1. Система программирования: определение, состав назначение. Классический жизненный цикл разработки программного обеспечения. Текстовый редактор. Интегрированная среда разработки. Программный продукт. Общая схема преобразования исходного кода в процесс операционной системы.

**Определение**: комплекс программных средств, предназначенный для реализации программного обеспечения (написания и отладки программного кода).

**Состав:**

* **трансляторы** - программа, преобразующая исходный код на одном языке программирования в исходный код на другом языке, **компоновщики** - программа, принимающая один или несколько объектных модулей и формирующая на их основе загрузочный модуль,
* **отладчики** - программа, позволяющая контролировать ход выполнения программы (приостанавливать, выполнять пошагово), просматривать и изменять области памяти,
* **профилировщики** - программа, позволяющая оптимизировать код программы (устранять утечки памяти, оптимизировать циклы, анализировать производительность),
* **программные библиотеки**,
* **редакторы кода**,
* **системы поддержки версий**.

**Назначение:** система программирования представляет собой совокупность средств разработки программ, обеспечивающих автоматизацию составления и отладки программ пользователя.

**Классический жизненный цикл разработки программного обеспечения:**

**Жизненный цикл программного обеспечения** (ПО) — период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл — процесс построения и развития ПО.

****

**Текстовый редактор:** компонента системы программирования - программа, позволяющая подготовить исходный код программы.

**Интегрированная система разработки:** набор инструментов для разработки и отладки программ, имеющий общую интерактивную графическую оболочку, поддерживающую выполнение всех основных функций жизненного цикла разработки программы.

**Интегрированная среда разработки** (IDE, Integrated development environment): редакторы кода, транслятор, компоновщик, отладчик, система поддержки версий. Примеры: Visual Studio, NetBeans, Eclipse, Embarcadero Delphi и пр.

**Программный продукт:** программа, работающая без авторского присутствия; исполняется, тестируется, конфигурируется без присутствия автора; сопровождается документацией.

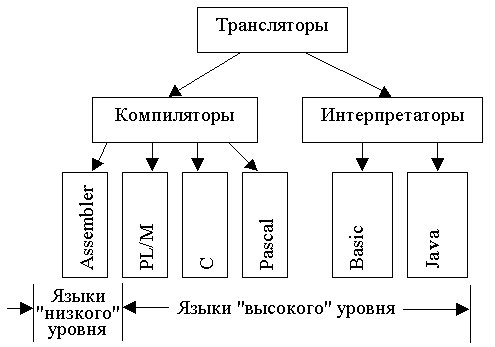
***Общая схема преобразования исходного кода в процесс операционной системы:***



**Транслятор**: программа, преобразующая исходный код на одном языке программирования в исходный код на другом языке.

Пример: **Ассемблер**: транслятор с исходного кода на языке ассемблера в программу на машинном языке (язык, который может интерпретироваться процессором).

**Интерпретатор** - разновидность транслятора. Переводит и выполняет программу с языка высокого уровня в машинный код строка за строкой.



1. Язык программирования: определение, назначение, примеры. Исходный код. Язык ассемблера. Транслятор: определение, назначение, примеры. Алфавит языка. Ассемблер. Объектный код, объектный модуль. Общая схема преобразования исходного кода в процесс операционной системы.

**Язык программирования**: формальная знаковая система, предназначенная для записи компьютерных программ. Знаковая система определяет набор лексических, синтаксических и семантических правил написания программы (программного кода). Язык программирования представляется в виде набора спецификаций, определяющих его синтаксис и семантику.

Примеры: процедурно-ориентированные, объектно-ориентированные, декларативные, операторные, функциональные, скриптовые, проблемно-ориентированные, машинно-зависимые (ассемблеры) и т.п.

**Исходный код (исходная программа)**: текст программы, написанный на языке программирования.

Программа на исходном языке (исходный код) готовится с помощью текстовых редакторов (программ, позволяющих подготовить исходный код программы) и в в

иде текстового файла или раздела библиотеки поступает на вход транслятора.

**Язык ассемблера:** машинно-ориентированный язык программирования (для конкретной архитектуры компьютера, команды которого соответствуют машинным командам)

**Транслятор:** программа, преобразующая исходный код на одном языке программирования в исходный код на другом языке.

**Трансляторы** (компиляторы) полностью обрабатывают весь исходный текст программы (исходный код). Они просматривают его в поисках синтаксических ошибок (иногда несколько раз), проверяют семантические правила языка.

**Интерпретатор** - разновидность транслятора. Переводит и выполняет программу с языка высокого уровня в машинный код строка за строкой

**Примеры:** диалоговый, синтаксически-ориентированный, однопроходной, многопроходной, оптимизирующий, тестовый, обратный

**Алфавит языка:** набор символов, разрешенных к использованию языком. Основывается на одной из кодировок.

**Ассемблер:** транслятор с исходного кода на языке ассемблера в программу на машинном языке (язык, который может интерпретироваться процессором).

**Объектный код:** результат работы транслятора. Один файл объектного кода – **объектный модуль**.

**Общая схема преобразования исходного кода в процесс операционной системы.**

1. Компоновщик: определение, назначение. Загрузочный код. Загрузчик: определение, назначение. Общая схема преобразования исходного кода в процесс операционной системы.

**Компоновщик:**

**Определение:** программа, принимающая один или несколько объектных модулей и формирующая на их основе загрузочный модуль.

**Назначение:** если программа собирается из нескольких объектных файлов, компоновщик может собирать эти файлы в единый исполнимый модуль, вычисляя и подставляя [адреса](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D1%80%D0%B5%D1%81_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29) вместо символов, в течение времени компоновки (статическая компоновка) или во время исполнения (динамическая компоновка).

**Загрузочный код:** результат работы компоновщика. Один файл загрузочного кода – **загрузочный модуль**.

**Загрузчик:**

**Определение:** программа, обычно входящая в состав операционной системы, предназначенная для запуска процесса операционной системы на основе загрузочного модуля.

**Назначение:** отвечает за загрузку исполнимых файлов и запуск соответствующих новых процессов.

***Общая схема преобразования исходного кода в процесс операционной системы:***



1. Кодировка: определение, назначение, примеры. Кодировка ASCII. Структура кодировки Windows-1251. Отличие ASCII и Windows-1251. Кодировка UNICODE: назначение, структура, UCS, UTF. Прямой (LE) и обратный (BE) порядок байт. BOM: определение, назначение, примеры.

**Кодировка** - таблица, задающая кодировку конечного множества символов алфавита (обычно элементов текста: букв, цифр, знаков препинания). Такая таблица сопоставляет каждому символу последовательность длиной в один или несколько символов другого алфавита (точек и тире в коде Морзе,  нулей и единиц (битов) в компьютере).

Примеры: ASCII, Windows-1251, Кодировки, UTF-8, UTF-16 и UTF-32 набора символов Юникод

**ASCII: American Standard Code for Information Interchange —** американский стандартный код для обмена информацией. ASCII представляет собой 8-битную кодировку для представления десятичных цифр, латинского и национального алфавитов, знаков препинания и управляющих символов. Нижнюю половину кодовой таблицы (0 — 127) занимают символы US-ASCII, а верхнюю (128 — 255) — разные другие нужные символы (CP866, CP1251).

**Windows-1251** — набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий MW.

Кодировка **ASCII**  - кодировка для представления латинского алфавита, десятичных цифр, некоторых знаков препинания, арифметических операций и управляющих символов. **Windows-1251** – для символов русского алфавита.

**UNICODE:** это стандарт кодирования символов, позволяющий представить знаки почти всех письменных [языков](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA), состоит из 2х разделов:

* UCS - universal character set (универсальный набор символов);
* UTF - Unicode transformation format (семейство кодировок).

Принято обозначение U+xxx, где xxx- число в шестнадцатеричном формате.

**UCS** расположены в 17 плоскостях (0-16), 216 (65 536) символов в каждой плоскости, плоскость 0 – основная (основные символы), 1-14 – дополнительные, 15-16 – для частного использования.

**UTF-8** — представление Юникода, обеспечивающее совместимость со старыми системами, использовавшими 8-битные символы.

Алгоритм кодирования в UTF-8:

* определить количество октетов (октет: 8 битов или 1 байт) – в какой диапазон значений попадает количество значащих символов (7, 11, 16, 21, 26, 31);
* подготовить старшие биты первого октета:
  + 0xxxxxxx для одного октета;
  + 110xxxxx – двух;
  + 1110xxxx - трех и т.д..
  + 10xxxxxx - для остальных октетов;
* заполнить оставшиеся биты (обозначены как x) в октетах кодом символа Юникода в двоичном виде. Начать с младших битов, поставив их в младшие биты последнего октета кода. И так далее, пока все биты кода символа не будут перенесены в свободные биты октетов.

В **UTF-16** символы кодируются двухбайтовыми словами с использованием всех возможных диапазонов значений (от 0 до FFFF16).

**LE** (Little endian order, прямой порядок, от младшего к старшему), **BE** (Big endian order, обратный порядок, от старшего к младшему).

**BOM**: Для определения формата представления Юникода в начало текстового файла записывается сигнатура (обозначение) — символ U+FEFF — маркер последовательности байтов.

|  |  |
| --- | --- |
| **Кодировка** | **Представление (**[**hex**](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D1%81%D1%82%D0%BD%D0%B0%D0%B4%D1%86%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)**)** |
| [UTF-8](https://ru.wikipedia.org/wiki/UTF-8) | EF BB BF |
| [UTF-16](https://ru.wikipedia.org/wiki/UTF-16) ([BE](https://ru.wikipedia.org/w/index.php?title=Big_Endian&action=edit&redlink=1)) | FE FF |
| [UTF-16](https://ru.wikipedia.org/wiki/UTF-16) ([LE](https://ru.wikipedia.org/w/index.php?title=Little_Endian&action=edit&redlink=1)) | FF FE |
| [UTF-32](https://ru.wikipedia.org/wiki/UTF-32) (BE) | 00 00 FE FF |
| [UTF-32](https://ru.wikipedia.org/wiki/UTF-32) (LE) | FF FE 00 00 |

1. Структура языка программирования: идентификаторы, зарезервированные идентификаторы, литералы, ключевые слова. Определения, примеры (C++).

**Идентификаторы:** имя компонента программы (переменной, функции, метки, типа и пр.), составленное программистом по определенным правилам, чувствительны к регистру

**С++:** a, b, c, i …

**Зарезервированные идентификаторы:** идентификаторы, которые предварительно определены в системе программирования.

**С++**: все имена с двумя подчеркиваниями считаются зарезервированными. Кроме того: **is**xxxx, **mem**xxxx, **str**xxxx, **t**oxxxx, **wcs**xxxx, **Eцифра**xxxx, **LC\_X**xxx, **SIGX**xxx, **SIG\_X**xxxx.

**Литералы:** запись в исходном коде программы, представляющая собой фиксированное значение.

В **C++** существует четыре типа литералов:

* целочисленный литерал: 2
* вещественный литерал: 3.0
* символьный литерал: ‘a’
* строковый литерал: “abc”

**Ключевые слова:** последовательности символов алфавита языка, имеющие специальное назначение, зарезервированные компилятором для обозначения типов переменных, класса хранения, элементов операторов.

**С++:** int, void, for, if …

1. Структура языка программирования: фундаментальные типы данных. Определение, примеры (C++). Строки. Массивы данных фундаментального типа. Преобразование типов.

**Фундаментальные типы** (встроенные типы) являются базовыми типами на которых строятся производные типы и типы определённые пользователем.

**Примеры**: Для описания основных типов определены следующие ключевые слова:

* int (целый); (4 байта)
* char (символьный); (1 байт)
* wchar\_t (расширенный символьный); (2 байта)
* bool (логический); (1 байт)
* float (вещественный); (4)
* double (вещественный с двойной точностью). (8)

Существует четыре модификатора основных типов, уточняющих внутреннее представление и диапазон значений стандартных типов:

* short (короткий); (2 байта)
* long (длинный); (4 байта)
* signed (знаковый);
* unsigned (беззнаковый).

— Указатель (pointer) — это переменная, в которой хранится адрес другого объекта (как правило, другой переменной).

— \*void: В C++ существует специальный тип указателя, который называется указателем на неопределённый тип.

Указатель на void применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов)

Указателю на void можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом.

— Ссылка (ссылочный тип) - это задание альтернативного имени объекта. Всякое изменение ссылки преобразует не ее, а тот объект, на который она ссылается.

— Указатель на функцию - переменная, которая содержит адрес некоторой функции.

**Строки:** В С++ поддерживаются два типа строк – встроенный тип и класс string из стандартной библиотеки С++.

Встроенный строковый тип перешел к С++ по наследству от С. Строка символов хранится в памяти как массив, и доступ к ней осуществляется при помощи указателя типа char\*.

Класс string (<string>):

минимальный набор операций, которыми должен обладать класс string:

* инициализация массивом символов (строкой встроенного типа) или другим объектом типа string. Встроенный тип не обладает второй возможностью;
* копирование одной строки в другую. Для встроенного типа приходится использовать функцию strcpy();
* доступ к отдельным символам строки для чтения и записи. Во встроенном массиве для этого применяется операция взятия индекса или косвенная адресация;
* сравнение двух строк на равенство. Для встроенного типа используется функция strcmp();
* конкатенация двух строк, получая результат либо как третью строку, либо вместо одной из исходных. Для встроенного типа применяется функция strcat(), однако чтобы получить результат в новой строке, необходимо последовательно задействовать функции strcpy() и strcat();
* вычисление длины строки. Узнать длину строки встроенного типа можно с помощью функции strlen();
* возможность узнать, пуста ли строка. У встроенных строк для этой цели приходится проверять два условия

Класс string стандартной библиотеки С++ реализует все перечисленные операции.

**Массивы данных фундаментальных типов:** коллекция однородных данных, размещенных последовательно в памяти и допускающие доступ по индексу (смещение = индекс\*sizeof(базовый\_тип)).

Массив(array) — это совокупность переменных, имеющих одинаковый тип и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса.

Многомерные массивы: Доступ к элементу, стоящему на пересечении первой строки и третьего столбца, можно получить двумя способами: либо индексируя массив — mm[0][2], либо используя указатель — \*((int \*)mm+2), где int – это базовый тип массива.

Для двухмерного массива справедлива следующая формула:

mm [j] [к] =\* ((базовый\_тип \*) mm + (i \* длина строки) + к)

Правила адресной арифметики требуют приведения типа указателя на массив к его базовому типу.

**Преобразование типов**  — преобразование значения одного типа в значение другого типа.

Выделяют приведения типов:

* явные
* неявные.

Явное преобразование задаётся программистом в тексте программы.

Явное преобразованиечасто применяется там, где это можно сделать неявно для того, чтобы указать, что преобразование делается осознано.

Механизм неявных преобразований может быть отключён посредством явного указания в тексте программы требуемого преобразования типов.

Неявное преобразование выполняется транслятором (компилятором или интерпретатором) по правилами, описанным в стандарте языка.

1. Структура языка программирования: пользовательские типы данных. Определение, примеры (C++). Массивы данных пользовательского типа. Перегрузка операторов для пользовательских типов.

**Пользовательские типы данных:** типы, создаваемые пользователем, на основе того, что доступно в языке; всегда есть объявление типа.

**С++: struct** addres{ //**структура** - это набор переменных, объединенных общим именем. Она обеспечивает удобный способ организации взаимосвязанных данных.

string street;

int number\_of\_house;

};

**struct** rgb\_color{ //**битовое поле** – особый вид полей структуры, используемый для плотной упаковки данных.

unsigned red\_value: 3;

unsigned green\_value: 3;

unsigned blue\_value: 3;

};

**union** bit{ //**объединение** - это пользовательская переменная, которая может хранить объекты различного типа и размера, для их размещения выделяется одна общая память, размерность определяется размерностью максимального элемента объединения.

char ch;

int i;

};

**enum** coin{ //**перечисление** - набор именованных целых констант

penny, nickel, dime

};

**typedef** unsigned int nomer; // определяет новое имя типа данных, новый тип при этом не создается, уже существующий тип получает новое имя

***Массивы данных пользовательского типа:***  это совокупность данных (переменных) пользовательского типа и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса.

* **Массив struct:**

struct{

int i;

char ch;

} S[5];

Обращение: S[0].i;

* **Массив union:**

union{

float f;

int i;

} U[5];

Обращение: U[1].f;

***Перегрузка операторов для пользовательских типов:***

(возможность перегрузки есть только для пользовательских типов/классов) реализация в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются

const Integer operator-(const Integer& i) { return Integer(-i.value); }

1. Структура языка программирования: область видимости переменных, пространства имен, псевдонимы пространства имен

**Область видимости переменных в C++**: доступность переменных по их идентификатору в разных частях (блоках программы).

Переменная должна быть объявлена до ее использования; переменная объявленная во внутреннем блоке (локальная переменная {…}) не доступна во внешнем; переменная объявленная во внешнем блоке доступна во внутреннем; во внутреннем блоке переменная может быть переобъявлена.

**Пространство имен:** именованная область видимости. Применяется для разрешения конфликтов имен. Имена, объявленные в одном пространстве имен, не будут вступать в конфликт с аналогичными именами, объявленными в другой области.

Примеры**:** namespace, using, псевдонимы пространства имен.

**Псевдоним пространства имен**: Имена пространств имен должны быть уникальными, из-за чего зачастую они получаются не слишком короткими. Если длинное имя оказывается трудночитаемым или его сложно вводить в файле заголовка, где нельзя использовать директиву using, можно создать псевдоним пространства имен, который будет служить в качестве сокращения фактического имени.

namespace AReallyLongName {

namespace AnotherReallyLongName {

int foo();

int bar();

void baz(int x, int y);

}

}

void qux() {

namespace N = AReallyLongName::AnotherReallyLongName;

N::baz(N::foo(), N::bar());

}

Пространства имен помогают предотвратить конфликты имен, но не устранить их полностью. Такой конфликт может произойти, когда одно и то же имя объявляется в двух разных пространствах имен и затем предпринимается попытка сделать видимым оба пространства. В таком случае для указания предполагаемого пространства имен явным образом можно воспользоваться описателем псевдонима пространства имен «::».

1. Структура языка программирования: инициализация переменных, присвоение значений, выражения (lvalue, rvalue, побочные эффекты, точка последовательности, унарные, бинарные и тернарные выражения), константные выражения.

**Инициализация переменных:** присвоение значения в момент объявления переменной. Отличие от присвоения: при присвоении явно перемещаются данные.

**Функциональный вид инициализации:**

int k(5\*25);

char ch(‘a’);

**Присвоение значений:**присваивание стирает старое значение из переменной, затем записывает в нее новое значение

**Выражения:** объединение литералов, имен (переменных, функций и пр.), операторов и специальных символов, служащих для вычисления выражения или достижения побочных эффектов

**lvalue:** именующее выражение -это ссылка на значение – могут использоваться в левой и правой части оператора присваивания (имя переменной, ссылка на элемент массива по индексу, вызов функции возвращающей указатель, всегда связаны с областью памяти, адрес которой известен)

**Пример:** int n; n = 3; выражение n – lvalue, ссылается на объект int

**rvalue**: значащее выражение – могут использоваться только в правой части оператора присваивания (не связано с адресом, а только со значением, литералы, вызов функции, возвращающей значение)

**Пример:** литералы

**Побочные эффекты:** любые действия**,** изменяющие состояние среды выполнения; если при вычислении выражения значение переменной, входящей в это выражение, изменился, то говорят, что произошел побочный эффект.

**Точка последовательности:** момент времени, когда побочные эффекты от вычисленных выражений осуществились, а побочные эффекты следующих в последовательности выражений не начались.

**С++**: точка последовательности обозначается символом точка с запятой (;)

Величины, над которыми выполняются операции, называются **операндами**. В зависимости от количества операндов операции могут быть унарными (один операнд) и бинарными (два операнда).

* **Унарные:** выражения с одним операндом
* **Бинарные:** выражения с двумя операндами
* **Тернарные:** выражения, возвращающие свой второй или третий операндов зависимости от значения логического выражения, заданного первым операндом

**Константные выражения:** выражение, которое должно быть вычислено на этапе компиляции.

**Пример:** const int k = 5;

1. Структура языка программирования: инструкции языка программирования, объявление, простые и составные инструкции, инструкции выбора, инструкции циклов, инструкции переходов, инструкции обработки исключений.

**Инструкция (оператор)** — наименьшая автономная часть языка программирования; команда или набор команд. Программа обычно представляет собой последовательность инструкций.

**Объявление** включает в себя указание идентификатора, типа, а также других аспектов элементов языка, например, переменных и функций. Объявление используется, чтобы уведомить компилятор о существовании элемента. (Определение – реализация, при этом компилятор выделяет место. Переменная может быть объявлена несколько раз, а определена только один раз. Объявление предоставляет основные свойства символа: его тип и его название. Определение предоставляет все детали этого символа — если это функция, что она делает; если это класс, какие у него поля и методы; если это переменная, где эта переменная находится. Инициализация – придание начального значения переменной при её объявлении)

**Простые инструкции** завершаются точкой с запятой. **Составные инструкции**, или блоки ({ }), состоят из одной или более инструкций (каждая из которых сама может быть составной), заключенных в фигурные скобки.

**Инструкции выбора** *if*и *switch*, позволяют выполнять те или иные участки кода в зависимости от выполнения условий.

**Инструкции циклов** используются для многократного повторения фрагментов кода. Цикл *for* мы можем использовать, если знаем точное количество действий (итераций). Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл *while* или *do…while*. Цикл *do…while* почти ничем не отличается от цикла *while*, за исключением того, что тело цикла гарантированно выполняется хотя бы один раз.

**Инструкции переходов** выполняют немедленную локальную передачу контроля. Оператор *break* завершает выполнение ближайшего внешнего цикла или условного оператора, в котором он находится. Управление передается оператору, который расположен после оператора, при его наличии. Инструкцию *continue* можно располагать только внутри цикла. Она вызывает переход к следующей итерации самого внутреннего содержащего ее цикла. С помощью *return* функция возвращает управление в программу, откуда была вызвана. Если за *return* следует выражение, то его значение возвращается вызвавшей эту функцию программе. Оператор *goto* осуществляет безусловную передачу управления оператору, метка которого задана идентификатором. Метка оператора имеет значение только для оператора *goto*; в остальных случаях метки операторов игнорируются. Повторное объявление меток невозможно.

**Инструкции обработки исключений** используют три ключевых слова: *try*, *catch* и *throw*. Те инструкции программы, где ожидается возможность появления исключительных ситуаций, содержатся в бло­ке *try*. Если в блоке *try* возникает исключение, т. е. ошибка, то генерируется исключение throw. Исклю­чение перехватывается, используя *catch*, и обрабатывается.

1. Структура языка программирования: программные конструкции (блоки, функции, процедуры и пр.). Передача параметров. Реализация программных конструкций в C++. Передача параметров по значению и по ссылке, передача переменного числа параметров в функции C++. Перегружаемые функции в C++. Inline-функции в C++. Соглашения о вызовах (\_cdecl, \_fastcall, \_stdcall) в С++. Шаблоны функций С++.

**Программные конструкции:**

**Блоки**: логически сгруппированный набор идущих подряд инструкций в [исходном коде](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) программы, является основой парадигмы [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%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)

**Пример С++:**

{

int a = 1;

int b = 2;

int c = a + b;

return c;

}

**Функции:** фрагмент программного кода, к которому можно обратиться из другого места программы

**Пример С++:**

void name(char\* text)

{

std::cout<<text<< std::endl;

}

**Процедуры:** функции не возвращающие значения, return не обязателен, но желателен для явного указания завершения функции

**Пример С++:**

 void ValidMessage()

 {

         cout << "Пароль введен верно" << endl;

 }

**Передача параметров:** передача параметров в функцию происходит через стек. Код, вызывающий функцию, знает, сколько параметров ей передать и каковы значения этих параметров

**Передача параметров**

**по значению:** они при выходе из функции не изменятся

**Пример С++:**

int func(int k){

k\*=2;

return k;

}

void main(){

int z=1, y=3, k;

k=func(z)+func(y);

cout<<z<<" "<<y;

}

Z = 1, y = 3

**по ссылке:** при выходе из функции из значения могут измениться

int func(int &k){

k\*=2;

return k;

}

void main(){

int z=1, y=3, k;

k=func(z)+func(y);

cout<<z<<" "<<y;

}

Z = 2, y = 6

**Передача переменного числа параметров в функции С++:**

По умолчанию параметры передаются в функцию через стек. Поэтому, технически, нет ограничения на количество передаваемых параметров. Функции с переменным числом параметров объявляются как обычные функции, но вместо недостающих аргументов ставится многоточие. Количество параметров и их типы известны только при вызове функции.

Передать функции параметры можно двумя способами:

* явно передать обязательный аргумент, задающий число параметров;
* добавить в конец списка параметр с уникальным значением, по которому будет определяться конец списка параметров.

Общий принцип работы следующий: в функции для доступа к списку параметров устанавливается указатель, значением которого будет адрес явного параметра в списке, далее изменяется значение этого указателя, чтобы переместиться на следующий параметр.

**unsigned summ(unsigned char num, unsigned first, ...) {**

**unsigned sum = 0;**

**unsigned testsum = 0;**

**unsigned \*p = &first;**

**}**

**Перегружаемые функции С++:** возможность использования одноимённых подпрограмм: процедур или функций в языках программирования. Под перегрузкой функции понимается, определение нескольких функций (две или больше) с одинаковым именем в одной области видимости, но с различными параметрами. Наборы параметров перегруженных функций могут отличаться порядком следования, количеством, типом. Говорят, что есть две функции, с разной сигнатурой, но одинаковыми именами (перегруженные функции). Сигнатура – это комбинация имени функции с её параметрами. Компилятор самостоятельно выберет нужную функцию, анализируя только лишь сигнатуры перегруженных функций.

**Inline-функции в С++:** inline – подсказка компилятору для выполнения подстановки, компилятор может не выполнить подстановку, если не установить параметры.

Встроенные функции или inline- функции определяются с помощью ключевого слова inline. Каждый раз, вместо вызова этой функции, компилятор будет заменять его фактическим кодом функции. Вызовы функций занимают больше времени, чем выполнение всего последовательного кода. При такой замене увеличивается размер программы.

**Пример С++:**

#include <iostream>

using namespace std;

inline void hello()

{

  cout<<"hello";

}

int main()

{

  hello();

  cin.get();

}

**Соглашения о вызовах в С++**

**\_cdecl:** Параметры функций помещаются в стек, порядок параметров "справа налево". Параметры, размер которых меньше 4-х байт, расширяются до 4-х байт. Адрес возврата кладется в стек поверх параметров.

Стек освобождается вызывающим кодом: после команды CALL следует инструкция, которая прибавляет к регистру ESP нужное значение (суммарный размер в байтах всех аргументов). Возвращаемый параметр передается через регистр EAX

**\_fastcall:** Параметры передаются через регистры. Если для сохранения всех параметров регистров не достаточно, используется стек; порядок параметров "справа налево". Очистку стека производит вызываемый код.

В компиляторе фирмы Borland параметры передаются через регистры слева направо, если параметров больше трёх — в стеке.

**\_stdcall:** Параметры помещаются в стек, порядок параметров "справа налево". Адрес возврата кладется в стек поверх параметров.

Стек освобождает вызываемый код. В последней инструкции вызываемого кода RET указывается суммарный размер в байтах всех параметров процедуры. Команда RET после извлечения адреса возврата прибавляет к регистру ESP указанное значение. Целостность стека восстанавливается вызываемым кодом.

Возвращаемый параметр передается через регистр EAX.

**Шаблоны функций С++:** Шаблоны функций определяют семейство функций. С помощью шаблонов функций можно задавать наборы функций, основанных на одном коде, но действующих для разных типов. Шаблон функции начинается с ключевого слова template, за которым в угловых скобках следует список параметров. Затем следует объявление функции.

**Пример С++:**

template<typename T>

T max(T x, T y)

{

return (x > y ? x : y);

}

void main()

{

int r = max<int>(3, 5);

float fr = max<float>(3.2f, 5.2f);

}

1. Структура языка программирования: стандартная библиотека. Реализация стандартной библиотеки STL в C++. Понятие контейнера, итератора и алгоритма. Стандартные функции STL для работы со строками: копирования, сравнения, вычисления длины, поиска символа и подстроки. Стандартные функции STL для работы со wc-строками: копирования, сравнения, вычисления длины, поиска символа и подстроки. Стандартные функции для работы с потоками данных. Стандартные функции для получения текущей даты и даты и времени.

**Стандартная библиотека шаблонов** (**STL**– «**S**tandard **T**emplate **L**ibrary») — это часть Стандартной библиотеки С++, которая содержит набор шаблонов контейнерных классов (например, [**std::vector**](https://ravesli.com/urok-95-vvedenie-v-std-vector-vektory/) и [**std::array**](https://ravesli.com/urok-94-vvedenie-v-std-array/)), алгоритмов и итераторов.

**Контейнер** –объект, содержащий другие объекты (стек, список, очередь, вектор и пр.), предназначенный для хранения однотипных объектов и обеспечения доступа к ним. Контейнеры: последовательные (последовательный доступ к элементам), ассоциативные (доступ по ключу).

**Итератор:** объект, обеспечивающий для алгоритма средство доступа к содержимому контейнера. Итераторы: произвольного доступа (для ассоциативных контейнеров), двунаправленные (для последовательных).

std::vector<int> v = { 1,2,3,4 };

std::vector<int>::iterator iter = v.begin();  // получаем итератор

Для получения итераторов контейнеры в C++ обладают такими функциями, как **begin()** и **end()**

* **\*iter**: получение элемента, на который указывает итератор
* **++iter**: перемещение итератора вперед для обращения к следующему элементу
* **--iter**: перемещение итератора назад для обращения к предыдущему элементу. Итераторы контейнера **forward\_list** не поддерживают операцию декремента.
* **iter1 == iter2**: два итератора равны, если они указывают на один и тот же элемент
* **iter1 != iter2**: два итератора не равны, если они указывают на разные элементы

**Алгоритм** определяет вычислительную процедуру (обобщённые алгоритмы для работы с контейнерами).

**Стандартные функции STL для работы со строками:**

strcpy — Копирует содержимое одной строки в другую

strcmp — Сравнивает в лексикографическом порядке две строки (если строки равны, возвращается ноль, иначе значение меньше или больше нуля)

strlen — Возвращает длину строки с завершающим нулевым символом

strchr – поиск первого вхождения символа в строку.

strstr — Возвращает указатель на первое вхождение одной строки в другую

**Стандартные функции STL для работы со wc-строками:**

wcscpy — Копирует содержимое одной строки в другую

wcscmp — Сравнивает в лексикографическом порядке две строки (если строки равны, возвращается ноль, иначе значение меньше или больше нуля)

wcslen — Возвращает длину строки с завершающим нулевым символом

wcschr – поиск первого вхождения символа в строку.

wcsstr — Возвращает указатель на первое вхождение одной строки в другую

**Стандартные функции для работы с потоками:**

**Потоки вывода**

Функции open, close и is\_open

flush очищает буфер

put выводит символ в файл

seekp устанавливает, а функция tellp читает позицию в файле для функции put

write пишет последовательность символов (байтов) в файл

str выполняет запись и чтение строки в поток

**Потоки ввода**

gcount возвращает количество символов, прочитанных последней не форматирующей операцией ввода

get вводит символ из входного потока

getline вводит строки из файла

ignore удаляет символы из буфера потока

peek читает символ, но не удаляет его из буфера

putback возвращает символ в буфер

read читает блок данных из файла

seekg устанавливает указатель позиции файла для метода get, а функция tellg получает указатель позиции файла для метода get

sync     синхронизирует буфер потока с внешним устройством

unget возвращает в поток символ, предварительно извлеченный из потока

**Стандартные функции для получения текущей даты и даты и времени**:

Функции библиотеки ctime

**Функции управления временем**

clock Функция возвращает количество тактов процессора, прошедших с момента запуска программы.

time Получить текущее время.

**Функции преобразования**

asctime Преобразование значения времени в строку (формат «день\_недели месяц день\_месяца часы:минуты:секунды год»).

ctime Преобразование значения типа данных времени time\_t в строку.

localtime Принимает значение текущего времени в секундах и инициализирует структуру в местном формате времени.

strftime Преобразование времени в символьную строку(для создания своего формата строки времени).

1. Классы памяти: код, стек, статические данные, динамическая область. Структура динамической памяти (Heap) C++. Фрагментация динамической памяти C++.

**Код -** Текст программы. Обычно различают исходный код (на языке программирования) и исполняемый (машинный) код - на языке машинных команд микропроцессора.

**стек –** структура данных с методом доступа к элементам LIFO (англ. Last In — First Out, «последним пришёл — первым вышел»).  
Наиболее распространены два принципа реализации стека: на базе массива и на базе связанного списка. В первом случае для хранения значений в памяти отводится сплошной массив ячеек, которые используются по мере необходимости. Во втором — для каждого элемента стека заказывается блок памяти, достаточный для хранения значения и ссылок на предыдущий и следующий элементы стека. Реализация на базе массива проще, эффективнее и экономичнее по расходу памяти, но она требует заранее знать предельный размер стека и может приводить к трудно выявляемым ошибкам. Реализация на базе списков более надежна, но менее эффективна.

* **локальные переменные – static статические данные – глобальные - extern**
* **динамическая область – Динами́ческая переме́нная** — [переменная](https://dic.academic.ru/dic.nsf/ruwiki/4894) в [программе](https://dic.academic.ru/dic.nsf/ruwiki/6823), место в [оперативной памяти](https://dic.academic.ru/dic.nsf/ruwiki/8491) под которую выделяется во время выполнения программы. По сути, она является даже не переменной, а участком памяти, выделенным [системой](https://dic.academic.ru/dic.nsf/ruwiki/2566) программе для конкретных целей
* **куча**  — это хранилище памяти, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. По завершении приложения все выделенные участки памяти освобождаются. Размер кучи задаётся при запуске приложения и это позволяет создавать динамические переменные.

**Кучи** обычно реализуются в виде массивов, что исключает наличие указателей между её элементами.

Над кучами обычно проводятся следующие операции:

* *найти максимум* или *найти минимум*: найти максимальный элемент в max-куче или минимальный элемент в min-куче, соответственно
* *удалить максимум* или *удалить минимум*: удалить корневой узел в max- или min-куче, соответственно
* *увеличить ключ* или *уменьшить ключ*: обновить ключ в max- или min-куче, соответственно
* *добавить*: добавление нового ключа в кучу.
* *слияние*: соединение двух куч с целью создания новой кучи, содержащей все элементы обеих исходных.

**Фрагментация памяти** - это когда большая часть вашей памяти распределяется в большом количестве несмежных блоков или фрагментов, оставляя хороший процент вашей общей памяти нераспределенной, но непригодной для большинства типичных сценариев. Это приводит к исключениям из памяти или ошибкам размещения (т.е. Malloc возвращает null).

1. Механизм исключений: определение, назначение, применение. Реализация исключений в C++.

**Механизм исключений** - механизм, предназначенный для описания реакции программы на ошибки (*исключения*), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Обработка исключений позволяет упорядочить обработку ошибок. Используя обработку исключений, программа может автоматически вызвать функцию-обработчик ошибок тогда, когда такая ошибка возникает. Принципиальным достоин­ством обработки исключений служит то, что она позволяет автоматизировать большую часть кода для обработки ошибок, для чего раньше требовалось ручное кодирование.

**Реализация исключений в C++**:

Обработка исключений в С++ использует три ключевых слова: *try*, *catch* и *throw*. Те инструкции программы, где ожидается возможность появления исключительных ситуаций, содержатся в бло­ке *try*. Если в блоке *try* возникает исключение, т. е. ошибка, то генерируется исключение *throw*. Исклю­чение перехватывается, используя *catch*, и обрабатывается.

Инструкция, генерирующая исключение, должна исполняться внутри блока *try*. Вызванные из блока *try* функции также могут генерировать исключения. Всякое исключение должно быть пере­хвачено инструкцией *catch*, которая непосредственно следует за инструкцией *try*, сгенерировав­шей исключение.

Когда исключение сгенерировано, оно перехватывается соответствующей инструкцией *catch*, обрабатывающей это исключение. Одному блоку *try* может отвечать несколько инструкций *catch*, Какая именно инструкция *catch* исполняется, зависит от типа исключения. Это означает, что если тип данных, указанных в инструкции *catch*, соответствует типу данных исключения, то только эта инструкция *catch* и будет исполнена. Когда исключение перехвачено, arg получает ее значение. Перехваченным может быть любой тип данных, включая созданные программистом классы. Если никакого исключения не сгенерировано, то есть никакой ошибки не возникло в блоке *try*, то инструкции *catch* выполняться не будут.

Общая форма записи инструкции *throw* имеет вид:

*throw исключение;*

Инструкция *throw* должна выполняться либо внутри блока *try*, либо в функции, вызванной из блока *try*. В записанном выше выражении исключение обозначает сгенерированное значение.

1. Препроцессор: определение, назначение, применение, директивы, выражения, макросы. Примеры на C++.

**Препроцессор:**

**Определение:** часть транслятора, которая выполняется до процесса трансляции; выполняет директивы препроцессора. Результатом выполнения препроцессора является текст, сформированный из исходного под управлением директив препроцессора.

**Назначение:** программа**,** подготавливающая код программы на языке [C](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%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))/[C++](https://ru.wikipedia.org/wiki/C%2B%2B) к [компиляции](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80); включает содержимое одних файлов в другие, заменяет в тексте исходного кода имена констант на их значения, удаляет символы конца строки

**Применение:**  Наиболее частый случай использования препроцессора — обработка исходного кода перед передачей его на следующий шаг компиляции.

**Директивы:**  строка в исходном коде, имеющая следующий формат: #ключевое\_слово параметры

**Примеры С++:**

include, #define, #if, #else, #elif, #endif, #ifdef, #ifndef, #error, #line, #pragma, #undef;

**#include:** препроцессор включает все содержимое указанного файла в исходный код программы

**#define:** объявляются и определяются символические константы; когда перед компиляцией исходный код будет обработан препроцессором, то все символьные константы (в примере это N и HELLO) в тексте исходного кода на языке C будут заменены на соответствующие им числовые или строковые константы

**Выражения:** Могут появляться в директивах #if и #elif. В выражениях препроцессора разрешены операции >, <, !, ==, !=, &&, ||. Для группирования могут использоваться скобки

Примеры С++:

#if MAX>99  
…  
#endif

#if defined(MAX) || defined(MIN)

…

#endif

**Макросы:** символьное имя, заменяемое при обработке препроцессором на последовательность программных инструкций

**Примеры C++:**

#define MACRO1(x) x \* x

1. Статическая библиотека: определение, назначение, применение. Реализация статической библиотеки в Windows C++. Утилита LIB. Компоновка статической библиотеки. Применение директивы препроцессора #pragma comment для компоновки статической библиотеки.

**Статическая библиотека:** файл (обычно с расширением **lib**), содержащий объектные модули. Является входным файлом для компоновщика (**linker**).

Библиотеки позволяют использовать разработанный ранее программный код в различных программах. Таким образом, программист может не разрабатывать часть кода для своей программы, а воспользоваться тем, что входит в состав библиотек. Обычно код библиотек отличается качеством, позволяет писать более ясный код, понятный большинству программистов.

В языке программирования C код библиотек представляет собой функции, размещенные в файлах, которые скомпилированы в объектные файлы, а те, в свою очередь, объединены в библиотеки. В одной библиотеке объединяются функции, решающие определенный тип задач.

**Реализация**: Проект типа «статическая библиотека». У каждой библиотеки должен быть свой заголовочный файл, в котором должны быть описаны прототипы (объявления) всех функций, содержащихся в этой библиотеке. С помощью заголовочных файлов вы "сообщаете" вашему программному коду, какие библиотечные функции есть и как их использовать.

**Утилита LIB**: Для работы с библиотекой предназначена утилита LIB.

## [Параметры LIB](javascript:void(0)):

## Построение библиотеки импорта и файла экспорта:

**/DEF** Создание библиотеки импорта и файла экспорта

**/EXPORT** Экспорт функции из программы.

**/INCLUDE** Добавление символа в таблицу символов.

**Запуск программы LIB:**

**/ERRORREPORT** Передача Майкрософт сведений о внутренних ошибках с помощью lib.exe.

**/LTCG** Инициирует построение библиотеки с помощью создания кода времени компоновки.

**/MACHINE** Задание целевой платформы для программы.

**/NOLOGO** Отключает вывод программой LIB уведомления об авторских правах и номере версии, а также отображение команд командного файла.

**/VERBOSE** Отображает подробные сведения о ходе сеанса, включая имена добавляемых OBJ-файлов.

**/WX** Обработка предупреждений, как ошибок.

**Извлечение члена библиотеки:**

**/EXTRACT** Создание объектного файла (OBJ-файла), содержащего копию элемента существующей библиотеки.

**Управление библиотекой:**

**/LIBPATH** Переопределяет путь к библиотеке среды.

**/LIST** Отображает информацию о выходной библиотеке в стандартном виде.

**/NAME** При построении библиотеки импорта указывает имя библиотеки DLL, для которой была создана библиотека импорта.

**/NODEFAULTLIB** Удаляет одну или несколько библиотек по умолчанию из списка искомых библиотек при разрешении внешних ссылок.

**/OUT** Переопределяет имя выходного файла используемое по умолчанию.

**/REMOVE** Пропуск объекта из выходной библиотеки.

**/SUBSYSTEM** Сообщает операционной системе способ запуска программы, созданной путем привязки к выходной библиотеке.

**Компоновка:** Создаем проект с исходником и заголовочным файлом -> в свойствах проекта указываем тип конфигурации «статическая библиотека» -> компилим (создается файл с расширением .lib) -> в свойствах своего проекта (компоновщик -> командная строка) в доп параметрах указываем в “ “ путь к файлу .lib

**Компоновка с применением директивы** (**Автоматическое связывание**) (англ. *Auto-linking*) — механизм автоматического определения необходимых библиотек при сборке программ на языках [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_%28%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%29) или [C++](https://ru.wikipedia.org/wiki/C%2B%2B).

Механизм активируется при помощи директивы препроцессора

**#pragma comment(lib, <спецификация-библиотеки>)**.

1. Теория формальных языков: алфавит языка, цепочка, пустая цепочка, длина цепочки, равенство (эквивалентность) цепочек, конкатенация цепочек, итерация цепочки, операторы + и \*, определение формального языка, эквивалентность двух языков, способы задания формального языка. Лексика, синтаксис и семантика языка.

**Алфавит языка:** конечное непустое множество элементов языка

**Пример:** V = {a, b} обозначает алфавит V, состоящий из двух символов a и b

**Цепочка:** конечная последовательность символов языка

**Пример:** abc — цепочка из трех символов

**Пустая цепочка:** цепочка, которая не содержит ни одного символа

**Длина цепочки:** число составляющих ее символов

**Пример:** цепочка abc – длина = 3

**Равенство (эквивалентность) цепочек:** цепочки равны, если они имеют один и тот же состав символов, одно и то же количество символов  и тот же порядок символов

**Конкатенация цепочек:** если  - конкатенация цепочек ,  и , то ,, - подцепочки ,  - префикс ,  - суффикс и 

**Итерация цепочки:** для любой цепочки α справедливы утверждения α0 = ε и αn = αn-1α=ααn-1 для n≥1. αn – называется итерацией цепочки.

**Пример:** ,, .

**Операторы:**

* **+:** Если  - алфавит, то  – множество всех цепочек, состоящих из символов . 
* **\*:** Если  – алфавит, то  – множество всех цепочек, состоящих из символов . . 

**Определение формального языка:**  множество конечных слов над конечным алфавитом.

**Эквивалентность двух языков:** Языки  и  совпадают, если язык  включает язык  и язык  включает язык . .

**Способы задания формального языка:**

Наиболее общим из конструктивных способов задания языков является способ, использующий так называемые *формальные грамматики*. *Формальной грамматикой* для порождения формального языка в алфавите  называется набор:

 где:  - множество терминальных символов,  - множество нетерминальных символов, - множество правил (говорят продукций) грамматики, -начальный символ грамматики.

Альтернативный способ задания формальных языков — их описание с помощью различных видов автоматов.

Регулярные выражения — это аналитический (формульный) способ задания *регулярных языков*.

**Лексика:** множество цепочек языка.

**Синтаксис:** набор формальных правил, определяющий конструкции (последовательности цепочек).

**Семантика:** набор неформальных правил (невозможно записать правила в виде формальных выражений), которые описываются словесно (например, в руководстве программиста). Пример: применению переменной должно предшествовать ее объявление, при конвертации типов следует обеспечить соответствующее значение переменной.

1. Теория формальных языков: грамматика (определение, назначение, примеры), выводимость цепочки символов в грамматике, сентенциальная форма грамматики, язык порождаемый грамматикой, способы задания грамматик. Общая характеристика формы Бэкуса-Наура.

**Формальная грамматика** или просто **грамматика** в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита. Различают *порождающие* и *распознающие* (или *аналитические*) грамматики — первые задают правила, с помощью которых можно построить любое слово языка, а вторые позволяют по данному слову определить, входит ли оно в язык или нет.

**Терминал (терминальный символ)** — объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В формальных языках, используемых на компьютере, в качестве терминалов обычно берут все или часть стандартных символов [ASCII](https://ru.wikipedia.org/wiki/ASCII) — латинские буквы, цифры и спец. символы.

**Нетерминал (нетерминальный символ)** — объект, обозначающий какую-либо *сущность* языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

Словами языка, заданного грамматикой, являются все последовательности терминалов, выводимые (порождаемые) из начального нетерминала по правилам вывода.

Чтобы задать грамматику, требуется задать алфавиты терминалов и нетерминалов, набор правил вывода, а также выделить в множестве нетерминалов начальный.

Итак, грамматика определяется следующими характеристиками:

 – грамматика языка (говорят порождающая грамматика) –

это четверка, где:

 - множество терминальных символов,

 - множество нетерминальных символов,

- множество правил (говорят продукций) грамматики,

-начальный символ грамматики.

 – множество терминальных символов (терминалы) (алфавит языка) – это символы языка, определяемые грамматикой.

 – множество нетерминальных символов (нетерминалы) – символы, применяемые в продукциях .  (символы, определяющие слова, понятия, конструкции языка).

 – множество правил вида , где , а . Символ  читается «порождает» или «по определению есть».  -  порождает  или  по определению есть .

 – стартовый символ, .

*Пример*: ,   
где ,

порождаемый язык 

 - словарь грамматики .

**Выводимость цепочки символов в грамматике**: Цепочка  непосредственно

выводима из цепочки  в грамматике , если , , где ,  и правило вывода содержится в . Записывается .

*Пример*:, , , , , ,, , .

Цепочка выводима из цепочки  в грамматике , если существуют цепочки, такие что . Записывается .

Последовательность  называется выводом длины .

*Пример*: , вывод , длина вывода  из  равна 2. (Запись второго правила рекурсивна).

Запись  предполагает  шагов вывода  из . В том случае, если , то число шагов вывода .

Запись  предполагает  шагов вывода  из .

**Сентенциальной форма грамматики** : Если  и , то  называется **сентенциальной** формой грамматики .

Если  и ,то  называется терминальной сентенциальной формой грамматики .

**Язык, порождаемый грамматикой**:  - язык, порождаемый грамматикой . Язык содержит все терминальные цепочки, выводимые из : .

- множество терминальных сентенциальных форм грамматики .

*Пример*: ,



 - грамматики эквивалентны, если они порождают один язык.

**Способы задания схем грамматик**. Схема грамматики содержит правила вывода, определяющие структуру цепочек порождаемого языка. Для задания правил используются различные формы описания: символическая, форма Бэкуса-Наура, итерационная форма и синтаксические диаграммы.

**Форма Бэкуса-Наура**. Джон Бэкус: 1924-2007, США, один из авторов (руководитель команды) языка ФОРТРАН. Петер Наур: 1928, Дания, один из авторов языка Алгол 60.

Цепочки языка могут содержать метасимволы, имеющие особое назначение. Метаязык, предложенный Бэкусом и Науром (БНФ) использует следующие обозначения:

* символ «::=» отделяет левую часть правила от правой (читается: «определяется как»);
* нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки «<» и «>»;
* терминалы - это символы, используемые в описываемом языке;
* правило может определять порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты «|» (читается: «или».

Для удобства и компактности описаний, в расширенных БНФ вводятся следующие дополнительные конструкции (метасимволы):

* квадратные скобки «[» и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать;
* фигурные скобки «{» и «}» означают повторение заключенной в них синтаксической конструкции ноль или более раз;
* сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один и более раз;
* круглые скобки «(» и «)» используются для ограничения альтернативных конструкций;
* кавычки используются в тех случаях, когда один из метасимволов нужно включить в цепочку обычным образом.

*Пример1*Правила, определяющие понятие «идентификатор» некоторого языка программирования:

<буква> ::= *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z*

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<идентификатор> ::= <буква> { (<буква> | <цифра>) }

*Пример2*: форма Бэкуса-Наура, описывающая целое число

**<целое> := <целое без знака>**

**<целое> := +<целое без знака>**

**<целое> := -<целое без знака>**

**<целое без знака> := <цифра>**

**<целое без знака> := <целое без знака><цифра>**

**<цифра> := 0**

**<цифра> := 1**

**<цифра> := 2**

**<цифра> := 3**

**<цифра> := 4**

**<цифра> := 5**

**<цифра> := 6**

**<цифра> := 7**

**<цифра> := 8**

**<цифра> := 9**

*Пример3*: сокращенная форма Бэкуса-Наура

**<целое> := <целое без знака>|+<целое без знака>**

**|-<целое без знака>**

**<целое без знака> := <цифра>|<целое без знака><цифра>**

**<цифра> := 0|1|2|3|4|5|6|7|8|9**

1. Теория формальных языков: иерархия Хомского (неограниченные, контекстно-зависимые, контекстно-независимые и регулярные грамматики), соотношения грамматик и порождаемых ими языков.

**Иерархия Хомского:**

**Неограниченные:**  - неограниченные грамматики, у которых нет никаких ограничений для правил. Правила имеют вид: , где , .

**Контекстно-зависимые:**  - контекстно-зависимые грамматики (неукорачивающие грамматики). Правила имеют вид: , где ,  и .

**Контекстно-независимые:**  - контекстно-свободные грамматики. Правила имеют вид: , где , .

**Регулярные:**  - регулярные грамматики. Регулярные грамматики бывают праволинейными и леволинейными. У праволинейной грамматики правила имеют вид: или , где , . У леволинейной грамматики правила имеют вид: или , где , .

**Соотношения грамматик:**

* любая регулярная грамматика является контекстно-свободной грамматикой;
* любая контекстно-свободная грамматика является контекстно-зависимой грамматикой;
* любая контекстно-зависимая грамматика является грамматикой типа 0.

**Порождаемые языки:**

* каждый регулярный язык является контекстно-свободным языком, но существуют контекстно-свободные языки, которые не являются регулярными;
* каждый контекстно-свободный язык является контекстно-зависимым, но существуют контекстно-зависимые, которые не являются контекстно-свободными.
* каждый контекстно-зависимый язык является языком типа 0.

1. Лексический анализ: определение, назначение, применение. Лексический анализатор: место в структуре транслятора, входная и выходная информация, способы взаимодействия с синтаксическим анализатором (последовательное и параллельное), нисходящий и восходящий разбор, дерево разбора.

**Лексический анализ** – первая (наиболее простая) фаза трансляции. Лексический анализ выполняется программой (входящей в состав транслятора), называемой лексическим анализатором (сканером).

**Лексический анализ**  — процесс аналитического разбора входной последовательности символов на распознанные группы — ***лексемы***, с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах).

**Лексический анализатор** (или сканер ) — это часть компилятора, которая читает ис- ходную программу и выделяет в ее тексте лексемы входного языка. На вход лексиче- ского анализатора поступает текст исходной программы. Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем. Этот перечень лексем можно представить в виде таблицы, называемой таблицей лексем. Таблица лексем фактически содержит весь текст исходной программы, обработанный лексическим анализатором.

**Структура транслятора**:



**Последовательное взаимодействие лексического и синтаксического анализаторов.**



При последовательном варианте лексический анализатор просматривает весь текст исходной программы от начала до конца и преобразует его в таблицу лек- сем. Таблица лексем заполняется сразу полностью, компилятор использует ее для последующих фаз компиляции. Дальнейшую обработку таблицы лексем выполняют следующие фазы компиляции. Если лексический анализатор не смог правильно определить тип лексемы, то считается, что исходная программа содержит ошибку

**Параллельное взаимодействие лексического и синтаксического анализаторов.**

При параллельном варианте лексический анализ текста исходной программы выполняется поэтапно по шагам. Лексический анализатор выделяет очередную лексему в исходном коде и передает ее синтаксическому анализатору. Синтаксический анализатор может подтвердить правильность найденной лексемы и обратиться к лексическому анализатору за следующей лексемой либо же отвергнуть найденную лексему. Во втором случае он может проинформировать лексический анализатор о том, что надо вернуться назад к уже просмотренному ранее фрагменту исходного кода и сообщить ему дополнительную информацию о том, лексему какого типа следует ожидать. Взаимодействуя между собой таким образом, лексический и синтаксический анализаторы могут перебрать несколько возможных вариантов лексем, и, если ни один из них не подойдет, будет считаться, что исходная программа содержит ошибку. Только после того, как синтаксический анализатор успешно выполнит разбор очередной конструкции исходного языка, лексический анализатор помещает найденные лексемы в таблицу лексем и в таблицу идентификаторов и продолжает разбор дальше в том же порядке.

**Синтаксический анализ** (разбор, парсинг) — процесс сопоставления линейной последовательности лексем (слов, токенов) формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

**Синтаксический анализатор** (парсер) — это программа или часть программы, выполняющая синтаксический анализ. В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки.

**Нисходящий парсинг** — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.

Пример: вывод (распознавание) цепочки **float f;**









Пример: дерево нисходящего разбора цепочки **float f;**



**Восходящий парсинг**— продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.

Пример: восходящий вывод (распознавание) цепочки **float f;**









Пример: дерево восходящего разбора цепочки **float f;**



**Деревом разбора грамматики** (англ. *parse tree*) называется дерево, в вершинах которого записаны терминалы или нетерминалы. Все вершины, помеченные терминалами, являются листьями. Все вершины, помеченные нетерминалами, имеют детей. Дети вершины, в которой записан нетерминал, соответствуют раскрытию нетерминала по одному любому правилу (в левой части которого стоит этот нетерминал) и упорядочены так же, как в правой части этого правила.

1. Регулярные выражения: рекурсивное определение, назначение, применение, метасимволы, примеры. Применение рекурсивных выражений в функциях стандартной библиотеке STL C++.

**Рекурсивное определение:**

*  - регулярное выражение и представляет пустое множество;
*  – регулярное выражение и представляет множество ;
* для каждого  символ  является регулярным выражением и представляет множество ;
* если  - регулярное приложение, представляющее множество , если  - регулярное приложение, представляющее множество , то , ,  являются регулярными выражениями и представляют множества ,  (конкатенация множеств) и  соответственно.
* 

**Назначение:** поиск текста по шаблону и работа со строками

***Применение:***

**Метасимволы:** символы, применяемые для описания регулярных выражений

**Примеры:**

|  |  |
| --- | --- |
| Регулярное выражение | Множество |
| a | a |
| a+b | a,b |
| a+b+c | a,b,c |
| a+ | a,aa, aaa,aaaa... |
| a\* | λ, a, aa, aaa, aaaa,... |
| ab | ab |
| ab+cd | ab, cd |
| (ab+cd)+ | ab,cd,abab,abcd,cdcd,cdab, ababab, cdcdcd, abcdab, cdabcd, abcdcdcdcd,  abababab,... |
| (ab+cd)\* | λ, (ab+cd)+ |
| a(bc+de) | abc, ade |
| a(bc+de)f | abcf, adef |
| ab+c | abc, abbc, abbbc, abbbbc,… |
| ab\*c | ac, abc, abbc, abbbc, abbbbc,… |
| a(bc+de)+f | abcf, adef, abcbcf, abcdef, adebcf, adedef, abcdedcdef,... |
| a(bc+de)\*f | af, a(bc+de)+f |
| (ab+cd)(ef+gh) | abef,abgh, cdef, cdgh |
| (ab+cd)e+ | abe, cde,abee, cdee, abeee, cdeee,... |
| (ab+cd)e\* | ab, cd, (ab+cd)e+ |

***Применение рекурсивных выражений в функциях стандартной библиотеки STL C++***

1. Конечный автомат: определение, назначение, схема работы, примеры. Недетерминированные и детерминированные конечные автоматы. Мгновенное описание конечного автомата. Последовательность мгновенных описаний, диаграмма мгновенных описаний. Граф переходов конечного автомата. Соотношение регулярного языка, регулярной грамматики, регулярного языка и конечного автомата.

**Конечный автомат** — автомат, число возможных внутренних состояний которого конечно.

Существуют различные способы задания алгоритма функционирования конечного автомата. Например, конечный автомат может быть задан в виде упорядоченной пятерки элементов некоторых множеств:

,где

 – конечное множество состояний устройства управления;

 – алфавит входных символов;

 – функция переходов, отображающая  во множество подмножеств : ;

- начальное состояние устройства управления;

- множество заключительных (допускающих) состояний устройства управления.

**Другие способы описания**:

**Граф переходов** — графическое представление множества состояний и функции переходов.

**Таблица переходов** — табличное представление функции *δ*. Обычно в такой таблице каждой строке соответствует одно состояние, а столбцу — один допустимый входной символ. В ячейке на пересечении строки и столбца записывается состояние, в которое должен перейти автомат, если в данном состоянии он считал данный входной символ.

**Назначение**: Конечные автоматы широко используются на практике, например, в синтаксических и лексических анализаторах. Конечный автомат довольно хорошо применим в много шаговых интерфейсах, где следующий шаг зависит от того, какие параметры были выбраны на текущем (предыдущем) шаге.

**Схема работы**: Принято полагать, что конечный автомат начинает работу в начальном состоянии, последовательно считывая по одному символу входного слова (цепочки входных символов). Считанный символ переводит автомат в новое состояние в соответствии с функцией переходов.

Читая входную цепочку символов {\displaystyle x} и делая переходы из состояния в состояние, автомат после прочтения последнего символа входного слова окажется в некотором конечном состоянии. {\displaystyle q'}.

Если это состояние является заключительным, то говорят, что автомат допустил слово

**Пример**: пусть  входная цепочка,

КА , где функция  задана следующей таблицей.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

{\displaystyle x}.

Конечные автоматы подразделяются на детерминированные и недетерминированные.

**Детерминированным конечным автоматом** (ДКА) называется такой автомат, в котором нет дуг с меткой *ε* (предложение, не содержащее ни одного символа), и из любого состояния по любому символу возможен переход не более, чем в одно состояние.

**Недетерминированный конечный автомат** (НКА) является обобщением детерминированного. Недетерминированность автоматов может достигаться двумя способами: либо могут существовать переходы, помеченные пустой цепочкой ε, либо из одного состояния могут выходить несколько переходов, помеченных одним и тем же символом.

**Мгновенное описание КА** – пара , где – текущее состояние КА, – неиспользованная часть входной цепочки.

– начальное мгновенное описание КА,  – анализируемая цепочка.

– допускающее мгновенное описание КА.

**Граф переходов** — графическое представление множества состояний и функции переходов. Представляет собой размеченный ориентированный граф, вершины которого — состояния КА, дуги — переходы из одного состояния в другое, а *метки дуг* — символы, по которым осуществляется переход из одного состояния в другое. Если переход из состояния *q1* в *q2* может быть осуществлен по одному из *нескольких* символов, то все они должны быть надписаны над дугой диаграммы.

**Соотношение регулярного языка, регулярной грамматики, регулярного языка и конечного автомата**:

Конечный автомат может быть однозначно задан своим графом переходов.

Доказаны 4 утверждения:

1. язык является регулярным множеством тогда и только тогда, когда он задан регулярной грамматикой;
2. язык может быть задан регулярной грамматикой (левосторонней или правосторонней) тогда и только тогда, когда язык является регулярным множеством;
3. язык является регулярным множеством тогда и только тогда, когда он задан конечным автоматом;
4. язык распознается с помощью конечного автомата тогда и только тогда, когда он является регулярным множеством.

Другими словами: любой регулярный язык может быть задан регулярной грамматикой, регулярным выражением или конечным автоматом.

Или: любой конечный автомат задает регулярный язык, а значит грамматику или регулярное выражение.

Доказана теорема (А. Ахо, Дж. Хопкрофт, Дж. Ульман):

пусть - регулярное выражение, тогда найдется недетерминированный конечный автомат , допускающий автомат, представленный , и обладающий следующими свойствами:

1) ;

2) ;

3) .

1. Построение графа переходов конечного автомата по регулярному выражению. Примеры построения графов переходов. Алгоритм разбора цепочки символов. Примеры алгоритма разбора. Реализация алгоритма разбора с помощью двух массивов.

**Построение графа переходов конечного автомата по регулярному выражению:**

Метод. Автомат для выражения строится композицией из автоматов, соответствующих подвыражениям. На каждом шаге построения строящийся автомат имеет в точности одно заключительное состояние, в начальное состояние нет переходов из других состояний и нет переходов из заключительного состояния в другие.









**Примеры построения графов переходов:**

****





|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Алгоритм разбора цепочки символов и примеры:**







***Реализация алгоритма разбора с помощью двух массивов***



1. Синтаксический анализ: определение, назначение, применение. Синтаксический анализатор: определение, входные данные, выполняемые задачи, результат работы. Соотношение между лексическим и синтаксическим анализом.

**Синтаксический анализ:** процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой.

Это вторая фаза трансляции, выполняемая после фазы лексического анализа и предназначенная для распознавания синтаксических конструкций и формирования промежуточного кода.

Синтаксический анализ **-** основная фаза трансляции, без нее процесс трансляции не имеет смысл. Все задачи лексического анализа могут быть решены в рамках синтаксического анализа. Т.е. можно создать транслятор без лексического анализатора. Лексический анализ необходим для освобождения алгоритма синтаксического разбора от рутинных алгоритмов.

**Применение**:

Всё что угодно, имеющее «синтаксис» (система правил, определяющих допустимые конструкции языка программирования из букв алфавита), поддается автоматическому анализу.

* Языки программирования — разбор исходного кода языков программирования, в процессе трансляции (компиляции или интерпретации);
* Структурированные данные — данные, языки их описания, оформления и т. д. Например, XML, HTML, CSS,  специализированные конфигурационные файлы и т. п.;
* Построение индекса в поисковой системе;
* SQL-запросы (DSL-язык);
* Математические выражения;
* Регулярные выражения (которые, в свою очередь, могут использоваться для автоматизации лексического анализа);
* Формальные грамматики;
* Лингвистика — естественные языки. Например, машинный перевод и другие генераторы текстов.

**Синтаксический анализатор:** частькомпилятора, выполняющая синтаксический анализ.

Входные данные: Входом для синтаксического анализа является таблица лексем (токенов) и таблица идентификаторов.

Выходные данные: дерево разбора.

Входная и выходная информация синтаксического анализатора.



**Задачи,** выполняемые синтаксическим анализатором:

1. поиск и выделение синтаксических конструкций в исходном тексте (разбор);
2. распознавание (проверка правильности) синтаксических конструкций;
3. выявление ошибок и продолжение процесса распознавания после обработки ошибок;
4. формирование дерева разбора в случае, если нет ошибок.

Исходный текст для синтаксического анализатора ⎯ таблица лексем (токенов).

**Соотношение между лексическим и синтаксическим анализом**: Нет четкой границы между лексическим и синтаксическим анализатором. Алгоритм разбора распределяется между лексическим и синтаксическим анализатором. Конструкции языка, которые разбираются лексическим или синтаксическим анализатором, определяет разработчик транслятора.

1. Приведение контекстно-свободной грамматики: удаление бесплодных символов, недостижимых символов, λ-правил, цепных правил.

**Удаление:**

**Бесплодных символов:** нетерминальный символ называется бесплодным, если из него нельзя вывести ни одной цепочки

Алгоритм удаления бесплодных символов в общем виде:



i=1, 

если , то i= i+1 b переход на шаг 2

иначе , где  - правила из , содержащие только символы 

**Недостижимых символов:** недостижимым символом называется символ, не достижимый ни в одном выводе из стартового символа грамматики

Алгоритм удаления недостижимых символов (ВНИМАНИЕ! перед удалением недостижимых символов, должны быть удалены бесплодные символы): строим множества :

i=1, 

, где 

если , то i=i+1 и переход на шаг 2

иначе , где , - правила из , содержащие только символы .

**λ-правил:** Правило вида  называется -правилом

Теорема: для грамматики , содержащей -правила, всегда можно найти эквивалентную грамматику  не содержащую -правил.

Алгоритм исключения -правил: выполнить все возможные подстановки пустой цепочки вместо аннулирующего нетерминала во все правила грамматики.

**Цепных правил:** Правило ,где  называется цепным

Алгоритм исключения цепных правил:

 - множество правил грамматики , Разобьем множество  на два множества  и  по следующему принципу:  содержит правила вида , в - все остальные.

На множестве  построим множество правил  для которых ,  правила из . Замещаем правила  в  на правила .

Правила грамматики  являются правилами грамматики , не содержащей цепных правил.

1. Автомат с магазинной памятью: определение, схема работы, пример, конфигурация (текущее состояние автомата).

**Определение:** Автомат с магазинной памятью (МП-автоматы): распознаватели контекстно-свободных языков, которые можно представить в виде следующей схемы:



Формальное описание МП-автомата:



 ⎯ множество состояний управляющего устройства;

 ⎯ алфавит входных символов;

 ⎯ специальный алфавит магазинных символов;

 ⎯ функция переходов автомата ,

где  - множество подмножеств ;

 ⎯ начальное состояние автомата;

 ⎯ начальное состояние магазина (маркер дна);

 ⎯ множество конечных состояний.

Функция переходов  отображает тройки  в пары  для детерминированного автомата или во множество таких пар для недерминорованного автомата где  ⎯ символ в вершине магазина. Эта функция описывает состояние магазинного автомата, при чтении символа с входной ленты и перемещении головки.

**Конфигурация (текущее состояние автомата)** описывается тройкой , где

- текущее состояние автомата,

 - остаток цепочки. Первый символ этой цепочки просматривается входной головкой автомата. Если = {e}, то входной символ прочитан,

 - цепочка-содержимое магазина (стека). Если = {e}, то магазин пустой.

Один такт работы автомата:

, (читается как «переходит в конфигурацию»)

если .

При выполнении такта (перехода) из стека удаляется верхний символ, соответствующий условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится вершиной стека. Допускаются переходы, при которых входной символ игнорируется. Эти переходы (такты) называются λ-переходами.

Начальное состояние , где  ⎯ начальное состояние автомата,  ⎯ входная цепочка,  ⎯ маркер дна магазина.

Цепочка  является допустимой (распознается) автоматом, если из начальной конфигурации за конечное число таков работы автомат перейдет в заключительное состояние:

, если  и .

**Схема работы** автомата 

1. состояние автомата 
2. читает символ  находящийся под головкой (сдвигает ленту);
3. не читает ничего (читает , не сдвигает ленту);
4. из  определяет новое состояние , если  или .
5. читает верхний (в стеке) символ  и записывает цепочку  т.к. , при этом, если , то верхний символ магазина просто удаляется.
6. работа автомата заканчивается 

На каждом шаге автомата возможны три случая:

1. функция  определена ⎯ осуществляется переход в новое состояние;
2. функция  не определена, но определена  ⎯ осуществляется переход в новое состояние (лента не продвигается);
3. функции и  не определены ⎯ дальнейшая работа автомата не возможна (цепочка не разобрана).

Язык  ⎯ допускаемый автоматом .

**Пример**:













Этот автомат является детерминированным.

Работу автомата при распознавании входной цепочки можно представить в виде последовательности конфигураций:













Разобрана

1. Нормальная форма Хомского контекстно-свободной грамматики. Алгоритм приведения контекстно-свободной грамматики к нормальной форме Хомского. Примеры.

**Нормальная форма Хомского контекстно-свободной грамматики:**

Нормальная форма Хомского: контекстно-свободная грамматика  имеет нормальную форму Хомского, если правила  имеют вид:

* , где ;
* , где ,;
* , где - начальный символ, если есть такое правило, то  не должен встречаться в правой части правил.

**Алгоритм преобразования контекстно-свободной грамматики  к грамматике  в нормальной форме Хомского:**

преобразовать исходную грамматику к приведенному виду (исключить бесплодные и недостижимые символы, цепные и -правил);

установить 

построение :

правила вида , где ,переносятся  без изменений;

правила вида , где  переносятся в  без изменений;

правила вида , где  переносятся в  без изменений;

правила вида , где , преобразуются в правила вида  и грамматики , нетерминальный символ  добавляется в ;

правила вида , где , преобразуются в правила вида  и грамматики , нетерминальный символ  добавляется в ;

правила вида , где , преобразуются в правила вида ,,  грамматики , нетерминальные символы  добавляется в ;

правила вида , где  и  преобразуются в правила вида ,,, … ,,… Правила вида  могут потребовать дальнейшего преобразования. Если достигнут вид правил, который указан в определении, то они добавляются в , а новые нетерминальные символы в .

Пример:,  
где 

















1. Нормальная форма Грейбах контекстно-свободной грамматики. Алгоритм приведения контекстно-свободной грамматики к нормальной форме Грейбах. Примеры.

(Леворекурсивное правило: правило вида , где , 

Для каждой грамматики , содержащей леворекурсивные правила можно построить грамматику  не содержащую леворекурсивное правило.)

**Нормальная форма Грейбах**: контекстно-свободная грамматика  имеет нормальную форму Грейбах, если она не леворекурсивная (не содержит леворекурсивных правил), а правила  имеют вид:

1. , где ;
2. , где  — начальный символ, и если такое правило существует, то нетерминал  не должен встречаться в правой части правил.

Эта НФ называется так по имени Шейлы Грейбах (Sheila Greibach), которая первая описала способ построения таких грамматик.

**Алгоритм** устранения левой рекурсии: пусть правила грамматики  имеют вид:

,

где и цепочки  не начинаются с нетерминала .

Тогда эквивалентные правила без левой рекурсии





**Пример**: пусть правила  грамматики имеют вид:





Преобразование:







**Пример**: пусть правила  грамматики имеют вид:







Правила  грамматики :









