**КС = Контекстно-свободный  
КА = конечный автомат  
МА = магазинный автомат  
  
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.](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *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 языкаобъекта, 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.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *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 **А** 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. A1 ® A** | **23. A1 ® W** | | **45. A2 ® s** |
| **2. A1 ® B** | 24. A1 ® X | | 46. A2 ® t |
| **3. A1 ® C** | 25. A1 ® Y | | 47. A2 ® u |
| **....** | .... | | ..... |
| **20. A1 ® T** | | | 42. A2 ® p | 64. A3 ® A3A1 | |
| **21. A1 ® U** | | | 43. A2 ® q | 65. A4 ® A3A2 | |

**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, **C** 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 C* 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 *C* все функции должны находиться на внешнем уровне вложенности. Язык *C++* 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 C 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 **V1**. |

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, С, 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: *функцияn ( ... функция2 (функция1 (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 п* → *последовательность операторов п*

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 объектноориентированная 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](http://dic.academic.ru/dic.nsf/ruwiki/1465) 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**,](http://dic.academic.ru/dic.nsf/ruwiki/5272) а не [**компилируются**,](http://dic.academic.ru/dic.nsf/ruwiki/6953) хотя сценарные языки программирования один за другим обзаводятся [JIT-](http://dic.academic.ru/dic.nsf/ruwiki/1435136)

[компиляторами.](http://dic.academic.ru/dic.nsf/ruwiki/1435136) JIT-компиляция ([*англ.*](http://ru.cyclopaedia.net/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Just-in-time compilation, компиляция «на лету»), динамическая компиляция ([*англ.*](http://ru.cyclopaedia.net/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) dynamic translation) — технология увеличения производительности программных систем, использующих [*байткод*,](http://ru.cyclopaedia.net/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) путём компиляции байт-кода в [*машинный код*](http://ru.cyclopaedia.net/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) 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**](http://dic.academic.ru/dic.nsf/ruwiki/976114) или текстового редактор[а](http://dic.academic.ru/dic.nsf/ruwiki/2332) и *средств администрирования* [операционных систем.](http://dic.academic.ru/dic.nsf/ruwiki/2566)

# 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 **А** 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. A1 **®** A 23. A1 **®** W 45. A2 **®** s   2. A1 **®** B 24. A1 **®** X 46. A2 **®** t 3. A1 **®** C 25. A1 **®** Y 47. A2 **®** u   .... .... .....   * 1. A1 **®** T 42. A2 **®** p 64. A3 **®** A3A1   2. A1 **®** U 43. A2 **®** q 65. A4 **®** A3A2 |

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 А 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. A1=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**3**A**1**+A**3**A**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> :: =   А|В|С|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.

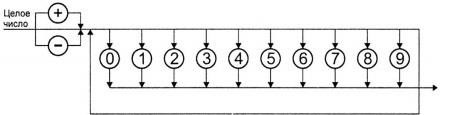
Каждой синтаксической конструкции соответствует одна диаграмма Вирта, в которой каждому сочетанию металингвистических переменных соответствует своя графическая диаграмма, на которой они посредством дуг. Альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием.

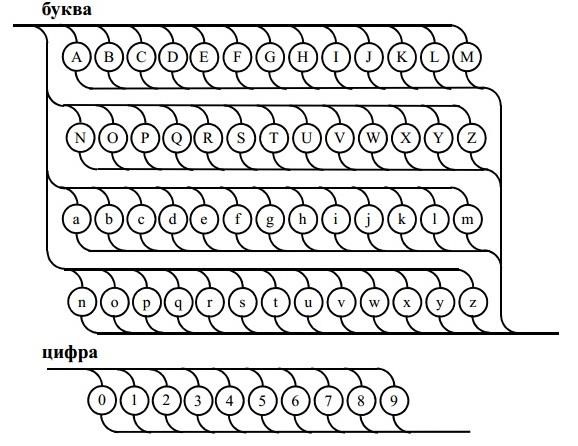
Имя определяемой синтаксической конструкции записывается над стрелкой, входящей в диаграмму (точка входа в диаграмму), которая, как правило, располагается в левом верхнем углу.

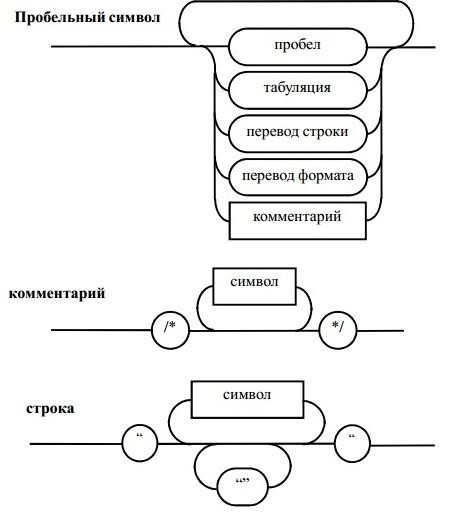
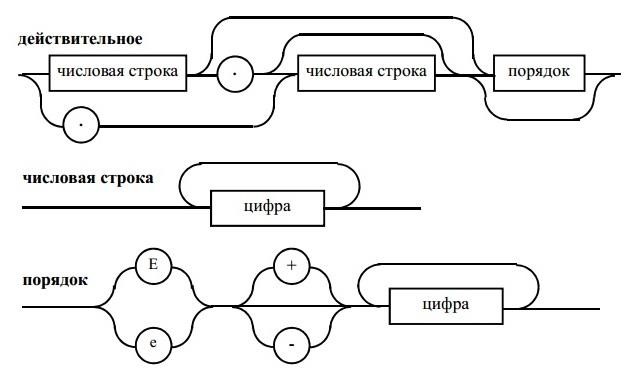
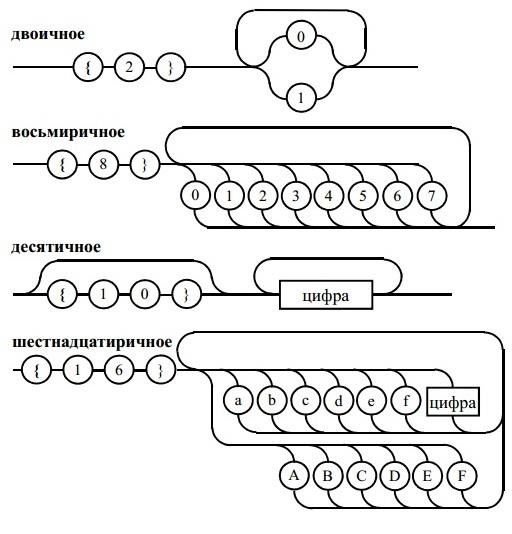
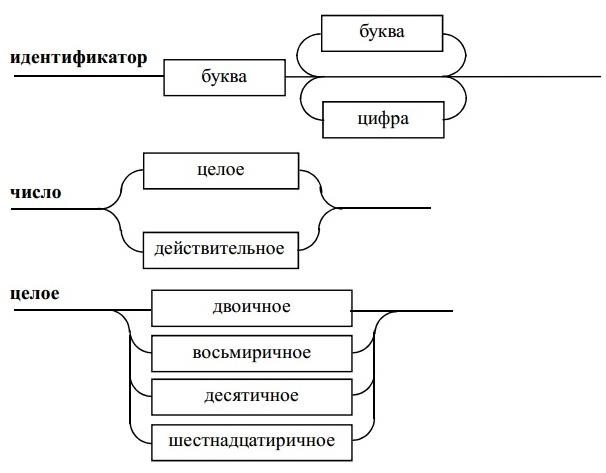
Любой путь от точки входа в синтаксическую диаграмму к выходу (исходящая из диаграммы стрелка) представляет собой цепочку металингвистических переменных и основных символов языка, соответствующую одному из вариантов правой части РБНФ. На рис. 1.2 приведены синтаксические диаграммы, определяющие множество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

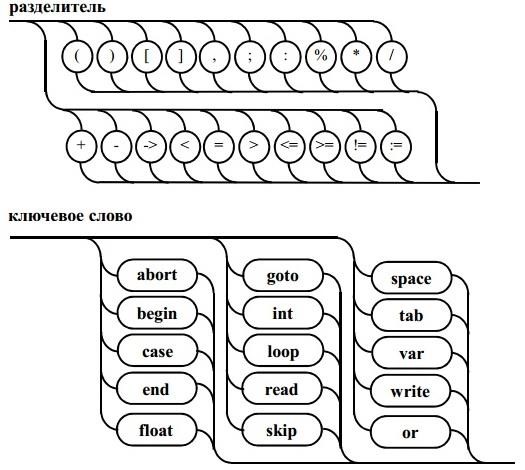
## Элементарные конструкции. Использование диаграмм Вирта

Синтаксические диаграммы для определения множества целых чисел









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*∈A*.* Знак ∈ означает принадлежность множеству A. Итак, А={*a1*, *a2*,… *an*, } – множество, *ai* ∈A *элемент* множество*. Отрицание этого утверждения записывается a*∉A*. Если* А содержит конечное число элементов, то А называется *конечным* множеством. Символ  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 A *содержится* во множестве B, и пишут A∈B, если каждый элемент из A является элементом из B. Если B содержит элемент, не принадлежащий Aи A⊆B, говорят, что A, *собственно* содержится в B (диаграмма Венна для включения множеств).

Диаграмма Венна

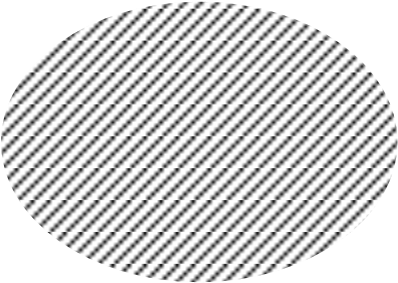
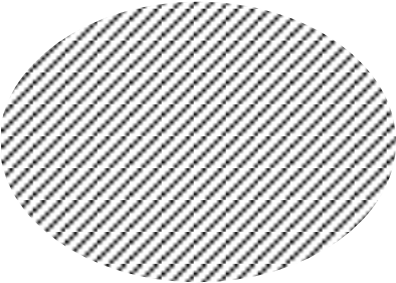
*А*

*В*

## Объединение множеств

A∪B = {x | x∈A или x∈B} -это множество, содержащее все элементы А и В (Диаграмма Венна для

объединения множеств):



*А*



*В*

**Пересечение множеств**

A∩B = {x | x∈A и x∈B} -

это множество, состоящее из всех тех элементов, которые принадлежат

обоим множествам *А* и *В* (рис. 1.5)*.*



*А*

*В*

Рис. 1.5. Диаграмма Венна для пересечений множеств

**Разность множеств** А – В - это множество элементов А, не принадлежащих В (рис. 1.6).

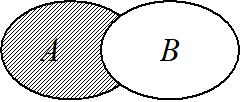


Рис. 1.6. Диаграмма Венна для разности множеств

*Декартово произведение А* и *В*

*A*✕B = {(a,b| aA и bB}

*Пример.* Если *А*={1,2}, *B*={2,3,4}, то *A**B*={(1,2), (1,3), (1,4), (2,2), (2,3), (2,4)}

### ***Отношения***

Пусть *А* и *В* – множества. **Отношением из *А* в *В* называется любое подмножество множеств *А******В.***

Если *А=В*, то отношение задано (определено) на *А*.

Если *R* отношение из *А* в *В* и (*a, b*)*R*, то пишут *aRb*.

Множество *А* называют областью определения, *В* - множеством значений.

*Пример.*

*А* – множество целых чисел. Отношение *L* представляет множество {(*a, b*) *| a < b*} *aLb.*

*Определение****.***

Отношение {(*b, а*) *|* (*a, b*)*R*} называют обратным к отношению *R*, т.е. *R-*1*.*

*Определение****.***

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

Тогда *R* называют:

1. *рефлексивным*, если *aRa* для всех пар из *А;*
2. *симметричным*, если *aRb* влечет *bRa* для всех *a* и *b* из *А;*
3. *транзитивным*, если *aRb* и *bRс* влекут *aRс* для *a, b, с* из *А.*

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

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

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

[0]

[1]

[2]

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 *А* (Rk) определяется:

1. *aR*1*b* тогда и только тогда, когда *aRb;*
2. *aR*i*b* для *i>1* тогда и только тогда, когда существует такое *c**A*, что

*aRc* и cRi-1

*Транзитивное замыкание* отношения множества *R* на *А (R+)* определяется так: *аR+b* тогда и только тогда, когда *аRib* для некоторого *i**1*.

Расшифровка понятия:

*аR+b,* если существует последовательность c1, c2,…, cn, состоящая из 0 или более элементов принадлежащих *А*, такая, что *aRc*1, *aRc*2,… *aRc*n-1, *aRc*n, *c*n*Rb*. Если *n=*0, то *aRb.*

*Рефлексивное и транзитивное* замыкание отношения *R* (*R\**) на множестве *А* определяется следующим образом:

1. *aR\*a* для всех *а**А;*
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*0*b* тогда и только тогда, когда *a=b*, то *aR\*b* тогда и только тогда, когда *aR*i*b* для некоторого *i**0*.

Единственное различие между *R+* и *R\** состоит в том, что *aR\*a* истинно для всех *а**А*, но *aR+а* 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 *А* называют отношение *R* на *А* такое, что:

1. *R* – транзитивно;
2. для всех *а**А* утверждение *aRa* ложно, т.е. отношение *R* иррефлексивно.

*Пример.*

S= {e1, e2, …, en}, - множество, состоящее из *n* элементов, и пусть *А=P*(*S*). Положим *aRb* для любых *a* и *b* из *А* тогда и только тогда, когда *a*  *b*.

Отношение *R* называется частичным порядком. Для случая *S*={0, 1, 2} имеем (рис. 1.8).

{0}

{2}

{



}

{

1,

2}

{0

,

1}

{1}

2}

{0

,1,

{0

,

2}

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 *А* – это такой частичный порядок, что, если *а* и *b**А*, то либо *aRb*, либо *bRa*, либо *a=b*. Удобно это понять из следующего.

Пусть *А* представлено в виде последовательности *а*1*, а*2*,…,an*, для которых *а*iR*а*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.

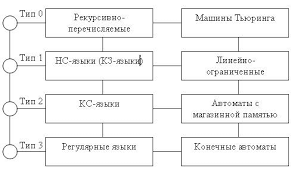
**Раздел 2. Общая модель компилятора. Фазы компиляции.**

# 1.4 Классификация языков по Хомскому(Не было в методе)

<https://ru.wikipedia.org/wiki/%D0%98%D0%B5%D1%80%D0%B0%D1%80%D1%85%D0%B8%D1%8F_%D0%A5%D0%BE%D0%BC%D1%81%D0%BA%D0%BE%D0%B3%D0%BE>

Определение

классификация [формальных языков](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) и [формальных грамматик](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0), согласно которой они делятся на 4 типа по их условной сложности. Предложена профессором [Массачусетского технологического института](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B0%D1%87%D1%83%D1%81%D0%B5%D1%82%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%81%D1%82%D0%B8%D1%82%D1%83%D1%82), лингвистом [Ноамом Хомским](https://ru.wikipedia.org/wiki/%D0%A5%D0%BE%D0%BC%D1%81%D0%BA%D0%B8%D0%B9,_%D0%90%D0%B2%D1%80%D0%B0%D0%BC_%D0%9D%D0%BE%D0%B0%D0%BC).

Примеры (Смотри по ссылке на вики выше)  
 

# 2.1 Общая модель компилятора

## 2.1.1 Основные задачи компиляторов

Трансляторы бывают двух типов: **компиляторы (compiler) и интерпретаторы (interpreter)**. **Компилятоp** (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.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *John Backus)*

Компилятор переводит программы с одного языка на другой. Входом компилятора служит цепочка символов, составляющая исходную программу на языке программирования *L1*. Выход компилятора (объектная программа) также представляет собой цепочку символов, но принадлежащую другому языку *L2*, например, языку некоторого компьютера. При этом сам компилятор написан на языке *L3*, возможно, отличающемся от первых двух. Будем называть язык *L1* исходным языком, язык *L2* – целевым языком, а язык *L3* – языком реализации. Таким образом, можно говорить о компиляторе *K* как об отображении множества *L1* в множество *L2*, т.е. *L1* ➾*K* *L2* (см. рис. 2.1).

|  |
| --- |
| **Программа на входном языке** - цепочка символов, составляющая исходную программу на языке **программирования *L1*.**  **Объектная программа** – код программы на **целевом языке *L2.***  **Компилятор*****K*** отображает множество ***L1*** в множество ***L2***, используя язык  реализации ***L3***– т.е. *L1* ➾*K* *L2* |



Программа на

входном языке



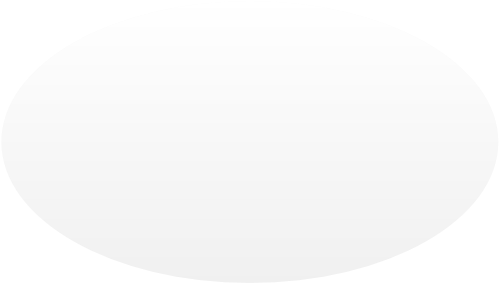
Сообщения об

ошибках



Объектная

программа



Компилятор

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.

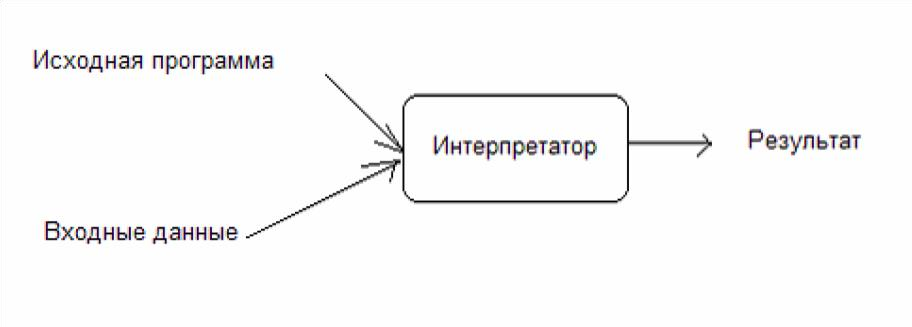
Важной исторической особенностью компилятора являлось то, что он мог производить и компоновку (то есть содержал две части - транслятор и компоновщик). Это связано с тем, что раздельная компиляция и компоновка как отдельная стадия сборки выделились значительно позже появления компиляторов, и многие популярные 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.

Начало

Выборка

очередной

инструкции

Анализ

инструкции

Выполнение

инструкции

## 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, позволяюшими создавать программы, некоторые характеристики которых (например, размеры и типы переменных) могут зависеть от входных данных.

Некоторые черты языков программирования таковы, что они не могут быть реализованы иначе, чем с использованием интерпретации.

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

### **2.1.4. Объектная программа**

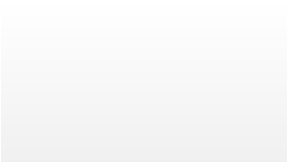
Объектная программа может быть:

* 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.

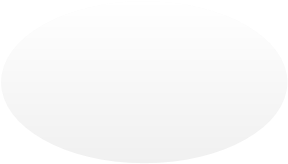
Однако для более сложных программ необходимо создавать объектные программы в форме последовательности *перемещаемых машинных команд*. Перемещаемая машинная команда представляет собой команду, в которой адресация ячеек памяти производится относительно некоторого подвижного начала. Объектную программу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

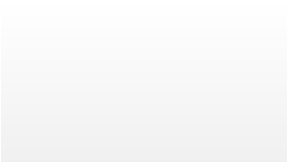


Объектная

программа

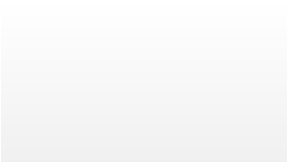


Компилятор



Загрузочный

модуль

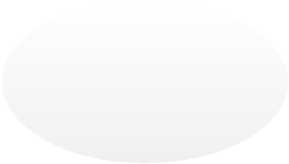


Программа на

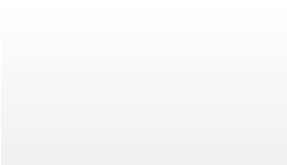
языке

L

1



Редактор связей



Объектная

программа

Рис.

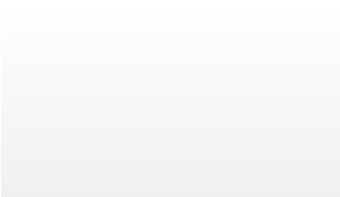
2

.

4

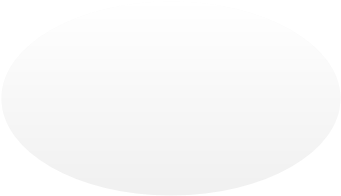
### **2.1.5. Трансляция в ассемблер**

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).

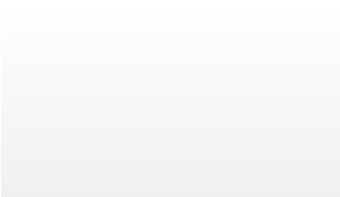


Объектная

программа

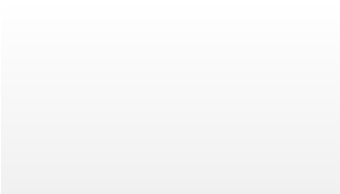


Компилятор



Объектная

программа

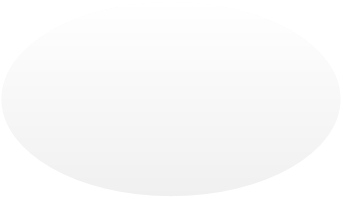


Программа на

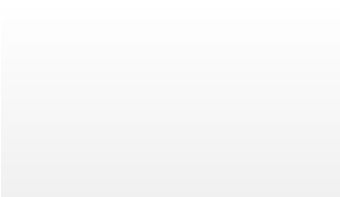
языке

L

1

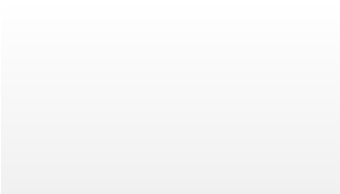


Редактор связей



Объектная

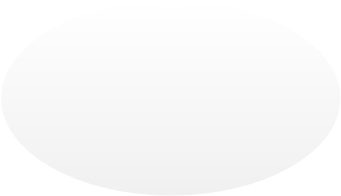
программа



Программа на

языке

A



Ассемблер

## 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).](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%82%D0%B5%D0%B3%D0%BE%D1%80%D0%B8%D1%8F:%D0%AF%D0%B7%D1%8B%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D1%81_%D1%80%D1%83%D1%81%D1%81%D0%BA%D0%B8%D0%BC%D0%B8_%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%B2%D1%8B%D0%BC%D0%B8_%D1%81%D0%BB%D0%BE%D0%B2%D0%B0%D0%BC%D0%B8)

[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.](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%82%D0%B5%D0%B3%D0%BE%D1%80%D0%B8%D1%8F:%D0%AF%D0%B7%D1%8B%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D1%81_%D1%80%D1%83%D1%81%D1%81%D0%BA%D0%B8%D0%BC%D0%B8_%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%B2%D1%8B%D0%BC%D0%B8_%D1%81%D0%BB%D0%BE%D0%B2%D0%B0%D0%BC%D0%B8)

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* ➾*K A1)*, а сам компьютер *M* по каким-то причинам не доступен либо пока еще не существует компилятор Pязыка ассемблер для компьютера *M* (*P* ➾*M A).* Нам необходимо создать компилятор *L* для компьютера *M* (*L* ➾*K A)*. В такой ситуации мы можем использовать *K* в качестве *инструментальной* машины и написать компилятор *L* для компьютера *M* (*L* ➾*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.

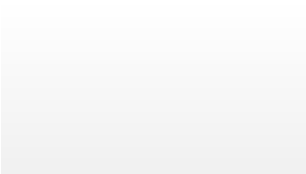
Под переносимой (*portable)* программой понимается программа, которая может без перетрансляции выполняться на нескольких (по меньшей мере, на двух) платформах. В связи с этим возникает вопрос о переносимости объектных программ, создаваемых компилятором. Компиляторы, созданные по методикам, рассмотренным 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.

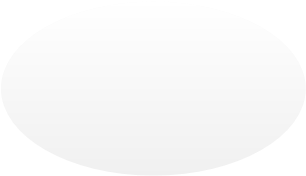
2.6). При выполнении этих условий можно достаточно быстро написать интерпретатор виртуальной шины. можно достаточно быстро написать интерпретатор виртуальной машины.



Байт

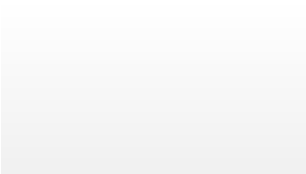
-

код



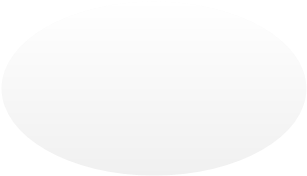
Компилятор

Результаты

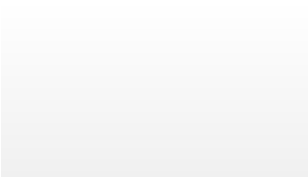


Программа на

исходном языке



Интерпретатор



Данные

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.



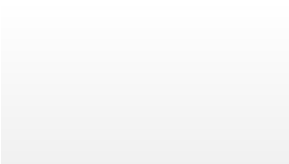
## Рис. 2.7 Никлаус Вирт [(нем.](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BC%D0%B5%D1%86%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Niklaus Wirth)*

**Никлаус Вирт -** известнейший теоретик в области разработки [языков программирования,](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) [профессор](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%84%D0%B5%D1%81%D1%81%D0%BE%D1%80) Швейцарской высшей технической школы Цюриха, создал языков программирования [Паскаль,](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%81%D0%BA%D0%B0%D0%BB%D1%8C_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) [Модула-2,](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D0%B0-2) [Оберон](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%B5%D1%80%D0%BE%D0%BD_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) (совм. с  [Юргом Гуткнехтом)](https://ru.wikipedia.org/w/index.php?title=%D0%93%D1%83%D1%82%D0%BA%D0%BD%D0%B5%D1%85%D1%82,_%D0%AE%D1%80%D0%B3&action=edit&redlink=1). 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**,](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) 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"**

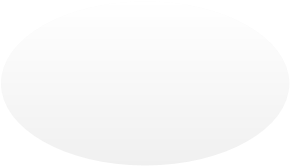
Основная неприятность, связанная с использованием виртуальных машин, заключается в том, что обычно время выполнения интерпретируемой программы значительно больше, чем время работы программы, оттранслированной в "родной" машинный код. Для того, чтобы увеличить скорость работы приложений, была разработана 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).



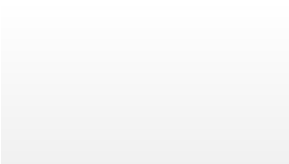
Байт

-

код

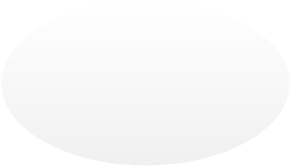


Компилятор



Программа на

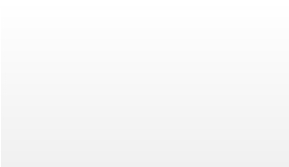
исходном языке



JIT

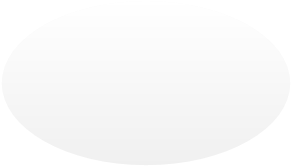
-

компилятор



Данные

Результаты



Счет

## 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 байткодами). 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-InTime-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

*Выделяются методы непрямого и прямого лексического анализа.*

***Непрямой*** *лексический анализ, или лексический анализ с возвратами, заключается в последовательной проверке версий о классах лексем. Если проверка текущей версии не подтверждается, то происходит откат назад по цепочке символов и осуществляется проверка следующей версии.*

***Прямой*** *лексический анализ позволяет определить значение лексемы без откатов назад по цепочке символов.*

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]](#footnote-2)), 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>

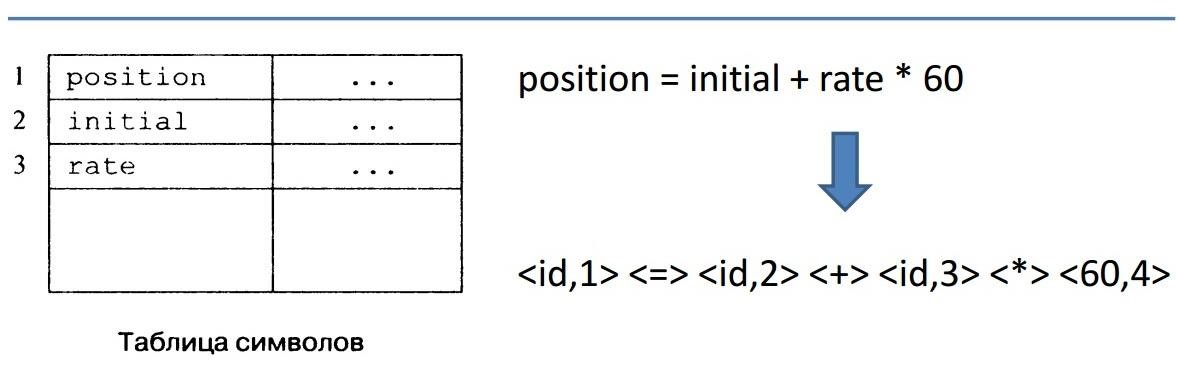


Рис. 2.9 Создание токенов и таблицы лексем

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

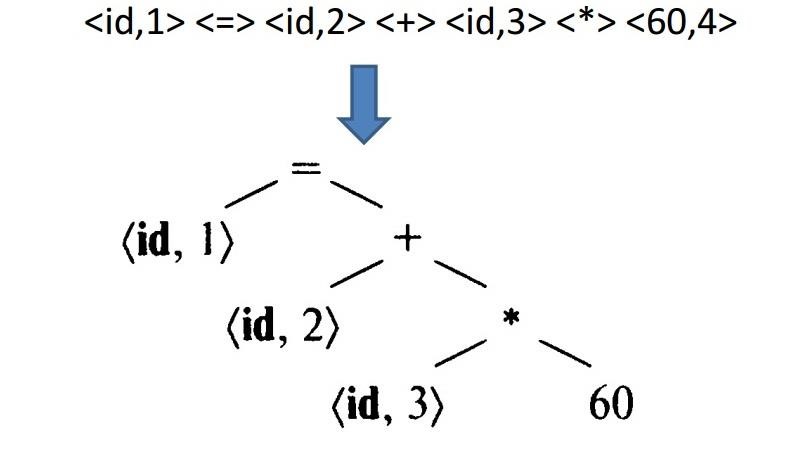


Рис. 2.10 Синтаксическое дерево грамматической структуры потока токенов

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.

Лексический

анализ

***position = position + rate\*60;***

Лексический

анализ

***id1 = id2 + id3\*60;***

### 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.

### **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.

Синтаксический

анализ

***id1 = id2 + id3\*60;***

Syntax analysis

=

+

\*

60

id1

id2

id3

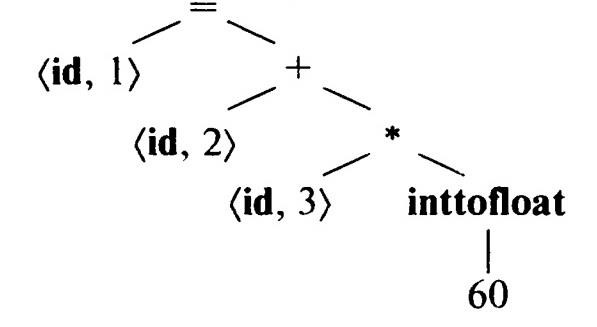
## Рис. 2.13 2.2.3. Семантический (видозависимый) анализ

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.



Видозависимый

анал

из

=

+

\*

60

id1

id2

id3

Видозависимый

анализ

=

+

\*

int\_to\_real

id1

id2

id3

60

### 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.

Оптимизация

temp1 =

int\_to\_real(60)

temp2 = id3\*temp1

temp3 = id2 + temp2

id1 = temp3

temp1 = id3\* 60.0

id1 = id2 + temp1

Оптимизация

### 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).

Генерация

кода

id3

ldsfld

ldc.r8 60.

mul

temp

stloc

ldsfld id2

ldlo

с

temp

add

stsfld

id1

temp1 = id3\* 60.0

id1 = id2 + temp1

Генератор

кода

### 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.

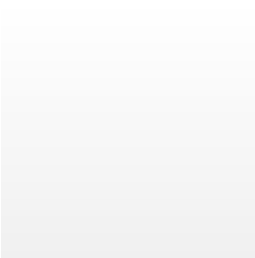
Желательно добиться как можно меньше просмотров, т.к. чтение программы на одном промежуточном языке и запись ее на другом промежуточном языке может занимать довольно много времени. С другой стороны, объединяя несколько фаз в один просмотр, мы должны помнить о том, что иногда мы не можем выполнить некоторую 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:

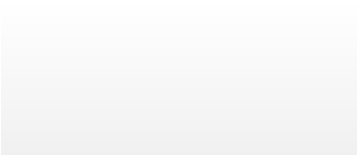


Исходная

программа

Язык

L

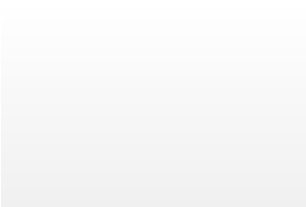


Блок

сканирования

за один

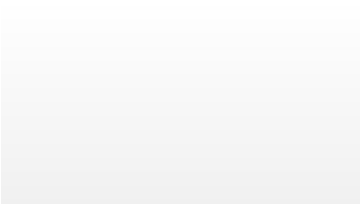
просмотр



Анализ

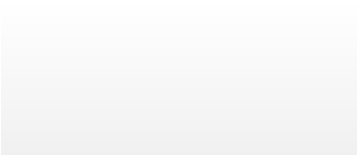
различных

таблиц



Синтаксический

анализатор



Генератор кода

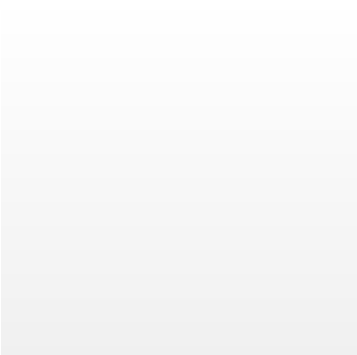


Семантика

Постфиксная

запись

Оптимизация



1

)

Объектный

код

+

объект

2

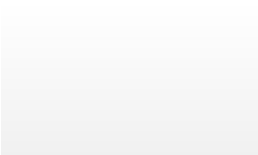
)

программа

3

)

…

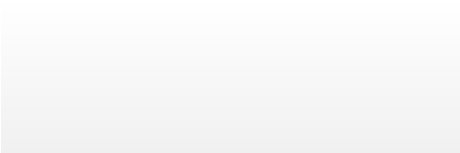


Таблицы

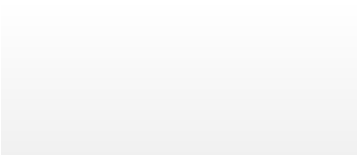
(

семантика

)

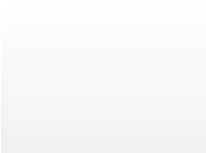


Сообщения об ошибках



Выполненная

программа



Редактор

связей

### 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.

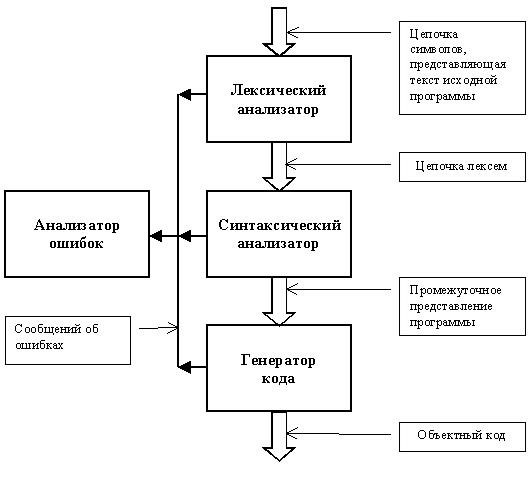


Рис. 2.18 Обобщенная структура компилятора

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.

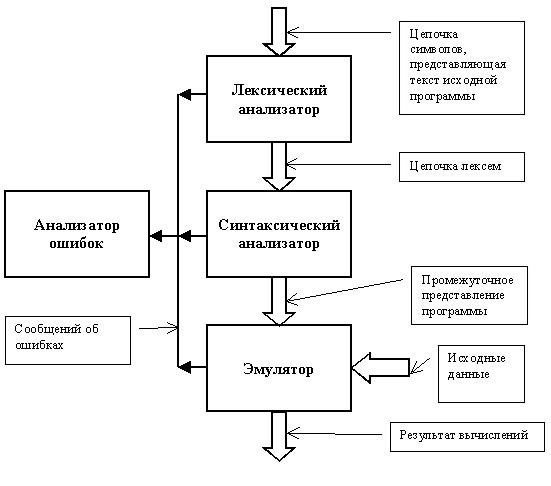


Рис. 2.19 Обобщенная структура интерпретатора

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

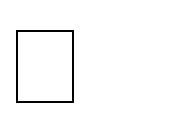
Распознаватель получает цепочку лексем и на ее основе осуществляет разбор в соответствии с используемыми правилами. Лексемы, при успешном разборе правил, передаются семантическому анализатору, который строит таблицу имен и фиксирует фрагменты синтаксической структуры. Кроме этого, между таблицей имен 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.

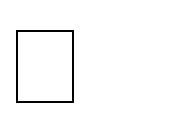
Анализатор ошибок получает информацию об ошибках, возникающих в различных блоках транслятора. Используя полученную информацию, он формирует сообщения пользователю. Кроме этого, данный блок может попытаться исправить ошибку, чтобы продолжить разбор дальше. На него также возлагаются действия, связанные 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

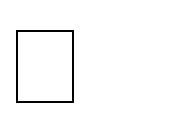
Данный вариант взаимодействия блоков, на примере компилятора, представлен на рис 2.21.

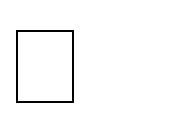


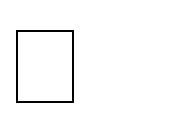
### Рис. 2.21 Многопроходная схема взаимодействия блоков компилятора

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

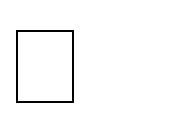
К достоинствам такого подхода можно отнести:

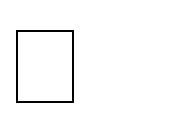
 Обособленность отдельных фаз, что позволяет обеспечить их независимую друг от друга реализацию и использование.

 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.

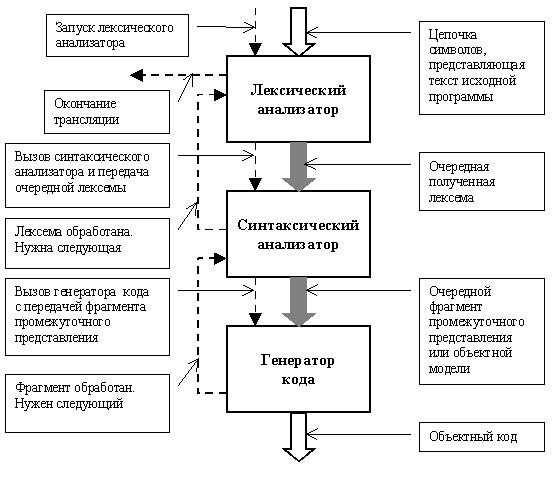


Рис. 2.22 Однопроходная схема взаимодействия блоков компилятора при управлении, инициируемом лексическим анализатором

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).

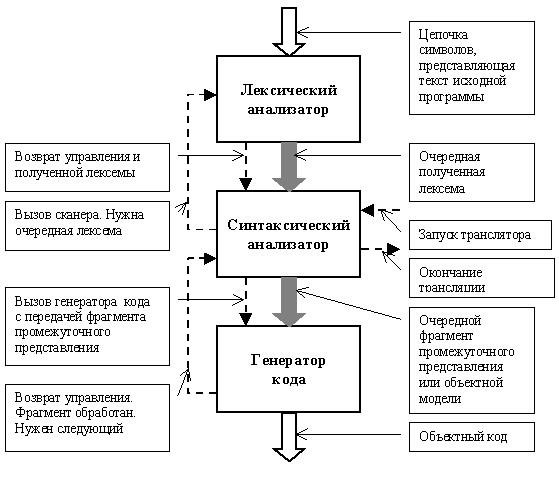


Рис. 2.23 Однопроходная схема взаимодействия блоков компилятора при управлении, инициируемом синтаксическим анализатором

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.

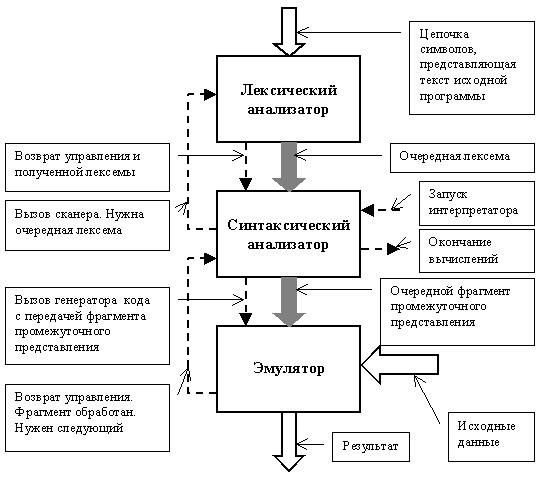


Рис. 2.24 Однопроходная схема взаимодействия блоков интерпретатора

## 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.

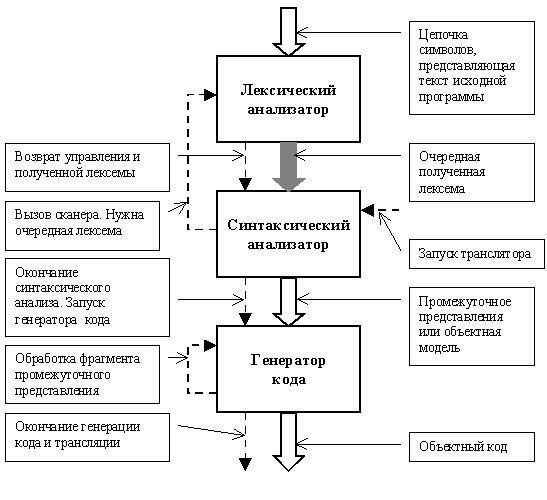


Рис. 2.25 2-х проходная схема с генерацией полного промежуточного представления

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.

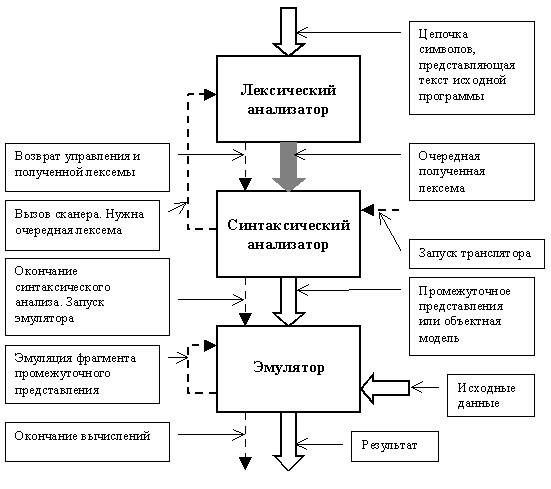


Рис. 2.26 Эмуляция промежуточного представления

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

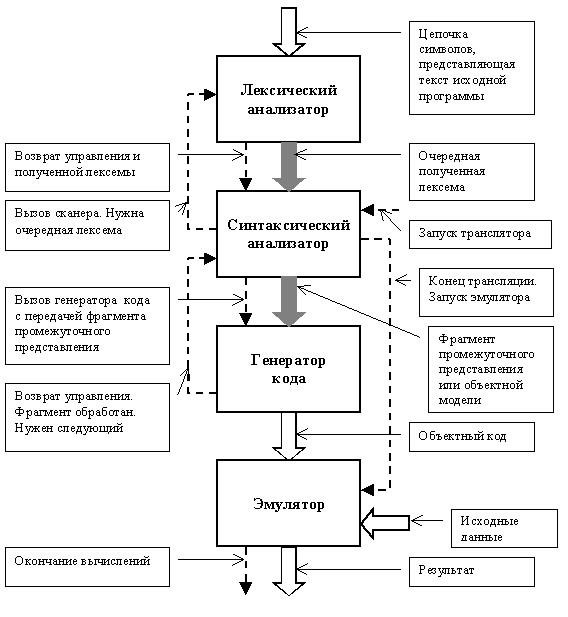


Рис. 2.27 Эмуляция скомпилированного объектного кода



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.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как стол

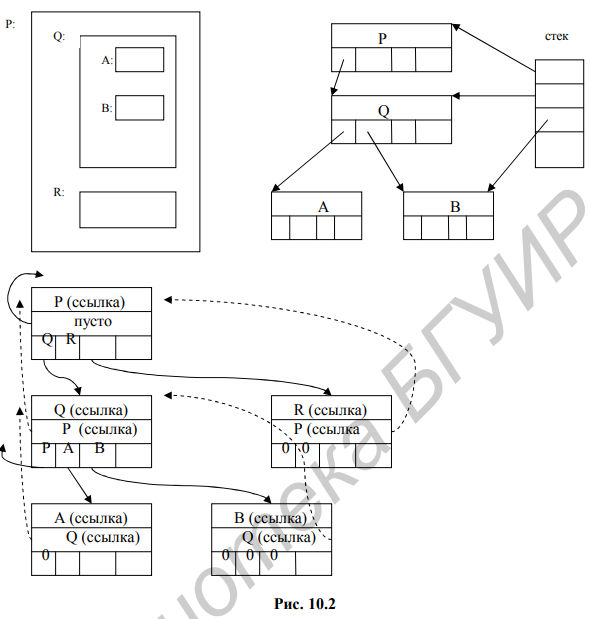
Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание  
Изображение выглядит как текст

Автоматически созданное описание  
Изображение выглядит как текст, человек, снимок экрана

Автоматически созданное описание  
Изображение выглядит как стол

Автоматически созданное описание  
  
Изображение выглядит как текст

Автоматически созданное описание

**МОДУЛЬ 2**

# Раздел 3. Регулярные грамматики

*Классы = леволинейная, праволинейная. Один пример ниже:*

Рассмотрим методы и средства, которые обычно используются при построении лексических анализаторов. В основе таких анализаторов лежат регулярные грамматики, поэтому рассмотрим грамматики этого класса более подробно.

**Соглашение:** в дальнейшем, если особо не оговорено, под **регулярной грамматикой будем понимать леволинейную автоматную грамматику** *G* = ⧼ *T*, *N*, *P*, *S* ⧽ без пустых правых частей. Напомним, что в такой грамматике каждое правило из *Р* имеет вид *A* → *Bt* либо *A* → *t*, где *A*, *B* ∈

*N*, *t* ∈ *T*.

**Соглашение:** предположим, что анализируемая цепочка заканчивается специальным символом ┴ - *признаком конца цепочки*.

**Для грамматик этого типа существует алгоритм определения того, принадлежит ли анализируемая цепочка языку, порождаемому этой грамматикой (*алгоритм разбора*).**

**3.1 Разбор по регулярным грамматикам**

Алгоритм разбора леволинейной автоматной грамматики:

1. первый символ исходной цепочки *a*1*a*2...*an*┴ заменяем нетерминалом *A*, для которого в грамматике есть правило вывода *A* → *a*1 (другими словами, производим свертку терминала *a*1 к нетерминалу *A*)
2. затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: полученный на предыдущем шаге нетерминал *A* и расположенный непосредственно справа от него очередной терминал *ai*исходной цепочки заменяем нетерминалом *B*, для которого в грамматике есть правило вывода *B* → *Aai* (*i* = 2, 3, ..., *n*); Это эквивалентно **построению дерева разбора методом снизу-вверх**:

на каждом шаге алгоритма строим один из уровней в дереве разбора, поднимаясь от листьев к корню.

1. первый символ исходной цепочки *a*1*a*2...*an*┴ заменяем нетерминалом *A*, для которого в грамматике есть правило вывода *A* → *a*1 (другими словами, производим свертку терминала *a*1 к нетерминалу *A*)
2. затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: полученный на предыдущем шаге нетерминал *A* и расположенный непосредственно справа от него очередной терминал *ai*исходной цепочки заменяем нетерминалом *B*, для которого в грамматике есть правило вывода *B* → *Aai* (*i* = 2, 3, ..., *n*);

Это эквивалентно построению дерева разбора методом снизу-вверх: на каждом шаге алгоритма строим один из уровней в дереве разбора, поднимаясь от листьев к корню.

При работе этого алгоритма возможны следующие ситуации:

1. прочитана вся цепочка; на каждом шаге находилась единственная нужная свертка; на последнем шаге свертка произошла к символу *S*. Это означает, что исходная цепочка *a*1*a*2...*an* ┴ ∈ *L*(*G*).
2. прочитана вся цепочка; на каждом шаге находилась единственная нужная свертка; на последнем шаге свертка произошла к символу, отличному от *S*. Это означает, что исходная цепочка *a*1*a*2...*an* ┴ ∉*L*(*G*).
3. на некотором шаге не нашлось нужной свертки, т. е. для полученного на предыдущем шаге нетерминала *A* и расположенного непосредственно справа от него очередного терминала *ai* исходной цепочки не нашлось нетерминала *B*, для которого в грамматике было бы правило вывода *B* → *Aai*. Это означает, что исходная цепочка *a*1*a*2...*an* ┴ ∉ *L*(*G*). на некотором шаге работы алгоритма оказалось, что есть более одной подходящей свертки, т. е. в грамматике разные нетерминалы имеют правила вывода с одинаковыми правыми частями, и поэтому непонятно, к какому из них производить свертку. Это говорит о***недетерминированности разбора****.*

Допустим, что разбор на каждом шаге детерминированный.

**Лексический анализ можно выполнить на основе таблицы не терминальных и терминальных символов или диаграммы состояний**

Для того, чтобы быстрее находить правило с подходящей правой частью, зафиксируем все возможные свертки (это определяется только грамматикой и не зависит от вида анализируемой цепочки).

## 3.2 Таблица лексического анализа

Создадим таблицу, столбцы которой помечены терминальными символами. Таблица фиксирует все возможные свертки.

Первая строка помечена символом *Н* (*Н*∉*N*), а значение каждого элемента этой строки - это нетерминал, к которому можно свернуть помечающий столбец терминальный символ.

Остальные строки помечены нетерминальными символами грамматики. Значение каждого элемента таблицы, начиная со второй строки — это нетерминальный символ, к которому можно свернуть пару «нетерминалтерминал», которыми помечены соответствующие строка и столбец.

Например, для леволинейной грамматики *Gleft* = ⧼{*a*, *b*, ┴}, {*S*, *A*, *B*, *C*},

*P*, *S* ⧽, где *P*:

*S →* *C*┴

*C → Ab* | *Ba*

1. *→ a* | *Ca*
2. *→ b* | *Cb*,  такая таблица будет выглядеть следующим образом:

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***a*** | ***b*** | ┴ |
| ***H*** | *A* | *B* | *—* |
| ***C*** | *A* | *B* | *S* |
| ***A*** | *—* | *C* | *—* |
| ***B*** | *C* | *—* | *—* |
| ***S*** | *—* | *—* | *—* |

Знак «*—*» ставится в том случае, если соответствующей свертки нет.

Но чаще информацию о возможных свертках представляют в виде *диаграммы состояний (ДС)* - неупорядоченного ориентированного помеченного графа. Граф строится следующим образом.

## 3.3 Диаграмма состояний (ДС) лексического анализа

**Диаграмма состояний (ДС)**- неупорядоченный ориентированный помеченный граф.

Граф строится следующим образом:

1. строим вершины графа, помеченные нетерминалами грамматики (для каждого нетерминала - одну вершину), и еще одну вершину, помеченную символом, отличным от нетерминальных, например, *H*. Эти вершины будем называть *состояниями*. *H* — начальное состояние.
2. соединяем эти состояния дугами по следующим правилам:

а) для каждого правила грамматики вида *W* → *t* соединяем дугой

состояния *H* и *W* (от *H* к *W*) и помечаем дугу символом *t*;

б) для каждого правила *W* → *Vt* соединяем дугой состояния *V* и *W* (от *V*

к *W*) и помечаем дугу символом *t*.

**Диаграмма состояний для грамматики *Gleft***



Алгоритм разбора по диаграмме состояний

1. объявляем текущим начальное состояние *H*;
2. затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: считываем очередной символ исходной цепочки и переходим из текущего состояния в другое состояние по дуге, помеченной этим символом. Состояние, в которое мы при этом попадаем, становится текущим.

При работе этого алгоритма возможны следующие ситуации (аналогичные ситуациям, которые возникают при разборе непосредственно по регулярной грамматике):

1. Прочитана вся цепочка; на каждом шаге находилась единственная дуга, помеченная очередным символом анализируемой цепочки; в результате последнего перехода оказались в состоянии *S*. Это означает, что исходная цепочка принадлежит *L*(*G*).
2. Прочитана вся цепочка; на каждом шаге находилась единственная «нужная» дуга; в результате последнего шага оказались в состоянии, отличном от *S*. Это означает, что исходная цепочка не принадлежит *L*(*G*).
3. На некотором шаге не нашлось дуги, выходящей из текущего состояния и помеченной очередным анализируемым символом. Это означает, что исходная цепочка не принадлежит *L*(*G*).

На некотором шаге работы алгоритма оказалось, что есть несколько дуг, выходящих из текущего состояния, помеченных очередным анализируемым символом, но ведущих в разные состояния. Это говорит о *недетерминированности разбора*.

**Диаграмма состояний** определяет **конечный автомат**, построенный **по регулярной грамматике**, который допускает множество цепочек, составляющих язык, определяемый этой грамматикой.

### **3.4 Детерминированный конечный** автомат (ДКА)

Различают детерминированные КА — автоматы, в которых следующее состояние однозначно определяется текущим состоянием и выход зависит только от текущего состояния и текущего входа, и недетерминированные КА, следующее состояние у которых в общем случае неопределённо и, соответственно, не определён выходной сигнал. Если переход в последующие состояния происходит с некоторыми вероятностями, то такой КА называют вероятностным КА.

***Детерминированный* *конечный автомат* (*ДКА***) - это пятерка ⧼ ***K*,*T*, δ,**

***H*, *S*** ⧽, где

***K*** — конечное множество состояний;

***T*** — конечное множество допустимых входных символов; **δ** — отображение множества *K* ✕ *T* в *K*, определяющее поведение

автомата;

***H*** ∈*K* — начальное состояние;

***S*** ∈*K* — заключительное состояние (либо множество заключительных состояний *S* ⊆ *K*).

Замечания к определению ДКА:

А) Заключительных состояний в ДКА может быть более одного, однако для любого регулярного языка, все цепочки которого заканчиваются маркером конца (┴), существует ДКА с единственным заключительным состоянием. Заметим также, что ДКА, построенный по регулярной грамматике рассмотренным выше способом, всегда будет иметь единственное заключительное состояние *S.*

В) Отображение δ: *K* ✕ *T* → *K* называют ***функцией переходов* ДКА**. δ (*A*, *t*) = *B* означает, что из состояния *A* по входному символу *t* происходит переход в состояние *B*. Иногдаδопределяют лишь на подмножестве *K* ✕ *T* (частичная функция). Если значение δ (*A*, *t*) не определено, то автомат не может дальше продолжать работу и останавливается в состоянии «ошибка».

**Диаграмма состояний** определяет **конечный автомат**, построенный **по регулярной грамматике**, который допускает множество цепочек, составляющих язык, определяемый этой грамматикой.

Состояния и дуги ДС — это графическое изображение функции переходов конечного автомата из состояния в состояние при условии, что очередной анализируемый символ совпадает с символом-меткой дуги.

Среди всех состояний выделяется начальное (считается, что в начальный момент своей работы автомат находится в этом состоянии) и **заключительное** (если автомат завершает работу переходом в это состояние, то анализируемая цепочка им допускается). На ДС эти состояния отмечаются короткими входящей и соответственно исходящей стрелками, не соединенными с другими вершинами.

**Определение:** ДКА *допускает цепочку* *a*1*a*2...*an*, если δ (*H*, *a*1) = *A*1; δ (*A*1, *a*2) = *A*2; …; δ (*An* − 2, *an* − 1) = *An* − 1; δ (*An* − 1, *an*) = *S*,

где *ai* ∈ *T*, *Aj* ∈ *K*, *j* = 1, 2, ..., *n* - 1; *i* = 1, 2, ..., *n*. *H* - начальное состояние, *S* - заключительное состояние.

**Определение:** множество цепочек, допускаемых ДКА, составляет определяемый им ***язык***.

Для более удобной работы с диаграммами состояний введем несколько соглашений:

* если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то будем изображать одну дугу, помечая ее списком из всех таких символов;
* непомеченная дуга будет соответствовать переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния;
* введем состояние ошибки (*ER*); переход в это состояние будет означать, что исходная цепочка языку не принадлежит.

По диаграмме состояний легко написать анализатор для регулярной грамматики.

Рассмотрим алгоритм анализа для праволинейной грамматики

*Gright* = ⧼{*a*, *b*, ┴}, {*H*, *A*, *B*, *C*}, *P*, *H* ⧽

на примере цепочки *abba*┴.

При анализе данной цепочки получим следующую последовательность переходов в ДС:



Вспомним, что каждый переход в ДС означает свертку сентенциальной формы путем замены в ней пары «нетерминал-терминал» *Nt* на нетерминал *L*, где *L → Nt* правило вывода в грамматике. Такое применение правила в обратную сторону будем записывать с помощью обратной стрелки *Nt* ← *L* (обращение правила вывода). Тогда получим следующую последовательность сверток, соответствующую переходам в ДС:

*abba*┴ ← ***A****bba*┴ ← ***C****ba*┴ ← ***B****a*┴ ← ***C***┴ ← ***S***

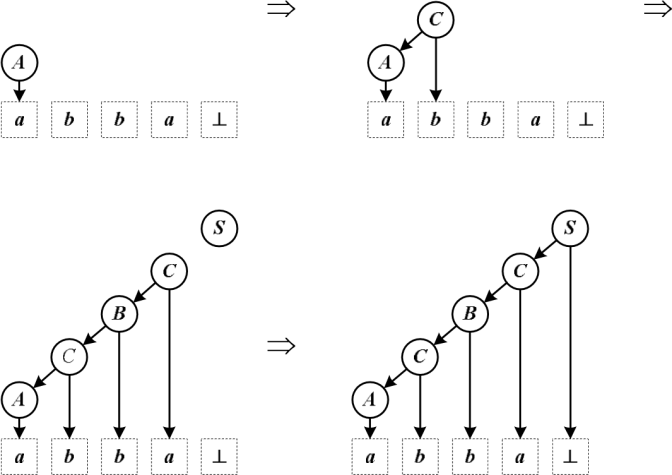
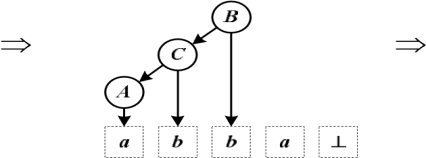
Эта последовательность не что иное, как обращение (правого) вывода цепочки *abba*┴ в грамматике *G*. Она соответствует построению дерева снизувверх.

➾

Построение дерева вывода снизу

-

вверх.



Разбор по праволинейной грамматике.

По диаграмме состояний построим праволинейную автоматную грамматику *Gright* следующим способом:

* нетерминалами будут состояния из ДС (кроме *S*);
* каждой дуге из состояния *V* в заключительное состояние *S* (помеченной признаком конца ┴) будет соответствовать правило *V* → ┴;
* каждой дуге из состояния *V* в состояние *W*, помеченной символом *t*, будет соответствовать правило *V* → *tW*;  начальное состояние *H* объявляется начальным символом грамматики.

*Gright* = ⧼{*a*, *b*, ┴}, {*H*, *A*, *B*, *C*}, *P*, *H* ⧽

*P*:

*H → aA* | *bB*

*A → bC*

*C → bB* | *aA* | ┴

*B → aC*

Заметим, что *L*(*Gright*) = *L*(*Gleft*), так как грамматики *Gright*и *Gleft*соответствуют одной и той же ДС

Нетрудно описать и обратный алгоритм для праволинейной автоматной грамматики, если все ее правила с односимвольной правой частью имеют вид *V* → ┴. Состояниями ДС будут нетерминалы грамматики и еще одно специальное заключительное состояние *S*, в которое для каждого правила вида *V* → ┴ проводится дуга из *V*, помеченная признаком конца ┴. Для каждого правила вида V → t W проводится дуга из V в W, помеченная символом t. Начальным состоянием в ДС будет начальный символ H.

### 3.5 О недетерминированном разборе

**Определение**.

***Недетерминированный конечный автомат (НКА)*** — это пятерка ⧼***K*,**

***T*, δ, *H*, *S***⧽, где

*K* — конечное множество состояний;

*T* — конечное множество допустимых входных символов; **δ** — отображение множества *K* ✕ *T* в множество подмножеств *K*;

***H*** ⊆*K* — конечное множество начальных состояний; *S* ⊆ *K* — конечное множество заключительных состояний.

**δ** (*A*, *t*) = {*B*1, *B*2,..., *Bn*} означает, что из состояния *A* по входному

символу *t* можно осуществить переход в любое из состояний *Bi*, *i* = 1, 2, ..., *n*. В алгоритме разбора по ДС для леволинейной грамматике было отмечено, что несколько нетерминалов могут иметь одинаковые правые части, и поэтому неясно, к какому из них делать свертку.

При анализе по праволинейной грамматике может оказаться, что нетерминал имеет две правые части с одинаковыми терминальными символами и поэтому неясно, на какую альтернативу заменить нетерминал.

В терминах диаграммы состояний эти ситуации означают, что из одного состояния выходит несколько дуг, ведущих в разные состояния, но помеченных одним и тем же символом.

В алгоритме разбора по ДС для леволинейной грамматике было отмечено, что несколько нетерминалов могут иметь одинаковые правые части, и поэтому неясно, к какому из них делать свертку. (см. ситуацию 4 в описании алгоритма).

Для грамматики *G* = ⧼ {*a*, *b*, ┴}, {*S*, *A*, *B*}, *P*, *S* ⧽, где

*P*:

*S → A*┴

1. *→ a* | *Bb*
2. *→ b* | *Bb*  разбор будет недетерминированным (т.к. у нетерминалов *A* и *B* есть одинаковые правые части — *Bb*).

Такой грамматике будет соответствовать недетерминированный конечный автомат.

Любой НКА, также как и ДКА, можно представить в виде таблицы (в одной ячейке такой таблицы можно указывать сразу несколько состояний, в которые возможен переход из заданного состояния по текущему символу) или в виде диаграммы состояний (ДС).

В ДС каждому состоянию из множества *K* соответствует вершина; из вершины *A* в вершину *B* ведет дуга, помеченная символом *t*, если *B*  **δ** (*A*, *t*).

(В НКА из одной вершины могут исходить несколько дуг с одинаковой пометкой). ***Успешный путь***- это путь из начальной вершины в заключительную;***пометка пути*** *-* это последовательность пометок его дуг.

*Язык,* допускаемый НКА, - это множество пометок всех успешных путей.

Если начальное состояние автомата (НКА или ДКА) одновременно является и заключительным, то автомат допускает пустую цепочку ɛ. **Замечание**

Автомат, построенный по регулярной грамматике без пустых правых частей, не допускает ɛ.

Для построения разбора по регулярной грамматике в недетерминированном случае можно предложить алгоритм, который будет перебирать все возможные варианты сверток (переходов) один за другим; если цепочка принадлежит языку, то будет найден успешный путь; если каждый из просмотренных вариантов завершится неудачей, то цепочка языку не принадлежит. Однако такой алгоритм практически неприемлем, поскольку при переборе вариантов мы, скорее всего, снова окажемся перед проблемой выбора и, следовательно, будем иметь «дерево отложенных вариантов» и экспоненциальный рост сложности разбора.

**Один из наиболее важных результатов теории конечных автоматов** состоит в том, что класс языков, определяемых недетерминированными конечными автоматами, совпадает с классом языков, определяемых детерминированными конечными автоматами.

**Утверждение**. Пусть *L* - формальный язык. Следующие утверждения эквивалентны:

1. *L* порождается регулярной грамматикой;
2. *L* допускается ДКА;
3. *L* допускается НКА.

Эквивалентность пунктов (1) и (2) следует из рассмотренных выше алгоритмов построения конечного автомата по регулярной грамматике и обратно — грамматики по автомату. Очевидно, что из (2) следует (3): достаточно записать вместо каждого перехода ДКА **δ** (*C*, *a*) = *b* эквивалентный ему переход в НКА **δ** (*C*, *a*) = {*b*}, начальное состояние ДКА поместить в множество начальных состояний НКА, а заключительное состояние ДКА поместить в множество заключительных состояний НКА. Далее рассмотрим алгоритм построения ДКА, эквивалентного НКА, - он обосновывает то, что из (3) следует (2).

### 3.6 Алгоритм построения ДКА по НКА

**Вход:** НКА *M* = ⧼ *K*, *T*, **δ**, *H*, *S* ⧽.

**Выход:** ДКА *M*1 = ⧼ *K*1, *T*, **δ** 1, *H*1, *S*1 ⧽, допускающий тот же язык, что и автомат *М*: *L*(*M*) = *L*(*M*1).

**Метод:**

1. Элементами *K*1, т. е. состояниями в ДКА, будут некоторые подмножества множества состояний НКА. Заметим, что в силу конечности множества *K*, множество *K*1 также конечно и имеет не более 2s элементов, где *s* — мощность *K*.

Подмножество {*А*1, *A*2, …, *An*} состояний из *К* будем для краткости записывать как *A*1*A*2*...An*. Множество *K*1 и переходы, определяющие функцию **δ**1, будем строить, начиная с состояния *H*1: *H*1 := *A*1*A*2*...An*, где *А*1, *A*2, …, *An* ∈ *H.* Другими словами, все начальные состояния НКА *M* объединяются в одно состояние *H*1 для ДКА *M*1. Добавляем в множество *K*1 построенное начальное состояние *H*1 и пока считаем его нерассмотренным (на втором шаге оно рассматривается и строятся остальные состояния множества *K*1, а также переходы **δ**1 .)

1. Пока в *K*1 есть нерассмотренный элемент *A*1*A*2*...Am*,

«рассматриваем» его и выполняем для каждого *t* ∈ *T* следующие действия:

* Полагаем **δ**1 (*A*1*A*2*...Am*, *t*) = *B1B2...Bk*, где для 1 ≤ *j* ≤ *k* в НКА **δ** (*Ai*, *t*) = *Bj*для некоторых 1 ≤ *i* ≤ *m*. Другими словами, *B*1*B*2*...Bk* — это множество всех состояний в НКА, куда можно перейти по символу *t* из множества состояний *A*1*A*2*...Am*. В ДКА *M*1 получается детерминированный переход по символу *t* из состояния *A*1*A*2*...Am* в состояние *B*1*B*2*...Bk*. (Если *k* =0, то полагаем **δ**1 (

*A*1*A*2*...Am*, *t* ) = ∅ ).

* Добавляем в *K*1 новое состояние *B*1*B*2*... Bk .*

Шаг 2 завершается, поскольку множество новых состояний *K*1 конечно.

3. Заключительными состояниями построенного ДКА *M*1 объявляются все состояния, содержащие в себе хотя бы одно заключительное состояние НКА *M*:

**Пояснения к алгоритму построения ДКА по НКА**

Множество *S*1 построенного ДКА может состоять более, чем из одного элемента. Не для всех регулярных языков существует ДКА с единственным заключительным состоянием (пример: язык всех цепочек в алфавите {*a*, *b*}, содержащих не более двух символов *b*). Однако для реализации алгоритма детерминированного разбора заключительное состояние должно быть единственным. В таком случае изменяют входной язык, добавляя маркер ┴ в конец каждой цепочки (на практике в роли маркера конца цепочки ┴ может выступать признак конца файла, символ конца строки или другие разделители). Вводится новое состояние *S*, и для каждого состояния *Q* из множества *S*1 добавляется переход по символу ┴: **δ**1 (*Q*,┴) = *S*. Состояния из *S*1 больше не считаются заключительными, а *S* объявляется единственным заключительным состоянием. Теперь по такому ДКА можно построить автоматную грамматику, допускающую детерминированный разбор. Проиллюстрируемработуалгоритманапримерах.

**Пример 1.**

Пусть имеется НКА *M* = ⧼{ *H*, *A*, *B*, *S* }, {0, 1}, **δ**, {*H* }, {*S* } ⧽, где **δ** (*H*, 1) = {*B*}

**δ** (*A*, 1) = {*B,* *S*}

**δ** (*B*, 0) = {*A*}

*L*(*M*) = {1(01)n| *n* ≥ 1}.

*H*

*B*

*S*

*A*

1

1

0

1

ДС для автомата *M* *.*

Грамматика, соответствующая *M*:

*S → A*1

1. *→ B*0
2. *→ A*1 | 1

Построим ДКА по НКА, пользуясь предложенным алгоритмом. Начальным состоянием будет *H*.

**δ** 1(*Н*, 1) = *B* **δ** 1(*B*, 0) = *A* **δ** 1(*A*, 1) = *BS* **δ** 1(*BS*, 0) = *A*

Заключительным состоянием построенного ДКА является состояние *BS*.

Таким образом, *M*1 = ⧼{*H*, *B*, *A*, *BS*}, {0, 1}, **δ**1, *H*, *BS* ⧽. Для удобства переименуем состояния в *M*1: *BS* обозначается теперь как *S*1, а в однобуквенных именах состояний вместо подчеркивания используется индекс 1.

Тогда *M*1 = ⧼{*H*1, *B*1, *A*1, *S*1}, {0, 1}, {**δ**1(*Н*1, 1) = *B*1; **δ**1 (*B*1, 0) = *A*1; **δ**1 (*A*1, 1) = *S*1; **δ**1 (*S*1, 0) = *A*1}, *H*1, *S*1⧽.

Грамматика, соответствующая *M*1:

*S*1 → *A*11

*A*1 *→* *S*10 | *B*10

*B*1 *→* 1

Построим диаграмму состояний (рис. 5).

Рис.

5

. Диаграмма состояний

*M*

1

.

1

0

1

0

*H*

1

*B*

1

*A*

1

*S*

1

**Пример 2.**

Дана регулярная грамматика *G* = ⧼ *T*, *N*, *P*, *S* ⧽

*P*:

*S → Sb* | *Aa*

| *a*

*A → Aa* | *Sb*

| *b*,

которой соответствует НКА. С помощью преобразования НКА в ДКА построить грамматику, по которой возможен детерминированный разбор.

Построим функцию переходов НКА:

**δ** (*H*, *a*) = {*S*} **δ** (*H*, *b*) = {*A*} **δ** (*A*, *a*) = {*A*, *S*} **δ** (*S*, *b*) = {*A*, *S*}

Процесс построения функции переходов ДКА удобно изобразить в виде таблицы, начав ее заполнение с начального состояния *H*, добавляя строки для вновь появляющихся состояний:

|  |  |
| --- | --- |
|  | Переходы ДКА:  **δ** 1(*H*, *a*) = *S* **δ** 1(*H*, *b*) = *A* **δ** 1 (*S*, *b*) = *AS* **δ** 1(*A*, *a*) = *AS* **δ** 1(*AS*, *a*) = *AS* **δ** 1(*AS*, *b*) = *AS* |

Переобозначим сотояния: *A* ➾*A, H* ➾ *H, AS* ➾ *Y*, *S* ➾ *X*.

Два заключительных состояния *X* и *Y* сведем в одно заключительное состояние *S*, используя маркер конца цепочки ┴. После такой модификации получаем ДКА с единственным заключительным состоянием *S* и функцией переходов:

**δ** 1(*H*, *a*) = *X* **δ** 1(*H*, *b*) = *A* **δ** 1(*X*, *b*) = *Y* **δ** 1(*X*, =) ┴ *S* **δ** 1(*A*, *a*) = *Y* **δ** 1(*Y*, *a*) = *Y* **δ** 1(*Y*, *b*) = *Y* **δ** 1(*Y*, ┴) = *S*

По ДКА строим грамматику *G*1, позволяющую воспользоваться алгоритмом детерминированного разбора: *G*1:

*S → X*┴ | *Y*┴

*Y → Ya* | *Yb* | *Aa* | *Xb*

*X → a*

*A → b*Методы построения конечных автоматов, которые смог нагуглить:   
Эвристические методы генерации конечных автоматов

Методы генерации, основанные на метаэвристических алгоритмах

Методы генерации, основанные на сведении к задачам из класса трудных в NP

Метод генерации по безошибочным/зашумленным обучающим словарям

По безошибочным сценариям работы

Короче диссертация на эту тему: <https://isu.ifmo.ru/pls/apex/f?p=2109:0:0:DWNLD_F:NO::FILE,FDIS:FE9AC9B6106B63606A2A7BA2C86884E0,M>

### 3.7 Регулярные выражения

Регулярки в контексте фронта = паттерны для email/масок телефона и проч. Чуть глубже = могут использоваться для корректного мапания JSONов, XML и прочих стандартизированных форматов. Но лучше этого не делать :)

**Регуля́рные выраже́ния** [(англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *regular expressions*) — формальный язык поиска и осуществления манипуляций с [подстроками](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0) в тексте, основанный на использовании метасимволов ([символов-джокеров,](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D1%8B-%D0%B4%D0%B6%D0%BE%D0%BA%D0%B5%D1%80%D1%8B) [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *wildcard characters*). По сути это строка-образец ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *pattern*, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска.

Для задания любого регулярного языка над заданным алфавитом есть еще один широко используемый в математических теориях и приложениях формализм, задающий регулярные языки. Это регулярные выражения. Они позволяют описать любой регулярный язык над заданным алфавитом, используя три вида операций: + (**объединение**), · (**конкатенация**), \* (**итерация**).

**Определение** Пусть *L*, *L*1, *L*2 - языки над алфавитом Σ.

Тогда будем называть:

* **язык *L*1** ∪ ***L*2** *объединением* языков *L*1 и *L*2;
* **язык *L*1·*L*2 = {φ·ψ |φ** ∈***L*1, ψ** ∈***L***2} - *конкатенацией* (*сцеплением*) языков *L*1 и *L*2 (содержит всевозможные цепочки, полученные сцеплением цепочек из *L*1 с цепочками из *L*2);
* ***i-ой степенью* языка *L***назовем язык *L*i = *L*i-1·*L* для *i* > 0, *L*0 = {**ɛ**}.

Язык назовем *итерацией* языка *L*.

**Определение.** Пусть Σ - алфавит, не содержащий символов \*, +, **ɛ**, ∅, (,). Определим рекурсивно *регулярное выражение* **γ** над алфавитом Σ и регулярный язык *L*(**γ**), задаваемый этим выражением:

1. *a* ∈ Σ ∪ {**ɛ**, ∅} есть регулярное выражение; *L*(*a*) = {*a*} для *a* ∈ Σ ∪ {**ɛ**}; *L*(∅) = ∅;
2. если **α** и **β** - регулярные выражения, то:

а) (**α**) + (**β**) - регулярное *L* ((**α**) + (**β**)) = *L* (**α**)∪ *L*(**β**); выражение;

б) (**α**) · (**β**) - регулярное *L* ((**α**) · (**β**) = *L* (**α**) · *L*(**β**); выражение;

в) (**β**)\* - регулярное выражение; *L* ((**β**)\*) = (*L* (**β**))\*;

1. все регулярные выражения конструируются только с помощью пунктов 1 и 2.

Если считать, что операция «+» имеет самый низкий приоритет, а операция «\*» - наивысший, то в регулярных выражениях можно опускать лишние скобки.

**Примеры** регулярных выражений над алфавитом {*a*, *b*}: *a* + *b*, (*a* + *b*)\*, (*aa* (*ab*) \**bb*) \*.

Соответствующие языки:

*L*(*a* + *b* ) = {*a*} ∪ {*b*} = {*a*, *b*},

*L*((*a* + *b*)\* ) = {*a*, *b*}\*,

*L*((*aa* (*ab*)\**bb*)\*) = {**ɛ**}∪{ *aa x*1*bb aa x*2*bb*... *xkbb* | *k* ≥1, *xi* ∈ {(*ab*)*n* | *n* ≥0} для *i* = 1, ..., *k* }.

Существуют алгоритмы построения регулярного выражения по регулярной грамматике или конечному автомату и обратные алгоритмы, позволяющие преобразовать выражение в эквивалентную грамматику или автомат.

Регулярные выражения используются для описания лексем языков программирования, в задачах редактирования и обработки текстов; подходят для многих задач сравнения изображений и автоматического преобразования. Расширенные их спецификации (эквивалентные по описательной силе, но более удобные для практики) входят в промышленный стандарт открытых операционных систем *POSIX* и в состав базовых средств языка программирования ***Perl***.

### 3.8 Задачи лексического анализа

***Лексический анализ* (ЛА) - это первый этап процесса компиляции.**

Лексический анализ важен для процесса компиляции по нескольким причинам:

а) замена в программе идентификаторов, констант, ограничителей и служебных слов лексемами делает представление программы более удобным для дальнейшей обработки;

б) лексический анализ уменьшает длину программы, устраняя из ее исходного представления несущественные пробельные символы и

комментарии;

в) если будет изменена кодировка в исходном представлении программы, то это отразится только на лексическом анализаторе.

Обработка лексем в фазе лексического анализа

На этом этапе литеры, составляющие исходную программу, группируются в **отдельные лексические элементы**, называемые ***лексемами***.

Выбор конструкций, которые будут выделяться как отдельные лексемы, зависит от языка и от точки зрения разработчиков компилятора. Обычно принято выделять следующие типы лексем: идентификаторы, служебные слова, константы и ограничители.

Каждой лексеме сопоставляется пара:

⧼ ***тип лексемы, указатель на информацию о ней*** ⧽. **Лексический анализатор (ЛА)**:

* должен выделить в исходном тексте цепочку символов, представляющую лексему, и проверить правильность ее записи;
* зафиксировать в специальных таблицах для хранения разных типов лексем факт появления соответствующих лексем в анализируемом тексте;
* преобразовать цепочку символов, представляющих лексему, в пару ⧼ ***тип лексемы, указатель на информацию о ней*** ⧽;  удалить пробельные литеры и комментарии.

Очевидно, что лексемы можно описать с помощью регулярных грамматик. Поскольку лексемы не пусты, для описания лексического состава

любого языка программирования достаточно автоматных грамматик без **ɛ**правил.

Например, идентификатор (*I*) описывается так:

*I* → *a* | *b*| ...| *z* | *Ia* | *Ib* |...| *Iz* | *I*0 | *I*1 |...| *I*9

Целое без знака (*N*):

*N* → 0 | 1 |...| 9 | *N*0 | *N*1 |...| *N*9 и т. д.

Для грамматик этого класса, как мы уже видели, существует простой и эффективный алгоритм анализа того, принадлежит ли заданная цепочка языку, порождаемому этой грамматикой. Однако перед лексическим анализатором стоит более сложная задача:

* он должен сам выделить в исходном тексте цепочку символов, представляющую лексему, и проверить правильность ее записи;
* зафиксировать в специальных таблицах для хранения разных типов лексем факт появления соответствующих лексем в анализируемом тексте; преобразовать цепочку символов, представляющих лексему, в пару ⧼

*тип\_лексемы, указатель\_на\_информацию\_о\_ней* ⧽;  удалить пробельные литеры и комментарии.

Для описания лексем на основе регулярных грамматик при анализе с помощью диаграммы состояний, введем на дугах дополнительный вид пометок - **пометки-действия**. Теперь каждая дуга в ДС может выглядеть так:

*t* 1*,*  *t* 2*,…,*  *tn*  *B*

*A*

*D* 1;  *D* 2; *…* ;*D* *m*

Смысл *ti* прежний: если в состоянии *A* очередной анализируемый символ совпадает с *ti* для какого-либо *i* = 1, 2, …, *n*, то осуществляется переход в состояние *B*, при этом необходимо выполнить действия *D*1, *D*2, ..., *Dm*.

Пусть задана регулярная грамматика, описывающей целое число без знака: *S* → *N*┴

*N* → 0 | 1 |...| 9 | *N*0 | *N*1 |...| *N*9

Необходимо построить диаграмму с действиями по нахождению и печати квадрата данного числа.

*n*

=

*n*

\*

10

-

*c*

–

’0’;

*cout << n\*n*

*;*

┴

9

, 1,…,

0

*n*

=

*c*

–

’0’;

9

0

, 1,…,

*N*

*S*

*H*

Перевод числа во внутренне представление (переменная *n* типа *int*) по схеме Горнера: распознав очередную цифру, умножаем текущее значениие *n* на 10 и добавляем числовое значение текущей цифры (текущий символ хранится в переменной *c* типа *char*). Встретив маркер конца, печатаем квадрат числа *n.*

### 3.9 Лексический анализатор для М-языка

**Грамматика модельного языка (М- языка)**

*P →* ***program*** *D*1; *B*┴

*D*1 *→* ***var*** *D* {,*D*}

*D → I* {, *I*}: [ ***int*** | ***bool*** ]

*B →* ***begin*** *S* {;*S*} ***end***

*S → I* := *E* | ***if*** *E* ***then*** *S* ***else*** *S* | ***while*** *E* ***do*** *S* | *B* | ***read*** (*I*) | ***write*** (*E*)

*E → E*1 [ = | < | > | != ] *E*1 | *E*1

*E*1 *→ T* {[ + | - | ***or*** ] *T*}

*T → F* {[ \* | / | ***and*** ] *F*}

*F → I* | *N* | *L* | ***not*** *F* | (*E*)

*L →* ***true*** | ***false***

*I → C* | *IC* | *IR*

*N → R* | *NR*

*C → a* | *b* | ... | *z* | *A* | *B* | .. | *Z R →* 0 | 1 | 2 | ... |9

Замечания к грамматике модельного языка:

а) запись вида {α} означает итерацию цепочки α, т. е. в порождаемой цепочке в этом месте может находиться либо ɛ, либо α, либо α α, либо α α α и т. д.;

б) запись вида [α | β] означает, что в порождаемой цепочке в этом месте

может находиться либо α, либо β;

в) *P* - цель грамматики; символ ┴ - маркер конца текста программы. Контекстные условия:

1. Любое имя, используемое в программе, должно быть описано и только один раз.
2. В операторе присваивания типы переменной и выражения должны совпадать.
3. В условном операторе и в операторе цикла в качестве условия возможно только логическое выражение.
4. Операнды операции отношения должны быть целочисленными.
5. Тип выражения и совместимость типов операндов в выражении определяются по обычным правилам; старшинство операций задано синтаксисом.

В любом месте программы, кроме идентификаторов, служебных слов и чисел, может находиться произвольное число пробелов и комментариев в фигурных скобках вида {⧼ *любые символы, кроме* «}» и «┴»⧽ }.

***true***, ***false***, ***read*** и ***write*** — служебные слова (в М-языке их нельзя

переопределять в отличие от аналогичных стандартных идентификаторов в Паскале).

Разделители между идентификаторами, числами и служебными словами употребляются так же, как в Паскале.

На вход лексического анализатора поступает текст исходной программы на **М-языке**. Результат - исходная программа в виде последовательности внутреннего представления **лексем**.

Типы лексем

**TW** — таблица служебных слов М-языка;

**TD** — таблица ограничителей М-языка;

**TID** — таблица идентификаторов анализируемой программы.

Таблицы TW и TD заполняются заранее, т. к. их содержимое не зависит от исходной программы; TID формируется в процессе анализа. Содержательно внутреннее представление лексем - это пара (***тип\_лексемы, значение\_лексемы***).

Вход лексического анализатора - символы исходной программы на Мязыке; результат работы - исходная программа в виде последовательности лексем (их внутреннего представления). В нашем случае лексический анализатор будет вызываться, когда потребуется очередная лексема исходной программы, поэтому результатом работы лексического анализатора будет очередная лексема анализируемой программы на М-языке.

Соглашение об используемых таблицах лексем:

**TW** — таблица служебных слов М-языка;

**TD** — таблица ограничителей М-языка;

**TID** — таблица идентификаторов анализируемой программы;

Таблицы TW и TD заполняются заранее, т. к. их содержимое не зависит от исходной программы; TID формируется в процессе анализа.

Необходимые таблицы можно представить в виде объектов, конкретный вид которых мы рассмотрим чуть позже, или в виде массивов строк, например, для служебных слов.

**Диаграмма состояний (ДС) для М-языка.**

Лексический анализ может проводиться независимо от последующих этапов трансляции. В этом случае исходный файл с текстом программы преобразуется в последовательность лексем во внутреннем представлении согласно построенной ДС; затем эта последовательность подается на вход синтаксическому анализатору.

Мы реализуем другой подход: лексический анализатор выдает очередную лексему по требованию синтаксического анализатора и затем

«замирает», пока к нему не обратятся за следующей лексемой.

При таком подходе действие *Lex*(*k, l*)*;* в ДС означает ***return*** *Lex*(*k, l*)*;*.

Кроме того, вместо перехода в состояние ошибки ER генерируется исключение с указанием символа, который привел к ошибочной ситуации.

Перехватываться и обрабатываться исключения будут не в лексическом анализаторе, а в основной программе, содержащей анализатор. Обработка исключения состоит в выводе сообщения об ошибке и корректном завершении программы.

Непосредственно реализует ЛА по ДС функция get\_lex():

Изображение выглядит как ночное небо

Автоматически созданное описание

# 4. Синтаксический анализ

Синтаксический анализ основывается на разных принципах, и использует различные техники построения дерева вывода. Каждый метод предполагает свой способ построения по грамматике программыанализатора, которая будет осуществлять разбор цепочек. Корректный анализатор завершает свою работу для любой входной цепочки и выдает верный ответ о принадлежности цепочки языку. Анализатор некорректен, если:

· не распознает хотя бы одну цепочку, принадлежащую языку; · распознает хотя бы одну цепочку, языку не принадлежащую; · зацикливается на какой-либо цепочке.

Обычно для синтаксического разбора используют КС-грамматики, **правила которых имеют вид *A* → α, где *A*** ∈ ***N*, α** ∈ **(*T*** ∪ ***N*)**. Грамматики этого класса, с одной стороны, позволяют вполне адекватно описать синтаксис реальных языков программирования; с другой стороны, для разных подклассов КС-грамматик построены достаточно эффективные алгоритмы разбора, расходующие на обработку входной цепочки линейно сопостовимое время.

**Говорят, что метод анализа примени́м к данной грамматике, если анализатор, построенный в соответствии с этим методом, корректен.**

Из теории синтаксического анализа известно, что существует алгоритм, который по любой данной КС-грамматике и данной цепочке выясняет, принадлежит ли цепочка языку, порождаемому этой грамматикой. Но время работы такого алгоритма (синтаксического анализа с возвратами) экспоненциально зависит от длины цепочки, что с практической точки зрения совершенно неприемлемо.

Существуют табличные методы анализа, применимые ко всему классу

КС-грамматик и требующие для разбора цепочек длины n времени Cn3 (алгоритм Кока-Янгера-Касами), где C — константа, либо Cn2 (алгоритм Эрли). Их разумно применять только в том случае, если для интересующего нас языка не существует грамматики, по которой можно построить анализатор с линейной временной зависимостью от длины цепочки (такими языками могут быть, например, подмножества естественного языка).

При разработке языков программирования их синтаксис обычно стараются сделать таким, чтобы время на анализ было прямо пропорционально длине программы. Алгоритмы анализа, расходующие на обработку входной цепочки линейное время, применимы только к некоторым подклассам КС-грамматик.

Сущность синтаксического анализа достаточно сложна, чтобы излагать ее «на пальцах». Даже такое формализованное средство представления как конечный автомат является недостаточно «мощным» для этого. В самом первом приближении это можно показать, ссылаясь на вложенный характер конструкций языка. Элементы лексики представляют собой линейные последовательности, анализ которых и производится конечными автоматами. Синтаксис же предложения не является линейной структурой. Если определения элементов синтаксиса языка (выражения, операторы) являются теми единицами, из которых строится программа, то взаимоотношение этих единиц в конкретной программе может быть представлено в виде дерева. А с деревом тесно связаны такие понятия, как рекурсия и стек. Таким образом, интуитивно становится ясным, что для определения и анализа синтаксиса языка необходим математический аппарат, который допускает рекурсивное определение и порождение своих элементов, а при их анализе используются деревья, рекурсивные функции и работа со стеком.

**Формальные грамматики (ФГ) являются математическим аппаратом, который исследует свойства цепочек символов, порожденных заданным набором правил.**

Формальные грамматики (ФГ), и контекстно-свободные (*Из всех классов формальных грамматик только контекстно-свободные (КС-грамматики) продуктивно используются в синтаксическом анализе.)* и контекстнозависимые, как математический аппарат появились именно по «производственной необходимости» Представление синтаксиса в компиляторах и **автоматизации синтаксического анализа (СА) на основе схем синтаксически управляемых трансляций** по «производственной необходимости» опирается на математический аппарат формальных грамматик. В отличие от лексики и семантики, которые можно в принципе реализовать, использую только содержательные (неформальные) средства, синтаксический анализ почти невозможно выполнить, руководствуясь «здравым смыслом» и очевидностью. Необходим некоторый промежуточный уровень между синтаксисом языка и программной системой, его распознающей. Таковым уровнем и является ФГ.ФГ является не только средством представления синтаксиса, на ее основе создается инструментарий СА. Самая общая схема синтаксического анализатора, построенного на основе формальных методов.

**4.1 Cинтаксический разбор.**

**Классификация методов.**

**Синтаксический разбор (распознавание**) является первым этапом синтаксического анализа. Именно при его выполнении осуществляется подтверждение того, что входная цепочка символов является программой, а отдельные подцепочки составляют синтаксически правильные программные объекты. На этапе синтаксического анализа нужно:

1. установить, имеет ли цепочка лексем структуру, заданную синтаксисом языка, и
2. зафиксировать эту структуру.

Следовательно, снова надо решать задачу разбора: дана цепочка лексем, и надо определить, выводима ли она в грамматике, определяющей синтаксис языка. Если да, то построить вывод этой цепочки или дерево вывода. Однако структура таких конструкций как выражение, описание, оператор и т.п., более сложная, чем структура идентификаторов и чисел.

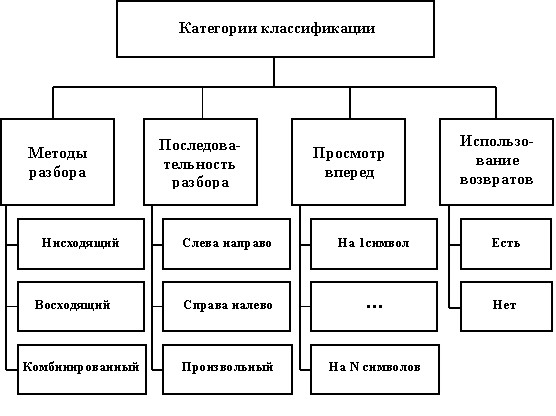
Вслед за распознаванием отдельных подцепочек можно осуществлять анализ их семантической корректности на основе накопленной информации. Затем проводится добавление новых объектов в объектную модель программы или в промежуточное представление.

Смысл синтаксического распознавания заключен в доказательстве того, что анализируемая входная цепочка принадлежит множеству цепочек, порождаемых грамматикой данного языка. Если подцепочка не является синтаксически правильным программным объектм, естественно на этом распознавание заканчивается. Выполнение синтаксического распознавания называется **распознаванием входной цепочки**. Цель доказательства в том, чтобы ответить на вопрос: принадлежит ли анализируемая цепочка множеству правильных цепочек заданного языка. Ответ "да" дается, если такая принадлежность установлена. Единственный ответ "нет" на любом уровне ведет к отрицанию принадлежности.

Чтобы получить ответ "да" относительно всей цепочки, надо его получить для каждого правила, обеспечивающего разбор отдельной подцепочки. Так как множество правил образуют иерархическую структуру, возможно с рекурсиями, то процесс получения общего положительного ответа можно интерпретировать как сбор по определенному принципу ответов для листьев, лежащих в основе дерева разбора, что дает положительный ответ для узла, содержащего эти листья. Далее анализируются обработанные узлы, и уже в них полученные ответы складываются в общий ответ нового узла. И так далее до самой вершины.

**Классификация методов синтаксического разбора**

Дан язык L(G) с грамматикой G = ⧼ T, N, P, S ⧽. Построить дерево разбора входной цепочки **a = a1a2a3...an.**



**Классификация методов организации синтаксического разбора**

Если попытаться формализовать задачу на уровне элементарного метаязыка, то она будет ставиться следующим образом. Дан язык L(G) с грамматикой G = ⧼ T, N, P, S ⧽. Построить дерево разбора входной цепочки **a**

**= a1a2a3...an.**

Естественно, что существует огромное количество путей решения данной задачи, и целью разработчика распознавателя является выделение приемлемых вариантов его реализации. Общая классификация рассматриваемых вариантов построения распознавателя представлена на слайде.

На самом верхнем уровне выделяются:  методы разбора;

 последовательность разбора;  использование просмотра вперед;  использование возвратов.

## Методы разбора

Выделяются два основных метода синтаксического разбора:  нисходящий разбор;  восходящий разбор.

Кроме этого можно использовать комбинированный разбор, сочетающий особенности двух предыдущих.

В теории автоматов, [автомат с магазинной памятью](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82_%D1%81_%D0%BC%D0%B0%D0%B3%D0%B0%D0%B7%D0%B8%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%D1%8E) — это конечный автомат, который использует стек для хранения состояний.

Сам принцип нисходящего и восходящего подхода широко

используются в различных областях человеческой деятельности, особенно в тех из них, которые связаны с анализом и синтезом искусственных систем. Например, методы разработки программного обеспечения сверху-вниз (нисходящий) и снизу-вверх (восходящий).

**Нисходящий разбор** заключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в заполнении промежутка между начальным нетерминалом и символами входной цепочки правилами, выводимыми из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом. Подставляемое правило в общем случае выбирается произвольно. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил.Построение дерева разбора подтверждает принадлежность входной цепочки данному языку. При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что грамматика данного языка является недетерминированной.

**Дерево синтаксического анализа (синтаксическое дерево)***Неоднозначность = В случае, когда для построения цепочки есть альтернатива, эта грамматика неоднозначная. Критерием неоднозначности являются различные деревья.*

*Если какое-либо предложение, генерированное грамматикой, имеет более 1-го дерева разбора, то говорят, что грамматика неоднозначна. Задача установления неоднозначности в какой-либо грамматике неразрешима, т.е. не существует алгоритма, который бы за конечное число шагов, рассмотрев предложение, ответил бы – однозначна грамматика его породившая, или нет.*

Система подстановок (непосредственных выводов) в такой ФГ образует древовидную структуру – **дерево синтаксического анализа (ДСА)**:

* каждому правилу грамматики соответствует поддерево, в котором символ левой части образует вершину-предка, а символы правой части – вершины-потомки;
* ·непосредственному выводу с заменой левой части на правую соответствует процесс построения дерева «сверху-вниз» от предка к потомкам;
* непосредственному выводу с заменой правой части правила на левую соответствует достраивание вершины – предка над группой вершин – потомков;
* корневой вершиной ДСА является вершина с начальным нетерминалом **S**.
* конечными (терминальными) вершинами ДСА являются вершины, содержащие терминальные символы грамматики;
* последовательность терминальных вершин, обойденная слева направо, образует предложение языка, для которого построено ДСА.

Синтаксическое дерево является не просто иллюстрацией последовательности подстановок (выводов). Оно является единственным результатом синтаксического анализа. Фактически оно выявляет все структурные характеристики транслируемого текста, такие как вложенность или приоритеты отдельных синтаксических элементов. Порядок обхода синтаксического дерева (или то же самое, что порядок его построения) определяют последовательность выполнения операций в транслируемой программе, соответствующих отдельным правилам грамматики.

В практике синтаксического анализа используются только контекстносвободные грамматики. Кроме формальных оснований (сложность и эффективность алгоритмов работы анализаторов) этому способствуют и особенности содержательной работы с такими грамматиками:

система подстановок (непосредственных выводов) в такой ФГ образует древовидную структуру – **дерево синтаксического анализа (ДСА) или синтаксическое дерево**:

Справедливости ради следует заметить, что «ветвление» дерева происходит только в том случае, если в правой части правила имеется несколько нетерминалов. В противном случае (при наличии единственного нетерминала) получается линейная цепочка, как это имеет место в регулярных грамматиках.

Использование синтаксического дерева не означает, что оно обязательно существует в распознавателе в виде структуры данных. Как раз наоборот. Большинство распознавателей в процессе работы производят последовательный выбор правил, соответствующий процедуре обхода дерева, при этом само дерево разворачивается «во времени». Здесь имеется прямая аналогия с рекурсивными функциями: дерево «экземпляров» представляет собой развернутую во времени последовательность вызовов функцией самой себя, а стек в каждый момент времени содержит локальный контекст (фреймы) текущей последовательности вызовов. Иногда, правда, синтаксическое дерево может быть построено, но уже как результат работы распознавателя.

И, наконец, из рекурсивного характера система правил и процесса их подстановки следует наличие **стека** в распознавателе. Но есть еще одно более жесткое утверждение: кроме стека распознаватель не нуждается более ни в какой дополнительной изменяемой памяти, кроме «зашитых» в нем управляющих таблиц, т.е. является **конечным автоматом,** работающим со входной строкой – анализируемым предложением языка. Естественно, что система команд такого распознавателя предполагает стандартные действия со стеком, продвижение по входной строке и систему состояний-переходов, зависящих от текущих символов стека и входного предложения. Содержимое стека в различных распознавателях может быть разным, но имеющим один и тот же смысл по отношению к синтаксическому дереву: это граница его **недостроенной части.**

## 4.2 Постановка задачи синтаксического анализа

Понятно, что любая задача преобразования цепочек символов в рамках любой грамматики может быть решена путем полного перебора всех возможных вариантов подстановок, что влечет за собой экспоненциальную (показательную) трудоемкость решения задачи, не приемлемую в реальных условиях. Любой человек, использующий транслятор, в праве ожидать, что последний не слишком уменьшает скорость работы при росте объема транслируемой программы (то есть трудоемкость близка к линейной). Для этого прежде всего требуется отказаться от «тупого» перебора вариантов подстановки с возвратами к промежуточным цепочкам и обеспечить на каждом шаге выбор единственно правильного направления движения из нескольких возможных (так называемый **жадный алгоритм**).

Теперь следует разобраться, в каких взаимоотношениях находятся формальные грамматики и синтаксический анализ.

Синтаксис любого языка программирования определяется формальной грамматикой (контекстно-свободной в нашем случае) - системой терминальных и нетерминальных символов и множеством правил. Анализируемая программа представляется предложением языка. **Задача синтаксического анализа** - определить, является ли это предложение правильным и построить для него последовательность непосредственных выводов из начального символа**,** или синтаксическое дерево.

Сам процесс построения дерева, равно и синтаксического анализа может быть как **нисходящим,** т.е. от вершины-предка к вершинам-потомкам с заменой левых частей правил на правые, и наоборот, **восходящим.**

В этом же контексте объясняется понятие **синтаксической ошибки.**  Если на каком-то этапе построения синтаксического дерева встречается недопустимая или тупиковая ситуация, то построенная последовательность терминальных символов соответствует синтаксически правильной части программы, а очередной «незакрытый» терминальный символ локализуется как синтаксическая ошибка.

Сам процесс должен быть однозначным, т.е. каждому правильному предложению должно соответствовать единственное ДСА, а процесс его построения не должен содержать «возвратов», т.е. на каждом шаге алгоритма распознавания при наличии альтернативных вариантов должен выбираться каждый раз единственно верный.

Пример дерева синтаксического анализа для распознавания цепочки вида **a\*(a+a).**

N = {S, E, T, F},

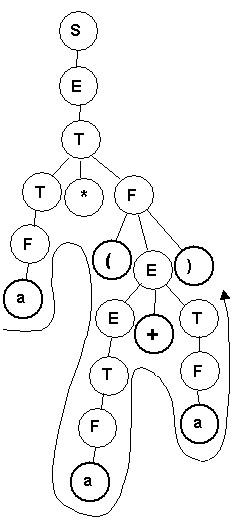
T = {+,-,/,\*,(,),a}

Z → E

E → E + T | E – T | T

R → T \* F | T / F | F

F → a | (E)



1. ДСА имеет единственно возможный вид, представленный на рисунке.
2. Обход терминальных вершин слева-направо соответствует распознаваемому предложению.
3. Размещение вершин соответствует приоритету операций, менее приоритетные соответствуют поддеревьям, расположенным ближе к корню.
4. Последовательность выполнения операций можно получить, используя рекурсивный обход ДСА и предполагая, что каждая вершина возвращает результат выполнения действий в своем поддереве (интерпретация) или цепочку команд, производящую эти действия (компиляция).
5. Сам способ построения ДСА (алгоритм синтаксического распознавания) пока обсуждать не будем.

***Взаимосвязь синтаксиса и формально определёных грамматик***

После предварительного анализа свойств грамматики необходимо напомнить, что формальные грамматики в целом играют роль **языка программирования синтаксиса,** а синтаксис языка программирования, представленный в виде конкретной грамматики – своеобразный аналог исходного текста программы. Здесь необходимо упомянуть тезисы, очевидные для любого программиста:

* одна и та же программа может быть написана различными способами. Точнее, одна и та же идея, воплощенная в результатах работы программы, может быть реализована на формальном языке множеством различных способов;
* программы пишутся «содержательно», не существует формальных методов разработки программ;
* невозможно определить формальными методами, эквивалентны ли две программы с точки зрения результата работы.

Возвращаясь к формальным грамматикам, можно аналогично утверждать:

* один и тот же синтаксис можно реализовать в виде множества грамматик;
* синтаксис преобразуется в ФГ «содержательно»;
* определить, идентичны ли две грамматики с точки зрения реализованного в них синтаксиса, формальными методами невозможно (в терминах теории алгоритмов **задача определения эквивалентности грамматик алгоритмически неразрешима).**

Наиболее продуктивно в синтаксическом анализе используются контекстно-свободные (КС-грамматики).

Независимо от конкретного языка присутствуют такие элементы как **альтернатива** (возможность выбора одной из синтаксических конструкций в данном контексте), **повторение, вложенность, приоритеты, необязательные элементы,** а также имеют место **ограничители, разделители** и т.п. Поэтому прежде всего стоит рассмотреть, как эти элементы синтаксиса реализуются по отдельности. Особенность использования для этих целей формальных грамматик состоит в том, что в них практически везде используется рекурсия как рабочий инструмент.

**Нетерминальный символ** – обозначение элемента синтаксиса и места его вхождения (подстановки) в другие элементы синтаксиса.

Кроме нетерминалов, явно обозначающих синтаксические единицы, для многих элементов синтаксиса используются вспомогательные нетерминалы. Иногда они вводятся, исходя из «технологических» соображений для придания грамматикам необходимых необходимых свойств (напр. конфликты «свертка-перенос»)

**Альтернатива.** Возможность выбора и подстановки одной из множества синтаксических конструкций обеспечивается наличием нескольких правил с одинаковой левой частью, нетерминал которой и обозначает соответствующую синтаксическую конструкцию.

**Повторение.** Построение повторяющихся цепочек базируется на известном свойстве: линейная (не ветвящаяся) рекурсия эквивалента циклу. В отношении формальных грамматик это означает, что нетерминал, “отвечающий за повторение”, непосредственно или косвенно рекурсивен. Остальные символы правил являются элементами повторения. Завершение процесса повторения должно сопровождаться применением правила, не содержащего такой рекурсии.

Из всех классов формальных грамматик только контекстно-свободные (КС-грамматики) продуктивно используются в синтаксическом анализе. Они дают дополнительную смысловую нагрузку термину "нетерминальный символ"

Общим недостатком курсов теории формальных языков является отрыв грамматик от продуцируемого синтаксиса. Обычно грамматика изучается как некоторая «данность», а не результат программирования определенного синтаксиса. Аналогично традиционным приемам, используемом в обычном программировании, в программировании синтаксиса независимо от конкретного языка присутствуют такие элементы как **альтернатива** (возможность выбора одной из синтаксических конструкций в данном контексте), **повторение, вложенность, приоритеты, необязательные элементы,** а также имеют место **ограничители, разделители** и т.п. Поэтому прежде всего стоит рассмотреть, как эти элементы синтаксиса реализуются по отдельности. Особенность использования для этих целей формальных грамматик состоит в том, что в них практически везде используется рекурсия как рабочий инструмент.

В качестве примера рассмотрим определение идентификатора: Использование **повторения для** определения идентификатора:

I → LX

X→ ɛ завершение повторения

X→ LX | DX повторение – прямая рекурсия

L→ a |…| z

D→ 0 |…| 9

Поскольку повторение часто используется в синтаксисе, существует еще одна, неканоническая форма представления правил с использованием метасимволов повторения. Круглые и фигурные скобки используются как метасимволы для обозначения ненулевого и любого, соответственно, числа повторений цепочки, заключенной в них. Кроме того, вертикальная черта внутри них может использоваться как обозначение выбора (альтернативы) одной из нескольких цепочек.

E→ T {+T | -T} цепочка сложений/вычитаний

C→ (D) десятичная константа

I → L{L | D} идентификатор

**Вложенность.** Принцип вложенности одних синтаксических конструкций в другие естественным образом реализуется в грамматике: нетерминал в левой части правила обозначает синтаксическую единицу, которая “расшифровывается” правой частью, нетерминалы в правой части обозначают “вложенные” элементы синтаксиса. Вложенность может быть и рекурсивной, когда в определении синтаксической единицы используются (прямо или косвенно) синтаксические единицы такого же типа. Пример:

синтаксис определения вложенных операторов:

O→ for (E;E;E)O | if (E)O else O | E; | {S}

S→ O | OS

**Приоритеты.** Наличие приоритетов в цепочке символов (операций) представляет собой не что иное, как неявную вложенность: любая цепочка операций более высокого приоритета образует синтаксическую единицу, которая входит в описание цепочки операций текущего приоритета. Поэтому для каждого уровня приоритета необходима своя группа правил, генерирующая цепочку (последовательность) таких операций.

Так, например, синтаксический элемент – арифметическое выражение (нетерминал **E**) представляет собой группу правил для генерации последовательности операций сложения вычитания, «операндами» в которой выступает нетерминалы следующего уровня – **T**. Они, в свою очередь, являются левой частью для группы правил, генерирующих последовательность операций умножения и деления для «операндов» следующего уровня – термов (**F**). То есть каждый элемент суммы может (потенциально) быть произведением и т.д. Естественно, что должно быть предусмотрено прямое приведение нетерминала одного уровня к нетерминалу другого при отсутствии соответствующих операций. «Приоритетные» скобки реализуются в виде правила, в котором нетерминал последовательности операций низшего приоритета, заключенный в скобки, сводится к нетерминалу высшего приоритета:

E → E + T | E - T | T

T → T \* F | T / F | F

F → a | (E)

Аналогично для грамматики, использующей аннулирующие правила:

E → TM

M → ɛ | +TM | -TM

T → FG

G → ɛ | \*FG | /FG

F → a | (E)

**Необязательные элементы** синтаксиса – это фрагменты

синтаксических конструкций, которые не обязательно могут присутствовать в синтаксическом элементе. Способ их описания может быть разным. При использовании аннулирующего правила необязательная часть обозначения отдельным нетерминалом, который раскрывается либо в виде такой конструкции, либо в виде пустой цепочки:

O → if (E)OX

X → else O | ɛ оператор if с else и без него

Возможно также явное перечисление вариантов синтаксиса в правых частях правил (например, вызов функции с пустым списком параметров, единственными параметром и их последовательностью):

F → a | a()| a(E) | a(SE) S → E| SE

**Разделители.** Терминальный символ-разделитель используется для “подсказки”, что вслед за ним следует еще один повторяющийся элемент синтаксиса. Таким образом, он обозначает повторение рекурсии, поэтому в грамматике, ориентированной на нисходящий разбор, за ним обычно следует рекурсивный нетерминал.

Типичный пример, список, разделенный запятыми (с использованием **аннулирующего правила** – правила с пустой правой частью):

L→ aX

X→ ɛ завершение списка

X→ ,aX продолжение списка

L => aX => a,aX => a,a,aX => a,a,a

В грамматиках, ориентированных на восходящий разбор, разделитель совместно с первым элементом перечисления (списка) используется для введения нетерминала, обозначающего незавершенный список:

L→ a | Sa S – обозначение незавершенного списка

S→ a, | Sa,

a,a,a => Sa,a => Sa => L

**Ограничители.** Терминальный символ-ограничитель, наоборот, обозначает завершение процесса повторения при генерации символов как одной, так и несколькими группами правил. При этом сам ограничитель встречается в правой части правила вслед за нетерминалом, из которого генерируются все повторяющиеся цепочки. Таким образом, ограничитель всегда является контекстом (окружением) “более высокого уровня”, чем сама цепочка. Типичные примеры: список операторов, заключенный в фигурные скобки, имеет ограничителем закрывающуюся скобку:

O → {S}

S → ɛ завершение списка

S → OS повторение

O => {S} => {OS} => {OOS} => {OOOS} => {OOO}

**Технология программирования» синтаксиса в формальных грамматиках**

Имея под рукой ряд приемов описания отдельных элементов синтаксиса, можно приступать к разработке ФГ для языка в целом. Эта задача сложна хотя бы в объемном отношении: ФГ реального языка программирования включает в себя несколько сот правил. Здесь опять же проступает аналогия с **технологией программирования:** можно пользоваться «методом северозападного угла», начиная записывать программу с первого выполняемого оператора, а можно пользоваться нисходящим структурным проектированием. При программировании синтаксиса тоже нужно стремиться к структурированному иерархическому описанию: каждый элемент синтаксиса реализуется отдельной, как можно более независимой группой правил, а входящие в нее нетерминалы обозначают вложенные или «не раскрытые» элементы синтаксиса. Причем по правилам нисходящего проектирования начинать надо с самых общих элементов верхнего уровня.

Аналогичные приемы следует применять для анализа ФГ на предмет продуцируемых ими цепочек: каждая группа правил рассматривается отдельно и анализируется вид генерируемых ею **промежуточных** цепочек, содержащих нетерминалы, соответствующие правилам следующего уровня.

**Отношения между символами в формальных грамматиках**

Любая ФГ продуцирует бесконечное множество предложений языка. Но, поскольку в них присутствуют внутренние закономерности (обусловленные самой грамматикой), то и между различными символами (как терминальными, так и нетерминальными) существуют отношения, которые характеризуют их взаимное положение в синтаксическом дереве.

### Множество FIRST и алгоритм его построения

Определим отношения, которые устанавливаются между нетерминальным символом (**промежуточной вершиной ДСА**) и терминальными символами – **конечными вершинами в текущем** **поддереве и смежных поддеревьях**. Для каждого из нетерминалов входящие в отношение терминальные символы образуют множество. Есть несколько видов множеств, каждому из них можно дать не только формальное определение, но и содержательное название и графическую интерпретацию:

1. множество **FIRST -** «Первых из…»;
2. множество **LAST –** «Последних из…»;
3. множество **FOLLOW** – «Следующих за…» **или последователей.** Формально множество **FIRST** для заданного нетерминального символа **U** определяется как множество терминальных символов, с которых может начинаться цепочка, выводимая из **U**, т.е.

**m**  **FIRST(U) : U** → **mb**

Графическая интерпретация – в деревьях с корневой вершиной **U** символы **FIRST(U)** – это крайние правые терминальные символы дерева. Соответственно, если **U** в грамматике обозначает некоторый элемент синтаксиса (выражение, оператор, определение, список параметров и т.п.), то множество **FIRST(U)** на самом деле является ответом на вопрос: «С чего может начинаться (выражение, оператор, определение, список параметров)»? Такое содержательное понимание позволяет в простых случаях построить множество **FIRST**, просто просматривая все возможные последовательности выводов, (учитывая на содержательном уровне продуцируемые правилами повторы и вложенности).

Изображение выглядит как текст, бильярдный шар

Автоматически созданное описание

Пример множества **FIRST**

**Z** → **U#**

**U** → **U,T | T**

**T** → **\*T | A**

**A** → **Aa | a**

1. **FIRST(U) : U=>U,T=>U,T,T=>T,… ,T**

Символ **U** производит цепочку, состоящую из нетерминалов **T**, разделенных запятыми.

1. **T=>\*T=>\*\*T**
2. **T=>Aa=>Aaa=>Aaaa=>aaaa**

Символ **T** производит цепочку «звездочек», пока не заменится на символ **A**, который производит цепочку символов **a**. Таким образом, **T** порождает последовательность «звездочек» (возможно, пустую), за которой следует последовательность символов **a**. Отсюда **FIRST(U)={a,\*}.**

Если формализовать просмотр возможных правил подстановки, то можно предложить простой рекурсивный алгоритм построения множества **FIRST** для заданного нетерминала **U.**

Простой рекурсивный алгоритм построения множества **FIRST** для заданного нетерминала **U:**

* Просматривается грамматика и выбираются правила с символом **U** в левой части;
* Если в правой части находится терминальный символ, то он добавляется к множеству, т.е. **U**→ **ab => a**  **FIRST(U) ;**
* Если в правой части находится нетерминальный символ, то для него также строится множество **FIRST,** которое включается в **FIRST(U),** т.е. **U**→ **Vb => FIRST(V)**  **FIRST(U).**

На практике здесь удобно использовать рекурсивную функцию, которая возвращает строку символов множества**FIRST,** тем более, что принцип рекурсии достаточно ясно отражает суть происходящих событий: для решения задачи с заданным параметром **U** требуется решить аналогичную задачу с другим значением входного параметра. Кроме того, следует учесть рекурсивное «зацикливание» алгоритма для прямой или косвенной левосторонней рекурсии (например, для правила **U**→ **U,T**). С этой целью рекурсивные вызовы должны передавать ссылку на строку, содержащую все нетерминалы, для которых **FIRST** на данный момент уже строится.

**Влияние аннулирующих правил. Множество FIRST\*.** Описанная выше идиллия нарушается, как только в грамматике появляются правила

(нетерминалы), которые могут порождать пустые цепочки. **Множество FIRST\*.**

Когда в грамматике имееются правила (нетерминалы), которые могут порождать пустые цепочки, или **аннулирующие правила (нетерминалы),** тогда «на первое место» выходит следующий за ним символ правила, для которого справедливы все те же действия. Кроме того, аннулирующий нетерминал может порождать пустую цепочку, а может и не порождать, что тоже усложняет алгоритм.

В связи с этим возникает необходимость в **множество FIRST\*.** Определим, когда нетерминал является **аннулирующим.** Здесь необходимо учесть, что пустая цепочка может быть произведена не только непосредственно правилом **U**→ ɛ**,** но и последовательностью выводов **U =>** ɛ**.** Свойство «аннулируемости» может быть определено рекурсивно и аналогично вычислено простым рекурсивным алгоритмом - нетерминал **U** порождает пустую цепочку -

 если имеется правило **U**→ ɛ**,**  если имеется правило вида **U**→ **ABC,** в правой части которого находятся только аннулирующие нетерминалы.

Изображение выглядит как текст

Автоматически созданное описание

Для начала определим, когда нетерминал является **аннулирующим.**

Еще один момент связан с возможностью быть аннулирующим самого нетерминала **U**, для которого строится **FIRST**. Это означает, что сам нетерминал может порождать пустую цепочку, но тогда «на первое место» выходит окружение (контекст), в котором может находиться нетерминал **U**. Поэтому нам придется рассмотреть два случая: множество **FIRST\*(U),** построенное без учета контекста, и **FIRST(U),** учитывающий окружение соответствующего нетерминала**.**

Алгоритм построения **FIRST\*(U):**

1. Просматривается грамматика и выбираются правила с символом **U** в левой части и непустой правой частью;
2. Выполняется цикл просмотра символов правой части выбранного правила (до первого терминала или до конца);
3. Если очередной символ **Ai** является терминальным, то он включается в множество (**Ai**  **FIRST\*(U))** и цикл просмотра завершается.
4. Для очередного нетерминала **Ai** строится множество **FIRST\*(Ai),** которое тоже добавляется к **FIRST\* (U),** т.е. **FIRST\*(Ai)** ⊆ **FIRST\* (U).** 5. Если **Ai =>** ɛ, т.е. нетерминал является аннулирующим, то происходит переход к следующему символу правой части, иначе цикл просмотра завершается.

Особенности построения **FIRST(U),** учитывающего окружение (контекст) нетерминала, легко можно увидеть на синтаксическом дереве (см. рисунок выше). Если нетерминальная вершина **U** порождает пустое поддерево, то необходимо подняться вверх по дереву и найти те терминалы, которые могут следовать за **U.**  Ниже мы обсудим все подробности построения множества последователей или **FOLLOW(U).** А пока заметим, что алгоритм построения **FIRST(U)** дополняется еще одним пунктом:

6. Если же при выполнении п.3-5 мы дошли до конца правой части правила, т.е. оно состоит только из аннулирующих нетерминалов, то необходимо подняться вверх по дереву к последователю левой части, т.е.

**FOLLOW(U)\*** ⊆ **FIRST(U).**

**Пример для грамматики без аннулирующих правил.**

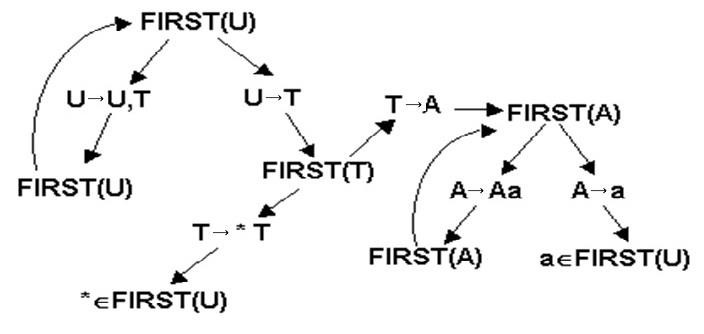
Чтобы «вручную» отследить выполнение рекурсивного алгоритма, можно построить граф (дерево) связей правил и множеств. В его вершины нужно помещать множества **FIRST**, вычисляемые для символов, и правила, на основе которых они вычисляются. Рассмотрим граф для нетерминала уже использованной грамматики:

Z→ U#

U→ U, T | T

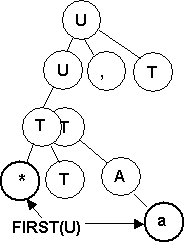
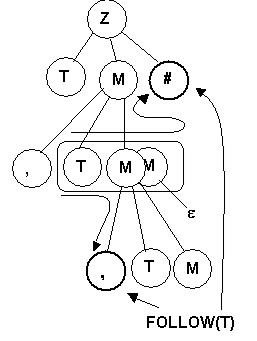
T→ \*T | A

A→ Aa | a



**Множество FOLLOW и алгоритм его построения**

Легко заметить, что множество **FIRST** связано с построением дерева «в глубину и влево». Образно говоря, оно представляет собой «левый нижний край» дерева, вернее, множества деревьев, выводимых из исходного нетерминала (на рисунке изображена сдвоенная вершина для вариантов построения цепочек **U=>T=>\*T** и**U=>T=>A=>a** ). С применением аннулирующих правил связано другое «перемещение» - вправо от текущего нетерминала с определением смежных символов. Такое движение «вдоль по правой части правила» с погружением в поддеревья нетерминальных вершин позволяет определить свойство **смежности** двух символов или **следования** одного символа за другим. Формально множество последователей **FOLLOW** для заданного нетерминального символа **U** определяется как множество терминальных символов, которые могут находиться вслед за **U** в произвольной промежуточной цепочке, выводимой из **Z**.



m  FOLLOW(U)

Z ➾ αUmβ

Z → TM#

M → ,TM | ɛ

T → \*T | A

A → Aa | a

Алгоритм построения **FOLLOW(T)**

1. Просматриваются правые части всех правил грамматики, для каждого найденного символа **T** выполняются следующие действия;
2. Правая часть просматривается от символа, следующего за **T,** до первого терминала (или до конца);
3. Если очередной символ **Ai** является терминальным, то он включается в множество (**Ai**  **FOLLOW(T))** и цикл просмотра прекращается;
4. Если очередной символ **Ai** является нетерминалом, то для него строится множество **FIRST\*(Ai),** которое добавляется к **FOLLOW(T),** т.е. **FIRST\*(Ai)** ⊆ **FOLLOW(T)**;
5. Если нетерминал **Ai** является аннулирующим, т.е. может порождать пустые цепочки, то происходит переход к следующему символу правой части, иначе просмотр завершается;
6. Если при выполнении п.2-5 мы дошли до конца правой части правила, т.е. оно от символа **T** до конца состоит только из аннулирующих нетерминалов (либо этот символ является последним), то необходимо подняться вверх по дереву к последователю левой части, т.е. **FOLLOW(M)** ⊆ **FOLLOW(T),** где **M –**нетерминал левой части.

Графическая интерпретация видна из определения: для нетерминала **T** ищутся ближайшие терминальные вершины «вправо вниз». При поиске учитываются аннулирующие нетерминалы: если вершина может порождать пустую цепочку, то движение вправо продолжается, а по достижении конца текущего уровня (правой части правила) происходит переход вверх (через символ левой части). Такое образно-интуитивное описание нужно дополнить формальным:

**Замечания по алгоритмам построения FIRST и FOLLOW.** Несмотря на то, что множества **FIRST** и **FOLLOW**различны по своей сути, их алгоритмы практически идентичны. Единственная разница состоит в следующем:

* при построении **FIRST** нетерминал ищется в левых частях правил и просматриваются правые части целиком;
* при построении **FOLLOW** нетерминал ищется в правых частях правил и они просматриваются от найденного символа до конца.

Кроме того, при построении **FOLLOW** всегда учитывается окружающий контекст, т.е. по окончании цикла просмотра правой части производится построение **FOLLOW** для нетерминала левой. Для **FIRST** выделяется как контекстный (**FIRST**), так и неконтекстный (**FIRST\***) варианты.

**Пример построения множеств FIRST и FOLLOW для простой грамматики**. Если Вы заметили, алгоритмы построения похожи, только **FIRST** ищет нетерминал в левых частях, а **FOLLOW** – в правых.

Граф-схема связей правил и множеств тоже аналогична.

Z → N#

N → UM

M → ,UM | M → ɛ

U → aSK

S → aS | S → ɛ

K → [N] | K → ɛ

Изображение выглядит как текст, устройство, снимок экрана

Автоматически созданное описание

Для начала определим аннулирующие нетерминалы грамматики, таковыми являются **K** и **S,** имеющие аннулирующие правила. Поскольку нет правых частей, состоящих исключительно из аннулирующих нетерминалов, то все другие таким свойством не обладают.

**Пример построения множеств FIRST и FOLLOW для грамматики**.

Для начала определим аннулирующие нетерминалы грамматики, таковыми являются **K** и **S,** имеющие аннулирующие правила. Поскольку нет правых частей, состоящих исключительно из аннулирующих нетерминалов, то все другие таким свойством не обладают.



Содержательная интерпретация множеств FIRST и FOLLOW

Содержательная интерпретация множеств FIRST и FOLLOWзависит от роли соответствующих нетерминалов в синтаксисе, который реализуется грамматикой. Так, если нетерминал используется как обозначение некоторой синтаксической единицы (или фрагмента синтаксиса), то множество **FIRST** является ответом на вопрос: «С какого символа может начинаться синтаксическая единица»? Иногда на него можно ответить, не проводя приведенных выше построений. Множество **FOLLOW** обычно применимо к аннулирующим правилам, которые могут использоваться для реализации различных элементов синтаксиса:

* если аннулирующее правило используется для ограничения повторения последовательности символов (в паре с другими правилами, обозначающими его продолжение), то множество **FOLLOW** для нетерминала левой части представляет ответ на вопрос: «Какие символы ограничивают повторение элемента синтаксиса, или служат признаком такого окончания»?
* если аннулирующее правило используется для обозначения необязательного элемента синтаксиса, то множество **FOLLOW** для нетерминала левой части представляет ответ на вопрос: «Какие символы могут находиться на месте отсутствующего элемента синтаксиса»?

## 4.3 Метод рекурсивного спуска

Рассмотрим один из фундаментальных методов разбора, применимый к некоторому подклассу КС-грамматик.

**Метод рекурсивного спуска (РС-метод)** реализует разбор сверху-вниз и делает это с помощью системы рекурсивных процедур.

Для **каждого нетерминала** грамматики создается своя процедура, носящая **его имя**; ее задача — начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала.

Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова.

Если такую подцепочку найти не удается, то процедура завершает свою работу, сигнализируя об ошибке. Это означает, что цепочка не принадлежит языку; разбор

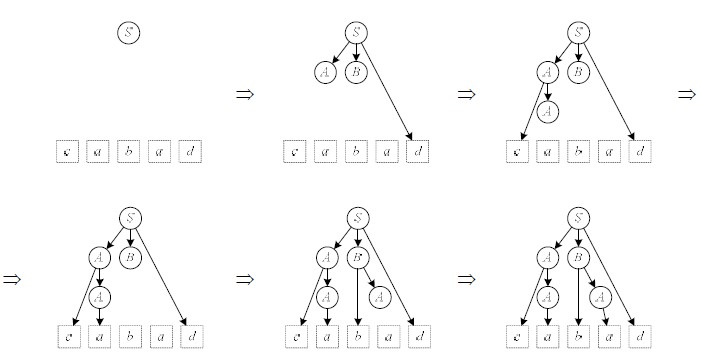
Рассмотрим грамматику *G*1 = ⧼{*a*, *b*, *c*, *d*}, {*S*, *A*, *B*}, *P*, *S* ⧽, где

*P*:

*S → ABd*

1. *→ a* | *cA*
2. *→ bA* и надо определить, принадлежит ли цепочка *cabad* языку *L*(*G*1). Построим левый вывод этой цепочки: *S → ABd → cABd → caBd → cabAd → cabad.* Следовательно, цепочка *cabad* принадлежит языку *L*(*G*1).

Построение левого вывода эквивалентно построению дерева вывода методом сверху-вниз (нисходящим методом), при котором на очередном шаге раскрывается самый левый нетерминал в частично построенном дереве.



Метод рекурсивного спуска (РС-метод) реализует разбор сверху-вниз и делает это с помощью системы рекурсивных процедур.

Для **каждого нетерминала** грамматики создается своя процедура, носящая **его имя**; ее задача — начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала.

Если такую подцепочку найти не удается, то процедура завершает свою работу, сигнализируя об ошибке. Это означает, что цепочка не принадлежит языку; разбор останавливается.

Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова.

Сформулируем достаточное условие применимости метода рекурсивного спуска.

**Достаточное условие применимости метода рекурсивного спуска**

Для применимости метода рекурсивного спуска достаточно, чтобы каждое правило в грамматике удовлетворяло одному из двух видов:

(а) *X* → α, где α ∈ (*T* ∪ *N*)\* и это единственное правило вывода для этого нетерминала;

(б) *X* → *a*1α1 | *a*2α2 | ... | *an*α*n*, где *ai* ∈ *T* для всех *i* = 1, 2,..., *n* ; *ai* ≠ *aj* для *i* ≠ *j*; α*i* ∈ (*T* ∪ *N*)\*, т. е. если для нетерминала *X* правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть попарно различными;

Это условие не является необходимым. Грамматику, удовлетворяющую данному условию, называют ***s*-*грамматикой***.

В принципе, по любой грамматике можно построить синтаксический анализатор, но грамматики, используемые на практике, имеют специальную форму.

Метод рекурсивного спуска является одной из возможных реализаций нисходящего анализа с прогнозируемым выбором альтернатив. Прогнозируемый выбор означает, что по грамматике можно заранее предсказать, какую альтернативу нужно будет выбрать на очередном шаге вывода в соответствии с текущим символом (т.е. первым символом из еще не прочитанной части входной цепочки).

Метод рекурсивного спуска (recursive descent method) это один из наиболее простых и, соответственно, один из наиболее популярных методов нисходящего синтаксического анализа.Наша дальнейщая цель - подробно рассмотреть этот подход и сформулировать критерий его применимости.

Принципы, лежащие в основе метода рекурсивного спуска довольно просты. К примеру, рассмотрим задачу вычисления значения арифметической формулы для целочисленных значений, бинарных операций сложения (+), вычитания (–), умножения (\*) и деления нацело (/), а также круглых скобок.   
*Также для арифм.выражений подходят нисходящий/восходящий методы (начало главы 4)*  
Как обычно, приоритеты операций умножения и деления равны и их приоритет больше, чем приоритеты операций сложения и вычитания, причем приоритеты этих операций также равны. Будем называть операции + и – операциями типа сложения, а операции \* и / – операциями типа умножения. Круглые скобки используются для изменения стандартного порядка выполнения операций. Тогда задача рекурсивного спуска заключается в определении алгоритма, вычисляющго значение формулы.

Отметим, что явно прослеживается связь синтаксического и лексического анализа. Вход синтаксического анализатора – последовательность лексем и таблицы, например, таблица внешних представлений, которые являются выходом лексического анализатора. Выход синтаксического анализатора – дерево разбора и таблицы, например, таблица идентификаторов и таблица типов, которые являются входом для следующего просмотра компилятора (например, это может быть просмотр, осуществляющий контроль типов). Анализаторы реально используемых языков обычно имеют линейную сложность; это достигается, например, за счет просмотра исходной программы слева направо с заглядыванием вперед на один терминальный символ (лексический класс).

Ну, и последнее замечание. Совсем необязательно, чтобы фазы лексичекого и синтаксического анализа выделялись в отдельные просмотры. Очень часто эти фазы взаимодействуют друг с другом на одном просмотре. Основной фазой такого просмотра считается фаза синтаксического анализа, при этом синтаксический анализатор обращается к лексическому анализатору каждый раз, когда у него появляется потребность в очередном терминальном символе.

## 4.4 Нисходящий анализ с прогнозируемым выбором альтернатив

В процессе построения левого вывода для произвольной цепочки в грамматике *G*1:

*S → ABd*

1. *→ a* | *cA*
2. *→ bA*

можно отметить следующее:

1. любой вывод начинается с применения правила *S* → *ABd*;
2. если на очередном шаге сентенциальная форма имеет вид ω*B*α, где ω *T*\*- начало анализируемой цепочки, нетерминал *B* - самый левый в сентенциальной форме, то для продолжения вывода его нужно заменить на *bA* (других альтернатив нет);
3. если на очередном шаге сентенциальная форма имеет вид ω*A*α, где ω *T*\*- начало анализируемой цепочки, то выбор нужной альтернативы для замены *A* можно однозначно предсказать по тому, какой символ в анализируемой цепочке следует за начальной подцепочкой ω: если символ α, то применяется альтернатива *A* → *a*, если символ *c*, то альтернатива *A* → *cA*;если какой-то иной символ - фиксируется ошибка: анализируемая цепочка не принадлежит языку *L*(*G*1).

Если на каком-то шаге получилась сентенциальная форма вида ωα, отличная от (2) и (3), где ω - максимально длинное начало, состоящее только из терминалов, то если α пуста и ω совпадает с анализируемой цепочкой, процесс вывода успешно завершается, иначе фиксируется ошибка: анализируемая цепочка не принадлежит языку *L*(*G*1).

Еще раз напомним, что сентенциальная форма грамматики ***G*** – это цепочка, выводимая из начального нетерминала грамматики ***G.***

Отмеченные факты по поводу выбора нужной альтернативы на очередном шаге вывода в грамматике *G*1 представим в виде так называемой таблицы прогнозов (или таблицы предсказаний):

**Таблица прогнозов / предсказаний:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ***a*** | ***b*** | ***c*** | ***d*** |
| ***S*** | *S* → *ABd* | *S* → *ABd* | *S* → *ABd* | *S* → *ABd* |
| ***A*** | *A* → *a* |  | *A* → *cA* |  |
| ***B*** |  | *B* → *bA* |  |  |

Имея такую таблицу прогнозов (предсказаний) для КС-грамматики *G*, можно предложить следующий алгоритм нисходящего анализа (построение левого вывода):

1. Начать построение вывода с сентенциальной формы, состоящей из одного начального символа *S.*
2. Пока не будет получена цепочка, совпадающая с анализируемой, повторять следующие действия:

Пусть ω*Y*α *-* очередная сентенциальная форма, где ω *T\**. Если *w* не совпадает с началом анализируемой цепочки, то прекратить построение вывода и сообщить об ошибке: цепочка не принадлежит *L*(*G*). В случае, когда *w* совпадает с началом, и следующим после ω символом в анализируемой цепочке является символ *z*,заменить нетерминал *Y* на правую часть правила, которое находится в ячейке таблицы прогнозов на пересечении строки *Y* и столбца *z.* Если указанная ячейка пуста, прекратить построение вывода и сообщить об ошибке: цепочка не принадлежит *L*(*G*).

Как мы видели выше, один из способов реализовать программуанализатор для нисходящего анализа с прогнозируемым выбором альтернатив заключается в построении системы рекурсивных процедур. Это метод рекурсивного спуска.

Другой способ, с явным использованием стека для хранения нетерминальной части сентенциальной формы, известен как *LL*(1)анализатор. Техника построения рекурсивных процедур уже была рассмотрена и продемонстрирована на примере, но остался открытым вопрос: как в общем случае «запрограммировать» процедуру на выбор нужной альтернативы по текущему символу. Ответ теперь известен - использовать таблицу прогнозов.

К сожалению, не для каждой КС-грамматики существует таблица с однозначными прогнозами, позволяющая безошибочно осуществить выбор альтернативы на каждом шаге вывода. В некоторых случаях заранее спрогнозировать выбор альтернативы невозможно: может оказаться, что подходящими в данной ситуации являются сразу несколько альтернатив (неоднозначный прогноз).

Таким образом, нисходящий анализ с прогнозируемым выбором альтернатив пригоден лишь для некоторого подкласса КС-грамматик.

## 4.5 О применимости метода рекурсивного спуска

Метод рекурсивного спуска **примени́м** к грамматике, если для нее существует **таблица однозначных прогнозов** и, соответственно, метод рекурсивного спуска (без возвратов) **непримени́м к неоднозначным грамматикам**.

Например, по грамматике

*G*2:

*S* → *aA |B* | *d*

1. *→* *d* | *aA*
2. *→* *aA* | *a*

нельзя дать однозначный прогноз, что делатьна первом шаге при анализе цепочки, начинающейся с символа *a,* т.е. по текущему символу *a* невозможно сделать однозначный выбор: *S* → *aA* или *S* → *B*.

***Множество first* (α)** в грамматике *G* = ⧼*T*, *N*, *P*, *S*⧽ *-* это множество терминальных символов, которыми начинаются цепочки, выводимые в *G* из цепочки α ∈ (*T* ∪ *N*)\*, т. е. *first* (α) = {*a* ∈ *T* | α ➾ *a*α´, α´ ∈ (*T* ∪ *N*)\*}.

Соответственно наличие в грамматике нетерминала X с правилами вида X → α и X → β, из правых частей которых выводятся цепочки, начинающиеся одним и тем же терминалом *a, т.е.* α ➾ *a*α´ и β ➾ *a*β´, делает неоднозначным прогноз по символу *a.*

Например, для альтернатив правила *S* → *A | B* в грамматике *G*3:

*S* → *A | B*

A *→* *aA* | *d* B *→* *aB* | *b*

имеем:*first* (*A*) = {*a*, *d*}, *first* (*B*) = {*a*, *b*}. Пересечение этих множеств непусто: *first* (*A*) ∩ *first* (*B*) = {*a*} ≠ ∅, и поэтому метод рекурсивного спуска к *G*3 непримени́м.

**Итак, наличие в грамматике правила с альтернативами *X* → α | β, такими что *first* (α) ∩ *first* (β) ≠** ∅**, делает метод рекурсивного спуска неприменимым.**

Рассмотрим пример грамматики, которая может быть **однозначной**, однако **однозначных прогнозов** для нее также ***не существует***:

*G*3:

*S* → *A | B*

1. *→* *aA* | *d*
2. *→* *aB* | *b*

Действительно, каждая цепочка, выводимая в *G*3 из *S*, оканчивается либо символом *b*, либо символом *d,* и имеет единственное дерево вывода. Но невозможно предсказать, с какой альтернативы (*S* → *A* или *S* → *B*) начинать вывод, не просмотрев всю цепочку до конца и не увидев последний символ.

Наличие в грамматике нетерминала *X* справилами вида *X* → α*A* и *X* → α*B*, из правых частей которых выводятся цепочки, начинающиеся одним и тем же терминалом *a*, т. е. α ➾ *a*α´ и β ➾ *a*β´, делает неоднозначным прогноз по символу *a*. Соответственно метод

рекурсивного спуска **непримени́м**.

Как показывает этот пример, грамматика может быть однозначной, однако однозначных прогнозов для нее также не существует. Так что нисходящий анализ с прогнозируемым выбором альтернатив невозможен по такой грамматике, и метод рекурсивного спуска непримени́м. Рассмотрим примеры грамматик с ɛ-правилами

Примеры **грамматик с ɛ-правилами**.

*G*4:

|  |  |
| --- | --- |
| *S* → *aA |BDc* | *first*(*aA*) = { *a* }, *first*(*BDc*) = { *b, c* }; |
| *A →* *BAa* | *aB* | *b* | *first*(*BAa*) = { *a, b* }, *first*(*aB*) = { *a* }, *first*(*b*) = { *b* }; |

*B →* ɛ *first*(ɛ) = ;

*D → B | b* *first*(*B*) = *, first*(*b*) = { *b* }*.*

Метод рекурсивного спуска непримени́м к грамматике *G*4, так как *first* (*BAa*) ∩ *first* (*aB*) = {*a*} ≠ ∅.

Следующий пример показывает еще одно свойство грамматик, наличие которого делает РС-метод неприменимым. *G*5: *S* → *aA*

1. *→* *BC* | *B*

*C →* *b* | ɛ

1. *→* ɛ

Пересечение множеств *first* пусто для любой пары альтернатив грамматики *G*5, однако наличие двух различных альтернатив, из которых выводится пустая цепочка, делает

данную грамматику неоднозначной и, следовательно, метод рекурсивного спуска к ней непримени́м. Действительно,

*BC* ➾ ɛи *B* ➾ ɛ.Цепочка *a* имеет два различных дерева вывода:

Таким образом, если в грамматике для **правила *X* → α |β** выполняются **соотношения α** ➾ **ɛ и β** ➾ **ɛ,** то метод рекурсивного спуска **неприменим**.

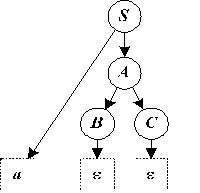
Осталось выяснить, как обстоят дела с применимостью метода, если для каждого нетерминала грамматики существует не более одной альтернативы, из которой выводится ɛ.

***S***

***a***

***B***

***A***



Пример грамматики, где для каждого нетерминала грамматики существует не более одной альтернативы, из которой выводится ɛ.

*G*6:

*S* → *cAd* | *d*

*A* *→* *aA* | ɛ *first* (*cAd* ) = {*c*}, *first* (*d* ) = {*d*};

Однозначные прогнозы для выбора альтернативы нетерминала *S* существуют, так как *first* (*cAd*) ∩ *first* (*d* ) = ∅. Таблица прогнозов для *G*6:

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***a*** | ***c*** | ***d*** |
| ***S*** |  | *S* → *cAd* | *S* → *d* |
| ***A*** | *A* → *aA* | *A* → ɛ | *A* → ɛ |

Выбор альтернативы для *A* в данной грамматике также можно однозначно спрогнозировать: если текущим символом является *a*, применяется правило *A* → *aA*, иначе правило *A* → ɛ. Это возможно благодаря тому, что за любой подцепочкой, выводимой из *A*, следует символ *d*, который сам в эту подцепочку не входит. Процедура *A*( ) при выборе альтернативы *A* → ɛ просто возвращает управление в точку вызова, не считывая следующий символ входной цепочки. Процедура *S*( ), получив управление после вызова *A*( ), проверяет, что текущим символом является *d*. Если это не так, фиксируется ошибка. Конечно, проверку символа *d* (без считывания следующего символа из входной цепочки) могла бы сделать и сама *A*( ), но это излишне, так как *S*( ) все равно будет проверять *d*,иесли вместо *d* обнаружит другой символ, ошибка будет зафиксирована.

Итак, для грамматики *G*6, имеющей для каждого нетерминала не более одной альтернативы, из которой выводится пустая цепочка, метод рекурсивного спуска применим.

Процедура  *A*( )для нетерминала *A*, имеющего пустую альтернативу в грамматике *G*6, реализуется так:

void A

() { if ( c =='a' )

{

cout << "A-

>aA, "; gc ();

A (); }

else

{

cout << "A->epsilon, "; // след. символ не считывается

}

}

Следующий пример показывает, что наличие альтернативы α,такой чтоα ➾ ɛ, все же может сделать метод рекурсивного спуска неприменимым.

|  |  |
| --- | --- |
| *G* 7: |  |
| *S* | → *Bd* |
| *B* | *→* *cAa* | *a* |
| *A* | *→ aA* | ɛ |

*first* ( *cAa* ) = {*c*}, *first* (*a* ) = {*a*};

У нетерминала *S* правая часть единственна и проблема выбора альтернативы для *S* не стоит. Для выбора альтернативы нетерминала *B* существуют однозначные прогнозы, поскольку *first* (*cAa*) ∩ *first* (*a*) = ∅.

Однако для нетерминала *A* прогноз по символу *a* неоднозначен. Дело в том, что любой вывод, содержащий *A*, имеет вид: *S* → *Bd* → *cAad* → … → *ca…aAad*. Поэтому альтернативу *A* → ɛ следует применять только тогда, когда текущим символом является *a*, а следующий за ним символ отличен от *a* (например, *d*). Если текущий - *a*  и следующий за ним символ *-* тоже *a*, то выбирается альтернатива *A* → *aA*. Но сделать однозначный выбор только по текущему символу в пользу какой-то одной из этих альтернатив невозможно, так как анализатор не умеет заглядывать вперед (в непрочитанную часть анализируемой цепочки).

Как видим, в *G* 7 существует сентенциальная форма, например *cAad*, в которой после нетерминала *A*, имеющего в грамматике пустую альтернативу, стоит символ *a*, c которого также начинается и непустая альтернатива для *A*. В таком случае процедура *A*( ) не сможет правильно определить по текущему символу *a*, считывать ли следующий символ и вызывать *A*( ) (т. е. применять правило *A* → *aA*) или возвращать управление без считывания символа (правило *A* → ɛ). Опишем эту ситуацию более формально.

Итак, на примерах мы рассмотрели все случаи, когда можно построить однозначные прогнозы по грамматике.

**Множеств*о follow(A****)* - это множество терминальных символов, которые могут появляться в сентенциальных формах грамматики *G* = ⧼ *T*, *N*, *P*, *S* ⧽ непосредственно справа от *A* (или от цепочек, выводимых из *A*), т.е.

*follow*(*A*) ={*a* ∈ *T* | *S* ➾ α*A*β, β ➾ *a***γ**, *A* ∈ *N*, α, β, **γ** ∈ (*T* ∪ *N*) }.

Тогда, **если в грамматике есть правило**

***X* →** α **|**β**, такое что** β➾ɛ, ***first*(**α**)** ∩ ***follow*(*X*)** ≠∅**,** то метод рекурсивного спуска **неприменим** к

данной грамматике.

**Критерий применимости** метода рекурсивного спуска.

Подытожив рассмотренные примеры, сформулируем **критерий применимости** метода рекурсивного спуска.

Пусть *G -* КС-грамматика. Метод рекурсивного спуска примени́м к *G,* если и только если для любой пары альтернатив *X* **→** α **|**β выполняются следующие условия:

1. *first*(α) ∩ *first* (β) = ∅;
2. справедливо не более чем одно из двух соотношений: α➾ɛ,β➾ɛ;
3. если β➾ɛ, то *first*(α) ∩ *follow*(*X*)=∅.

***LL*(*k*)- и *LR*(*k*)-грамматики**

К классу грамматик, для которых существуют эффективные анализаторы, относятся ***LL*(*k*)-грамматики**, по которым, как правило, реализуется анализ сверху-вниз — нисходящий; ***LR*(*k*)-грамматики**, грамматики предшествования, по которым, как правило, реализуется анализ снизу-вверх — восходящий; и некоторые другие

**Анализатор для *LL*(*k*)-грамматик** просматривает входную цепочку слева направо и осуществляет детерминированный левый вывод, принимая во внимание *k* входных символов, расположенных справа от текущей позиции. Выбор альтернативы осуществляется на основе заранее составленной таблицы прогнозов.

**Анализатор для *LR*(*k*)-грамматик** просматривает входную цепочку слева направо и осуществляет детерминированный правый вывод, принимая во внимание *k* входных символов, расположенных справа от текущей позиции. Вывод строится методом сверток, как при разборе по леволинейной автоматной грамматике. Предварительно по *LR*(*k*)-грамматике строится таблица, которая на каждом шаге вывода позволяет анализатору однозначно выбрать нужную свертку.

Грамматика *G* является LL(1)-грамматикой тогда и только тогда, когда для любых двух различных ее выводов *А→α|β* выполняются следующие условия:

1. Не существует такого терминала *а*, для которого и α*,* и *β* порождают

строку, начинающуюся c *а.*

1. Пустую строку может порождать только один из выводов αили *β*.
2. Если *β=>* λ*,* то *а* не порождает ни одну строку, начинающуюся с терминала из FOLLOW(*A*).Перечисленные идеи нисходящего разбора соответствуют классу грамматик, именуемому **LL(k),** т.е. идеи эти «работают» на грамматиках соответствующего класса:

* Первая **L** обозначает принцип просмотра входного предложения (цепочки) в направлений слева-направо (**left**) c последовательным «закрытием» символов терминальными вершинами дерева;
* **k** – обозначает глубину просмотра «незакрытой» части цепочки для принятия решения о выборе одной из правых частей правила, соответствующего очередному нетерминалу. Обычно k=1;
* Вторая **L** соответствуют термину «левосторонний вывод» (**left**), т.е. нисходящий разбор с заменой левой части правила на правую.

Какими свойствами должны обладать **LL(1)-грамматики**, будет ясно позднее, после того, как будет изложены принципы организации и алгоритм работы нисходящего распознавателя. Однако основное принцип должен быть соблюден всегда: для каждого нетерминала и очередного незакрытого символа входной строки должна быть обеспечена возможность однозначного и окончательного (безвозвратного) выбора правой части правила, содержащего этот нетерминал в левой части.

*[Ссылка на itmo](https://neerc.ifmo.ru/wiki/index.php?title=LR(0)-%D1%80%D0%B0%D0%B7%D0%B1%D0%BE%D1%80) с примером восходящего LR(0) разбора с таблицей и описанием алгоритма*

*При LR(k)-анализе применяется метод перенос-свертка (англ. shift-reduce). Суть метода сводится к следующему:*

*Программа анализатора читает последовательно символы входной строки до тех пор, пока не накопится цепочка, совпадающая с правой частью какого-нибудь из правил. Рассмотренные символы переносим в стек (операция перенос).*

*Далее все символы совпадающей цепочки извлекаются из стека и на их место помещается нетерминал, находящийся в левой части этого правила (операция свертка).*

# 5. Семантика формальных языков

Классическая теория формальных языков, как уже отмечалось, занималась исключительно синтаксисом языков, изучая методы порождения и распознавания множеств слов. Семантика формальных языков, сравнительно молодая ветвь теории, занимается способами сопоставления некоего "смысла" словам (цепочкам) языка.

Необходимость в построении точного математического понятия "смысла" диктуется развитием информационных технологий, прежде всего технологии проектирования компиляторов.

Рассмотренные выше языковые модели определенным образом связаны с этапами этой технологии.

Текст входной программы, как известно, анализируется в несколько проходов. На первом проходе производится лексический анализ, а именно проверяется правильность простейших элементов текста, называемых лексемами. Примерами лексем могут служить идентификаторы и константы, разрешенные синтаксисом входного языка программирования. В процессе лексического анализа не проверяется синтаксическая правильность всей программы в целом, а проверяется только синтаксическая правильность лексем (в частности, правильность написания идентификаторов и констант). Так как лексемы обычно являются элементами некоторого регулярного языка, то базовой моделью для лексического анализатора является модель конечного автомата.

Если текст программы успешно прошел этап лексического анализа, то тогда проверяется его глобальная синтаксическая правильность. При этом каждая лексема рассматривается как буква. Здесь применяются методы синтаксического анализа, в частности рассмотренные ранее. В предположении, что синтаксис языка программирования описан КСграмматикой, основой для построения синтаксических анализаторов, как мы уже видели, выбирается модель синтаксического анализатора.

По завершении синтаксического анализа строится дерево вывода входной программы. После этого переходят к этапу генерации объектного кода, т.е. внутреннего машинного представления входного текста. Это значит, что выполняется перевод с некоторого языка программирования на язык машинных кодов. Но чтобы выполнить перевод текста на другой язык, необходимо каким-то образом понять его "смысл". Следовательно, анализ уже синтаксически проверенной программы с точки зрения ее "смысла" (семантический анализ) необходимо предшествует самой генерации объектного кода. И прежде всего необходимо уточнить математически, что такое "смысл" (как раньше мы математически определяли синтаксис в терминах грамматик).

Т.о. наша задача - рассмотреть возможные методы формального (математического) определения семантики для КС-языков. Тем самым мы всюду в дальнейшем предполагаем, что язык, семантика которого определяется, может быть задан некоторой КС-грамматикой.

*Фаза контроля типов*

*Идентификация. Работа с таблицами*

*Идентификация. Работа с типами*

*Причины использования промежуточных языков в компиляторах*

*Различные формы представления промежуточных языков (ПЯ) в компиляторах:*

1. *Атрибутивные деревья разбора*
2. *Прямая и обратная польские записи*
3. *Триады/тетрады*
4. *RTL*

*На этапе семантического анализа можно:*

1. *Проверить соответствие типов*

*2)Проверить управление (различных переходов)*

*3)Проверить единственность (чтобы один раз объявлялась переменная или функция)*

*4)Проверить имена (чтобы имя встречалось определенное количество раз)*

*Сравнение может быть статическим (на этапе трансляции) и динамическим (во время выполнения).*

*Приведение типов*

*Расширяющее (short -> long)*

*Сужающее (long -> short)*

*Формализм атрибутных грамматик оказался очень удобным средством для описания семантики языков программирования*

Сразу же необходимо сделать уточнение. Как мы уже заметили ранее, исследуя явление неоднозначности в КС-языках, "смысл" следует сопоставлять не самим словам языка, а деревьям их вывода: меняя дерево вывода данной цепочки, мы меняем и ее "смысл", понимаем ее по-другому. Далее, можно сопоставлять "смысл" не только словам языка, точнее, деревьям вывода этих слов из начального символа грамматики, но и так называемым фразам языка — терминальным цепочкам, выводимым из разных нетерминалов грамматики.

Например, фраза "...а так как мне бумаги не хватило" не является законченным предложением русского языка, но имеет, очевидно, какой-то "смысл". Точно так же оператор присваивания, "вынутый" из какой-то программы на каком-то языке программирования, не является "программой", не может рассматриваться как элемент данного языка, но мы в состоянии сопоставить ему тот или иной "смысл". Тем самым возникает **идея** определить "*смысл*" через **отображение множества "синтаксических объектов"** — деревьев выводов фраз языка в некоторое "предметное множество", **множество "семантических объектов".**

Нашей целью будет элементарное введение в формальное описание семантики КС-языков и описание некоторых простых алгоритмов формального разбора семантики языков программирования.

### 5.1 Формальное определение фразы КСязыка

***Фразой*** *языка* L(G)*, где* G = ⧼T, N, P, S⧽, назовем***терминальную цепочку***x*, если* A∈ x *для некоторого нетерминала* A ∈ N.

Если T(x) - дерево вывода фразы x с корневым нетерминалом A, то ***подфразой*** фразы **x** назовём выводимую из некоторого вхождения B терминальную цепочку

*Пусть* G = ⧼T, N, P, S⧽, *- КС-грамматика.* ***Терминальную цепочку* x** *называют фразой языка* L(G)*, если* A∈ x *для некоторого нетерминала* A ∈ N*.*

Допустим T(x) - дерево вывода фразы x с корневым нетерминалом A. Возьмем в T(x) некоторое поддерево с корневым нетерминалом B. Выводимую из этого вхождения Bтерминальную цепочку называют подфразой фразы x. Если вершина дерева T(x), соответствующая данному вхождению B, имеет глубину 1 (или, что равносильно, уровень, на единицу меньший уровня корня A), то данная подфраза называется подфразой первого уровня фразы x.

Рассмотрим в качестве примера следующую **грамматику G (1)** арифметических выражений:

**Expr → Atom** ∣ **(Expr+Expr)** ∣ **(Expr**∗**Expr),**

**Atom → a1** ∣ **a2** ∣**…**∣ **an.**

Предполагается, что вместо вхождения нетерминала Atom может быть подставлен любой символ некоторого алфавита V = {a1,a2,,…, an} (атом в арифметическом выражении может быть либо переменным, либо константой).

Нарисуем дерево вывода выражения (a+(b∗(c+(d∗(e+g))))), где a,b,c,d,e,g - некоторые атомы из V.

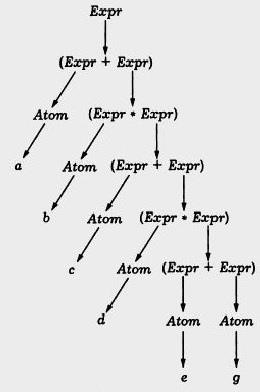


рис. 5.1 Дерево вывода выражения (a+(b∗(c+(d∗(e+g)))))

Это выражение в качестве подфраз первого уровня имеет цепочки a и (b(c+(d(e+g)))). Вторая фраза имеет подфразы первого уровня b и (c+(d(e+g))) и т.д. Заметим, что фраза a, выводимая из самого левого узла Expr глубины 1 имеет в качестве подфразы первого уровня ту же цепочку a, т.е. подфраза первого уровня может совпадать с самой исходной фразой.

Множество L(G) **всех фраз языка** определим как **PH(L)**, а множество **всех деревьев вывода фраз L** - определим как **TPH(L)**. Для однозначного языка L(G) эти множества эквивалентны.

### 5.2 Семантическая функция языка L

***Семантическая функция языка* L** *есть отображение* SEM(L):

TPH(L)→U(L)

- множества **всех деревьев вывода фраз L** в некоторое множество U(L), называемое **универсумом** (предметной областью, семантической областью, областью интерпретации) **языка L***.*

**Замечание.** Ради общности следовало бы определить семантическую функцию как частичное отображение, но мы не будем этого делать, вводя для "бессмысленных" фраз специальный "неопределенный" элемент в универсум ("нуль" предметной области, "неопределенный смысл"). Заметим также, что для однозначного языка семантическая функция может быть определена как отображение из множества фраз языка.

Способы определения семантической функции могут быть различными. Нас будет интересовать важный частный случай, когда значение семантической функции на некоторой фразе (под фразой здесь и далее понимается дерево ее вывода, причем разные деревья для одной и той же цепочки рассматриваются как разные фразы) определяется однозначно через значение этой функции на подфразах первого уровня. Более формально: представим фразу x синтаксически как ‘’соединение" своих подфраз первого уровня:

x= φ(y1,y2,…,ym) =u1y1u2y2…umymum+1,

где **u1,u2,…,um+1**- некоторые терминальные цепочки. Например, для определенного выше языка арифметических выражений фраза раскладывается следующим образом:

(a + (b ∗ (c + (d ∗ (e + g))))) = u1au2(b ∗ (c + (d ∗ (e + g))))u3, где u1 есть цепочка (, u2 - цепочка +, а u3 - цепочка ).

Таким образом, любая фраза представляется, вообще говоря, не как простое соединение своих подфраз первого уровня, а как соединение с "прослойками" в виде определенных терминальных цепочек. **Операцию φ**, задающую такие прослойки, **называют часто операцией конкатенации**. Эта операция определяется грамматикой языка.

Определим отображение **ψ: U(L)m → U(L)** (т.е. некоторая операция на предметной области), такое, что

SEM(L)(x)=ψ(SEM(L)(y1),…,SEM(L)(ym)).

Это ограничение называют также принципом гомоморфной интерпретации. Примем соглашение об обозначении **семантической функции** языка L(G)*,* [⋅]L, опуская индекс, указывающий на язык, если это не ведет к недоразумению. Тогда формула примет вид:

[x] = [φ(y1,…,ym)] = ψ([y1], …, [ym]).

Тогда предполагается, что, если фраза x представлена в виде (7.1), то определено отображение ψ: U(L)m → U(L) (т.е. некоторая операция на предметной области), такое, что

SEM(L)(x)=ψ(SEM(L)(y1),…,SEM(L)(ym))

Это ограничение, накладываемое на семантическую функцию, назовем принципом гомоморфной интерпретации.

Примем теперь некоторые соглашения об обозначениях.

**Семантическую функцию** языка L будем обозначать [⋅]L, опуская индекс, указывающий на язык, если это не ведет к недоразумению. Тогда равенство можно переписать в виде

[x] = [φ(y1,…,ym)] = ψ([y1], …, [ym]).

Подобные определения семантики фразы через семантику ее подфраз первого уровня называют семантическими правилами языка. Семантические правила соответствуют синтаксическим правилам — правилам исходной КС-грамматики. Так, для приведенной выше грамматики арифметических выражений, полагая U=R, можно записать следующие семантические правила:

[Expr] = [Expr]+ [Expr].

(для синтаксического правила Expr → (Expr+Expr)), т.е. здесь конкатенарной операции φ, такой, что φ(u,v)=(u+v) для любых двух фраз u и v, сопоставляется обычное арифметическое сложение; [Expr] = [Expr]\* [Expr].

(для синтаксического правила Expr → (Expr ∗ Expr)) операция - арифметическое умножение; [Expr] = [Atom]

(для синтаксического правила Expr → Atom);

[Atom]= [ai] (для синтаксического правила Atom→ai, i=1, …, n).

Строго говоря, в первых двух правилах мы должны различать три разных вхождения одного и того же нетерминала Expr, так как они соответствуют разным деревьям.

Семантическая функция языка арифметических выражений будет определена, если мы зададим значение этой функции на атомах: это естественно ассоциируется с хорошо знакомой процедурой вычисления значения арифметического выражения при подстановке вместо входящих в него переменных конкретных числовых значений (при этом не исключается, что атом может быть константой - обозначением конкретного числа; в таком случае само синтаксическое правило задает сразу значение семантической функции на данном атоме).

Для языка арифметических выражений семантическая функция будет определена, если мы зададим значение этой функции на **атомах**: это естественно ассоциируется с хорошо знакомой процедурой вычисления значения арифметического выражения при подстановке вместо входящих в него переменных конкретных числовых значений.

Для выражения **(a+(b**∗**(c+(d**∗**(e+g**))))), полагая

[a]=1, [b]=2, [c]=3, [d]=4, [e]=5, [g]=6,

получим значение всего выражения, равное 95.

Таким образом, в этом конкретном случае семантика фразы есть числовое значение представленного данной фразой арифметического выражения. Мы рассмотрели в качестве универсума множество вещественных чисел и получили одну семантику. Задав универсум какнибудь иначе (например, как множество комплексных чисел или множество функций некоторого класса), получим совсем другую семантику. Для одного и того же синтаксиса, следовательно, могут быть определены различные семантики (семантические функции).

Разобранный выше на примере языка арифметических выражений подход к определению семантики называют иногда экстенсиональным подходом. Суть его состоит в том, что явно определяется универсум как множество "внеязыковых" объектов (экстенсионалов) и каждой языковой фразе сопоставляется некоторый экстенсионал. Для разобранного выше примера экстенсионал — это вещественное число. Слово "внеязыковых" взято в кавычки потому, что универсум сам может быть некоторым языком (выступающим тогда по отношению к исходному языку как метаязык). Не исключено даже, что метаязык совпадает с самим определяемым языком - примером могут служить всевозможные толковые словари.

Существует и другой подход к определению семантики, называемый интенсиональным, одной из разновидностей которого является аксиоматический метод определения семантики языка.

Аксиоматический метод предполагает рассмотрение исходного, "объектного" языка, семантика которого определяется, как формальной теории (или формальной системы). Не давая строгого определения формальной теории в его общности, поясним его суть и рассмотрим пример.

Формальная теория задается как некоторый язык, цепочки которого называются в этом случае утверждениями (или предложениями). В этом языке определяется подъязык так называемых доказуемых утверждений: задается некоторое начальное множество утверждений, которые считаются априори доказанными (множество аксиом теории), и задается некоторое множество правил вывода, применяя которые к некоторым утверждениям (в частности, уже доказанным), можно получать новые утверждения. Если мы применяем правило вывода к доказанному утверждению, то получаем новое доказанное утверждение. Утверждение, которое таким образом может быть выведено из аксиом, называют теоремой данной теории. Утверждение считается имеющим смысл, если оно есть либо аксиома, либо теорема данной теории. В отличие от экстенсионального подхода при интенсиональном подходе априори не определяется никакой универсум, и при таком подходе к определению семантики "иметь смысл" означает "быть теоремой или аксиомой данной теории".

Вернемся к рассмотренному выше языку арифметических выражений.

Зададим его в виде формальной теории следующего вида.

Зададим язык арифметических выражений следующим образом:

1. аксиома — любой атом

a1, …, an;

1. правила вывода: e1, e2 ⇒ (e1+e2), (1) e1, e2⇒ (e1∗e2), (2)

Построим доказательство приведенного ранее арифметического выражения:

a,b,c,d,e,g аксиомы

(e+g) правило вывода (1);

(d(e+g)) правило вывода (2);

(c+(d(e+g))) правило вывода (1); (b(c+(d(e+g)))) правило вывода (2); (a+(b(c+(d(e+g))))) правило вывода (1).

Нетрудно видеть, что мы определили тот же язык другим способом. Множество доказуемых утверждений совпадает здесь с множеством всех утверждений, и, таким образом, семантика в данном случае совпала с синтаксисом: утверждение имеет смысл тогда и только тогда, когда может быть доказано, т.е. тогда и только тогда, когда является арифметическим выражением, порождаемым приведенной выше грамматикой.

В этой связи полезно заметить, что и правила вывода формальной теории следует трактовать как просто "**правила вывода**" (в полной аналогии с порождающими грамматиками), согласно которым разрешается от определенных цепочек (в левой части правила) переходить к новым цепочкам (в правой его части).

Тогда по правилам вывода можно получать из цепочек, не обязательно теорем или аксиом, какие-то другие цепочки, т.е. строить выводы, не являющиеся доказательствами. Это опять-таки аналогично грамматикам, в которых применение продукций, вообще говоря, не обязано быть выводом из аксиомы.

Такая широкая и чисто синтаксическая трактовка правил вывода позволяет в теории формальных систем и в связанной с нею теории доказательств строить выводы из "гипотез" — цепочек, которые предполагаются доказанными. Если затем удается действительно доказать их, то построенное первоначально "относительное" доказательство превратится в "абсолютное" (т.е. начинающееся с аксиом).

Построив аксиоматическую семантику языка, мы можем на этой базе определить семантику уже экстенсионально, положив, что экстенсионал утверждения есть множество всех его доказательств (если утверждение не имеет доказательства, то его экстенсионал считается не определенным — формально мы включаем тогда в предметную область специальный "неопределенный элемент" с тем, как уже отмечалось, чтобы можно было вводимую семантическую функцию считать отображением).

**5.3 Семантика простейшего языка программирования.**

В заключение рассмотрим экстенсиональное определение семантики простейшего языка программирования, который будем называть MILAN (MIniLANguage). Синтаксис языка определим посредством форм БэкусаНаура.

Определение семантики языка программирования MILAN

(MIniLANguage).

(**программа**) ::= (последовательность\_операторов)

(**последовательность\_операторов**) ::= (оператор) | (оператор) последовательность\_операторов)

(**оператор**) ::= (присваивание) | (условный лереход) | (цикл)

(**присваивание**) ::= (переменное) := (терм)

(**условный\_переход**) ::= if (условие) then(последовательность\_операторов) else (последовательность-операторов) | if (условие) then

последовательность\_операторов)

(**цикл**) ::= while (условие) do (последовательность\_операторов) end

(переменное) ::= zi | ... | zn

(**терм**) ::= (функциональный\_символ) ((последовательность\_термов)) | (переменное) | (константа)

(**последовательность\_термов**) ::= (терм) | (терм)

((последовательность\_термов))

(**функциональный\_символ**) ::= f1∣ … ∣fm

(**константа**) ::= c1∣… ∣ck

(**условие**) ::= (предикатный\_символ) ((последовательность\_термов))

(**предикатный\_символ**) ::= p1∣ … ∣ps

Заданная таким образом КС-грамматика определяет так называемый абстрактный синтаксис MILANa: мы игнорируем некоторые синтаксические детали, такие, как слова begin, end, обрамляющие последовательность операторов, а также то, что операторы разделяются точкой с запятой, что между термами в последовательности термов ставится запятая и т.п. Мы также не уточняем структуру переменных (идентификаторов), констант, функциональных и предикатных символов, считая, что эти нетерминалы "пробегают" каждый свой алфавит.

Чтобы формально описать экстенсиональную семантику введенного посредством написанных выше синтаксических правил языка, нужно определить универсум.

Различают:

1. **денотационную** семантику;
2. **операциональную** семантику;
3. **трансформационную** семантику (семантику смешанных вычислений).

Мы будем рассмотривать только денотационную семантику языка, подобного MILANy, универсум которой определяется как множество преобразователей состояний памяти.

***Состояние памяти*** *— это произвольное* ***отображение множества переменных* I** *языка программирования в* ***множество* D *данных (значений)****.*

В зависимости от того, какой универсум рассматривается, в рамках экстенсионального подхода различают:

В денотационной семантике языка, подобного MILANy, универсум определяется как множество преобразователей состояний памяти. К построению этого множества и переходим.

**Определение.** Состояние памяти — это произвольное отображение множества переменных I языка программирования в множество D данных (значений).

Т.о. будем отождествлять память с множеством переменных

(идентификаторов, "ячеек", имен) языка программирования I **σ: I→D,**

сопоставляющее каждой переменной значение, принадлежащее некоторому множеству значений (данных) D.

В множество D включен также неопределенный элемент **O**

(неопределенное значение).

Множество всех состояний памяти обозначим Σ.

В силу сформулированного определения мы отождествляем память с множеством переменных (идентификаторов, "ячеек", имен) данного языка программирования (конкретно — языка MILAN), а состояние памяти σ есть отображение вида σ: I→D, сопоставляющее каждой переменной значение, принадлежащее некоторому множеству значений (данных) D. Последнее можно рассматривать как объединение некоторого семейства множеств, служащих носителями многосортной алгебры данных языка. Их можно отождествить с типами данных, используемыми в рассматриваемом языке: числами, массивами, строками, стуктурами и т.п. В множество D включен также неопределенный элемент O (неопределенное значение). Множество всех состояний памяти обозначим Σ.

Введем понятие преобразователя состояний памяти.

**Преобразователь состояний памяти — это произвольное отображение множества Σ состояний памяти в себя.**

Множество всех преобразователей состояний памяти обозначим (**Σ →**

**Σ**).

На множестве Σ определяется структура индуктивного упорядоченного множества.

**Область определенности состояния памяти σ** - это множество, обозначаемое D(σ), всех переменных x, таких, что σ(x) ≠ **O**.

Тогда положим для двух произвольных состояний σ и ρ

Σ ⩽ ρ ⇔ (D(σ) ⊆ D(ρ)) & ((∀x ∈ D(σ)) (σ(x) = ρ(x))).

Множество Σ имеет по отношению ⩽ наименьший элемент, а именно такое состояние **O**, что (∀x ∈ I) (**O**(x) = **O**).

Его называют **всюду неопределенным состоянием памяти**; ясно, что D(O) = ∅.

Кроме того, для любой неубывающей последовательности состояний памяти σ1⩽σ2⩽ … ⩽σn⩽ …

состояние σ0, такое, что σ0(x) = σn(x) ⇔ x∈D(σn), есть точная верхняя грань этой последовательности, причем D(σ0) = ⋃n⩾1D(σn).

Таким образом, множество состояний памяти Σ, снабженное отношением порядка ⩽, **является индуктивным упорядоченным множеством**.

Из этого следует, что и множество преобразователей состояний (Σ → Σ) есть индуктивное упорядоченное множество. В этом множестве отношение порядка ⩽ определяется условием **f** ⩽ **g** ⇔ **(**∀**σ** ∈ **Σ) (f(σ)** ⩽ **g(σ))**,

наименьшим элементом является стирающий преобразователь 0, такой, что 0(σ)=**O** для всякого σ∈Σ, а точной верхней гранью неубывающей последовательности преобразователей состояний **f1** ⩽ **f2** ⩽ **…** ⩽ **fn** ⩽ **…** является преобразователь f0, определяемый

следующим образом:

**(**∀**σ** ∈**Σ) (**∀**x** ∈ **I)(f0(σ)(x) = fn (σ)(x)** ⇔ **x** ∈ **D(fn (σ)))**, причем **D(f0(σ)) =** ⋃**n**⩾**1 D(fn(σ))**.

Более того, оказывается, что композиция преобразователей состояний (как отображений) непрерывна в смысле сохранения точных верхних граней, точнее, **для любой** неубывающей последовательности преобразователей состояний f1 ⩽ f2 ⩽ … ⩽fn ⩽ … и произвольного преобразователя состояний g имеем…

Для любой неубывающей последовательности преобразователей состояний f1 ⩽ f2 ⩽ … ⩽fn ⩽ … и произвольного преобразователя состояний g имеем

**g**∘**supfn = sup(g**∘**fn), sup fn**∘**g = sup(fn**∘**g)**.

Докажем 1-е из этих равенств. Для произвольного σ∈Σ имеем

**g**∘**supfn(σ) = supfn (g(σ))**.

Тогда для всякого **x** ∈ **D(g**∘**fn(σ))**

имеем

**g**∘**supfn(σ)(x) = supfn(g(σ))(x) = fn (g(σ))(x) = g**∘**fn(σ)(x),**

т.е. поскольку, как можно показать, композиция монотонна в смысле

определения, данного ранее, то

**D(g**∘**supfn(σ)) =** ⋃**n**⩾**1D(g**∘**fn)(σ)**

и **g**∘**supfn = sup(g**∘**fn).**

**5.4 Денотационная семантика языка MILAN.**

Имея в виду все рассмотренные выше свойства множества (Σ→Σ), определим денотационную семантику языка MILAN.

Семантическая функция есть отображение [] множества фраз в множество (Σ→Σ) преобразователей состояний памяти.

**Семантика оператора присваивания:**



где через tσ обозначено значение терма t в состоянии σ, равное 0, если

хотя бы для одного переменного, входящего в терм, его значение не определено, и равное значению терма при подстановке на место каждого его переменного x его значения σ(x).

Суть записанного выше семантического правила очень проста: оператору присваивания x := t сопоставляется преобразователь состояний, меняющий состояние таким образом, что все переменные, кроме x, сохраняют свои значения, а новое значение переменного x есть значение правой части оператора присваивания (терма t) в состоянии σ.

**Семантика оператора условного перехода**



Это правило написано в предположении, что значения всех термов в состоянии σ определены. В противном случае оператору условного перехода сопоставляется стирающий преобразователь . Подобное же соглашение принимается и далее в аналогичных ситуациях.

Семантика условного перехода без else-альтернативы записывается аналогично, но при условии ложности предиката p в состоянии σ берется тождественный преобразователь состояний ID.

**Семантика цикла**



Таким образом, семантика цикла определяется как решение уравнения. Это определение корректно, так как правая часть уравнения есть непрерывное отображение множества (Σ→Σ) в себя (что следует из доказанной выше непрерывности композиции преобразователей состояний и очевидной непрерывности тождественного отображения).

**Семантика последовательности операторов**:

[S1;S2] = [S1]  [S2].

Определенная таким образом формальная семантика языка MILAN становится базой для семантического анализа программ, т.е. для строгого математического доказательства утверждений о программах. Этот анализ, проводимый технологически после синтаксического анализа программы и получения ее дерева вывода, основан на результате, известном под названием принципа индукции по не неподвижной точке.

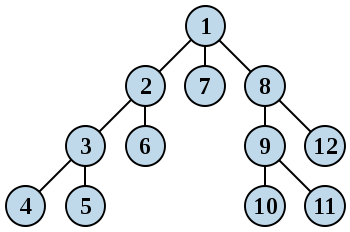
**МОДУЛЬ 3**

# Раздел 6. Методы построения трансляторов

В процессе трансляции компилятор часто используют промежуточное представление (ПП) исходной программы, предназначенное прежде всего для удобства генерации кода и/или проведения различных оптимизаций. Сама форма ПП зависит от целей его использования.

Наиболее часто используемыми формами ПП является ориентированный граф (в частности, абстрактное синтаксическое дерево, в том числе атрибутированное), трехадресный код (в виде троек или четверок), префиксная и постфиксная запись.

## 6.1 Представление в виде ориентированного графа

*Например,* [*правое скобочное представление для дерева*](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B0%D0%B2%D0%BE%D0%B5_%D1%81%D0%BA%D0%BE%D0%B1%D0%BE%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) *на иллюстрации:****(((4 5)3 6)2 7 ((10 11)9 12)8)1*** **

Простейшей формой промежуточного представления является синтаксическое дерево программы. Ту же самую информацию о входной программе, но в более компактной форме дает ориентированный ациклический граф (ОАГ), в котором в одну вершину объединены вершины синтаксического дерева, представляющие общие подвыражения. Синтаксическое дерево и ОАГ для оператора присваивания a := b \*-c + b \*-c приведены на рис. 8.1.

На рис. 8.1 приведены два представления в памяти синтаксического дерева на рис. 8.1, а. Каждая вершина кодируется записью с полем для операции и полями для указателей на потомков(дерево ссылок). На рис. 8.2, б, вершины размещены в массиве записей и индекс (или вход) вершины служит указателем на нее. (дерево термов???)

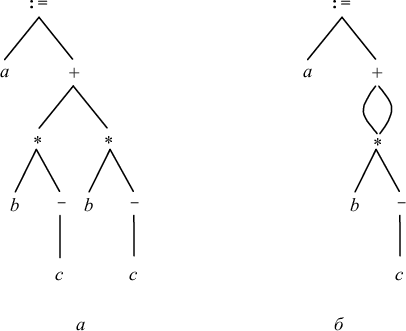


Рис. 8.1

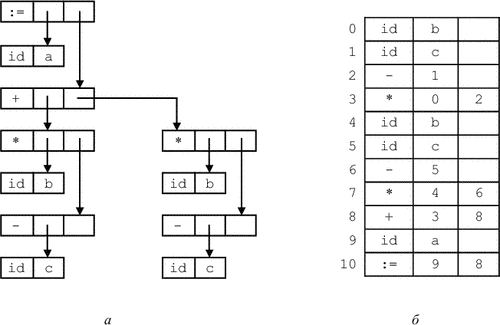


Рис. 8.2

## 6.2 Трехадресный код

Трехадресный код - это последовательность операторов вида x := y op z, где x, y и z - имена, константы или сгенерированные компилятором временные объекты. Здесь op - двуместная операция, например, операция плавающей или фиксированной арифметики, логическая или побитовая. В правую часть может входить только один знак операции.

Составные выражения должны быть разбиты на подвыражения, при этом могут появиться временные имена (переменные). Смысл термина «трехадресный код» в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата. Трехадресный код - это линеаризованное представление синтаксического дерева или ОАГ, в котором временные имена соответствуют внутренним вершинам дерева или графа. Например, выражение x+y\*z может быть протранслировано в последовательность операторов

t1 := y \* z

t2 := x + t1

где t1 и t2 - имена, сгенерированные компилятором.

В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор

if A > B then S1 else S2

может быть представлен следующим кодом:

t := A - B JGT t, S2

...

Здесь JGT - двуместная операция условного перехода, не вырабатывающая результата.

Разбиение арифметических выражений и операторов управления делает трехадресный код удобным при генерации машинного кода и оптимизации. Использование имен промежуточных значений, вычисляемых в программе, позволяет легко переупорядочивать трехадресный код.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 8.3

Представления синтаксического дерева и графа рис. 8.1 в виде трехадресного кода дано на рис. 8.3, а, и 8.3, б, соответственно.

Трехадресный код - это абстрактная форма промежуточного кода. В реализации трехадресный код может быть представлен записями с полями для операции и операндов. Рассмотрим три способа реализации трехадресного кода: четверки, тройки и косвенные тройки.

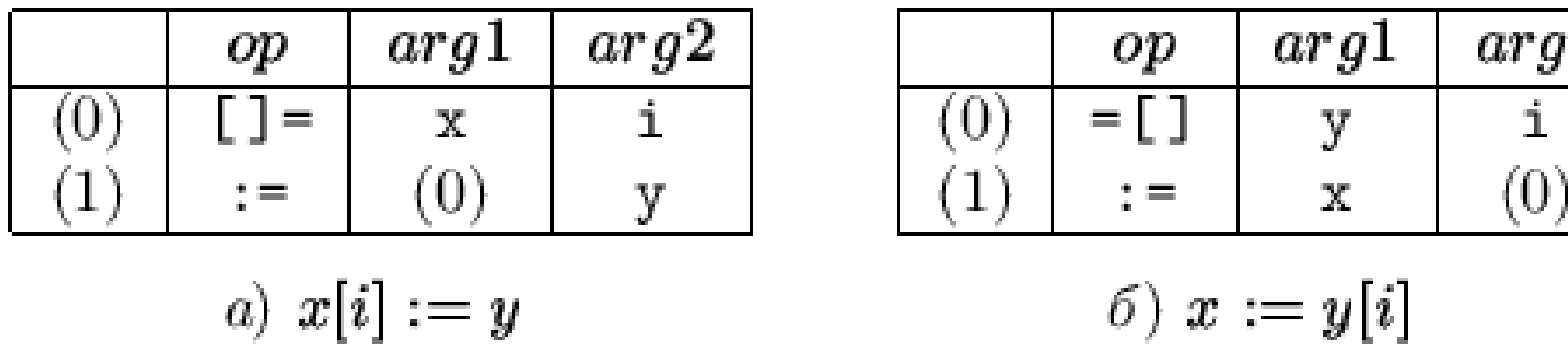
Изображение выглядит как стол

Автоматически созданное описаниеЧетверка - это запись с четырьмя полями, которые будем называть op, arg1, arg2 и result. Поле op содержит код операции. В операторах с унарными операциями типа x := -y или x := y поле arg2 не используется. В некоторых операциях (типа «передать параметр») могут не использоваться ни arg2, ни result. Условные и безусловные переходы помещают в result метку перехода. На рис. 8.4, а, приведены четверки для оператора присваивания a := b \*-c + b \*-c. Они получены из трехадресного кода на рис. 8.3, а.

Рис. 8.4

Обычно содержимое полей arg1, arg2 и result - это указатели на входы таблицы символов для имен, представляемых этими полями. Временные имена вносятся в таблицу символов по мере их генерации.

Чтобы избежать внесения новых имен в таблицу символов, на временное значение можно ссылаться, используя позицию вычисляющего его оператора. В этом случае трехадресные операторы могут быть представлены записями только с тремя полями: op, arg1 и arg2, как это показано на рис. 8.3, б. Поля arg1 и arg2 - это либо указатели на таблицу символов (для имен, определенных программистом, или констант), либо указатели на тройки (для временных значений). Такой способ представления трехадресного кода называют тройками. Тройки соответствуют представлению синтаксического дерева или ОАГ с помощью массива вершин.

Числа в скобках - это указатели на тройки, а имена - это указатели на таблицу символов. На практике информация, необходимая для интерпретации различного типа входов в поля arg1 и arg2, кодируется в поле op или дополнительных полях. Тройки рис. 8.4, б, соответствуют четверкам рис. 8.4, а.

Для представления тройками трехместной операции типа x[i] := y требуется два входа, как это показано на рис. 8.5, а, представление x := y[i] двумя операциями показано на рис. 8.5, б.

Трехадресный код может быть представлен не списком троек, а списком указателей на них. Такая реализация обычно называется косвенными тройками. Например, тройки рис. 8.4, б, могут быть реализованы так, как это изображено на рис. 8.6.

Рис. 8.6

Рис. 8.7

При генерации объектного кода каждой переменной, как временной, так и определенной в исходной программе, назначается память периода исполнения, адрес которой обычно хранится в таблице генератора кода. При использовании четверок этот адрес легко получить через эту таблицу.

Изображение выглядит как стол

Автоматически созданное описаниеБолее существенно преимущество четверок проявляется в оптимизирующих компиляторах, когда может возникнуть необходимость перемещать операторы. Если перемещается оператор, вычисляющий x, не требуется изменений в операторе, использующем x. В записи же тройками перемещение оператора, определяющего временное значение, требует изменения всех ссылок на этот оператор в массивах arg1 и arg2. Из-за этого тройки трудно использовать в оптимизирующих компиляторах.

В случае применения косвенных троек оператор может быть перемещен переупорядочиванием списка операторов. При этом не надо менять указатели на op, arg1 и arg2. Этим косвенные тройки похожи на четверки. Кроме того, эти два способа требуют примерно одинаковой памяти. Как и в случае простых троек, при использовании косвенных троек выделение памяти для временных значений может быть отложено на этап генерации кода. По сравнению с четверками при использование косвенных троек можно сэкономить память, если одно и то же временное значение используется более одного раза. Например, на рис. 8.6 можно объединить строки (14) и (16), после чего можно объединить строки (15) и (17).

## 6.3 Линеаризованные представления

В качестве промежуточных представлений весьма распространены линеаризованные представления деревьев. Линеаризованное представление позволяет относительно легко хранить промежуточное представление во внешней памяти и обрабатывать его. Наиболее распространенной формой линеаризованного представления является польская запись - префиксная (прямая) или постфиксная (обратная).

Постфиксная запись - это список вершин дерева, в котором каждая вершина следует (при обходе снизу-вверх слева-направо) непосредственно за своими потомками. Дерево на рис. 8.1, а, в постфиксной записи может быть представлено следующим образом:



В постфиксной записи вершины синтаксического дерева явно не присутствуют. Они могут быть восстановлены из порядка, в котором следуют вершины и из числа операндов соответствующих операций. Восстановление вершин аналогично вычислению выражения в постфиксной записи с использованием стека.

В префиксной записи сначала указывается операция, а затем ее операнды.

Например, для приведенного выше выражения имеем



Рассмотрим детальнее одну из реализаций префиксного представления - Лидер [9]. Лидер - это аббревиатура от «ЛИнеаризованное ДЕРево». Это машинно-независимая префиксная запись. В Лидере сохраняются все объявления и каждому из них присваивается свой уникальный номер, который используется для ссылки на объявление. Рассмотрим пример.

module M;

var X,Y,Z: integer;

procedure DIF(A,B:integer):integer;

var R:integer; begin R:=A-B; return(R); end DIF; begin Z:=DIF(X,Y); end M.

Этот фрагмент имеет следующий образ в Лидере.

program 'M'

var int var int var int

procbody proc int int end int

var int

begin assign var 1 7 end int int mi par 1 5 end par 1 6 end result 0 int var 1 7 end

return

end

begin assign var 0 3 end int icall 0 4 int var 0 1 end int var 0 2 end end end

Рассмотрим его более детально:

program Имя модуля нужно для редактора 'M' связей.

var int Это образ переменных X, Y, Z; var int переменным X, Y, Z присваиваются номера

var int 1, 2, 3 на уровне 0.

procbod Объявление процедуры с двумя

y proc

int int e целыми параметрами, возвращающей nd целое.

int Процедура получает номер 4 на уровне

0 и

параметры имеют номера 5, 6 на уровне 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | var int | 1. | Переменная R имеет номер 7 на уровне | |
|  | begin |  | Начало тела процедуры. | |
|  | assign |  | Оператор присваивания. | |
| nd | var 1 7 e |  | Левая часть присваивания (R). | |
|  | int |  | Тип присваиваемого значения. | |
|  | int mi |  | Целое вычитание. | |
| nd | par 1 5 e |  | Уменьшаемое (A). | |
| nd | par 1 6 e |  | Вычитаемое (B). | |
|  | result 0 |  | Результат процедуры уровня 0. | |
|  | int |  | Результат имеет целый тип. | |
| nd | var 1 7 e |  | Результат - переменная R. | |
|  | return |  | Оператор возврата. | |
|  | end |  | Конец тела процедуры. | |
|  | begin |  | Начало тела модуля. | |
|  | assign |  | Оператор присваивания. | |
| nd | var 0 3 e |  | Левая часть - переменная Z. | |
|  | int |  | Тип присваиваемого значения. | |
|  | icall 0 4 |  | Вызов локальной процедуры DIF. | |
| int var 0 1 end | | | Фактические параметры X |
| int var 0 2 end | | | и Y. |
| end | | | Конец вызова. |
| end | | | Конец тела модуля. |

## 6.4 Виртуальная машина Java

Программы на языке Java транслируются в специальное промежуточное представление, которое затем интерпретируется так называемой «виртуальной машиной Java». Виртуальная машина Java представляет собой стековую машину: она не имеет памяти прямого доступа, все операции выполняются над операндами, расположенными на верхушке стека. Чтобы, например, выполнить операцию с участием константы или переменной, их предварительно необходимо загрузить на верхушку стека. Код операции - всегда один байт. Если операция имеет операнды, они располагаются в следующих байтах.

К элементарным типам данных, с которыми работает машина, относятся short, integer, long, float, double (все знаковые).

### 6.4.1 Организация памяти

Машина имеет следующие регистры:

pc - счетчик команд;

optop - указатель вершины стека операций; frame - указатель на стек-фрейм исполняемого метода; vars - указатель на 0-ю переменную исполняемого метода.

Все регистры 32-разрядные. Стек-фрейм имеет три компоненты: локальные переменные, среду исполнения, стек операндов. Локальные переменные отсчитываются от адреса в регистре vars. Среда исполнения служит для поддержания самого стека. Она включает указатель на предыдущий фрейм, указатель на собственные локальные переменные, на базу стека операций и на верхушку стека. Кроме того, здесь же хранится некоторая дополнительная информация, например, для отладчика.

Куча сборки мусора содержит экземпляры объектов, которые создаются и уничтожаются автоматически. Область методов содержит коды, таблицы символов и т.д.

С каждым классом связана область констант. Она содержит имена полей, методов и другую подобную информацию, которая используется методами.

### 6.4.2 Набор команд виртуальной машины

Виртуальная Java-машина имеет следующие команды:

помещение констант на стек,

помещение локальных переменных на стек, запоминание значений из стека в локальных переменных, обработка массивов, управление стеком, арифметические команды, логические команды, преобразования типов, передача управления, возврат из функции, табличный переход, обработка полей объектов, вызов метода, обработка исключительных ситуаций, прочие операции над объектами, мониторы, отладка.

Рассмотрим некоторые команды подробнее.

### 6.4.3 Помещение локальных переменных на стек

Команда iload - загрузить целое из локальной переменной. Операндом является смещение переменной в области локальных переменных. Указываемое значение копируется на верхушку стека операций. Имеются аналогичные команды для помещения плавающих, двойных целых, двойных плавающих и т.д.

Команда istore - сохранить целое в локальной переменной. Операндом операции является смещение переменной в области локальных переменных. Значение с верхушки стека операций копируется в указываемую область локальных переменных. Имеются аналогичные команды для помещения плавающих, двойных целых, двойных плавающих и т.д.

**6.4.4 Вызов метода** Команда invokevirtual.

При трансляции объектно-ориентированных языков программирования из-за возможности перекрытия виртуальных методов, вообще говоря, нельзя статически протранслировать вызов метода объекта. Это связано с тем, что если метод перекрыт в производном классе, и вызывается метод объектапеременной, то статически неизвестно, объект какого класса (базового или производного) хранится в переменной. Поэтому с каждым объектом связывается таблица всех его виртуальных методов: для каждого метода там помещается указатель на его реализацию в соответствии с принадлежностью самого объекта классу в иерархии классов.

В языке Java различные классы могут реализовывать один и тот же интерфейс. Если объявлена переменная или параметр типа интерфейс, то динамически нельзя определить объект какого класса присвоен переменной: interface I;

class C1 implements I; class C2 implements I;

I O;

C1 O1;

C2 O2;

...

O=O1;

...

O=O2;

...

В этой точке программы, вообще говоря, нельзя сказать, какого типа значение хранится в переменной O. Кроме того, при работе программы на языке Java имеется возможность использования методов из других пакетов. Для реализации этого механизма в Java-машине используется динамическое связывание.

Предполагается, что стек операндов содержит handle объекта или массива и некоторое количество аргументов. Операнд операции используется для конструирования индекса в области констант текущего класса. Элемент по этому индексу в области констант содержит полную сигнатуру метода. Сигнатура метода описывает типы параметров и возвращаемого значения. Из handle объекта извлекается указатель на таблицу методов объекта. Просматривается сигнатура метода в таблице методов. Результатом этого просмотра является индекс в таблицу методов именованного класса, для которого найден указатель на блок метода. Блок метода указывает тип метода (native, synchronized и т.д.) и число аргументов, ожидаемых на стеке операндов.

Если метод помечен как synchronized, запускается монитор, связанный с handle. Базис массива локальных переменных для нового стек-фрейма устанавливается так, что он указывает на handle на стеке. Определяется общее число локальных переменных, используемых методом, и после того, как отведено необходимое место для локальных переменных, окружение исполнения нового фрейма помещается на стек. База стека операндов для этого вызова метода устанавливается на первое слово после окружения исполнения. Затем исполнение продолжается с первой инструкции вызванного метода.

### 6.4.5 Обработка исключительных ситуаций

Команда athrow - возбудить исключительную ситуацию.

С каждым методом связан список операторов catch. Каждый оператор catch описывает диапазон инструкций, для которых он активен, тип исключения, который он обрабатывает. Кроме того, с оператором связан набор инструкций, которые его реализуют. При возникновении исключительной ситуации просматривается список операторов catch, чтобы установить соответствие. Исключительная ситуация соответствует оператору catch, если инструкция, вызвавшая исключительную ситуацию, находится в соответствующем диапазоне и исключительная ситуация принадлежит подтипу типа ситуации, которые обрабатывает оператор catch. Если соответствующий оператор catch найден, управление передается обработчику. Если нет, текущий стек-фрейм удаляется, и исключительная ситуация возбуждается вновь.

Порядок операторов catch в списке важен. Интерпретатор передает управление первому подходящему оператору catch.

## 6.5 Организация информации в генераторе кода

Синтаксическое дерево в чистом виде несет только информацию о структуре программы. На самом деле в процессе генерации кода требуется также информация о переменных (например, их адреса), процедурах (также адреса, уровни), метках и т.д. Для представления этой информации возможны различные решения. Наиболее распространены два:  информация хранится в таблицах генератора кода;

 информация хранится в соответствующих вершинах дерева.

Рассмотрим, например, структуру таблиц, которые могут быть использованы в сочетании с Лидер-представлением. Поскольку Лидерпредставление не содержит информации об адресах переменных, значит, эту информацию нужно формировать в процессе обработки объявлений и хранить в таблицах. Это касается и описаний массивов, записей и т.д. Кроме того, в таблицах также должна содержаться информация о процедурах (адреса, уровни, модули, в которых процедуры описаны, и т.д.).

При входе в процедуру в таблице уровней процедур заводится новый вход - указатель на таблицу описаний. При выходе указатель восстанавливается на старое значение. Если промежуточное представление - дерево, то информация может храниться в вершинах самого дерева.

## 6.6 Уровень промежуточного представления

Как видно из приведенных примеров, промежуточное представление программы может в различной степени быть близким либо к исходной программе, либо к машине. Например, промежуточное представление может содержать адреса переменных, и тогда оно уже не может быть перенесено на другую машину. С другой стороны, промежуточное представление может содержать раздел описаний программы, и тогда информацию об адресах можно извлечь из обработки описаний. В то же время ясно, что первое более эффективно, чем второе. Операторы управления в промежуточном представлении могут быть представлены в исходном виде (в виде операторов языка if, for, while и т.д.), а могут содержаться в виде переходов. В первом случае некоторая информация может быть извлечена из самой структуры (например, для оператора for - информация о переменной цикла, которую, может быть, разумно хранить на регистре, для оператора case - информация о таблице меток и т.д.). Во втором случае представление проще и унифицированней.

Некоторые формы промежуточного представления удобны для различного рода оптимизаций, некоторые - нет (например, косвенные тройки, в отличие от префиксной записи, позволяют эффективное перемещение кода).

# 7. Генерация кода

Задача генератора кода - построение для программы на входном языке эквивалентной машинной программы. Обычно в качестве входа для генератора кода служит некоторое промежуточное представление программы.

Генерация кода включает ряд специфических, относительно независимых подзадач: распределение памяти (в частности, распределение регистров), выбор команд, генерацию объектного (или загрузочного) модуля. Конечно, независимость этих подзадач относительна: например, при выборе команд нельзя не учитывать схему распределения памяти, и, наоборот, схема распределения памяти (регистров, в частности) ведет к генерации той или иной последовательности команд. Однако удобно и практично эти задачи все же разделять, обращая при этом внимание на их взаимодействие.

В какой-то мере схема генератора кода зависит от формы промежуточного представления. Ясно, что генерация кода из дерева отличается от генерации кода из троек, а генерация кода из префиксной записи отличается от генерации кода из ориентированного графа. В то же время все генераторы кода имеют много общего, и основные применяемые алгоритмы отличаются, как правило, только в деталях, связанных с используемым промежуточным представлением.

В дальнейшем в качестве промежуточного представления мы будем использовать префиксную нотацию. А именно, алгоритмы генерации кода будем излагать в виде атрибутных схем со входным языком Лидер.

## 7.1 Модель машины

При изложении алгоритмов генерации кода мы будем следовать некоторой модели машины, в основу которой положена система команд микропроцессора Motorola MC68020. В микропроцессоре имеется регистр - счетчик команд PC, 8 регистров данных и 8 адресных регистров.

В системе команд используются следующие способы адресации:

ABS - абсолютная: исполнительным адресом является значение адресного выражения.

IMM - непосредственный операнд: операндом команды является константа, заданная в адресном выражении.

D - прямая адресация через регистр данных, записывается как Хn, операнд находится в регистре Хn.

А - прямая адресация через адресный регистр, записывается как An, операнд находится в регистре An.

INDIRECT - записывается как (An), адрес операнда находится в адресном регистре An.

POST - пост-инкрементная адресация, записывается как (Аn)+, исполнительный адрес есть значение адресного регистра An и после исполнения команды значение этого регистра увеличивается на длину операнда.

PRE - пре-инкрементная адресация, записывается как -(Аn): перед исполнением операции содержимое адресного регистра An уменьшается на длину операнда, исполнительный адрес равен новому содержимому адресного регистра.

INDISP - косвенная адресация со смещением, записывается как (bd,An), исполнительный адрес вычисляется как (An)+d - содержимое An плюс d.

INDEX - косвенная адресация с индексом, записывается как (bd,An,Xn\*sc), исполнительный адрес вычисляется как (An)+bd+(Xn)\*sc - содержимое адресного регистра + адресное смещение + содержимое индексного регистра, умноженное на sc.

INDIRPC - косвенная через PC (счетчик команд), записывается как (bd,PC), исполнительный адрес определяется выражением (PC)+bd. INDEXPC - косвенная через PC со смещением, записывается как (bd,PC,Xn\*sc), исполнительный адрес определяется выражением

(PC)+bd+(Xn)\*sc.

INDPRE - пре-косвенная через память, записывается как ([bd,An,sc\*Xn],od) (схема вычисления адресов для этого и трех последующих способов адресации приведена ниже).

INDPOST - пост-косвенная через память: ([bd,An],sc\*Xn,od).

INDPREPC - пре-косвенная через PC: ([bd,PC,sc\*Xn],od). INDPOSTPC - пост-косвенная через PC: ([bd,PC],Xn,od).

Здесь bd - это 16- или 32-битная константа, называемая смещением, od - 16- или 32-битная литеральная константа, называемая внешним смещением. Эти способы адресации могут использоваться в упрощенных формах без смещений bd и/или od и без регистров An или Xn. Следующие примеры иллюстрируют косвенную постиндексную адресацию: MOVE D0, ([A0]) MOVE D0, ([4,A0])

MOVE D0, ([A0],6)

MOVE D0, ([A0],D3)

MOVE D0, ([A0],D4,12)

MOVE D0, ([$12345678,A0],D4,$FF000000)

Индексный регистр Xn может масштабироваться (умножаться) на 2,4,8, что записывается как sc\*Xn. Например, в исполнительном адресе ([24,A0,4\*D0]) содержимое квадратных скобок вычисляется как

[A0] + 4 \* [D0] + 24.

Эти способы адресации работают следующим образом. Каждый исполнительный адрес содержит пару квадратных скобок [...] внутри пары круглых скобок, т.е. ([...], ... ). Сначала вычисляется содержимое квадратных скобок, в результате чего получается 32-битный указатель. Например, если используется постиндексная форма [20,A2], то исполнительный адрес - это

20 + [A2]. Аналогично, для преиндексной формы [12,A4,D5] исполнительный адрес - это 12 + [A4] + [D5].

Указатель, сформированный содержимым квадратных скобок, используется для доступа в память, чтобы получить новый указатель (отсюда термин косвенная адресация через память). К этому новому указателю добавляется содержимое внешних круглых скобок и таким образом формируется исполнительный адрес операнда.

В дальнейшем изложении будут использованы следующие команды (в частности, рассматриваются только арифметические команды с целыми операндами, но не с плавающими):

MOVEA ИА, А - загрузить содержимое по исполнительному адресу ИА на адресный регистр А.

MOVE ИА1, ИА2 - содержимое по исполнительному адресу ИА1 переписать по исполнительному адресу ИА2.

MOVEM список\_регистров, ИА - сохранить указанные регистры в памяти, начиная с адреса ИА (регистры указываются маской в самой команде).

MOVEM ИА, список\_регистров - восстановить указанные регистры из памяти, начиная с адреса ИА (регистры указываются маской в самой команде).

LEA ИА, А - загрузить исполнительный адрес ИА на адресный регистр

А.

MUL ИА, D - умножить содержимое по исполнительному адресу ИА на содержимое регистра данных D и результат разместить в D (на самом деле в системе команд имеются две различные команды MULS и MULU для чисел со знаком и чисел без знака соответственно; для упрощения мы не будем принимать во внимание это различие).

DIV ИА, D - разделить содержимое регистра данных D на содержимое по исполнительному адресу ИА и результат разместить в D.

ADD ИА, D - сложить содержимое по исполнительному адресу ИА с содержимым регистра данных D и результат разместить в D.

SUB ИА, D - вычесть содержимое по исполнительному адресу ИА из содержимого регистра данных D и результат разместить в D.

Команды CMP и TST формируют разряды регистра состояний. Всего имеется 4 разряда: Z - признак нулевого результата, N - признак отрицательного результата, V - признак переполнения, C - признак переноса.

CMP ИА, D - из содержимого регистра данных D вычитается содержимое по исполнительному адресу ИА, при этом формируется все разряды регистра состояний, но содержимое регистра D не меняется.

TST ИА - выработать разряд Z регистра состояний по значению, находящемуся по исполнительному адресу ИА.

BNE ИА - условный переход по признаку Z = 1 (не равно) по исполнительному адресу ИА.

BEQ ИА - условный переход по признаку Z = 0 (равно) по исполнительному адресу ИА.

BLE ИА - условный переход по признаку N or Z (меньше или равно) по исполнительному адресу ИА.

BGT ИА - условный переход по признаку not N (больше) по исполнительному адресу ИА.

BLT ИА - условный переход по признаку N (меньше) по исполнительному адресу ИА.

BRA ИА - безусловный переход по адресу ИА.

JMP ИА - безусловный переход по исполнительному адресу.

RTD размер\_локальных - возврат из подпрограммы с указанием размера локальных.

LINK A, размер\_локальных - в стеке сохраняется значение регистра А, в регистр А заносится указатель на это место в стеке и указатель стека продвигается на размер локальных.

UNLK A - стек сокращается на размер локальных и регистр А восстанавливается из стека.

## 7.2 Динамическая организация памяти

Динамическая организация памяти - это организация памяти периода исполнения программы. Оперативная память программы обычно состоит из нескольких основных разделов: стек (магазин), куча, область статических данных (инициализированных и неинициализированных). Наиболее сложной является работа со стеком. Вообще говоря, стек периода исполнения необходим для программ не на всех языках программирования. Например, в ранних версиях Фортрана нет рекурсии, так что программа может исполняться без стека. С другой стороны, исполнение программы с рекурсией может быть реализовано и без стека (того же эффекта можно достичь, например, и с помощью списковых структур). Однако, для эффективной реализации пользуются стеком, который, как правило, поддерживается на уровне машинных команд.

Рассмотрим схему организации магазина периода выполнения для простейшего случая (как, например, в языке Паскаль), когда все переменные в магазине (фактические параметры и локальные переменные) имеют известные при трансляции смещения. Магазин служит для хранения локальных переменных (и параметров) и обращения к ним в языках, допускающих рекурсивные вызовы процедур. Еще одной задачей, которую необходимо решать при трансляции языков с блочной структурой - обеспечение реализации механизмов статической вложенности. Пусть имеется следующий фрагмент программы на Паскале:

procedure P1;

var V1; procedure P2; var V2; begin

...

P2;

V1:=...

V2:=... ... end; begin

... P2; ...

end;

В процессе выполнения этой программы, находясь в процедуре P2, мы должны иметь доступ к последнему экземпляру значений переменных процедуры P2 и к экземпляру значений переменных процедуры P1, из которой была вызвана P2. Кроме того, необходимо обеспечить восстановление состояния программы при завершении выполнения процедуры.

Мы рассмотрим две возможные схемы динамической организации памяти: схему со статической цепочкой и с дисплеем в памяти. В первом случае все статические контексты связаны в список, который называется статической цепочкой; в каждой записи для процедуры в магазине хранится указатель на запись статически охватывающей процедуры (помимо, конечно, указателя динамической цепочки - указателя на «базу» динамически предыдущей процедуры). Во втором случае для хранения ссылок на статические контексты используется массив, называемый дисплеем. Использование той или иной схемы определяется, помимо прочих условий, прежде всего числом адресных регистров.

### 7.2.1 Организация магазина со статической цепочкой

Итак, в случае статической цепочки магазин организован, как это изображено на рис. 7.1.

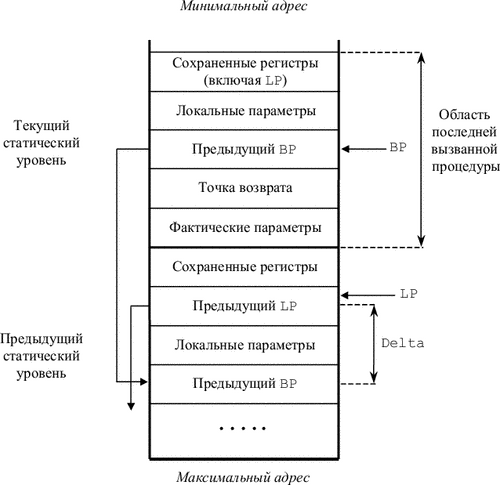


Рис. 7.1

Таким образом, на запись текущей процедуры в магазине указывает регистр BP (Base Pointer), с которого начинается динамическая цепочка. На статическую цепочку указывает регистр LP (Link Pointer). В качестве регистров BP и LP в различных системах команд могут использоваться универсальные, адресные или специальные регистры. Локальные переменные отсчитываются от регистра BP вверх, фактические параметры - вниз с учетом памяти, занятой точкой возврата и самим сохраненным регистром BP.

Вызов подпрограмм различного статического уровня производится несколько по-разному. При вызове подпрограммы того же статического уровня, что и вызывающая подпрограмма (например, рекурсивный вызов той же самой подпрограммы), выполняются следующие команды:

Занесение фактических параметров в магазин

JSR A

Команда JSR A продвигает указатель SP, заносит PC на верхушку магазина и осуществляет переход по адресу A. После выполнения этих команд состояние магазина становится таким, как это изображено на рис. 9.2. Занесение BP, отведение локальных, сохранение регистров делает вызываемая подпрограмма (см. ниже).

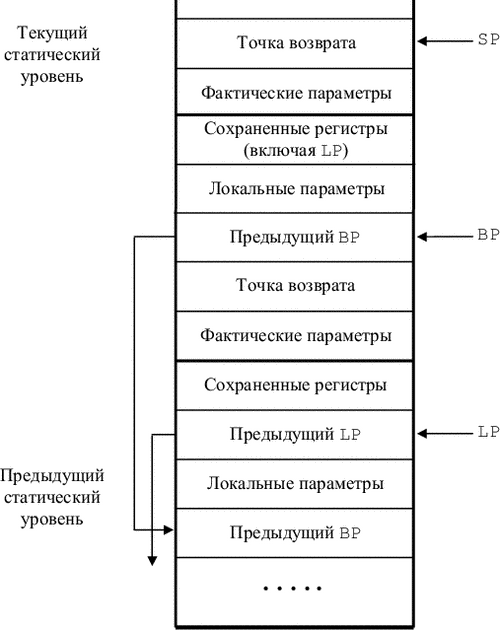


Рис. 7.2

При вызове локальной подпрограммы необходимо установить указатель статического уровня на текущую подпрограмму, а при выходе - восстановить его на старое значение (охватывающей текущую). Для этого исполняются следующие команды:

Занесение фактических параметров в магазин

MOVE BP, LP

SUB Delta, LP

JSR A

Здесь Delta - размер локальных вызывающей подпрограммы плюс двойная длина слова. Магазин после этого принимает состояние, изображенное на рис. 7.3. Предполагается, что регистр LP уже сохранен среди сохраняемых регистров, причем самым первым (сразу после локальных переменных).

После выхода из подпрограммы в вызывающей подпрограмме

выполняется команда MOVE (LP), LP

которая восстанавливает старое значение статической цепочки. Если

выход осуществлялся из подпрограммы 1-го уровня, эту команду выполнять не надо, поскольку для 1-го уровня нет статической цепочки.

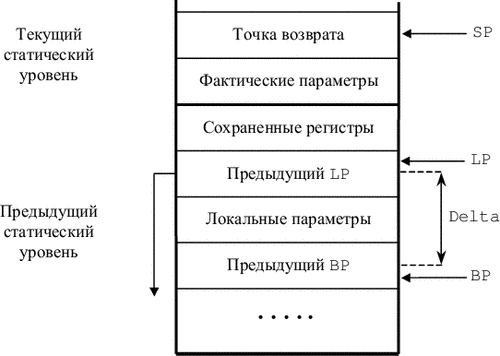


Рис. 7.3

При вызове подпрограммы меньшего, чем вызывающая, уровня выполняются следующие команды:

Занесение фактических параметров в магазин MOVE (LP), LP /\* столько раз, какова разность уровней вызывающей и вызываемой ПП \*/

JSR A

Тем самым устанавливается статический уровень вызываемой подпрограммы. После выхода из подпрограммы выполняется команда MOVE -Delta(BP), LP восстанавливающая статический уровень вызывающей подпрограммы.

Тело подпрограммы начинается со следующих команд:

LINK BP , -размер\_локальных MOVEM -(SP)

Команда LINK BP, размер\_локальных эквивалентна трем командам: MOVE BP, -(SP) MOVE SP, BP

ADD -размер\_локальных, SP

Команда MOVEM сохраняет в магазине регистры.

В результате выполнения этих команд магазин приобретает вид, изображенный на рис. 9.1.

Выход из подпрограммы осуществляется следующей последовательностью команд: MOVEM (SP)+

UNLK BP

RTD размер\_фактических

Команда MOVEM восстанавливает регистры из магазина. Команда UNLK BP эквивалентна такой последовательности команд: MOVE BP,SP

MOVE (SP),BP

ADD #4, SP /\* 4 - размер слова \*/

Команда RTD размер\_фактических, в свою очередь, эквивалентна последовательности

ADD размер\_фактических+4, SP JMP -размер\_фактических-4(SP)

После ее выполнения магазин восстанавливается до состояния, которое было до вызова.

В зависимости от наличия локальных переменных, фактических параметров и необходимости сохранения регистров каждая из этих команд может отсутствовать.

### 7.2.2 Организация магазина с дисплеем

Рассмотрим теперь организацию магазина с дисплеем. Дисплей - это массив (DISPLAY) , i-й элемент которого представляет собой указатель на область активации последней вызванной подпрограммы i-го статического уровня. Доступ к переменным самой внутренней подпрограммы осуществляется через регистр BP. Дисплей может быть реализован либо через регистры (если их достаточно), либо через массив в памяти.

При вызове процедуры следующего (по отношению к вызывающей) уровня в дисплее отводится очередной элемент. Если вызывающая процедура имеет статический уровень i, то при вызове процедуры уровня j i элементы дисплея j, ..., i должны быть скопированы (обычно в стек вызывающей процедуры), текущим уровнем становится j и в DISPLAY[j] заносится указатель на область активации вызываемой процедуры. По окончании работы вызываемой процедуры содержимое дисплея восстанавливается из стека.

Иногда используется комбинированная схема - дисплей в магазине. Дисплей хранится в области активации каждой процедуры. Формирование дисплея для процедуры осуществляется в соответствии с правилами, описанными выше.

Отдельного рассмотрения требует вопрос о технике передачи фактических параметров. Конечно, в случае простых параметров (например, чисел) проблем не возникает. Однако передача массивов по значению - операция довольно дорогая, поэтому с точки зрения экономии памяти целесообразнее сначала в подпрограмму передать адрес массива, а затем уже из подпрограммы по адресу передать в магазин сам массив. В связи с передачей параметров следует упомянуть еще одно обстоятельство.

Рассмотренная схема организации магазина допустима только для языков со статически известными размерами фактических параметров. Однако, например, в языке Модула-2 по значению может быть передан гибкий массив, и в этом случае нельзя статически распределить память для параметров. Обычно в таких случаях заводят так называемый «паспорт» массива, в котором хранится вся необходимая информация, а сам массив размещается в магазине в рабочей области выше сохраненных регистров.

## 7.3 Назначение адресов

Назначение адресов переменным, параметрам и полям записей происходит при обработке соответствующих объявлений. В однопроходном трансляторе это может производиться вместе с построением основной таблицы символов и соответствующие адреса (или смещения) могут храниться в этой же таблице. В промежуточном представлении Лидер объявления сохранены, что делает это промежуточное представление машинно-независимым. Напомним, что в Лидер-представлении каждому описанию соответствует некоторый номер. В процессе работы генератора кодов поддерживается таблица Table, в которой по этому номеру (входу) содержится следующая информация:

* для типа: его размер;
* для переменной: смещение в области процедуры (или глобальной области);
* для поля записи: смещение внутри записи;
* для процедуры: размер локальных параметров;
* для массива: размер массива, размер элемента, значение левой и правой границы.

Функция IncTab вырабатывает указатель (вход) на новый элемент таблицы, проверяя при этом наличие свободной памяти.

Для вычисления адресов определим для каждого объявления два синтезируемых атрибута: DISP будет обозначать смещение внутри области процедуры (или единицы компиляции), а SIZE - размер. Тогда семантика правила для списка объявлений принимает вид RULE

DeclPart ::= ( Decl )

SEMANTICS

Disp<1>=0;

1A: Disp<1>=Disp<1>+Size<1>; Size<0>=Disp<1>.

Все объявления, кроме объявлений переменных, имеют нулевой размер. Размер объявления переменной определяется следующим правилом: RULE

Decl ::= 'VAR' TypeDes

SEMANTICS

Tablentry Entry;

0: Entry=IncTab;

Size<0>=((Table[VAL<2>]+1) / 2)\*2;

// Выравнивание на границу слова Table[Entry]=Disp<0>+Size<0>.

В качестве примера трансляции определения типа рассмотрим обработку описания записи: RULE

TypeDes ::= 'REC' ( TypeDes ) 'END'

SEMANTICS

int Disp;

Tablentry Temp;

0: Entry<0>=IncTab;

Disp=0;

2A: {Temp=IncTab;

Table[Temp]=Disp;

Disp=Disp+Table[Entry<2>]+1) / 2)\*2;

// Выравнивание на границу слова

}

Table[Entry<0>]=Disp.

## 7.4 Трансляция переменных

Переменные отражают все многообразие механизмов доступа в языке. Переменная имеет синтезированный атрибут ADDRESS - это запись, описывающая адрес в команде МС68020. Этот атрибут сопоставляется всем нетерминалам, представляющим значения. В системе команд МС68020 много способов адресации, и они отражены в структуре значения атрибута ADDRESS, имеющего следующий тип: enum Register

{D0,D1,D2,D3,D4,D5,D6,D7,A0,A1,A2,A3,A4,A5,A6,SP,NO};

enum AddrMode

{D,A,Post,Pre,Indirect,IndPre,IndPost,IndirPC,

IndPrePC,IndPostPC,InDisp,Index,IndexPC,Abs,Imm};

struct AddrType{

Register AddrReg,IndexReg; int IndexDisp,AddrDisp;

short Scale; };

Значение регистра NO означает, что соответствующий регистр в адресации не используется.

Доступ к переменным осуществляется в зависимости от их уровня:

глобальные переменные адресуются с помощью абсолютной адресации; переменные в процедуре текущего уровня адресуются через регистр базы А6.

Если стек организован с помощью статической цепочки, то переменные предыдущего статического уровня адресуются через регистр статической цепочки А5; переменные остальных уровней адресуются «пробеганием» по статической цепочке с использованием вспомогательного регистра. Адрес переменной формируется при обработке структуры переменной слева направо и передается сначала сверху вниз как наследуемый атрибут нетерминала VarTail, а затем передается снизу-вверх как глобальный атрибут нетерминала Variable. Таким образом, правило для обращения к переменной имеет вид (первое вхождение Number в правую часть - это уровень переменной, второе - ее Лидер-номер): RULE

Variable ::= VarMode Number Number VarTail

SEMANTICS

int Temp;

struct AddrType AddrTmp1, AddrTmp2; 3: if (Val<2>==0) // Глобальная переменная

{Address<4>.AddrMode=Abs;

Address<4>.AddrDisp=0;

}

else // Локальная переменная {Address<4>.AddrMode=Index;

if (Val<2>==Level) // Переменная текущего уровня

Address<4>.AddrReg=A6; else if (Val<2>==Level-1)

// Переменная предыдущего уровня

Address<4>.AddrReg=A5; else

{Address<4>.Addreg=GetFree(RegSet);

AddrTmp1.AddrMode=Indirect;

AddrTmp1.AddrReg=A5;

Emit2(MOVEA,AddrTmp1,Address<4>.AddrReg);

AddrTmp1.AddrReg=Address<4>.AddrReg;

AddrTmp2.AddrMode=A;

AddrTmp2.AddrReg=Address<4>.AddrReg;

for (Temp=Level-Val<2>;Temp>=2;Temp-)

Emit2(MOVEA,AddrTmp1,AddrTmp2);

}

if (Val<2>==Level)

Address<4>.AddrDisp=Table[Val<3>]; else

Address<4>.AddrDisp=Table[Val<3>]+Table[LevelTab[Val<2>]];

}.

Функция GetFree выбирает очередной свободный регистр (либо регистр данных, либо адресный регистр) и отмечает его как использованный в атрибуте RegSet нетерминала Block. Процедура Emit2 генерирует двухадресную команду. Первый параметр этой процедуры - код команды, второй и третий параметры имеют тип AddrType и служат операндами команды. Смещение переменной текущего уровня отсчитывается от базы (А6), а других уровней - от указателя статической цепочки, поэтому оно определяется как алгебраическая сумма размера локальных параметров и величины смещения переменной. Таблица LevelTab - это таблица уровней процедур, содержащая указатели на последовательно вложенные процедуры.

Если стек организован с помощью дисплея, то трансляция для доступа к переменным может быть осуществлена следующим образом: RULE

Variable ::= VarMode Number Number VarTail

SEMANTICS

int Temp;

3: if (Val<2>==0) // Глобальная переменная

{Address<4>.AddrMode=Abs;

Address<4>.AddrDisp=0;

}

else // Локальная переменная {Address<4>.AddrMode=Index;

if (Val<2>=Level) // Переменная текущего уровня

{Address<4>.AddrReg=A6;

Address<4>.AddrDisp=Table[Val<3>];

} else

{Address<4>.AddrMode=IndPost;

Address<4>.AddrReg=NO;

Address<4>.IndexReg=NO;

Address<4>.AddrDisp=Display[Val<2>];

Address<4>.IndexDisp=Table[Val<3>];

} }.

Рассмотрим трансляцию доступа к полям записи. Она описывается следующим правилом (Number - это Лидер-номер описания поля):

RULE

VarTail ::= 'FIL' Number VarTail SEMANTICS

if (Address<0>.AddrMode==Abs)

{Address<3>.AddrMode=Abs;

Address<3>.AddrDisp=Address<0>.AddrDisp+Table[Val<2>];

}

else

{Address<3>=Address<0>; if (Address<0>.AddrMode==Index)

Address<3>.AddrDisp=Address<0>.AddrDisp+Table[Val<2>]; else

Address<3>.IndexDisp=Address<0>.IndexDisp+Table[Val<2>];

}.

## 7.5 Трансляция целых выражений

Трансляция выражений различных типов управляется синтаксически благодаря наличию указателя типа перед каждой операцией. Мы рассмотрим некоторые наиболее характерные проблемы генерации кода для выражений.

Система команд МС68020 обладает двумя особенностями, сказывающимися на генерации кода для арифметических выражений (то же можно сказать и о генерации кода для выражений типа «множества»):

1. один из операндов выражения (правый) должен при выполнении операции находиться на регистре, поэтому если оба операнда не на регистрах, то перед выполнением операции один из них надо загрузить на регистр;
2. система команд довольно «симметрична», т.е. нет специальных требований к регистрам при выполнении операций (таких, например, как пары регистров или требования четности и т.д.).

Поэтому выбор команд при генерации арифметических выражений определяется довольно простыми таблицами решений. Например, для целочисленного сложения такая таблица приведена на рис. 7.5.1.

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 7.5.1

Здесь имеется в виду, что R - операнд на регистре, V - переменная или константа. Такая таблица решений должна также учитывать коммутативность операций. RULE

IntExpr ::= 'PLUS' IntExpr IntExpr

SEMANTICS

if (Address<2>.AddrMode!=D) && (Address<3>.AddrMode!=D)

{Address<0>.AddrMode=D;

Address<0>.Addreg=GetFree(RegSet);

Emit2(MOVE,Address<2>,Address<0>);

Emit2(ADD,Address<2>,Address<0>);

}

else

if (Address<2>.AddrMode==D)

{Emit2(ADD,Address<3>,Address<2>);

Address<0>:=Address<2>);

}

else {Emit2(ADD,Address<2>,Address<3>); Address<0>:=Address<3>);

}.

### 7.6 Трансляция арифметических выражений

Одной из важнейших задач при генерации кода является распределение регистров. Рассмотрим хорошо известную технику распределения регистров при трансляции арифметических выражений, называемую алгоритмом СетиУльмана. (Замечание: в целях большей наглядности, в данном параграфе мы немного отступаем от семантики арифметических команд MC68020 и предполагаем, что команда Op Arg1, Arg2

выполняет действие Arg2:=Arg1 Op Arg2.)

Пусть система команд машины имеет неограниченное число универсальных регистров, в которых выполняются арифметические команды. Рассмотрим, как можно сгенерировать код, используя для данного арифметического выражения минимальное число регистров.

Изображение выглядит как стрела

Автоматически созданное описание

Рисунок 7.6.1

Пусть имеется синтаксическое дерево выражения. Предположим сначала, что распределение регистров осуществляется по простейшей схеме сверху-вниз слева-направо, как изображено на рис. 7.6.1 Тогда к моменту генерации кода для поддерева LR занято n регистров. Пусть поддерево L требует nl регистров, а поддерево R - nr регистров. Если nl = nr, то при вычислении L будет использовано nl регистров и под результат будет занят (n + 1)-й регистр. Еще nr(= nl) регистров будет использовано при вычислении R. Таким образом, общее число использованных регистров будет равно n + nl +

1.

Если nl > nr, то при вычислении L будет использовано nl регистров. При вычислении R будет использовано nr < nl регистров, и всего будет использовано не более чем n+nl регистров. Если nl < nr, то после вычисления L под результат будет занят один регистр (предположим, (n + 1)-й) и nr регистров будет использовано для вычисления R. Всего будет использовано n+nr+1 регистров.

Видно, что для деревьев, совпадающих с точностью до порядка потомков каждой вершины, минимальное число регистров при распределении их слева-направо достигается на дереве, у которого в каждой вершине слева расположено более «сложное» поддерево, требующее большего числа регистров. Таким образом, если дерево таково, что в каждой внутренней вершине правое поддерево требует меньшего числа регистров, чем левое, то, обходя дерево слева направо, можно оптимально распределить регистры. Без перестройки дерева это означает, что если в некоторой вершине дерева справа расположено более сложное поддерево, то сначала сгенерируем код для него, а затем уже для левого поддерева.

Алгоритм работает следующим образом. Сначала осуществляется разметка синтаксического дерева по следующим правилам.

Правила разметки:

1. если вершина - правый лист или дерево состоит из единственной вершины, помечаем эту вершину числом 1, если вершина - левый лист, помечаем ее 0 (рис. 7.6.2).

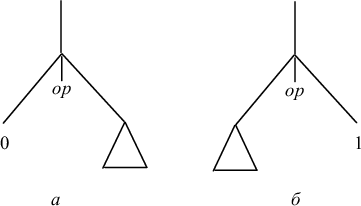


Рисунок 7.6.2

1. если вершина имеет прямых потомков с метками l1 и l2, то в качестве метки этой вершины выбираем наибольшее из чисел l1 или l2 либо число l1 +

1, если l1 = l2 (рис. 7.6.3).

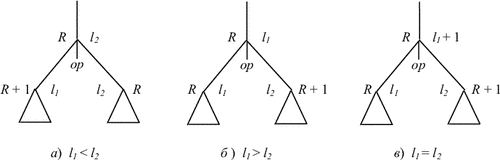


Рисунок 7.6.3

Эта разметка позволяет определить, какое из поддеревьев требует большего количества регистров для своего вычисления. Далее осуществляется распределение регистров для результатов операций по следующим правилам:

1. Корню назначается первый регистр.
2. Если метка левого потомка меньше метки правого, то левому потомку назначается регистр на единицу больший, чем предку, а правому - с тем же номером (сначала вычисляется правое поддерево и его результат помещается в регистр R), так что регистры занимаются последовательно. Если же метка левого потомка больше или равна метке правого потомка, то наоборот, правому потомку назначается регистр на единицу больший, чем предку, а левому - с тем же номером (сначала вычисляется левое поддерево и его результат помещается в регистр R - рис. 7.6.3).

После этого формируется код по следующим правилам:

1. если вершина - правый лист с меткой 1, то ей соответствует код

MOVE X, R где R - регистр, назначенный этой вершине, а X - адрес переменной,

связанной с вершиной (рис. 7.6.4, б);

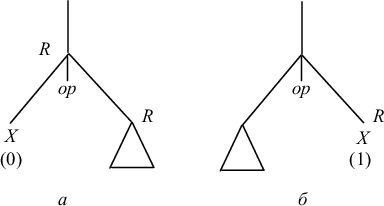


Рисунок 7.6.4

1. если вершина внутренняя и ее левый потомок - лист с меткой 0, то ей соответствует код

Код правого поддерева

Op X, R где R - регистр, назначенный этой вершине, X - адрес переменной,

связанной с вершиной, а Op - операция, примененная в вершине (рис. 9.8, а);

1. если непосредственные потомки вершины не листья и метка правой вершины больше метки левой, то вершине соответствует код

Код правого поддерева

Код левого поддерева

Op R+1, R где R - регистр, назначенный внутренней вершине, и операция Op,

вообще говоря, не коммутативная (рис. 7.6.5, б);

Изображение выглядит как стрела

Автоматически созданное описание

Рисунок 7.6.5

1. если непосредственные потомки вершины не листья и метка правой вершины меньше или равна метке левой вершины, то вершине соответствует код

Код левого поддерева

Код правого поддерева

Op R, R+1

MOVE R+1, R

Последняя команда генерируется для того, чтобы получить результат в нужном регистре (в случае коммутативной операции операнды операции можно поменять местами и избежать дополнительной пересылки - рис. 7.6.5, а).

Рассмотрим атрибутную схему, реализующую эти правила генерации кода (для большей наглядности входная грамматика соответствует обычной инфиксной записи, а не Лидер-представлению). В этой схеме генерация кода происходит не непосредственно в процессе обхода дерева, как раньше, а изза необходимости переставлять поддеревья код строится в виде текста с помощью операции конкатенации. Практически, конечно, это нецелесообразно: разумнее управлять обходом дерева непосредственно, однако для простоты мы будем пользоваться конкатенацией. RULE

Expr ::= IntExpr

SEMANTICS

Reg<1>=1; Code<0>=Code<1>; Left<1>=true.

RULE

IntExpr ::= IntExpr Op IntExpr

SEMANTICS

Left<1>=true; Left<3>=false;

Label<0>=(Label<1>==Label<3>)

? Label<1>+1

: Max(Label<1>,Label<3>);

Reg<1>=(Label<1> < Label<3>)

? Reg<0>+1

: Reg<0>;

Reg<3>=(Label<1> < Label<3>)

? Reg<0>

: Reg<0>+1;

Code<0>=(Label<1>==0)

? Code<3> + Code<2>

+ Code<1> + "," + Reg<0>

: (Label<1> < Label<3>)

? Code<3> + Code<1> + Code<2> +

(Reg<0>+1) + "," + Reg<0>

: Code<1> + Code<3> + Code<2> +

Reg<0> + "," + (Reg<0>+1)

+ "MOVE" + (Reg<0>+1) + "," + Reg<0>.

RULE

IntExpr ::= Ident

SEMANTICS

Label<0>=(Left<0>) ? 0 : 1;

Code<0>=(!Left<0>)

? "MOVE" + Reg<0> + "," + Val<1>

: Val<1>.

RULE

IntExpr ::= '(' IntExpr ')'

SEMANTICS

Label<0>=Label<2>; Reg<2>=Reg<0>; Code<0>=Code<2>.

RULE

Op ::= '+'

SEMANTICS Code<0>="ADD".

RULE

Op ::= '-'

SEMANTICS

Code<0>="SUB".

RULE

Op ::= '\*'

SEMANTICS Code<0>="MUL".

RULE

Op ::= '/'

SEMANTICS Code<0>="DIV".

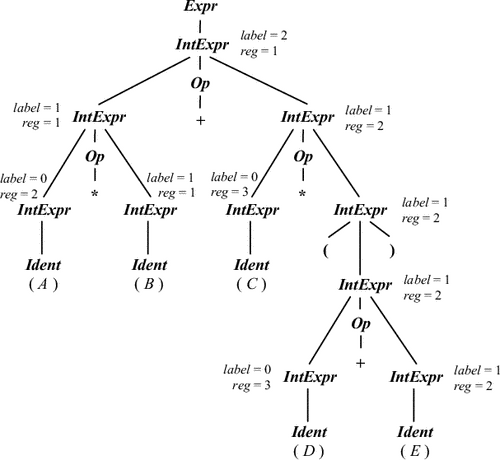


Рисунок 7.6.6

Атрибутированное дерево для выражения A\*B+C\*(D+E) приведено на рис. 7.6.6. При этом будет сгенерирован следующий код: MOVE B, R1

MUL A, R1

MOVE E, R2

ADD D, R2

MUL C, R2

ADD R1, R2

MOVE R2, R1

Приведенная атрибутная схема требует двух проходов по дереву выражения. Рассмотрим теперь другую атрибутную схему, в которой достаточно одного обхода для генерация программы для выражений с оптимальным распределением регистров [7].

Пусть мы произвели разметку дерева разбора так же, как и в предыдущем алгоритме. Назначение регистров будем производить следующим образом.

Левому потомку всегда назначается регистр, равный его метке, а правому - его метке, если она не равна метке его левого брата, и метке + 1, если метки равны. Поскольку более сложное поддерево всегда вычисляется раньше более простого, его регистр результата имеет больший номер, чем любой регистр, используемый при вычислении более простого поддерева, что гарантирует правильность использования регистров.

Приведенные соображения реализуются следующей атрибутной схемой: RULE

Expr ::= IntExpr

SEMANTICS

Code<0>=Code<1>; Left<1>=true.

RULE

IntExpr ::= IntExpr Op IntExpr

SEMANTICS

Left<1>=true; Left<3>=false;

Label<0>=(Label<1>==Label<3>)

? Label<1>+1

: Max(Label<1>,Label<3>);

Code<0>=(Label<3> > Label<1>)

? (Label<1>==0)

? Code<3> + Code<2> + Code<1>

+ "," + Label<3>

: Code<3> + Code<1> + Code<2> +

Label<1> + "," + Label<3>

: (Label<3> < Label<1>)

? Code<1> + Code<3> + Code<2> +

Label<1> + "," + Label<3> +

"MOVE" + Label<3> + "," + Label<1>

: // метки равны

Code<3> + "MOVE" + Label<3> +

"," + Label<3>+1 + Code<1> + Code<2> + Label<1> + "," + Label<1>+1.

RULE

IntExpr ::= Ident

SEMANTICS

Label<0>=(Left<0>) ? 0 : 1;

Code<0>=(Left<0>) ? Val<1> : "MOVE" + Val<1> + "R1".

RULE

IntExpr ::= '(' IntExpr ')'

SEMANTICS Label<0>=Label<2>; Code<0>=Code<2>.

RULE

Op ::= '+'

SEMANTICS Code<0>="ADD".

RULE

Op ::= '-'

SEMANTICS

Code<0>="SUB".

RULE

Op ::= '\*'

SEMANTICS Code<0>="MUL".

RULE

Op ::= '/'

SEMANTICS Code<0>="DIV".

Команды пересылки требуются для согласования номеров регистров, в которых осуществляется выполнение операции, с регистрами, в которых должен быть выдан результат. Это имеет смысл, когда эти регистры разные. Получиться это может из-за того, что по приведенной схеме результат выполнения операции всегда находится в регистре с номером метки, а метки левого и правого поддеревьев могут совпадать.

Для выражения A\*B+C\*(D+E) будет сгенерирован следующий код: MOVE E, R1

ADD D, R1

MUL C, R1

MOVE R1, R2

MOVE B, R1

MUL A, R1

ADD R1, R2

В приведенных атрибутных схемах предполагалось, что регистров достаточно для трансляции любого выражения. Если это не так, приходится усложнять схему трансляции и при необходимости сбрасывать содержимое регистров в память (или магазин).

## 7.7 Трансляция логических выражений

Логические выражения, включающие логическое умножение, логическое сложение и отрицание, можно вычислять как непосредственно, используя таблицы истинности, так и с помощью условных выражений, основанных на следующих простых правилах:

A AND B эквивалентно if A then B else False,

A OR B эквивалентно if A then True else B.

Если в качестве компонент выражений могут входить функции с побочным эффектом, то, вообще говоря, результат вычисления может зависеть от способа вычисления. В некоторых языках программирования не оговаривается, каким способом должны вычисляться логические выражения (например, в Паскале), в некоторых требуется, чтобы вычисления производились тем или иным способом (например, в Модуле-2 требуется, чтобы выражения вычислялись по приведенным формулам), в некоторых языках есть возможность явно задать способ вычисления (Си, Ада). Вычисление логических выражений непосредственно по таблицам истинности аналогично вычислению арифметических выражений, поэтому мы не будем их рассматривать отдельно. Рассмотрим подробнее способ вычисления с помощью приведенных выше формул (будем называть его «вычислением с условными переходами»). Иногда такой способ рассматривают как оптимизацию вычисления логических выражений.

Рассмотрим следующую атрибутную грамматику со входным языком логических выражений: RULE

Expr ::= BoolExpr

SEMANTICS

FalseLab<1>=False; TrueLab<1>=True; Code<0>=Code<1>.

RULE

BoolExpr ::= BoolExpr 'AND' BoolExpr

SEMANTICS

FalseLab<1>=FalseLab<0>; TrueLab<1>=NodeLab<3>; FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>; Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE

BoolExpr ::= BoolExpr 'OR' BoolExpr

SEMANTICS

FalseLab<1>=NodeLab<3>; TrueLab<1>=TrueLab<0>; FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>; Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE

BoolExpr ::= '(' BoolExpr ')'

SEMANTICS

FalseLab<2>=FalseLab<0>;

TrueLab<2>=TrueLab<0>;

Code<0>=NodeLab<0> + ":" + Code<2>.

RULE

BoolExpr ::= F

SEMANTICS

Code<0>=NodeLab<0> + ":" + "GOTO" + FalseLab<0>.

RULE

BoolExpr ::= T

SEMANTICS

Code<0>=NodeLab<0> + ":" + "GOTO" + TrueLab<0>.

Здесь предполагается, что все вершины дерева занумерованы и номер вершины дает атрибут NodeLab. Метки вершин передаются, как это изображено на рис. 7.7.1.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 7.7.1

Таким образом, каждому атрибутированному дереву в этой атрибутной грамматике сопоставляется код, полученный в результате обхода дерева сверху-вниз слева-направо следующим образом. При входе в вершину BoolExpr генерируется ее номер, в вершине F генерируется текст GOTO значение атрибута FalseLab<0>, в вершине T - GOTO значение атрибута

TrueLab<0>. Например, для выражения F OR ( F AND T AND T ) OR T получим атрибутированное дерево, изображенное на рис. 7.7,2 и код 1:7: GOTO 2

2:8:4:9: GOTO 3

5:10: GOTO 6

6: GOTO True

3: GOTO True True: ...

False: ...

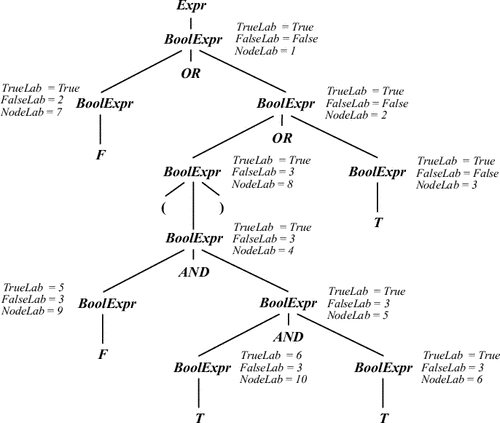


Рисунок 7.7.2

Эту линеаризованную запись можно трактовать как программу вычисления логического значения: каждая строка может быть помечена номером вершины и содержать либо переход на другую строку, либо переход на True или False, что соответствует значению выражения true или false. Будем говорить, что полученная программа вычисляет (или интерпретирует) значение выражения, если в результате ее выполнения (от первой строки) мы придем к строке, содержащей GOTO True или GOTO False.

**Утверждение 7.1**. В результате интерпретации поддерева с некоторыми значениями атрибутов FalseLab и TrueLab в его корне выполняется команда GOTO TrueLab, если значение выражения истинно, и команда

GOTO FalseLab, если значение выражения ложно.

Доказательство. Применим индукцию по высоте дерева. Для деревьев высоты 1, соответствующих правилам BoolExpr ::= F и BoolExpr ::= T,

справедливость утверждения следует из соответствующих атрибутных правил. Пусть дерево имеет высоту n > 1. Зависимость атрибутов для дизъюнкции и конъюнкции приведена на рис. 7.7.3.

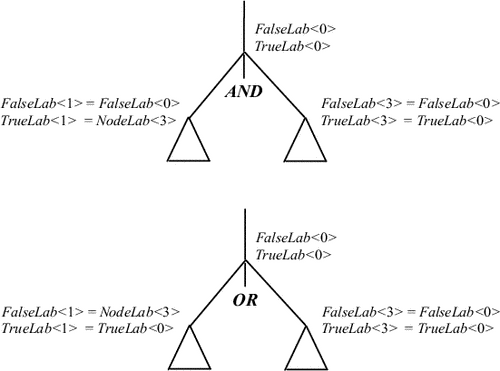


Рисунок 7.7.3

Если для конъюнкции значение левого поддерева ложно и по индукции вычисление левого поддерева завершается командой GOTO FalseLab<1>, то получаем, что вычисление всего дерева завершается командой перехода GOTO FalseLab<0> (= FalseLab<1>). Если же значение левого поддерева истинно, то его вычисление завершается командой перехода

GOTO TrueLab<1> (= NodeLab<3>). Если значение правого поддерева ложно, то вычисление всего дерева завершается командой GOTO FalseLab<0> (= FalseLab<3>). Если же оно истинно, вычисление всего дерева завершается командой перехода GOTO TrueLab<0> (= TrueLab<3>). Аналогично - для дизъюнкции.

**Утверждение 7.2.** Для любого логического выражения, состоящего из констант, программа, полученная в результате обхода дерева этого выражения, завершается со значением логического выражения в обычной интерпретации, т.е. осуществляется переход на True для значения, равного true, и переход на метку False для значения false.

Доказательство. Это утверждение является частным случаем предыдущего. Его справедливость следует из того, что метки корня дерева

равны соответственно TrueLab = True и FalseLab = False. \_

Добавим теперь новое правило в предыдущую грамматику: RULE

BoolExpr ::= Ident

SEMANTICS

Code<0>=NodeLab<0> + ":" + "if (" + Val<1> + "==T) GOTO" + TrueLab<0> + "else GOTO" + FalseLab<0>.

Тогда, например, для выражения A OR ( B AND C AND D ) OR E получим следующую программу:

1:7: if (A==T) GOTO True else GOTO 2

2:8:4:9: if (B==T) GOTO 5 else GOTO 3

5:10: if (C==T) GOTO 6 else GOTO 3

6: if (D==T) GOTO True else GOTO 3 3: if (E==T) GOTO True else GOTO False

True: ... False: ...

При каждом конкретном наборе данных эта программа превращается в программу вычисления логического значения.

**Утверждение 7.3.** В каждой строке программы, сформированной предыдущей атрибутной схемой, одна из меток внутри условного оператора совпадает с меткой следующей строки.

Доказательство. Действительно, по правилам наследования атрибутов TrueLab и FalseLab, в правилах для дизъюнкции и конъюнкции либо атрибут FalseLab, либо атрибут TrueLab принимает значение метки следующего поддерева. Кроме того, как значение FalseLab, так и значение TrueLab, передаются в правое поддерево от предка. Таким образом, самый правый потомок всегда имеет одну из меток TrueLab или FalseLab, равную метке правого брата соответствующего поддерева. Учитывая порядок генерации команд, получаем справедливость утверждения.\_

Дополним теперь атрибутную грамматику следующим образом: RULE

Expr ::= BoolExpr

SEMANTICS

FalseLab<1>=False; TrueLab<1>=True; Sign<1>=false;

Code<0>=Code<1>.

RULE

BoolExpr ::= BoolExpr 'AND' BoolExpr

SEMANTICS

FalseLab<1>=FalseLab<0>; TrueLab<1>=NodeLab<3>;

FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;

Sign<1>=false; Sign<3>=Sign<0>;

Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE

BoolExpr ::= BoolExpr 'OR' BoolExpr

SEMANTICS

FalseLab<1>=NodeLab<3>; TrueLab<1>=TrueLab<0>;

FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;

Sign<1>=true; Sign<3>=Sign<0>;

Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE

BoolExpr ::= 'NOT' BoolExpr

SEMANTICS

FalseLab<2>=TrueLab<0>; TrueLab<2>=FalseLab<0>; Sign<2>=! Sign<0>; Code<0>=Code<2>.

RULE

BoolExpr ::= '(' BoolExpr ')'

SEMANTICS

FalseLab<2>=FalseLab<0>;

TrueLab<2>=TrueLab<0>;

Sign<2>=Sign<0>;

Code<0>=NodeLab<0> + ":" + Code<2>.

RULE

BoolExpr ::= F

SEMANTICS

Code<0>=NodeLab<0> + ":" + "GOTO" + FalseLab<0>.

RULE

BoolExpr ::= T

SEMANTICS

Code<0>=NodeLab<0> + ":" + "GOTO" + TrueLab<0>.

RULE

BoolExpr ::= Ident

SEMANTICS

Code<0>=NodeLab<0> + ":" + "if (" + Val<1> + "==T) GOTO" + TrueLab<0> + "else GOTO" + FalseLab<0>.

Правила наследования атрибута Sign приведены на рис. 7.7.4.

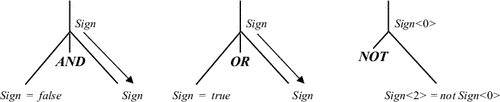


Рисунок 7.7.4

Программу желательно сформировать таким образом, чтобы else-метка была как раз меткой следующей вершины. Как это можно сделать, следует из следующего утверждения.

**Утверждение 7.4.** В каждой терминальной вершине, метка ближайшего правого для нее поддерева равна значению атрибута FalseLab этой вершины, тогда и только тогда, когда значение атрибута Sign этой вершины равно true, и наоборот, метка ближайшего правого для нее поддерева равна значению атрибута TrueLab этой вершины, тогда и только тогда, когда значение атрибута Sign равно false.

Доказательство. Действительно, если ближайшей общей вершиной является AND, то в левого потомка передается NodeLab правого потомка в качестве TrueLab и одновременно Sign правого потомка равен true. Если же ближайшей общей вершиной является OR, то в левого потомка передается NodeLab правого потомка в качестве FalseLab и одновременно Sign правого потомка равен false. Во все же правые потомки значения TrueLab, FalseLab и Sign передаются из предка (за исключением правила для NOT, в котором TrueLab и FalseLab меняются местами, но одновременно на противоположный меняется и Sign). \_

Эти два утверждения (3 и 4) позволяют заменить последнее правило атрибутной грамматики следующим образом: RULE

BoolExpr ::= Ident

SEMANTICS

Code<0>=NodeLab<0> + ":" +

(Sign<0>

? "if (" + Val<1> + "==T) GOTO" + TrueLab<0> : "if (" + Val<1> + "==F) GOTO" + FalseLab<0>).

В свою очередь, при генерации машинных команд это правило можно заменить на следующее: RULE

BoolExpr ::= Ident

SEMANTICS

Code<0>=NodeLab<0> + ":" + "TST" + Val<1> +

(Sign<0>

? "BNE" + TrueLab<0> : "BEQ" + FalseLab<0>).

Таким образом, для выражения A OR ( B AND C AND D ) OR E получим следующий код на командах перехода:

1:7: TST A

BNE True

2:8:4:9: TST B

BEQ 3

5:10: TST C

BEQ 3

6: TST D

BNE True

3: TST E BEQ False True: ... False: ...

Если элементом логического выражения является сравнение, то генерируется команда, соответствующая знаку сравнения (BEQ для =, BNE для <>, BGE для >= и т.д.), если атрибут Sign соответствующей вершины имеет значение true, и отрицание (BNE для =, BEQ для <>, BLT для >= и т.д.), если атрибут Sign имеет значение false.

## 7.8 Выделение общих подвыражений

Выделение общих подвыражений относится к области оптимизации программ. В общем случае трудно (или даже невозможно) провести границу между оптимизацией и «качественной трансляцией». Оптимизация - это и есть качественная трансляция. Обычно термин «оптимизация» употребляют, когда для повышения качества программы используют ее глубокие преобразования такие, например, как перевод в графовую форму для изучения нетривиальных свойств программы.

В этом смысле выделение общих подвыражений - одна из простейших оптимизаций. Для ее осуществления требуется некоторое преобразование программы, а именно построение ориентированного ациклического графа, о котором говорилось в главе, посвященной промежуточным представлениям.

Линейный участок - это последовательность операторов, в которую управление входит в начале и выходит в конце без остановки и перехода изнутри.

Рассмотрим дерево линейного участка, в котором вершинами служат операции, а потомками - операнды. Будем говорить, что две вершины образуют общее подвыражение, если поддеревья для них совпадают, т.е. имеют одинаковую структуру и, соответственно, одинаковые операции во внутренних вершинах и одинаковые операнды в листьях. Выделение общих подвыражений позволяет генерировать для них код один раз, хотя может привести и к некоторым трудностям, о чем вкратце будет сказано ниже.

Выделение общих подвыражений проводится на линейном участке и основывается на двух положениях.

1. Поскольку на линейном участке переменной может быть несколько присваиваний, то при выделении общих подвыражений необходимо различать вхождения переменных до и после присваивания. Для этого каждая переменная снабжается счетчиком. Вначале счетчики всех переменных устанавливаются равными 0. При каждом присваивании переменной ее счетчик увеличивается на 1.
2. Выделение общих подвыражений осуществляется при обходе дерева выражения снизу вверх слева направо. При достижении очередной вершины (пусть операция, примененная в этой вершине, есть бинарная op; в случае унарной операции рассуждения те же) просматриваем общие подвыражения, связанные с op. Если имеется выражение, связанное с op и такое, что его левый операнд есть общее подвыражение с левым операндом нового выражения, а правый операнд - общее подвыражение с правым операндом нового выражения, то объявляем новое выражение общим с найденным и в новом выражении запоминаем указатель на найденное общее выражение. Базисом построения служит переменная: если операндами обоих выражений являются одинаковые переменные с одинаковыми счетчиками, то они являются общими подвыражениями. Если выражение не выделено как общее, оно заносится в список операций, связанных с op.

Рассмотрим теперь реализацию алгоритма выделения общих подвыражений. Поддерживаются следующие глобальные переменные:

Table - таблица переменных; для каждой переменной хранится ее счетчик (Count) и указатель на вершину дерева выражений, в которой переменная встретилась в последний раз в правой части (Last);

OpTable - таблица списков (типа LisType) общих подвыражений, связанных с каждой операцией. Каждый элемент списка хранит указатель на вершину дерева (поле Addr) и продолжение списка (поле List).

С каждой вершиной дерева выражения связана запись типа NodeType, со следующими полями:

Left - левый потомок вершины,

Right - правый потомок вершины,

Comm - указатель на предыдущее общее подвыражение,

Flag - признак, является ли поддерево общим подвыражением, Varbl - признак, является ли вершина переменной, VarCount - счетчик переменной.

Выделение общих подвыражений и построение дерева осуществляются приведенными ниже правилами. Атрибут Entry нетерминала Variable дает указатель на переменную в таблице Table. Атрибут Val символа Op дает код операции. Атрибут Node символов IntExpr и Assignment дает указатель на запись типа NodeType соответствующего нетерминала. RULE

Assignment ::= Variable IntExpr SEMANTICS

Table[Entry<1>].Count=Table[Entry<1>].Count+1.

// Увеличить счетчик присваиваний переменной

RULE

IntExpr ::= Variable

SEMANTICS

if ((Table[Entry<1>].Last!=NULL)

// Переменная уже была использована

&& (Table[Entry<1>].Last->VarCount

== Table[Entry<1>].Count ))

// С тех пор переменной не было присваивания

{Node<0>->Flag=true;

// Переменная - общее подвыражение

Node<0>->Comm= Table[Entry<1>].Last;

// Указатель на общее подвыражение

}

else Node<0>->Flag=false;

Table[Entry<1>].Last=Node<0>;

// Указатель на последнее использование переменной

Node<0>->VarCount= Table[Entry<1>].Count;

// Номер использования переменной Node<0>->Varbl=true.

// Выражение - переменная

RULE

IntExpr ::= Op IntExpr IntExpr

SEMANTICS

LisType \* L; //Тип списков операции if ((Node<2>->Flag) && (Node<3>->Flag)) // И справа, и слева - общие подвыражения

{L=OpTable[Val<1>];

// Начало списка общих подвыражений для операции while (L!=NULL) if ((Node<2>==L->Left) && (Node<3>==L->Right))

// Левое и правое поддеревья совпадают break;

else L=L->List;// Следующий элемент списка

}

else L=NULL; //Не общее подвыражение

Node<0>->Varbl=false; // Не переменная

Node<0>->Comm=L;

//Указатель на предыдущее общее подвыражение или NULL

if (L!=NULL)

{Node<0>->Flag=true; //Общее подвыражение

Node<0>->Left=Node<2>; // Указатель на левое поддерево

Node<0>->Right=Node<3>;

// Указатель на правое поддерево

}

else {Node<0>->Flag=false;

// Данное выражение не может рассматриваться как общее

// Если общего подвыражения с данным не было,

// включить данное в список для операции

L=alloc(sizeof(struct LisType));

L->Addr=Node<0>;

L->List=OpTable[Val<1>];

OpTable[Val<1>]=L;

}.

Рассмотрим теперь некоторые простые правила распределения регистров при наличии общих подвыражений. Если число регистров ограничено, можно выбрать один из следующих двух вариантов.

1. При обнаружении общего подвыражения с подвыражением в уже просмотренной части дерева (и, значит, с уже распределенными регистрами) проверяем, расположено ли его значение на регистре. Если да, и если регистр после этого не менялся, заменяем вычисление поддерева на значение в регистре. Если регистр менялся, то вычисляем подвыражение заново.
2. Вводим еще один проход. На первом проходе распределяем регистры. Если в некоторой вершине обнаруживается, что ее поддерево общее с уже вычисленным ранее, но значение регистра потеряно, то в такой вершине на втором проходе необходимо сгенерировать команду сброса регистра в рабочую память. Выигрыш в коде будет, если стоимость команды сброса регистра + доступ к памяти в повторном использовании этой памяти не превосходит стоимости заменяемого поддерева. Поскольку стоимость команды MOVE известна, можно сравнить стоимости и принять оптимальное решение: пометить предыдущую вершину для сброса либо вычислять поддерево полностью.

## 7.9 Генерация оптимального кода методами синтаксического анализа

### 7.9.1 Сопоставление образцов

Техника генерации кода, рассмотренная выше, основывалась на однозначном соответствии структуры промежуточного представления и описывающей это представление грамматики. Недостатком такого «жесткого» подхода является то, что как правило одну и ту же программу на промежуточном языке можно реализовать многими различными способами в системе команд машины. Эти разные реализации могут иметь различную длину, время выполнения и другие характеристики. Для генерации более качественного кода может быть применен подход, изложенный в настоящей главе.

Этот подход основан на понятии «сопоставления образцов»: командам машины сопоставляются некоторые «образцы», вхождения которых ищутся в промежуточном представлении программы, и делается попытка «покрыть» промежуточную программу такими образцами. Если это удается, то по образцам восстанавливается программа уже в кодах. Каждое такое покрытие соответствует некоторой программе, реализующей одно и то же промежуточное представление.

Изображение выглядит как стрела

Автоматически созданное описание

Рисунок 7.9.1

На рис. 7.9.1 показано промежуточное дерево для оператора a = b[i] + 5, где a, b, i - локальные переменные, хранимые со смещениями x, y, z соответственно в областях данных с одноименными адресами.

Элемент массива b занимает память в одну машинную единицу. 0местная операция const возвращает значение атрибута соответствующей вершины промежуточного дерева, указанного на рисунке в скобках после оператора. Одноместная операция @ означает косвенную адресацию и возвращает содержимое регистра или ячейки памяти, имеющей адрес, задаваемый аргументом операции.

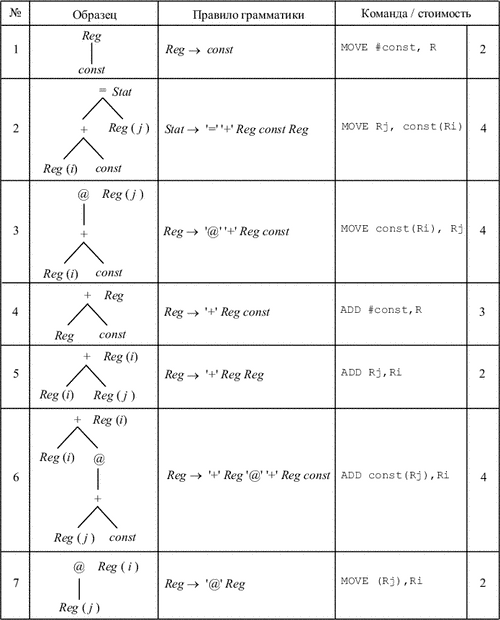


Рисунок 7.9.2

На рис. 7.9.2 показан пример сопоставления образцов машинным командам. Приведены два варианта задания образца: в виде дерева и в виде правила контекстно-свободной грамматики. Для каждого образца указана машинная команда, реализующая этот образец, и стоимость этой команды.

В каждом дереве-образце корень или лист может быть помечен терминальным и/или нетерминальным символом. Внутренние вершины помечены терминальными символами - знаками операций. При наложении образца на дерево выражения, во-первых, терминальный символ образца должен соответствовать терминальному символу дерева, и, во-вторых, образцы должны «склеиваться» по типу нетерминального символа, т.е. тип корня образца должен совпадать с типом вершины, в которую образец подставляется корнем. Допускается использование «цепных» образцов, т.е. образцов, корню которых не соответствует терминальный символ, и имеющих единственный элемент в правой части. Цепные правила служат для приведения вершин к одному типу. Например, в рассматриваемой системе команд одни и те же регистры используются как для целей адресации, так и для вычислений. Если бы в системе команд для этих целей использовались разные группы регистров, то в грамматике команд могли бы использоваться разные нетерминалы, а для пересылки из адресного регистра в регистр данных могла бы использоваться соответствующая команда и образец.

Нетерминалы Reg на образцах могут быть помечены индексом (i или j), что (неформально) соответствует номеру регистра и служит лишь для пояснения смысла использования регистров. Отметим, что при генерации кода рассматриваемым методом не осуществляется распределение регистров. Это является отдельной задачей.

Стоимость может определяться различными способами, например числом обращений к памяти при выборке и исполнении команды. Здесь мы не рассматриваем этого вопроса. На рис. 7.9.3 приведен пример покрытия промежуточного дерева рис. 7.9.2 образцами рис. 7.9.2. В рамки заключены фрагменты дерева, сопоставленные образцу правила, номер которого указывается в левом верхнем

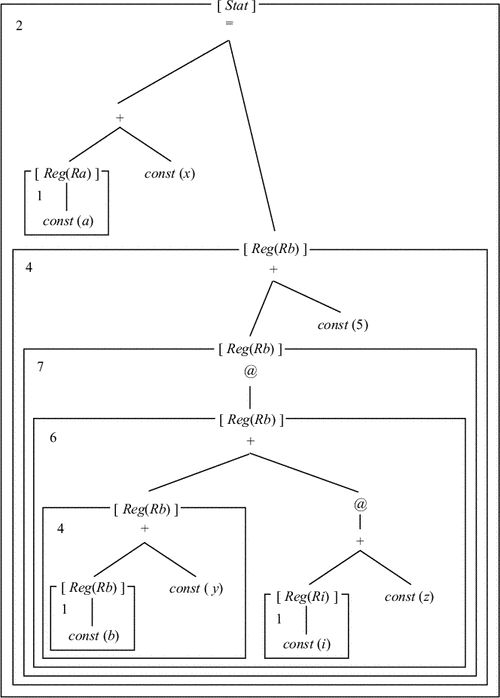


Рисунок 7.9.3

Приведенное покрытие дает такую последовательность команд: MOVE b,Rb

ADD #y,Rb

MOVE i,Ri

ADD z(Ri),Rb

MOVE (Rb),Rb

ADD #5,Rb

MOVE a,Ra

MOVE Rb,x(Ra)

Основная идея подхода заключается в том, что каждая команда машины описывается в виде такого образца. Различные покрытия дерева промежуточного представления соответствуют различным последовательностям машинных команд. Задача выбора команд состоит в том, чтобы выбрать наилучший способ реализации того или иного действия или последовательности действий, т. е. выбрать в некотором смысле оптимальное покрытие.

Для выбора оптимального покрытия было предложено несколько интересных алгоритмов, в частности использующих динамическое программирование. Мы здесь рассмотрим алгоритм, комбинирующий возможности синтаксического анализа и динамического программирования. В основу этого алгоритма положен синтаксический анализ неоднозначных грамматик (модифицированный алгоритм Кока, Янгера и Касами), эффективный в реальных приложениях. Этот же метод может быть применен и тогда, когда в качестве промежуточного представления используется дерево.

### 7.9.2 Синтаксический анализ для T-грамматик

Обычно код генерируется из некоторого промежуточного языка с довольно жесткой структурой. В частности, для каждой операции известна ее размерность, то есть число операндов, большее или равное 0. Операции задаются терминальными символами, и наоборот - будем считать все терминальные символы знаками операций. Назовем грамматики, удовлетворяющие этим ограничениям, T-грамматиками. Правая часть каждой продукции в Т-грамматике есть правильное префиксное выражение, которое может быть задано следующим определением:

* Операция размерности 0 является правильным префиксным выражением;
* Нетерминал является правильным префиксным выражением;
* Префиксное выражение, начинающееся со знака операции размерности n > 0, является правильным, если после знака операции следует n правильных префиксных выражений;
* Ничто другое не является правильным префиксным выражением.

Образцы, соответствующие машинным командам, задаются правилами грамматики (вообще говоря, неоднозначной). Генератор кода анализирует входное префиксное выражение и строит одновременно все возможные деревья разбора. После окончания разбора выбирается дерево с наименьшей стоимостью. Затем по этому единственному оптимальному дереву генерируется код.

Для T-грамматик все цепочки, выводимые из любого нетерминала A, являются префиксными выражениями с фиксированной арностью операций. Длины всех выражений из входной цепочки a1...an можно предварительно вычислить (под длиной выражения имеется ввиду длина подстроки, начинающейся с символа кода операции и заканчивающейся последним символом, входящим в выражение для этой операции). Поэтому можно проверить, сопоставимо ли некоторое правило с подцепочкой ai...ak входной цепочки a1...an, проходя слева-направо по ai...ak. В процессе прохода по цепочке предварительно вычисленные длины префиксных выражений используются для того, чтобы перейти от одного терминала к следующему терминалу, пропуская подцепочки, соответствующие нетерминалам правой части правила.

Цепные правила не зависят от операций, следовательно, их необходимо проверять отдельно. Применение одного цепного правила может зависеть от применения другого цепного правила. Следовательно, применение цепных правил необходимо проверять до тех пор, пока нельзя применить ни одно из цепных правил. Мы предполагаем, что в грамматике нет циклов в применении цепных правил. Построение всех вариантов анализа для Tграмматики дано ниже в алгоритме 9.1. Тип Titem в алгоритме 9.1 ниже служит для описания ситуаций (т.е. правил вывода и позиции внутри правила). Тип Tterminal - это тип терминального символа грамматики, тип Tproduction - тип для правила вывода.

### Алгоритм 7.1

Tterminal a[n]; setofTproduction r[n];

int l[n]; // l[i] - длина a[i]-выражения Titem h; // используется при поиске правил,

// сопоставимых с текущей подцепочкой

// Предварительные вычисления

Для каждой позиции i вычислить длину a[i]-выражения l[i];

// Распознавание входной цепочки for (int i=n-1;i>=0;i-){

for (каждого правила A -> a[i] y из P){ j=i+1; if (l[i]>1)

// Первый терминал a[i] уже успешно сопоставлен

{h=[A->a[i].y];

do // найти a[i]y, сопоставимое с a[i]..a[i+l[i]-1]

{Пусть h==[A->u.Xv]; if (X in T)

if (X==a[j]) j=j+1; else break;

else // X in N

if (имеется X->w in r[j]) j=j+l[j]; else break; h=[A->uX.v];

//перейти к следующему символу

}

while (j!=i+l[i]);

} // l[i]>1

if (j==i+l[i]) r[i]=r[i] + { (A->a[i]y) }

} // for по правилам A -> a[i] y

// Сопоставить цепные правила while (существует правило C->A из P такое, что имеется некоторый элемент (A->w) в r[i] и нет элемента (C->A) в r[i]) r[i]=r[i] + { (C->A) };

} // for по i

Проверить, принадлежит ли (S->w) множеству r[0];

Множества r[i] имеют размер O(|P|). Можно показать, что алгоритм имеет временную и емкостную сложность O(n).

Рассмотрим вновь пример рис. 7.9.4. В префиксной записи приведенный фрагмент программы записывается следующим образом:

= + a x + @ + + b y @ + i z 5

На рис. 9.18 приведен результат работы алгоритма. Правила вычисления стоимости приведены в следующем разделе. Все возможные выводы входной цепочки (включая оптимальный) можно построить, используя таблицу l длин префиксных выражений и таблицу r применимых правил.

Изображение выглядит как текст, квитанция, снимок экрана

Автоматически созданное описание

Рисунок 7.9.4

Пусть G - это T-грамматика. Для каждой цепочки z из L(G) можно построить абстрактное синтаксическое дерево соответствующего выражения (рис. 9.15). Мы можем переписать алгоритм так, чтобы он принимал на входе абстрактное синтаксическое дерево выражения, а не цепочку. Этот вариант алгоритма приведен ниже. В этом алгоритме дерево выражения обходится сверху вниз и в нем ищутся поддеревья, сопоставимые с правыми частями правил из G. Обход дерева осуществляется процедурой PARSE. После обхода поддерева данной вершины в ней применяется процедура MATCHED, которая пытается найти все образцы, сопоставимые поддереву данной вершины. Для этого каждое правило-образец разбивается на компоненты в соответствии с встречающимися в нем операциями. Дерево обходится справа налево только для того, чтобы иметь соответствие с порядком вычисления в алгоритме 9.1. Очевидно, что можно обходить дерево вывода и слева направо.

Структура данных, представляющая вершину дерева, имеет следующую форму:

struct Tnode {

Tterminal op;

Tnode \* son[MaxArity];

setofTproduction RULEs;

};

В комментариях указаны соответствующие фрагменты алгоритма 9.1.

### Алгоритм 7.2

Tnode \* root;

bool MATCHED(Tnode \* n, Titem h)

{ bool matching;

пусть h==[A->u.Xvy], v== v\_1 v\_2 ... v\_m, m=Arity(X); if (X in T)// сопоставление правила

if (m==0) // if l[i]==1 if (X==n->op) //if X==a[j]

return(true); else return(false);

else // if l[i]>1

if (X==n->op) //if X==a[j]

{matching=true;

for (i=1;i<=m;i++) //j=j+l[j] matching=matching && //while (j==i+l[i]) MATCHED(n->son[i-1],[A->uXv'.v\_i v"y]); return(matching); //h=[A->uX.v]

} else return(false);

else // X in N поиск подвывода

if (в n^.RULEs имеется правило с левой частью X) return(true); else

return(false);

}

void PARSE(Tnode \* n)

{

for (i=Arity(n->op);i>=1;i-)

// for (i=n; i>=1;i-)

PARSE(n->son[i-1]); n->RULEs=EMPTY;

for (каждого правила A->bu из P такого, что b==n->op)

if (MATCHED(n,[A->.bu])) //if (j==i+l[i]) n->RULEs=n->RULEs+{(A->bu)};

// Сопоставление цепных правил

while (существует правило C->A из P такое, что некоторый элемент (A->w) в n->RULEs

и нет элемента (C->A) в n->RULEs) n->RULEs=n->RULEs+{(C->A)};

}

Основная программа

// Предварительные вычисления

Построить дерево выражения для входной цепочки z;

root = указатель дерева выражения;

// Распознать входную цепочку

PARSE(root);

Проверить, входит ли во множество root->RULEs правило с левой частью S;

Выходом алгоритма является дерево выражения для z, вершинам которого сопоставлены применимые правила. С помощью такого дерева можно построить все выводы для исходного префиксного выражения.

#### 7.9.3 Выбор дерева вывода наименьшей стоимости

T-грамматики, описывающие системы команд, обычно являются неоднозначными. Чтобы сгенерировать код для некоторой входной цепочки, необходимо выбрать одно из возможных деревьев вывода. Это дерево должно представлять желаемое качество кода, например размер кода и/или время выполнения.

Для выбора дерева из множества всех построенных деревьев вывода можно использовать атрибуты стоимости, атрибутные формулы, вычисляющие их значения, и критерии стоимости, которые оставляют для каждого нетерминала единственное применимое правило. Атрибуты стоимости сопоставляются всем нетерминалам, атрибутные формулы - всем правилам T-грамматики.

Предположим, что для вершины n обнаружено применимое правило



где zi  T\* для 0 i k и Xj  N для 1 j k. Вершина n имеет



потомков n1, ..., nk, которые соответствуют нетерминалам X1, ..., Xk. Значения атрибутов стоимости вычисляются обходя дерево снизу вверх. Вначале атрибуты стоимости инициализируются неопределенным значением UndefinedValue. Предположим, что значения атрибутов стоимости для всех потомков n1, ..., nk вершины n вычислены. Если правилу p сопоставлена формула



то производится вычисление значения атрибута a нетерминала A в вершине n. Для всех примененных правил ищется такое, которое дает минимальное значение стоимости. Отсутствие примененных правил обозначается через Undefined, значение которого полагается большим любого определенного значения.Добавим в алгоритм 9.2 реализацию атрибутов стоимости, формул их вычисления и критериев отбора. Из алгоритма можно исключить поиск подвыводов, соответствующих правилам, для которых значение атрибута стоимости не определено. Структура данных, представляющая вершину дерева, принимает следующий вид:

struct Tnode {

Tterminal op;

Tnode \* son[MaxArity]; struct \* { unsigned CostAttr; Tproduction Production; } nonterm [Tnonterminal]; OperatorAttributes ...

Тело процедуры PARSE принимает вид void PARSE(Tnode \*n)

{for (i=Arity(n->op);i>=1;i-) PARSE(n->son[i]); for (каждого A из N)

{n->nonterm[A].CostAttr=UndefinedValue;

n->nonterm[A].production=Undefined;

}

for (каждого правила A->bu из P такого, что b==n->op) if (MATCHED(n,[A->.bu]))

{ВычислитьАтрибутыСтоимостиДля(A,n,(A->bu)); ПроверитьКритерийДля(A,n->nonterm[A].CostAttr); if ((A->bu) лучше,

чем ранее обработанное правило для A) {Модифицировать(n->nonterm[A].CostAttr); n->nonterm[A].production=(A->bu);

} }

// Сопоставить цепные правила

while (существует правило C->A из P, которое лучше, чем ранее обработанное правило для A)

{ВычислитьАтрибутыСтоимостиДля(C,n,(C->A)); ПроверитьКритерийДля(C,n->nonterm[C].CostAttr); if ((C->A) лучше)

{Модифицировать(n->nonterm[C].CostAttr); n->nonterm[C].production=(C->A);

}

}

}

Дерево наименьшей стоимости определяется как дерево, соответствующее минимальной стоимости корня. Когда выбрано дерево вывода наименьшей стоимости, вычисляются значения атрибутов, сопоставленных вершинам дерева вывода, и генерируются соответствующие машинные команды. Вычисление значений атрибутов, генерация кода осуществляются в процессе обхода выбранного дерева вывода сверху вниз, слева направо. Обход выбранного дерева вывода выполняется процедурой вычислителя атрибутов, на вход которой поступают корень дерева выражения и аксиома грамматики. Процедура использует правило A  z0X1z1...Xkzk, связанное с указанной вершиной n, и заданный нетерминал A, чтобы определить соответствующие им вершины n1, ..., nk и нетерминалы X1, ..., Xk. Затем вычислитель рекурсивно обходит каждую вершину ni, имея на входе нетерминал Xi.

#### 7.9.4 Атрибутная схема для алгоритма сопоставления образцов

Алгоритмы 9.1 и 9.2 являются «универсальными» в том смысле, что конкретные грамматики выражений и образцов являются, по-существу, параметрами этих алгоритмов. В то же время, для каждой конкретной грамматики можно написать свой алгоритм поиска образцов. Например, в случае нашей грамматики выражений и приведенных на рис. 9.16 образцов алгоритм 9.2 может быть представлен атрибутной грамматикой, приведенной ниже.

Наследуемый атрибут Match содержит упорядоченный список (вектор) образцов для сопоставления в поддереве данной вершины. Каждый из образцов имеет вид либо <op op-list> (op - операция в данной вершине, а oplist - список ее операндов), либо представляет собой нетерминал N. В первом случае op-list «распределяется» по потомкам вершины для дальнейшего сопоставления. Во втором случае сопоставление считается успешным, если есть правило N op {Pati}, где w состоит из образцов, успешно сопоставленных потомкам данной вершины. В этом случае по потомкам в качестве образцов распределяются элементы правой части правила. Эти два множества образцов могут пересекаться. Синтезируемый атрибут Pattern - вектор логических значений, дает результат сопоставления по векторуобразцу Match.

Таким образом, при сопоставлении образцов могут встретиться два случая:

* Вектор образцов содержит образец <op {Pati}>, где op - операция, примененная в данной вершине. Тогда распределяем образцы Pati по потомкам и сопоставление по данному образцу считаем успешным (истинным), если успешны сопоставления элементов этого образца по всем потомкам.
* Образцом является нетерминал N. Тогда рассматриваем все правила вида N op {Pati}. Вновь распределяем образцы Pati по потомкам и сопоставление считаем успешным (истинным), если успешны сопоставления по всем потомкам. В общем случае успешным может быть сопоставление по нескольким образцам.



Отметим, что в общем случае в потомки одновременно передается несколько образцов для сопоставления.

В приведенной ниже атрибутной схеме не рассматриваются правила выбора покрытия наименьшей стоимости (см. предыдущий раздел). Выбор оптимального покрытия может быть сделан еще одним проходом по дереву, аналогично тому, как это было сделано выше. Например, в правиле с '+' имеется несколько образцов для Reg, но реального выбора одного из них не осуществляется. Кроме того, не уточнены некоторые детали реализации. В частности, конкретный способ формирования векторов Match и Pattern. В тексте употребляется термин «добавить», что означает добавление к вектору образцов очередного элемента. Векторы образцов записаны в угловых скобках.

RULE

Stat ::= '=' Reg Reg

SEMANTICS

Match<2>=<'+' Reg Const>;

Match<3>=;

Pattern<0>[1]=Pattern<2>[1]&Pattern<3>[1].

Этому правилу соответствует один образец 2. Поэтому в качестве образцов потомков через их атрибуты Match передаются, соответственно, <'+' Reg Const> и .

RULE

Reg ::= '+' Reg Reg

SEMANTICS

if (Match<0> содержит Reg в позиции i)

{Match<2>=;

Match<3>=<const,reg,<'@' '+' reg const>>;

}

if (Match<0> содержит образец <'+' Reg Const> в позиции j)

{добавить Reg к Match<2> в некоторой позиции k;

добавить Const к Match<3> в некоторой позиции k;

}

if (Match<0> содержит образец <'+' Reg Const> в позиции j)

Pattern<0>[j]=Pattern<2>[k]&Pattern<3>[k]; if (Match[0] содержит Reg в i-й позиции) Pattern<0>[i]=(Pattern<2>[1]&Pattern<3>[1])

|(Pattern<2>[2]&Pattern<3>[2])

|(Pattern<2>[3]&Pattern<3>[3]). </const,reg,<'@' '+' reg const>

Образцы, соответствующие этому правилу, следующие:

1. Reg  '+' Reg Const,
2. Reg  '+' Reg Reg,
3. Reg  '+' Reg '@' '+' Reg Const.

Атрибутам Match второго и третьего символов в качестве образцов при сопоставлении могут быть переданы векторы <reg,< span=""> Reg, Reg> и <const,< span=""> Reg, <'@' '+' Reg Const>>, соответственно. Из анализа других правил можно заключить, что при сопоставлении образцов предков левой части данного правила атрибуту Match символа левой части может быть передан образец <'+' Reg Const> (из образцов 2, 3, 6) или образец Reg. </const,<></reg,<>

RULE

Reg ::= '@' Reg

SEMANTICS

if (Match<0> содержит Reg в i-й позиции) Match<2>=<<'+' Reg Const>,Reg>; if (Match<0> содержит <'@' '+' Reg Const> в j-й позиции) добавить к Match<2> <'+' Reg Const> в k позиции; if (Match<0> содержит Reg в i-й позиции) Pattern<0>[i]=Pattern<2>[1]|Pattern<2>[2]; if (Match<0> содержит <'@' '+' Reg Const> в j-й позиции) Pattern<0>[j]=Pattern<2>[k].

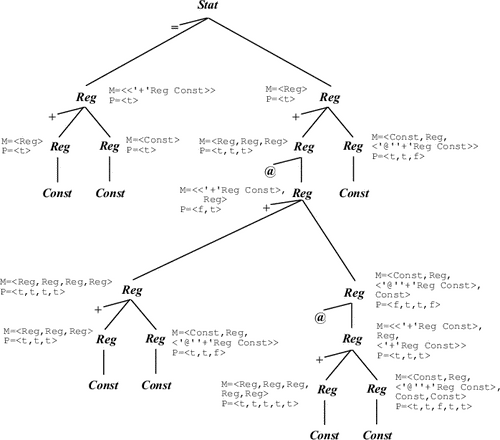


Рисунок 7.9.5

Образцы, соответствующие этому правилу, следующие:

(3) Reg  '@' '+' Reg Const,

(7) Reg  '@' Reg.

Соответственно, атрибуту Match второго символа в качестве образцов при сопоставлении могут быть переданы <'+' Reg Const> (образец 3) или (образец 7). Из анализа других правил можно заключить, что при сопоставлении образцов предков левой части данного правила атрибуту Match могут быть переданы образцы <'@' '+' Reg Const> (из образца 6) и Reg. RULE

Reg ::= Const

SEMANTICS

if (Pattern<0> содержит Const в j-й позиции) Pattern<0>[j]=true;

if (Pattern<0> содержит Reg в i-й позиции) Pattern<0>[i]=true.

**8. Intel Parallel Studio XE для Python-**

**приложений**

Включает комплекс решений и инструментов компании Intel для современных задач искусственного интеллекта: **Intel® Distribution for Python** и **Intel® Nervana Neon**.Они предоставляют доступ к инструментам и методам высокой производительности для преобразования Pythonприложений на современных платформах Intel. Для этих целей в Intel® Distribution for Python включены **библиотека ускорения анализа данных** **Python API of Intel Data Analytics Acceleration Library (DAAL)**.

Платформа **Intel® Nervana Neon** предназначена для раскрытия потенциала современных обучающих средств: технические речевые интерфейсы, поиск изображения, языки перевода, геномики, финансовые услуги, поиск аномалий в области IoT, обеспечивает его этапы от исследования до развертывания.

## 8.1. Версии пакета Intel Parallel Studio XE

Последняя версия пакета Intel Parallel Studio XE доступна в трёх вариантах: Composer, Professional и Cluster Edition.

В первый вариант входят компиляторы (С/С++ и Fortran), а также различные библиотеки:

**В Composer** входят компиляторы (С/С++ и Fortran), а также различные библиотеки:  быстрая математическая библиотека Intel Math Kernel Library (MKL)

* библиотека для обработки данных и мультимедиа Intel Integrated

Performance Primitives (IPP)

* библиотека шаблонов С++ Intel Threading Building Blocks (TBB)

библиотека для машинного обучения и аналитики данных Intel Data Analytics Acceleration Library (DAAL).

**В Professional Edition** добавляются средства динамического анализа приложений

* библиотека для обработки данных и мультимедиа Intel Integrated

Performance такие как профилировщик Intel VTune Amplifier

* средство для прототипирования параллелизма и работы с

векторизацией кода Intel Advisor

поиска ошибок при работе с памятью и потоками Intel Inspector.

### Intel Parallel Studio XE Cluster Edition. В флагманской версии Intel

Parallel Studio XE Cluster Edition доступен весь набор инструментов, который даёт возмлжность создавать и оптимизировать приложения уже на кластере, с помощью Intel MPI и средств для работы с ним.

Во всех версиях студии - появился пакет Intel Distribution for Python, который позволяет использовать Python (2.7 и 3.5), «оптимизированный» инженерами компании Intel, на Windows, Linux и OS X. В самом языке ничего нового не появилось, просто пакеты NumPy, SciPy, pandas, scikit-learn, Jupyter, matplotlib, и mpi4py теперь оптимизированы под «железо» с использованием библиотек Intel MKL, TBB, DAAL и MPI, за счёт чего приложение выполняется более эффективно и быстро.

Кроме всего этого доступна библиотека IMSL от сторонних разработчиков **Rogue Wave**, часто используемая при вычислениях. Она доступна как в пакете с компилятором Fortran, так и как отдельное дополнение. Следующая табличка определяет распределение средств разработки по пакетам:

Изображение выглядит как текст

Автоматически созданное описание

Скачать этот пакет можно абсолютно безвозмездно с сайта Intel. Там же показаны и его преимущества:



Отметим также, что производительность Intel Python приближает его по скорости работы к С/Intel MKL, значительно опережая конкурентов. Кстати, известный профилировщик VTune Amplifier теперь так же поддерживает Python.

**8.2. Состав пакета Intel Parallel Studio XE**.

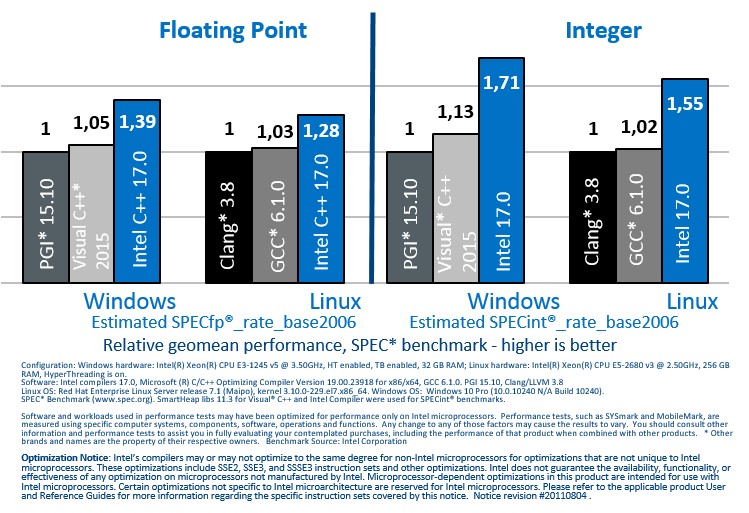
Компиляторы

* Библиотека шаблонов С++ SIMD Data Layout Templates (SDLT) для решения проблемы перехода от массива структур (AoS – Array of Structures) к структуре массивов (SoA – Structure of Arrays), что приводит к уменьшению непоследовательных доступов к памяти и gather/scatter инструкций при векторизации.
* Поддержка процессоров второго поколения Intel Xeon Phi и новое имя mic\_avx512.
* Ещё более полная поддержка OpenMP (как стандарта 4.0, так и более позднего 4.5).
* Новый функционал по оффлоду на Intel Xeon Phi с помощью OpenMP
* Возможность генерировать файлы (html или текстовые) с исходным кодом и интегрированным отчетом об оптимизации, что будет очень полезно разрабочтикам, работающим из командной строки без среды разработки. Кроме того, информативность самих отчетов об оптимизации и векторизации улучшена по многим направлениям.
* Новый атрибут, директива и опция компилятора для выравнивания кода (не данных, а именно самих инструкций для функций или циклов).  Более широкая поддержка стандарта C++14 (полное название: «*International Standard ISO/IEC 14882:2014(E) Programming Language*

*C++*»), в частности теперь можно использовать шаблоны переменных, освобождать память определенного размера глобальным оператором delete, пользоваться constexpr функциями с заметно меньшими ограничениями, чем было в стандарте С++11.

* Стандарт С11 поддерживается теперь полностью (не путать с С++11, который уже давно поддерживается), за исключением ключевого слова \_Atomic и соответствующего атрибута \_\_attribute((atomic)).
* Появился ряд новых опций компилятора, призванных упростить его использование. Например, ключ /fp:consistent (Windows) и -fp-model consistent (Linux), который включает в себя ряд других (/fp:precise /Qimfarch-consistency:true /Qfma-).
* Новые возможности по *offload* вычислений на интегрированную графику с использованием OpenMP для **Intel Xeon Ph**. Например, теперь возможен асинхронный оффлод с помощью клаузы DEPEND в TARGET директиве. Кроме этого, компилятор может производить векторизацию с типом short.

Внесённые изменения, как и ожидалось, позволили генерировать ещё более производительный код, поддерживая все последние новшества в «железе». А сравнения компилятора Intel на бенчмарках с аналогами лишний раз доказывает это.

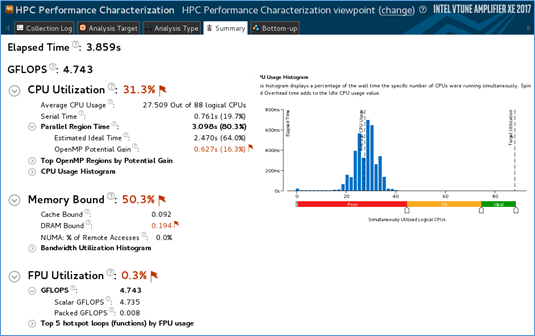


Как видно, на тестах SPECfp и SPECint преимущество версии 17.0 компилятора достаточно внушительное.

## 8.3. Intel VTune Amplifier XE

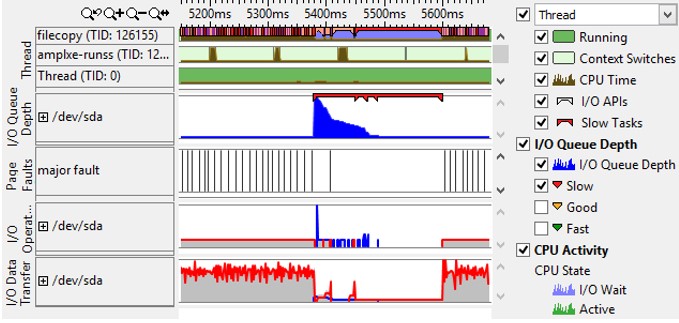
На рынок вышел процессор второго поколения **Intel Xeon Phi (кодовое имя Knights Landing (KNL))**, то конечно же в профилировщике появилась поддержка различных типов анализа для него, чтобы ваши приложения просто «полетели» после оптимизации. Здесь и анализ пропускной способности памяти, чтобы понять, какие данные нужно положить в быструю MCDRAM, и микроархитектурный анализ, и анализ масштабируемости MPI и OpenMP, и ещё много всего вкусного.

Кстати, у VTune имеется анализ HPC Performance Characterization, являющийся своего рода точкой входа для оценки производительности вычислительно-интенсивных приложений, и позволяющий посмотреть в деталях (в отличии от Application Performance Snapshot) на три важных аспекта утилизации «железа» - СPU, память и FPU:



Как уже отмечалось, теперь VTune умеет профилировать приложения на Python, выполняя анализ хотспотов через программное сэмплирование (Basic Hotspots analysis). Ещё одно нововведение – возможность профилировать приложения на языке Go, собирая аппаратные события PMU.

Новый тип анализа появился в версии 2017 – анализ дискового вводавывода (Disk Input and Output analysis). С его помощью можно отслеживать использование дисковой подсистемы, шин CPU и PCIe:



Кроме того, анализ помогает находить операции ввода-вывода с высокой латентностью, а также дисбаланс между ними и реальными вычислениями.

Размер максимальной пропускной способности DRAM теперь определяется автоматически при анализе доступа к памяти (Memory Access analysis), позволяя тем самым понимать насколько хорошо она используется. В этом типе анализа так же добавлена поддержка кастомных аллокаторов памяти, что позволяет правильно определять объекты в памяти. Есть возможность соотнести промахи по кэшу к конкретной структуре данных, а не просто коду, приводящему к нему. Кстати, теперь не требуется установка специальных драйверов на Линуксе для выполнения этого анализа.

Дополнения появились и при профилировании OpenCL и GPU. VTune покажет общую сводку по проблемам с подробным описанием:

Изображение выглядит как текст

Автоматически созданное описание

Кроме этого можно посмотреть исходный код или ASM для ядер OpenCL, а так же отлавливать случаи использование общей виртуальной памяти (Shared Virtual Memory). Более простым стало использование профилировщика удалённо и через командную строку.

Новая группа Arbitrary Targets позволяет сгенерировать командную строку для анализа производительности на системе, недоступной с текущего хоста. Это может быть весьма полезным для микроархитектурного анализа, так как дает доступ к аппаратным событиям, доступным на платформе, которую мы выбрали:

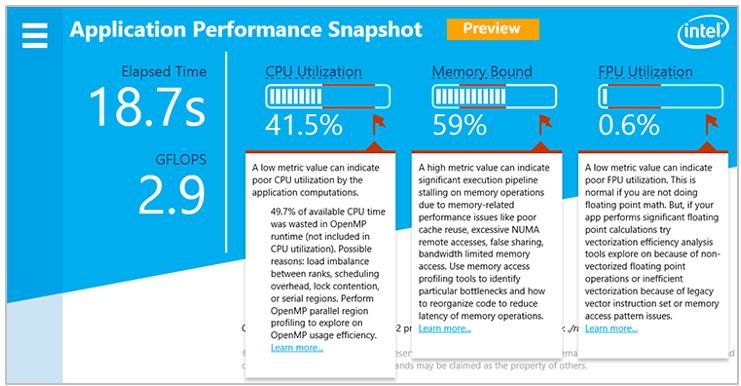
Изображение выглядит как текст

Автоматически созданное описание

В пакете с Parallel Studio появляются специальные «легковесные» инструменты, позволяющие получать общее представление о нашем приложении. Они получили вывеску Performance Snapshots, и имеют на данный момент три варианта:

* Application – для не MPI приложений
* MPI – правильно, для MPI приложений
* Storage – для анализа использования хранилища

Что они из себя представляют? Рассмотрим, для примера, инструмент Application Performance Snapshot:



Запустив его, на выходе мы имеем выосокоуровневую сводку по тому, где наше приложение может потеницально выиграть. С другой стороны, можно назвать это слабыми местами, которые нужно поправить. Мы получаем время выполнения приложения, производительность в флопсах, а также ответ на 3 вопроса: как эффективно использовались вычислительные ядра CPU (CPU Utilization), как много мы тратили на работу с памятью (Memory Bound), и как использовался FPU (FPU Utilization).

В итоге представляется общая картина о приложении. Если CPU Utilization показывает слишком маленькие значения, значит имеются проблемы с «параллельностью» приложения и эффективностью использования ядер. Например, дисбаланс нагрузки, слишком большие накладные расходы при планировке, проблемы синхронизации или же просто большое количество регионов, выполняемых последовательно. Следующая метрика скажет нам, упирается ли приложение в ограничения по памяти. Если цифры велики, значит возможны проблемы с использованием кэшей, ложным разделением данных (false sharing) или мы упираемся в пропускную способность.

Ну а характеристика FPU Utilization может показать, что имеются проблемы с векторизацией (её отсутствием или неэффективностью), при условии наличия вычислений с числами с плавающей точкой.

## Intel Advisor

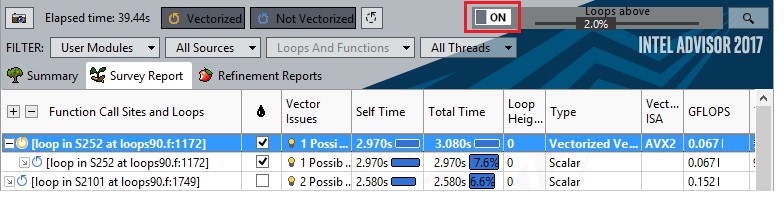
В предыдущей версии Advisor получил огромный функционал для работы с векторизацией, который был расширен и улучшен в новой версии.

Конечноб, появилась поддержка нового поколения Xeon Phi, как и в других средствах, и AVX-512. Например, Advisor может показать использование нового набора инструкций AVX-512 ERI, специфичного для KNL. Но даже если у нас нет пока новой железки, мы сможем получить определенные данные для AVX-512. Для этого необходимо дать указание компилятору генерировать две ветки кода (AVX и AVX-512) для циклов с помощью опций –axMIC-AVX512 –xAVX, а Advisor в своих результатах покажет информацию даже для той ветки, которая реально не выполнялась:

Изображение выглядит как стол

Автоматически созданное описание

Кроме этого, появилась возможность измерения FLOPS'ов для всех процессоров, за исключением Xeon Phi предыдущего поколения, основанная на использовании инструментации и сэмплирования. Был также улучшен анализ доступа к памяти (Memory Access Analysis), который теперь даст нам ещё больше полезной информации (сравнение с размером кэша, выявление ненужных gather/scatter). В плане удобства использования также ряд улучшений. «Умный режим» (Smart Mode) позволит показать только те циклы, которые нам наиболее интересны, нажатием одной кнопки:



Выбрать сразу несколько типов анализа и запустить их за один раз стало возможным с помощью пакетного режима Batch Mode.

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) [↑](#footnote-ref-2)