**Методический материал для лабораторной работы 1 . Элементы теории трансляции. Формальные языки, грамматики и их свойства.**

1. **Основные задачи компиляторов**

Компьютеры сами по себе способны выполнять только очень ограниченный набор операций, называемых машинными кодами. При появлении первых компьютеров, программы писались в машинных кодах, представляющих собой последовательности двоичных чисел, однозначно воспринимаемых компьютером. В конце 50-х кодов прошлого века появились первые языки программирования, **такие как язык ассемблера и Фортран.** Для того, чтобы компьютер мог понять программу, написанную на каком-то языке программирования, необходим переводчик (транслятор) такой программы в машинные коды. Отметим, что, если оператор языка ассемблера отображается при трансляции чаще всего в одну машинную инструкцию, предложения ( высказывания, инструкции, statement) языков более высокого уровня отображаются, вообще говоря, в несколько машинных инструкций. **Трансляторы бывают двух типов: компиляторы (compiler) ( Рис. 1 ) и интерпретаторы (interpreter) (Рис. 2 ).**

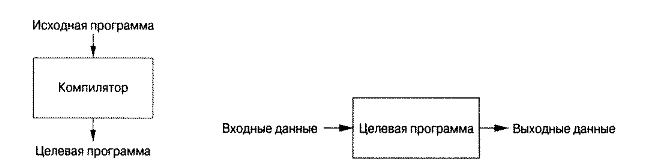


Рис. 1 Компилятор

**В отличие от компилятора, интерпретатор не создает никакой новой программы (целевой программы) ( Рис. 2 ),** а просто выполняет каждое предложение ( инструкцию, высказывание) языка программирования. Можно сказать, что результатом работы интерпретатора является "число".

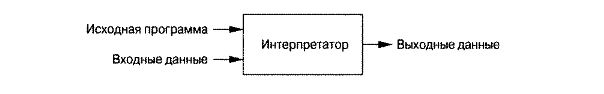


Рис. 2 Интерпретатор

Вообще говоря, 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.

1. **Компилятор**

Компилятор переводит программы с одного языка на другой. Входом компилятора служит цепочка символов, составляющая исходную программу на языке программирования L1. Выход компилятора (объектная программа или целевая 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 как об отображении множества L1 в множество L2, т.е. KL3: L1 —> L2.

1. **Структура компилятора**

Процесс компиляции состоит из двух частей: **анализа** (analysis) и **синтеза** (synthesis).

**Анализирующая часть** компилятора разбивает исходную программу на составляющие ее элементы (конструкции языка) и создает промежуточное представление исходной программы. В качестве промежуточного представления обычно используются деревья, в частности, так называемые деревья разбора. Под деревом разбора понимается дерево, каждый узел которого соответствует некоторой операции, а сыновья этого узла - операндам. Анализ также собирает информацию об исходной программе и сохраняет ее в структуре данных, именуемой таблицей символов, которая передается вместе с промежуточным представлением синтезу.

**Синтезирующая часть** из промежуточного представления создает новую программу (целевую программу), которую компьютер в состоянии понять. То есть **Синтез** строит целевую программу на основе промежуточного представления и информации из таблицы символов. Целевая программа (объектная программа) может в дальнейшем выполняться без перетрансляции.

**Анализ** часто называют начальной стадией (front end), а **Синтез** – заключительной (back end). Если рассматривать процесс компиляции более детально, то он представляет собой последовательность фаз, каждая из которых преобразует одно из представлений исходной программы в другое.

**Таблица символов**, в которой хранится информация обо всей исходной программе, используется всеми фазами компилятора. Важная функция компилятора состоит в том, чтобы записывать имена переменных в исходной программе и накапливать информацию о разных атрибутах каждого имени. Эти атрибуты могут представлять информацию о выделенной памяти для данного имени, его типе, области видимости, в случае процедур, такие сведения как количество аргументов, метод передачи каждого аргумента, а также возвращаемый тип.

**Типичное разложение компилятора на фазы приведено на Рис. 3**

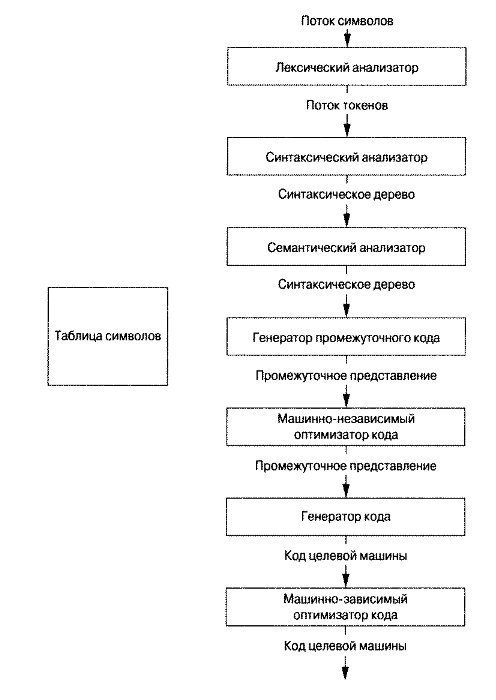


Рис. 3. Фазы компилятора

1. **Лексический анализ или сканирование.** Лексический анализатор читает поток символов, составляющих исходную программу, и группирует исходные символы в значащие последовательности называющимися лексемами. Для каждой лексемы анализатор строит выходной токен вида: < Имя\_токена , Значение\_атрибута >.
2. **Синтаксический анализ или разбор.** Синтаксический анализатор использует первые компоненты токенов, полученных при лексическом анализе для создания древовидного промежуточного представления, которое описывает грамматическую структуру потока токенов. Типичным представлением является синтаксическое дерево, где каждый внутренний узел представляет операцию, а дочерние узлы – аргументы этой операции.
3. **Семантический анализатор** использует семантическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность исходной программы с определением языка. Он также сохраняет информацию о типах либо в синтаксическом дереве, либо в таблице символов для последующего использования в процессе генерации промежуточного кода. Важной частью семантического анализа является проверка типов.
4. **Генерация промежуточного кода.** Выход генератора промежуточного кода представляет собой последовательность трехадресных кодов (трехадресных команд). Такое промежуточное представление обладает двумя важными свойствами: легко генерируется и легко транслируется в целевой машинный язык. Трехадресная команда содержит три операнда, и каждый операнд может действовать, как регистр.
5. **Оптимизация кода.**
6. **Генерация кода.** Генератор кода получает промежуточное представление исходной программы и отображает его в целевой язык. Если целевой язык представляет собой машинный код, то для каждой переменной, используемой программой, выбираются соответствующие регистры и ячейки памяти. Затем промежуточные команды транслируются в последовательность машинных команд выполняющих те же действия. Организация памяти во время выполнения программы зависит от компилируемого языка. Решение о распределение памяти принимаются либо в процессе генерации промежуточного кода, либо при генерации целевого кода.

На рисунке 4 приведен **пример компиляции инструкции присваивания** –

**position = initial +rate\*60**

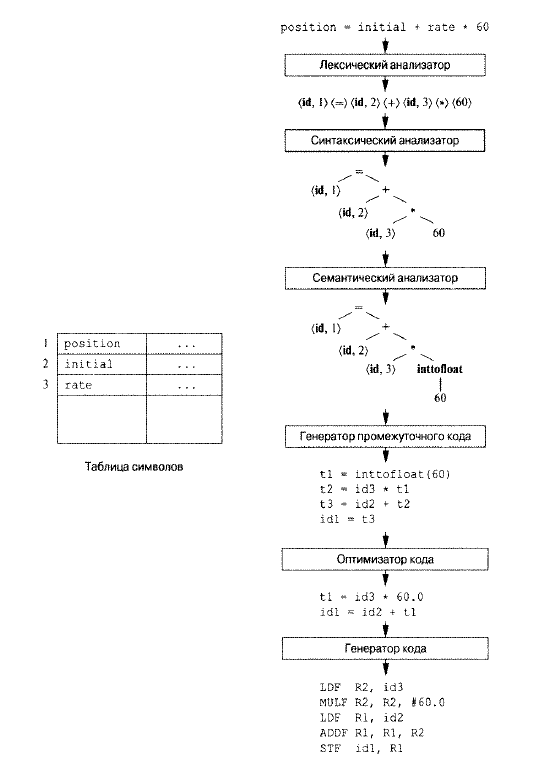


Рис 4 . Компиляция инструкции присваивания

**Пояснение по оптимизирующей фазе** компиляции. Использование возможностей автоматической оптимизации компилятора может дать значительный выигрыш в производительности. ***Примеры оптимизации:***

**Удаление общих подвыражений.** Если одно и то же подвыражение встречается несколько раз, компилятор может вычислить его один раз, а все последующие включения заменяются вычисленным значением:

a = (b+35) \* (b+35);

c = 12.0 \* (b+35);

в результате после удаления общего подвыражения имеем следующее представление:

tmp = b + 35;

a = tmp \* tmp;

c = 12.0 \* tmp;

**Развертка цикла.** Развертка заключается в преобразовании цикла в линейную последовательность операций. Выигрыш в производительности достигается благодаря снижению накладных расходов на организацию цикла. Эффективным этот прием может быть в случае, когда тело цикла очень маленькое:

for (i = 0; i < 2; i++) a[i] = 2.0 \* d[i];

После развертки:

a[0] = 2.0 \* d[0];

a[1] = 2.0 \* d[1];

a[2] = 2.0 \* d[2];

**Удаление инвариантных выражений из тела цикла.** Выражения, не зависящие от счетчика цикла (инварианты цикла), целесообразно выносить за пределы цикла, так как это позволяет уменьшить количество операций:

for (i = 0; i < 10; i++) {

a[i] = b \* 2.0 + 0.35;

}

преобразуется следующим образом:

temp = b \* 2.0 + 0.35;

for (i = 0; i < 10; i++) {

a[i] = temp;}

**Подстановка функций.** Это преобразование заключается в замене обращения к функции ее кодом. Выигрыш в производительности может быть получен благодаря уменьшению накладных расходов на вызов функции и возвращение из нее, а также на передачу параметров.Часто подстановка функции увеличивает эффективность использования кэш-памяти, так как код становится непрерывным.Недостатком является увеличение размера программы, которое тем более заметно, чем больше размер функции и чем больше число обращений к ней.

**Свертка и распространение констант.** Подвыражение, содержащее только константы, вычисляется, а результат его вычисления подставляется в код программы:

a = b + 2.0 / 3.0;

Компилятор заменит данный код следующим:

a = b + 0.666666666666666666667;

Первый вариант более удобен для программиста, поскольку улучшает читаемость кода.

**Исключение указателей.** При автоматической оптимизации указатель или ссылка могут быть удалены, если известен их адресат:

void sub(int \* p) {

\*p = \*p - 5;}

int main()

{

int a;

a = 10;

sub(&a);

}

В результате преобразования генерируется следующий код:

a = 10;

a -= 5;

**Объединение ветвей.** Этот вид оптимизации заключается в объединении одинаковых фрагментов кода, например фрагмент кода:

if (b) { y = x\*x; *z = y \* 1.5;* }

else {

y = sin(x); *z = y \* 1.5;*

}

после оптимизации будет выглядеть следующим образом:

if (b) { y = x\*x; }

else { y = sin(x); }

*z = y \* 1.5;*

1. **Инструментарий для создания компиляторов**
2. **Генераторы синтаксических анализаторов,**  которые автоматически создают синтаксические анализаторы на основе грамматического описания языка программирования.
3. **Генераторы сканеров,** которые создают лексические анализаторы на основе описания токенов языка с использованием регулярных выражений.
4. **Средства синтаксически управляемой трансляции,** которые создают наборы программ для обхода синтаксического дерева и генерации промежуточного кода.
5. **Генераторы генераторов кода,** которые создают генераторы кода на основе набора правил трансляции каждой операции промежуточного языка в машинный язык для целевой машины.
6. **Средства работы с потоком данных**, которые облегчают сбор информации о передаче значений от одной части программы ко всем другим. Анализ потоков данных представляет собой ключевую часть оптимизации кодов.
7. **Наборы для построения компиляторов**, которые представляют собой интегрированные множества подпрограмм для построения разных фаз компиляторов.
8. **Эволюция языков программирования и влияние на компиляторы**

**Языки первого поколения –** это машинные языки, **языки второго поколения** – языки ассемблера, **языки третьего поколения** - высокоуровневые языки программирования такие, как Fortran, Cobol, Lisp, Cи ++, C# и Java. **Язык четвертого поколения** – это языки программирования разработанные для конкретных применений, например NOMAND – для генерации отчетов , SQL – для запросов к базам данных. Термин **языки пятого поколения** применяются к языкам программирования, основанным на логике, как пример язык Prolog .

Новые достижения в языках программирования приводят к новым требованиям, возникающим перед разработчиками компиляторов, которые должны разрабатывать алгоритмы и представления для трансляции и поддержки новых возможностей языка. Кроме того после 1940х годов произошли существенные изменения в архитектуре вычислительных систем. В связи с эти разработчики должны учитывать не только новые свойства языков программирования, но и разрабатывать такие алгоритмы трансляции, которые смогут максимально использовать преимущества новых аппаратных возможностей. В действительности производительность вычислительной системы настолько зависит от технологии компиляции, что компиляторы используются в качестве инструмента для оценки архитектурных концепций перед созданием компьютера.

1. **Порождающие модели**

Для описания синтаксиса языков программирования наибольшее распространение получили следующие порождающие модели:

* форма Бекуса-Наура и ее модификации,
* диаграммы Вирта
* **формальные грамматики.**

В порождающих моделях **правила задаются продукциями**, состоящими из двух частей: левой и правой. В левой части, записывается определение, в правой части варианты, из которых может состоять определение.

**Определение.**  Унарный оператор, замыка́ние Кли́ни или звезда́ Кли́ни, обозначается V\* определяется над множеством символов V, замкнут относительно операции конкатенации символов из V. Результатом **замыка́ния Кли́ни** является бесконечное надмножество множества V или **множество цепочек символов**, включая пустую цепочку **ε**. Например, если конечное множество V={0,1}, то V\* = {ε, 0, 1, 00, 11, 01, 10, 000, 001, 011, ...} есть бесконечное множество цепочек символов.

Часто применяется замыка́ние Кли́ни без пустой цепочки ε, обозначается как V+ – бесконечное множество всех цепочек алфавита V, исключая пустую цепочку ε. Следовательно, V\* = V+ {ε}.

**Определение.** Оператор импликации (от лат. implicatio — «связь») **A → B** - **бинарная логическая связка или бинарное отношение**, читается "если A, то B". Применяется в правилах продукций (порождений) грамматик A → B, читается "символ A порождает символ B".

**Определение**. **Бинарный оператор вывода** означает, что символы слева от оператора заменяется символами справа, в соответствии с правилом продукции грамматики оператор применяется в порождающих системах. Например, A → Bb, B → c, тогда A Bb bc. Принцип подстановки. Полученная цепочка называется **цепочкой вывода или выводом**.

**Определение.** Выводом называется процесс порождения предложения языка на основе правил определяющей язык грамматики.

**Определение**. Грамматика G = (T, V, P, S0),

где Т - конечное множество терминальных символов (терминалов) алфавита;

V – конечное множество нетерминальных символов алфавита, не пересекающихся с T, T ∩V= ∅,

S0 - начальный символ (или аксиома или **цель грамматики** ), S0 ∈ V;

P - конечное множество правил порождения (продукций), P = (T ∪V)+×(T ∪V)\*. Элемент (α, β) множества P называется **правилом порождения** и записывается в виде α → β. α, β ∈ (T ∪V)\*, слова в объединенном алфавите.

Правила порождения с одинаковыми левыми частями α → β1, α → β2 ,..., α → βn , записывают сокращенно α→ β1 | β2 |...| βn , где βi , i= 1, 2,..., n и называют альтернативой правил порождения из цепочки α.

*Грамматика позволяет осуществить грамматический анализ предложения, а значит, и явно описать его структуру. Одно из назначений грамматики – это определить бесконечное число предложений языка с помощью конечного числа правил.*

*Грамматика языка может быть реализована двумя способами: в виде* ***порождающей системы*** *и в виде устройств, называемых* ***распознавателями****.*

**Определение.** Языком, порождаемым грамматикой G = (T, V, P, S0), называется множество L(G)={α ∈T\* | S0 ⇒\* α}.

**Другими словами, L(G) - это все цепочки над множеством алфавита T, которые выводимы из S0 подстановкой правил из P.**

Цепочка α∈ (T ∪V)\*, для которой S0 ⟹\* α (то есть цепочка, которая может быть выведена из начального( целевого) символа), называется сентенциальной формой в грамматике G. **Язык, порождаемый грамматикой, можно определить как множество терминальных сентенциальных форм (конечных** **сентенциальных форм).**

**Пример.**  “он смотрит кино” имеем грамматику (грамматика реализована в виде порождающей системы)

G = ({“он”, “кино”, “смотрит”},{Предложение, Подлежащее, Группа сказуемого, Сказуемое, Дополнение}, P, S0), где S0 = Предложение, P состоит из правил Р= {p1,p2,p3,p4,p5};

p1: Предложение → Подлежащее, Группа сказуемого

p2: Подлежащее → он

p3: Группа сказуемого → Сказуемое, Дополнение

p4: Сказуемое → смотрит

p5: Дополнение → кино

**Определение.** Грамматики G1 и G2 называются эквивалентными, если L(G1) = L(G2).

**Определение.** Грамматики G2 и G3 называются почти эквивалентными L(G2) = L(G3), если L(G2) {ε}=L(G3){ε}. Другими словами, грамматики почти эквивалентны, если языки, ими порождаемые, отличаются не более чем на ε.

1. **Классификация грамматик и языков по Хомскому (англ. Chomsky hierarchy)**

*(Грамматики классифицируются по виду их правил вывода)*

**Иерархия Хомского** — классификация формальных языков и формальных грамматик, согласно которой они делятся на 4 типа по их условной сложности. Классификация предложена профессором Массачусетского технологического института, лингвистом Ноамом Хомским. Каждому классу грамматик из иерархии Хомского соответствует класс распознавателей, определяющий тот же класс языков.

Языки классифицируются согласно иерархии Хомского в соответствии с типами грамматик, с помощью которых они заданы, причем из всех эквивалентных грамматик, задающих один и тот же язык, выбирается грамматика с максимально возможным номером, т.е. самая простая. Сложность языков соответствует сложности грамматик**. От классификационного типа языка зависит и сложность распознавателя этого языка.**



Рис. 5. Классификация языков в соответствии с типами грамматик

**1. Тип 0 – языки с фразовой структурой**

Это самые сложные языки, для распознавания которых требуются вычислители, равномощные машине Тьюринга. Для такого языка невозможно построить компилятор, который выполнил бы разбор за ограниченное время на основе ограниченных вычислительных ресурсов. Практически все естественные языки относятся к этому типу. Одно и то же слово в естественном языке может иметь различный смысл в зависимости от контекста и играть различную роль в предложении. Такие языки далее рассматриваться не будут.

Грамматика G = (T, V, P, S0) называется **грамматикой типа 0**, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики).

**2.** **Тип 1 – контекстно-зависимые (КЗ) языки .** В общем случае время на распознавание языка типа 1 экспоненциально зависит от длины исходной цепочки символов. Языки и грамматики этого типа используются в переводе текстов на естественных языках. Распознаватели, построенные на их основе, позволяют анализировать тексты с учётом контекстной зависимости в предложениях входного языка, хотя в общем случае для точного перевода всё же требуется вмешательство человека. Такие грамматики могут использоваться в сервисных функциях проверки орфографии в языковых процессорах. Однако языки программирования имеют более простую структуру, поэтому в компиляторах КЗ-языки не применяются.

Грамматика G = (T, V, P, S0) называется **неукорачивающей грамматикой**, если каждое правило из P имеет вид α → β, где α ∈ (T ∪ V)+, β ∈ (VT ∪ VN)+ и | α | <= | β |.

Грамматика G = (VT, VN, P, S) называется контекстно-зависимой ( КЗ ), если каждое правило из P имеет вид α → β, где α = ξ1Aξ2; β = ξ1γξ2; A ∈ V; γ ∈ (T ∪ V)+; ξ1,ξ2 ∈ (T ∪ V)\*.

**Грамматику типа 1** можно определить *как неукорачивающую либо как контекстно-зависимую.* Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых неукорачивающими грамматиками, совпадает с множеством языков, порождаемых КЗ-грамматиками.

**3. Тип 2 – контекстно-свободные (КС) языки**

КС-языки лежат в основе большинства современных языков программирования, на их основе работают некоторые командные процессоры, допускающие управляющие команды цикла и условия. В общем случае время на распознавание предложений языка этого типа полиномиально зависит от длины цепочки символов (это кубическая или квадратичная зависимость в зависимости от класса языка). Но среди КС языков существует много классов, для которых эта зависимость линейна, и многие языки программирования можно отнести к одному из таких классов. КС-языки будут рассматриваться подробно.

**Грамматика G = (T, V, P, S0) называется контекстно-свободной ( КС )**, если каждое правило из Р имеет вид A → β, где A ∈ V, β ∈ (T ∪ V)+.

**Грамматика G = (T, V, P, S0) называется укорачивающей контекстно-свободной** ( УКС ), если каждое правило из Р имеет вид A → β, где A ∈ V, β ∈ (T ∪ V)\*.

Грамматику типа 2 можно определить как контекстно-свободную либо как укорачивающую контекстно-свободную. Возможность выбора обусловлена тем, что для каждой УКС-грамматики существует почти эквивалентная КС-грамматика.

**4.**  **Тип 3**  **– регулярные языки**

Это самый простой тип языков, и они являются наиболее широко распространенным типом, используемым в вычислительных системах. Время на распознавание цепочек языка линейно зависит от их длины. Поэтому иногда эти языки ещё называют линейными.

Грамматика G = (T, V, P, S0) **называется праволинейной**, если каждое правило из Р имеет вид

A → tB либо A → t, где A ∈ V, B ∈ V, t ∈ T.

**Вывод называется левосторонним**, если в нём на каждом шаге вывода правило грамматики применяется всегда к крайнему левому нетерминальному символу в цепочке. Другими словами, если на каждом шаге вывода происходит подстановка цепочки символов на основании правила грамматики вместо крайнего левого нетерминального символа в исходной цепочке.

Грамматика G = (T, V, P, S0) **называется леволинейной**, если каждое правило из Р имеет вид

A → Bt либо A → t, где A ∈ V, B ∈ V, t ∈ T.

**Вывод называется правосторонним**, если в нём на каждом шаге вывода правило грамматики применяется всегда к крайнему правому нетерминальному символу в цепочке.

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) для любой сентенциальной формы всегда можно построить левосторонний или правосторонний вывод. Для грамматик других типов это не всегда возможно, так как по структуре их правил не всегда можно выполнить замену крайнего левого или крайнего правого нетерминального символа в цепочке.

Грамматику типа 3 (регулярную, Р-грамматику или автоматную грамматику) можно определить как праволинейную либо как леволинейную. Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых праволинейными грамматиками, совпадает с множеством языков, порождаемых леволинейными грамматиками.

**Соотношения между типами грамматик:**

* любая регулярная грамматика является КС-грамматикой;
* любая регулярная грамматика является УКС-грамматикой;
* любая КС- грамматика является УКС-грамматикой;
* любая КС-грамматика является КЗ-грамматикой;
* любая КС-грамматика является неукорачивающей грамматикой;
* любая КЗ-грамматика является грамматикой типа 0.
* любая неукорачивающая грамматика является грамматикой типа 0.
* любая УКС-грамматика является грамматикой типа 0.

**Замечание:** УКС-грамматика, содержащая правила вида A → ε, не является

КЗ-грамматикой и не является неукорачивающей грамматикой.

**Определение:** язык L(G) является языком типа k, если его можно описать грамматикой типа k.

**Соотношения между типами языков:**

* каждый регулярный язык является КС-языком, но существуют КС-языки,

которые не являются регулярными ( например, L = { | n>0}).

* каждый КС-язык является КЗ-языком, но существуют КЗ-языки, которые

не являются КС-языками ( например, L = { | n>0}).

* каждый КЗ-язык является языком типа 0.

**Замечание:** УКС-язык, содержащий пустую цепочку, не является КЗ-языком.

**Замечание:** следует подчеркнуть, что если язык задан грамматикой типа k, то это не значит, что не существует грамматики типа k’ (k’>k), описывающей тот же язык. Поэтому, когда говорят о языке типа k, обычно имеют в виду максимально возможный номер k.

**Пример**

**Грамматика типа 0 - G1** = ({0,1}, {A,S}, P1, S) , где

P1: S → 0A1 0A → 00A1

A → ε

**КС-грамматика - G2** = ({0,1}, {S}, P2, S), где

P2: S → 0S1 | 01

Грамматики G1 и G2 описывают один и тот же язык L = L(G1) = L(G2) = { | n>0}. Язык L называют КС-языком, т.к. существует КС-грамматика, его описывающая. Но он не является регулярным языком, т.к. не существует регулярной грамматики, описывающей этот язык.

Если язык L может быть задан грамматиками G1 и G2 относящимися к типу 1 (КЗ ) грамматикой G3 относящейся к типу 2 ( КС ) и грамматикой G4 относящейся к типу 3 (регулярные) сам язык должен быть отнесен к типу 3 и является регулярным языком.

Классификация Хомского дает представление о 4 типах грамматик и об эквивалентных формальных автоматах (машина Тьюринга (для грамматик type-0 , type-1), конечные автоматы (для грамматик type-3 – регулярных языков) и магазинные автоматы (для грамматик type-2 – КС языки)), располагающихся по иерархии от 0 до 3 в порядке убывания их общности. Грамматики классифицируются по ограничениям накладываемым на правила вывода P. В правилах вывода Р большими латинскими буквами обозначают **нетерминальные** символы, маленькими латинскими буквами - **терминальные**.

1. **Примеры грамматик и языков.**

Замечание: если при описании грамматики указаны только правила вывода Р, то будем считать, что большие латинские буквы обозначают нетерминальные символы, S - цель грамматики или начальный нетерминал, все остальные символы - терминальные.

1) **Язык типа 0:** L(G) = { | n >= 1}

G: S → aaCFD

F → AFB | AB

AB → bBA

Ab → bA

AD → D

Cb → bC

CB → C

bCD → ε

2) **Язык типа 1:** L(G) = { , n >= 1}

G: S → aSBC | abC

CB → BC

bB → bb

bC → bc

cC → cc

3) **Язык типа 2:** L(G) = { | n > 0}

G: S → aQb | accb

Q → cSc

4) **Язык типа 3:** L(G) = {ω ⊥ | ω ∈ {a,b}+, где нет двух рядом стоящих а}

G: S → A⊥ | B⊥

A → a | Ba

B → b | Bb | Ab