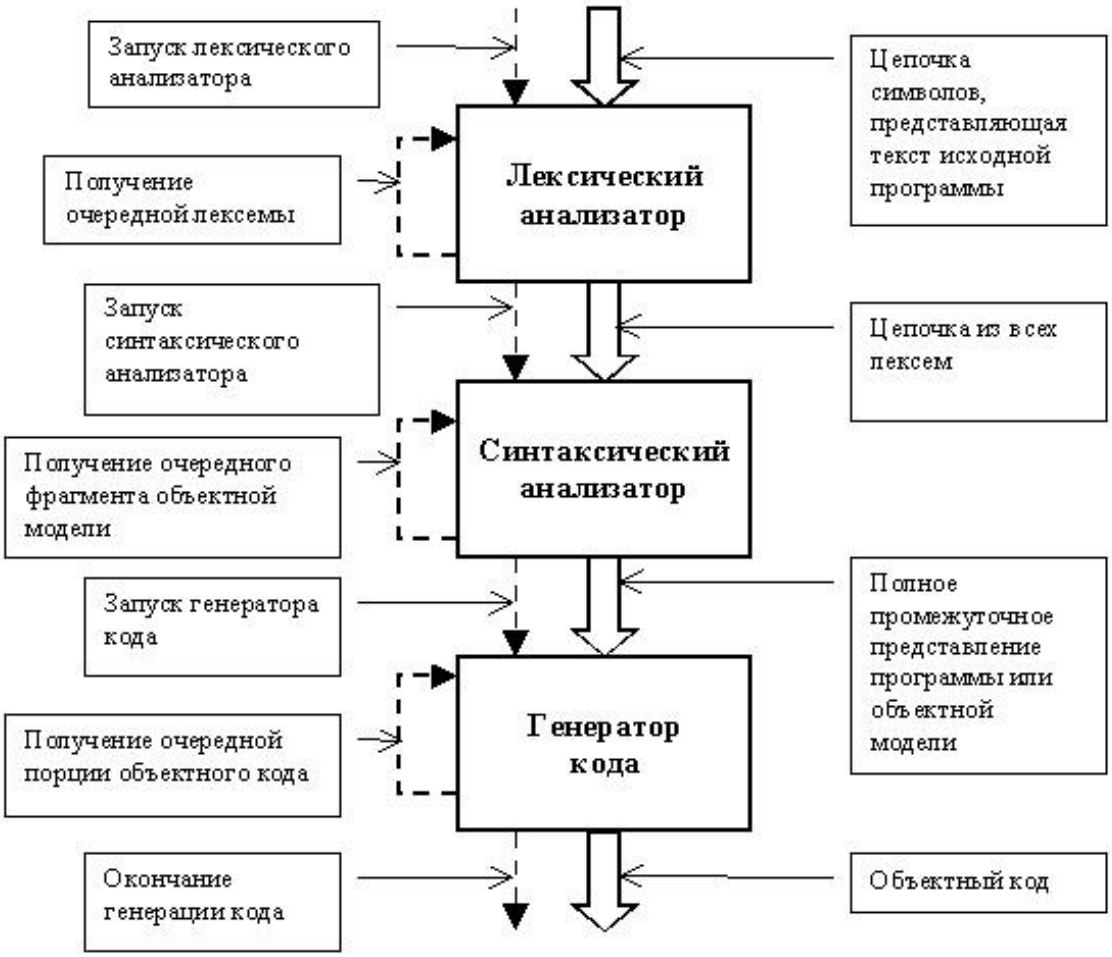


Рис. 2.21 Многопроходная схема взаимодействия блоков компилятора

Лексический анализатор полностью обрабатывает исходный текст, формируя на выходе цепочку, состоящую из всех полученных лексем. Только после этого управление передается синтаксическому анализатору. Синтаксический анализатор получает сформированную цепочку лексем и на ее основе формирует промежуточное представление или объектную модель. После получения всей объектной модели он передает управление генератору кода. Генератор кода, на основе объектной модели языка, строит требуемый машинный код

К достоинствам такого подхода можно отнести:

 Обособленность отдельных фаз, что позволяет обеспечить их независимую друг от друга реализацию и использование.

 Возможность хранения данных, получаемых в результате работы каждой из фаз, на внешних запоминающих устройствах и их использования по мере надобности.

 Возможность уменьшения объема оперативной памяти, требуемой для работы транслятора, за счет последовательного вызова фаз.

К недостаткам следует отнести.

 Наличие больших объемов промежуточной информации, из которой в данный момент времени требуется только небольшая часть.

 Замедление скорости трансляции из-за последовательного выполнения фаз и использования для экономии оперативной памяти внешних запоминающих устройств.

Данный подход может оказаться удобным при построении трансляторов с языков программирования, обладающей сложной синтаксической и семантической структурой (например, PL/I). В таких ситуациях трансляцию сложно осуществить за один проход, поэтому результаты предыдущих проходов проще представлять в виде дополнительных промежуточных данных. Следует отметить, что сложные семантическая и синтаксическая структуры языка могут привести к дополнительным проходам, необходимым для установления требуемых зависимостей. Общее количество проходов может оказаться более десяти. На число проходов может также влиять использование в программе конкретных возможностей языка, таких как объявление переменных и процедур после их использования, применение правил объявления по умолчанию и т. д.

Однопроходная организация взаимодействия блоков транслятора

Один из вариантов взаимодействия блоков компилятора при однопроходной организации представлено на рис. 2.22.

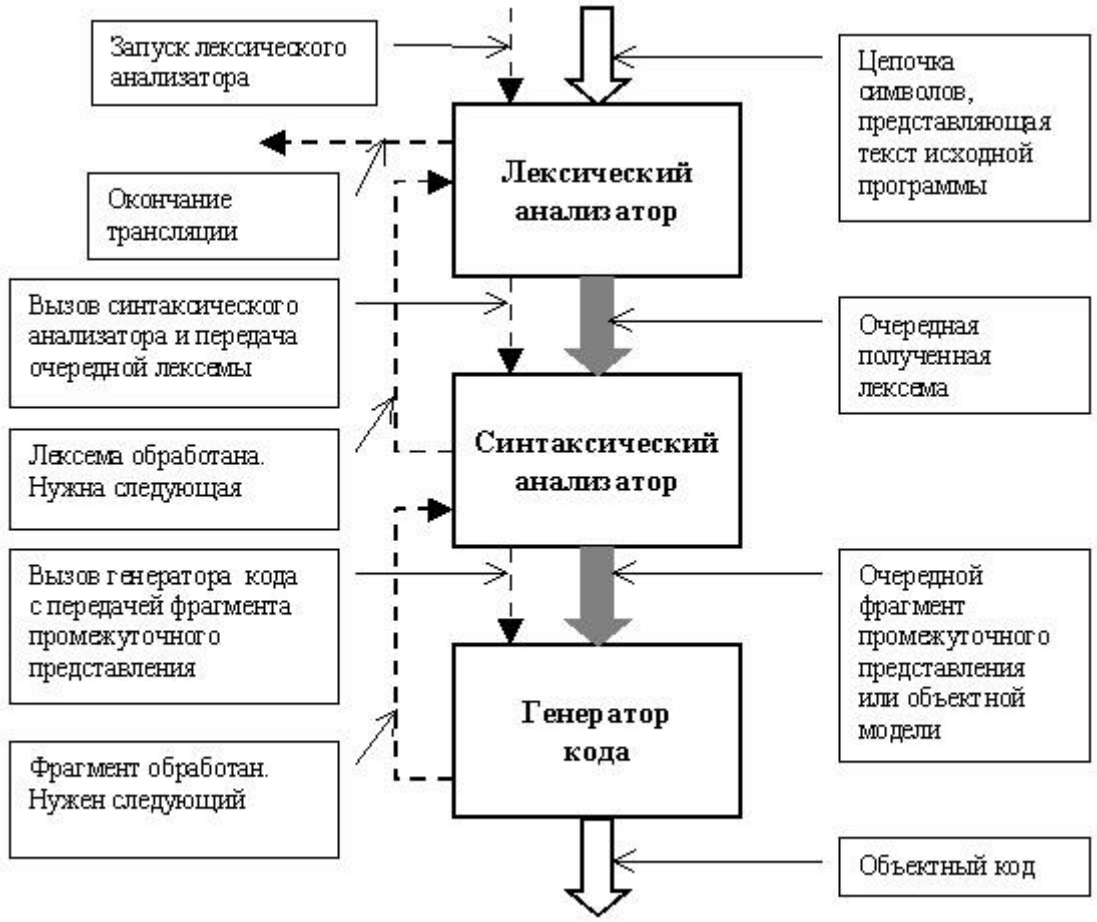


Рис. 2.22 Однопроходная схема взаимодействия блоков компилятора при управлении, инициируемом лексическим анализатором

В этом случае процесс трансляции протекает следующим образом. Лексический анализатор читает фрагмент исходного текста, необходимый для получения одной лексемы. После формирования лексемы им осуществляется вызов синтаксического анализатора и передача ему созданной лексемы в качестве параметра. Если синтаксический анализатор может построить очередной элемент промежуточного представления, то он делает это и передает построенный фрагмент генератору кода. В противном случае синтаксический анализатор возвращает управление сканеру, давая, тем самым, понять, что очередная лексема обработана и нужны новые данные.

Генератор кода функционирует аналогичным образом. По полученному фрагменту промежуточного представления он создает соответствующий фрагмент объектного кода. После этого управление возвращается синтаксическому анализатору.

По окончании исходного текста и завершении обработки всех промежуточных данных каждым из блоков лексический анализатор инициирует процесс завершения программы.

Чаще всего в однопроходных трансляторах используется другая схема управления, в которой роль основного блока играет синтаксический анализатор (рис. 2.23).

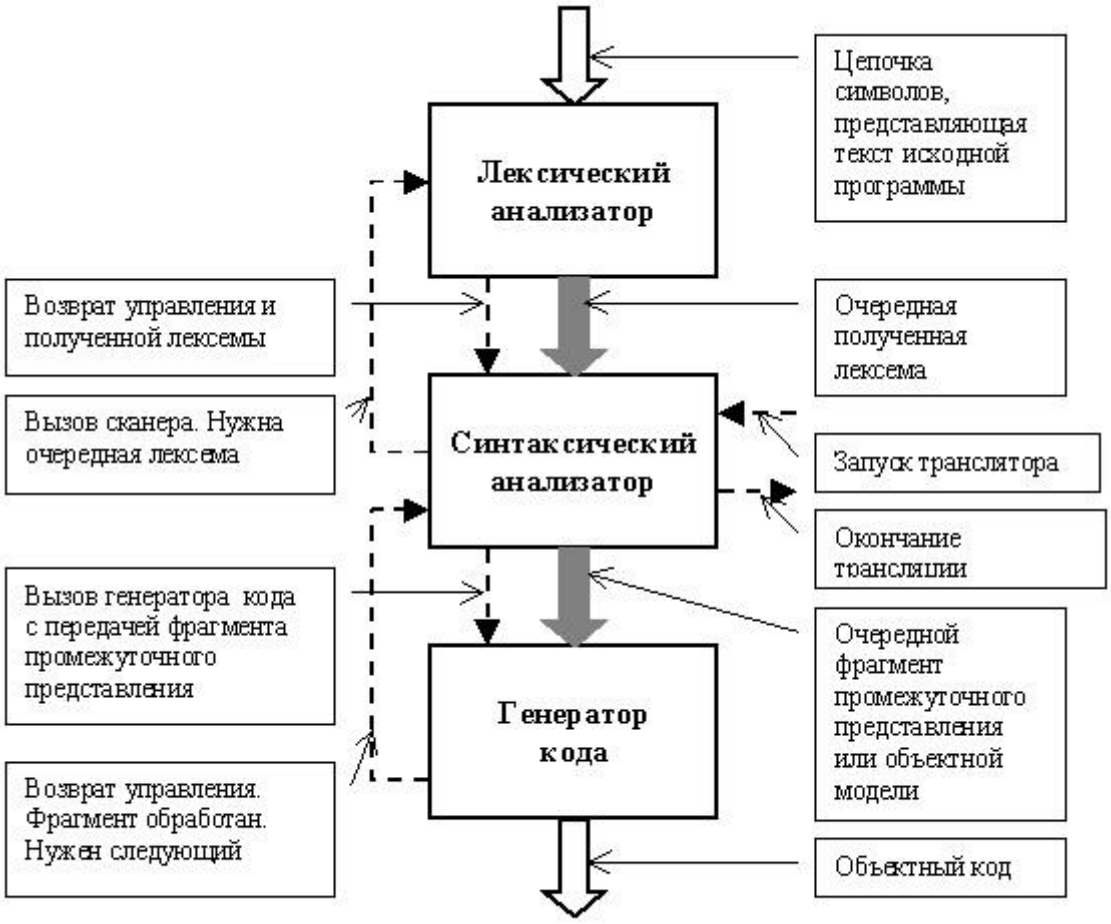


Рис. 2.23 Однопроходная схема взаимодействия блоков компилятора при управлении, инициируемом синтаксическим анализатором

Лексический анализатор и генератор кода выступают в роли вызываемых им подпрограмм. Как только синтаксическому анализатору нужна очередная лексема, он вызывает сканер. При получении фрагмента промежуточного представления осуществляется обращение к генератору кода. Завершение процесса трансляции происходит после получения и обработки последней лексемы и инициируется синтаксическим анализатором.

К достоинствам однопроходной схемы следует отнести отсутствие больших объемов промежуточных данных, высокую скорость обработки из-за совмещения фаз в едином процессе и отсутствие обращений в внешним запоминающим устройствам.

К недостаткам относятся: невозможность реализации такой схемы трансляции для сложных по структуре языков и отсутствие промежуточных

данных, которые можно использовать для комплексного анализа и оптимизации.

Такая схема часто применяется для простых по семантической и синтаксической структурам языков программирования, как в компиляторах, так и в интерпретаторах. Примерами таких языков могут служить Basic и Pascal. Классический интерпретатор обычно строится по однопроходной схеме, так как непосредственное исполнение осуществляется на уровне отдельных фрагментов промежуточного представления. Организация взаимодействия блоков такого интерпретатора представлена на рис. 2.24.

Рис. 2.24 Однопроходная схема взаимодействия блоков интерпретатора

Комбинированные взаимодействия блоков транслятора

Сочетания многопроходной и однопроходной схем трансляции порождают разнообразные комбинированные варианты, многие из которых успешно используются. В качестве примера можно рассмотреть некоторые из них.

На рис. 2.25 представлена схема взаимодействия блоков транслятора, разбивающая весь процесс на два прохода. На первом проходе порождается полное промежуточное представление программы, а на втором осуществляется генерация кода. Использование такой схемы позволяет легко переносить

транслятор с одной вычислительной системы на другую путем переписывания генератора кода.

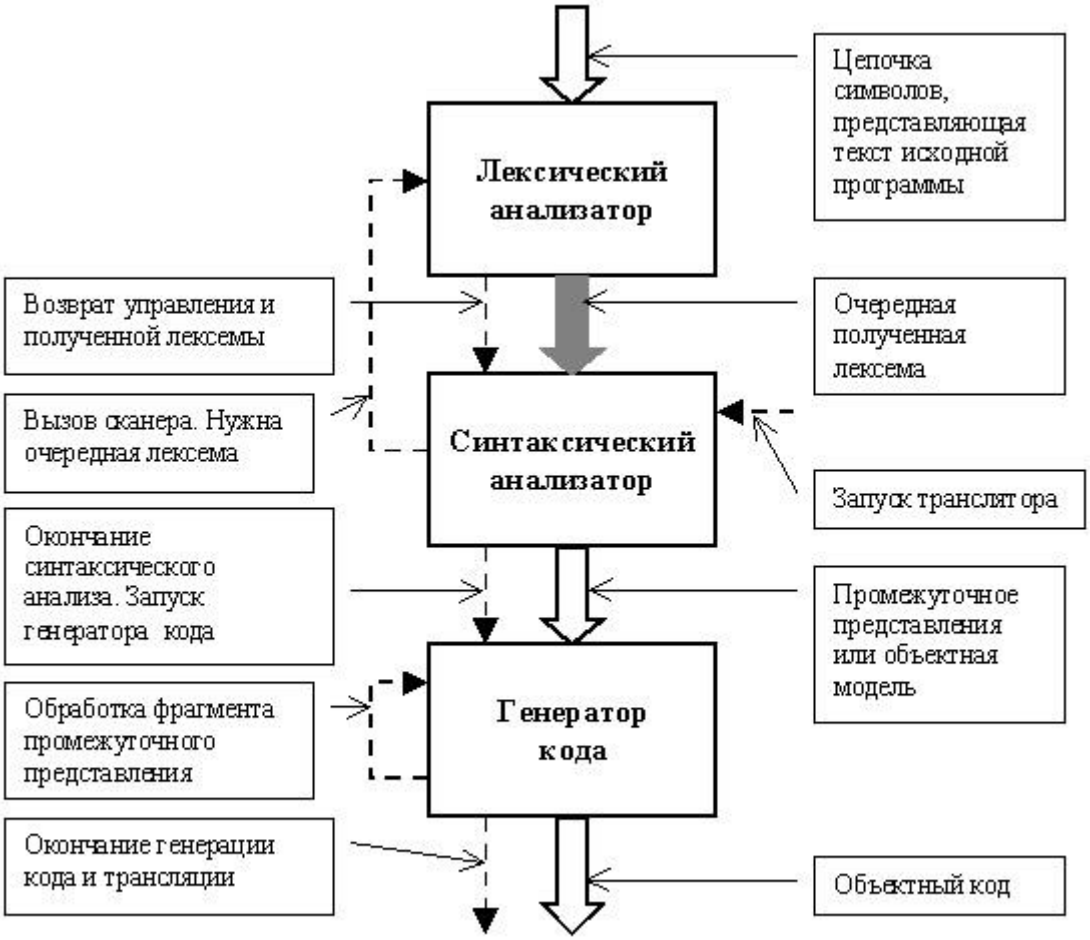


Рис. 2.25 2-х проходная схема с генерацией полного промежуточного представления

Кроме этого, вместо генератора кода легко подключить эмулятор промежуточного представления, что достаточно просто позволяет разработать систему программирования на некотором языке, ориентированную на различные среды исполнения. Пример подобной организации взаимодействия блоков транслятора представлен на рис. 2.26.

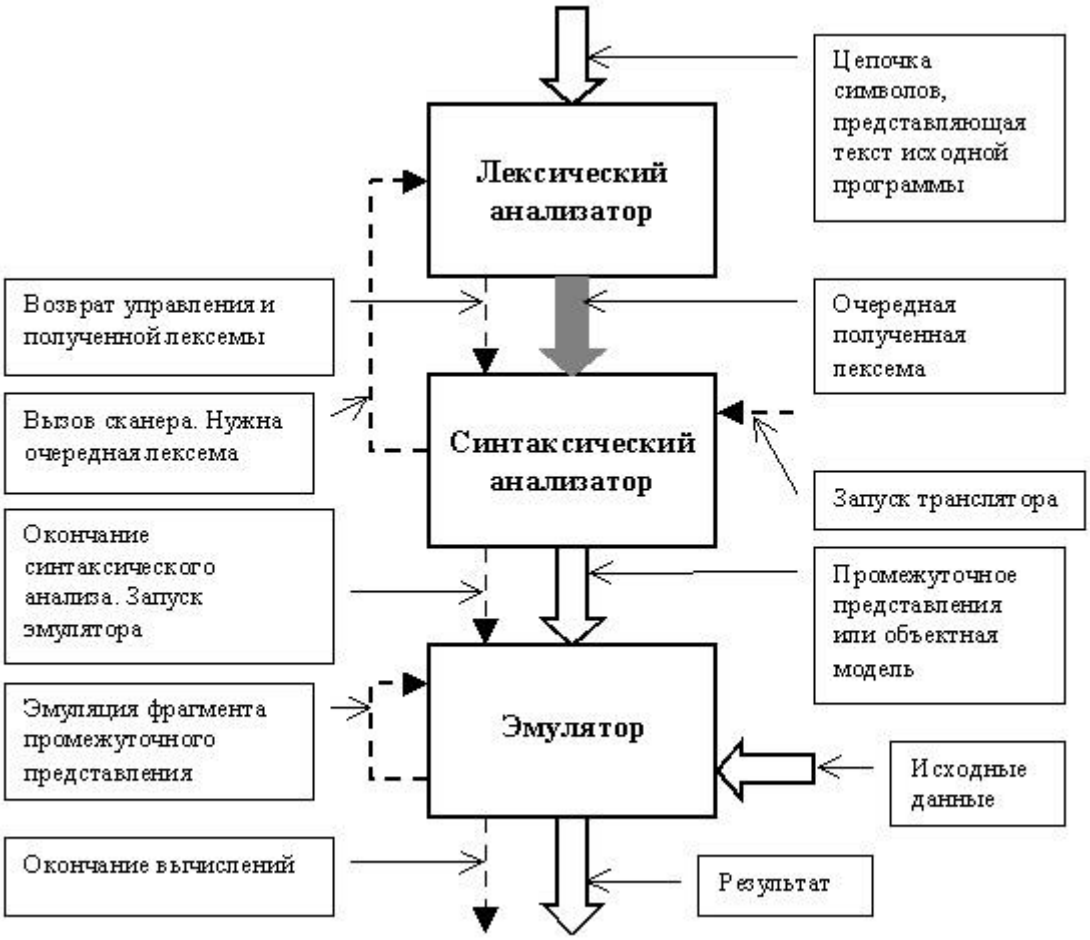


Рис. 2.26 Эмуляция промежуточного представления

Наряду со схемами, предполагающими замену генератора кода на эмулятор, существуют трансляторы, допускающие их совместное использование. Одна из таких схем представлена на рис. 2.27.

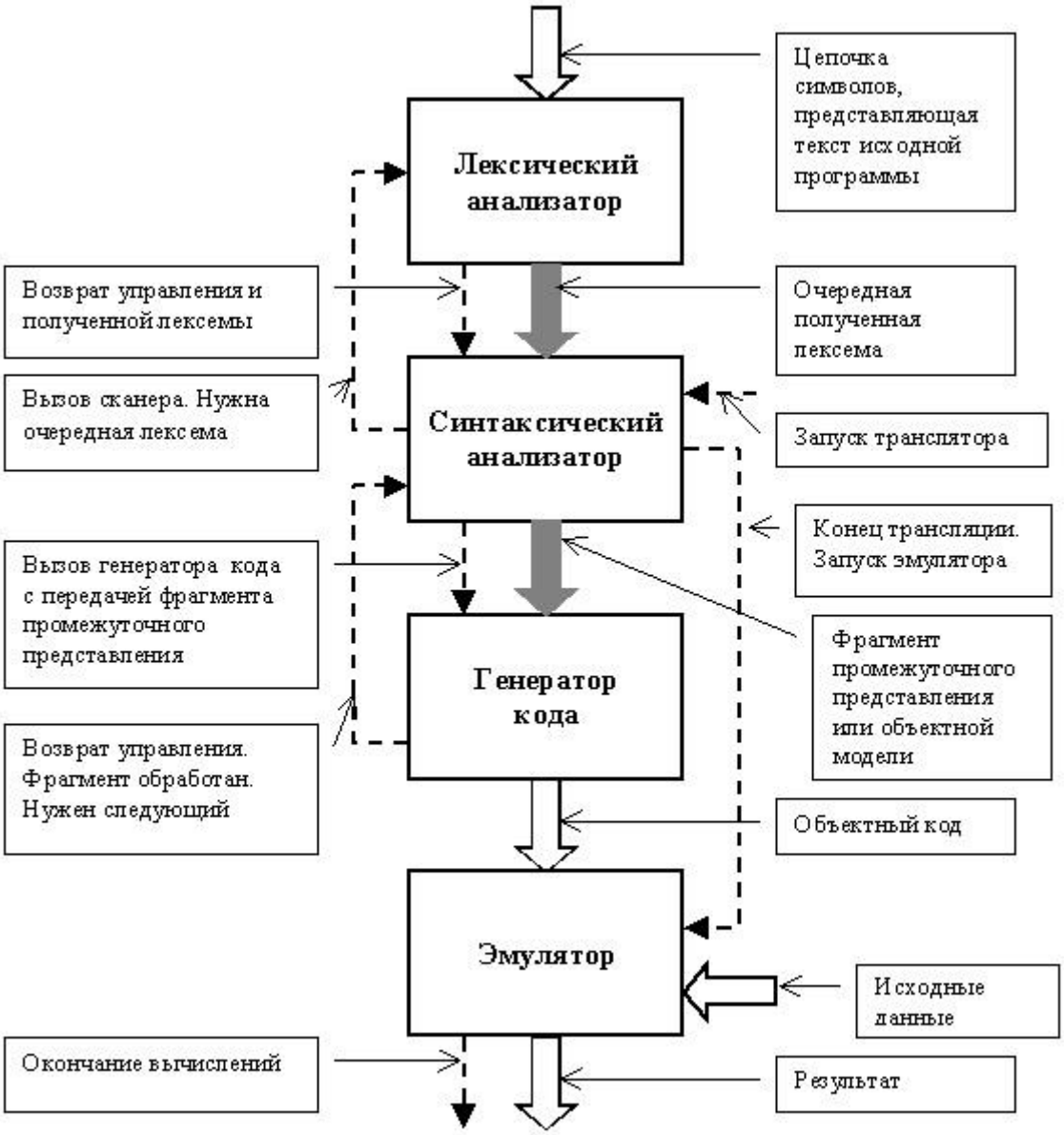


Рис. 2.27 Эмуляция скомпилированного объектного кода

Рис. 2.28 Стадии трансляции

Почти на каждой стадии могут выполняться дополнительные оптимизации, а также происходит анализ и обработка ошибок (см. рис. 2.28).

*Ф*азы лексического и синтаксического анализа реализуются на основе теории формальных языков. Обычно лексические анализаторы генерируются с помощью *регулярных выражений*.

Синтаксический анализ основывается на теории контекстно-свободных (КС) грамматик. Общая форма КС-грамматики не позволяет разбирать язык достаточно простым (в частности, автоматически сгенерированным) парсером, потому языки программирования обычно принадлежат одному из нескольких *специальных подклассов КС-языков* (LL, LR, LALR), которые проще разбирать.

Основные достижения теории формальных языков насчитывают уже несколько десятков лет, и в этой области не ожидается каких-либо новых разработок, тем более что имеющиеся теоретические результаты хорошо решают практические задачи. По-другому обстоит дело с изучением других двух разделов компиляции: семантическим анализом и оптимизацией. Работы по формальному описанию семантики языков программирования интенсивно появлялись в 70–80-х годах. Однако ни одна из построенных теорий не вошла в

широкое употребление при построении компиляторов, и сегодня дело обстоит таким образом, что семантика вначале описывается на естественном языке, а затем реализуется в компиляторе большим набором вручную закодированных условных операторов.

Напротив, в последние годы растёт актуальность теории оптимизирующих преобразований — это объясняется появлением всё более высокоуровневых языков и разнообразных аппаратных платформ. Оптимизация также требует привлечения развитого математического аппарата (в частности, для решения вопроса о корректности преобразований). Многие из достижений активно внедряются в промышленные компиляторы. Всё больше это касается преобразований, связанных с распараллеливанием.

На этапе построения семантического дерева строится *таблица символов* - в нее заносятся имена, присутствовавшие в исходном тексте, к ним добавляются такие атрибуты как тип, отведённая память, область видимости.