

ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ  
УНИВЕРСИТЕТ

Методическое пособие для студентов специальности 230105

Эффективная подготовка к государственным  
междисциплинарным экзаменам для студентов технических  
специальностей

## ПРЕДИСЛОВИЕ

Уважаемый читатель, мы представляем вашему вниманию очередной релиз Методического пособия для студентов специальности 230105 «Программное обеспечение вычислительной техники и автоматизированных систем», которое дает вам неограниченные возможности в процессе сдачи междисциплинарного экзамена.

В этом релизе собраны лучшие лекции, шпаргалки, конспекты наших студентов. Мы постарались как можно лучше проанализировать всю имеющуюся информацию и довести ее до вас в наиболее удобном для усвоения виде.

Особую благодарность хотим выразить всем, кто помог создать, собрать, а самое главное переработать информацию, помещенную в данном пособии.

2013 г. специальность ПС.

## **Оглавление**

I. СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.	11
1. Линейные списки. Стеки и очереди	11
2. Деревья и способы их организации в памяти. Рекурсивные алгоритмы обхода бинарных деревьев.	16
3. Представления графов с помощью матрицы смежности и списковых структур	18
4. Бинарные деревья поиска и их корректировка	19
5. АВЛ-деревья и их балансировка	21
6. Хеширование	22
7. Быстрая сортировка Хоара	26
8. Методы внешней сортировки	28
II. БАЗЫ ДАННЫХ.	31
1. Реляционная модель данных. Операции реляционной алгебры.	31
2. Нормализация отношений. Первая, вторая и третья нормальная формы.	36
3. Семантическое моделирование данных. ER-диаграммы.	40
4. Оператор SELECT языка SQL. Запросы на чтение из одной таблицы. Виды условий поиска.	44
5. Многотабличные запросы SQL. Внутренние соединения.	48
6. Псевдонимы. Внешнее соединение таблиц в SQL.	52
7. Запросы с группировкой и вложенные запросы в SQL.	52
8. Целостность данных. Транзакции.	54
9. Представления и работа с ними. Триггеры и хранимые процедуры.	58
III. ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.	61
1. Технология программирования. Система разработки ПО.	61
2. Процесс разработки программного обеспечения	64

3. Проектирование, как составляющая Система разработки ПО	66
4. Основные технологии разработки ПО	69
5. Модели жизненного цикла (ЖЦ).	72
6. Качество программной системы. Модель характеристик качества. Характеристики качества ПО	77
7 Сравнение технологий разработки ПО	81
8. Уровень формализма. Количество итерации	84
9. Требования. Анализ требований.	87
10. Управление проектом. Этапы. Задачи. Треугольник проекта.	91
11. Риски. Управление рисками.	94
12. Технология SADT. Системы и модели. Пример.	98
13. Информационно-потоковая технология проектирования. Область применения? Пример.	101
14. Экстремальное программирование.	103
15. RUP	108
16. Документирование проекта	111
17. Индивидуальный процесс разработки программного обеспечения (PSP). Оценка.	115
18. Командный процесс разработки программного обеспечения (TSP). Оценка.	118
19. Основы структурных методов проектирования	119
20. UML. Назначение	121
21. UML. Диаграмма вариантов использования (прецедентов) (use case diagram). Назначение. Пример использования. UML. Диаграмма классов (class diagram). Пример использования.	124
22. UML. Диаграммы поведения (behavior diagrams). Назначение. Пример использования.	127
23. UML. Диаграмма состояний (statechart diagram). Назначение. Пример использования.	130
24. UML. Диаграмма активности (activity diagram) . Назначение. Пример использования.	134

25. UML. Диаграммы взаимодействия (interaction diagrams). Назначение. Пример использования.	135
26. UML. Диаграмма последовательности (sequence diagram). Назначение. Пример использования.	137
27. UML. Диаграмма кооперации (collaboration diagram) . Назначение. Пример использования.	140
28. UML. Диаграммы реализации (implementation diagrams) . Назначение. Пример использования.	142
29. UML. Диаграмма компонентов (component diagram). Назначение. Пример использования.	146
30. UML. Диаграмма размещения(развертывания) (deployment diagram) . Назначение. Пример использования.	150
31. Управление проектами. Сущность управления проектами.	153
32. Управление проектами. Этапы структурного руководства проектом. Индикатор вероятности успеха (psi).	153
33. Управление проектом. Этапы. Задачи. Треугольник проекта.	157
34. Принципы тестирования. Философия тестирования.	161
35. Уровни тестирования. Этапы тестирования.	164
36. Метрики проекта.	171
37. Отладка. Основные методы отладки.	174
38. Архитектура программы. Цели выбора архитектуры. Декомпозиция.	174
39. Сопровождение ПО.	174
<b>IV. СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ</b>	<b>184</b>
1. Назовите и охарактеризуйте уровни управления ИВС по эталонной модели ВОС. Назовите сетевые устройства и ПО, работающие на этих уровнях.	184
2. Адресация в протоколах TCP/IP для сети Internet. Протокол ARP. Схемы рекурсивного и нерекурсивного режимов работы DNS-серверов.	187
3. Реализация случайных методов доступа к моноканалу в ЛВС (МДКН и МДКН/ОК). Каким образом на основе	

МДКН/ОК мосты и маршрутизаторы имеют преимущество для доступа к моноканалу по сравнению с другими узлами сети?	191
4. Объясните фазы работы протокола УЛК с установлением и без установления логического соединения. Ответ дополните диаграммой. Как для таких сетей отслеживается потеря передаваемых кадров?	194
5. Назовите принципы формирования протокольных блоков данных в рамках протоколов ЛВС. Инкапсуляция и декапсуляция сообщений. Принципы передачи команд между смежными протоколами одного узла сети и одинаковыми протоколами двух взаимодействующих узлов.	197
6. Зарисуйте структуру и назовите основные функциональные отличия повторителей, трансиверов и концентраторов ЛВС. На каком уровне эталонной модели ВОС функционирует каждое из этих устройств?	200
7. Реализация маркерного метода доступа к моноканалу в ЛВС с кольцевой топологией. Особенности организации сети Token Ring на переключающих концентраторах.	203
8. Объясните основные отличия в методе доступа для таких локальных сетей, как Token Ring и FDDI. Чем вызваны эти отличия. Синхронный и асинхронный режимы работы сети FDDI. Каким образом в сети FDDI определяется обрыв кабеля или отказ станции?	207
9. Основные функции транспортных и сетевых протоколов ИВС на примере протоколов TCP и IP. Взаимосвязь этих протоколов с другими протоколами ЭМ ВОС. Стратегии управления потоком данных.	208
10. Объясните понятие “окно конфликтов”. Как в сети Ethernet определяется эта величина и на что она влияет? Как в сети Ethernet на витой паре проводов уменьшить окно конфликтов?	211
<b>V. ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ</b>	<b>213</b>
1. Трансляция. Интерпретация и компиляция. Общие синтаксические критерии. Стадии трансляции.	213

2. Грамматики и автоматы. Классификация Хомского	217
3. Практические ограничения, налагаемые на грамматики. Отношения применимые к грамматикам.	221
4. Синтаксический анализ. Синтаксические деревья. Задача разбора. Однозначность разбора. Канонический разбор. Основа. Разбор сверху вниз, снизу вверх.	227
5. Сканер. Принципы построения.	235
6. Синтаксический анализ. Нисходящий разбор, рекурсивный спуск. Проблемы нисходящего разбора.	238
7. LL(K)-грамматики. Направляющие символы. Идея разбора.	242
8. Построение LL(1)таблицы разбора.Разбор по LL(1)таблице.Проблемы LL(1)-разбора. Достоинство и недостатки метода.	244
9. Восходящий разбор. Проблемы. Общий метод разбора.LR(K)-грамматики. Идея разбора.	249
10. Построение таблиц разбора(LR(0), SLR(1),LALR(1)).	258
11. Разбор по LR(1)таблице	265
12. Включение действий в синтаксис. Транслирующие грамматики.	269
13. Атрибутивные грамматики. Синтезируемый и наследуемый атрибуты.	275
14.Таблица символов. Назначение, структура.	280
15. Распределение памяти. Статическая и динамическая память.	286
16. Распределение памяти. Адреса времени компиляции.	290
17. Организация памяти во время выполнения. Области данных при статическом и динамическом распределении памяти.	295
18. Генерация кода. Генерация кода на примере одного из операторов Паскаля.	298
19.Свойства КС-грамматик. Лемма подкачки.	306
20.Автоматы с магазинной памятью.	307

<b>VI. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ</b>	<b>И ЛОГИЧЕСКОЕ</b>	
1. Рекурсия и циклы в Лиспе		311
2. Внутреннее представление списков в Лиспe		315
3. Декларативная и процедурная семантика Пролог-программ		317
4. Отсечение. Графическая иллюстрация действия cut. Формальное описание действия отсечения		320
5. Сравнительная характеристика функционального, логического и процедурного подхода к программированию.		323
<b>VII. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ</b>		<b>328</b>
1. Определение класса в языке C++. Функции-члены класса в языке C++. Друзья класса в языке C++. Область видимости класса в языке C++. Инициализация класса в языке C++.		328
2. Наследование в языке C++.		332
3. Виртуальные функции в языке C++.		334
4. Полиморфизм. На примере C++.		337
5. Инкапсуляция. На примере C++.		339
<b>VIII. МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ</b>		<b>341</b>
1. Системотехнические основы построения датчиков случайных чисел. Принципы аналого-цифрового преобразования. Причины выбора данного принципа аналого-цифрового преобразования.		341
2. Понятие Фон Нэймановской архитектуры вычислительной системы. Базовые принципы. Проблема получения случайных чисел в рамках данной архитектуры. Основной вывод		344
3. Системы гарантированной секретности. Теоретические основы.		346
4. Длиннопериодические ключевые последовательности. Датчики псевдослучайных чисел и их роль для создания длиннопериодических ключевых последовательностей. Анализ стойкости		

длиннoperиодических последовательностей	ключевых	
		347
5. Простейшие протоколы обеспечения многократной электронной цифровой подписи. Пример применения		349
6. Модель угроз «Несанкционированный доступ к передаваемой через открытый канал информации». Криптографические методы противодействия данной угрозе		350
7. Модель угроз «Искажение передаваемой в открытом канале информации». Криптографические методы противодействия данной угрозе. Классификация методов. Пример.		351
8. Модель угроз «Нарушение целостности программного обеспечения внутри периметра защиты». Формализация. Субъектно-объектный подход.		356
9. Теорема о неразрешимости множества доверенных субъектов в вычислительной системе Фон Нэймановской архитектуры. Связь с одним из базовых принципов Фон Нэймановской архитектуры. Понятие доверенной аппаратной компоненты.		362
10. Примеры аппаратных решений для создания изолированных программных сред.		363
<b>IX. ОПЕРАЦИОННЫЕ СИСТЕМЫ</b>	<b>366</b>	
1. Классификация ОС		366
2. Структура сетевой операционной системы		372
3. Управление процессами. Понятие процесса. Дескриптор и контекст процесса. Алгоритмы планирования процессов. Вытесняющая и не вытесняющая многозадачность.		376
4. Средства синхронизации взаимодействия процессов. Блокирующие переменные, семафоры		383
5. Взаимные блокировки процессов. Тупики распознавание, рекомендации как избежать тупик, выход из тупика.		389
6. Проблемы взаимодействия процессов. Основные задачи, возникающие при взаимодействии процессов.		392
7. Нити и процессы		392

8. Управление памятью. Типы адресов. Обзор методов распределения памяти.	395
9. Методы управления памятью без использования внешней памяти	398
10. Оверлеи. Виртуальная память. Способы организации виртуальной памяти	402
11. Свопинг и кэширование	412
12. ОС. Управление вводом-выводом	415
13. Файловая система. Основные функции. Общая схема.	419
14. Логическая и физическая организация файлов. Права доступа к файлу. Кэширование файла. Отображение файла в оперативную память. Проблемы совместного использования.	423
<b>X. ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР</b>	<b>429</b>
1. Операции над формальными языками	429
2. Двоичное кодирование переменных и функций трехзначной логики	432
3. Перечислить способы представления конечного автомата	435
4. Определение недетерминированного и конечного автомата	437
5. Программная реализация логических функций	439
<b>XI. АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ</b>	<b>442</b>
1. Виды систем обработки данных. Режимы обработки данных. Сформулируйте различия между многомашинными вычислительными комплексами и вычислительными сетями.	442
2. Уровни комплексирования устройств в вычислительных системах. Постройте структурную схему ПЭВМ, состоящей из двух процессоров. Покажите на ней используемые уровни комплексирования. Ответ поясните.	444
3. Методы улучшения ОКОД структуры. Степень, уровни и виды параллелизма. Какой из видов параллелизма реализуется в современных универсальных	

процессорах (например, в процессоре Pentium)? Ответ обоснуйте.	448
4. Подсистема памяти. Методы повышения быстродействия памяти. Виды ЗУ. Иерархическая организация памяти. Какие вычислительные системы, на каком уровне иерархической организации требуют организации пакетного доступа к памяти. Ответ поясните.	452
5. Организация кэш-памяти. Зарисуйте структуру памяти (ОЗУ и кэш) для секторированного наборно-ассоциативного кэша, состоящего из трех банков. Поясните её.	456
6. Операционный и командный конвейер. Необходимые условия организации конвейеров этих типов. Режимы работы конвейеров. Объясните, почему при организации конвейера команд не целесообразно использовать Принстонскую архитектуру ЭВМ?	458
7. Многопроцессорные вычислительные комплексы (МПВК). Типы структур. Проблемы организации. Способы распределения ресурсов в МПВК. Сравните типы структур МПВК по следующим критериям: а) быстродействию; в) аппаратным затратам на систему коммутации; г) надежности.	463
8. Машины, управляемые потоком данных. Недостатки принципа управления потоком данных. Граф потока данных. Типы вершин графа потока данных. Можно ли использовать принцип управления потоком данных в конвейерных вычислительных системах? Ответ обоснуйте.	466
<b>XII. ПРАКТИЧЕСКАЯ ЧАСТЬ</b>	<b>469</b>
1. Разработка баз данных.	469
2. UML диаграммы.	486
3. Написание технического задания	496
4. Написание постановки задачи.	499
5. ГОСТ о стадиях разработки.	500
6. ГОСТ о техническом задании.	502
7. ГОСТ о видах программ и программных документов.	506
8. Принципы Юзабилити	509

## I. СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

### 1. Линейные списки. Стеки и очереди

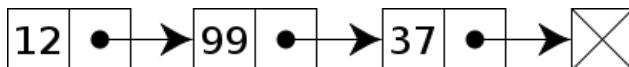
**Линейный список** – это упорядоченная структура, каждый элемент которой связан со следующим элементом. Списки могут быть реализованы статически (на массивах) и динамически (на указателях).

Статические списки более просты в реализации, но у них есть недостатки:

- 1) они не обладают достаточной гибкостью при необходимости изменения их структуры,
- 2) память под них должна быть выделена на этапе компиляции и не будет освобождена до выхода из области действия списка.

Динамические списки характеризуются высокой гибкостью. Это достигается благодаря возможности выделять и освобождать память под элементы в любой момент времени работы программы и возможности установить связь между любыми 2-я элементами с помощью указателей. Для организации связей между элементами динамической структуры данных требуется, чтобы каждый элемент содержал кроме информационных значений как минимум 1 указатель. Поэтому, наиболее часто в качестве элементов списков используют записи, которые могут объединять в единое целое разнородные элементы. В простейшем случае элемент динамической структуры должен состоять из 2 полей: информационного и указательного.

Схематично:

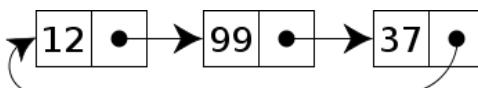


Наибольшее распространение получили два вида линейных односвязных списков: стеки и очереди.

**Кольцевой список** – список, в котором последний элемент соединен с первым.

Достоинства.

- 1) Удобство создания структур данных для циклического обслуживания.
- 2) Из любого элемента можно попасть в любой.
- 3) Экономия памяти.



**Очередь** - упорядоченный набор элементов, которые могут удаляться с одного его конца (называемого началом очереди) и помещаться в другой конец этого набора (называемого концом очереди). При работе с очередью первый помещенный в неё элемент удаляется первым (т.е. реализуется принцип FIFO). Примеры - очередь в банке, на автобусной остановке, в ЭВМ – очередь на получение ресурсов, последовательность операций. При использовании массива начало очереди обычно находится в первом элементе, а конец задается индексом и меняется при изменении очереди. При создании динамической очереди требуются указатели: на начало очереди и на конец очереди. Кроме того, для освобождения памяти удаляемых элементов требуется дополнительный временный указатель:

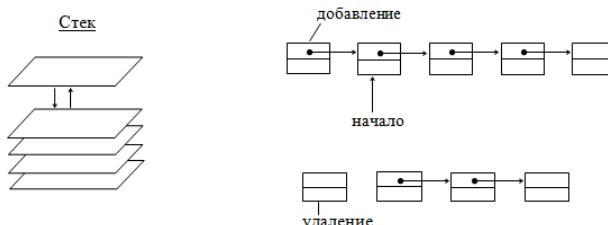
```
struct node {  
    int data;  
    struct node *next;  
};  
  
struct node *dequeue(struct node *head) {  
    assert(head != NULL);  
    struct node *tmp = head->next;  
    free(head);  
    return tmp;  
}  
  
struct node* enqueue(struct node *head, int data) {  
    if (head == NULL) {  
        head = (struct node *)malloc(sizeof(struct node));  
        assert(head != NULL);  
        head->data = data;  
        head->next = NULL;  
  
        return head;  
    }  
  
    struct node *tmp = head;  
    while (tmp->next != NULL) {  
        tmp = tmp->next;  
    }  
    tmp->next = (struct node *)malloc(sizeof(struct node));  
    assert(tmp->next != NULL);  
    tmp = tmp->next;
```

## СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

```
tmp->data = data;  
tmp->next = NULL;  
  
    return head;  
}  
  
void destroy(struct node *head) {  
    struct node *current = head;  
    while (current != NULL) {  
        head = current;  
        current = current->next;  
        free(head);  
    }  
}
```

**Стек** - частный случай линейного односвязного списка, для которого разрешено добавлять и удалять элементы только с одного конца списка, который называется вершиной (головой) стека. Т.о. элемент, помещенный в стек первым удалится из него последним (принцип FILO). В статическом представлении стек задается одномерным массивом, величина которого определяется с запасом. Необходимо обрабатывать ошибку переполнения и попытку удаления из пустого стека.

Для работы с динамическим стеком, в отличие от очереди, необходимо иметь один основной указатель на вершину стека и один дополнительный временный указатель, который используется для выделения и освобождения.



```
struct node {  
    int data;  
    struct node *next;  
};
```

```
struct node* push(struct node *stack, int data) {  
    struct node* tmp = (struct node*)malloc(sizeof(struct node));
```

```

СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

assert(tmp != NULL);

tmp->data = data;
tmp->next = stack;
return tmp;
}

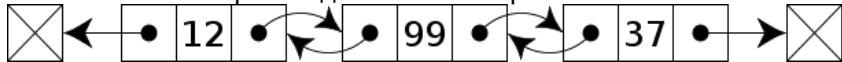
struct node* pop(struct node *stack, int *element) {
    assert(stack != NULL);
    assert(element != NULL);

    struct node *tmp = stack;
    *element = stack->data;
    stack = stack->next;
    free(tmp);
    return stack;
}

int empty(struct node *stack) {
    return (stack == NULL ? 1 : 0);
}

```

**Двусвязные списки** – списки, элементы которых имеют по 2 указателя – на правого и левого «соседа» по очереди. Удаление из таких списков происходит очень быстро и легко.



Интересным свойством такого списка является то, что для доступа к его элементам вовсе не обязательно хранить указатель на первый элемент. Достаточно иметь указатель на любой элемент списка. Первый элемент всегда можно найти по цепочке указателей на предыдущие элементы, а последний - по цепочке указателей на следующие.

Но наличие указателя на заголовок списка в ряде случаев ускоряет работу со списком.

## Мультисписки

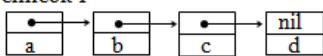
Иногда возникают ситуации, когда имеется несколько разных списков, которые включают в свой состав одинаковые элементы. В таком случае при использовании традиционных списков происходит многократное дублирование динамических

## СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

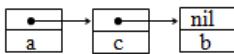
переменных и нерациональное использование памяти. Списки фактически используются не для хранения элементов данных, а для организации их в различные структуры. Использование мультилисписков позволяет упростить эту задачу.

Мультилисписок состоит из элементов, содержащих такое число указателей, которое позволяет организовать их одновременно в виде нескольких различных списков. За счет отсутствия дублирования данных память используется более рационально.

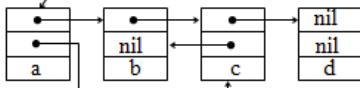
список 1



список 2



начало списков 1 и 2



мультисписок

Элементы мультилисписка

A - множество элементов списка 1

B - множество элементов списка 2

C - множество элементов мультилисписка ( $C = A \cup B$ )

Рис.1.5. Объединение двух линейных списков в один мультилисписок.

Экономия памяти – далеко не единственная причина, по которой применяют мультилисписки. Многие реальные структуры данных не сводятся к типовым структурам, а представляют собой некоторую комбинацию из них. Причем комбинируются в мультилисписках самые различные списки – линейные и циклические, односвязанные и двунаправленные.

## **2. Деревья и способы их организации в памяти. Рекурсивные алгоритмы обхода бинарных деревьев.**

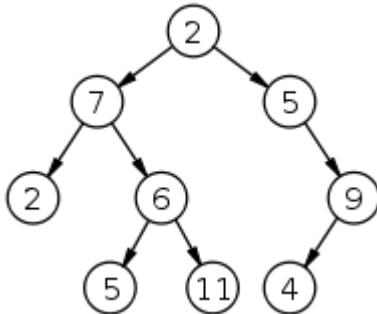
Дерево — одна из наиболее широко распространённых структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Является связанным графом, не содержащим циклы. Большинство источников также добавляют условие на то, что рёбра графа не должны быть ориентированными. В дополнение к этим трём ограничениям, в некоторых источниках указываются, что рёбра графа не должны быть взвешенными.

### **Представление деревьев**

Существует множество различных способов представления деревьев. Наиболее общий способ представления изображает узлы как записи, расположенные в динамически выделяемой памяти с указателями на своих потомков, предков (или и тех и других), или как элементы массива, связанные между собой отношениями, определёнными их позициями в массиве (например, двоичная куча).

### **Методы обхода**

Пошаговый перебор элементов дерева по связям между предками-узлами и потомками-узлами называется обходом дерева, а сам процесс называется обходом по дереву. Зачастую, операция может быть выполнена переходом указателя по отдельным узлам. Обход, при котором каждый узел-предок просматривается прежде его потомков называется предупорядоченным обходом или обходом в прямом порядке (pre-order walk), а когда просматриваются сначала потомки, а потом предки, то обход называется поступорядоченным обходом или обходом в обратном порядке (post-order walk). Существует также симметричный обход, при котором посещается сначала левое поддерево, затем узел, затем — правое поддерево, и обход в ширину, при котором узлы посещаются уровень за уровнем (N-й уровень дерева — множество узлов с высотой N). Каждый уровень обходится слева направо.



### Применение

- 1) управление иерархией данных;
  - 2) упрощение поиска информации (см. обход дерева);
  - 3) управление сортированными списками данных;
  - 4) синтаксический разбор арифметических выражений (англ. parsing), оптимизация программ;
  - 5) в качестве технологии компоновки цифровых картинок для получения различных визуальных эффектов;
- форма принятия многоэтапного решения

```
struct treeNode {  
    int element;  
    treeNode *left;  
    treeNode *right;  
};  
  
treeNode* insert(int element, struct treeNode *root) {  
    if (root == NULL) {  
        root = (struct treeNode *)malloc(sizeof(struct treeNode));  
        assert(root != NULL);  
        root->element = element;  
        root->left = NULL;  
        root->right = NULL;  
    } else if (element < root->element) {  
        root->left = insert(element, root->left);  
    } else if (element > root->element) {  
        root->right = insert(element, root->right);  
    }  
    return root;  
}  
  
void destroy_up(struct treeNode *root) {  
    if (root != NULL) {
```

```

СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

    destroy(root->left);
    destroy(root->right);
    free(root);
}
}

void down(struct treeNode *root)
{
    if (root != NULL) {
        printf("%d\n", root->element);
        down(root->left);
        down(root->right);
    }
}

void leftToRight(struct treeNode *root) {
    if (root != NULL) {
        leftToRight(root->left);
        printf("%d\n", root->element);
        leftToRight(root->right);
    }
}

```

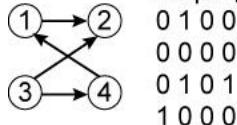
### 3. Представления графов с помощью матрицы смежности и списковых структур

Граф=структура, состоящая из вершин и связей между ними (ребер). Если ребра имеют направление, то их называют дугами, а граф - ориентированным.

Представление графа

1) Матрица смежности = матрица, в которой  $a_{ij}$  равно 1, если вершина  $a_i$  связана с вершиной  $a_j$ , иначе 0. Для неориентированных графов эта матрица симметрична.

Матрица смежности



	0	1	0	0
0	0	0	0	0
1	0	1	0	1
0	0	0	0	0

Достоинства: наглядность, легкий доступ к сыновьям и предшественникам, возможность использования аппарата матричной алгебры.

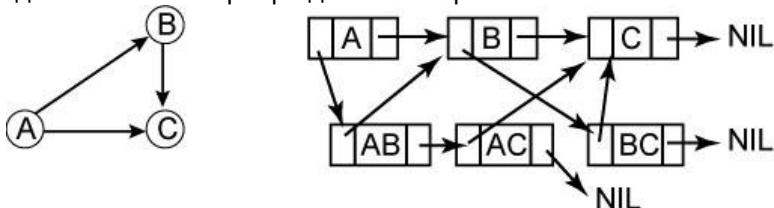
Недостатки: неэкономичность (особенно для разреженных графов), неудобство корректировки (добавления и удаления вершин).

2) Списковое представление.

## СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

Часто использование матрицы смежности неудобно, т. к. число узлов требуется знать заранее. Для решения этой проблемы используют динамическое представление графа (аналогично представлению бинарных деревьев).

Граф представляется в виде двух списков: Первый описывает вершины графа и содержит ссылку на ту часть второго списка, где описываются ребра для этой вершины:



Достоинства: экономичность, возможность корректировки

Недостатки: сложность поиска, невозможность сразу найти предшественников текущей вершины. Эти недостатки можно преодолеть введением дополнительных списков, но это еще больше усложняет структуру.

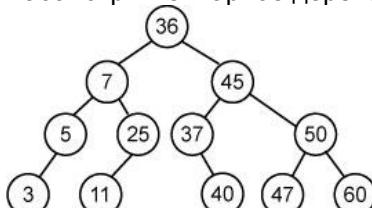
Возможны и смешанные варианты представления графа. Например, для графов, допускающих существование нескольких дуг между двумя вершинами, элементами матрицы смежности могут быть указатели на соответствующие списки дуг.

## 4. Бинарные деревья поиска и их корректировка

Бинарное дерево = дерево, у каждой вершины которого не более 2 сыновей.

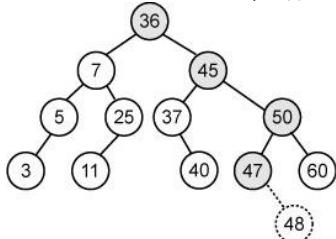
Бинарное дерево поиска = бинарное дерево, в котором у каждой вершины ключевое значение больше ключевых значений левого поддерева и меньше ключевых значений правого поддерева.

Рассмотрим бинарное дерево



Вставка. Для того чтобы определить место вставки нового ключа, вершины просматриваются начиная с корня – Если ключ для вставки меньше значения ключа текущей вершины, то переходим по левой ветке, а если больше – то по правой.

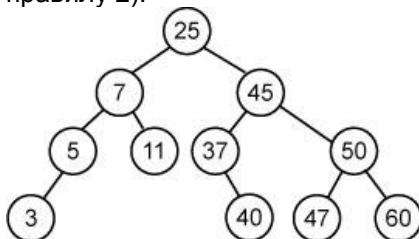
Например, вставка ключа 48:



Удаление.

- 1) Если удаляемый узел не имеет сыновей – он может быть удален без дальнейших изменений дерева (напр., вершины 3, 11, 40 и т.д)
- 2) Если узел имеет только одно поддерево, то его единственный сын перемещается на его место (Например при удалении вершин 5, 25, 37)
- 3) Если узел имеет двух сыновей: сначала на место удаленной вершины помещается один из двух соседних элементов (это может быть 1)меньший ключ самой правой вершины левого под дерева 2)больший ключ самой левой вершины правого под дерева). Найденная вершина имеет не более одного сына, поэтому дальнейшее удаление идет по описанному выше алгоритму.

Например, при удалении вершины 36 из исходного дерева, она может быть заменена вершиной 25 либо 37. При замене вершиной 25, на месте вершины 25 оказывается узел 11 (по правилу 2):



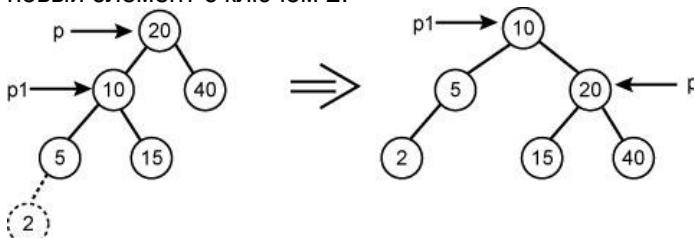
## 5. АВЛ-деревья и их балансировка

Трудоемкость поиска по бинарному дереву поиска зависит от структуры дерева. Наболее эффективен поиск по т.н. сбалансированному дереву – дереву, в котором для каждого узла высота его поддеревьев отличается не более, чем на 1. Такие деревья поиска называют АВЛ-деревьями (по имени авторов Адельсон-Вельский и Лендис)  $hl$  - высота левого поддерева;  $hr$  - высота правого поддерева  $|hl-hr|=2$  - признак нарушения балансировки. Высота пустого дерева = -1 (для удобства)

Чтобы получить сбалансированное дерево, необходимо выполнить некоторую трансформацию данного дерева так, чтобы: 1) прохождение трансформированного дерева в симметрично порядке должно быть таким же, как для первоначального дерева; 2) трансформированное дерево должно быть сбалансированным

Преобразование дерева с целью балансировки наз-ся поворотом. Сущ. 4 вида поворотов: LL, RR, LR, RL. Повороты LL, RR – одинарные. LR, RL – двойные.

Пример LL-поворота. Пусть в исходное АВЛ дерево включается новый элемент с ключом 2:

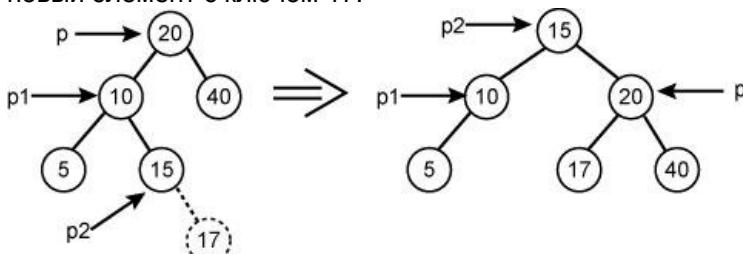


Поворот осуществляется в три операции:

$p1 := p^.left$ ;  $p^.left := p1^.right$ ;  $p1^.right := p$ ;

RR поворот осуществляется симметрично.

Пример LR поворота. Пусть в исходное АВЛ дерево включается новый элемент с ключом 17:



LR поворот требует шести операций:  $p1 := p^.left$ ;  $p2 := p1^.right$ ;  $p1^.right := p2^.left$ ;  $p2^.left := p1$ ;  $p^.left := p2^.right$ ;  $p2^.right := p$

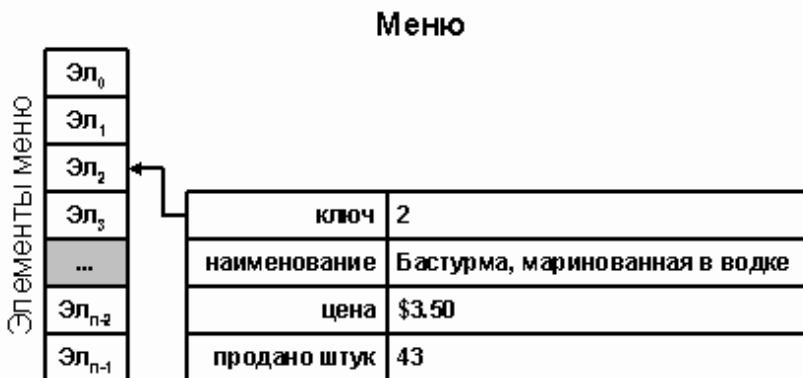
RL поворот осуществляется симметрично.

При любом включении рассматривается число уровней в левом и правом поддереве для всех вершин на пути от включаемой вершины к корню. При первом же случае нарушения баланса он восстанавливается. При включении достаточно 1 раз восстанавливать баланс.

Исключение из АВЛ дерева происходит как из обычного дерева поиска, далее требуется анализ, есть ли нарушение баланса и соответствующие повороты на пути от исключаемой вершины к корню.

## 6. Хеширование

Хеширование (иногда хэширование, англ. hashing) — преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хеш-функциями или функциями свёртки, а их результаты называют хешем, хеш-кодом или дайджестом сообщения (англ. message digest).



Хеширование применяется для сравнения данных: если у двух массивов хеш-коды разные, массивы гарантированно различаются; если одинаковые — массивы, скорее всего, одинаковы. В общем случае однозначного соответствия между исходными данными и хеш-кодом нет в силу того, что количество значений хеш-функций меньше, чем вариантов входного массива; существует множество массивов, дающих одинаковые хеш-коды — так называемые коллизии. Вероятность возникновения коллизий играет немаловажную роль в оценке качества хеш-функций.

## СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

Существует множество алгоритмов хеширования с различными характеристиками (разрядность, вычислительная сложность, криптостойкость и т. п.). Выбор той или иной хеш-функции определяется спецификой решаемой задачи. Простейшими примерами хеш-функций могут служить контрольная сумма или CRC.

**Хеш-таблица** — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Существует два основных варианта хеш-таблиц: **с цепочками и открытой адресацией**. Хеш-таблица содержит некоторый массив  $H$ , элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица с цепочками).

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение  $i = \text{hash}(\text{key})$  играет роль индекса в массиве  $H$ . Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива  $H[i]$ .

Ситуация, когда для различных ключей получается одно и то же хеш-значение, называется **коллизией**. Такие события не так уж и редки — например, при вставке в хеш-таблицу размером 365 ячеек всего лишь 23-х элементов вероятность коллизии уже превысит 50 % (если каждый элемент может равновероятно попасть в любую ячейку). Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы.

В некоторых специальных случаях удается избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую совершенную хеш-функцию, которая распределит их по ячейкам хеш-таблицы без коллизий. Хеш-таблицы, использующие подобные хеш-функции, не нуждаются в механизме разрешения коллизий, и называются хеш-таблицами с прямой адресацией.

Число хранимых элементов, делённое на размер массива  $H$  (число возможных значений хеш-функции), называется коэффициентом заполнения хеш-таблицы (load factor) и является важным параметром, от которого зависит среднее время выполнения операций.

## **Свойства хеш-таблицы**

Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время  $O(1)$ . Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хеш-таблицы: увеличить значение размера массива  $H$  и заново добавить в пустую хеш-таблицу все пары.

### **Разрешение коллизий**

Несмотря на то, что два или более ключей могут хешироваться одинаково, они не могут занимать в хеш-таблице одну и ту же ячейку. Остаётся два пути: либо найти для нового ключа другую позицию в таблице, либо создать для каждого значения хеш-функции отдельный список, в котором будут все ключи, отображающиеся при хешировании в это значение. Оба варианта представляют собой две классические стратегии разрешения коллизий – **открытую адресацию с линейным перебором** и **метод цепочек**.

### **Открытая адресация с линейным перебором**

Эта методика предполагает, что каждая ячейка таблицы помечена как незанятая. Поэтому при добавлении нового ключа всегда можно определить, занята ли данная ячейка таблицы или нет. Если да, алгоритм осуществляет перебор по кругу, пока не встретится «открытый адрес» (свободное место). Отсюда и название метода. Если размер таблицы велик относительно числа хранимых там ключей, метод работает хорошо, поскольку хеш-функция будет равномерно распределять ключи по всему диапазону и число коллизий будет минимальным. По мере того как коэффициент заполнения таблицы приближается к 1, эффективность процесса заметно падает.

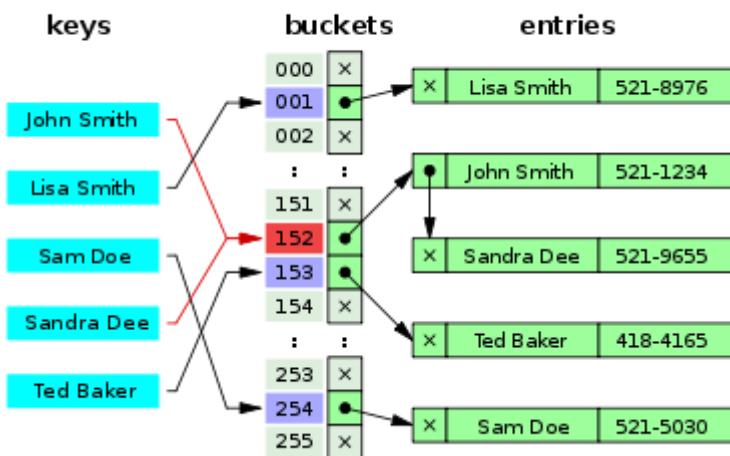
### **Метод цепочек**

При другом подходе к хешированию таблица рассматривается как массив связанных списков или деревьев. Каждый такой список называется **блоком** (bucket) и содержит записи, отображаемые хеш-функцией в один и тот же табличный адрес. Эта стратегия разрешения коллизий называется **методом цепочек (chaining with separate lists)**.

## СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

Если таблица является массивом связанных списков, то элемент данных просто вставляется в соответствующий список в качестве нового узла. Чтобы обнаружить элемент данных, нужно применить хеш-функцию для определения нужного связанного списка и выполнить там последовательный поиск.

В общем случае метод цепочек быстрее открытой адресации, так как просматривает только те ключи, которые попадают в один и тот же табличный адрес. Кроме того, открытая адресация предполагает наличие таблицы фиксированного размера, в то время как в методе цепочек элементы таблицы создаются динамически, а длина списка ограничена лишь количеством памяти. Основным недостатком метода цепочек являются дополнительные затраты памяти на поля указателей. В общем случае динамическая структура метода цепочек более предпочтительна для хеширования.



## **7. Быстрая сортировка Хоара**

часто называемая `qsort` по имени реализации в стандартной библиотеке языка Си — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром в 1960 году. Один из быстрых известных универсальных алгоритмов сортировки массивов (в среднем  $O(n \log n)$  обменов при упорядочении  $n$  элементов), хотя и имеющий ряд недостатков.

### **Краткое описание алгоритма**

- 1) выбрать элемент, называемый опорным.
- 2) сравнить все остальные элементы с опорным, на основании сравнения  
разбить множество на три — «меньшие опорного», «равные» и «большие»,  
расположить их в порядке меньшие-равные-большие.
- 3) повторить рекурсивно для «меньших» и «больших».

### **Оценка эффективности**

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного, в том числе, своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь меняются местами наиболее удалённые друг от друга элементы массива. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате самый эффективный улучшенный метод.

- 1) Лучший случай. Для этого алгоритма самый лучший случай — если в каждой итерации каждый из подмассивов делился бы на два равных по величине массива. В результате количество сравнений, делаемых быстрой сортировкой, было бы равно значению рекурсивного выражения  $CN = 2CN/2+N$ . Это дало бы наименьшее время сортировки.
- 2) Среднее. Даёт в среднем  $O(n \lg n)$  обменов при упорядочении  $n$  элементов. В реальности именно такая ситуация обычно имеет место при случайному порядке элементов и выборе опорного элемента из середины массива либо случайно.

На практике быстрая сортировка значительно быстрее, чем другие алгоритмы с оценкой  $O(n \lg n)$ , по причине того, что

## СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

внутренний цикл алгоритма может быть эффективно реализован почти на любой архитектуре.  $2CN/2$  покрывает расходы по сортировке двух полученных подмассивов;  $N$  — это стоимость обработки каждого элемента, используя один или другой указатель. Известно также, что примерное значение этого выражения равно  $CN = N \lg N$ .

3) Худший случай. Худшим случаем, очевидно, будет такой, при котором на каждом этапе массив будет разделяться на вырожденный подмассив из одного опорного элемента и на подмассив из всех остальных элементов. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых.

Худший случай даёт  $O(n^2)$  обменов, но количество обменов и, соответственно, время работы — это не самый большой его недостаток. Хуже то, что в таком случае глубина рекурсии при выполнении алгоритма достигнет  $n$ , что будет означать  $n$ -кратное сохранение адреса возврата и локальных переменных процедуры разделения массивов. Для больших значений  $n$  худший случай может привести к исчерпанию памяти во время работы алгоритма. Впрочем, на большинстве реальных данных можно найти решения, которые минимизируют вероятность того, что понадобится квадратичное время.

### **Достоинства**

- 1) Один из самых быстродействующих (на практике) из алгоритмов внутренней сортировки общего назначения.
- 2) Прост в реализации.
- 3) Требует лишь  $O(\lg n)$  дополнительной памяти для своей работы.
- 4) Хорошо сочетается с механизмами кэширования и виртуальной памяти.
- 5) Существует эффективная модификация (алгоритм Седжвика) для сортировки строк — сначала сравнение с опорным элементом только по нулевому символу строки, далее применение аналогичной сортировки для «большего» и «меньшего» массивов тоже по нулевому символу, и для «равного» массива по уже первому символу.

### **Недостатки**

- 1) Сильно деградирует по скорости (до  $\Theta(n^2)$ ) при неудачных выборах опорных элементов, что может случиться при неудачных входных данных. Этого можно избежать, используя такие модификации алгоритма, как Introsort, или

- вероятностно, выбирая опорный элемент случайно, а не фиксированным образом.
- 2) Наивная реализация алгоритма может привести к ошибке переполнения стека, так как ей может потребоваться сделать  $O(n)$  вложенных рекурсивных вызовов. В улучшенных реализациях, в которых рекурсивный вызов происходит только для сортировки большей из двух частей массива, глубина рекурсии гарантированно не превысит  $O(\lg n)$ .
  - 3) Неустойчив — если требуется устойчивость, приходится расширять ключ.

## **8. Методы внешней сортировки**

Внешняя сортировка — сортировка данных, расположенных на периферийных устройствах и не вмещающихся в оперативную память, то есть когда применить одну из внутренних сортировок невозможно. Стоит отметить, что внутренняя сортировка значительно эффективней внешней, так как на обращение к оперативной памяти затрачивается намного меньше времени, чем к магнитным дискам, лентам и т. п.

Наиболее часто внешняя сортировка используется в СУБД. Основным понятием при использовании внешней сортировки является понятие отрезка. Отрезком длины  $K$  является последовательность записей  $A_i, A_{i+1}, \dots, A_{i+k}$ , что в ней все записи упорядочены по некоторому ключу. Максимальное количество отрезков в файле  $N$  (все элементы не упорядочены). Минимальное количество отрезков 1 (все элементы являются упорядоченными).

### **Основные методы сортировок**

- 1) Естественная сортировка (метод естественного слияния)
- 2) Сортировка методом двухпутевого сбалансированного слияния
  - Сортировка методом  $n$ -путевого слияния.
- 3) Многофазная сортировка

Сортировка слиянием. Слияние означает объединение двух (или более) упорядоченных последовательностей в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент. Слияние — намного более простая операция, чем сортировка; она используется в качестве вспомогательной в более сложном

## СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

процессе последовательной сортировки. Один из методов сортировки слиянием называется простым слиянием.

### **Метод простого слияния состоит в следующем:**

- 1) Последовательность, а разбивается на две половины  $b$  и  $c$ .
- 2) Последовательности  $b$  и  $c$  сливаются при помощи объединения отдельных элементов в упорядоченные пары.
- 3) Полученной последовательности присваивается имя  $a$ , и повторяются шаги 1 и 2; на этот раз упорядоченные пары сливаются в упорядоченные четверки.
- 4) Предыдущие шаги повторяются; четверки сливаются в восьмерки, и весь процесс продолжается до тех пор, пока не будет упорядочена вся последовательность, ведь длины сливаемых последовательностей каждый раз удваиваются.

В качестве примера рассмотрим последовательность

44 55 12 42 94 18 06 67

На первом шаге разбиение дает последовательности

44 55 12 42

94 18 06 67

Слияние отдельных компонент (которые являются упорядоченными последовательностями длины 1) в упорядоченные пары дает

44 94 ' 18 55 ' 06 12 ' 42 67

Новое разбиение пополам и слияние упорядоченных пар дают

06 12 44 94 ' 18 42 55 67

Третье разбиение и слияние приводят, наконец, к нужному результату.

06 12 18 42 44 55 67 94.

### **Многофазная сортировка:**

- 1) На первом шаге мы прочитаем  $S$  записей и отсортируем их с помощью подходящей внутренней сортировки. Этот набор уже отсортированных записей перепишем в файл А. Затем прочитаем еще  $S$  записей, отсортируем их и перепишем в файл В. Этот процесс продолжается, причем отсортированные блоки записей пишутся попеременно то в файл А, то в файл В.
- 2) После того, как входной файл полностью разбит на отсортированные отрезки, мы готовы перейти ко второму шагу - слиянию этих отрезков. Каждый из файлов А и В содержит некоторую последовательность

## СТРУКТУРЫ И ОРГАНИЗАЦИЯ ДАННЫХ В ЭВМ.

отсортированных отрезков, однако, как и в случае сортировки слиянием, мы ничего не можем сказать о порядке записей в двух различных отрезках. Мы начинаем с чтения половинок первых отрезков из файлов А и В. Читаем мы лишь по половине отрезков, поскольку мы уже выяснили, что в памяти может находиться одновременно лишь S записей, а нам нужны записи из обоих файлов. Будем теперь сливать эти половинки отрезков в один отрезок файла С. После того, как одна из половинок закончится, мы прочтем вторую половинку из того же файла. Когда обработка одного из отрезков будет завершена, конец второго отрезка будет переписан в файл С. После того, как слияние первых двух отрезков из файлов А и В будет завершено, следующие два отрезка сливаются в файл D. Этот процесс слияния отрезков продолжается с попаренной записью слитых отрезков в файлы С и D. По завершении мы получаем два файла, разбитых на отсортированные отрезки длины 2S. Затем процесс повторяется, причем отрезки читаются из файлов С и D, а слитые отрезки длины 4S записываются в файлы А и В. Ясно, что в конце концов отрезки сольются в один отсортированный список в одном из файлов.

A: 2 4 1 5 7 8

B: 0 9 3 6

s=2

Преобразовывается:

A: 2 4 1 5 7 8

B: 0 9 3 6

Получается:

C: 0 2 4 9 7 8

D: 1 3 5 6

s=4

Преобразовывается:

C: 0 2 4 9 7 8

D: 1 3 5 6

Получается:

A: 0 1 2 3 4 5 6 9

B: 7 8

s=8

Преобразовывается:

A: 0 1 2 3 4 5 6 9

B: 7 8

Получается:

## БАЗЫ ДАННЫХ.

C: 0 1 2 3 4 5 6 7 8 9

D:

Все!

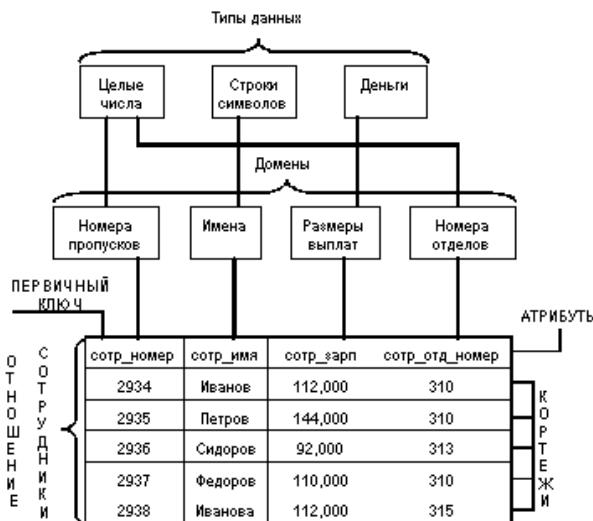
C: 0 1 2 3 4 5 6 7 8 9

D:

## **II. БАЗЫ ДАННЫХ.**

### **1. Реляционная модель данных. Операции реляционной алгебры.**

**Основными понятиями реляционных баз данных являются тип данных, домен, атрибут, кортеж, первичный ключ и отношение.** Для начала покажем смысл этих понятий на примере отношения СОТРУДНИКИ, содержащего информацию о сотрудниках некоторой организации:



Понятие тип данных в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Обычно в современных реляционных БД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (таких как "деньги"), а также специальных "temporalных" данных (дата, время, временной интервал).

Понятие домена более специфично для баз данных, хотя и имеет некоторые аналогии с подтипами в некоторых языках программирования. В самом общем виде домен определяется

## БАЗЫ ДАННЫХ.

заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных. Если вычисление этого логического выражения дает результат "истина", то элемент данных является элементом домена.

Схема отношения - это именованное множество пар {имя атрибута, имя домена (или типа, если понятие домена не поддерживается)}. Степень или "арность" схемы отношения - мощность этого множества. Степень отношения СОТРУДНИКИ равна четырем, то есть оно является 4-арным. Если все атрибуты одного отношения определены на разных доменах, осмысленно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это является всего лишь удобным способом именования и не устраниет различия между понятиями домена и атрибута).

Схема БД (в структурном смысле) - это набор именованных схем отношений.

Кортеж, соответствующий данной схеме отношения, - это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. "Значение" является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень или "арность" кортежа, т.е. число элементов в нем, совпадает с "арностью" соответствующей схемы отношения. Попросту говоря, кортеж - это набор именованных значений заданного типа.

Отношение - это множество кортежей, соответствующих одной схеме отношения. Иногда, чтобы не путаться, говорят "отношение-схема" и "отношение-экземпляр", иногда схему отношения называют заголовком отношения, а отношение как набор кортежей - телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования.

Обычным житейским представлением отношения является таблица, заголовком которой является схема отношения, а строками - кортежи отношения-экземпляра; в этом случае имена атрибутов именуют столбцы этой таблицы. Поэтому иногда говорят "столбец таблицы", имея в виду "атрибут отношения".

Реляционная база данных - это набор отношений, имена которых совпадают с именами схем отношений в схеме БД.

## **Фундаментальные свойства отношений**

Отсутствие кортежей-дубликатов. То свойство, что отношения не содержат кортежей-дубликатов, следует из определения отношения как множества кортежей. В классической теории

## БАЗЫ ДАННЫХ.

множеств по определению каждое множество состоит из различных элементов. Из этого свойства вытекает наличие у каждого отношения так называемого первичного ключа - набора атрибутов, значения которых однозначно определяют кортеж отношения. Для каждого отношения по крайней мере полный набор его атрибутов обладает этим свойством. Однако при формальном определении первичного ключа требуется обеспечение его "минимальности", т.е. в набор атрибутов первичного ключа не должны входить такие атрибуты, которые можно отбросить без ущерба для основного свойства - однозначно определять кортеж.

Отсутствие упорядоченности кортежей. Свойство отсутствия упорядоченности кортежей отношения также является следствием определения отношения-экземпляра как множества кортежей.

Отсутствие упорядоченности атрибутов. Атрибуты отношений не упорядочены, поскольку по определению схема отношения есть множество пар {имя атрибута, имя домена}. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута.

Атомарность значений атрибутов. Значения всех атрибутов являются атомарными. Это следует из определения домена как потенциального множества значений простого типа данных, т.е. среди значений домена не могут содержаться множества значений (отношения). Принято говорить, что в реляционных базах данных допускаются только нормализованные отношения или отношения, представленные в первой нормальной форме. Наиболее распространенная трактовка **реляционной модели данных**, по-видимому, принадлежит Дейту, который воспроизводит ее (с различными уточнениями) практически во всех своих книгах. Согласно Дейту реляционная модель состоит из трех частей, описывающих разные аспекты реляционного подхода: структурной части, манипуляционной части и целостной части.

В структурной части модели фиксируется, что единственной структурой данных, используемой в реляционных БД, является нормализованное n-арное отношение. По сути дела, в предыдущих двух разделах этой лекции мы рассматривали именно понятия и свойства структурной составляющей реляционной модели.

В манипуляционной части модели утверждаются два фундаментальных механизма манипулирования реляционными БД - реляционная алгебра и реляционное исчисление. Первый механизм базируется в основном на классической теории

## БАЗЫ ДАННЫХ.

множеств (с некоторыми уточнениями), а второй - на классическом логическом аппарате исчисления предикатов первого порядка.

Наконец, в целостной части реляционной модели данных фиксируются два базовых требования целостности, которые должны поддерживаться в любой реляционной СУБД.

Первое требование называется требованием целостности сущностей. Объекту или сущности реального мира в реляционных БД соответствуют кортежи отношений. Конкретно требование состоит в том, что любой кортеж любого отношения отличим от любого другого кортежа этого отношения, т.е. другими словами, любое отношение должно обладать первичным ключом. Как мы видели в предыдущем разделе, это требование автоматически удовлетворяется, если в системе не нарушаются базовые свойства отношений.

Второе требование называется требованием целостности по ссылкам и является несколько более сложным. Требование целостности по ссылкам, или требование внешнего ключа, состоит в том, что для каждого значения внешнего ключа, появляющегося взывающемся отношении, в отношении, на которое ведет ссылка, должен найтись кортеж с таким же значением первичного ключа, либо значение внешнего ключа должно быть неопределенным (т.е. ни на что не указывать). Для примера это означает, что если для сотрудника указан номер отдела, то этот отдел должен существовать.

Существует много подходов к определению реляционной алгебры, которые различаются набором операций и способами их интерпретации, но в принципе, более или менее равносильны.

Мы опишем **немного расширенный начальный вариант алгебры**, который был предложен Коддом. В этом варианте набор основных алгебраических операций состоит из восьми операций, которые делятся на два класса - теоретико-множественные операции и специальные реляционные операции.

В состав теоретико-множественных операций входят операции:

- объединения отношений; При выполнении операции объединения двух отношений производится отношение, включающее все кортежи, входящие хотя бы в одно из отношений-операндов.
- пересечения отношений; Операция пересечения двух отношений производит отношение, включающее все кортежи, входящие в оба отношения-операнда.
- взятия разности отношений; Отношение, являющееся разностью двух отношений включает все кортежи, входящие в

## БАЗЫ ДАННЫХ.

- отношение - первый операнд, такие, что ни один из них не входит в отношение, являющееся вторым операндом.
- прямого произведения отношений. При выполнении прямого произведения двух отношений производится отношение, кортежи которого являются конкатенацией (сцеплением) кортежей первого и второго операндов.

Специальные реляционные операции включают:

- ограничение отношения; Результатом ограничения отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющее этому условию.
- проекцию отношения; При выполнении проекции отношения на заданный набор его атрибутов производится отношение, кортежи которого производятся путем взятия соответствующих значений из кортежей отношения-операнда.
- соединение отношений; При соединении двух отношений по некоторому условию образуется результатирующее отношение, кортежи которого являются конкатенацией кортежей первого и второго отношений и удовлетворяют этому условию.
- деление отношений. У операции реляционного деления два операнда - бинарное и унарное отношения. Результатирующее отношение состоит из одноатрибутных кортежей, включающих значения первого атрибута кортежей первого операнда таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) совпадает со множеством значений второго операнда.

Кроме того, в состав алгебры включается операция присваивания, позволяющая сохранить в базе данных результаты вычисления алгебраических выражений, и операция переименования атрибутов, дающая возможность корректно сформировать заголовок (схему) результатирующего отношения. Операция присваивания позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД. Операция переименования производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены.

## 2. Нормализация отношений. Первая, вторая и третья нормальная формы.

### **1НФ (Первая Нормальная Форма)**

Свойства отношения:

- В отношении нет одинаковых кортежей.
- Кортежи не упорядочены.
- Атрибуты не упорядочены и различаются по наименованию.
- Все значения атрибутов атомарны.

**Первая нормальная форма (1НФ)** - это обычное отношение.

### **Определение функциональной зависимости**

**Определение 1.** Пусть R - отношение. Множество атрибутов Y **функционально зависимо** от множества атрибутов X (X **функционально определяет** Y) тогда и только тогда, когда для любого состояния отношения R для любых кортежей  $r_1, r_2$  принадл. R из того, что  $r_1.X = r_2.X$  следует что  $r_1.Y = r_2.Y$  (т.е. во всех кортежах, имеющих одинаковые значения атрибутов X, значения атрибутов Y также совпадают в любом состоянии отношения R). Символически функциональная зависимость записывается  $X \rightarrow Y$ . Множество атрибутов X называется **дeterminантом функциональной зависимости**, а множество атрибутов Y называется **зависимой частью**.

**Определение 2.** Функциональная зависимость (**функция**) - это тройка объектов  $\{X, Y, f\}$ , где X- множество (**область определения**), Y- множество (**множество значений**), f- правило, согласно которому каждому элементу x принадл. X ставится в соответствие один и только один элемент у принадл. Y (**правило функциональной зависимости**).

Функциональная зависимость обычно обозначается как  $f: X \rightarrow Y$  или  $y=f(x)$ .

Отношения, находящиеся в 1НФ являются "плохими" в том смысле, что они не удовлетворяют выбранным критериям - имеется большое количество аномалий обновления, для поддержания целостности БД требуется разработка сложных триггеров.

### **2НФ (Вторая Нормальная Форма)**

**Определение 3.** Отношение R находится во **второй нормальной форме (2НФ)** тогда и только тогда, когда отношение находится в 1НФ и *нет неключевых атрибутов, зависящих от части сложного ключа.* (**Неключевой атрибут** - это атрибут, не входящий в состав никакого потенциального ключа).

### БАЗЫ ДАННЫХ.

Отношения в 2НФ "лучше", чем в 1НФ, но еще недостаточно "хороши" - остается часть аномалий обновления, по-прежнему требуются триггеры, поддерживающие целостность БД.

**Пример приведения таблицы ко второй нормальной форме**  
Пусть Сотрудник и Должность вместе образуют первичный ключ в такой таблице:

Сотрудник	Должность	Зарплата	Наличие компьютера
Гришин	Кладовщик	20000	Нет
Васильев	Программист	40000	Есть
Васильев	Кладовщик	25000	Нет

зарплату сотруднику каждый начальник устанавливает сам, но её границы зависят от должности. Наличие же компьютера у сотрудника зависит только от должности, то есть зависимость от первичного ключа неполная

В результате приведения к 2NF получаются две таблицы:

Сотрудник	Должность	Зарплата
Гришин	Кладовщик	20000
Васильев	Программист	40000
Васильев	Кладовщик	25000

Здесь первичный ключ, как и в исходной таблице, составной, но единственный не входящий в него атрибут Зарплата зависит теперь от всего ключа, то есть полно.

Должность	Наличие компьютера
Кладовщик	Нет
Программист	Есть

### 3НФ (Третья Нормальная Форма)

**Определение 4.** Атрибуты называются *взаимно независимыми*, если ни один из них не является функционально зависимым от другого.

**Определение 5.** Отношение R находится в *третьей нормальной форме (3НФ)* тогда и только тогда, когда отношение находится в 2НФ и *все неключевые атрибуты взаимно независимы*.

Отношения в 3НФ являются самыми "хорошими" с точки зрения выбранных нами критериев - устраниены аномалии обновления, требуются только стандартные триггеры для поддержания ссылочной целостности.

### **Нормализация отношений.**

## БАЗЫ ДАННЫХ.

Переход от ненормализованных отношений к отношениям в ЗНФ может быть выполнен при помощи **алгоритма нормализации** - последовательной декомпозиции отношений для устранения функциональных зависимостей атрибутов от части сложного ключа (приведение к 2НФ) и устранения функциональных зависимостей неключевых атрибутов друг от друга (приведение к 3НФ).

### **Алгоритм нормализации (приведение к ЗНФ)**

1. *Приведение к 1НФ.* На первом шаге задается одно или несколько отношений, отображающих понятия предметной области. По модели предметной области (не по внешнему виду полученных отношений!) выписываются обнаруженные функциональные зависимости. Все отношения автоматически находятся в 1НФ.
2. *Приведение к 2НФ.* Если в некоторых отношениях обнаружена зависимость атрибутов от части сложного ключа, то проводим декомпозицию этих отношений на несколько отношений следующим образом: те атрибуты, которые зависят от части сложного ключа выносятся в отдельное отношение вместе с этой частью ключа. В исходном отношении остаются все ключевые атрибуты:  
Исходное отношение:  $R(K_1, K_2, A_1, \dots, A_n, B_1, \dots, B_m)$ . Ключ:  $\{K_1, K_2\}$ - сложный.  
Функциональные зависимости:  
 $\{K_1, K_2\} \rightarrow \{A_1, \dots, A_n, B_1, \dots, B_m\}$  - зависимость всех атрибутов от ключа отношения.  
 $\{K_1\} \rightarrow (A_1, \dots, A_n)$  - зависимость некоторых атрибутов от части сложного ключа.  
Декомпозированные отношения:  
 $R_1(K_1, K_2, B_1, \dots, B_m)$  - остаток от исходного отношения. Ключ  $\{K_1, K_2\}$ .  
 $R_2(K_1, A_1, \dots, A_n)$  - атрибуты, вынесенные из исходного отношения вместе с частью сложного ключа. Ключ  $K_1$ .
3. *Приведение к 3НФ.* Если в некоторых отношениях обнаружена зависимость некоторых неключевых атрибутов других неключевых атрибутов, то проводим декомпозицию этих отношений следующим образом: те неключевые атрибуты, которые зависят других неключевых атрибутов выносятся в отдельное отношение. В новом отношении ключом становится детерминант функциональной зависимости:  
Исходное отношение:  $R(K, A_1, \dots, A_n, B_1, \dots, B_m)$ . Ключ:  $K$ .  
Функциональные зависимости:

### БАЗЫ ДАННЫХ.

$K \rightarrow \{A_1, \dots, A_n, B_1, \dots, B_m\}$  - зависимость всех атрибутов от ключа отношения.

$\{A_1, \dots, A_n\} \rightarrow \{B_1, \dots, B_m\}$  - зависимость некоторых неключевых атрибутов других неключевых атрибутов.

Декомпозированные отношения:

$R_1(K, A_1, \dots, A_n)$  - остаток от исходного отношения. Ключ  $K$ .

$R_2(A_1, \dots, A_n, B_1, \dots, B_m)$  - атрибуты, вынесенные из исходного отношения вместе с детерминантой функциональной зависимости. Ключ  $\{A_1, \dots, A_n\}$ .

### **Сравнение нормализованных и ненормализованных моделей**

Критерий	Отношения слабо нормализованы (1НФ, 2НФ)	Отношения сильно нормализованы (3НФ)
Адекватность базы данных предметной области	ХУЖЕ (-)	ЛУЧШЕ (+)
Легкость разработки и сопровождения базы данных	СЛОЖНЕЕ (-)	ЛЕГЧЕ (+)
Скорость вставки, обновления, удаления	МЕДЛЕННЕЕ (-)	БЫСТРЕЕ (+)
Скорость выполнения выборки данных	БЫСТРЕЕ (+)	МЕДЛЕННЕЕ (-)

### **Пример приведения таблицы к третьей нормальной форме**

Есть у нас таблица:

Имя шпиона	Государство
Джеймс Бонд	Великобритания
Ким Филби	СССР
Штирлиц	СССР

В этой таблице ключом является имя шпиона. А не ключевым полем – государство, на которое он работает. Вполне логично предположить, что в этой таблице государства могут быть одинаковыми для нескольких записей. И для того, чтобы эта таблица находилась в третьей нормальной форме, не необходимо ее разделить на две:

## БАЗЫ ДАННЫХ.

ID	Государство
1	Великобритания
2	СССР

Имя шпиона	Государство
Джеймс Бонд	1
Ким Филби	2
Штирлиц	2

### **3. Семантическое моделирование данных. ER-диаграммы.**

Моделирование структуры базы данных при помощи алгоритма нормализации имеет серьезные недостатки:

- Первоначальное размещение всех атрибутов в одном отношении является очень неестественной операцией. Интуитивно разработчик сразу проектирует несколько отношений в соответствии с обнаруженными сущностями. Даже если совершил насилие над собой и создать одно или несколько отношений, включив в них все предполагаемые атрибуты, то совершенно неясен смысл полученного отношения.
- Невозможно сразу определить полный список атрибутов. Пользователи имеют привычку называть разными именами одни и те же вещи или наоборот, называть одними именами разные вещи.
- Для проведения процедуры нормализации необходимо выделить зависимости атрибутов, что тоже очень нелегко, т.к. необходимо явно выписать все зависимости, даже те, которые являются очевидными.

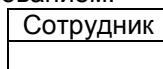
В реальном проектировании структуры базы данных применяются другой метод - так называемое, **семантическое моделирование**. Семантическое моделирование представляет собой моделирование структуры данных, опираясь на смысл этих данных. В качестве инструмента семантического моделирования используются различные варианты **диаграмм сущность-связь** (**ER - Entity-Relationship**).

**Определение 1. Сущность** - это класс однотипных объектов, информация о которых должна быть учтена в модели.

Каждая сущность должна иметь наименование, выраженное существительным в единственном числе. Примерами сущностей могут быть такие классы объектов как "Поставщик", "Сотрудник", "Накладная".

## БАЗЫ ДАННЫХ.

Каждая сущность в модели изображается в виде прямоугольника с наименованием:



**Определение 2. Экземпляр сущности** - это конкретный

представитель данной сущности.

Например, представителем сущности "Сотрудник" может быть  
"Сотрудник Иванов".

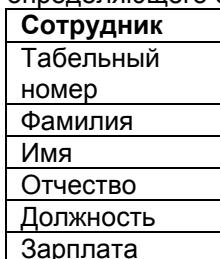
Экземпляры сущностей должны быть **различимы**, т.е.  
сущности должны иметь некоторые свойства, уникальные для  
каждого экземпляра этой сущности.

**Определение 3. Атрибут сущности** - это именованная  
характеристика, являющаяся некоторым свойством сущности.

Наименование атрибута должно быть выражено  
существительным в единственном числе (возможно, с  
характеризующими прилагательными).

Примерами атрибутов сущности "Сотрудник" могут быть  
такие атрибуты как "Табельный номер", "Фамилия", "Имя",  
"Отчество", "Должность", "Зарплата" и т.п.

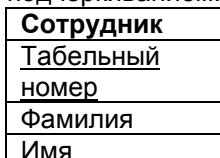
Атрибуты изображаются в пределах прямоугольника,  
определяющего сущность:



**Определение 4. Ключ сущности** - это **неизбыточный** набор  
атрибутов, значения которых в совокупности являются  
уникальными для каждого экземпляра сущности. Неизбыточность  
заключается в том, что удаление любого атрибута из ключа  
нарушается его уникальность.

Сущность может иметь несколько различных ключей.

Ключевые атрибуты изображаются на диаграмме  
подчеркиванием:



## БАЗЫ ДАННЫХ.

Отчество
Должность
Зарплата

**Определение 5. Связь** - это некоторая ассоциация между двумя сущностями. Одна сущность может быть связана с другой сущностью или сама с собою.

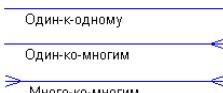
Связи позволяют по одной сущности находить другие сущности, связанные с нею.

Например, связи между сущностями могут выражаться следующими фразами - "СОТРУДНИК может иметь несколько ДЕТЕЙ", "каждый СОТРУДНИК обязан числиться ровно в одном ОТДЕЛЕ".

Графически связь изображается линией, соединяющей две сущности:



Каждая связь имеет два конца и одно или два наименования. Наименование обычно выражается в неопределенной глагольной форме: "иметь", "принадлежать" и т.п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности. Каждая связь может иметь один из следующих **типов связи**:



Связь типа **один-к-одному** означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.

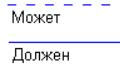
Связь типа **один-ко-многим** означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны "один") называется **родительской**, правая (со стороны "много") - **дочерней**. Характерный пример такой связи приведен на Рис. 4.

Связь типа **много-ко-многим** означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой

## БАЗЫ ДАННЫХ.

сущности. Тип связи много-ко-многим является *временным* типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен двумя связями типа один-ко-многим путем создания промежуточной сущности.

Каждая связь может иметь одну из двух **модальностей связи**:



Модальность "**может**" означает, что экземпляр одной сущности *может быть связан с одним или несколькими экземплярами другой сущности, а может быть и не связан ни с одним экземпляром*.

Модальность "**должен**" означает, что экземпляр одной сущности *обязан быть связан не менее чем с одним экземпляром другой сущности*.

Связь может иметь *разную модальность* с разных концов (как на Рис. 4).

Описанный графический синтаксис позволяет однозначно читать диаграммы, пользуясь следующей схемой построения фраз:

<Каждый экземпляр СУЩНОСТИ 1> <МОДАЛЬНОСТЬ СВЯЗИ> <НАИМЕНОВАНИЕ СВЯЗИ> <ТИП СВЯЗИ> <экземпляр СУЩНОСТИ 2>.

Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на Рис. 4 читается так:

Слева направо: "каждый сотрудник может иметь несколько детей".

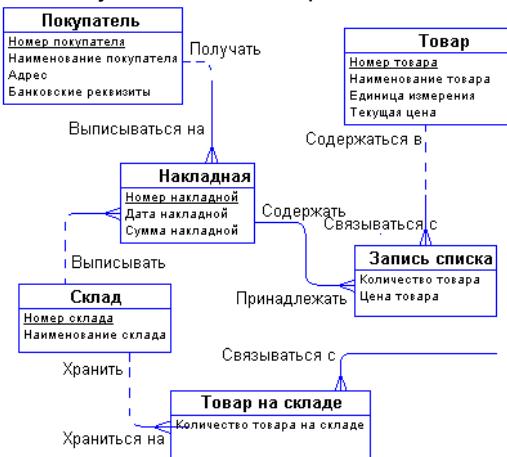
Справа налево: "Каждый ребенок обязан принадлежать ровно одному сотруднику".

Различают **концептуальные и физические** ЕР-диаграммы. Концептуальные диаграммы не учитывают особенностей конкретных СУБД. Физические диаграммы строятся по концептуальным и представляют собой прообраз конкретной базы данных. Сущности, определенные в концептуальной диаграмме становятся таблицами, атрибуты становятся колонками таблиц (при этом учитываются допустимые для данной СУБД типы данных и наименования столбцов), связи реализуются путем **миграции** ключевых атрибутов родительских сущностей и создания внешних ключей.

При правильном определении сущностей, полученные таблицы будут сразу находиться в ЗНФ. Основное достоинство

## БАЗЫ ДАННЫХ.

метода состоит в том, модель строится методом последовательных уточнений первоначальных диаграмм. Пример концептуальной ER-диаграммы:



## 4. Оператор **SELECT** языка SQL. Запросы на чтение из одной таблицы. Виды условий поиска.

Оператор **SELECT** является фактически самым важным для пользователя и самым сложным оператором SQL. Он предназначен для выборки данных из таблиц, т.е. он, собственно, и реализует одно из основных назначение базы данных - предоставлять информацию пользователю.

Оператор **SELECT** всегда выполняется над некоторыми таблицами, входящими в базу данных и его результатом всегда является таблица.

### **Синтаксис оператора выборки**

Оператор выборки ::=

Табличное выражение

[**ORDER BY**

{ {Имя столбца-результата [**ASC | DESC**] } | {Положительное целое [**ASC | DESC**] } },...];

Табличное выражение ::=

Select-выражение

[

{**UNION | INTERSECT | EXCEPT**} [**ALL**]

{Select-выражение | **TABLE** Имя таблицы | Конструктор значений}

```

БАЗЫ ДАННЫХ.
таблицы}
]
Select-выражение ::= 
SELECT [ALL | DISTINCT]
{{Скалярное выражение | Функция агрегирования | Select-
выражение} [AS Имя столбца]...}
| {{Имя таблицы|Имя корреляции}.*}
|
FROM {
{Имя таблицы [AS] [Имя корреляции] [(Имя столбца,...)]}
| {Select-выражение [AS] Имя корреляции [(Имя столбца,...)]}
| Соединенная таблица }...
[WHERE Условное выражение]
[GROUP BY {{Имя таблицы|Имя корреляции}.]Имя столбца}...]
[HAVING Условное выражение]

```

Select-выражение в разделе SELECT, используемое в качестве значения для отбираемого столбца, должно возвращать таблицу, состоящую из одной строки и одного столбца, т.е. скалярное выражение.

Условное выражение в разделе WHERE должно вычисляться для каждой строки, являющейся кандидатом в результирующее множество строк. В этом условном выражении можно использовать подзапросы. Синтаксис условных выражений, допустимых в разделе WHERE рассматривается ниже.

Раздел HAVING содержит условное выражение, вычисляемое для каждой группы, определяемой списком группировки в разделе GROUP BY. Это условное выражение может содержать функции агрегирования, вычисляемые для каждой группы.

Условное выражение, сформулированное в разделе WHERE, может быть перенесено в раздел HAVING. Перенос условий из раздела HAVING в раздел WHERE невозможен, если условное выражение содержит агрегатные функции. Перенос условий из раздела WHERE в раздел HAVING является плохим стилем программирования - эти разделы предназначены для различных по смыслу условий (условия для строк и условия для групп строк).

Если в разделе SELECT присутствуют агрегатные функции, то они вычисляются по-разному в зависимости от наличия раздела GROUP BY. Если раздел GROUP BY отсутствует, то результат запроса возвращает не более одной строки. Агрегатные функции вычисляются по всем строкам, удовлетворяющим условному выражению в разделе WHERE. Если раздел GROUP BY

## БАЗЫ ДАННЫХ.

присутствует, то агрегатные функции вычисляются по отдельности для каждой группы, определенной в разделе GROUP BY.

**Скалярное выражение** - в качестве скалярных выражений в разделе SELECT могут выступать либо имена столбцов таблиц, входящих в раздел FROM, либо простые функции, возвращающие скалярные значения.

Выборка из одной таблицы:

Простейший вид оператора SELECT для выборки данных из одной таблицы:

```
SELECT [DISTINCT] <список полей> FROM <таблица>  
WHERE <условие>
```

*Пример 1.* Выбрать все данные из таблицы поставщиков

(ключевые слова **SELECT... FROM...**):

```
SELECT * FROM P;
```

*Пример 2.* Выбрать все строки из таблицы поставщиков,

удовлетворяющих некоторому условию:

```
SELECT * FROM P WHERE P.PNUM > 2;
```

*Пример 3.* Выбрать некоторые колонки из исходной таблицы:

```
SELECT P.NAME FROM P;
```

*Пример 4.* Выбрать некоторые колонки из исходной таблицы,

удалив из результата повторяющиеся строки:

```
SELECT DISTINCT P.NAME FROM P;
```

*Пример 5.* Использование скалярных выражений и

переименований колонок в запросах:

```
SELECT TOVAR.TNAME, TOVAR.KOL, TOVAR.PRICE, "=" AS  
EQU, TOVAR.KOL * TOVAR.PRICE AS SUMMA FROM TOVAR;
```

В результате получим таблицу с колонками, которых не было в исходной таблице TOVAR:

TNAME	KOL	PRICE	EQU	SUMMA
Болт	10	100	=	1000
Гайка	20	200	=	4000

*Пример 6.* Упорядочение результатов запроса (ORDER BY):

```
SELECT PD.PNUM, PD.DNUM, PD.VOLUME FROM PD ORDER BY  
DNUM;
```

## БАЗЫ ДАННЫХ.

В результате получим следующую таблицу, упорядоченную по полю DNUM:

PNUM	DNUM	VOLUME
1	1	100
2	1	150
2	2	250
1	3	300

## **Виды условий поиска**

Выражение в разделе WHERE оператора SELECT может содержать следующие предикаты:

- Предикат **сравнения**

Пример. Сравнение поля таблицы и скалярного значения:  
POSTAV.VOLUME > 100

- Предикат **between**

Конструктор значений строки [NOT] BETWEEN

Конструктор значений строки AND Конструктор значений строки

Пример: PD.VOLUME BETWEEN 10 AND 100

- Предикат **in**

Конструктор значений строки [NOT] IN

{(Select-выражение) | (Выражение для вычисления значения,...)}

Пример: P.NUM IN (1, 2, 3)

- Предикат **like**

Выражение для вычисления значения строки-поиска  
[NOT] LIKE

Выражение для вычисления значения строки-шаблона  
[ESCAPE Символ]

Предикат LIKE производит поиск строки-поиска в строке-шаблоне.

- Предикат **null**

Конструктор значений строки IS [NOT] NULL

Предикат NULL применяется специально для проверки, не равно ли проверяемое выражение null-значению.

- Предикат **количественного сравнения**

Конструктор значений строки {= | < | > | <= | >= | <>}  
{ANY | SOME | ALL} (Select-выражение)

- Предикат **exist**

EXIST (Select-выражение)

## БАЗЫ ДАННЫХ.

- Предикат EXIST возвращает значение TRUE, если результат подзапроса (select-выражения) не пуст.
- Предикат **unique**  
**UNIQUE** (*Select-выражение*)  
Предикат UNIQUE возвращает TRUE, если в результате подзапроса (select-выражения) нет совпадающих строк.
- Предикат **match**  
**Конструктор значений строки MATCH [UNIQUE] [PARTIAL | FULL] (Select-выражение)**  
Предикат MATCH проверяет, будет ли значение, определенное в конструкторе строки совпадать со значением любой строки, полученной в результате подзапроса.
- Предикат **overlaps**  
**Конструктор значений строки OVERLAPS**  
**Конструктор значений строки**  
Предикат OVERLAPS, является специализированным предикатом, позволяющим определить, будет ли указанный период времени перекрывать другой период времени.

## **5. Многотабличные запросы SQL. Внутренние соединения.**

Запросы могут выбирать данные из нескольких таблиц. Эти таблицы должны быть перечислены после слова FROM. Если таблицы не связаны между собой, то результатом запроса будут всевозможные комбинации (декартово произведение) записей отдельных таблиц, что не имеет практического смысла.

*Пример:* Рассмотрим отношения по поставщикам S (S#, Sn, Scity), изделиям P (P#, Pn, Pcity, W) и поставкам SP (S#, P#, Q). Здесь S# - номер поставщика, Sn – его имя, Scity – место проживания, P# - номер изделия, Pn – его наименование, Pcity – место хранения, W – вес, Q – объем поставки.

Пусть требуется получить список поставок поставщиков из Москвы с указанием имени поставщика. Запрос

```
SELECT S.S#, Sn, P#, Q FROM S, SP WHERE Scity  
='Москва'
```

присоединит к каждому поставщику из Москвы все поставки из таблицы SP независимо от места проживания поставщика, что, очевидно, не соответствует заданию. Правильным решением является запрос

```
SELECT S.S#, Sn, P#, Q FROM S, SP WHERE Scity  
='Москва' AND S.S#=SP.S#
```

Условие S.S#=SP.S# обеспечивает связь таблиц S и SP, что соответствует операции соединения реляционной алгебры. Если

## БАЗЫ ДАННЫХ.

атрибут присутствует более чем в одной таблице, сначала указывается имя таблицы, а затем после точки имя атрибута.

Как и в реляционной алгебре, возможно и  $\theta$ -соединение таблиц, где  $\theta$  задает знак операции сравнения ('<', '>' и т. п.), но такой вариант соединения редко применяется на практике. Для соединений таблиц имеются и другие синтаксические конструкции, которые будут описаны ниже. Рассмотрим еще два примера.

*Примеры:*

1. Найти список поставщиков из Казани, поставляющих изделие с номером P2

```
SELECT Sname FROM S, SP WHERE S.S#=SP.S# AND Scity  
='Казань' AND P#=P1
```

Здесь выбирается поле Sname из таблицы S, но в качестве источника данных указывается и таблица SP, поскольку условия выборки и связи используют поля этой таблицы.

2. Получить список поставок объема более 100 единиц с указанием имен поставщиков и наименований деталей

```
SELECT Sname, Pname, Q FROM S, P, SP WHERE S.S#=SP.S#  
AND P.P#=SP.P# AND
```

$Q > 100$

Соединение таблиц может быть и по нескольким атрибутам. Например, деталь в разных таблицах может идентифицироваться двумя атрибутами совместно: номером изделия и номером детали в изделии.

Таблица может соединяться и сама с собой. Рассмотрим для примера таблицу сотрудников и их непосредственных руководителей S (S#, Sname, Shef#), где S# - номер сотрудника, Sname – его имя, Shef# - номер руководителя. Пусть требуется дать полный список имен сотрудников и их руководителей.

Если бы существовали две копии этой таблицы S1 и S2, то решением был бы запрос

```
SELECT S1.Sname, S2.Sname FROM S1, S2 WHERE S1.Shef# =  
S2.S#
```

А как обойтись без копирования содержимого таблицы S?

Для подобных случаев в SQL введены локальные псевдонимы или алиасы таблиц. Псевдоним действует в пределах запроса и позволяет обращаться к одной и той же таблице, как к двум разным таблицам. При использовании псевдонимов S1 и S2 в нашем случае вид запроса почти не изменится

```
SELECT S1.Sname, S2.Sname FROM S1, S2 WHERE S1.Shef#  
= S2.S#
```

## **Внутренние и внешние соединения.**

Во фразе FROM можно использовать следующие операторы соединений:

- CROSS JOIN - перекрестное соединение.
- NATURAL JOIN - естественное соединение. Стандарт SQL определяет это соединение как результат объединения таблиц по всем одноименным столбцам. Естественное соединение может быть следующих типов:
  - INNER JOIN - внутреннее соединение, используется по умолчанию.
  - LEFT JOIN [OUTER] - левое внешнее соединение.
  - RIGHT JOIN [OUTER] - правое внешнее соединение.
  - FULL JOIN [OUTER] - полное внешнее соединение.
  - UNION JOIN - соединение объединения.

При внутреннем естественном соединении группируются только те строки, значения которых по соединяемым (одноименным) столбцам совпадают, т.е. которые соответствуют условию, указанному после ключевого слова ON.

При внешнем левом соединении в результирующий набор будут выбраны все строки из левой таблицы (указываемой первой). При совпадении значений по соединяемым (одноименным) столбцам значения второй таблицы заносятся в результирующий набор в соответствующие строки. При отсутствии совпадений в качестве значений второй таблицы проставляется значение NULL.

При внешнем правом соединении в результирующий набор будут выбраны все строки из правой таблицы (указываемой второй). При совпадении значений по соединяемым (одноименным) столбцам значения первой таблицы заносятся в результирующий набор в соответствующие строки. При отсутствии совпадений в качестве значений первой таблицы проставляется значение NULL.

При полном внешнем соединении в результирующий набор будут выбраны все строки - как из правой, так и из левой таблицы. При совпадении значений по соединяемым (одноименным) столбцам строка содержит значения как из левой, так и из правой таблицы. В противном случае, вместо отсутствующих значений в столбцы таблицы (левой или правой) заносится значение NULL.

## БАЗЫ ДАННЫХ.

Фраза USING позволяет выполнить естественное соединение по указываемым столбцам, что, в свою очередь, позволяет соединять таблицы, имеющие несколько одноименных столбцов, нужным образом (по одному или двум столбцам). Список столбцов, по которым выполняется соединение, указывается после фразы USING.

*Например:*

SELECT t1.f1, t1.f2, t2.f1 FROM tbl1 t1 JOIN tbl2 t2 USING f2;

Естественное соединение по указываемому предикату выполняется с помощью фразы ON. В результирующий набор выбираются строки, удовлетворяющие заданному условию. Этот способ соединения аналогичен соединению по предикату, указываемому фразой WHERE.

Например: SELECT t1.f1, t1.f2, t2.f1, t2.f2 FROM tbl1 t1 JOIN tbl2 t2 ON t1.f1 = t2.f2;

Для наглядности приведем две таблицы driver(id, ФИО водителя и айдишник закрепленным за ним автомобилем) и avto(id машины, марка)

Внутреннее естественное соединение(INNER JOIN)

Выведем список водителей с закрепленными за ними авто.

SELECT d.name, a.marka FROM driver d INNER JOIN avto a ON d.id\_avto = a.id

Внешнее левое соединение(LEFT OUTER JOIN)

Получим данные по всем водителям, будут все люди работающие в фирме с автомобилями и без

SELECT d.name, a.marka FROM driver d LEFT OUTER JOIN avto a ON d.id\_avto = a.id

Если нам нужно посмотреть водителей без авто можно воспользоваться запросом типа

SELECT d.name, a.marka FROM driver d LEFT OUTER JOIN avto a ON d.id\_avto = a.id WHERE a.marka IS null

Внешнее правое соединение(RIGHT OUTER JOIN)

Необходимы данные по всем машинам находящихся в парке для списка на техосмотр и в случае проблемы данные водителя чтобы к нему обратиться. У авто которая простоявает без водителя поле водитель будет NULL.

SELECT d.name, a.marka FROM driver d RIGHT OUTER JOIN avto a ON d.id\_avto = a.id

Полное внешнее соединение(FULL (OUTER))

## БАЗЫ ДАННЫХ.

Необходимо вывести весь штат техники и водителей в одном отчете.

SELECT d.name, a.marka FROM driver d full OUTER JOIN avto a ON d.id\_avto = a.id

Вывести свободные авто и незанятых водителей например:

SELECT d.name, a.marka FROM driver d full OUTER JOIN avto a ON d.id\_avto = a.id WHERE d.id\_avto IS null OR a.id IS null

Перекрестное соединение(CROSS JOIN)

Необходимо составить все возможные варианты объединения двух множеств, в нашем примере увидеть у кого имени сочетается с маркой машины:

SELECT d.name, a.marka FROM driver d cross join avto a

Естественное соединение (NATURAL JOIN)

SELECT d.name, a.marka FROM driver d NATURAL JOIN avto a  
объединение произошло по полю id и id\_avto

Соединение объединения (UNION JOIN)

Получить автомобили без водителей и водителей без авто  
SELECT \* FROM driver d UNION JOIN avto a ON d.id\_avto = a.id

**6. Псевдонимы. Внешнее соединение таблиц в SQL.**

См. пункт 5. "Многотабличные запросы SQL. Внутренние соединения".

**7. Запросы с группировкой и вложенные запросы в SQL.**

Группировка данных в операторе SELECT осуществляется с помощью ключевых слов GROUP BY и HAVING.

GROUP BY неразрывно связано с агрегирующими функциями, без них оно практически не используется. GROUP BY разделяет таблицу на группы, а агрегирующая функция вычисляет для каждой из них итоговое значение.

*Пример.* Определить количество книг каждого издательства:

```
SELECT publishers.publisher, count(titles.title)
FROM titles,publishers WHERE titles.pub_id=publishers.pub_id
GROUP BY publisher;
```

Ключевое слово HAVING работает следующим образом: сначала GROUP BY разбивает строки на группы, затем на полученные наборы накладываются условия HAVING.

*Пример.* УстраниТЬ из предыдущего запроса те

издательства, которые имеют только одну книгу:

```
SELECT publishers.publisher, count(titles.title) FROM
titles,publishers
```

## БАЗЫ ДАННЫХ.

```
WHERE titles.pub_id=publishers.pub_id GROUP BY publisher  
HAVING COUNT(*)>1;
```

## **Вложенные запросы.**

**Вложенный подзапрос** - это подзапрос, заключенный в круглые скобки и вложенный в WHERE (HAVING) фразу предложения SELECT или других предложений, использующих WHERE фразу. Вложенный подзапрос может содержать в своей WHERE (HAVING) фразе другой вложенный подзапрос и т.д. Вложенный подзапрос создан для того, чтобы при отборе строк таблицы, сформированной основным запросом, можно было использовать данные из других таблиц (например, при отборе блюд для меню использовать данные о наличии продуктов в кладовой пансионата).

Существуют **простые и коррелированные** вложенные подзапросы. Они включаются в WHERE (HAVING) фразу с помощью условий IN, EXISTS или одного из условий сравнения (= | <> | < | <= | > | >= ).

**Простые вложенные подзапросы** обрабатываются системой "снизу вверх". Первым обрабатывается вложенный подзапрос самого нижнего уровня. Множество значений, полученное в результате его выполнения, используется при реализации подзапроса более высокого уровня и т.д.

*Пример.* Выдать название и статус поставщиков продукта с номером 11:

```
SELECT Название, Статус FROM Поставщики WHERE ПС IN  
(SELECT ПС FROM Поставки WHERE ПР = 11);
```

Результат:

Название	Статус
СЫТНЫЙ	рынок
УРОЖАЙ	коопторг

**Запросы с коррелированными вложенными подзапросами** обрабатываются системой в обратном порядке. Сначала выбирается первая строка рабочей таблицы, сформированной основным запросом, и из нее выбираются значения тех столбцов, которые используются во вложенном подзапросе (вложенных подзапросах). Если эти значения удовлетворяют условиям вложенного подзапроса, то выбранная строка включается в результат. Затем выбирается вторая строка и т.д., пока в результат не будут включены все строки, удовлетворяющие вложенному подзапросу (последовательности вложенных подзапросов).

## 8. Целостность данных. Транзакции.

Термин “целостность данных” относится к правильности и полноте информации, содержащейся в БД. Вероятно, корректнее говорить о непротиворечивости данных, поскольку невозможно предотвратить появление неверных данных. При изменении содержимого БД с помощью операторов INSERT, DELETE, UPDATE может произойти нарушение целостности данных.

Например, могут возникнуть следующие ситуации:

- в базу внесены заведомо неправильные данные (отрицательная цена товара);
- данные оказались несогласованными (заказ билета на несуществующий рейс);
- внесенные изменения потеряны из-за системной ошибки или сбоя в электропитании;
- изменения внесены частично (учтена выручка от продажи товара, но количество товара на складе не уменьшено);
- изменена нумерация домов на улице, но остались старые адреса жильцов.

Одной из важнейших задач реляционной СУБД является поддержка целостности данных.

Для обеспечения непротиворечивости данных в реляционных СУБД определяются условия целостности данных.  
**Виды условий целостности данных.**

1. **Обязательность данных.** Некоторые столбцы должны содержать значения в каждой строке, то есть иметь NULL-значения и не оставаться незаполненными. Например, для каждой поставки должен быть указан поставщик. Условие обязательности данных задается в операторе создания таблицы CREATE TABLE.
2. **Проверка на правильность.** Можно, например, указать СУБД, что значения поля должны находиться в определенном диапазоне. Это условие также задается в операторе CREATE TABLE.
3. **Целостность таблицы.** Первичный ключ таблицы должен в каждой строке иметь уникальное значение. Свойство уникальности может быть задано и для других полей. Условие задается в операторе CREATE TABLE.
4. **Ссылочная целостность.** Каждая строка таблицы-потомка с помощью внешнего ключа связана со строкой таблицы-предка, содержащей первичный ключ, значение которого равно значению внешнего ключа.
5. **Деловые правила.** Изменение информации может быть ограничено деловыми правилами, учитывающими

## БАЗЫ ДАННЫХ.

специфику БД. Например, компания может установить правило, запрещающее принимать заказ на товар, если его недостаточно на складе. Деловые правила могут быть реализованы с помощью триггеров, которые будут описаны ниже.

6. **Непротиворечивость.** Многие операции в БД вызывают несколько изменений одновременно. Например, регистрация продажи товара может потребовать обновления таких полей разных таблиц, как общий доход, количество товара на складе, объем продаж определенного продавца и т. п. Если эти изменения выполнены не в полном объеме, целостность данных нарушается. Условия непротиворечивости данных обеспечивает механизм транзакций.

Существует четыре типа изменений БД, которые могут нарушить ссылочную целостность отношений предок-потомок.

1. Добавление новой строки-потомка. Значение внешнего ключа строки-потомка должно быть одним из значений первичного ключа в таблице- предке.
2. Обновление внешнего ключа в строке-потомке. Новое значение внешнего ключа строки-потомка должно быть одним из значений первичного ключа в таблице-предке.
3. Удаление строки-предка. Если из таблицы предка будет удалена строка, у которой есть хотя бы один строки-потомки, то значения внешних ключей в этих строках не будут равны ни одному из значений первичного ключа в таблице-предке.
4. Обновление первичного ключа в строке-предке - все аналогично предыдущему случаю.

Первые две ситуации приводят к сообщению об ошибке. Для двух последних ситуаций в операторе CREATE TABLE можно задать одно из четырех возможных правил

1. RESTRICT – запрещает удаление строки из таблицы- предка или корректировку первичного ключа, если строка имеет потомков.
2. CASCADE – при удалении строки-предка все строки-потомки также удаляются из таблицы-потомка. При обновлении первичного ключа в строке-предке значение внешнего ключа обновляется во всех строках-потомках.
3. SET NULL – при удалении или корректировке первичного ключа строки-предка внешним ключам во всех строках-потомках присваивается NULL-значение.

## БАЗЫ ДАННЫХ.

4. SET DEFAULT - при удалении или корректировке первичного ключа строки-предка внешним ключам во всех строках-потомках присваивается значение по умолчанию.

Правила удаления и обновления могут быть разными для одной и той же пары таблицы-предка и таблицы потомка. Пусть, например, в таблице предка содержится информация о читателях библиотеки, а в таблице потомке – данные о взятых ими книгах. При выписке читателя нельзя забывать о книгах на его руках, поэтому для удаления разумно определить правило RESTRICT. При смене номера читательского билета, являющегося первичным ключом в таблице читателей, можно автоматически поменять этот номер в таблице выдач, то есть определить правило на обновление CASCADE.

Каскадные удаления и обновления могут приводить к непредвиденным последствиям. Рассмотрим, например три таблицы А - животные, В – девочки и С – мальчики с одинаковыми полями Name и Likes (любит). Будем считать, что животные любят мальчиков, мальчики – девочек, а девочки – животных. Таким образом, каждая из таблиц является как родительской, так и дочерней. Таблица А является родительской для В и дочерней для С, а В – родительской для С и дочерней для А.

Пусть таблицы имеют следующее содержание.

Записи таблицы А		Записи таблицы В		Записи таблицы С	
Name	Likes	Name	Likes	Name	Likes
Джек	Леша	Таня	Джек	Петя	Гля
Мурка	Петя	Гая	Хрюша	Леша	Лена
Хрюша	Толя	Лена	Мурка	Толя	Таня
				Стас	Гая

Предположим, что все правила удаления CASCADE. Удалим запись с данными о Пете из таблицы С. Тогда из таблицы А каскадно удалится запись с информацией о Мурке, потом не останется Лены в таблице В и т. д. Легко проследить, что в итоге произойдет полное удаление содержимого всех таблиц. Часть внешнего ключа может иметь значения NULL. По стандарту SQL92 в операторе CREATE TABLE можно указать один из двух режимов

- MATCH FULL (полное совпадение) – ни одна часть внешнего ключа не может иметь значение NULL;

## БАЗЫ ДАННЫХ.

- MATCH PARTIAL (частичное совпадение) – часть внешнего ключа может принимать значение NULL.

## **Транзакции.**

Под транзакцией понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации) такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в БД, либо воздействие всех этих операторов полностью отсутствует. Лозунг транзакции - "Все или ничего": при завершении транзакции оператором COMMIT результаты гарантированно фиксируются во внешней памяти (смысл слова commit - "зафиксировать" результаты транзакции); при завершении транзакции оператором ROLLBACK результаты гарантированно отсутствуют во внешней памяти (смысл слова rollback - ликвидировать результаты транзакции).

Понятие транзакции имеет непосредственную связь с понятием целостности БД. Очень часто БД может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения БД.

Поэтому для поддержания подобных ограничений целостности допускается их нарушение внутри транзакции с тем условием, чтобы к моменту завершения транзакции условия целостности были соблюдены.

Различаются два вида ограничений целостности:

- немедленно проверяемые;
- откладываемые.

**К немедленно проверяемым** ограничениям целостности относятся такие ограничения, проверку которых бессмысленно или даже невозможно откладывать. К таким ограничениям относятся ограничения домена (пример: возраст сотрудника не может превышать 150 лет). Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор.

**Откладываемые ограничения** целостности - это ограничения на базу данных, а не на какие-либо отдельные операции. По умолчанию такие ограничения проверяются при конце транзакции, и их нарушение вызывает автоматическую замену оператора COMMIT на оператор ROLLBACK. Однако в некоторых системах поддерживается специальный оператор насильтственной проверки ограничений целостности внутри транзакции.

## БАЗЫ ДАННЫХ.

**Изолированность пользователей.** В многопользовательских системах с одной базой данных одновременно могут работать несколько пользователей или прикладных программ. Предельной задачей системы является обеспечение изолированности пользователей, т.е. создание достоверной и надежной иллюзии того, что каждый из пользователей работает с БД в одиночку.

В связи со свойством сохранения целостности БД транзакции являются подходящими единицами изолированности пользователей.

## **9. Представления и работа с ними. Триггеры и хранимые процедуры.**

Механизм представлений (view) является мощным средством языка SQL, позволяющим скрыть реальную структуру БД от некоторых пользователей за счет определения представления БД, которое реально является некоторым хранимым в БД запросом с именованными столбцами, а для пользователя ничем не отличается от базовой таблицы БД (с учетом технических ограничений). Любая реализация должна гарантировать, что состояние представляющей таблицы точно соответствует состоянию базовых таблиц, на которых определено представление. Обычно вычисление представляющей таблицы (материализация соответствующего запроса) производится каждый раз при использовании представления.

В стандарте SQL/89 оператор определения представления имеет следующий синтаксис:

```
<view definition> ::=  
    CREATE VIEW <table name> [(<view column list>)]  
        AS <query specification> [WITH CHECK OPTION]  
    <view column list> ::= <column name> [{,<column name>}...]
```

Определяемая представляемая таблица V является изменяемой (т.е. по отношению к V можно использовать операторы DELETE и UPDATE) в том и только в том случае, если выполняются следующие условия для спецификации запроса:

- В списке выборки не указано ключевое слово DISTINCT;
- Каждое арифметическое выражение в списке выборки представляет собой одну спецификацию столбца, и спецификация одного столбца не появляется более одного раза;
- В разделе FROM указана только одна таблица, являющаяся либо базовой таблицей, либо изменяемой представляющей таблицей;

## БАЗЫ ДАННЫХ.

- В условии выборки раздела WHERE не используются подзапросы;
- В табличном выражении отсутствуют разделы GROUP BY и HAVING.

Если в списке выборки спецификации запроса имеется хотя бы одно арифметическое выражение, состоящее не из одной спецификации столбца, или если одно имя столбца участвует в списке выборки более одного раза, определение представления должно содержать список имен столбцов представляющей таблицы (т.е. нужно явно именовать столбцы представляющей таблицы, если эти имена не наследуются от столбцов таблиц раздела FROM спецификации запроса).

**Хранимые процедуры** - это процедуры и функции, хранящиеся непосредственно в базе данных в откомпилированном виде и которые могут запускаться пользователями или приложениями, работающими с базой данных. Хранимые процедуры обычно пишутся либо на специальном процедурном расширении языка SQL. Основное назначение хранимых процедур - реализация бизнес-процессов предметной области.

Хранимые процедуры позволяют повысить производительность, расширяют возможности программирования и поддерживают функции безопасности данных.

Вместо хранения часто используемого запроса, клиенты могут ссылаться на соответствующую хранимую процедуру. При вызове хранимой процедуры ее содержимое сразу же обрабатывается сервером.

Кроме собственно выполнения запроса, хранимые процедуры позволяют также производить вычисления и манипуляцию данными - изменение, удаление, выполнять DDL-операторы и вызывать другие хранимые процедуры, выполнять сложную транзакционную логику. Один-единственный оператор позволяет вызвать сложный сценарий, который содержится в хранимой процедуре, что позволяет избежать пересылки через сеть сотен команд и, в особенности, необходимости передачи больших объемов данных с клиента на сервер.

**Триггеры** - это хранимые процедуры, связанные с некоторыми событиями, происходящими во время работы базы данных. В качестве таких событий выступают операции вставки, обновления и удаления строк таблиц. Если в базе данных определен некоторый триггер, то он запускается *автоматически*

## БАЗЫ ДАННЫХ.

всегда при возникновении события, с которым этот триггер связан. Очень важным является то, что пользователь не может обойти триггер. Триггер срабатывает независимо от того, кто из пользователей и каким способом инициировал событие, вызвавшее запуск триггера. Таким образом, основное назначение триггеров - автоматическая поддержка целостности базы данных. Триггеры могут быть как достаточно простыми, например, поддерживающими ссылочную целостность, так и довольно сложными, реализующими какие-либо сложные ограничения предметной области или сложные действия, которые должны произойти при наступлении некоторых событий. Например, с операцией вставки нового товара в накладную может быть связан триггер, который выполняет следующие действия: проверяет, есть ли необходимое количество товара, при наличии товара добавляет его в накладную и уменьшает данные о наличии товара на складе, при отсутствии товара формирует заказ на поставку недостающего товара и тут же посыпает заказ по электронной почте поставщику.

Кроме того, триггеры могут быть привязаны не к таблице, а к представлению (VIEW). В этом случае с их помощью реализуется механизм «обновляемого представления». В этом случае ключевые слова BEFORE и AFTER влияют лишь на последовательность вызова триггеров, так как собственно событие (удаление, вставка или обновление) не происходит.

В некоторых серверах триггеры могут вызываться не для каждой модифицируемой записи, а один раз на изменение таблицы. Такие триггеры называются табличными.

*Пример (MS SQL?):*

```
Rem Табличный
CREATE OR REPLACE TRIGGER tr2
AFTER UPDATE ON rayon
BEGIN
    insert into info values ('table "rayon" has changed');
END;
```

Для отличия табличных триггеров от строчных вводится дополнительные ключевые слова при описании строчных триггеров. В MS SQL (?) это словосочетание FOR EACH ROW.

*Пример:*

```
Rem Строчный
CREATE OR REPLACE TRIGGER tr1
AFTER UPDATE ON rayon FOR EACH ROW
BEGIN
```

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

```
insert into info values ('one string in table "rayon" has
changed');
END;
```

Очевидно, что чем больше программного кода в виде триггеров и хранимых процедур содержит база данных, тем сложнее ее разработка и дальнейшее сопровождение.

### **III. ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.**

#### **1. Технология программирования. Система разработки ПО.**

**Технология программирования** - это совокупность методов и средств, используемых в процессе разработки ПО, она включает:

- указания последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется операция;
- описание самих операций (исходные данные, результаты, методы оценки);
- способ описания проектируемой системы.

Сходство между разработкой программного продукта и другими областями инженерной науки необходимость подробного описания того что должно быть сделано - анализ требований Отличие - программные проекты особенно часто подвергаются изменениям, пока продукт находится еще на стадии разработки.

Две тенденции в области разработки ПО в последние два десятилетия:

1. Быстрый рост количества приложений,
2. Появление большого числа подходов-технологий, инструм. средств.

Только хорошо организованные коллективы программистов, владеющие методами разработки ПО, способны правильно построить работу над проектом.

Система разработки ПО включает в себя (четыре «п»):

- 1. Персонал (кто делает)**
- 2. Процесс (способ которым делается)**
3. Выбор модели разработки (Водопадная, Итерационная, Спиральная) часто делается на высшем уровне руководства компанией - важной проблемой становится оценка возможностей коллектива разработчиков.
4. Системы принципов разработки – основа профессиональной деятельности разработчиков ПО в 21 веке:

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Индивидуальный процесс разработки(PSP)Хамфри,
  - Командный(TSP),
  - Модель зрелости возможностей(CMM Capability Maturity Model).
5. Хорошо продуманные стандарты документации облегчают создание ПО, направляют процесс его создания.
6. **Проект (создание этого)** - совокупность действий, необходимая для создания продукта. Проект включает: контакт с заказчиком, написание документации, проектирование, написание кода, тестирование.
7. Некоторые аспекты проекта:
- 8. - ООП
  - 9. - Проектирование слоев
  - 10. - UML- это не методология это система обозначений.
  - 11. - Унаследованные системы.
12. **Продукт (что делается)** – В соответствие с унифицированным процессом разработки продукт это не только программа, но и все составляющие его:
- Спецификация требований к ПО;
  - Проектная модель;
  - Исходный и объектный код;
  - Тестовые процедуры и тестовые варианты
  - Документация
13. Будем называть эти составляющие – артефактами.
- Чаще всего говорят о трех основных ограничениях («железный треугольник») проекта **Содержание, Время, Стоимость**. В приложении к ПО добавляют четвертое ограничение – качество (приемлемое качество).
- Качество (с точки зрения заказчика или пользователя продукта)** - это пригодность к использованию. Делает ли данный продукт то, в чем нуждается пользователь, облегчает ли он работу, можно ли его использовать так, как удобно пользователю.
- Качество (с точки зрения разработчика)** – это соответствие специфицированным и собранным требованиям делает ли данный продукт все то, что указано в требованиях.
- Качество** это соответствие реальным требованиям, явным и неявным. Очень часто неявные требования настолько очевидны для заказчика или пользователя, что он даже не предполагает, что они неизвестны разработчикам. Для примера рассмотрим автомобиль – заказчик может детально описать требования к дизайну, параметрам двигателя, оформлению салона, цвету

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

кузова, но нигде не указать, что шины должны быть круглыми, а лобовое стекло – прозрачным.

Заказчик будет удовлетворен только тогда, когда купленный продукт будет полностью удовлетворять его реальным и жизненным требованиям, как специфицированным, так и нет.

Вспомним рассмотренные во введении **четыре «П»** разработки программ. Цель любого программного проекта состоит в производстве некоторого программного продукта (например, текстового редактора). То, как в рамках проекта производится продукт, представляет собой процесс. Поскольку критичным для успеха дела является взаимодействие членов команды, мы включаем в рассмотрение персонал.



В современной практике разработки программного обеспечения существует пять ключевых требований, в основном сформулированных Хэмфри. Первое из них — заранее выбрать свою шкалу измерения качества для проекта и продукта. Например, «500 строк полностью протестированного программного кода, написанного одним человеком за месяц» или «не более трех ошибок на тысячу строк программного кода». Второе требование состоит в сборе информации по всем проектам с целью создания базы для оценки будущих проектов. Третье положение гласит, что все требования, схемы, программные коды и материалы тестирования должны быть легко доступны всем членам команды. Суть четвертого условия состоит в том, что все участники команды должны следовать избранному

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

процессу разработки. Пятое – оценка достижения уровня качества. Роль четырех «П» в достижении пяти ключевых требований демонстрирует рисунок. При этом подчеркивается тот факт, что ничто не может быть достигнуто без участия членов команды, работающей над проектом.

### **2. Процесс разработки программного обеспечения**

Проектирование программного обеспечения представляет собой процесс построения приложений реальных размеров и практической значимости, удовлетворяющих заданным требованиям функциональности и производительности, таких, например, как текстовый редактор, электронная таблица, операционная система или, скажем, программа контроля неисправностей космической станции. Программирование — это один из видов деятельности, входящих в цикл разработки программного обеспечения.

Хорошие разработчики программного обеспечения избегают повторения ошибок предыдущих проектов за счет документирования и совершенствования своего процесса разработки.

По масштабам работы, требуемым профессиональным знаниям и общественной значимости, различие между просто программированием и проектированием программного обеспечения можно сравнить с различием между изготовлением скамейки у ворот своего загородного дома и возведением моста. Эти две задачи различаются на порядок по значимости и требуемым профессиональным знаниям. В отличие от постройки скамейки возведение моста включает в себя как профессиональную, так и социальную ответственность. Технология разработки программного обеспечения должна охватывать разнообразные типы программ, включая перечисленные ниже.

- ◆ Автономное:
  - устанавливаемое на одиночный компьютер;
  - не связанное с другим программным и аппаратным обеспечением;
- 14. пример — текстовый редактор.
- ◆ Встроенное:
  - часть уникального приложения с привлечением аппаратного обеспечения;
- пример — автомобильный контроллер.
- ◆ Реального времени:

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

должны выполнять функции в течение малого интервала времени, обычно нескольких микросекунд;

пример — программное обеспечение радиолокатора.

◆ Сетевое:

состоит из частей, взаимодействующих через сеть;

пример — основанная на веб-технологии видеоигра.

Принципы применимы ко всем этим типам.

Стандартная последовательность шагов такова:

1. Понять природу и сферу применения продукта. Бизнес-моделирование - описывается деятельность компании и определяются требования к системе — те подпроцессы и операции, которые подлежат автоматизации в разрабатываемой системе. Используется SADT, RUP: диаграмма деятельности. Это кажется очевидным, однако для того, чтобы понять, чего хочет заказчик, требуется ощутимое время, особенно если заказчик сам не знает достаточно хорошо, чего он в действительности хочет. Нужно составить представление о масштабах проекта и с этой целью оценить, какими сроками, финансами и персоналом мы располагаем. Если, например, для построения видеоигры нам предоставляется 10 тыс. долларов, один разработчик и один месяц срока, можно говорить разве что о реализации прототипа игры. Если же мы располагаем бюджетом в 5 млн долларов, 20 разработчиками и сроком в 18 месяцев, то речь уже может идти о создании полноценного конкурентоспособного программного продукта и совсем других масштабах производственной деятельности.

2. Анализ требований (Requirements Engineering) — это процесс сбора требований к системе, их систематизации, документирования, анализа, выявления противоречий, неполноты, разрешения конфликтов.

Требование — это потребность или ожидание, которое

1 - установлено в явном виде (например, заказчиком)

2 - наследуется, как обязательное из других систем требований (ГОСТа)

3 - подразумевается (например, выход по ESC) Описание системы

### 3. Проектирование ПО. Разработка архитектуры.

Архитектура ПО - это представление системы ПО, процесс и дисциплина, посвященные эффективной разработке проекта такой системы. Это представление, потому что оно дает информацию о компонентах составляющих систему, о взаимосвязях между этими компонентами и правилах регламентирующих эти взаимосвязи. Это процесс, потому что

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

существует предписанная последовательность действий для создания или изменения архитектуры системы, и/или проекта системы по этой архитектуре, с учетом множества ограничений. Это дисциплина, т.к. область знания или общие принципы (программной) архитектуры призваны донести до практика наиболее эффективный способ проектирования системы с учетом данных ограничений.

**4. Кодирование** – реализация одного или нескольких взаимосвязанных алгоритмов на некотором языке программирования.

**5. Тестирование** (частей ПО → интеграция и тест-е системы в целом) - процесс, позволяющий определить корректность, полноту и качество разработанного ПО; процесс формальной проверки или верификации может доказать, что дефекты отсутствуют, с точки зрения используемого метода.

В зависимости от используемого процесса разработки (модели разработки, см вопрос 1 по трпо) шаги 2 - 5 могут быть выполнены несколько раз.

**6. Документирование** - это документы, сопровождающие некоторое ПО архитектурная/проектная -обзор программного обеспечения, включающий описание рабочей среды и принципов, которые должны быть использованы при создании ПО; техническая - документация на код, алгоритмы, интерфейсы, API; пользовательская - руководства для конечных пользователей, администраторов системы и другого персонала; маркетинговая

**7. Внедрение** - процесс настройки программного обеспечения под определенные условия использования, а также обучения пользователей работе с программным продуктом

**8. Сопровождение** - процесс улучшения, оптимизации и устранения дефектов (ПО) после передачи в эксплуатацию. Может потребовать до 80 % ресурсов, потребовавшихся на разработку.

## **3. Проектирование, как составляющая Система разработки ПО**

Проектирование программного обеспечения — процесс создания проекта программного обеспечения (ПО), а также дисциплина, изучающая методы проектирования.

Проектирование подразумевает выработку свойств системы на основе анализа постановки задачи, а именно: моделей предметной области, требований к ПО, а также опыта проектировщика. Модель предметной области накладывает ограничения на бизнес-логику и структуры данных. Требования к ПО определяют внешние (видимые) свойства программы,

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

рассматриваемой как чёрный ящик. Определению внутренних свойств системы и детализации её внешних свойств собственно и посвящено проектирование. Проектирование ПО является частным случаем Проектирования продуктов и Проектирования систем.

В зависимости от создаваемого ПО, процесс проектирования может обеспечиваться как «ручным» проектированием, так и различными средствами его автоматизации. В процессе проектирования ПО для выражения его характеристик используются различные нотации — блок-схемы, ER-диаграммы, UML-диаграммы, DFD-диаграммы, а также макеты. (см соотв. вопрос по трпо)

Проектированию обычно подлежат:

- \* Архитектура ПО
- \* Устройство компонентов ПО
- \* Пользовательские интерфейсы

В российской практике результат проектирования представляется в виде комплекса документов под названием «Эскизный проект», «Технический проект», в зарубежной — Software Architecture Document, Software Design Document.

Архитектура ПО - это представление системы ПО, процесс и дисциплина, посвященные эффективной разработке проекта такой системы. Это представление, потому что оно дает информацию о компонентах составляющих систему, о взаимосвязях между этими компонентами и правилах регламентирующих эти взаимосвязи. Это процесс, потому что существует предписанная последовательность действий для создания или изменения архитектуры системы, и/или проекта системы по этой архитектуре, с учетом множества ограничений. Это дисциплина, т.к. область знания или общие принципы (программной) архитектуры призваны донести до практика наиболее эффективный способ проектирования системы с учетом данных ограничений.

Проектирование охватывает три основные области:

- 1.проектирование объектов данных;
- 2.проектирование программ, экранных форм, отчетов, которые будут обеспечивать выполнение запросов к данным;
- 3.учет конкретной среды или технологии, а именно: топологии сети, конфигурации аппаратных средств, используемой архитектуры (файл-сервер или клиент-сервер), параллельной и распределенной обработки и т.п.

Проектирование всегда начинается с определения цели проекта (в общем виде - решение ряда взаимосвязанных задач,

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

включающих в себя обеспечение на момент запуска системы и в течение всего времени ее эксплуатации):

- требуемой функциональности системы и уровня ее адаптивности к изменяющимся условиям функционирования;
- требуемой пропускной способности системы;
- требуемого времени реакции системы на запрос;
- безотказной работы системы;
- необходимого уровня безопасности;
- простоты эксплуатации и поддержки системы.

На этапе проектирования прежде всего формируются логическая и физическая модели данных. Проектировщики в качестве исходной информации получают результаты анализа. Полученная в процессе анализа информационная модель сначала преобразуется в логическую, а затем в физическую модель данных. Паралельно с проектированием схемы БД выполняется проектирование процессов, чтобы получить спецификации (описания) всех модулей ИС. Главная цель проектирования процессов заключается в отображении функций, полученных на этапе анализа, в модули ИС. При проектировании модулей определяют интерфейсы программ: разметку меню, вид окон, горячие клавиши и связанные с ними вызовы. Этап проектирования завершается разработкой технического проекта ИС.

Конечными продуктами этапа проектирования являются:  
схема базы данных (на основании ER-модели, разработанной на этапе анализа); набор спецификаций модулей системы (они строятся на базе моделей функций). На этапе проектирования осуществляется также разработка архитектуры ПО, включающая в себя выбор платформы и ОС.

При проектировании сложной ИС она разделяется на части, и каждая из них затем исследуется и создается отдельно. В настоящее время используются два различных способа такого разбиения ИС на подсистемы: структурное (или функциональное) разбиение и объектная (компонентная) декомпозиция.

С позиций проектирования ИС суть функционального разбиения может быть выражена известной формулой: "Программа = Данные + Алгоритмы". При функциональной декомпозиции программной системы ее структура описывается блок-схемами, узлы которых представляют собой

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

"обрабатывающие центры" (функции), а связи между узлами описывают движение данных.

При объектном разбиении в системе выделяются "активные сущности" – объекты (или компоненты), которые взаимодействуют друг с другом, обмениваясь сообщениями и выполняя соответствующие функции (методы) объекта.

Если при проектировании ИС разбивается на объекты, то для ее визуального моделирования следует использовать UML. Если в основу проектирования положена функциональная декомпозиция ИС, то UML не нужен и следует использовать структурные нотации.

### **4. Основные технологии разработки ПО**

Как получится... правил ведения разработки либо нет вообще, либо они разработаны и приняты, но не выполняются. Крайне низкий уровень формализма разработки. Итеративный подход, как правило, в таких проектах не используется.

Структурные методологии - группа методологий, разработанных, как правило, еще до широкого распространения ОО языков. Все они предполагают каскадную разработку. Т.е., основу этих методологий составляет последовательный переход от работы к работе и передача результатов (документов) очередного этапа участникам следующего как основа процесса. Предполагают достаточно высоко формализованный подход

Гибкие методологии – большинство методологий этой группы нацелены на минимизацию рисков, путём сведения разработки к серии коротких циклов – итераций. Основной метрикой является рабочий продукт, отдавая предпочтение непосредственному общению, уменьшают объем письменной документации по сравнению с другими методами. Это привело к критике этих методов как не дисциплинированных.

1. Главное — удовлетворить заказчика и предоставить ему продукт как можно скорее

2. Новые выпуски продукта должны появляться раз в несколько недель.

3 Наиболее эффективный способ передачи знаний участникам разработки и между ними – личное общение

#### **4 Работающая программа — лучший показатель прогресса**

Таким образом, эти методы явно ориентированы на итеративную разработку ПО и на минимальную формализацию ДЛЯ КОНКРЕТНОГО ПРОЕКТА процесса (минимально допустимый уровень формализации).

Примеры: eXtreme Programming(XP), Crystal Clear(CC), Feature Driven Development (FDD).

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

ГОСТы как и СММ, не являются методологиями. Они, как правило, не описывают сами процессы разработки ПО. Но они формулируют определенные требования к процессам, которым в той или иной степени соответствуют различные методологии.

Модели зрелости процесса разработки (СММ, СММ) – модель зрелости процессов создания ПО, которая предназначена для оценки уровня зрелости процесса разработки в конкретной компании.

В соответствие с этой моделью есть пять уровней зрелости процесса разработки. Первый уровень соответствует разработке «как получится», когда на каждый проект разработчики идут как на подвиг. Второй уровень соответствует более-менее налаженным процессам, когда можно с достаточной уверенностью надеяться на положительный исход проекта. Третий уровень соответствует наличию разработанных и хорошо описанных процессов, используемых при разработке. Четвертый — активному использованию метрик в процессе управления для постановки целей и контроля их достижения. И, наконец, пятый уровень означает способность компании оптимизировать процесс по мере необходимости.

После появления СММ стали разрабатываться специализированные модели зрелости для разработки информационных систем, для процесса выбора поставщиков и некоторые другие. На их основе была разработана интегрированная модель СММ (Capability Maturity Model Integration). Кроме того, в СММ была предпринята попытка преодолеть проявившиеся к тому времени недостатки СММ. А именно, преувеличение роли формальных описаний процессов, когда наличие определенной документации оценивалось значительно выше, чем просто хорошо налаженный, но не описанный процесс. Тем не менее, СММ также ориентирован на использование весьма формализованного процесса.

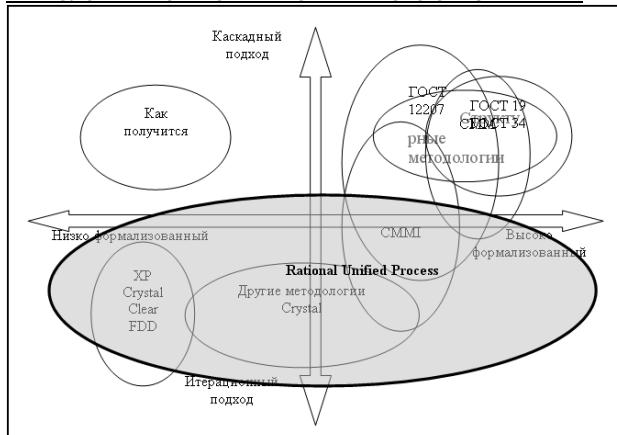
Таким образом, основой СММ и СММ является формализация процесса разработки. Они нацеливают разработчиков, желающих сертифицироваться на достаточно высокую степень зрелости процесса разработки, на внедрение жесткого формализованного процесса.

RUP – итеративная методология. (см вопрос 16 в ТРПО) Хотя выполнение всех фаз или какого-то минимального числа итераций нигде в RUP не оговаривается, весь подход ориентирован на то, что их достаточно много. Только при этом можно настроить процесс для последующих итераций, основываясь на результатах начальных. Ограниченнное количество итераций не позволяет использовать в полной мере

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

все преимущества RUP. Вместе с тем, RUP можно использовать и в «практически каскадных» проектах, включающих реально всего пару итераций: одну в фазе Построение и одну в фазе Передача. Кстати говоря, именно такое количество итераций, как правило, реально используется в каскадных проектах. Ведь проведение испытаний и опытной эксплуатации системы предполагает внесение исправлений, которые могут предполагать определенные действия, связанные с анализом, проектированием и разработкой, то есть фактически являются еще одним проходом через все фазы разработки.

Что касается формальности методологии, то здесь RUP представляет пользователю весьма широкий диапазон возможностей. Если выполнять все работы и задачи, создавать все артефакты и достаточно формально (то есть с официальным назначением рецензента, с предоставлением рецензентом достаточно полной рецензии в виде электронного или бумажного документа и т.д.) проводить все рецензирования, RUP представляет собой крайне формальную, тяжеловесную методологию. С другой стороны, RUP позволяет разрабатывать только те артефакты и выполнять только те работы и задачи, которые необходимы в конкретном проекте. А выбранные артефакты могут выполняться и рецензироваться с произвольной степенью формальности. Можно требовать детальной проработки и тщательного оформления каждого документа. Можно требовать предоставления столь же тщательно выполненной и оформленной рецензии. А, кроме того, всегда остается еще одна возможность — сформировать документ «в голове». Попросту говоря, продумать соответствующий вопрос, принять соответствующее решение. И, если это решение касается только Вас, то ограничиться, например, комментарием в коде программы.



Схематично можно представить так ^ ^ ^

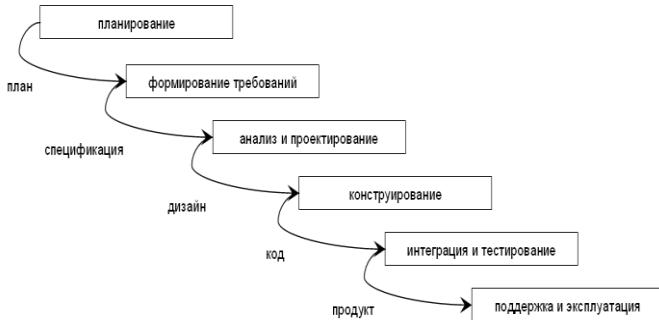
## 5. Модели жизненного цикла (ЖЦ).

Под жизненным циклом ПС понимают весь период его разработки и эксплуатации, начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования. Или последовательность фаз проекта, задаваемая исходя из потребностей управления проектом.

В рамках методологии Института управления проектами ЖЦ проекта включает: Инициация; Планирование; Выполнение; Контроль и мониторинг; Завершение.

**Каскадная (водопадная)** модель предполагает строго последовательное (во времени) и однократное выполнение всех фаз проекта с жестким (детальным) предварительным планированием в контексте предопределенных или однажды и целиком определенных требований к ПС. В каскадной модели переход от одной фазы проекта к другой предполагает полную корректность результата (выхода) предыдущей фазы -> накопление возможных ошибок на ранних этапах. Кроме того, эта модель не способна гарантировать необходимую скорость отклика и внесение соответствующих изменений в ответ на быстро меняющиеся потребности пользователей, для которых программная система является одним из инструментов исполнения бизнес-функций.

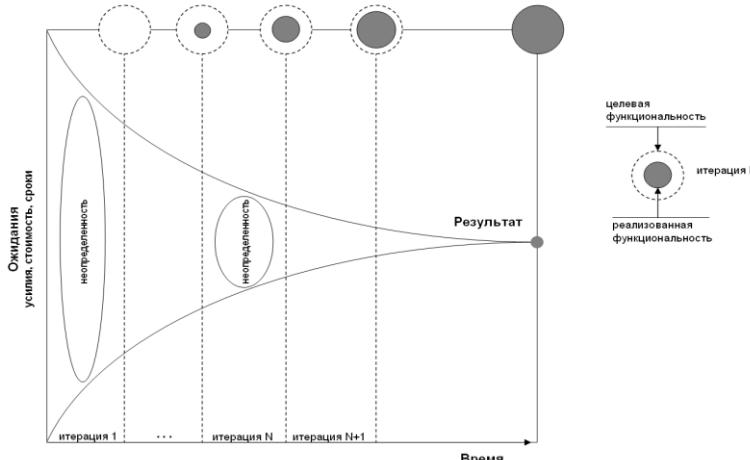
## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.



Существует каскадная модель жизненного цикла с возможными возвратами на предыдущий уровень (при изменении требований происходит возврат в нужную фазу разработки).

*Итеративная и инкрементальная модель – эволюционный подход*

**Итеративная модель** - выполнение работ паралельно с непрерывным анализом полученных результатов и корректировкой предыдущих этапов работы; предполагает разбиение ЖЦ проекта на последовательность итераций, каждая из которых напоминает "мини-проект", и проходит цикл Планирование - Реализация - Проверка – Оценка в применении к созданию меньших фрагментов функциональности, по сравнению с проектом, в целом. Цель каждой итерации – получение работающей версии ПС, включающей функциональность, определенную интегрированным содержанием всех предыдущих и текущей итерации. Результат финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации, продукт развивается инкрементально.



#### Преимущества итеративного подхода:

\*снижение воздействия серьезных рисков на ранних стадиях проекта, что ведет к минимизации затрат на их устранение;

\*организация эффективной обратной связи проектной команды с потребителем (заказчиками) и создание продукта, реально отвечающего его потребностям;

\*акцент усилий на наиболее важные и критичные направления проекта;

\*непрерывное итеративное тестирование, позволяющее оценить успешность всего проекта в целом;

\*раннее обнаружение конфликтов между требованиями, моделями и реализацией проекта;

\*более равномерная загрузка участников проекта;

\*эффективное использование накопленного опыта;

\*реальная оценка текущего состояния проекта и, как следствие, большая уверенность в его успешном завершении.

Эволюционная модель подразумевает не только сборку работающей (с точки зрения результатов тестирования) версии системы, но и её развертывание в реальных операционных условиях с анализом откликов пользователей для определения содержания и планирования следующей итерации. Поскольку на каждом шаге мы имеем работающую систему то:

- можно очень рано начать тестирование пользователями
- можно принять стратегию разработки в соответствии с бюджетом, полностью защищающую от перерасхода времени или средств.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Таким образом, значимость эволюционного подхода на основе организации итераций особо проявляется в снижении неопределенности с завершением каждой итерации. Идея эволюционного подхода, предполагает, что итеративному разбиению может быть подвержен не только ЖЦ в целом, включающий перекрывающиеся фазы – формирование требований, проектирование, конструирование и т.п., но и каждая фаза может, в свою очередь, разбиваться на уточняющие итерации, связанные, например, с детализацией архитектуры модулей системы.

### **Сpirальная модель**

Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию ЖЦ. Главное достижение спиральной модели состоит в том, что она предлагает возможность адаптации удачных аспектов существующих моделей процессов ЖЦ для конкретного проекта.

#### Преимущества:

Модель уделяет специальное внимание раннему анализу возможностей повторного использования.

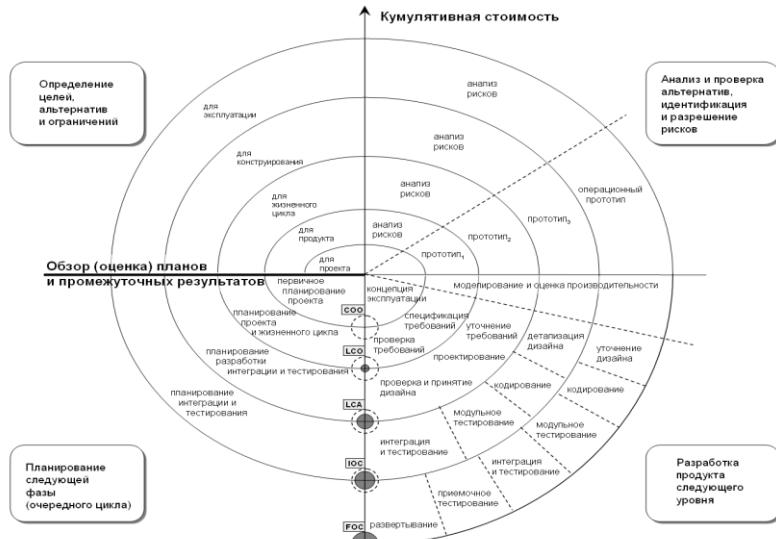
Модель предполагает возможность эволюции ЖЦ, развитие и изменение программного продукта. Подход, предусматривающий скрытие информации о деталях на определенном уровне дизайна, позволяет рассматривать различные архитектурные альтернативы так, как если бы мы говорили о единственном проектном решении, что уменьшает риск невозможности согласования функционала продукта и изменяющихся целей (требований).

Модель уделяет специальное внимание предотвращению ошибок и отbrasыванию ненужных или неудвл альтернатив на ранних этапах проекта.

Модель не проводит различий между разработкой нового продукта и расширением (или сопровождением) существующего.

Модель позволяет решать интегрированные задачи системной разработки, охватывающей и программную и аппаратную составляющие создаваемого продукта.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.



6 ключевых характеристик, обеспечивающих успешное применение СМ:

1. Паралельное, а не последовательное определение артефактов проекта
2. Согласие в том, что на каждом цикле уделяется внимание:
  - целям и ограничениям, важным для заказчика
  - альтернативам организации процесса и технологических решений, закладываемых в продукт
  - идентификации и разрешению рисков
  - оценки со стороны заинтересованных лиц (в первую очередь заказчика)
  - достижению согласия в том, что можно и необходимо двигаться дальше
3. Использование соображений, связанных с рисками, для определения уровня усилий, необходимого для каждой работы на всех циклах спирали.
4. Использование соображений, связанных с рисками, для определения уровня детализации каждого артефакта, создаваемого на всех циклах спирали.
5. Управление ЖЦ в контексте обязательств всех заинтересованных лиц на основе трех контрольных точек: Life Cycle Objectives (LCO) - цели и содержание жизненного цикла, Life Cycle Architecture (LCA), Initial Operational Capability (IOC)
6. Уделение специального внимания проектным работам и артефактам создаваемой системы (включая непосредственно

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

разрабатываемое ПО, ее окружение, эксплуатационные характеристики) и ЖЦ разработки и используя-ся.

### Набор контрольных точек в сегодняшней спиральной модели:

- Concept of Operations (COO) – концепция использования системы;
- Life Cycle Objectives (LCO) – цели и содержание ЖЦ;
- Life Cycle Architecture (LCA) – архитектура ЖЦ; здесь же возможно говорить о готовности концептуальной архитектуры целевой программной системы;
- Initial Operational Capability (IOC) – первая версия создаваемого продукта, пригодная для опытной эксплуатации;
- Final Operational Capability (FOC) – готовый продукт, развернутый (установленный и настроенный) для реальной эксплуатации.

## **6. Качество программной системы. Модель характеристик качества. Характеристики качества ПО**

Процессы разработки, приобретения и внедрения сложных систем, к которым относятся в частности программные комплексы, должны находиться под жестким управлением контролем. В настоящее время практически во всех организациях обеспечивается контроль важнейших характеристик, связанных с производством и использованием программных продуктов, таких как время, финансовые средства, ресурсы и т.п. Однако в большинстве случаев вне пределов сферы контроля оказывается наиболее важная характеристика программных продуктов – это качество продукта, поскольку «невозможно контролировать то, что нельзя измерить». В настоящее время уже существуют организации, в которых накоплен достаточно большой опыт использования метрик в управлении качеством разрабатываемых и внедряемых программных продуктов. Использование их апробированных подходов значительно повышает предсказуемость проектов, снижает финансовые и ресурсные издержки.

Среди используемых метрик качества ПО есть универсальные метрики, которые применимы практически ко всем видам ПО. В то же время большая часть наиболее важных метрик в успешных проектах разрабатывается индивидуально на основе особенностей проекта и характеристик предметной области.

Сейчас существует несколько определений качества, которые в целом совместимы друг с другом. Приведем наиболее распространенные:

**Определение ISO:** Качество - это полнота свойств и характеристик продукта, процесса или услуги, которые

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

обеспечивают способность удовлетворять заявленным или подразумеваемым потребностям.

**Определение IEEE:** Качество программного обеспечения - это степень, в которой оно обладает требуемой комбинацией свойств.

Общее качество программной системы включает в себя на верхнем уровне ряд составляющих, которые должны быть приняты во внимание при управлении качеством

Цена качества – стоимость, которая может быть сэкономлена, если все исполнители работают безупречно.

1. Качество накапливается в продукте, вклад на ранних стадиях имеет более сильное влияние.

2. Чем выше качество процесса разработки, тем выше качество разработанного в этом процессе ПО.

QA – инспектирование (“белый ящик”; проводится коллегами на уровне модулей ПО) и тестирование.

Метрики качества: адаптируемость, сложность интерфейсов и интеграции, тестовое покрытие, профили ошибок, степень удовлетворения пользователей.

Составляющие качества информационной системы:

Качество инфраструктуры: качество аппаратного и поддерживающего программного обеспечения (например, качество операционных систем, компьютерных сетей и т.п.).

Качество программного обеспечения: качество ПО информационной системы.

Качество данных: качество данных, использующихся инфор-ой системой на входе.

Качество информации: качество информации, производимое инф-ой системой.

Качество административного управления – качество менеджмента, включая качество бюджетирования, планирования и календарного контроля.

Качество сервиса – качество обучения, системной поддержки и т.п.

Кроме перечисленных составляющих качества должно быть принято во внимание качество обслуживаемого бизнес процесса.

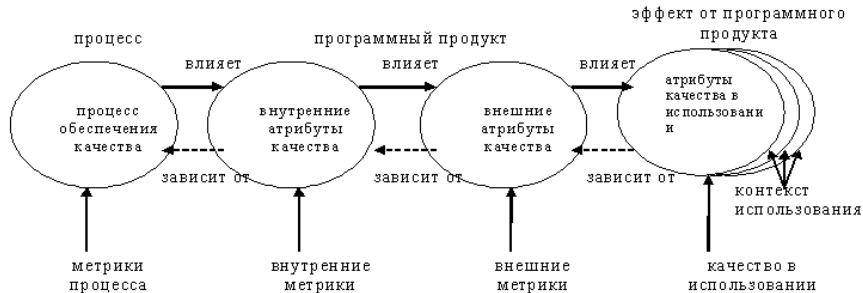
**Модель характеристик качества:** Стандарты ISO определяют модель характеристик качества ПС, которая состоит из нескольких видов атрибутов качества:

- внутренние атрибуты качества (требования к качеству кода и внутренней архитектуре);

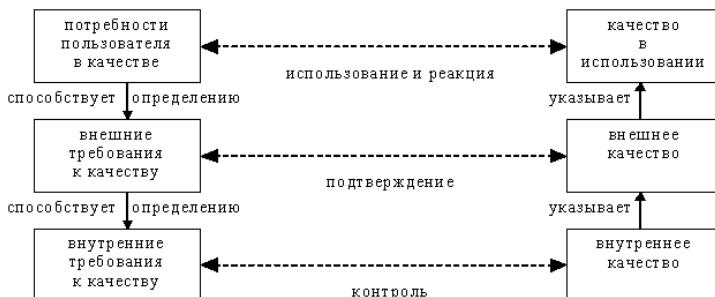
- внешние атрибуты качества (требования к функциональным возможностям и т.д.);

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- атрибуты «качества в использовании»(данные атрибуты качества относятся не только к ПС, а ко всей информационной системе)



Требования пользователя к качеству в спецификациях должны в процессе верификации преобразовываться в требования к внешнему качеству, а затем в требования к внутреннему качеству. Процессы реализации требований к внутреннему качеству должны обеспечивать внешнее качество, а последнее - в качестве для пользователей.



Модель внутренних и внешних характеристик качества ПС состоит из шести групп базовых показателей, каждая из которых детализирована несколькими нормативными субхарактеристиками:

1.функциональная пригодность - набор атрибутов характеризующий, соответствие функциональных возможностей ПО набору требуемой пользователем функциональности. Детализируется следующими подхарактеристиками :

- 1.1.пригодностью для применения;
- 1.2.корректностью (правильностью, точностью);
- 1.3.способностью к взаимодействию;
- 1.4.защищенностью;

2.надежность - набор атрибутов, относящихся к способности ПО сохранять свой уровень качества функционирования в

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.  
установленных условиях за определенный период времени.  
Детализируется следующими подхарактеристиками :

- 2.1. уровнем завершенности (отсутствия ошибок);
- 2.2. устойчивостью к дефектам;
- 2.3. восстанавливаемостью;
- 2.4. доступностью;
- 2.5. готовностью;

3. эффективность - набор атрибутов, относящихся к соотношению между уровнем качества функционирования ПО и объемом используемых ресурсов при установленных условиях.  
Детализируется следующими подхарактеристиками :

- 3.1. временной эффективностью;
- 3.2. используемостью ресурсов;

4. применимость - Набор атрибутов, относящихся к объему работ, требуемых для исполнения и индивидуальной оценки такого исполнения определенным или предполагаемым кругом пользователей. Детализируется следующими подхарактеристиками :

- 4.1. понятностью;
- 4.2. простотой использования;
- 4.3. изучаемостью;
- 4.4. привлекательностью;

5. сопровождаемость - набор атрибутов, относящихся к объему работ, требуемых для проведения конкретных изменений (модификаций). Детализируется следующими подхарактеристиками :

- 5.1. удобством для анализа;
- 5.2. изменяемостью;
- 5.3. стабильностью;
- 5.4. тестируемостью;

6. переносимость - набор атрибутов, относящихся к способности ПО быть перенесенным из одного окружения в другое.  
Детализируется следующими подхарактеристиками :

- 6.1. адаптируемостью;
- 6.2. простотой установки (инсталляции);
- 6.3. существованием (соответствием);
- 6.4. замещаемостью.

В стандартах также определена модель характеристик качества в использовании. В этой модели используются несколько другие базовые характеристики по сравнению с моделью внутреннего и внешнего качества. Основными характеристиками качества ПС в использовании являются:

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- системная эффективность - применения программного продукта (ПП) по назначению;
- продуктивность - производительность при решении основных задач ПС, достигаемая при реально ограниченных ресурсах в конкретной внешней среде применения;
- безопасность - надежность функционирования комплекса программ и возможный риск от его применения для людей, бизнеса и внешней среды;
- удовлетворение требований и затрат пользователей в соответствии с целями применения ПС.

Характеристики качества отражают свойства, определяющие качество программного обеспечения. В силу сложной природы количественной оценки характеристик качества программного обеспечения для их оценки используют иерархические системы измерений. Иерархию характеристик качества образуют факторы, критерии, метрики и оценочные элементы (рис.3). Факторы и критерии, составляющие два верхних уровня иерархии измерений, отражают функциональные характеристики программного обеспечения, а нижние (метрики и оценочные элементы) – конструктивные характеристики, от которых зависит качество программного обеспечения. Измеримость характеристик качества обеспечивается составом характеристик самого нижнего уровня – оценочных элементов.

## **7 Сравнение технологий разработки ПО**

*ключевыми* характеристиками для сравнения методологий следует считать, прежде всего:

отношение методологии к итеративной разработке и степень формальности в оформлении рабочих материалов и вообще в проведении разработки

**Отличие итеративной разработки от каскадной** («водопада»). Каскадные методологии разработки ПО исходят из того, что разработка ПО делится на фазы, каждая из которых характеризуется своим набором работ. В отличие от них итеративный подход разбивает разработку на несколько итераций, в ходе каждой из которых выполняются практически все типы работ, и создается реальная работающая система с все более развитыми функциональными возможностями. При каскадном подходе сначала происходит выявление всех требований к проекту и их анализ. Затем проектная группа приступает к проектированию системы. После этого начинается разработка кода и модульное тестирование. После этого идет сборка и системное тестирование. При итерационном подходе разработка ПО разбивается на относительно короткие итерации.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Практически во всех итерациях выполняется и выявление требований, и проектирование, и тестирование. Так, в самой первой итерации еще до выявления всех требований может начаться разработка прототипа, на котором проверяются основные архитектурные решения. По мере детализации требований на отдельные подсистемы или компоненты на последующих итерациях начинается их проектирование и кодирование. Разработанные «начерно» подсистемы и компоненты собираются в единую систему (не дожидаясь завершения разработки всех подсистем) и тут же начинается их системное тестирование. В ходе разработки всегда выявляются дополнительные требования или изменяются выявленные ранее. Также появляются новые ограничения, связанные с принятыми техническими решениями. В наиболее полной мере их удается учесть именно в итерационной разработке, поскольку именно при таком подходе руководство проекта в полной мере готово к изменениям.

С какими методологиями имеет смысл сравнивать RUP? Конечно, с теми, которые наиболее распространены или про которые хотя бы можно прочитать.

Видимо, все еще самый распространенный метод — это отсутствие какого-либо сознательно выбранного метода. Разработка ведется так, как сложилось, как привыкли. Конечно, в каждой команде свой подход к разработке, тем не менее, и в них можно выявить некоторые общие черты.

Структурные методологии, в частности, основанные на подходах Эдварда Йордана, диаграммах «сущность-отношение» и потоках данных, были первыми активно продвигаемыми в нашей стране методологиями. Хотя они нередко связывались (по крайней мере, у слушателей презентаций) с реализующими их CASE средствами, а не рассматривались как самостоятельные методологии, тем не менее, они оказались достаточно известными, хотя нельзя сказать, что широко используемыми. Тем не менее, сравнение с ними имеет определенный смысл. По крайней мере, оно должно показать, насколько RUP отличается от них.

Наибольший интерес в настоящее время, видимо, представляет сравнение с гибкими (Agile) методологиями, которые в последние годы активно развиваются и приобрели определенную популярность. Так называется группа относительно новых методов, развиваемых участниками Agile Manifest — объединения в поддержку гибких методов. Общее число таких методологий достаточно велико. Но не все из них широко известны, и не по всем из них можно найти материалы на русском

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

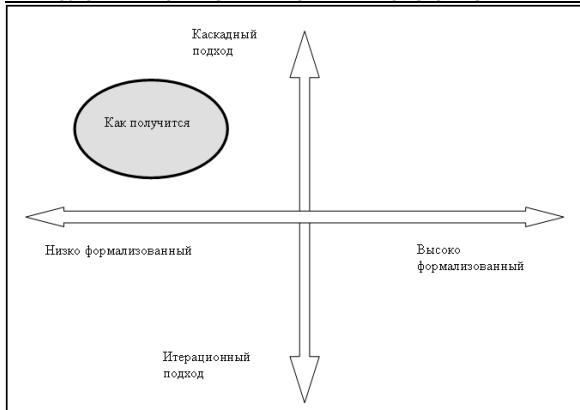
языке. Поэтому для сравнения были выбраны Экстремальное программирование (XP), Crystal Clear и FDD (Feature Driven Development).

Кроме методологий, описывающих что, как и в каком порядке надо делать, есть еще один тип документов, регламентирующих разработку ПО. Это международные и государственные стандарты и другие разработки, направленные на определение требований к процессам разработки. Среди стандартов наибольший интерес для отечественного производителя представляют, конечно, ГОСТы 19-ой и 34-ой серий и ГОСТ 12207 Р ИСО МЭК. А среди других регламентирующих документов наиболее известна модель зрелости процессов разработки ПО CMM, разработанная Software Engineering Institute.

Как получится...

Увы, это самая сложная для описания категория. Ибо в нее попадают как судорожные метания новичка, пытающегося любой ценой выполнить свой первый проект, так и вполне зрелые и устоявшиеся методологии, вобравшие в себя многолетний и разнообразный опыт конкретных команд разработчиков и даже описанные во внутренних регламентах. Поскольку люди, способные разработать собственную методологию, как правило, могут сами оценить ее с точки зрения итеративности и формализованности, будем ориентироваться на новичков. Увы, чаще всего это означает, что правил ведения разработки либо нет вообще, либо они разработаны и приняты, но не выполняются. Естественным в таких условиях является крайне низкий уровень формализма разработки. Так что с этим все понятно.

А что с использованием итеративного подхода? Увы, как правило, он в таких проектах не используется. Прежде всего, потому, что он бы позволил еще на первых итерациях оценить проект как крайне сомнительный. И требующий срочного вмешательства более высокого руководства для наведения порядка. Ведь в итеративном проекте традиционный ответ программиста, что у него уже все на 90% готово, проходит только до момента завершения первой итерации...



## 8. Уровень формализма. Количество итерации

В понятие формализма входит:

- 1) количество документов;
  - 2) степень аккуратности их оформления и формальность процедур рецензирования, одобрения и передачи;
- Формализм очень сильно влияет на скорость и трудоемкость разработки. Детальная документация требует много времени и много сил, отсутствие или недостаточный уровень формализма при выполнении проекта может приводить к несогласованности решений.

Долгое время в информационной индустрии бытовало мнение, что чем больше тщательно оформленной документации выпускается в ходе выполнения проекта — тем лучше, но это не так. Иногда толстые тома можно заменить компактными диаграммами.

### Факторы, влияющие на уровень формализации

- Масштаб проекта. Чем больше людей участвуют, тем формально он, как правило, должен вестись;
- Критичность проекта. Проекты, в которых ошибки могут приводить к тяжелым последствиям вплоть до гибели людей, должны проводиться существенно более формально, чем те проекты, в которых ошибки приводят только к временным неудобствам (как, например, при разработке домашней интернет странички);
- Распределение участников. Чем компактнее расположены участники, чем легче им общаться между собой, тем менее формализованным может быть проект;
- Новизна проекта. Чем более новые для разработчиков технологии вы используете, чем меньше вы знакомы с

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

предметной областью, тем более тщательно надо прорабатывать проектные решения, и, соответственно, тем более формально это должно происходить. Однако, при наличии компактной высококвалифицированной группы программистов, прорабатывающих такие решения, вы, возможно, предпочтете тщательно оформлять уже отработанные решения перед тем, как приступать к их тиражированию в проекте;

- Требования заказчика. Они существенно различаются в зависимости от отрасли и статуса организации-заказчика. Как правило, эти требования выше для государственных предприятий;

- Ожидаемая долговечность проекта. Если разрабатываемое для конкретного заказчика ПО предполагается через пару лет будет заменено новым, то вряд ли есть смысл тратить много сил на документацию, которая могла бы значительно удешевить более длительный процесс сопровождения ПО. Зато если срок жизни проекта предполагается достаточно длинным — без хорошей документации не обойтись. Автору приходилось иметь дело с системой, созданной более 15 лет тому назад и продолжающей эксплуатироваться. В такой ситуации без качественной документации поддержка системы в работоспособном состоянии была бы просто невозможна.

Таким образом, если ваша организация достаточно велика (больше десятка разработчиков) и если вы выполняете проекты для различных отраслей или для систем различного уровня критичности, вам желательно выбирать для каждого проекта свой оптимальный уровень формализации. И здесь возможности RUP существенно превосходят другие методологии. Он позволяет без переучивания персонала просто за счет выбора используемых в проекте артефактов и способа их разработки и рецензирования получить тот уровень формализма, который нужен. И который гарантирует минимизацию затрат на выполнение проекта и максимально быстрое завершение проекта.

## **Количество итераций**

Показательным параметром, чем количество итераций является длительность итерации.

Итерация — это не просто календарный срок, это определенный этап выполнения проекта:

- на который составляется детальный план
- который завершается выпуском итогового релиза (в ходе итерации сборка программы может проводиться еженедельно

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

или даже ежедневно, но итоговый релиз выпускается единственный раз в конце итерации)

- по завершении которого могут меняться приоритеты дальнейшей разработки
- по завершении которого проводится оценка и если надо вносятся соответ. изменения.

### Итерации в процессе разработки позволяют:

- контролировать и корректировать ход выполнения вашего проекта (необходимость продемонстрировать работающий релиз в конце итерации не позволяет программисту ограничиться типичным ответом о 90% -ной готовности класса или метода);
- эффективнее работать с изменяющимися требованиями (например, осуществляя анализ изменившихся требований и решая, что будет переработано в продукте, в ходе подготовки к очередной итерации);
- эффективнее работать с рисками, корректируя планы очередной итерации в соответствие с текущим состоянием списка наиболее приоритетных рисков;
- на ранних этапах оценивать потенциальные характеристики системы (поскольку наличие работоспособного релиза позволяет начать тестирование системы с самых первых итераций);
- не тратить время на разработку беспол. дет. планов на далекую перспективу (если требования к проекту могут существенно меняться раз в месяц, а принципиальная архитектура еще не выбрана, стоит ли детально планировать работы на следующий квартал?).

Для проведения итерации нужны дополнительные накладные расходы на подведение итогов и (пере)планирование. Размер этих накладных расходов зависит от числа и распределения участников проекта. Для группы в пять человек, работающей в одной комнате, подведение итогов итерации может занять час времени. Для группы из сотни разработчиков, работающих в нескольких различных городах, на сбор и подведение итогов может уйти пара недель. И еще почти столько же уйдет на доведение новых планов и организацию работы по ним.

Продолжительность итерации должна быть тем больше, чем больше и распределенее команда, и тем меньше, чем больше неопределенность проекта. Например, чем более новая технология используется или чем менее стабильны требования к проекту.

Средства управления изменениями и конфигурациями и средства автоматизации тестирования позволяют снизить не только

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.  
трудоемкость процесса разработки, но и накладные расходы на подведение итогов итерации.

## **9. Требования. Анализ требований.**

**Требования** - это спецификация того, что должно быть реализовано. В них описано поведение системы, свойства системы и её атрибуты. Они могут быть ограничены процессом разработки системы.

**Спецификация требований** – документ, определяющий требования к приложению и являющийся подобием контракта и путеводной нити для заказчика и разработчиков.

### **УРОВНИ ТРЕБОВАНИЙ**

- Бизнес – требования;
- Требования пользователей;
- Функциональные требования;
- Системные требования.

### **ХАРАКТЕРИСТИКИ ТРЕБОВАНИЙ**

- Полнота;
- Корректность;
- Осуществимость;
- Необходимость;
- Назначение приоритетов;
- Недвусмысленность;
- Проверяемость;
- Согласованность;
- Способность к модификации;
- Трассируемость.

### **ПРАВА КЛИЕНТОВ ПРИ ФОРМИРОВАНИИ ТРЕБОВАНИЙ**

- 1) Иметь дело с аналитиком, который разговаривает на вашем языке;
- 2) Иметь дело с аналитиком, хорошо изучившим ваш бизнес и цели, для которых создаётся система;
- 3) Потребовать, чтобы аналитик преобразовал требования, представленные вами устно, в письменную спецификацию требований к программному продукту;
- 4) Получить от аналитика подробный отчёт обо всех рабочих продуктах, созданных в процессе формулирования требований;
- 5) На уважительное и профессиональное отношение к вам со стороны аналитиков и разработчиков;
- 6) Знать о вариантах и альтернативах требований и их реализации;

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- 7) Описать характеристики, упрощающие работу с продуктом;
- 8) Изменить требования или разрешить использование имеющихся программных компонентов;
- 9) Получить исчерпывающие сведения о сумме затрат, ожидаемом эффекте и необходимых компромиссах, которые возникают в связи с изменениями в ПО;
- 10) Потребовать, чтобы система функциональностью и качеством удовлетворяла требованиям заказчика .

## **ОБЯЗАННОСТИ КЛИЕНТОВ ПРИ ФОРМИРОВАНИИ ТРЕБОВАНИЙ**

- 1) Ознакомить аналитиков и разработчиков с особенностями вашего бизнеса;
- 2) Потратить столько времени, сколько необходимо, на объяснение требований;
- 3) Точно и корректно описать требования к системе;
- 4) Принимать своевременные решения;
- 5) Уважать определённую разработчиком оценку стоимости и возможность реализации ваших требований;
- 6) Определять приоритеты требований;
- 7) Просматривать документы с требованиями и оценивать прототипы;
- 8) Своевременно сообщать об изменениях требований;
- 9) Поддерживать принятый в организации порядок контроля изменений;

## **РАЗРАБОТКА ТРЕБОВАНИЙ**

### **1) Извлечение**

- Определите процесс формулирования требований;
- Определите образ и границы проекта;
- Определите классы пользователей;
- Выделите из пользователей ярого сторонника продукта;
- Создайте фокус – группу;
- Определите назначение проекта;
- Определите системные события и реакцию на них;
- Проводите совместные семинары, упрощающие сбор требований;
- Наблюдайте за пользователями на рабочих местах;
- Изучите отчёты о проблемах;
- Используйте требования многократно.

### **2) Анализ**

- Нарисуйте контекстную диаграмму;
- Создайте прототипы;
- Проанализируйте осуществимость;

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Расставьте приоритеты для требований;
- Смоделируйте требования;
- Создайте словарь терминов;
- Распределите требования по подсистемам;
- Воспользуйтесь технологией развёртывания функций качества.

### **3) Документирование**

- Используйте шаблон спецификации требований к ПО;
- Определите источники требований;
- Задайте каждому требованию уникальный идентификатор;
- Задокументируйте бизнес – правила;
- Укажите атрибуты качества.

### **4) Проверка**

- Изучите документы с требованиями;
- Протестируйте требования;
- Определите критерии приемлемости.

**После того, как разработка требований успешно завершена, мы переходим к анализу этих самых требований.**

- Создание контекстной диаграммы;
- Создание пользовательского интерфейса и технических прототипов;
- Анализ осуществимости требований;
- Определение приоритетов требований;
- Моделирование требований;
- Создание словаря терминов;
- Распределение требований по подсистемам;
- Применение технологий развёртывания функций качества.

**Контекстная диаграмма** – простая модель анализа, отображающая место новой системы в соответствующей среде. Определяет границы и интерфейсы между разрабатываемой системой и сущностями, внешними для этой системы, например пользователями, устройствами и прочими информационными системами.

Проанализируйте, насколько реально реализовать каждое требование при разумных затратах и с приемлемой производительностью в предполагаемой среде.

Рассмотрите риски, связанные с реализацией каждого требования, включая конфликты с другими требованиями, зависимость от внешних факторов и препятствия технического характера.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Воспользуйтесь аналитич-м подходом и определите относительные приоритеты реализации функций продукта, решаемых задач или отдельных требований.

На основании приоритетов установите, в какой версии будет реализована та или иная функция или набор требований.

В ходе работы над проектом периодически корректируйте приоритеты в соответствии с потребностями клиента, условиями рынка и бизнесцелями.

Когда ожидания клиентов высоки, а сроки поджимают, вам нужно, чтобы в продукте были реализованы самые ценные функции как можно раньше.

**Приоритеты** – способ разрешения борьбы между конкурирующими требованиями за ограниченные ресурсы.

При оценке приоритетов учитывают два измерения: **важность и срочность.**

	ВАЖНЫЕ	НЕ ВАЖНЫЕ
СРОЧНЫЕ	ВЫСОКИЙ ПРИОРИТЕТ	НЕ ЗАНИМАЙТЕСЬ ИМИ!
НЕ СРОЧНЫЕ	СРЕДНИЙ ПРИОРИТЕТ	НИЗКИЙ ПРИОРИТЕТ

**Анализ требований** — это процесс сбора требований к системе, их систематизации, документир-ия, анализа, выявления противоречий, неполноты, разрешения конфликтов

### **Анализ требований**

- Создание контекстной диаграммы – простая модель анализа, отображающая место новой системы в соответствующей среде. Она определяет границы и интерфейсы между разрабатываемой системой и сущностями, внешними для этой системы, н-р польз., устройствами и прочими информационными системами
- Создание пользовательского интерфейса и технических прототипов;
- Анализ осуществимости требований;
- Определение приоритетов требований – определение относительных приоритетов реализации функций продукта, решаемых задач или отдельных требований. На основе приоритетов устанавливается, в какой версии будет реализована та или иная функция или набор требований. В ходе работы над проектом периодически происходит корректировка приоритетов в соответствии с потребностями клиента, условиями рынка и бизнесцелями.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Моделирование требований – нужно сосредоточиться на наиболее сложных и опасных участках системы, а так же, где наиболее вероятны неясности и неопределённости.;
- Создание словаря терминов – определения всех элементов и структур данных, связанных с системой, что позволяет всем участникам проекта использовать согласованные определения данных;
- Распределение требований по подсистемам – требования к сложному продукту, включающему несколько подсистем, следует соразмерно распределять между программными, аппаратными и операторскими подсистемами и компонентами. Применение технологий развёртывания функций качества.

## **10. Управление проектом. Этапы. Задачи. Треугольник проекта.**

**Проект** - это временное предприятие, предназначенное для создания уникальных продуктов или услуг. **Управление проектами** - дисциплина применения методов, практик, опыта, и средств к работам проекта для достижений целей проекта, при условии удовлетворения ограничений, определяющих рамки проекта.



Управление проектами - дисциплина применения методов, практик, опыта, и средств к работам проекта для достижений целей проекта, при условии удовлетворения ограничений, определяющих рамки проекта.

Чаще всего говорят о трех основных ограничениях(«железный треугольник»)

- Содержание

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Время
- Стоимость

В приложении ПО добавляют четвертое ограничение – качество(приемлемое качество). В любом проекте существует функция четырех переменных: Функция(функциональность, дата сдачи(время),объем работ(стоимость), качество) = **Const**

Варианты треугольников:

- 1) Содержание, сроки, бюджет
- 2) Содержание, сроки, стоимость
- 3) Содержание, сроки, ресурсы
- 4) Содержание, сроки, качество
- 5) Содержание, ресурсы, качество

Любой проект в процессе своей реализации проходит различные стадии, называемые в совокупности **жизненным циклом проекта**.

Для реализации различных функций управления проектом необходимы действия, которые в дальнейшем именуются **процессами управления проектами**.

- Проект состоит из процессов. **Процесс** - это совокупность действий, приносящая результат.
- **Процессы Управления Проектами** - касающиеся организации и описания работ проекта;

**Процессы, ориентированные на продукт** - касающиеся спецификации и производства продукта.

- Эти процессы определяются жизненным циклом проекта и зависят от области приложения.
- Процессы управления проектами могут быть разбиты на 6 основных групп:
  - процессы инициации - принятие решения о начале выполнения проекта;
  - процессы планирования - определение целей и критериев успеха проекта и разработка рабочих схем их достижения;
  - процессы исполнения - координация людей и других ресурсов для выполнения плана;
  - процессы анализа - определение соответствия плана и исполнения проекта поставленным целям и критериям и принятие решений о необходимости корректирующих воздействий;
  - процессы управления - определение корректирующих воздействий, их согласование, утверждение и применение;
  - процессы завершения - формализация выполнения проекта и подведение его к упорядоченному финалу.



PSI - число, лежащее в диапазоне 0-100, которое присваивается проектам и может быть рассчитано в любой точке жизни проекта и определяет вероятность того, что проект завершится успешно. PSI рассчитывается путем присвоения баллов каждому из Десяти Этапов, привязанных к определенному проекту.

### **Десять этапов руководства структурного руководства проектом;**

1. Цель(Наглядное представление цели);
2. Список задач(Разработка списка задач, кот-е необходимо выполнить);
3. Один руководитель
4. Распределение людей по задачам;
5. Управление ожидаемыми результатами, расчет резервов для ошибок, выработка запасных позиций;
6. Стиль руководства;
7. Знать то, что происходит;
8. Сообщать людям, что происходит;
9. Повтор этапов с 1 до 8;
10. Приз;

ЭТАП	1	2	3	4	5	6	7	8	9	10	Общая сумма
max	20	20	10	10	10	10	10	10	0	0	
итого					70					30	100

### **Создание проекта.**

#### **■ Трудозатраты (объем работ, работа)**

Трудозатраты представляют собой количество труда в определенных единицах. Измеряются в человеко-дни, человеко-недели или человеко-годы.

#### **■ График (затраченное время)**

График - это затраченное или календарное время, за которое что-то должно быть выполнено. Измеряется в час, день, месяц, год.

#### **■ Критический путь**

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Критический путь представляет собой самое короткое время (то есть самый короткий график или затраченное время), за которое что-то может быть завершено.

### ■ Контрольные точки (промежуточные этапы)

Они - "вехи" проекта в специфических точках во времени. Они показывают, какая часть проекта была закончена и сколько он еще должен длиться.

## 11. Риски. Управление рисками.

Принятие решений в условиях неопределенности всегда связано с рисками

- PMBOK(Project Management Body of Knowledge) :
  - Неопределенное событие или условие, которое, если осуществится, может иметь как негативное, так и позитивное влияние на итоги проекта
- Риск— Может не оказать влияния на проект
- Проблема— Уже сейчас влияет на проект

### Если рисками не управлять они могут стать проблемами

Решения, принятые в условиях неопределенности, могут иметь самые различные последствия. Предстоящие перемены могут нести не только плохое, но и хорошее, надо быть готовым к этим изменениям, чтобы извлечь из них максимум преимуществ

- Большое количество быстро меняющихся факторов, влияющих на успех проекта
  - Требования пользователей
  - Новые технологии
  - Рыночная конкуренция
  - Эволюция стандартов
  - Требования к безопасности

Подходы к управлению рисками

- PMBOK: Количественный анализ рисков, Непрерывный процесс
- Дисциплина управления рисками MSF:Основана на PMBOK, Превентивное управление рисками, Интеграция с другими компонентами MSF
- eXtreme Programming: Откладывать принятие решения как можно дольше: информации для его принятия будет больше, а возможно оно вообще не понадобится

### Основные принципы управления рисками

- Профилактика
- Готовность к изменениям
- Открытость
- Непрерывность

**2 подхода:**

Профилактика	Борьба с последствиями
Предотвращаем проблемы	Решаем возникшие проблемы
Выявляем причины	Лечим симптомы и последствия
Готовимся заранее	Реагируем на кризис
Действуем по плану	Действуем спонтанно

**Преимущества превентивного управления рисками**

- Предвидение вместо реакции на проблемы
- Позволяет сосредоточиться на корне проблемы, на истинных причинах, а не на симптомах
- Общий управляемый и воспроизводимый подход к борьбе с возможными проблемами

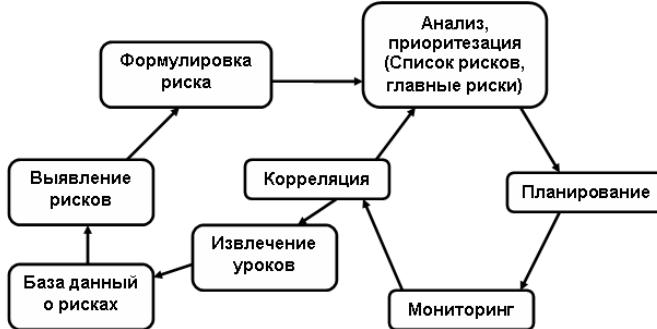
**Примеры рисков и их решение в XP:**

- Заказчик может остаться недовольным результатами и внезапно менять решение - Представитель заказчика работает в команде разработчиков каждый день
- Продукт может содержать ошибки - После каждого изменения вся система подвергается юнит-тестированию
- Качество кода может оказаться низким - Каждый участок программы пишется вдвоем за один компьютером – что не увидел один, заметит другой

**Что нужно делать, чтобы эффективно управлять рисками?**

- Планирование управления рисками
- Идентификация рисков
- Качественная оценка рисков
- Количественная оценка
- Планирование реагирования на риски
- Мониторинг и контроль рисков

**Управление рисками по MSF**



### Шаг 1: Выявление рисков

- Команда выявляет риски, которые связаны с проектом
- Рассматриваются как следствия, так и причины рисков
- Риски документируются в четкой и однозначной форме
- Выполняется классификация (категоризация) рисков
- Создается список рисков

### Шаг 2: Анализ и приоритизация рисков

- Приоритетизация выявленных рисков
  - Невозможно управлять сразу всеми рисками
- Существуют риски, которым необходимо уделить больше внимания
  - Некоторые риски имеют фатальные последствия, но крайне маловероятны
  - Некоторые риски очень вероятны, но их влияние на проект ничтожно

### Пример: анализ риска

- Риск: Если система не будет реализована и протестирована к началу соревнований, они будут сорваны
- Оценка вероятности – 2 (средняя)
- Угроза – 3 (высокая)
- Общее влияние –  $2 \cdot 3 = 6$
- Риск попал в первую десятку

### Шаг 3: Планирование рисков

Цели: Разработка планов действий по отношению к основным рискам, Внесение мероприятий по управлению рисками в расписание проекта

### Предотвращение риска

- Что можно сделать, чтобы уменьшить вероятность риска?

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Что можно сделать, чтобы уменьшить угрозу риска?
- Уменьшений вероятности не есть избегание риска

### Смягчение последствий

- Если риск все-таки осуществляется, надо быть готовым к этому
- Действия должны быть продуманы заранее
- Должны быть определены условия, при которых вступает в силу план смягчения последствий

Триггеры - условия, которые определяют переход от профилактики к реакции на последствия:

- Триггер устанавливается для каждого риска
- Виды триггеров
  - Событие
  - Величина
- Триггеры нужно контролировать

### Шаг 4: Мониторинг

- Цели
  - Отслеживание рисков, триггеров
  - Наблюдение за выполнением планов предотвращения и смягчения последствий
  - Информирование членов команды о событиях, связанных с рисками

### Шаг 5: Корректирование ситуации

- Успешное выполнение планов предотвращения
- Своевременное задействование планов смягчения последствий
- Постоянная деятельность во время работы над проектом

### Шаг 6: Извлечение опыта

- Обратная связь в процессе управления рисками
- Обмен опытом с другими проектными группами
- Усовершенствование процесса управления рисками
- Происходит на протяжении всей жизни проекта

## Опыт, получаемый в результате управления рисками

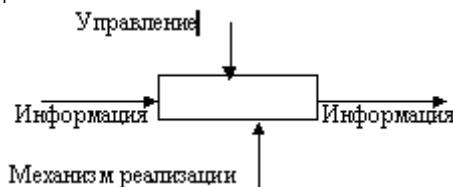
- Новые риски
  - Проект столкнулся с проблемой, которая не была предвидена заранее
- Успешные стратегии предотвращения и смягчения последствий
- База знаний по рискам

## 12. Технология SADT. Системы и модели. Пример.

SADT (аббревиатура выражения Structured Analysis and Design Technique - методология структурного анализа и проектирования) - это методология, разработанная специально для того, чтобы облегчить описание и понимание искусственных систем, попадающих в разряд средней сложности. SADT была создана и опробована на практике в период с 1969 по 1973г.

Методология SADT разработана Дугласом Россом и получила дальнейшее развитие в работе. На ее основе разработана, в частности, известная методология IDEF0 (Icam DEFinition), которая является основной частью программы ICAM (Интеграция компьютерных и промышленных технологий), проводимой по инициативе BBC США.

Методология SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. В основе методологии лежит практический язык SA. Суть его в следующем: каждый блок диаграммы представляется [], в отличие от ИПД, здесь есть не только информационный поток, но и поток по управлению и механизм реализации.



Основные элементы этой методологии основываются на следующих концепциях:

- графическое представление блочного моделирования. Графика блоков и дуг SADT-диаграммы отображает функцию в виде блока, а интерфейсы входа/выхода представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описываются посредством интерфейсных дуг, выражающих "ограничения", которые в свою очередь определяют, когда и каким образом функции выполняются и управляются;
- строгость и точность. Выполнение правил SADT требует достаточной строгости и точности, не накладывая в то же время чрезмерных ограничений на действия аналитика. Правила SADT включают:
- ограничение количества блоков на каждом уровне декомпозиции (правило 3-6 блоков);
- связность диаграмм (номера блоков);
- 的独特性ность меток и наименований (отсутствие повторяющихся имен);
- синтаксические правила для графики (блоков и дуг);
- разделение входов и управлений (правило определения роли данных).
- отделение организации от функции, т.е. исключение влияния организационной структуры на функциональную модель.

Методология SADT может использоваться для моделирования широкого круга систем и определения требований и функций, а затем для разработки системы, которая удовлетворяет этим требованиям и реализует эти функции. Для уже существующих систем SADT может быть использована для анализа функций, выполняемых системой, а также для указания механизмов, посредством которых они осуществляются.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

### **Типы связей между функциями**

Одним из важных моментов при проектировании ИС с помощью методологии SADT является точная согласованность типов связей между функциями. Различают по крайней мере семь типов связывания:

Ниже каждый тип связи кратко определен и проиллюстрирован с помощью типичного примера из SADT.

#### **1. Тип случайной связности:** наименее желательный.

Случайная связность возникает, когда конкретная связь между функциями мала или полностью отсутствует. Это относится к ситуации, когда имена данных на SADT-дугах в одной диаграмме имеют малую связь друг с другом.

**2. Тип логической связности.** Логическое связывание происходит тогда, когда данные и функции собираются вместе вследствие того, что они попадают в общий класс или набор элементов, но необходимых функциональных отношений между ними не обнаруживается.

**3. Тип временной связности.** Связанные по времени элементы возникают вследствие того, что они представляют функции, связанные во времени, когда данные используются одновременно или функции включаются параллельно, а не последовательно.

**4. Тип процедурной связности.** Процедурно-связанные элементы появляются сгруппированными вместе вследствие того, что они выполняются в течение одной и той же части цикла или процесса.

**5. Тип коммуникационной связности.** Диаграммы демонстрируют коммуникационные связи, когда блоки группируются вследствие того, что они используют одни и те же входные данные и/или производят одни и те же выходные данные.

**6. Тип последовательной связности.** На диаграммах, имеющих последовательные связи, выход одной функции служит входными данными для следующей функции. Связь между элементами на диаграмме является более тесной, чем на рассмотренных выше уровнях связок, поскольку моделируются причинно-следственные зависимости.

**7. Тип функциональной связности.** Диаграмма отражает полную функциональную связность, при наличии полной зависимости одной функции от другой. Диаграмма, которая является чисто функциональной, не содержит чужеродных элементов, относящихся к последовательному или более слабому типу связности.

### **Организация бригады**

При разработке программной системы с использованием SADT используется бригадный метод, в котором каждый разработчик наделен вполне определенными функциями:

- авторы составляют требования, спецификации, описывают систему с помощью SADT;
- комментаторы - проектировщики, которые анализируют работу авторов;
- читатели – лица, которые занимаются анализом проектов. Они могут участвовать в обсуждениях, но они не обязаны комментировать действия авторов. Это люди, которые в курсе дела;
- технический комитет – это группа специалистов, анализирующих проект, начиная с высших уровней, подготавливающих замечания;
- библиотекарь проекта – хранитель всех версий разработки программного продукта;
- инструктор SADT – это специалист по непосредственной технологии SADT;
- руководитель проекта – ответственный за разработку

### **Достоинства SADT:**

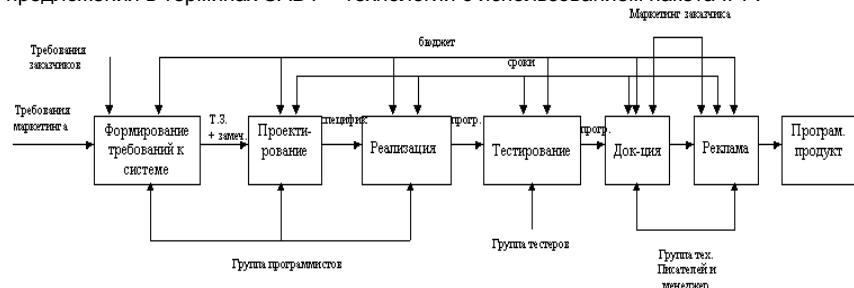
- система способствует организации коллективной работы, благодаря графическому представлению всего проекта;

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- письменные отчеты технических комитетов позволяют непосредственно производить контрольный анализ системы. На любой стадии возможно осуществить испытание системы;
- использование SADT позволяет осмысливать разработку не профессионалам – программистам;
- SADT позволяет использовать и другие методы проектирования. Например, на низких стадиях проектирования возможен переход из технологии Джексона в блок-схемы и т.д.

SADT - это единственная методология с помощью которой легко представить такие аспекты системы как управление, обратная связь, механизмы реализации. Это объясняется тем, что она возникла при проектировании систем более общего назначения;

С 81 года BBC США потребовала, чтобы все фирмы, конкурирующие за заключение договоров с этим ведомством, представили и обосновали свои предложения в терминах SADT – технологии с использованием пакета IPF.



Описание системы с помощью SADT называется моделью.

В SADT-моделях используются как естественный, так и графический языки.

Для передачи информации о конкретной системе источником естественного языка служат люди, описывающие систему, а источником графического языка - сама методология SADT.

SADT-модель дает полное, точное и адекватное описание системы, имеющее конкретное назначение. Это назначение, называемое целью модели, вытекает из формального определения модели в SADT:

Модель является некоторым толкованием системы. Поэтому субъектом моделирования служит сама система. Однако моделируемая система никогда не существует изолированно: она всегда связана с окружающей средой. Причем зачастую трудно сказать, где кончается система и начинается среда. По этой причине в методологии SADT подчеркивается необходимость точного определения границ системы. SADT-модель всегда ограничивает свой субъект, т.е. модель устанавливает точно, что является и что не является субъектом моделирования, описывая то, что входит в систему, и подразумевая то, что лежит за ее пределами. Ограничивая субъект, SADT-модель помогает сконцентрировать внимание именно на описываемой системе и позволяет избежать включения посторонних субъектов. Вот почему SADT-модель должна иметь единственный субъект.

С определением модели тесно связана позиция, с которой наблюдается система и создается ее модель. Поскольку качество описания системы резко снижается, если оно не сфокусировано ни на чем, SADT требует, чтобы модель рассматривалась все время с одной и той же позиции. Эта позиция называется "точкой зрения" данной модели.

"Точку зрения" лучше всего представлять себе как место (позицию) человека или объекта, в которое надо встать, чтобы увидеть систему в действии

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

После того как определены субъект, цель и точка зрения модели, начинается первая интеграция процесса моделирования по методологии SADT.

Субъект определяет, что включить в модель, а что исключить из нее.

Точка зрения диктует автору модели выбор нужной информации о субъекте и форму ее подачи.

После того как определены субъект, цель и точка зрения модели, начинается первая интеграция процесса моделирования по методологии SADT.

Субъект определяет, что включить в модель, а что исключить из нее.

Точка зрения диктует автору модели выбор нужной информации о субъекте и форму ее подачи.

SADT-диаграмма содержит от трех до шести блоков, связанных дугами, и имеет при построении модели несколько версий. Для того чтобы различать версии одной и той же диаграммы, используются С-номера. Блоки на диаграмме изображают системные функции, а дуги изображают множество различных объектов системы. Блоки обычно располагаются на диаграмме в соответствии с порядком их доминирования, т.е. их важностью относительно друг друга. Дуги, связывающие блоки, изображают наборы объектов и могут разветвляться и соединяться различными сложными способами. Однако, разветвляясь и соединяясь, дуги должны во всех случаях сохранять представляемые ими объекты.

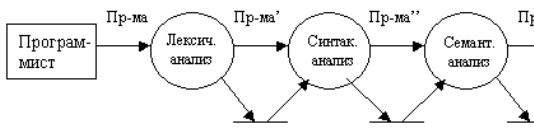
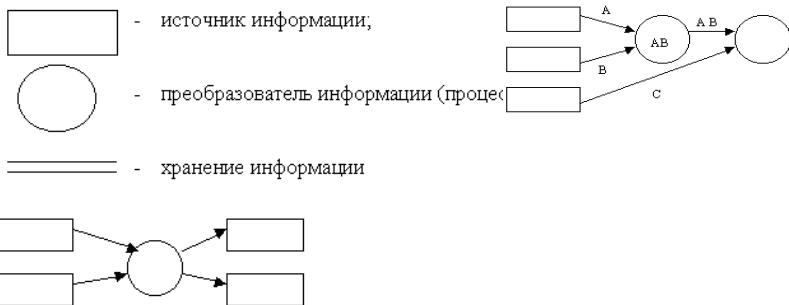
SADT-диаграммы являются декомпозициями ограниченных объектов. Объект ограничивается блоком и касающимися его дугами. Диаграмма, содержащая границу, называется родительской диаграммой, а диаграмма, декомпозирующая блок родительской диаграммы, называется диаграммой-потомком. Для связывания родительской диаграммы и диаграммы-потомка используются С-номера, так что модель всегда сохраняет актуальность. Коды ICOM используются для того, чтобы стыковать диаграмму-потомка с родительской диаграммой. Номер узла идентифицирует уровень данной диаграммы в иерархии модели. Когда диаграммы в модели становятся слишком трудными для чтения, для упрощения описания системы могут разумным образом использоваться специальные технические приемы типа "вхождения дуг в тоннель".

## **13. Информационно-потоковая технология проектирования. Область применения? Пример.**

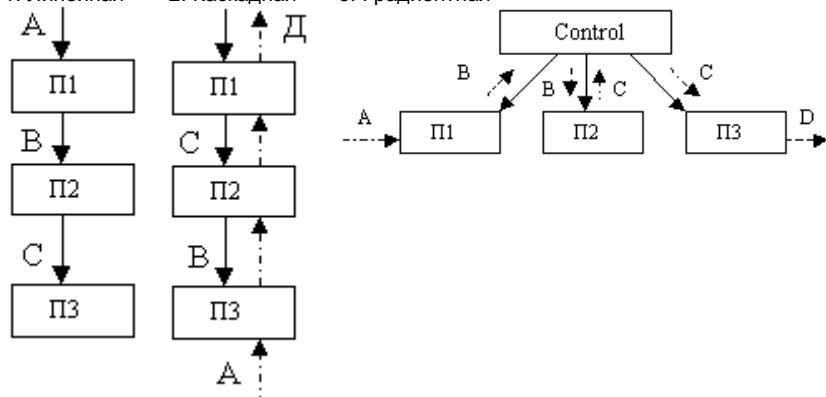
Метод пытается бороться с недостатком функциональной декомпозиции, который не предлагает никакого критерия для разбиения функций на подфункции. Данный метод предлагает структуру программы строить, отталкиваясь от потока данных всей задачи с применением приемов анализа проекта. Суть заключается в следующем:

- 1)Определяются потоки данных и строится граф потока данных системы.
- 2)Определяются входные, центральные выходные преобразующие элементы (процессы).
- 3)На основе ИП графа строится структура программы, причем моделей структуры программы может быть много. На основе анализа задачи данных выбирается одна модель.
- 4)Детализируется и оптимизируется структура программы, сформированная на третьем шаге.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

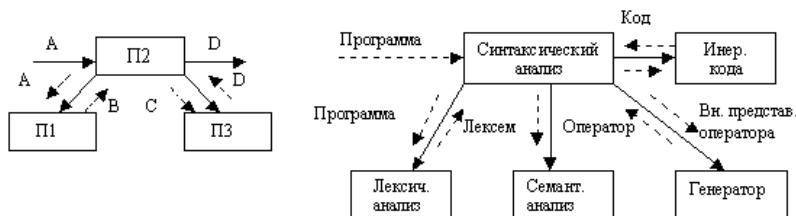


- 1) переход от ИП графа к структурной модели:
1. Линейная
  2. Каскадная
  3. Градиентная

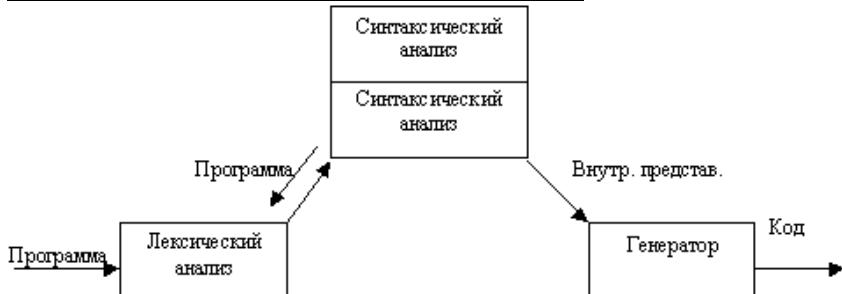


### 4. Иерархическая

Она предполагает, что некоторый процесс будет преобразован в блок-программы, которые по иерархии будут выше других



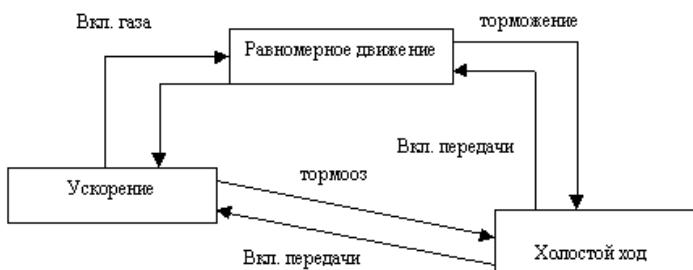
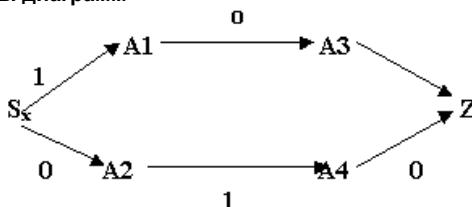
### 5. Смешанная структура



- 2) После построения всевозможных моделей структуры системы на основе анализа выбирается лучшая.
- 3) Проводится сквозное тестирование, строится некоторый тест. Этот тест прогоняется на информационно-потоковых моделях и на полученной (или выбранной) структуре системы.

Разновидностью ИПД является диаграмма переходов состояния.

#### Автоматы диаграмм



Если программная система легко описывается автоматом, то в этом случае составляется таблица переходов, а программируется таблицу или работа с ней достаточно просто.

## 14. Экстремальное программирование.

Экстремальное программирование (далее ХР) – методология создания ПО, позволяет сделать этот процесс более прогнозируемым, гибким и быстрым, в соответствии с требованиями современного бизнеса. ХР основывается на идеи адаптации изменений в программном проекте вместе с отказом

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

от детального планирования. XP отходит от традиционного процесса создания программной системы и вместо единоразового планирования, анализа и проектирования с расчетом на долгосрочную перспективу при XP все эти операции реализуются постепенно в ходе разработки, добиваясь тем самым значительного сокращения времени разработки и стоимости изменений в программе. Возникло в первой половине 90-х годов. Автор термина XP Кент Бек – пришел к выводу, что разработку любого программного проекта можно сделать более эффективной, если приложить усилия в четырех основных направлениях: усовершенствовать взаимосвязь разработчиков, упростить проектные решения, усилить обратную связь с заказчиком и проявлять больше активности. Эти четыре направления и стали приоритетными в XP. **Основные концепции Экстремального Программирования:**

### **Планирование:**

- Пишутся User Stories. Через которые заказчик рассказывает какую программу он хочет получить.
- Собственно план создается в результате Планирования Релиза - определяет даты релизов и формулировки заказчика, которые будут воплощены в каждом из них.
- Выпускать частые небольшие Релизы – Заказчик всегда имеет работающую версию программы с последними реализованными фичами.
- Измеряется Скорость проекта – позволяет понять все ли мы делаем правильно и укладываемся ли в срок.
- Проект делится на Итерации – это позволяет получать отдачу от заказчика и корректировать программу.
- Каждая итерация начинается с собрания по планированию.
- Люди постоянно меняются задачами – знания по проекту распространяются в команде.
- Каждый день начинается с утреннего Собрания стоя.
- XP правила корректируются, если что-то не так – это добавляет гибкость методологии.

### **Дизайн:**

- Простота. Все фичи реализуются максимально простым способом.
- Писать Пробные решения для уменьшения риска.
- Не добавлять никаких функций раньше времени. Добавлять только ту функциональность которая действительно требуется в данный момент.
- Рефакторить безжалостно - Упрощать написанный код.

### **Кодирование:**

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Заказчик всегда рядом. Заказчик является членом команды и отвечает на вопросы разработчиков.
- Весь код должен соответствовать принятому стандарту.
- Любая строчка программы написана 2 программистами.
- Частая интеграция кода – избавляет от трудностей интеграции модулей.
- Оставлять оптимизацию на потом.

### **Тестирование:**

- Любой код должен иметь Unit Test. Тесты пишутся до написания кода.
- Все Unit тесты должны проходить перед отдачей.
- Если найден баг то тесты корректируются или создаются.
- Функциональные тесты периодически выполняются и их результаты публикуются.

### **Плюсы:**

- XP строится на том, что создавать простую и понятную программу выгоднее, чем сложную и запутанную.
- В XP тестированию уделяется особое внимание. Тесты разрабатываются до того, как начнется написание программы, во время работы и после того, как кодирование завершено.
- Готовность программистов к постоянным изменениям в проекте. За счет постоянной обратной связи с заказчиком ЭП позволяет вносить изменения именно на той стадии, когда это действительно эффективно.

### **Минусы:**

- Невозможно использовать XP на гигантских проектах — оно подходит для небольших групп программистов (от 2 до 10 человек).
- Не подходит для распределенных команд, связанных между собой с помощью Интернета.

### **Ключевые концепции XP**

Ценности это то, что отличает набор индивидуалов от команды. Кент Бек в своей книге "Extreme Programming Explained: Embrace Change" выделил основные ценности XP:

Общение. Довольно часто в проектах возникают проблемы, если кто-то не сказал кому-то что-то с некоторой точки зрения важное. XP делает практически невозможным не общаться.

Простота. XP предлагает в процессе написания кода всегда делать самую простую вещь, которая смогла бы работать. Бек описывает это так: "XP делает ставку на то, что лучше сегодня сделать простую вещь ... чем более сложную, но которая все равно никогда не будет использоваться".

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Обратная связь. Постоянная обратная связь с заказчиком, группой или реальными конечными пользователями дает Вам больше возможностей регулировать вашу работу. Обратная связь позволяет Вам придерживаться правильного пути.

Смелость. Смелость подразумевается в контексте трех остальных положений, которые поддерживают друг друга.

-Требуется смелость предположить, что постоянная обратная связь лучше, чем попытка знать все с самого начала.

-Требуется смелость общаться с другими членами группы, что может продемонстрировать часть вашего собственного незнания.

-Требуется смелость сохранить простоту системы, откладывая завтрашние решения на завтра.

Без простой системы постоянного обмена знаниями и обратной связи для исправления ошибок, трудно быть смелым.

## **12 практик XP**

Практики XP действуют в совокупности поэтому изучение одной из них влечет за собой понимание и изучение других. После знакомства с этими принципами, станут понятны приемы, используемые в методике экстремального программирования

1)Игра планирования XP признает, в самом начале Вы не можете знать абсолютно все. Главная идея этого принципа состоит в том, чтобы быстро сделать приблизительную схему и дорабатывать ее, как только все становится более понятным.

### 2)Программирование в парах.

- В XP весь программный код пишут пары разработчиков, что предоставляет много экономических и прочих выгод:
- Все проектные решения принимаются, по крайней мере, двумя мозгами.
- Как минимум, два человека знакомы с каждой частью системы.
- Имеется меньшее количество шансов, что оба человека станут пренебречь тестированием или другими задачами.
- Замена в парах распространяет знания внутри группы.
- Код всегда проверяется, по крайней мере, одним человеком.
- Исследования также показывают, что программирование в парах является действительно более эффективным, чем одиночное программирование.

3)Тестирование. Есть два вида тестов в XP: функциональные тесты и тесты модулей. Функциональные тесты (приемочные тесты) пишутся на основе директивы заказчика. Они рассматривают систему как черный ящик. Заказчик ответственен за проверку корректности функциональных тестов.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Никакой код не может быть выпущен без Unit теста. Перед отдачей кода разработчик должен удостовериться что все тесты проходят без ошибок. Никто не может отдать код, если все не прошли 100%.

4)Рефакторинг (разложение программы на элементарные операции) - это методика улучшения кода без изменения его функциональных возможностей. XP-группа постоянно занимается рефакторингом.

5)Простой дизайн. XP выбирает самое простое решение.

Самый простой дизайн, который сможет работать это такой дизайн, который (по Кенту Беку):

- Выполняет все тесты;
- Не содержит дублирующийся код;
- Ясно отражает цели программистов для всего кода;
- Содержит наименьшее количество возможных классов и методов.

6)Коллективное владение кодом стимулирует разработчиков подавать идеи для всех частей проекта, а не только для своих модулей. Любой разработчик может изменять любой код для расширения функциональности, исправления ошибок или рефакторинга.

7)Непрерывное интегрирование кода помогает избегать кошмаров интегрирования. XP-группы интегрируют свой код несколько раз в день, после того, как они выполнили все тестирования модулей.

8)Доступный для связи заказчик. Чтобы оптимально функционировать, XP-группа должна иметь заказчика, расположенного недалеко, чтобы разъяснять директивы и принимать важные деловые решения.

9)Частые Релизы. Разработчики должны выпускать версии системы пользователям (или бета-тестерам) как можно чаще.

10)40-часовая рабочая неделя. Постоянное напряжение и интенсификация труда быстро истощает силы разработчиков, что заметно понижает эффективность труда.

11)Стандарт кодирования предохраняет группу от отвлечения на несущественные параметры тех вещей, которые не имеют такого значения, как продвижение с максимальной скоростью.

12)Метафора системы в XP аналогична тому, что большинство методологий называет архитектурой. Метафора дает группе непротиворечивое изображение, которое они могут использовать, чтобы описать, как работает существующая система, где нужны новые части и какую форму они должна принять.

## **15. RUP**

Статическую структуру RUP составляют описания работ и задач (части работы), описания создаваемых артефактов, а также рекомендации по их выполнению, которые группируются в дисциплины: шесть основных — бизнес-моделирование (Business Modeling), управление требованиями (Requirements), анализ и проектирование (Analysis and Design), реализация (Implementation), тестирование (Test), внедрение (Deployment), и три вспомогательных — управление конфигурациями и изменениями (Configuration and Change Management), управление проектом (Project Management), поддержка среды разработки (Environment).

Динамическую структуру процесса составляют фазы и итерации. Проект, как правило, делится на четыре фазы: начало (Inception), проработка (Elaboration), построение (Construction) и передача (Transition). Фазы, в свою очередь, делятся на итерации. В ходе каждой итерации выполняются работы и задачи из различных дисциплин; соотношение этих работ меняется в зависимости от фазы.

Работы и задачи в RUP привязаны к стандартному набору ролей участников процесса. Роли объединяют более узкие группы работ и задач, которые могут выполняться одним человеком с узкой специализацией. Как правило, реальный исполнитель выполняет одну или несколько ролей в соответствии со своей квалификацией. Скажем, менеджер проекта может выполнять также обязанности архитектора. Одну роль в рамках проекта могут выполнять и несколько человек.

Например, в проекте, как правило, участвует несколько разработчиков.

Создатели RUP определяют его как итеративный, архитектурно-ориентированный и управляемый прецедентами использования процесс разработки программного обеспечения. Согласно последней доступной автору версии RUP (Version 2003.06.00.65) к этому надо добавить, что RUP использует лучшие практические методы (итеративная разработка, управление требованиями, использование компонентной архитектуры, визуальное моделирование, непрерывный контроль качества, управление изменениями) и десять элементов, представляющих квинтэссенцию RUP (разработка концепции; управление по плану; снижение рисков и отслеживание их последствий; тщательная проверка экономического обоснования; использование компонентной архитектуры; прототипирование, инкрементное создание и тестирование продукта; регулярные

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

оценки результатов; управление изменениями; создание продукта, пригодного к употреблению; адаптация RUP под нужды своего проекта).

Пользоваться таким объемным определением, конечно, неудобно. Поэтому для характеристики RUP Первом Кроллом введено понятие «Дух RUP». Хотя оно не входит в «канонический» текст RUP, но предложено человеком, который, являясь директором соответствующего направления в компании IBM, связан с RUP самым непосредственным образом.

Дух RUP заключен в восьми принципах:

- атаковать риски как можно раньше, пока они сами не перешли в атаку;
- разрабатывать именно то, что нужно заказчику;
- главное внимание — исполняемой программе;
- приспосабливаться к изменениям с самого начала проекта;
- создавать архитектурный каркас как можно раньше;
- разрабатывать систему из компонентов;
- работать как одна команда;
- сделать качество стилем жизни.

Эти принципы весьма полно характеризуют RUP и в наибольшей степени соответствуют современному стилю разработки программного обеспечения.

Особенностью RUP является то, что в результате работы над проектом создаются и совершенствуются модели. Вместо создания громадного количества бумажных документов, RUP опирается на разработку и развитие семантически обогащенных моделей, всесторонне представляющих разрабатываемую систему. RUP — это руководство по тому, как эффективно использовать UML. Стандартный язык моделирования, используемый всеми членами группы, делает понятными для всех описания требований, проектирование и архитектуру системы.

RUP поддерживается инструментальными средствами, которые автоматизируют большие разделы процесса. Они используются для создания и совершенствования различных промежуточных продуктов на различных этапах процесса создания ПО, например, при визуальном моделировании, программировании, тестировании и т. д.

RUP — это конфигурируемый процесс, поскольку, вполне понятно, что невозможно создать единого руководства на все случаи разработки ПО. RUP пригоден как для маленьких групп разработчиков, так и для больших организаций, занимающихся

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

созданием ПО. В основе RUP лежит простая и понятная архитектура процесса, которая обеспечивает общность для целого семейства процессов. Более того, RUP может конфигурироваться для учёта различных ситуаций. В его состав входит Development Kit, который обеспечивает поддержку процесса конфигурирования под нужды конкретных организаций. RUP описывает, как эффективно применять коммерчески обоснованные и практически опробованные подходы к разработке ПО для коллективов разработчиков, где каждый из членов получает преимущества от использования передового опыта в:

- итерационной разработке ПО,
- управлении требованиями,
- использовании компонентной архитектуры,
- визуальном моделировании,
- тестировании качества ПО,
- контроле за изменениями в ПО.

RUP организует работу над проектом в терминах последовательности действий (workflows), продуктов деятельности, исполнителей и других статических аспектов процесса с одной стороны, и в терминах циклов, фаз, итераций и временных отметок завершения определенных этапов в создании ПО (milestones), т. е. в терминах динамических аспектов процесса, с другой.

При итерационном подходе, каждая из фаз процесса разработки состоит из нескольких итераций, целью которых является последовательное осмысление стоящих проблем, наращивание эффективных решений и снижение риска потенциальных ошибок в проекте. В то же время, каждая из последовательностей действий по созданию ПО выполняется в течение нескольких фаз, проходя пики и спады активности.

Каждый цикл итерации проекта начинается с планирования того, что должно быть выполнено. Результатом выполнения должен быть значимый продукт. Заканчивается же цикл оценкой того, что было сделано и были ли цели достигнуты.

RUP достаточно обширен. Это набор рекомендаций и примеров по всем стадиям и fazам разработки программ. Хотя в основу этих рекомендаций положен многолетний опыт разработки программных систем, не для каждого проекта RUP подходит на сто процентов. Каждый программный проект по-своему уникален. Нельзя бездумно копировать чужой проект, создавая артефакты, имеющие незначительную ценность. Во многих небольших организациях по разработке программного обеспечения,

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

особенно в тех, которые не имеют собственной мощной системы разработки, RUP можно использовать «как есть» или в готовом виде. Также для максимального его приближения к нуждам, требованиям, характеристикам и ограничениям организации-разработчика процесс может быть уточнен, расширен и специфически настроен.

### **16. Документирование проекта**

Документирование — это написание и поддерживание документов, сопровождающих программное обеспечение. Существует следующие основные типы документации на ПО:

- требования — документация свойств, возможностей, характеристик и качеств системы;
- архитектурная/проектная — обзор программного обеспечения, включающий описание рабочей среды и принципов, которые должны быть использованы при создании ПО;
- техническая — документация на код, алгоритмы, интерфейсы, API;
- пользовательская — руководства для конечных пользователей, администраторов системы и другого персонала;
- маркетинговая — документация для маркетинга продукта и анализа рыночного спроса, рекламные материалы.

#### Стандарты документации

Стандарты обеспечивают совместимость между проектами. Это означает, что идеи или артефакты, разработанные для одного случая, могут быть перенесены и на другой. Стандарты улучшают понимание среди инженеров. Наиболее крупные компании создали стандарты разработки программного обеспечения. Некоторые заказчики, такие как Министерство обороны США, настаивают, чтобы подрядчики следовали их стандартам.

Практика многих компаний показывает, что просто издание и распространение стандартов не приводит к их принятию. Для того чтобы быть эффективными, стандарты должны восприниматься инженерами как нечто полезное для них, а не как набор препятствий. Кроме того, четкие и измеримые цели, требующие дисциплинированного и документированного подхода, обычно являются хорошим мотивом для разработчиков.

Хэмфири предлагает командам коллективно решать, какой из стандартов ведения документации им применять. Преимуществом является то, что компания при этом перебирает различные подходы, а в результате этого процесса, могут быть приняты наиболее выгодные варианты на долгое время работы.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Недостатком самостоятельного выбора стандарта командами является то, что группы, работающие в одной компании, зачастую выбирают разные стандарты. Это уменьшает возможности сравнения проектов и требует от инженеров, переключающихся на другой проект, изучения нового стандарта документации.

Организации могут допускать некоторую гибкость и автономность в вопросе создания документации, но при этом рассчитывать на получение определенной стандартной информации для улучшения всего процесса производства. Улучшение процесса включает в себя эволюционный мета-процесс (процесс, имеющий дело с другими процессами) внутри организации. Одним из примеров является модель зрелости возможностей (CMM), которая классифицирует организации, занимающиеся разработкой программного обеспечения, по пяти категориям возрастающих возможностей.

Перечисленные ниже организации публикуют важные стандарты. Не все стандарты могут быть абсолютно актуальны, так как совещания, требуемые для их создания, проходят намного медленнее, чем появление новых технологий на рынке.

- Институт инженеров по электротехнике и радиоэлектронике (IEEE) в течение многих лет остается очень активным в создании стандартов документации программного обеспечения. Большинство стандартов разработаны различными комитетами, состоящими из опытных и ответственных инженеров-профессионалов. Некоторые из стандартов IEEE стали также стандартами ANSI.
- Международная организация по стандартизации (ISO) имеет огромное влияние во всём мире, особенно среди организаций-производителей, имеющих дело с Евросоюзом (ЕС). ЕС предписывает следование стандартам ISO любой компании, имеющей дело со странами-членами Евросоюза, что является мощным стимулом для поддержания этих стандартов странами всего мира.
- Институт технологий разработки программного обеспечения (SEI) был учрежден Министерством обороны США в университете Карнеги-Меллон для поднятия уровня технологии программного обеспечения у подрядчиков Министерства обороны. Работа SEI также была принята многими коммерческими компаниями, которые считают улучшение процесса разработки программного обеспечения своей стратегической

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

корпоративной задачей.

- Консорциум по технологии манипулирования объектами (OMG) является некоммерческой организацией, в которую в качестве членов входят около 700 компаний. OMG устанавливает стандарты для распределенных объектно-ориентированных вычислений. В частности, OMG использует унифицированный язык моделирования UML в качестве своего стандарта для описания проектов.

Документы, сопровождающие проект, сильно различаются среди организаций, но примерно соответствуют водопадным фазам. Стандарт ISO 12207 является одним из примеров такого набора документов.

Когда стандарты ведения документации не применяются, инженерам приходится затрачивать огромное количество времени, самостоятельно приводя документы в порядок. Типичный набор документации с использованием терминологии IEEE показан на рис.



Использование набора документации для водопадного процесса не означает, что обязательно должна использоваться водопадная

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

модель. Однако если она не используется, то необходимо провести обновление всех документов и пополнять их каждый раз, когда применяются водопадные фазы. Это означает, что документы должны быть очень хорошо организованы.

Ниже приводится описание каждого документа из набора IEEE. Другие стандарты внутренне организованы по тому же принципу.

- SVVP (Software Verification and Validation Plan): план экспертизы программного обеспечения. Этот план определяет, каким образом и в какой последовательности должны проверяться стадии проекта, а также сам продукт на соответствие поставленным требованиям.  
Верификация — это процесс проверки правильности сборки приложения; валидация проверяет тот факт, что собран требуемый продукт. Зачастую валидацию и верификацию осуществляют сторонние организации, в этом случае экспертиза называется независимой (IV&V — Independent V&V).
- SQAP (Software Quality Assurance Plan): план контроля качества программного обеспечения. Этот план определяет, каким образом проект должен достигнуть соответствия установленному уровню качества.
- SCMP (Software Configuration Management Plan): план управления конфигурациями программного обеспечения. SCMP определяет, как и где должны храниться документы, программный код и их версии, а также устанавливает их взаимное соответствие. Было бы крайне неразумным начинать работу без такого плана, так как самый первый созданный документ обречен на изменения, но необходимо знать, как управлять этими изменениями до того, как будет начато составление документа. Средние и большие компании, как правило, стараются выработать единое управление конфигурациями для всех своих проектов. Таким образом, инженерам требуется только научиться следовать предписанным процедурам в соответствии с SCMP.
- SPMP (Software Project Management Plan): план управления программным проектом. Этот план определяет, каким образом управлять проектом. Обычно он соответствует известному процессу разработки, например стандартному процессу компании.
- SRS (Software Requirements Specification): спецификация требований к программному обеспечению. Этот документ определяет требования к приложению и является

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- подобием контракта и руководства для заказчика и разработчиков.
- SSD (Software Design Document): проектная документация программного обеспечения. SSD представляет архитектуру и детали проектирования приложения, обычно с использованием диаграмм объектных моделей и потоков данных.
  - STD (Software Test Documentation): документация по тестированию программного обеспечения. Этот документ описывает, каким образом должно проводиться тестирование приложения и его компонентов.

Иногда в проектах привлекается дополнительная документация. Документация для итеративной разработки может быть организована двумя способами. Некоторые документы, в частности SSD, могут содержать свою версию для каждой итерации. Другой способ — дописывать дополнения, которые появляются по мере развития приложения.

Унифицированный язык моделирования (UML) был разработан для стандартизации описания программных проектов, в особенности объектно-ориентированных. UML был принят в качестве стандарта консорциумом OMG.

### **17. Индивидуальный процесс разработки программного обеспечения (PSP). Оценка.**

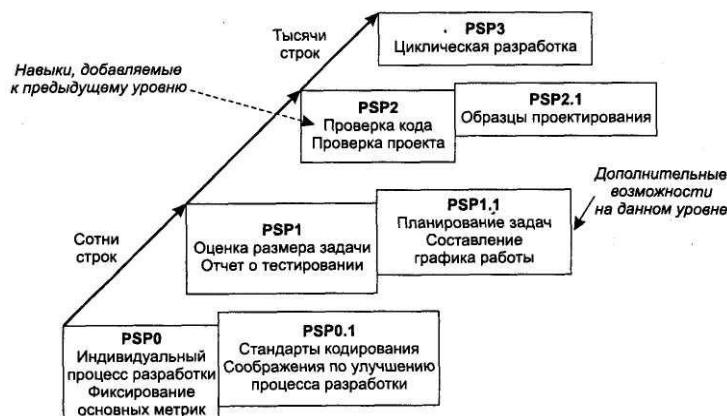
Индивидуальный процесс разработки программного обеспечения (Personal Software Process, PSP) предоставляет инженерам систему индивидуальной дисциплины при разработке программного обеспечения. Процесс PSP состоит из набора методов, форм и сценариев, указывающих разработчику ПО способы планирования, оценки и управления работой. PSP разработан для использования с любым языком программирования или методологией разработки.

PSP основан на следующих принципах планирования и качества:

- Все инженеры разные; чтобы быть наиболее эффективными, разработчики
- должны планировать свою работу в соответствии с личными качествами.
- Для согласованного улучшения производительности, разработчики должны индивидуально использовать точные и продуманные процессы.
- Чтобы производить качественные продукты, разработчики должны лично отвечать за качество своего продукта.
- Дешевле находить и исправлять дефекты как можно

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- раньше.
- Эффективнее предотвращать дефекты, чем находить и исправлять их.
  - Правильные способы всегда наиболее быстрые и дешёвые.
  - При использовании PSP, инженеры оценивают свою работу для составления более точных планов в будущем. В методы оценки входят:
  - Оценка времени — разработчики оценивают время, затраченное на каждую фазу разработки.
  - Оценка объема работы.
  - Оценка качества. Основные меры качества:
    - Распределение дефектов в коде.
    - Количество обзоров кода.
    - Соотношение времени разработки по фазам разработки.
    - Соотношение количества дефектов по фазам.
    - Отношение между количеством введённых и исправленных дефектов.
    - Количество дефектов в час.
    - Усилия для устранения дефекта.
    - Отношение затрат на оценку качества к затратам на исправление дефектов.



PSP делится на стадии постепенного роста, названные PSP0, PSP1, PSP2 и PSP3.

**PSP0:** Базовый процесс. Разрешает студенту пользоваться собственными методами разработки, но требует от него:

- записывать время, затраченное на работу над проектом;
- записывать найденные дефекты;

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- записывать типы дефектов.
- PSP0 дополняется PSP0.1. Предписывает разработчику выбрать:
- стандартное определение понятия «строка кода»;
- способ для указания того, как он мог бы улучшить свои навыки разработчика.

PSP1: Индивидуальный процесс планирования. Призван помочь инженеру полнее понять отношение между размером программы и временем, которое он тратит на её разработку. PSP1 должен сформировать упорядоченную систему, в рамках которой инженер может производить оценки, выполнять задачи, проверять статус работы и записывать результаты.

PSP1 добавляет следующие требования к PSP0:

- способность оценивать размер задачи;
- систематический подход к описанию результатов тестирования.
- PSP1 дополняется PSP1.1, который описывает способности:
- планировать программные задачи;
- распределять их по времени и составлять график работы.

PSP2: Индивидуальный процесс контроля качества. PSP2 разработан, чтобы помочь инженерам объективно и реалистично справляться с программными дефектами. Идея заключается в том, чтобы обнаружить и исправить максимальное количество дефектов перед сдачей работы до формальной проверки-инспектирования.

PSP2 включает в себя:

- индивидуальную проверку проекта и архитектуры;
- индивидуальную проверку кода.
- PSP2 дополняется PSP2.1, который включает:
- набор контрольных вопросов для проверки целостности избранных программных решений.

PSP3: Циклический индивидуальный процесс. PSP3 предназначен для применения процесса PSP к большим программным модулям путём разбиения больших программ на меньшие этапы.

PSP3 описывает:

- способ применения PSP к каждому этапу, что позволяет получить высококачественный результат, пригодный для следующей итерации;
- регressiveонное тестирование, которое призвано проверять, что тесты, разработанные для предыдущих итераций, успешно выполняются и на новых этапах.

## **18. Командный процесс разработки программного обеспечения (TSP). Оценка.**

В 1999 году Уоттс Хэмфри представил положительные результаты в постановке целей и процедур зрелости для командной разработки программ. Он назвал этот процесс командным процессом разработки программного обеспечения (TSP — Team Software Process).

Командный процесс управляет разработкой преимущественно программных продуктов командами разработчиков. TSP разработан для использования командами с количеством участников от 2 до 20.

Команда — группа людей с общей целью. Обязательными свойствами являются:

- Команда должна состоять минимум из двух человек.
- Члены команды должны работать в направлении общей цели.
- У каждого есть чёткая назначеннная роль.
- Выполнение задачи требует некоторой формы зависимости между членами

группы.

Эффективные команды должны обладать следующими характеристиками:

- Члены команды квалифицированы.
- Цель команды важна, определена, явна и реалистична.
- Ресурсы команды адекватны задачам.
- Члены команды мотивированы на достижение цели.
- Участники взаимодействуют и поддерживают друг друга.
- Члены команды дисциплинированы в своей работе.

Задачи TSP.

- Собрать самоуправляемые команды:
  - установить собственные цели;
  - составить свой процесс и планы;
  - отслеживать работу.
- Показать менеджерам, как управлять командами:
  - инструктаж;
  - мотивация;
  - поддержка максимальной производительности.
- Ускорить продвижение по шкале СММ:
  - сделать пятый уровень СММ нормой.
- Обеспечить пути улучшения для высокоразвитых организаций.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Содействовать университетскому образованию для команд промышленного уровня.

Ставка TSP на командную инициативу поддерживает высокую степень профессионализма среди программных разработчиков. Также заслуживает внимания упор TSP на внешнее управление командой. Управление предназначено не только для раздачи указаний и утверждения последних сроков, но и для обеспечения руководства, соответствующих инструментов и других необходимых ресурсов.

### **19. Основы структурных методов проектирования**

#### **Структурный анализ**

Структурным анализом принято называть метод исследования системы, которое начинается с её общего обзора и затем детализируется, приобретая иерархическую структуру с всё большим числом уровней.

Идеи, лежащие в основе структурных методов

Первым шагом упрощения сложной системы является её разбиение на чёрные ящики, при этом такое разбиение должно удовлетворять следующим критериям:

- каждый чёрный ящик должен реализовывать единственную функцию системы;
- функция каждого чёрного ящика должна быть легко понимаема независимо от сложности её реализации;
- связь между чёрными ящиками должна вводиться только при наличии связи между соответствующими функциями системы;
- связи между чёрными ящиками должны быть простыми, насколько это возможно, для обеспечения независимости между ними.

Второй важной идеей, лежащей в основе структурных методов является идея иерархии. Для понимания сложной системы недостаточно разбиения её на части, необходимо эти части организовать определенным образом, а именно в виде иерархических структур.

Наконец, третий момент: структурные методы широко используют графические нотации, также служащие для облегчения понимания сложных систем.

#### **Принципы структурного анализа**

В качестве двух базовых принципов используются следующие: принцип «разделяй и властвуй» и принцип иерархического упорядочивания. Первый является принципом решения трудных проблем путем разбиения их на множество меньших независимых задач, легких для понимания и решения. Второй

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

принцип декларирует, что устройство этих частей также существенно для понимания. Понимаемость проблемы резко повышается при организации её частей в древовидные иерархические структуры, т. е. система может быть понята и построена по уровням, каждый из которых добавляет новые детали.

Выделение двух базовых принципов инженерии программного обеспечения вовсе не означает, что остальные принципы являются второстепенными, игнорирование любого из них может привести к непредсказуемым последствиям (в том числе и к неуспеху всего проекта). Основные из таких принципов:

- Принцип абстрагирования — заключается в выделении существенных с некоторых позиций аспектов системы и отвлечение от несущественных с целью представления проблемы в простом общем виде.
- Принцип формализации — заключается в необходимости строгого методического подхода к решению проблемы.
- Принцип упаковывания — заключается в упаковывании несущественной на конкретном этапе информации: каждая часть «знает» только необходимую ей информацию.
- Принцип концептуальной общности — заключается в следовании единой философии на всех этапах жизненного цикла (структурный анализ — структурное проектирование — структурное программирование — структурное тестирование).
- Принцип полноты — заключается в контроле на присутствие лишних элементов.
- Принцип непротиворечивости — заключается в обоснованности и согласованности элементов.
- Принцип логической независимости — заключается в концентрации внимания на логическом проектировании для обеспечения независимости от физического проектирования.
- Принцип независимости данных — заключается в том, что модели данных должны быть проанализированы и спроектированы независимо от процессов их логической обработки, а также от их физической структуры и распределения.
- Принцип структурирования данных — заключается в том, что данные должны быть структурированы и иерархически организованы.
- Принцип доступа конечного пользователя — заключается

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

в том, что пользователь должен иметь средства доступа к данным, которые он может использовать непосредственно (без программирования).

Соблюдение указанных принципов необходимо при организации работ на начальных этапах жизненного цикла независимо от типа разрабатываемого ПО и используемых при этом методологии.

Средства структурного анализа и их взаимоотношения

Для целей моделирования систем вообще, и структурного анализа в частности, используются три группы средств, иллюстрирующих:

- функции, которые система должна выполнять;
- отношения между данными;
- зависящее от времени поведение системы (аспекты реального времени).

Среди всего многообразия средств решения данных задач в методологиях структурного анализа наиболее часто и эффективно применяемыми являются следующие:

- DFD (Data Flow Diagrams) — диаграммы потоков данных совместно со словарями данных и спецификациями процессов или миниспецификациями;
- ERD (Entity-Relationship Diagrams) — диаграммы «сущность-связь»;
- STD (State Transition Diagrams) — диаграммы переходов состояний;
- SADT — методология структурного анализа и проектирования, интегрирующая процесс моделирования, управление конфигурацией проекта, использование дополнительных языковых средств и руководство проектом со своим графическим языком.

Все они содержат графические и текстовые средства моделирования: первые — для удобства демонстрации основных компонент модели, вторые — для обеспечения точного определения её компонент и связей.

## **20. UML. Назначение**

UML (сокр. от англ. *Unified Modeling Language* — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения. UML является языком широкого профиля, это открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой *UML моделью*. UML был создан для определения,

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.  
визуализации, проектирования и документирования в основном программных систем.

Использование UML не ограничивается моделированием программного обеспечения. Его также используют для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

UML позволяет разработчикам ПО достигнуть соглашения в графических обозначениях для представления общих понятий (таких как класс, компонент, обобщение, объединение и поведение) и больше сконцентрироваться на проектировании и архитектуре.

В 1994 году Гради Буч и Джеймс Рамбо, работавшие в компании Rational Software, объединили свои усилия для создания нового языка объектно-ориентированного моделирования. В октябре 1995 года была выпущена предварительная версия 0.8 унифицированного метода (англ. *Unified Method*). Осенью 1995 года к компании Rational присоединился Айвар Якобсон.

На волне растущего интереса к UML к разработке новых версий языка в рамках консорциума UML Partners присоединились такие компании, как Digital Equipment Corporation, Hewlett-Packard, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle Corporation, Rational Software, Texas Instruments и Unisys. Результатом совместной работы стала спецификация UML 1.0, вышедшая в январе 1997 года.

Формальная спецификация последней версии UML 2.0 опубликована в августе 2005 года. Семантика языка была значительно уточнена и расширена для поддержки методологии Model Driven Development — MDD (англ.).

UML 1.4.2 принят в качестве международного стандарта ISO/IEC 19501:2005.

### Преимущества UML

- UML объектно-ориентированный, в результате чего методы описания результатов анализа и проектирования семантически близки к методам программирования на современных ОО-языках;
- UML позволяет описать систему практически со всех возможных точек зрения и разные аспекты поведения системы;
- Диаграммы UML сравнительно просты для чтения после достаточно быстрого ознакомления с его синтаксисом;
- UML расширяем и позволяет вводить собственные текстовые и графические стереотипы, что позволяет

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- применять не только в сфере программной инженерии;
- UML получил широкое распространение и динамично развивается.

### Основные диаграммы UML:

- Классов (class diagram)
- Объектов (object diagram)
- Использования (use case diagram)
- Последовательности (sequencediagram)
- Кооперации (collaboration diagram)
- Состояний (state chart diagram)
- Деятельности (activity diagram)
- Компонентов (component diagram)
- Размещения (deployment diagram)

Это язык для специфирования (создания спецификации), конструирования, визуализирования и документирования артифактов программных систем.

Артифактами, в контексте проектирования на UML, можно называть сделанные в процессе анализа и проектирования решения, которые определенным образом визуализированы с помощью различных диаграмм, условных обозначений на диаграммах и различных связей между этими обозначениями.

UML не привязан к какому-либо конкретному языку программирования, просто существуют инструменты моделирования поддерживающих UML, которые позволяют на основе описанной модели проверить ее "на корректность" и сгенерировать "скелет" системы или ее части.

UML как бы "снимает обертку" с того что может быть сделано с помощью существующих методов. В качестве примера, авторы UML воспользовались им при моделировании конкурентной, распределенной системы, для того чтобы убедиться, что он компетентно адресован и этой предметной области.

UML фокусируется на стандартизации языка моделирования, а не на стандартизации процесса моделирования. Хотя UML должен применяться в контексте этого процесса. Различные организации и различные предметные/проблемные области требуют различных процессов. Авторы UML предлагают процесс разработки, который: основан на сценариях, сфокусирован на архитектуре системы, является итеративным и поступательным.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Кроме этого UML может найти применение в самых различных областях человеческой жизнедеятельности, где требуется анализ и оптимизация функционирования систем любой природы.

UML специфицирует язык, который объединяет базовые концепции общепринятые в объектно-ориентированном сообществе. Он также позволяет выражать "отклонения" с помощью механизма расширения.

### **21. UML.Диаграмма вариантов использования (прецедентов) (use case diagram). Назначение. Пример использования. UML.Диаграмма классов (class diagram). Пример использования.**

Диаграмма прецедентов использования была впервые предложена Айваром Якобсоном в 1992 и быстро завоевала всеобщее признание за счет простоты и легкости восприятия и применения.

Суть ее состоит в следующем: проектируемая система представляется в виде наборов актеров, взаимодействующих с системой с помощью так называемых прецедентов использования.

Актером является любая сущность, взаимодействующая с системой извне. Им может быть человек, оборудование, другая система, то есть мы определяем, что взаимодействует с системой.

В свою очередь Прецедент Использования описывает, что система предоставляет актеру, то есть определяет некоторый набор транзакций, совершаемый актором при диалоге с системой, при этом ничего не говориться о том, каким образом будет реализовано взаимодействие. Прецедент — это типичное взаимодействие пользователя с системой, которое при этом:

- описывает видимую пользователем функцию,
- может представлять различные уровни детализации,
- обеспечивает достижение конкретной цели, важной для пользователя.

Детальное описание - удел других техник моделирования UML, диаграмма же прецедентов использования несет в себе высокий уровень абстракции, что позволяет еще на ранних этапах проекта определить и зафиксировать функциональные требования к системе и обеспечить гибкий и эффективный механизм взаимодействия между разработчиком и заказчиком проекта, определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Диаграммы использования описывают функциональность системы, которая будет видна пользователям системы. Позволяет:

1. Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы
2. Сформулировать общие требования к функциональному поведению проектируемой системы
3. Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей
4. Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Главное назначение - показать основные функции системы для каждого типа пользователей.

Отношение — семантическая связь между отдельными элементами модели. Ассоциация служит для обозначения специфической роли актера при его взаимодействии с отдельным вариантом использования. Включение (include) в языке UML — это разновидность отношения зависимости между базовым вариантом использования и его специальным случаем. При этом отношением зависимости является такое отношение между двумя элементами модели, при котором изменение одного элемента (независимого) приводит к изменению другого элемента (зависимого). Отношение расширения определяет взаимосвязь базового варианта использования с другим вариантом использования, функциональное поведение которого задействуется базовым не всегда, а только при выполнении дополнительных условий. Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми особенностями поведения родительских вариантов.



**Диаграмма классов** показывает статическую структуру части системы. Составляющими данного типа диаграмм являются классы, объекты и отношения между ними. Класс представлен прямоугольником с тремя разделами, в которых соответственно помещаются имя класса, атрибуты и операции. Схожая нотация применяется и для объектов - экземпляров класса, с тем различием, что к имени класса добавляется имя объекта и вся надпись подчеркивается. Нотация UML предоставляет широкие возможности для отображения дополнительной информации - абстрактные операции и классы, стереотипы, общие и частные методы, интерфейсы, параметризованные классы и т.д. Отношение обобщения также имеет собственную графическую нотацию в виде треугольника и связующей линии, позволяя представить иерархию наследования: от суперкласса к подклассам.

Класс — абстрактное описание множества однородных объектов, имеющих одинаковые атрибуты, операции и отношения с объектами других классов.



**Атрибут** — содержательная характеристика класса, описывающая множество значений, которые могут принимать отдельные объекты этого класса. Отражает некоторое свойство моделируемой сущности + - public, # - protected, - - private.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Квантор видимости может быть опущен. **Операция** - это сервис, предоставляемый каждым экземпляром или объектом класса по требованию своих клиентов, в качестве которых могут выступать другие объекты, в том числе и экземпляры данного класса.



**Ассоциация** — это отношение, показывающее, что объекты одного типа неким образом связаны с объектами другого типа. При необходимости направление навигации может задаваться стрелкой. Допускается задание ассоциаций на одном классе.

**Обобщение**- таксономическое отношение между более общим понятием и менее общим понятием. Менее общий элемент модели должен быть согласован с более общим элементом и может содержать дополнительную информацию. Данное отношение используется для представления иерархических взаимосвязей между различными элементами языка UML.

**Наследование** - специальный механизм, посредством которого более специальные элементы включают в себя структуру и поведение более общих элементов.

**Агрегация** - специальная форма ассоциации, которая служит для представления отношения типа "часть-целое" между агрегатором (целое) и его составной частью. Отличие от обобщения заключается в том, что части системы никак не обязаны наследовать ее свойства и поведение, поскольку являются самостоятельными сущностями.

## 22. UML. Диаграммы поведения (behavior diagrams).

### Назначение. Пример использования.

Диаграммы поведения:

- Вариантов использования
- Состояний

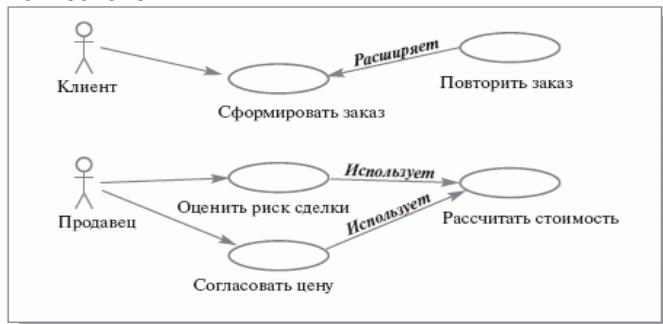
## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Активности
- Диаграммы взаимодействия:
  - Коммуникации (UML 2.0) / Кооперации (UML 1.x)
  - Обзора взаимодействия (UML 2.0)
  - Последовательности
  - Синхронизации (UML 2.0)

Суть *диаграммы прецедентов использования* состоит в следующем: проектируемая система представляется в виде наборов актеров, взаимодействующих с системой с помощью так называемых прецедентов использования.

Прецедент — это типичное взаимодействие пользователя с системой, которое при этом:

- описывает видимую пользователем функцию,
- может представлять различные уровни детализации,
- обеспечивает достижение конкретной цели, важной для пользователя.



Диаграммы состояний описывают поведение объекта во времени, то есть моделируют все возможные изменения в состоянии объекта, вызванные внешними воздействиями со стороны других объектов или извне. Диаграммы Состояний применяются для описания поведения объектов и для описания операций классов. Описывается изменение состояния только одного класса или объекта. Каждое состояние объекта представляется на Диаграмме Состояний в виде прямоугольника с закругленными углами, содержащего имя состояния, и, возможно, значение атрибутов объекта в данный момент времени. Переход осуществляется при наступлении некоторого события: получении объектом сообщения или приемом сигнала и изображается в виде стрелки, соединяющей два соседних состояния. Имя событие указывается на переходе. Кроме того, на переходе могут указываться действия, производимые объектом в ответ на

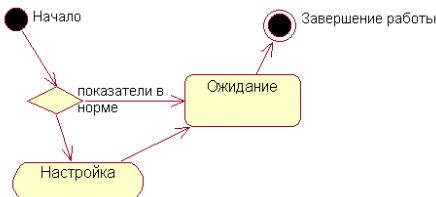
## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

внешние события (при переходе из одного состояния в другое или при нахождении в определенном состоянии).



**Диаграммы Активности** - частный случай Диаграмм Состояний. Каждое состояние - это выполнение некоторой операции, и переход в следующее состояние срабатывает только при завершении операции в исходном состоянии. Таким образом, реализуется принцип процедурного, синхронного управления, обусловленного завершением внутренних действий. Описываемое состояние не имеет внутренних переходов и переходов по внешним событиям.

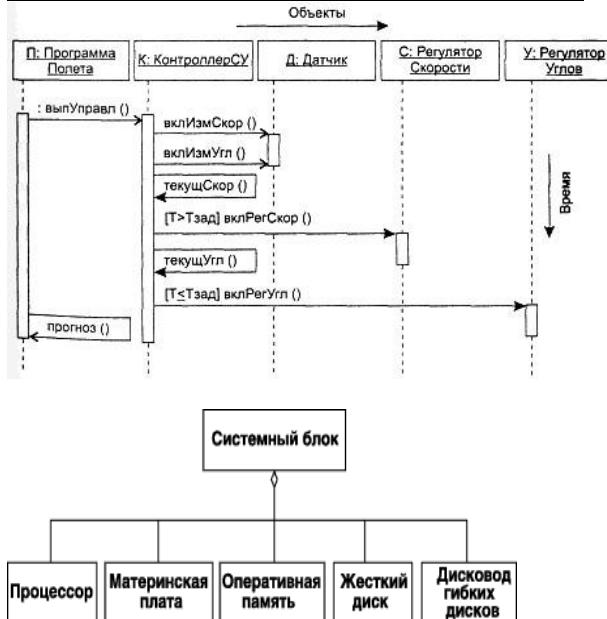
Основным направлением использования Диаграммы Активности является описание операций классов, когда необходимо представить алгоритм ее реализации, при этом каждый шаг является выполнением операции некоторого класса.



**Диаграмма Следования** делает упор на временную последовательность передаваемых сообщений, на диаграмме изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами. Таким образом, для Диаграмм Следования ключевым моментом является динамика взаимодействия.

Диаграмма Следования имеет два измерения. Одно - слева направо в виде вертикальных линий, изображающих объекты, участвующие во взаимодействии. Верхняя часть линий дополняется прямоугольником, содержащим имя класса объекта или имя экземпляра объекта. Второе измерение - вертикальная временная ось. Сообщения, посылаемые одним объектом другому, изображаются в виде стрелок с именем сообщения и упорядочены по времени возникновения.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.



**Композиция** - разновидность отношения агрегации, при которой составные части целого имеют такое же время жизни, что и само целое. Эти части уничтожаются вместе с уничтожением целого.



### 23. UML. Диаграмма состояний (statechart diagram).

**Назначение.** Пример использования.

Диаграмма состояний используется для представления поведения системы, представленной конечным числом состояний. Есть много форм диаграмм состояний, которые немного отличаются и имеют разную семантику. Каждая диаграмма обычно представляет объекты единственного класса и переходы между разными состояниями объекта при

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

срабатывании определенных событий. Была представлена в 1967 Taylor Booth в его книге "Sequential Machines and Automata Theory". Вольном переводе, можно определить как «Конечные автоматы».

### Направленный граф

Классическая форма диаграммы состояний для системы с конечным числом состояний – направленный граф со следующими элементами:

- **Состояния  $Q$ :** конечный набор вершин, обычно представленных окружностями, с уникальными обозначениями, подписываемыми внутри этих окружностей;
- **Входные символы  $\Sigma$ :** конечный набор входных символов;
- **Выходные символы  $Z$ :** конечный набор выходных символов;

Функция  $\omega$  представляет соответствие входных символов – выходным. Математически она описывается следующим образом:  $\omega : \Sigma \times Q \rightarrow Z$ .

- **Переходы  $\delta$ :** представляют переходы между двумя состояниями, вызываемые при помощи входных данных (идентифицируются при помощи своих обозначений, подписываемых на ребрах). ‘Переход’ обычно рисуется как стрелка, указывающая из текущего состояния в следующее. Это соответствие раскрывает смены состояний, которые должны происходить при вводе определенного символа. Оно описывается математически следующим образом:  $\delta : \Sigma \times Q \rightarrow Z$
- **Начальное состояние  $q_0$ :** начальное состояние  $q_0 \in Q$  обычно представляется при помощи стрелки не выходящей из какого-то определенного состояния. В более старых текстах, начальное состояние не показывалось, а должно было описываться в тексте.
- **Конечное состояни(е,я)  $F$ :** конечное состояния  $F \subseteq Q$ . Обычно изображается в виде двойного круга.

Для детерминированных, недетерминированных, обобщенных недетерминированных конечных автоматов и автомата Мура входные данные описывают каждый переход. Для автомата Мили, вход и выход назначаются на каждый переход, разделенные "/": "1/0" обозначает, что состояние меняется при

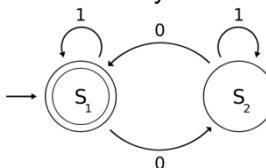
## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

получении символа "1" и на выход дает "0". Для автомата Мура выходной символ для состояния обычно пишется внутри окружности состояния, так же отделенный от обозначения состояния при помощи "/". Есть так же варианты, комбинирующие эти две нотации.

Например, если состояние имеет следующие выходные значения: "a=1, b=0" диаграмма должна отразить это: например, "q5/1,0" обозначает состояние q5 с выходными значениями a=1, b=0. Это обозначение будет написано внутри окружности состояния.

Пример: DFA, NFA, GNFA, или Moore machine

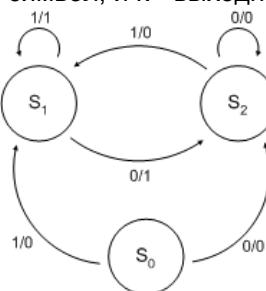
(детерминированный, недетерминированный, обобщенный недетерминированный конечные автоматы или автомат Мура) S<sub>1</sub> и S<sub>2</sub> состояния, и S<sub>1</sub> – конечное состояние. Каждый переход подписан входным символом. Этот пример показывает автомат, принимающий строки из символов {0, 1}, которые содержат четное количество нулей.



Пример: Mealy machine

(автомат Мили)

S<sub>0</sub>, S<sub>1</sub>, и S<sub>2</sub> состояния. Каждый переход обозначен как "j / k" где j – входной символ, и k - выходной.



Harel statechart

(диаграмма состояний Харела)

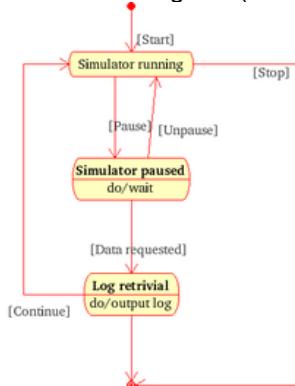
Диаграммы состояний Harel получают широкое применение с тех пор, как стали частью UML.

Классические диаграммы состояний дизъюнктивны ("или"), потому что автомат может быть только в одном из возможных состояний. С диаграммами состояний Harel стало возможным моделировать "и"-автоматы, где автомат может находиться в двух и более состояниях одновременно. В соответствие с этим,

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

появляется возможность моделирования сверхсостояний и пересекающихся (совпадающих) автоматов.

### UML state diagram (UML диаграмма состояний)



Пример UML диаграммы состояний.

Диаграмма состояний UML (Unified Modeling Language) по существу – диаграмма состояний Harel со стандартизированной нотацией, которая может описать множество систем, начиная компьютерными программами и заканчивая бизнес-процессами. Ниже перечислены основные элементы нотации, которые могут быть использованы для создания диаграмм:

- Закрашенный круг – начальное состояние.
- Полый круг, с маленьким закрашенным кругом внутри – конечное состояние (если таковое имеется)
- Закругленный четырехугольник указывает на состояние. Верхняя часть четырехугольника содержит имя состояния. Может содержать горизонтальную линию в середине, ниже которой располагаются действия, производимые в этом состоянии.
- Стрелка – переход. Имя события, (если есть) вызывающего этот переход, подписывается рядом со стрелкой. Можно добавить выражение (guard expression) перед "/", заключенное в квадратные скобки ( **eventName[guardExpression]** ), что будет означать, что это выражение должно быть истинным для того, что переход произошел. Если действие выполняется в течение этого перехода, то оно добавляется к записи после "/" ( **eventName[guardExpression]/action** ).

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Жирная горизонтальная линия с либо  $x>1$  линиями, входящими и одной выходящей, либо с 1 входящей и  $x>1$  выходящими линиями. Это обозначает объединение и разделение, соответственно.

### **24. UML. Диаграмма активности (activity diagram) .**

#### **Назначение. Пример использования.**

Диаграммы активности – это свободно определенная диаграммная техника для отображения последовательности шагов для поэтапных действий с поддержкой условий, итеративности и параллельности. В UML (Unified Modeling Language), диаграммы активности могут быть использованы для того, чтобы описать последовательность действий компонентов в системе. Диаграмма активности отображает общий поток управления.

В UML 1.x, диаграмма активности – это вариация диаграммы состояний (UML State diagram), в которой “состояния” представляют активности (действия), а переходы представляют результаты соответствующих действий.

Диаграммы активностей обычно используется для моделирования бизнес-процессов. Они состоят из:

- Закрашенный круг – начальный узел(initial node).
- Полый круг, с маленьким закрашенным кругом внутри – конечный узел (если имеется) (activity final node)
- Закругленный четырехугольник указывает на промежуточные действия активности.
- Стрелка – переход между действиями.
- Жирная горизонтальная линия с либо  $x>1$  линиями, входящими и одной выходящей, либо с 1 входящей и  $x>1$  выходящими линиями. Это обозначает объединение и разделение, соответственно.
- Так же при помощи ромба, можно задавать условные переходы (внутри ромба указывается выражение, на исходящих переходах – значения, при которых происходит данный переход).

Начальный узел – начальная точка на диаграмме.

Конечный узел – конечная точка диаграммы (их может быть 0 и больше).

for(A;B;C)

D;

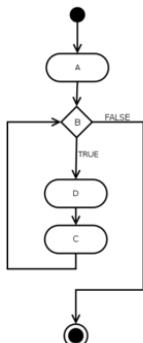


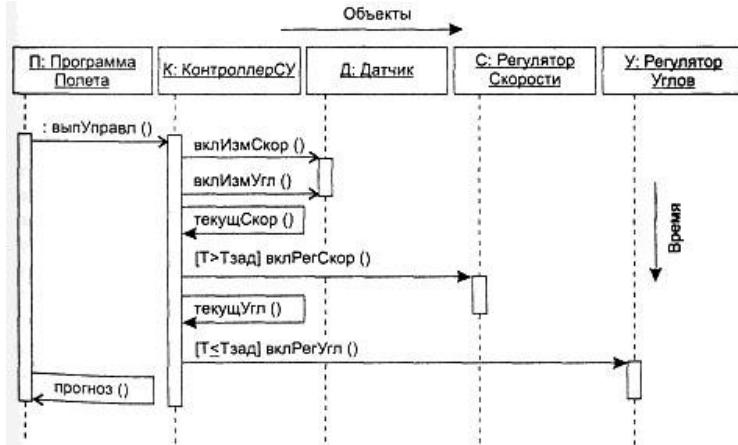
Диаграмма активности для цикла for.

## 25. UML. Диаграммы взаимодействия (interaction diagrams). Назначение. Пример использования.

Взаимодействие между объектами в системе представляются диаграммами взаимодействия. В унифицированном языке моделирования UML диаграммы взаимодействия (interaction diagrams) делятся на два вида: диаграммы последовательности (sequence diagram) и диаграммы кооперации (collaboration diagram).

Диаграмма последовательности делает упор на временную последовательность передаваемых сообщений, важен порядок, вид и имя сообщения, на диаграмме изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами. Таким образом, для диаграмм последовательности ключевым моментом является динамика взаимодействия.

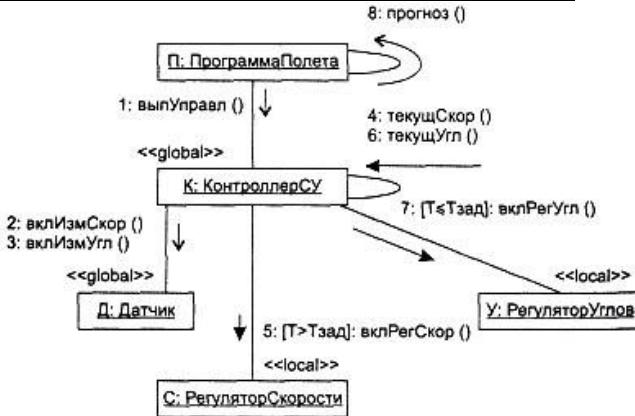
Диаграмма последовательности имеет два измерения. Одно - слева направо в виде вертикальных линий, изображающих объекты, участвующие во взаимодействии. Верхняя часть линий дополняется прямоугольником, содержащим имя класса объекта или имя экземпляра объекта. Второе измерение - вертикальная временная ось. Сообщения, посылаемые одним объектом другому, изображаются в виде стрелок с именем сообщения и упорядочены по времени возникновения.



Для диаграммы кооперации главным является возможность отобразить не только последовательность взаимодействия, сколько все окружение объектов, участвующих в нем. То есть показаны не только посылаемые и принимаемые сообщения, но и косвенные связи между ассоциированными объектами. Говорят, что диаграммы кооперации описывают полный контекст взаимодействия и представляют собой своеобразный временной "срез" конфигурации сети объектов, взаимодействующих для выполнения определенной бизнес-цели программной системы.

Диаграмма кооперации изображает объекты, участвующие во взаимодействии в виде прямоугольников, содержащих имя объекта, его класс и, возможно, значение атрибутов.

Ассоциации между объектами, как и на диаграммах классов, изображаются в виде соединительных линий. Возможно указание имени ассоциации и ролей, которые играют объекты в данной ассоциации. Динамические связи - потоки сообщений, представляются также в виде соединительных линий между объектами, сверху которых располагается стрелка с указанием направления и имени сообщения.



Таким образом, диаграмма последовательностей - диаграмма взаимодействия, в которой основной акцент сделан на упорядочении сообщений во времени, а диаграмма кооперации - диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения.

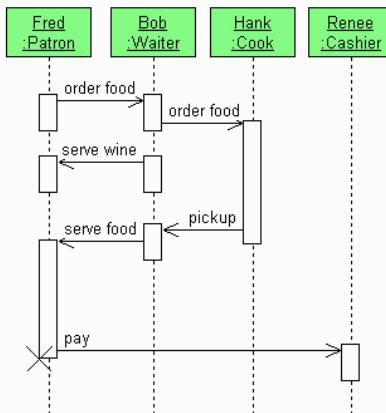
## 26. UML. Диаграмма последовательности (sequence diagram). Назначение. Пример использования.

Диаграмма последовательности – это вид диаграммы взаимодействия в UML (Unified Modeling Language), которая показывает, как процессы взаимодействуют друг с другом, и в каком порядке.

Диаграммы последовательности иногда называют Event-trace diagrams (диаграммы прослеживания событий) и event scenarios (сценарии событий).

Диаграмма последовательности, показывает параллельные вертикальные линии ("lifelines" – линии жизни объектов) в качестве различных процессов или объектов, которые "живут" в одно и то же время. Горизонтальные стрелки показывают сообщения, которыми они обмениваются, и расставляются в том порядке, в котором они происходят. Это разрешает спецификацию простых динамических сценариев в графическом виде.

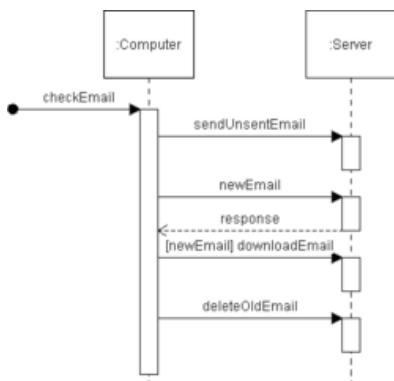
Например, следующая UML 1.x диаграмма описывает последовательность сообщений простой системы ресторана (слева направо: Fred – посетитель, Bob – официант, Hank – повар, Renee - кассир). Обратите внимание, что стрелки на чертеже двух типов!



Простая диаграмма последовательности ресторана

Эта диаграмма представляет, как посетитель заказывает еду и вино, пьет вино, затем обедает, и, затем, оплачивает еду. Пунктирные линии (или линии жизни) показывают временной отрезок, при этом: чем ниже точка на линии, тем более позднему моменту времени она соответствует (время протекает сверху вниз). Стрелки представляют сообщения от одного действующего лица или объекта к другим. Например, “посетитель отправляет сообщение ‘pay’ кассиру”. “Полустрелки” (вид стрелок с незаполненной головкой) указывают асинхронные вызовы метода. То есть это те вызовы, которые могут не совпадать по времени.

Диаграмма последовательности UML 2.0 придерживается похожей нотации. Здесь добавлена поддержка моделирования вариаций к стандартному потоку событий.



Пример UML 2 диаграммы

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Если сущность является объектом, то ее имя подчеркивается. Если – роль, то не подчеркивается. Оставляя имя сущности пустым, можно представлять анонимные и безымянные сущности.

Сообщения используются для того, чтобы показать взаимодействие. Они бывают трех типов:

1. Сплошная, полная головка стрелки – синхронный вызов.
2. Сплошная, полая головка стрелки – асинхронный вызов.
3. Штриховая, полая головка стрелки – возвращаемое сообщение.

Это определение верно для UML 2, значительно отличающегося от UML 1.x.

Боксы активации или боксы вызова методов – это четырехугольники, нарисованные на временных линиях сущностей, для представления того, какие процессы будут выполнены в ответ на сообщение.

Когда объект уничтожается (удаляется из памяти), “X” ставится в верхней части линии жизни объекта, а сама линия прекращает рисоваться дальше. Это должно быть результатом сообщения, либо от самого объекта, либо от другого объекта.

Сообщение, отправленное извне, может быть представлено как сообщение исходящее из закрашенного круга или из границы диаграммы последовательности.

UML диаграмма может выполнять серию шагов, называемых superstep (супершаг), в ответ на одно-единственное внешнее сообщение.

Некоторые системы имеют простое динамическое поведение, которое может быть выражено при помощи специфических последовательностей сообщений между небольшим, фиксированным числом объектов или процессов. В таких случаях, диаграммы последовательностей могут полностью описать поведение системы. Зачастую, поведение бывает более сложным, например, когда число взаимодействующих объектов велико или переменно. Или, например, есть много точек ветвления (например, исключения). В таких случаях, диаграммы последовательности не могут полностью раскрыть поведение системы, но могу специфицировать типичные способы использования (use cases) системы, небольшие детали в ее поведении и его упрощенный обзор.

**27. UML. Диаграмма кооперации (collaboration diagram) .****Назначение. Пример использования.**

Одна из четырех диаграмм взаимодействия.

Диаграмма кооперации моделирует взаимодействия между объектами или частями с точки зрения направляемых сообщений. Диаграмма коопераций представляет комбинацию информации, получаемой из диаграмм классов, последовательностей и использования, раскрывая как статическую структуру, так и динамическое поведение.

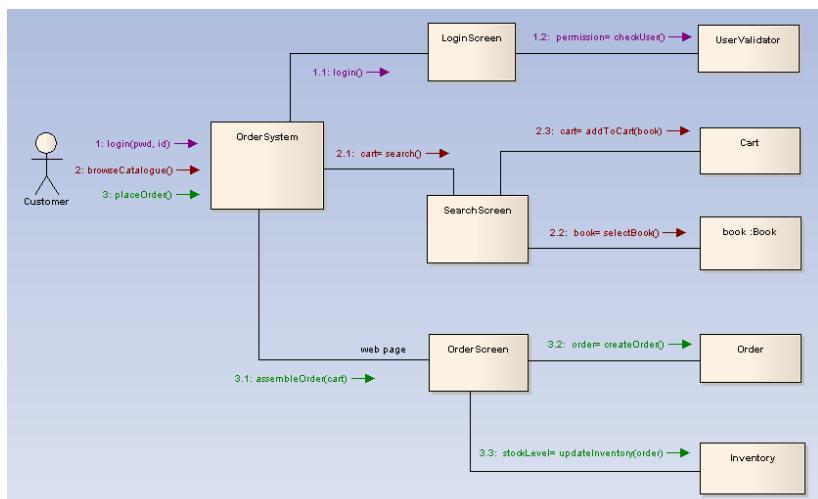
Диаграмма кооперации использует свободную форму расположения объектов и связей, как в диаграмме объектов. Для того чтобы поддерживать упорядочивание сообщений в диаграмме с такой свободной формой, сообщения обозначаются хронологическим номером, ставящимся рядом со связью, по которой пришло сообщение. Чтение диаграммы кооперации подразумевает начало с сообщения 1.0.

Схема нумерации следующая:

- 1
- 1.1
- 1.1.1
- 1.1.2
- 1.2, и т.д.

Новый числовой сегмент начинается для нового слоя обработки, что было бы эквивалентно обращению к методу.

Пример ниже показывает диаграмму кооперации между сущностями взаимодействующих объектов.



## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

### Communication Diagram Elements

 Actor	Действующее лицо – пользователь системы.
 Object	Объект. Отдельный экземпляр класса.
 Boundary	Граница – шаблонный объект, моделирующий некоторые границы/ограничения системы. Типичный пример: пользовательский интерфейс.
 Control	Контрол – шаблонный объект, моделирующий контролирующую (управляющую) сущность или менеджер. Организует выполнение других элементов.
 Communication	Сущность – шаблонный объект, моделирующий хранилище или персистентный механизм, который получает/содержит/захватывает информацию или знания в системе.
 Package	Пакет – пространство имен, которое так же может входить в другие пространства имен.

### Communication Diagram Connectors

—	Ассоциация между объектами
—⊕	Вложение объектов
— — — →	Объект реализует свойства другого объекта

С UML (Unified Modeling Language) 1.4 collaboration diagram (диаграмма кооперации) была переименована в communication diagram и упрощена.

## **28. UML. Диаграммы реализации (implementation diagrams). Назначение. Пример использования.**

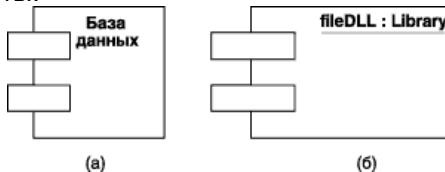
Для создания конкретной физической системы необходимо реализовать все элементы логического представления в конкретные материальные сущности. Для описания таких реальных сущностей предназначен другой аспект модельного представления, а именно – физическое представление модели. В контексте языка UML это означает совокупность связанных физических сущностей, включая программное и аппаратное обеспечение, а также персонал, которые организованы для выполнения специальных задач.

### Диаграмма компонентов:

Этот тип диаграмм предназначен для распределения классов и объектов по компонентам при физическом проектировании системы.

Диаграмма компонентов описывает особенности физического представления системы. Диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Во многих средах разработки модуль или компонент соответствует файлу. Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ. Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

Компонент — физически существующая часть системы, которая обеспечивает реализацию классов и отношений, а также функционального поведения моделируемой программной системы. Он предназначен для представления физической организации ассоциированных с ним элементов модели. Компонентом может быть исполняемый код отдельного модуля, командные файлы или файлы, содержащие интерпретируемые скрипты.



## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

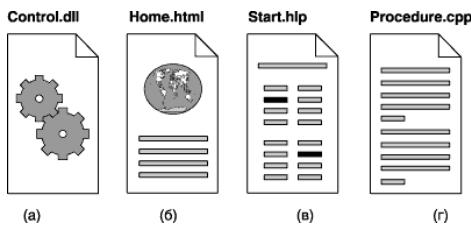
Компонент служит для общего обозначения элементов физического представления модели и может реализовывать некоторый набор интерфейсов. Имя компонента подчиняется общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и знаков препинания. Если компонент представляется на уровне типа, то записывается только имя типа с заглавной буквы в форме: <Имя типа>. Если же компонент представляется на уровне экземпляра, то его имя записывается в форме: <имя компонента ‘:’ Имя типа>.

В качестве собственных имен компонентов принято использовать имена исполняемых файлов (динамических библиотек, Web-страниц, текстовых файлов).

Для более наглядного изображения компонентов были предложены и стали общепринятыми следующие графические стереотипы:

1. стереотипы для компонентов развертывания, которые обеспечивают непосредственное выполнение системой своих функций: динамически подключаемые библиотеки, Web-страницы.
2. стереотипы для компонентов в форме рабочих продуктов. Как правило – это файлы с исходными текстами программ.

Эти элементы иногда называют артефактами, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих компонентов.



Другой способ спецификации различных видов компонентов — указание текстового стереотипа компонента перед его именем. В языке UML для компонентов определены

следующие стереотипы:

- <<file>> – произвольный физический файл.
- <<executable>> – исполнимый файл.
- <<document>> – документ произвольного содержания, не являющийся исполнимым файлом или файлом с исходным текстом программы.
- <<library>> – динамическая или статическая библиотека.
- <<source>> файл с исходным текстом программы.

- <<table>> – таблица базы данных.

Диаграмма развертывания:

Диаграмма развертывания - диаграмма, на которой представлены узлы выполнения программных компонентов реального времени, а также процессов и объектов.

Диаграмма развертывания применяется для представления общей конфигурации и топологии распределенной программной системы и содержит изображение размещения компонентов по отдельным узлам системы. Она показывает наличие физических соединений - маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.

Диаграмма развертывания предназначена для визуализации элементов и компонентов программы, существующих только на этапе ее исполнения (run-time). При этом представляются только те компоненты программы, которые являются исполнимыми файлами или динамическими библиотеками. Компоненты, не используемые на этапе исполнения, на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов. На диаграмме развертывания они не указываются.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними. В отличие от диаграмм логического представления, диаграмма развертывания является единственной для системы в целом, поскольку должна отражать все особенности ее реализации. Диаграмма развертывания разрабатывается совместно системными аналитиками, сетевыми инженерами и системотехниками.

Узел

Узел представляет собой физически существующий элемент системы, который может обладать вычислительным ресурсом или являться техническим устройством.

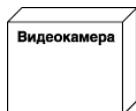
В качестве вычислительного ресурса узла может рассматриваться один или несколько процессоров, а также объем электронной или магнитооптической памяти, датчики, принтеры, модемы, цифровые камеры, сканеры и манипуляторы.

Графически узел на диаграмме развертывания изображается в форме трехмерного куба. Узел имеет имя, которое указывается внутри этого графического символа. Сами

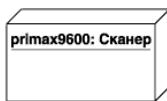
## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

узлы могут представляться как на уровне типа, так и на уровне экземпляра.

Имя типа узла указывает на разновидность узлов,



(а)

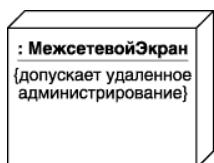


(б)

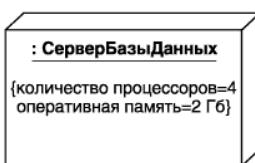
присутствующих в модели системы. Так, на представленном рисунке узел с именем Видеокамера относится к общему типу и никак не конкретизируется. Второй узел является узлом-экземпляром конкретной модели сканера.

Изображения узлов могут расширяться, чтобы включить дополнительную информацию о спецификации узла.

В качестве дополнения к имени узла могут использоваться различные текстовые стереотипы, которые явно специфицируют назначение этого узла.



(а)



(б)

Для этой цели были предложены следующие текстовые стереотипы: "processor"

(процессор), "sensor" (датчик), "modem" (модем), "net" (сеть), "printer" (принтер) и другие, смысл которых понятен из контекста.

Наиболее известны два специальных графических стереотипа для обозначения разновидностей узлов:

- ресурсоемкий узел (processor) - узел с процессором и памятью, необходимыми для выполнения исполняемых компонентов. Он изображается в форме куба с боковыми гранями, окрашенными в серый цвет (а).
- устройство (device) - узел без процессора и памяти (б).



(а)



(б)



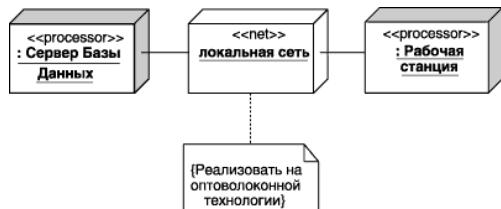
(в)

## Соединения и зависимости на диаграмме развертывания

В качестве отношений выступают физические соединения между узлами, а также зависимости между узлами и компонентами, которые допускается изображать на диаграммах развертывания.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Соединения являются разновидностью ассоциации и изображаются отрезками линий без стрелок. Наличие такой линии указывает на необходимость организации физического канала для обмена информацией между соответствующими узлами. Характер соединения может быть дополнительно специфицирован примечанием, стереотипом, помеченным значением или ограничением. Так, на представленном ниже фрагменте диаграммы развертывания явно определены рекомендации по технологии физической реализации соединений в форме примечания.



При большом количестве развернутых на узле компонентов информацию можно представить в форме отношения зависимости. Разработка информационных систем, обеспечивающих доступ в режиме реального времени, предполагает не только создание программного кода, но и использование дополнительных аппаратных средств. Вариант физического представления модели мобильного доступа к корпоративной базе данных показан на следующей диаграмме развертывания.



### **29. UML. Диаграмма компонентов (component diagram).**

**Назначение. Пример использования.**

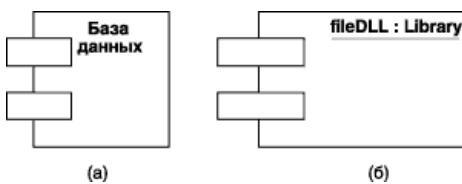
Этот тип диаграмм предназначен для распределения классов и объектов по компонентам при физическом проектировании системы.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Диаграмма компонентов описывает особенности физического представления системы. Диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Во многих средах разработки модуль или компонент соответствует файлу. Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ. Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

Компонент — физически существующая часть системы, которая обеспечивает реализацию классов и отношений, а также функционального поведения моделируемой программной системы. Он предназначен для представления физической организации ассоциированных с ним элементов модели. Компонентом может быть исполняемый код отдельного модуля, командные файлы или файлы, содержащие интерпретируемые скрипты.

Компонент служит для общего обозначения элементов физического представления модели и может реализовывать некоторый набор интерфейсов. Имя компонента подчиняется



общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и знаков препинания. Если компонент представляется

на уровне типа, то записывается только имя типа с заглавной буквы в форме: <Имя типа>. Если же компонент представляется на уровне экземпляра, то его имя записывается в форме: <имя компонента > <Имя типа>.

В качестве собственных имен компонентов принято использовать имена исполняемых файлов (динамических библиотек, Web-страниц, текстовых файлов).

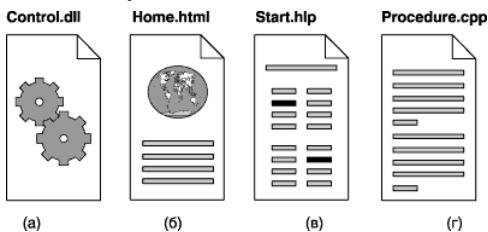
Для более наглядного изображения компонентов были предложены и стали общепринятыми следующие графические стереотипы:

3. стереотипы для компонентов развертывания, которые обеспечивают непосредственное выполнение системой своих функций: динамически подключаемые библиотеки, Web-страницы.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

4. стереотипы для компонентов в форме рабочих продуктов. Как правило – это файлы с исходными текстами программ.

Эти элементы иногда называют артефактами, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих компонентов.



Другой способ спецификации различных видов компонентов — указание текстового стереотипа компонента перед его именем. В языке UML для

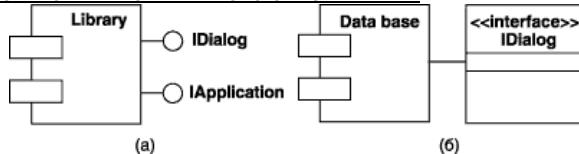
компонентов определены следующие стереотипы:

- <<file>> – произвольный физический файл.
- <<executable>> – исполнимый файл.
- <<document>> – документ произвольного содержания, не являющийся исполнимым файлом или файлом с исходным текстом программы.
- <<library>> – динамическая или статическая библиотека.
- <<source>> файл с исходным текстом программы.
- <<table>> – таблица базы данных.

### Интерфейсы

Интерфейсы обеспечивают не только совместимость различных версий, но и возможность вносить существенные изменения в одни части программы, не изменяя другие. Характер применения интерфейсов отдельными компонентами может отличаться.

Различают два способа связи интерфейса и компонента. Если компонент реализует некоторый интерфейс, то такой интерфейс называют экспортруемым или поддерживаемым, поскольку этот компонент предоставляет его в качестве сервиса другим компонентам. Если же компонент использует некоторый интерфейс, который реализуется другим компонентом, то такой интерфейс для первого компонента называется импортируемым. Особенность импортируемого интерфейса состоит в том, что на диаграмме компонентов это отношение изображается с помощью зависимости.



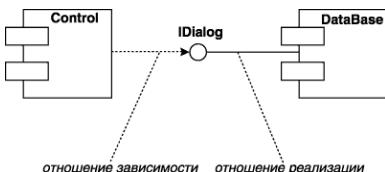
### Зависимости между компонентами

Отношение зависимости служит для представления факта наличия специальной формы связи между двумя элементами модели, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели.

Применительно к диаграмме компонентов зависимости могут связывать компоненты и импортируемые этим компонентом интерфейсы, а также различные виды компонентов между собой.

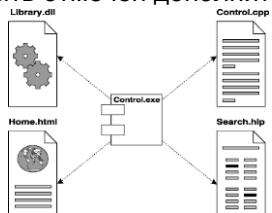
В этом случае рисуют стрелку от компонента-клиента к импортируемому интерфейсу. Наличие такой стрелки означает, что компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения. При этом на этой же диаграмме может присутствовать и другой компонент, который реализует этот интерфейс. Отношение реализации интерфейса обозначается на диаграмме компонентов обычной линией без стрелки.

Другим случаем отношения зависимости на диаграмме компонентов является отношение программного вызова и



компиляции между различными видами компонентов. Для рассмотренного фрагмента диаграммы компонентов наличие подобной зависимости означает, что исполняемый компонент

Control .exe использует функциональность компонента Library .dll, вызывает страницу гипертекста Home .html и файл помощи Search .hlp, а исходный текст этого исполняемого компонента хранится в файле Control .cpp. При этом характер отдельных видов зависимостей может быть отмечен дополнительно с помощью текстовых стереотипов.



На диаграмме компонентов могут быть также представлены отношения зависимости между компонентами и реализованными в них классами. Эта информация имеет значение для обеспечения согласования логического

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

и физического представлений модели системы. Ниже приводится фрагмент зависимости подобного рода, когда исполимый компонент Control .exe зависит от соответствующих классов.

Если требуется подчеркнуть, что некоторый компонент реализует отдельные классы, то для обозначения компонента используется расширенный символ прямоугольника. При этом прямоугольник компонента делится на две секции горизонтальной линией. Верхняя секция служит для записи имени компонента и, возможно, дополнительной информации, а нижняя секция – для указания реализуемых данным компонентом классов.

### **30. UML.Диаграмма размещения(развертывания) (deployment diagram) . Назначение. Пример использования.**

Диаграмма развертывания - диаграмма, на которой представлены узлы выполнения программных компонентов реального времени, а также процессов и объектов.

Диаграмма развертывания применяется для представления общей конфигурации и топологии распределенной программной системы и содержит изображение размещения компонентов по отдельным узлам системы. Она показывает наличие физических соединений - маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.

Диаграмма развертывания предназначена для визуализации элементов и компонентов программы, существующих только на этапе ее исполнения (run-time). При этом представляются только те компоненты программы, которые являются исполнимыми файлами или динамическими библиотеками. Компоненты, не используемые на этапе исполнения, на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов. На диаграмме развертывания они не указываются.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними. В отличие от диаграмм логического представления, диаграмма развертывания является единственной для системы в целом, поскольку должна отражать все особенности ее реализации. Диаграмма развертывания разрабатывается

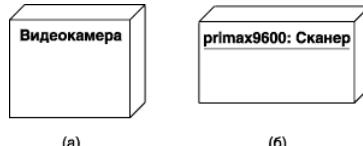
ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ,  
совместно системными аналитиками, сетевыми инженерами и  
системотехниками.

### Узел

Узел представляет собой физически существующий элемент системы, который может обладать вычислительным ресурсом или являться техническим устройством.

В качестве вычислительного ресурса узла может рассматриваться один или несколько процессоров, а также объем электронной или магнитооптической памяти, датчики, принтеры, модемы, цифровые камеры, сканеры и манипуляторы.

Графически узел на диаграмме развертывания изображается в форме трехмерного куба. Узел имеет имя, которое указывается внутри этого графического символа. Сами узлы могут представляться как на уровне типа, так и на уровне экземпляра.



(a)

(б)

Имя типа узла указывает на разновидность узлов, присутствующих в модели системы. Так, на представленном рисунке узел с именем Видеокамера относится к общему типу и никак не конкретизируется. Второй узел является узлом-экземпляром конкретной модели сканера.

Изображения узлов могут расширяться, чтобы включить дополнительную информацию о спецификации узла.

В качестве дополнения к имени узла могут использоваться различные текстовые стереотипы, которые явно специфицируют



(а)

(б)

назначение этого узла. Для этой цели были предложены следующие текстовые стереотипы: "processor" (процессор), "sensor" (датчик), "modem" (модем), "net"

(сеть), "printer" (принтер) и другие, смысл которых понятен из контекста.

Наиболее известны два специальных графических стереотипа для обозначения разновидностей узлов:

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

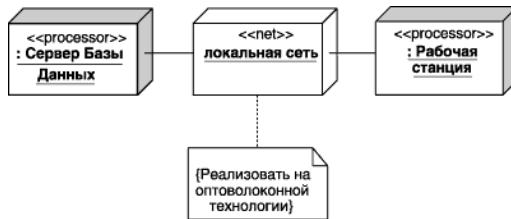
- ресурсоемкий узел (processor) - узел с процессором и памятью, необходимыми для выполнения исполняемых компонентов. Он изображается в форме куба с боковыми гранями, окрашенными в серый цвет (а).
- устройство (device) - узел без процессора и памяти (б).



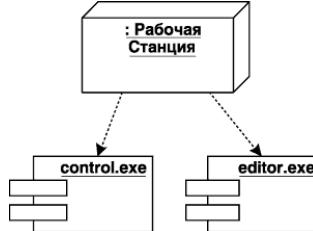
### Соединения и зависимости на диаграмме развертывания

В качестве отношений выступают физические соединения между узлами, а также зависимости между узлами и компонентами, которые допускается изображать на диаграммах развертывания.

Соединения являются разновидностью ассоциации и изображаются отрезками линий без стрелок. Наличие такой линии указывает на необходимость организации физического канала для обмена информацией между соответствующими узлами. Характер соединения может быть дополнительно специфицирован примечанием, стереотипом, помеченным значением или ограничением. Так, на представленном ниже фрагменте диаграммы развертывания явно определены рекомендации по технологии физической реализации соединений в форме примечания.



При большом количестве развернутых на узле компонентов информацию можно представить в форме отношения зависимости. Разработка информационных систем, обеспечивающих доступ в режиме реального времени, предполагает не только создание программного кода, но и использование дополнительных аппаратных средств. Вариант физического представления модели мобильного доступа к корпоративной базе данных показан на следующей диаграмме развертывания.



Данная диаграмма содержит общую информацию о развертывании рассматриваемой системы и может быть детализирована при разработке программных компонентов управления. Как видно из рисунка, в этой диаграмме развертывания использованы дополнительные стереотипы "приемопередатчик" и "мобильный телефон", которые отсутствуют в описании языка UML, а также специальные графические изображения (стереотипы) для отдельных аппаратных устройств.



### 31. Управление проектами. Сущность управления проектами. Управление проектами. Этапы структурного руководства проектом. Индикатор вероятности успеха (psi).

Проект - это временное предприятие, предназначенное для создания уникальных продуктов или услуг.

Управление проектами - дисциплина применения методов, практик, опыта, и средств к работам проекта для достижений

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

целей проекта, при условии удовлетворения ограничений, определяющих рамки проекта.



Процессы управления проектом осуществляются на всех стадиях жизненного цикла проекта и могут быть классифицированы по двум следующим основаниям:

- по области применения (области знаний)
- по целевому результату (фазы управления)

К областям знаний в проекте относится управление содержанием и границами проекта, управление проектом по временным и стоимостным параметрам, управление качеством, отклонениями и др.

Под фазой процесса управления понимается совокупность мероприятий (процессов), обеспечивающих достижение одного из следующих результатов:

- санкционирование начала проекта или очередной стадии его жизненного цикла - инициализация;
- определение наилучшего способа действий для достижения целей стадии жизненного цикла проекта с учетом складывающейся обстановки - планирование;
- реализация плана стадии жизненного цикла проекта (от выдачи задания до получения результата) - выполнение;
- выявление фактов отклонения фактического выполнения стадии жизненного цикла проекта от запланированного и принятие корректирующих действий - контроль;
- завершение и закрытие проекта или стадии жизненного цикла проекта - завершение.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.



### Планирование ресурсов:

Предложений по проектам всегда больше, чем ресурсов. Необходима система приоритетов, которая поможет выбрать проекты, наилучшим образом содействующие целям организации, в рамках имеющихся ресурсов. Если графики всех проектов и соответствующие им ресурсы выполнены с помощью компьютера, то можно быстро определить реальную ситуацию и влияние нового проекта на проекты, находящиеся в работе. Имея такую информацию, команда по приоритетам добавит новый проект только в том случае, если имеются ресурсы, и они формально предназначены для этого конкретного проекта. В этой теме рассматриваются методы календарного планирования ресурсов, с тем, чтобы команда могла составить мнение о реальном наличии ресурсов и времени продолжительности проекта. Если во время осуществления проекта происходят какие-то изменения, то компьютерный график легко корректировать, и результаты легко оценить.

### Управление риском:

Планирование проектного риска формально связано с выявлением, анализом и оценкой потенциальных проблемных участков до начала работы над проектом.

Основными составляющими процесса управления риском являются:

1. **Выявление источников риска:** составление списка всех факторов, которые могут затормозить работу над проектом или вовсе помешать его реализации, а также результатов их воздействия

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

2. **Анализ и оценка риска:** дать количественную оценку степени серьезности выявленного события, вероятности его наступления и чувствительности проекта к нему
3. **Определение реакции на риск**
4. **Планирование расходов в чрезвычайных обстоятельствах:** превентивные действия, призванные снизить или смягчить негативное влияние риска
5. **Создание резервов на случай чрезвычайных обстоятельств:** покрытие ошибок в расчетах, упущений или неопределенности, которые могут вскрыться по мере выполнения проекта



(Дополнительно смотри 32 пункт)

## **32. Управление проектами. Этапы структурного руководства проектом. Индикатор вероятности успеха (psi).**

(смотри 31 пункт)

Десять этапов руководства структурного руководства проектом:

1. Цель(Наглядное представление цели);
2. Список задач(Разработка списка задач, которые необходимо выполнить);
3. Один руководитель
4. Распределение людей по задачам;
5. Управление ожидаемыми результатами, расчет резервов для ошибок, выработка запасных позиций;
6. Стиль руководства;
7. Знать то, что происходит;
8. Сообщать людям, что происходит;
9. Повтор этапов с 1 до 8;
10. Приз;

Индикатор вероятности успеха (PSI):

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

1. PSI - число, лежащее в диапазоне 0-100, которое присваивается проектам с помощью схемы подсчета, приведенной ниже. PSI может быть рассчитан в любой точке жизни проекта и определяет вероятность того, что проект завершится успешно.
2. PSI может быть рассчитан различными способами. Отдельный человек (например, организатор проекта или внешний консультант) может произвести оценку проекта и вычислить PSI, применяя приведенные ниже правила. Альтернативно можно опросить ряд лиц, участвующих в проекте (руководитель проекта, члены группы, управлеченческое звено, заказчики) для получения их мнений (скажем, используя анкетный опрос), а затем усреднить полученные результаты.
3. Кто вычисляет PSI, кто дает исходные данные и как эти исходные данные собираются - анкетным опросом, в беседах, подсчетом поднятых рук(!) - это все факторы, которые могут рассматриваться, если вы собираетесь использовать PSI. Здесь самый быстрый и самый простой путь, то есть один человек (я!) проводит анализ данного проекта.
4. PSI рассчитывается путем присвоения баллов каждому из Десяти Этапов, привязанных к определенному проекту. Например:
  1. Цель(Наглядное представление цели); (20 баллов)
  2. Список задач(Разработка списка задач, которые необходимо выполнить); (20)
  3. Один руководитель (10)
  4. Распределение людей по задачам; (10)
  5. Управление ожидаемыми результатами, расчет резервов для ошибок, выработка запасных позиций; (10)
  6. Стиль руководства; (10)
  7. Знать то, что происходит; (10)
  8. Сообщать людям, что происходит; (10)
  9. Повтор этапов с 1 до 8; (0)
  10. Приз; (0)

## **33. Управление проектом. Этапы. Задачи. Треугольник проекта.**

*Проект* - это *временное* предприятие, предназначенное для создания *的独特性* продуктов или услуг.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Управление проектами (англ. project management) — область деятельности, в ходе которой определяются и достигаются четкие цели при балансируении между объемом работ, ресурсами (такими как время, деньги, труд, материалы, энергия, пространство и др.), временем, качеством и рисками в рамках некоторых проектов, направленных на достижение определенного результата при указанных ограничениях. Ключевым фактором успеха проектного управления является наличие четкого заранее определенного плана, минимизации рисков и отклонений от него (в отличие от процессного, функционального управления, управления уровнем услуг).

### **Процедуры управления проектом по традиционной методологии**

- Определение среды проекта.
- Формулирование проекта.

По существу подразумевает функцию выбора проекта. Проекты инициируются в силу возникновения потребностей, которые нужно удовлетворить. Однако в условиях дефицита ресурсов невозможно удовлетворить все потребности без исключения. Приходится делать выбор. Одни проекты выбираются, другие отвергаются. Решения принимаются исходя из наличия ресурсов, и, в первую очередь, финансовых возможностей, сравнительной важности удовлетворения одних потребностей и игнорирования других, сравнительной эффективности проектов. Решения по отбору проектов к реализации тем важнее, чем масштабнее предполагается проект, поскольку крупные проекты определяют направление деятельности на будущее (иногда на годы) и связывают имеющиеся финансовые и трудовые ресурсы. Для сравнительного анализа проектов на данном этапе применяются методы проектного анализа, включающие в себя финансовый, экономический, коммерческий, организационный, экологический, анализ рисков и другие виды анализа проекта.

- Планирование проекта.

Планирование в том или ином виде производится в течение всего срока реализации проекта. В самом начале жизненного цикла проекта обычно разрабатывается неофициальный предварительный план, что потребуется выполнить в случае реализации проекта. Решение о выборе проекта в значительной степени основывается на оценках предварительного плана. Формальное и детальное планирование проекта начинается после принятия решения о его реализации. Определяются ключевые точки (вехи) проекта, формулируются задачи (работы) и их взаимная зависимость. Именно на этом этапе используются

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

системы для управления проектами, предоставляющие руководителю проекта набор средств для разработки формального плана: средства построения иерархической структуры работ, сетевые графики и диаграммы Ганнта, средства назначения и гистограммы загрузки ресурсов.

Как правило, план проекта не остается неизменным, и по мере осуществления проекта подвергается постоянной корректировке с учетом текущей ситуации.

- Техническое выполнение проекта (за исключением планирования и контроля).

После утверждения формального плана на менеджера ложиться задача по его реализации. По мере осуществления проекта руководители обязаны постоянно контролировать ход работ. Контроль заключается в сборе фактических данных о ходе работ и сравнении их с плановыми. К сожалению, в управлении проектами можно быть абсолютно уверенным в том, что отклонения между плановыми и фактическими показателями случаются всегда. Поэтому задачей менеджера является анализ возможного влияния отклонений в выполненных объемах работ на ход реализации проекта в целом и в выработке соответствующих управленческих решений. Например, если отставание от графика выходит за приемлемый уровень отклонения, может быть принято решение об ускорении выполнения определенных критических задач за счет выделения на них большего объема ресурсов.

- Контроль над выполнением проекта.
- Завершение

Рано или поздно, но проекты заканчиваются. Проект заканчивается, когда достигнуты поставленные перед ним цели. Иногда окончание проекта бывает внезапным и преждевременным, как в тех случаях, когда принимается решение прекратить проект до его завершения по графику. Как бы то ни было, но когда проект заканчивается, его руководитель должен выполнить ряд мероприятий, завершающих проект. Конкретный характер этих обязанностей зависит от характера самого проекта. Если в проекте использовалось оборудование, надо произвести его инвентаризацию и, возможно, передать его для нового применения. В случае подрядных проектов надо определить, удовлетворяют ли результаты условиям подряда или контракта. Может быть, необходимо составить окончательные отчеты, а промежуточные отчеты по проекту организовать в виде архива.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

### Процедуры управления проектом по методологии PMI

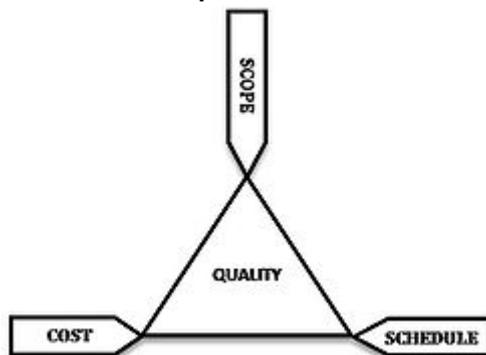
- Определение требований к проекту
- Постановка чётких и достижимых целей
- Балансирование конкурирующих требований по качеству, возможностям, времени и стоимости
- Адаптация спецификаций, планов и подходов для нужд и проблем различных заинтересованных лиц (стейкхолдеров)

### Процедуры управления проектом по методологии PRINCE2

- Начало проекта (SU).
- Запуск проекта (IP).
- Планирование проекта (PL).
- Управление проектом (DP).
- Контроль стадий (CS).
- Контроль границ стадий (SB).
- Управление производством продукта (MP).
- Завершение проекта (CP).

### Классическая форма Тройственной Ограниченностии

Тройственная ограниченность описывает баланс между содержанием проекта, стоимостью, временем и качеством. Качество было добавлено позже, поэтому изначально именовалась как тройственная ограниченность.



Как того требует любое начинание, проект должен протекать и достигать финала с учетом определенных ограничений. Классически эти ограничения определены как содержание проекта, время и стоимость. Они также относятся к Треугольнику Управления проектами, где каждая его сторона представляет ограничение. Изменение одной стороны треугольника влияет на другие стороны. Дальнейшее уточнение ограничений выделило из содержания качество и действие, превратив качество в четвертое ограничение.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Ограниченностю времени определяется количеством доступного времени для завершения проекта. Ограниченностю стоимости определяется бюджетом, выделенным для осуществления проекта. Ограниченностю содержания определяется набором действий, необходимых для достижения конечного результата проекта. Эти три ограниченности часто соперничают между собой. Изменение содержания проекта обычно приводит к изменению сроков (времени) и стоимости. Сжатые сроки (время) могут вызвать увеличение стоимости и уменьшение содержания. Небольшой бюджет (стоимость) может вызвать увеличение сроков (времени) и уменьшение содержания.

Иной подход к управлению проектами рассматривает следующие три ограниченности: финансы, время и человеческие ресурсы. При необходимости сократить сроки (время) можно увеличить количество занятых людей для решения проблемы, что непременно приведет к увеличению бюджета (стоимость). За счет того, что эта задача будет решаться быстрее, можно избежать роста бюджета, уменьшая затраты на равную величину в любом другом сегменте проекта.

### **Подходы**

- Предположение о неограниченности ресурсов, критичен только срок выполнения. Метод PERT, Метод критического пути
- Предположение о критичности качества (полноте удовлетворения потребностей, как известных, так и неизвестных заранее, часто создаваемых выходом нового продукта). Метод Гибкая методология разработки
- Предположение о неизменности требований и низких рисках. Классические методы PMBOK, во многом опирающийся на модель водопада
- Предположение о высоких рисках проекта. Метод Инновационные проекты (стартапы)
- Варианты нейтральных (сбалансированных) подходов:
- Акцент на взаимодействие исполнителей. Метод PRINCE2
- Акцент на взаимодействие процессов. Метод Process-based management

### **34. Принципы тестирования. Философия тестирования.**

На этот этап (тестирование) приходится около 50% общей стоимости разработки программного обеспечения.

**Тестирование** - это процесс выполнения программы с целью обнаружения в ней ошибок.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Классическое определение гласит, что **Тестирование** – это процесс, направленный на выявление характеристик системы и демонстрации различий между ее требуемым и фактическим состоянием. Другое определение исходит из самой сущности процесса тестирования. Оно гласит: **Тестирование** - это измерение качества программного продукта.

**Цель тестирования** - распознать дефекты в объекте тестирования и увеличить вероятность того, что он при любых обстоятельствах будет работать в соответствии с установленными требованиями.

Разберемся, а что представляет из себя дефект? Дефекты бывают внешние и внутренние. Отказ (наблюдаемый дефект, bug) – это внешнее проявление внутреннего изъяна.

У Майерса сформулированы основные принципы организации тестирования:

- 1) необходимой частью каждого теста должно являться описание ожидаемых результатов работы программы, чтобы можно было быстро выяснить наличие или отсутствие ошибки в ней;
- 2) следует по возможности избегать тестирования программы ее автором, т.к. кроме уже указанной объективной сложности тестирования для программистов здесь присутствует и тот фактор, что обнаружение недостатков в своей деятельности противоречит человеческой психологии;
- 3) по тем же соображениям организация - разработчик программного обеспечения не должна “единолично” его тестировать;
- 4) должны являться правилом доскональное изучение результатов каждого теста, чтобы не пропустить малозаметную на поверхностный взгляд ошибку в программе;
- 5) необходимо тщательно подбирать тест не только для правильных (предусмотренных) входных данных, но и для неправильных (непредусмотренных);
- 6) при анализе результатов каждого теста необходимо проверять, не делает ли программа того, что она не должна делать;
- 7) следует сохранять использованные тесты (для повышения эффективности повторного тестирования программы после ее модификации или установки у заказчика);
- 8) тестирования не должно планироваться исходя из предположения, что в программе не будут обнаружены ошибки (в частности, следует выделять для тестирования достаточные временные и материальные ресурсы);
- 9) следует учитывать так называемый “принцип скопления ошибок”: вероятность наличия не обнаруженных ошибок в

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

некоторой части программы прямо пропорциональна числу ошибок, уже обнаруженных в этой части;

**10)** следует всегда помнить, что тестирование - творческий процесс, а не относиться к нему как к рутинному занятию.

Существует два основных вида тестирования : функциональное и структурное.

1) При **функциональном тестировании** программа рассматривается как “черный ящик”. Происходит проверка соответствия поведения программы ее внешней спецификации. Очевидно, что критерием полноты тестирования в этом случае являлся бы перебор всех возможных значений входных данных, что невыполнимо.

2) При **структурном тестировании** программа рассматривается как “белый ящик”. Происходит проверка логики программы. Полным тестированием в этом случае будет такое, которое приведет к перебору всех возможных путей на графе передач управления программы (ее управляющем графе).

Но даже если предположить, что удалось достичь полного структурного тестирования некоторой программы, в ней, тем не менее, могут содержаться ошибки, т.к. 1) программа может не соответствовать своей внешней спецификации, что в частности, может привести к тому, что в ее управляющем графе окажутся пропущенными некоторые необходимые пути; 2) не будут обнаружены ошибки, появление которых зависит от обрабатываемых данных (т.е. на одних исходных данных программа работает правильно, а на других - с ошибкой).

В тестирование многомодульных программных комплексов можно выделить четыре этапа:

- 1) тестирование отдельных модулей;
- 2) совместное тестирование модулей;
- 3) тестирование функций программного комплекса;
- 4) тестирование всего комплекса в целом.

На первых двух этапах используются прежде всего методы структурного тестирования; последующие этапы тестирования ориентированы на обнаружение ошибок различного типа, которые не обязательно связаны с логикой программы.

Рассмотрим типы тестирования по качествам продукта

- Функциональное тестирование – соответствие внешним спецификациям
- Тестирование usability – в Gamedev сюда относиться играбельность и захватываемость.
- Тестирование производительности – быстрота отклика системы и ее производительность (например, количество FPS)

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Тестирование установки – вид тестирования о котором зачастую забывают или вспоминают в последнюю очередь. Заключается в тестирование правильности и удобства установки на различные целевые платформы.
- Конфигурационное тестирование – совместимость с разл. типами оборуд-я и ПО.

По методу черного ящика в основном тестируются функциональные условия и Тестирование удобства использования. Для тестирования производительности и конфигурационного тестирования зачастую используют белый ящик.

Какие требования предъявляются к рядовым тестировщикам?

- Умение устно и письменно излагать проблему
- Способность предугадать, где находятся ошибки
- Способность одновременного выполнения нескольких задач
- Навыки работы по расписанию
- Умение работать в условиях давления
- Наблюдательность, внимательность к деталям
- Способность представить себя на месте другого
- Умение читать и писать спецификации

Полное тестирование практически невозможно. Если так то возникает логический вопрос, когда прекращать тестирования, какой уровень качества считать достаточным. Этот вопрос считается основным вопросом тестирования. Существует множества вариантов ответов на этот вопрос, один из них перед вами: Тестирование следует продолжать до тех пор, пока затраты на обнаружение и исправление дефектов НИЖЕ, чем будущие потери от сбоев продукта при его эксплуатации (Тим Комен, Мартин Пол)

Другими словами важно определить, что достигнут достаточно хороший уровень качества.

Философия тестирования: стремиться к более высокому качеству продукта, находить все возможные дефекты.  $\{ \text{Качество} \} = \{ \text{Удовлетворенность заказчика} \} = \{ \text{Ценност} \} / \{ \text{Стоимость} \}$  (формула из неофиц. источника, придумал некто на форуме)

## **35. Уровни тестирования. Этапы тестирования.**

**Тестирование программного обеспечения** — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. **В более широком смысле, тестирование** - это одна из техник контроля качества, включающая в себя активности по планированию работ (**Test**

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. **Management**), проектированию тестов (**Test Design**), выполнению тестирования (**Test Execution**) и анализу полученных результатов (**Test Analysis**).

## **Уровни тестирования**

**Модульное тестирование** (юнит-тестирование) — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Обычно компонентное (модульное) тестирование проводится вызывая код, который необходимо проверить и при поддержке сред разработки, таких как фреймворки (frameworks - каркасы) для модульного тестирования или инструменты для отладки. Все найденные дефекты, как правило исправляются в коде без формального их описания в системе менеджмента багов (Bug Tracking System).

Один из наиболее эффективных подходов к компонентному (модульному) тестированию — это **подготовка автоматизированных тестов** до начала основного кодирования (разработки) программного обеспечения. Это называется разработка от тестирования (**test-driven development**) или подход тестирования вначале (**test first approach**). При этом подходе создаются и интегрируются небольшие куски кода, напротив которых запускаются тесты, написанные до начала кодирования. Разработка ведется до тех пор пока все тесты не будут успешными.

### **Разница между компонентным и модульным тестированием**

По-существу эти уровни тестирования представляют одно и тоже, разница лишь в том, что в компонентном тестировании в качестве параметров функций используют реальные объекты и драйверы, а в модульном тестировании — конкретные значения.

**Интеграционное тестирование** — тестируются интерфейсы между компонентами, подсистемами. При наличии резерва времени на данной стадии тестирование ведётся итерационно, с постепенным подключением последующих подсистем.

### **Уровни интеграционного тестирования:**

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- **Компонентный интеграционный уровень** (*Component Integration testing*)

Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

- **Системный интеграционный уровень** (*System Integration Testing*)

Проверяется взаимодействие между разными системами после проведения системного тестирования.

## **Подходы к интеграционному тестированию:**

- **Снизу вверх** (*Bottom Up Integration*)

Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения

- **Сверху вниз** (*Top Down Integration*)

Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, зачем по мере готовности они заменяются реальными активными компонентами. Таким образом мы проводим тестирование сверху вниз.

- **Большой взрыв** ("Big Bang" *Integration*)

Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования.

**Системное тестирование** — тестируется интегрированная система на её соответствие требованиям. Основной задачей системного тестирования является **проверка как функциональных, так и не функциональных требований в системе в целом**. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения в системы в той или иной среде, **во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи**.

- **Альфа-тестирование** — имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приёмочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться ПО.
- **Бета-тестирование** — в некоторых случаях выполняется распространение версии с ограничениями (по функциональности или времени работы) для некоторой группы лиц, с тем чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Можно выделить два подхода к системному тестированию:

- **на базе требований** (*requirements based*)

Для каждого требования пишутся тестовые случаи (*test cases*), проверяющие выполнение данного требования.

- **на базе случаев использования** (*use case based*)

На основе представления о способах использования продукта создаются случаи использования системы (**Use Cases**). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся тест кейсы (*test cases*), которые должны быть протестированы.

**Приемочное тестирование или Приемо-сдаточное испытание (Acceptance Testing)** - формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:

- определения удовлетворяет ли система приемочным критериям;
- вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

Приемочное тестирование выполняется на основании набора типичных тестовых случаев и сценариев, разработанных на основании требований к данному приложению.

**Решение о проведении приемочного тестирования** принимается, когда:

- продукт достиг необходимого уровня качества;
- заказчик ознакомлен с **Планом Приемочных Работ (Product Acceptance Plan)** или иным документом, где

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

описан набор действий, связанных с проведением приемочного тестирования, дата проведения, ответственные и т.д.

**Фаза приемочного тестирования** длится до тех пор, пока заказчик не выносит решение об отправлении приложения на доработку или выдаче приложения.

### **Методы тестирования**

- Метод черного ящика – тестирование без знания реализации
- Метод белого ящика – тестирование на основе кода
- Тестирование моделей - объект тестирования - не сама система, а ее модель, спроектированная формальными средствами.
- Анализ программного кода (инспекции) – ручной анализ программного кода на корректность, называемый также просмотрами или инспекциями кода.

### **Этапы тестирования**

1. Анализ требований: тестирование должно начинаться, когда определяются требования к продукту. На этапе разработки, тестеры вместе с разработчиками определяют, какие аспекты дизайна необходимо проверить и с какими параметрами.
2. Планирование тестирования: создается план тестирования.
3. Разработка тестов: создаются тестовые сценарии, тесты данных, тестовые скрипты, которые будут использоваться при тестировании программного обеспечения.
4. Выполнение тестов: тестировщики выполняют тесты на основе плановых документов и тестовой документации и сообщают о любых ошибках команде разработчиков.
5. Тестовые отчеты: после проведенных тестов, тестеры подсчитывают метрики и создают финальный отчет об их тестах и сообщают, действительно ли протестированный продукт готов к выпуску.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

6. Анализ результатов тестов: делается командой разработчиков вместе с заказчиком, чтобы выяснить какие из найденных ошибок стоит исправлять в-первую очередь, какие отложить на потом.
7. Повторное тестирование: после того, как ошибки были исправлены, команда тестеров прогоняет свои тесты еще раз.
8. Регрессионное тестирование: при тестировании нового функционала тестеры сначала прогоняют все старые тесты, чтобы убедиться, что система работает правильно.

Окончание тестирования: тестирование заканчивается, если продукт удовлетворяет заявленным требованиям. Вся документация и тесты сохраняются, чтобы при необходимости использовать в других проектах.

### **Виды работ при тестировании**

- Планирование и управление
- Спецификация/проектирование
- Подготовка среды и инструментов тестирования
- Выполнение тестов
- Анализ и отчетность
- Рецензирование

### **Распределение ролей при тестировании**

- Руководитель тестирования
  - выбор стратегии тестирования
  - планирование и управление процессом
- Тест-аналитик
  - проектирование тестовых сценариев
  - подготовка среды тестирования
- Тестер
  - выполнение тестов
  - написание отчетов

### **36. Метрики проекта.**

**Метрики программных проектов** – это количественные показатели, отражающие их отдельные характеристики. Точнее, метрики представляют собой все, что можно измерить, при условии, конечно, что в этом есть смысл.

**Метрики программного продукта** включают:

- внешние метрики, обозначающие свойства продукта, видимые пользователю;
- внутренние метрики, обозначающие свойства, видимые только команде разработчиков.

Внешние метрики продукта - это метрики:

- надежности продукта, которые служат для определения числа дефектов;
- функциональности, с помощью которых устанавливаются наличие и правильность реализации функций в продукте;
- сопровождения, с помощью которых измеряются ресурсы продукта (скорость, память, среда); применимости продукта, которые способствуют определению степени доступности для изучения и использования;
- стоимости, которыми определяется стоимость созданного продукта.

Внутренние метрики продукта включают:

- метрики размера, необходимые для измерения продукта с помощью его внутренних характеристик;
- метрики сложности, необходимые для определения сложности продукта;
- метрики стиля, которые служат для определения подходов и технологий создания отдельных компонентов продукта и его документов.

### **Количественные метрики**

Количество строк кода, количество пустых строк, количество комментариев, процент комментариев (отношение числа строк, содержащих комментарии к общему количеству строк, выраженное в процентах), среднее число строк для функций (классов, файлов), среднее число строк, содержащих

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.  
исходный код для функций (классов, файлов), среднее число строк для модулей.

## Объектно-ориентированные метрики

С переходом от структурной к объектно-ориентированной (ОО) парадигме программирования возникла потребность и в создании соответствующих метрик. Поскольку основные элементы конструирования ОО программ – это классы, реализующие функциональность посредством методов, то именно классы и методы являются категориями, которыми преимущественно оперируют ОО метрики. Помимо таких очевидных метрик, как общее число классов, методов, атрибутов, средних показателей числа методов и атрибутов на класс и пр., применяются гораздо более комплексные метрики, с помощью которых можно судить не только об объеме исходного кода ОО проекта, но и о его сложности, качестве, соответствии основным принципам ОО парадигмы и т. д.

К самым распространенными ОО метриками относятся:

- *количество вызываемых удаленных методов (Number Of Remote Methods, NORM).* При его вычислении просматриваются все конструкторы и методы класса и подсчитывается число вызываемых удаленных методов (не определенных в классе и его родителях);
- *отклик на класс (Response For Class, RFC)* – количество методов, которые могут вызываться экземплярами класса, вычисляется как сумма локальных и удаленных методов;
- *взвешенная насыщенность класса 1 (Weighted Methods Per Class 1, WMPC1)* – отражает относительную меру сложности класса на основе цикломатической сложности каждого его метода. Класс с более сложными методами и большим их количеством считается более сложным (при вычислении метрики родительские классы не учитываются);
- *взвешенная насыщенность класса 2 (WMPC2)* – мера сложности класса, основанная на том, что класс с большим количеством методов и метод с большим количеством параметров являются более сложными (при вычислении метрики родительские классы не учитываются);
- *недостаток связности методов 1 (Lack Of Cohesion Of Methods 1, LOCOM1)* – отражает меру взаимозависимости методов (связность характеризует степень взаимодействия

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

между элементами отдельного модуля и его насыщенность). Чем выше значение данной метрики, тем более взаимозависимыми и, соответственно, сложными считаются методы;

- недостаток связности методов 2 (*LOCOM2*) – отражает меру взаимозависимости классов. Вычисляется как процентное отношение методов, не имеющих доступа к специфичным атрибутам класса, ко всем атрибутам данного класса. Класс, который может вызывать значительно больше методов, чем равные ему по уровню, является более сложным;
- недостаток связности методов 3 (*LOCOM3*) – измеряет степень различия методов в классе по атрибутам. Низкое значение свидетельствует о хорошей декомпозиции в классе, выражаемой в его простоте, понятности и готовности к повторному использованию. Высокое значение недостатка связности увеличивает сложность и повышает вероятность ошибок в процессе разработки.

## **Метрики Надежности**

Следующий тип метрик - метрики, близкие к количественным, но основанные на количестве ошибок и дефектов в программе. Нет смысла рассматривать особенности каждой из этих метрик, достаточно будет их просто перечислить: количество структурных изменений, произведенных с момента прошлой проверки, количество ошибок, выявленных в ходе просмотра кода, количество ошибок, выявленных при тестировании программы и количество необходимых структурных изменений, необходимых для корректной работы программы. Для больших проектов обычно рассматривают данные показатели в отношении тысячи строк кода, т.е. среднее количество дефектов на тысячу строк кода.

## **Метрики менеджмента**

- 1) цена – расходы на приобретение/разработку
- 2) время разработки – мера времени формирования заказа на программу до поставки
- 3) среда разработки – процент целевых компьютерных ресурсов используемых системой

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- 4) использование системных ресурсов – мера способности производителя разрабатывать программное обеспечение высокого качества

### **Метрики требований**

- 1) соответствие требованиям – дает возможность контролировать спецификации, изменение требований, а также степень их удовлетворения
- 2) стабильность требований

### **Метрики качества**

- 1) адаптируемость - мера гибкости системы
- 2) сложность интерфейсов и интеграции - метрика, измеряющая степень сложности интерфейса или дополнительного программирования требуемого для интеграции компоненты в систему
- 3) тестовое покрытие - степень полноты различных типов тестирования
- 4) надежность - вероятность работы системы без отказов
- 5) профили ошибок - кумулятивное число обнаруженных ошибок

степень удовлетворения заказчика - степень соответствия программного обеспечения ожиданиям и требованиям заказчика

## **37. Отладка. Основные методы отладки.**

**Отладка** — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла ошибка, приходится:

- узнавать текущие значения переменных;
- и выяснить, по какому пути выполнялась программа.

Существуют две взаимодополняющие технологии отладки.

- Использование отладчиков — программ, которые включают в себя пользовательский интерфейс для пошагового выполнения программы: оператор за оператором, функция за функцией, с остановками на некоторых строках исходного кода или при достижении определённого условия.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- Вывод текущего состояния программы с помощью расположенных в критических точках программы операторов вывода — на экран, принтер, громкоговоритель или в файл. Вывод отладочных сведений в файл называется журналированием.

Процесс отладки начинается при обнаружении ошибки и проводится в два этапа: 1) определяется природа и местонахождение ошибки в программе, 2) фиксируется и исправляется ошибка.

### **Методы "грубой силы"**

Наиболее общими при отладке программы являются довольно неэффективные методы «грубой силы». Причина популярности этих методов, возможно, заключается в том, что они не требуют значительного внимания и больших умственных затрат.

Методы грубой силы можно разделить по крайней мере на три категории: 1) отладка с использованием дампа памяти; 2) отладка в соответствии с общим предложением «расставить операторы печати по всей программе»; 3) отладка с использованием автоматических средств.

### **Метод индукции**

Считается, что большинство ошибок может быть обнаружено посредством тщательного анализа, даже без выхода на машину. Одним из таких методов является индукция, в процессе которой осуществляется анализ от частного к целому. При этом, просматривая детали (симптомы ошибок) и взаимосвязи между ними, часто можно прийти к ошибке.

Процесс индукции разбивается на следующие шаги:

#### **1. Определение данных, имеющих отношение к ошибке.**

Первым шагом должно быть перечисление всех действий, свидетельствующих о правильном выполнении программы и всех ее неправильных действий (т. е. симптомов, которые приводят к выводу о наличии ошибки).

**2. Организация данных.** Индукция подразумевает анализ от частного к общему, поэтому второй шаг заключается в структурировании данных, имеющих отношение к ошибке, с целью выявления неких закономерностей.

**3. Выдвижение гипотезы.** Следующими шагами являются изучение взаимосвязи между признаками и выдвижение одной или нескольких гипотез о причине ошибки с учетом закономерностей, выявленных в структуре симптомов ошибки.

**4. Доказательство гипотезы.** Пропуск этого шага и переход непосредственно к заключениям и попыткам решить проблему

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

является серьезной ошибкой, обычно влияющей на результат проведения отладки. Необходимо доказать приемлемость гипотезы, прежде чем взять ее за основу.

### **Метод дедукции**

Процесс дедукции позволяет на основании некоторых общих теорий или предпосылок, используя операции исключения и уточнения, прийти к определенному заключению (обнаружить место ошибки). Процесс дедукции состоит в следующем:

- 1. Перечисление возможных причин или гипотез.**
- 2. Использование данных для исключения возможных причин.** Путем тщательного анализа данных и поиска противоречий исключаются все возможные причины, кроме одной. Если же остается более чем одна причина, то первой выбирается наиболее вероятная из них - основная гипотеза.
- 3. Уточнение выбранной гипотезы.** Возможная причина может быть определена верно, но маловероятно, чтобы она достаточно полно отражала специфику ошибки. Поэтому следующим шагом должно быть использование доступных данных для уточнения версии с учетом некоторой специфики.
- 4. Доказательство выбранной гипотезы.** Этот шаг совпадает с шагом 4 в методе индукции.

### **Прослеживание логики в обратном порядке**

Отладка начинается в точке программы, где был обнаружен некорректный результат. Для этой точки на основании полученного результата следует установить, какими должны быть значения перемен. Мысленно выполняя из данной точки программу в обратном порядке и опять рассуждая примерно так: «если в этой точке состояние программы было таким, то в другой точке должно быть следующее состояние», можно достаточно быстро и точно локализовать ошибку.

### **Метод тестирования**

Последний метод отладки, основанный на «обдумывании», заключается в использовании тестов. Этот метод может показаться несколько странным, так как в начале главы отмечались различия между отладкой и тестированием. Однако существуют два типа тестов: тесты для тестирования, целью которых является обнаружение заранее не определенной ошибки, и тесты для отладки, цель которых - обеспечить программиста информацией, полезной для выявления местонахождения подозреваемой ошибки. Тесты для тестирования имеют тенденцию быть «обильными» (небольшим числом тестов пытаются покрыть большое число условий), а

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.  
тесты для отладки пытаются покрыть только одно условие или небольшое число условий.

### Принципы отладки

#### **Принципы локализации ошибок.**

- Думайте.
- Если вы зашли в тупик, отложите рассмотрение программы.
- Если вы зашли в тупик, изложите задачу кому-нибудь еще.
- Используйте средства отладки только как вспомогательные.
- Избегайте экспериментирования. Пользуйтесь им, как последним средством.

#### **Принципы исправления ошибок.**

- Там, где есть одна ошибка, вероятно, есть и другие.
- Находите ошибку, а не ее симптом
- Вероятность правильного нахождения ошибки не равна 100%.
- Вероятность правильного нахождения ошибки уменьшается с увеличением объема программы.

## **38. Архитектура программы. Цели выбора архитектуры. Декомпозиция.**

**Архитектура** – концепция, определяющая модель, структуру, выполняемые функции и взаимосвязь компонентов информационной системы.

Архитектура программной системы охватывает не только ее структурные и поведенческие аспекты, но и правила ее использования и интеграции с другими системами, функциональность, производительность, гибкость, надежность, возможность повторного применения, полноту, экономические и технологические ограничения, а также вопрос пользовательского интерфейса.

В архитектуре любой конкретной информационной системы часто можно найти влияния нескольких общих архитектурных решений.

Классификация программных систем по их архитектуре:

- Централизованная архитектура;
- Архитектура "файл-сервер";
- Двухзвенная архитектура "клиент-сервер";
- Многозвенная архитектура "клиент-сервер";
- Архитектура распределенных систем;
- Архитектура Веб-приложений;
- Сервис-ориентированная архитектура.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

### **Централизованная архитектура**

Реализовывалась на базе мейнфреймов; особенность такой архитектуры – полная "неинтеллектуальность" терминалов, их работой управляет хост-ЭВМ, пользователь не может настроить рабочую среду под свои потребности – все используемое программное обеспечение является коллективным

### **Архитектура "файл-сервер"**

Файл-серверные приложения – приложения, схожие по своей структуре с локальными приложениями и использующие сетевой ресурс для хранения программы и данных.

Функции сервера: хранения данных и кода программы.

Функции клиента: обработка данных происходит исключительно на стороне клиента.

### **Многозвенная архитектура "клиент-сервер"**

Это разновидность архитектуры клиент-сервер, в которой функция обработки данных вынесена на один или несколько отдельных серверов. Это позволяет разделить функции хранения, обработки и представления данных для более эффективного использования возможностей серверов и клиентов.

Среди многоуровневой архитектуры клиент-сервер наиболее распространена трехуровневая архитектура, предполагающая наличие следующих компонентов приложения: клиентское приложение ("тонкий клиент" или терминал), подключенное к серверу приложений, который в свою очередь подключен к серверу базы данных.

### **Архитектура распределенных систем**

Суть распределенной системы заключается в том, чтобы хранить локальные копии важных данных.

Каждый АРМ независим, содержит только ту информацию, с которой должен работать, а актуальность данных во всей системе обеспечивается благодаря непрерывному обмену сообщениями с другими АРМами. Плюс: обеспечение возможности персональной ответственности за сохранность данных. Такая архитектура системы также позволяет организовать распределенные вычисления между клиентскими машинами.

### **Архитектура Веб-приложений**

Особенности:

- отсутствие необходимости использовать дополнительное ПО на стороне клиента – это позволяет

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

- автоматически реализовать клиентскую часть на всех платформах;
- возможность подключения практически неограниченного количества клиентов;
- благодаря единственному месту хранения данных и наличия системы управления базами данных обеспечиваются минимальные требования для поддержания целостности данных;
- доступность при работоспособности сервера и каналов связи и недоступность при ее отсутствии
- достаточно низкая скорость Веб сервера и каналов передачи данных;

## **Сервис-ориентированная архитектура**

(SOA) - модульный подход к разработке программного обеспечения, основанный на использовании сервисов (служб) со стандартизованными интерфейсами; это парадигма организации и использования распределенных информационных ресурсов таких как: приложения и данные, находящихся в сфере ответственности разных владельцев, для достижения желаемых результатов потребителем, которым может быть: конечный пользователь или другое приложение.

Основными целями применения SOA являются:

- сокращение издержек при разработке приложений, за счет упорядочивания процесса разработки;
- независимость от используемых платформ, инструментов, языков разработки;
- повышение масштабируемости создаваемых систем;
- улучшение управляемости создаваемых систем.

## **Цели выбора архитектуры:**

Для того чтобы построить правильную и надежную архитектуру и грамотно спроектировать интеграцию программных систем необходимо четко следовать современным стандартам в этих областях. Без этого велика вероятность создать архитектуру, которая неспособна развиваться и удовлетворять растущим потребностям пользователей ИТ. В качестве законодателей стандартов в этой области выступают такие международные организации как SEI (Software Engineering Institute), WWW (консорциум World Wide Web), OMG (Object Management Group), организация разработчиков Java – JCP (Java Community Process), IEEE (Institute of Electrical and Electronics Engineers) и другие.

## **Декомпозиция**

Таким образом, сутью декомпозиции является уменьшение сложности информационной системы, а назначением — обеспечение возможности ее наилучшего осмысления с целью воплощения в заданной техническими требованиями форме.

Разбиение системы на части и к созданию общей архитектуры системы происходит на основе критериев декомпозиции:

1. Критерий разбиения на функции системы
2. Критерий разбиения на подсистемы
3. Критерий разбиения на классы
4. Критерий разбиения на объекты
5. Критерий разбиения на состояния
6. Критерий разбиения на задачи
7. Критерий определения интерфейсов

Рассмотрим применение функциональной декомпозиции на следующем примере: пусть необходимо разработать консольное приложение для представления десятичного действительного числа в систему счисления с основанием р.

Проектирование программы начинаем с функциональной декомпозиции задачи и построения на ее основе схемы иерархии логических модулей. Каждый логический модуль преобразует некоторые входные данные в определенный результат и может быть снова разделен на части.

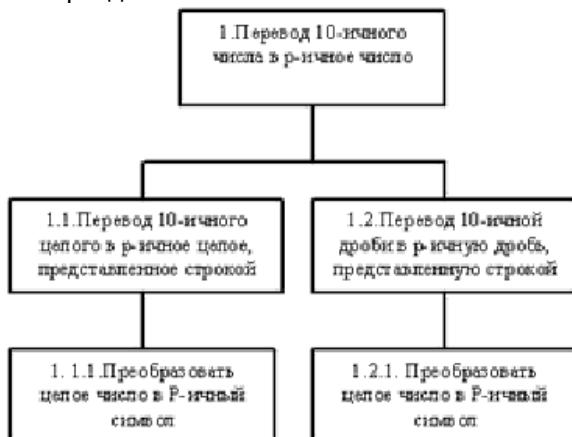


Рис. 5.1. Схема иерархии логических модулей.

Паттерны проектирования непосредственно связаны с предельными критериями декомпозиции, т.к. они были получены

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

в результате их применения. Они применяются в качестве основы проектирования информационных систем.

**Паттерн** – это многократно применяемая архитектурная конструкция, предоставляющая решение общей проблемы проектирования в рамках конкретного контекста и описывающая значимость этого решения. Он не является законченным образцом проекта, который может быть прямо преобразован в код, скорее это описание или образец для того, как решить задачу, таким образом, чтобы это можно было использовать в различных ситуациях.

### **39. Сопровождение ПО.**

**Сопровождение программного обеспечения** — процесс улучшения, оптимизации и устранения дефектов программного обеспечения (ПО) после передачи в эксплуатацию. Сопровождение ПО — это одна из фаз жизненного цикла программного обеспечения, следующая за фазой передачи ПО в эксплуатацию. В ходе сопровождения в программу вносятся изменения, с тем, чтобы исправить обнаруженные в процессе использования дефекты и недоработки, а также для добавления новой функциональности, с целью повысить удобство использования (юзабилити) и применимость ПО.

В модели **водопада**, сопровождение ПО выделяется в отдельную фазу цикла разработки. В спиральной модели, возникшей в ходе развития объектно-ориентированного программирования, сопровождение не выделяется как отдельный этап. Тем не менее, эта деятельность занимает значительное место, учитывая тот факт, что обычно около 2/3 жизненного цикла программных систем занимает сопровождение.

Сопровождаемость программного обеспечения — характеристики программного продукта, позволяющие минимизировать усилия по внесению в него изменений:

- для устранения ошибок;
- для модификации в соответствии с изменяющимися потребностями пользователей.

Программные средства являются одним из наиболее гибких видов промышленных изделий и эпизодически подвергаются изменениям в течение всего времени их использования.

Иногда достаточно при корректировке программного обеспечения внести только одну ошибку для того, чтобы резко снизилась его надежность или его корректность при некоторых исходных данных.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Для сохранения и повышения качества программного обеспечения необходимо регламентировать процесс модификации и поддерживать его соответствующим тестированием и контролем качества. В результате программное изделие со временем обычно улучшается как по функциональным возможностям, так и по качеству решения отдельных задач.

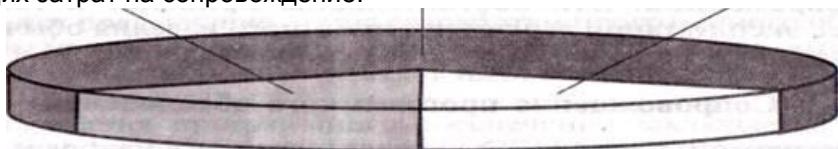
Работы, обеспечивающие контроль и повышение качества, а также развитие функциональных возможностей программ, составляют процесс сопровождения.

В процессе сопровождения в программное обеспечение вносятся следующие изменения, значительно отличающиеся причинами и характеристиками;

- исправление ошибок - корректировка программ, выдающих неправильные результаты в условиях, ограниченных техническим заданием и документацией. Исправление ошибок требуют около 20% общих затрат на сопровождение.

- регламентированная документами адаптация программного обеспечения к условиям конкретного использования, с учетом характеристик внешней среды или конфигурации аппаратуры, на которой предстоит функционировать программам. Адаптация занимает около 20% общих затрат на сопровождение.

- модернизация - расширение функциональных возможностей или улучшение характеристик решения отдельных задач в соответствии с новым или дополнительным техническим заданием на программное изделие. Модернизация занимает до 60% общих затрат на сопровождение.



Исправление ошибок 20%  
60% Адаптация 20%      Модернизация ПО

Затраты на сопровождение программного обеспечения

Первый вид изменений (исправление ошибок) является непредсказуемым и его трудно регламентировать.

Остальные виды корректировок носят упорядоченный характер и проводятся в соответствии с заранее подготавливаемыми планами и документами. Эти корректировки в наибольшей степени изменяют программные изделия и требуют наибольших затрат.

## ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Поэтому изменения, обусловленные ошибками, в большинстве случаев целесообразно по возможности накапливать и реализовывать их, приурочивая к изменениям, регламентированным модернизациями.

Однако некоторые ошибки вызывают необходимость срочного исправления программ. В этих случаях допустимо некоторое отставание корректировки документации при более срочном и регистрируемом исправлении самих программ.

Сопровождение программ — это «ложка дегтя» для каждого программиста, всегда помеха при начале разработки какого-либо нового проекта, заставляющая отвлекаться от его разработки и возвращаться к старым программам и старым проблемам.

Что делает сопровождение программного обеспечения крайне непривлекательным? Это плохо документированный код, недостаточно полное начальное проектирование и отсутствие внешней документации.

Если все этапы жизненного цикла разработки программного обеспечения выполнялись правильно, то сопровождение не будет вызывать серьезных проблем, а будет элементарной технической поддержкой и модификацией внедренного программного продукта.

Со временем, иногда через десятки лет, сопровождение программного обеспечения прекращается. Это может быть обусловлено: разработкой более совершенных программных средств; прекращением использования сопровождаемого программного продукта; нерентабельным возрастанием затрат на его сопровождение.

Отметим, однако, что программное изделие может долго применяться кем-либо и после прекращения его сопровождения от лица разработчика, потому, что этот некто может плодотворно использовать программное изделие у себя самостоятельно, без помощи разработчика.

Для того чтобы со временем прийти к обоснованному решению о прекращении сопровождения программного обеспечения, необходимо периодически оценивать эффективность его эксплуатации, возможный ущерб от отмены сопровождения. В некоторых случаях решение о прекращении сопровождения принимается при противодействии со стороны отдельных пользователей.

#### **IV. СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ**

##### **1. Назовите и охарактеризуйте уровни управления ИВС по эталонной модели ВОС. Назовите сетевые устройства и ПО, работающие на этих уровнях.**

Вычислительные сети, построенные на модели ВОС д. удовлетворять требованиям открытости, гибкости, эффективности. *Открытость* – возможность включения дополнительных ЭВМ, терминалов, узлов и линий связи без изменения технических и программных средств сети. *Гибкость* – сохранение работоспособности при изменении структуры сети в результате выхода из строя ЭВМ, линий, узлов связей. Допустимость изменения типов ЭВМ, а также возможность работы любых главных ЭВМ с терминалами различных типов. *Эффективность* – обеспечение требуемого качества обслуживания пользователя при минимальных затратах.

Архитектура эталонной модели ВОС является семиуровневой. Под уровнем понимается иерархическое подмножество функций ВОС, определяющих услуги смежному верхнему уровню по обмену данными и использующие для этого услуги смежного нижнего уровня. Услуга – это функциональная возможность, представляемая одному или нескольким вышерасположенным уровням.

7 – пользовательские службы; 6 – преобразование (представление) данных; 5 – организация и проведение диалога; 4 – представление сквозных соединений; 3 – прокладка соединений между системами; 2 – передача данных между смежными системами; 1 –сопряжение систем с физическими функциями системы.

Физический уровень 1: предоставляет механические, электрические, функциональные и процедурные средства для установления, поддержания и разъединения логических соединений между логическими объектами канального уровня; реализует функции передачи битов данных через физические среды.

Спецификации физического уровня определяют такие характеристики, как уровни напряжений, синхронизацию изменения напряжений, скорость передачи физической информации, максимальные расстояния передачи информации, физические соединители и другие аналогичные характеристики. На этом уровне работают электрические схемы передающих/принимающих звеньев сетевых устройств (адаптеров, напр.), репитеры, хабы. Единицы информации – простые биты данных.

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

Канальный уровень 2: предоставляет услуги по обмену данными между логическими объектами сетевого уровня и выполняет функции, связанные формированием и передачей кадров, обнаружением и исправлением ошибок, возникающих на физическом уровне посредством вычисления контрольной суммы, проверяет доступность среды передачи. Кадром называется пакет канального уровня; поскольку пакет на предыдущих уровнях может состоять из одного или многих кадров. Канальный уровень обеспечивает надежный транзит данных через физический канал. На этом уровне работают сетевые адAPTERы, коммутаторы. Протоколы – Ethernet, Token Ring, FDDI

Сетевой уровень 3: Этот уровень служит для образования единой транспортной системы, объединяющей несколько сетей с различными принципами передачи информации между конечными узлами. На этом уровне вводится понятие "сеть". В данном случае под сетью понимается совокупность компьютеров, соединенных между собой в соответствии с одной из стандартных типовых топологий и использующих для передачи данных один из протоколов канального уровня, определенный для этой топологии. Таким образом, внутри сети доставка данных регулируется канальным уровнем, а вот доставкой данных между сетями занимается сетевой уровень. Сообщения сетевого уровня принято называть пакетами (packets). При организации доставки пакетов на сетевом уровне используется понятие "номер сети". В этом случае адрес получателя состоит из номера сети и номера компьютера в этой сети. На этом уровне происходит формирование пакетов по правилам тех промежуточных сетей, через которые проходит исходный пакет и маршрутизация пакетов, т.е. определение и реализация маршрутов, по которым передаются пакеты. Маршрутизация сводится к образованию логических каналов. Еще одной важной функцией сетевого уровня является контроль нагрузки на сеть с целью предотвращения перегрузок. Примерами протоколов сетевого уровня являются протокол межсетевого взаимодействия IP стека TCP/IP и протокол межсетевого обмена пакетами IPX стека Novell. На этом уровне работают маршрутизаторы (аппаратные и программные).

Транспортный уровень 4: Работа транспортного уровня заключается в том, чтобы обеспечить приложениям или верхним уровням стека - прикладному и сеансовому - передачу данных с той степенью надежности, которая им требуется. Модель OSI определяет пять классов сервиса, предоставляемых транспортным уровнем. Эти виды сервиса отличаются качеством

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

предоставляемых услуг: срочностью, возможностью восстановления прерванной связи, наличием средств мультиплексирования нескольких соединений между различными прикладными протоколами через общий транспортный протокол, а главное - способностью к обнаружению и исправлению ошибок передачи, таких как искажение, потеря и дублирование пакетов. Как правило, все протоколы, начиная с транспортного уровня и выше, реализуются программными средствами конечных узлов сети - компонентами их сетевых операционных систем. В качестве примера транспортных протоколов можно привести протоколы TCP и UDP стека TCP/IP и протокол SPX стека Novell.

Сеансовый уровень 5 обеспечивает управление диалогом для того, чтобы фиксировать, какая из сторон является активной в настоящий момент, а также предоставляет средства синхронизации. Последние позволяют вставлять контрольные точки в длинные передачи, чтобы в случае отказа можно было вернуться назад к последней контрольной точке, вместо того, чтобы начинать все с начала. На этом уровне определяется тип связи (дуплекс или полудуплекс), начало и окончание сообщения. На практике немногие приложения используют сеансовый уровень, и он редко реализуется. (Предназначен для организации синхронизации диалога, ведущегося станциями сети. Последовательность и режим обменов запросами и ответами)

Представительный уровень 6: реализуются функции представления данных (кодирование, форматирование, структурирование). Например, на этом уровне выделенные для передачи данные преобразуются в кода EBCDIC в ASCII и т.п. Представительный уровень отвечает за то, чтобы информация, посылаемая из прикладного уровня одной системы, была читаемой для прикладного уровня другой системы. На этом уровне может выполняться шифрование и дешифрование данных, благодаря которому секретность обмена данными обеспечивается сразу для всех прикладных сервисов. Примером протокола, работающего на уровне представления, является протокол Secure Socket Layer (SSL).

Прикладной уровень 7 включает средства управления прикладными процессами. На этом уровне определяются и оформляются в блоки те данные, которые подлежат передачи по сети. Уровень включает, например, такие средства взаимодействия прикладных программ, как прием и хранение пакетов в "почтовых ящиках". Примерами таких прикладных процессов могут служить программы обработки крупномасштабных таблиц, программы обработки слов, программы банковских терминалов и т.д. Прикладной уровень

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

идентифицирует и устанавливает наличие предполагаемых партнеров для связи, синхронизирует совместно работающие прикладные программы, а также устанавливает соглашение по процедурам устранения ошибок и управления целостностью информации. Прикладной уровень также определяет, имеется ли в наличии достаточно ресурсов для предполагаемой связи. Единица данных, которой оперирует прикладной уровень, обычно называется *сообщением* (*message*).

## **2. Адресация в протоколах TCP/IP для сети Internet. Протокол ARP. Схемы рекурсивного и нерекурсивного режимов работы DNS-серверов.**

Различают два типа адресов. На канальном уровне используются адреса, называемые физическими. Это шестибайтовые адреса сетевых плат. На сетевом уровне используют сетевые адреса, называемые виртуальными или логическими. Эти адреса имеют иерархическую структуру и строятся на основе цифровых и буквенных выражений. Для поддержания таких адресов в Internet применяется система имен доменов (Domain Name System DNS). Единицей измерения здесь является домен (т.е. территория или область).

Узлы в Internet имеют адрес и имя. Адрес – это уникальная совокупность чисел: адреса сети и компьютера, которая указывает их местонахождение. Имя характеризует пользователя. Оно составляется в соответствии с доменной системой имен. Соответствие между IP-адресом и IP-именем узла сети устанавливается специальной службой директорий. В Internet это DNS.

IP-имя, называемое доменным именем, отражает иерархическое построение, глобальных сетей и поэтому состоит из нескольких частей. Корень иерархии обозначает либо страну, либо отрасль знаний (например, ru – Россия, edu – наука и образование). Корень занимает в IP-имени правую позицию, левее записываются в порядке подчинения остальные домены, составляющие локальную часть адреса. Перед символом @ указывается имя почтового ящика пользователя. Любой узел сети или домен в Internet однозначно идентифицируется таким полным доменным именем. Длина каждой метки в этом имени, разделенной точкой, не должна превышать 63 символов, а полная длина имени – 255 символов.

IP-адрес – это 32-битовое слово, записываемое побайтно в виде четырех частей, разделенных точками. Каждые подсеть и узел в подсети получают свои номера, причем для сети или подсети

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

может использоваться от одного до трех старших байтов, а оставшиеся байты – для номера узла. Узел в сети – это сетевое устройство, имеющее собственный адрес в сети, им может быть компьютер или маршрутизатор. Сейчас узел сети принято называть хостом. Какая часть IP-адреса относится к сети, определяется ее маской, выделяющей соответствующие биты в IP-адресе. Например, для некоторой сети маска может быть 255.0.0.0, а для ее подсети – 255.255.0.0. Тем самым описывается иерархия сетей.

Номера при включении нового узла выдает организация, предоставляющая телекоммуникационные услуги и называемая провайдером. Провайдер обеспечивает включение IP-адреса и соответствующего ему IP-имени в сервер службы адресов DNS. Это означает запись данных об узле в базу данных адресов локального узла DNS.

Сама система доменов представляет собой распределенную базу данных, размещенную на множестве компьютеров. Такие компьютеры называются серверами имен или просто DNS-серверами. Каждый сервер имен содержит обычно лишь информацию по одному домену, но знает адреса DNS-серверов вышестоящих и нижестоящих доменов. Программное обеспечение, которое обращается с серверами имен, называется клиентом DNS. Клиент DNS исполняет роль посредника между сетевыми приложениями и серверами имен и может функционировать как на отдельном компьютере, так и на сервере имен.

Сервер имен служит для перевода имени узла в соответствующий ему адрес при маршрутизации сообщения. Поскольку маршрутизация в сети осуществляется по IP-адресам, то перевод указанного пользователем IP-имени в IP-адрес с помощью DNS обязателен.

Клиенты DNS и серверы имен кэшируют в своей оперативной памяти данные, получаемые от других серверов имен. Время, в течение которого информация хранится в кэше, определяется источником и обычно составляет от десятков минут до нескольких суток. Это время зависит от частоты обращения к некоторому домену. Кэширование позволяет уменьшить трафик сети и снизить нагрузку на серверы имен.

Для повышения отказоустойчивости доменной системы имен одной зоны сети должны управлять как минимум два сервера имен – один выделяют как первичный и один или два – как вторичные серверы. При добавлении нового компьютера в сеть или изменении его IP-адреса информация о нем изменяется только на первичном сервере имен. Обновление содержимого

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

других серверов имен данной зоны сети происходит по мере устаревания содержимого их кэш-памяти.

Серверы имен могут работать в 2х режимах: рекурсивном и нерекурсивном. При нерекурсивном режиме работы сервер имен получает запрос от клиента DNS, например, на преобразование доменного имени в IP-адрес. Если доменное имя входит в зону управления сервера, то сервер возвращает клиенту ответ: положительный, т.е. IP-адрес, или отрицательный, если такого имени нет. Если имя не относится к зоне управления сервера, но присутствует в его кэше, то сервер ищет там. Если же требуемая информация не присутствует в кэше, то клиенту DNS отсыпается IP-адрес сервера имен, который ближе к нужному домену.

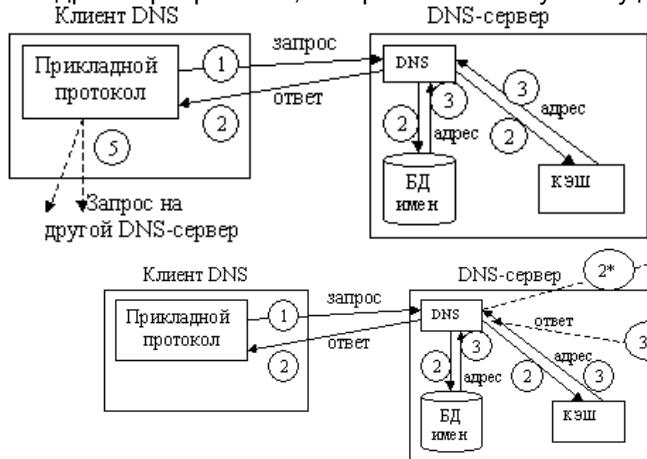


Рис. 10.6. Схема нерекурсивного (а) и рекурсивного (б) режима работы DNS-серверов.

### Схема работы

При рекурсивном режиме работы, в случае отсутствия нужной информации, DNS-сервер сам обращается по цепочке к другим серверам имен, а клиенту отсыпается уже готовый результат. В этом случае клиент освобождается от большей части работы по поиску информации в DNS. Однако рекурсивный режим работы используется намного реже нерекурсивного, т.к. нагрузка на серверы имен в этом случае значительно возрастает. А это является не оптимальным для клиента, поскольку при большой задержке ответа ему трудно определить произошел ли сбой в линии или просто опрашивается очень длинная цепочка серверов имен.

### Протокол ARP

Сетевая карта отправителя не знает MAC- адреса получателя. Поэтому какой- либо протокол должен этот адрес найти. Таким

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

протоколом является протокол ARP. ARP ищет MAC- адрес узла по его IP- адресу. Для бездисковых рабочих станций существует обратный протокол – реверсивный RARP, который выполняет обратное действие: по MAC- адресу сетевой карты сервера ищет его IP-адрес. Протокол ARP является универсальный протоколом, который работает с любым типом сетей и с любым типом физических и логических адресов. Этот протокол поддерживает специальную ARP- таблицу

IP- адрес узла	MAC- адрес	Статус записи
.....	.....	.....
- // -	FFFFFFFFFF	- // -

В этой таблице есть обязательно строчка, в которой в качестве MAC- адреса стоит широковещательный адрес, для того чтобы узел мог принимать широковещательный ARP- пакет. Статус записи определяет время нахождения строки в этой таблице. Статические записи могут заноситься вручную и хранятся в таблице до перезагрузки компьютера.

Динамические записи имеют два тайм- аута:

- первый таймаут наз. предельное время жизни, он равен 10 минутам.

- второй таймаут – таймаут активности узла. Он равен 2 минутам. Если от узла нет пакета в течение 2 минут, то запись удаляется. Если в течении 2 минут узел ответил, то таймаут сбрасывается, но через 10 минут запись будет удалена. В некоторых реализациях используется только таймаут активности узла, как и в таблице маршрутизации.

ARP- таблица хранится и формируется на специальных маршрутизаторах, но в каждом узле есть своя ARP- таблица, в которой хранятся данные об узлах, с которыми устанавливается связь. Если в ARP- таблице нет нужной записи, то протокол ARP отправляет ARP- запросы, т. е. пакеты специального формата.

Формат ARP- пакета

16 бит	16	8 бит	8	16	48	32		
48	32							
Тип сети	Тип прото- ко- ла	Длин- а - адре- са	Длин- а IP- адре- са	Тип опе- рац- ии	МА С А АО	IP А получ- ателя	MAC О ателя	IP АП
		адре- са						

802.2	
УИП	ПД
802.3	
УИП	ПД

Тип сети определяет тип локальной сети, в которую будет отправлен пакет. Для Ethernet тип сети = 1.

Тип протокола. В этом поле указывается какому протоколу сетевого уровня нужно вернуть ответ. Обычно к ARP обращается протокол IP. Но также могут быть обращения и др. протоколов (RIP, OSPF).

Длина MAC- адреса и длина IP- адреса нужны, чтобы ARP мог работать с любым типом адресов. Узел отправителя заполняет все поля, кроме MAC- адреса получателя. Если

RARP – незаполненным остается IP- адрес получателя. Это поле заполняет найденный узел, в котором формируется ответ.

ARP- запрос отправляется широковещательно на все сетевые карты. Причем, протокол ARP не инкапсулируется в IP- пакет.

ARP помещается в поле данных кадра канального уровня. Ответ отсылается по адресу узла отправителя. Станция, получившая ARP- пакет, сравнивает IP-адрес получателя со своим адресом, и если он совпал, то она заполняет поле MAC-адреса получателя и отправляет ответ. При таком обмене ARP- пакетами, ARP-таблицы пополняются в обоих узлах. Поскольку ARP- протокол не оперирует IP-адресами, значит ARP- пакет не может быть маршрутизируемым (отправлен в другую сеть), поэтому пакеты не проходят через маршрутизаторы. Если искомого узла нет в данной сети, то ARP протокол отправляет запрос маршрутизатору.

### **3. Реализация случайных методов доступа к моноканалу в ЛВС (МДКН и МДКН/ОК). Каким образом на основе МДКН/ОК мосты и маршрутизаторы имеют преимущество для доступа к моноканалу по сравнению с другими узлами сети?**

К случайным методам доступа относятся:

1. простой множественный доступ;
2. тактируемый множественный доступ;
3. множественный доступ с контролем несущей;
4. множественный доступ с контролем несущей/обнаружением коллизий.

При случайном доступе к среде передачи узел, желавший передать данные, начинает передачу в любой момент времени,

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

как только в этом возникла необходимость. При одновременной передаче 2-х и более узлов могут возникнуть конфликтные ситуации, т.е. могут наложиться во времени два или более сообщений, которые будут испорчены.

Распознавание конфликтов и оповещение о них пользователей сети выполняется центральной станцией, специально созданной для этих целей, или путем применения подтверждений о правильности принятых сообщений. В любом случае при обнаружении конфликта, пострадавшие станции предпринимают попытку повторной передачи потерянных сообщений. Станции должны распределять время начала попыток повторной передачи случайнym образом. Такой метод получил название простой множественный доступ(ПМД). При малой нагрузке в сети конфликты происходят редко. Однако когда нагрузка в сети начинает расти, приближаясь к максимальному значению, число конфликтов быстро увеличивается. Для этого метода относительный коэффициент использования канала  $k=0,18$ , т.е. очень мал.

Эффективность ПМД может быть повышена: с помощью разметки шкалы времени и разрешения пользователям начинать передачу сообщений только в начале каждого временного интервала, равного длительности сообщения. Указанный метод носит название тактируемого метода доступа (ТМД). Такая схема доступа требует синхронизации работы всех пользователей во времени. Для ТМД относительное использование канала составляет 0,38.

Одним из путей повышения общей эффективности методов ПМД и ТМД является реализация отказа от передачи кадра, если какой-то другой пользователь передает свой кадр. Для этого необходимо, чтобы станция "прослушивала" канал на наличие несущей частоты до того, как она приступит к передаче. Если канал уже занят, данная станция ожидает завершения текущей передачи, а затем начинает передачу собственного кадра. Этот метод получил название множественный доступ с контролем несущей. Между тем, сигналу требуется конечное время для того, чтобы достичь крайних точек сети, поэтому могут быть ситуации, когда две или более станций начнут передачу в одно и тоже время. В этом случае все передаваемые кадры искажаются, например как, показано на рис



## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

станция-отправитель перестает следить за передающей средой сразу же после того, как приступила к передаче собственного кадра. При этом, кадры хотя и искажаются, но передаются целиком до конца. В этом случае положительное подтверждение о приеме кадров не высыпается, и, по истечении некоторого времени, станции-отправители считают, что отправленные кадры подтверждены, и пытаются передать их повторно. Для этого следующая попытка передать поврежденные кадры возобновляется через случайный интервал времени.

Множественный доступ с контролем несущей и обнаружением конфликтов (МДКН/ОК, CSMA/CD) наиболее распространен среди случайных методов доступа. Перечисленные разновидности МДКН являются неэффективными, поскольку даже конфликтующие кадры передаются полностью. В сетях, где расстояния между станциями малы, время распространения сигнала по всем участкам сети невелико по сравнению с временем передачи кадра. Таким образом, период времени, в течении которого канал кажется свободным, хотя одна из станций передает информацию, очень короток. В этот период, называемый окном конфликтов, может быть передано более одного кадра, которые столкнутся друг с другом и будут испорчены.

Значительное усовершенствование может быть достигнуто посредством введения "прослушивания" сети как до начала передачи (т.е. контроль несущей), так и во время передачи (обнаружение конфликтов). Когда отправитель знает, что его кадр конфликтует с другим, то он прекращает передачу, и, тем самым, экономит время бесполезного захвата сети. В этом методе время повторной передачи кадра задается случайным способом.

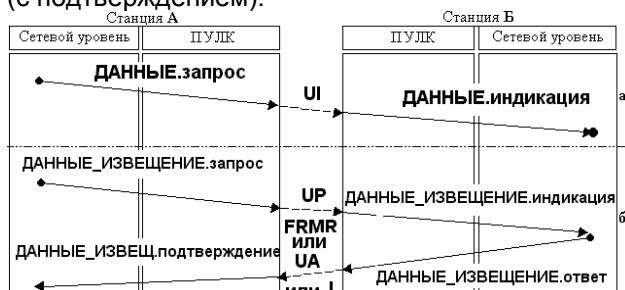
Если после определенного цикла повторных попыток все же не удается осуществить передачу, то станция-отправитель прекращает попытку и сообщает своему пользователю о возможности какой-то ошибки. По мере роста нагрузки на сеть падает интенсивность передач от отдельных станций.



Станция-отправитель может обнаружить конфликт передачи двумя способами: 1) сравнением данных, передаваемых в линию (неискаженных) и получением из нее (искаженных); 2) по появлению постоянной составляющей напряжения в линии, что обусловлено искажением используемого для представления данных манчестерского кода. Обнаружив конфликт, станция должна оповестить об этом партнера по конфликту, послав дополнительный сигнал затора после чего станции должны отложить попытки выхода в линию на случайный промежуток времени. Как станция-партнер по конфликту узнает, что сигнал затора предназначен именно ей, и как станция обнаружившая первый конфликт, узнает: с кем она конфликтует и кому передавать сигнал затора. → МДКН/OK – это широковещательный метод: первая станция пошлет сигнал всем, а станция, которая в это время передает, получив сигнал затора, поймет, что это ей.

#### 4. Объясните фазы работы протокола УЛК с установлением и без установления логического соединения. Ответ дополните диаграммой. Как для таких сетей отслеживается потеря передаваемых кадров?

Существует 2 режима обычный (без подтверждения) и расширенный (с подтверждением).



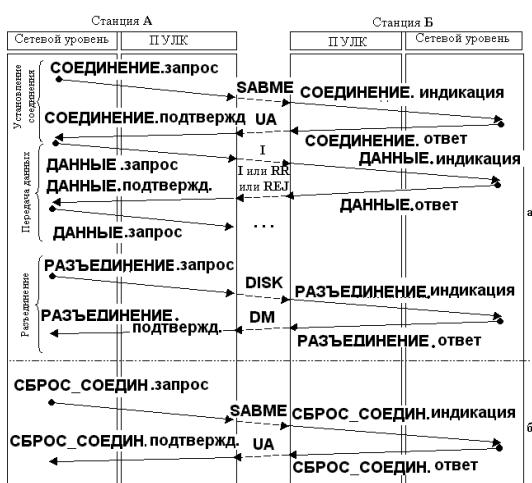
В основном режиме используется НПБД – UI (ненумерованная инф-ия). Протокол сетевого уровня выбирает режим работы

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

протоколов УЛК. Пр-л сет. ур-ня генерирует примитив **ДАННЫЕ.запрос**. Пр-л УЛК берет ПБД сетевого ур-ня по фиксир. адресу памяти и формирует ПБД ПУЛК – UI. Этот ПБД передается в сетевую карту. Пр-л УДС формирует на основе этого ПБД кадр, который передается в сеть. Пр-л УДС станции-получателя принимает этот кадр, отделяет от него ПБД, но не дешифрирует его. Этот БД переписывается из сетевой карты в ОЗУ компьютера и вызывается пр-л УЛК. Этот пр-л анализирует управляющие биты и узнает тип, затем формирует примитив **ДАННЫЕ.индикация**.

В расширенном режиме станция отправитель после отправки одного или нескольких ПБД формирует ПБД – UP(запрос передачи ответа). Правильность приема проверяет пр-л УДС. Если данные приняты правильно, то сетевой уровень в команде **ДАННЫЕ\_ИЗВЕЩЕНИЕ.ответ** указывает на необходимость передать UA (ненумерованное подтверждение) или I (передача инф-ии). Некорректный кадр подтверждается FRMR.

Кроме обмена данными по запросу сетевого уровня предусмотрен еще обмен идентифицирующей информацией



(XID) и выполнение тестовых функций. Каждая станция в любой момент времени может передать команду XID. Станция, которая приняла эту команду, должна послать ответ XID. И команда и ответ в поле данных содержат идентификатор класса станции.

Команда TEST также м.б. выдана в произвольный момент

времени. Станция, получившая данную команду, отправляет обратно ответ TEST. Размер поля данных обычно не превышает максимальную длину, установленную для конкретной сети. Однако, если известно, что некоторые станции могут обрабатывать кадры с полем данных большего размера, то эти станции распознаются с помощью команды TEST. Станция, которая может принять и отправить обратно кадр с полем данных слишком большой длины, сама реализует эти возможности. Если

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

станция располагает средствами для вычисления избыточной контрольной последовательности кадра, но не может принять и сохранить весь кадр, то в ответный кадр приноситься усеченное поле данных.

### а) Установление логического соединения

Прикладная программа сетевого уровня получив запрос от пользователя на пересылку данных, записывает в ОЗУ по фиксируемому адресу передаваемый пакет. При этом передается примитив СОЕДИНЕНИЕ.запрос. Каждый примитив кроме кода операции содержит дополнительный признаки. По этому примитиву пр-л УЛК формирует ПБД –SABME (установить асинхронный режим). На основе этого ПБД пр-л УДС формирует кадры, которые передаются в сеть. Пр-л УЛК на приемной стороне формирует протоколу сетевого уровня примитив СОЕДИНЕНИЕ.индикация. Если пр-л сетевого уровня подтверждает соединение, протоколу УЛК формируется примитив СОЕДИНЕНИЕ.ответ с установленными признаками соединения (передается UA – ненумерованное подтверждение). Если пр-л сетевого уровня не может уст-ть соед-е, то формирует DM (фаза разъединения). Пр-л УЛК 1-й станции дешифрирует принятый ПБД и передает пр-лу сетевого ур-ня примитив СОЕДИНЕНИЕ.подтверждение.

После установления соединения возможна передача данных в обоих направлениях. Если канал полудуплексный, то передачу начинает станция, запросившая соединение.

б) Фаза передачи состоит из множества циклов, которые заключаются в передаче данных и получении ответа, принимающая сторона обязательно формирует ответ на правильно принятые кадры.

Если на приемной стороне вычисленная контрольная последовательность совпадает с переданной, то пакет передается сетевому уровню и выдается подтверждение передающей стороне одним из двух способов: либо номер принятого кадра включается в передаваемый кадр, либо создается специальный кадр RR, содержащий номер принятого кадра

в) Фаза разъединения выполняется посылкой DISC, ответ – DM. начинает станция, запросившая соединение.

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

**5. Назовите принципы формирования протокольных блоков данных в рамках протоколов ЛВС. Инкапсуляция и декапсуляция сообщений. Принципы передачи команд между смежными протоколами одного узла сети и одинаковыми протоколами двух взаимодействующих узлов.**

В модели OSI сетевые функции распределены м/у 7 уровнями. Каждому ур. соотв-ют разн. сетевые операции, оборудование и протоколы.

На каждом уровне вып-ся опред. сетевые функции, которые взаимодействуют с функциями соседних уровней, вышележащего и нижележащего. Чем выше ур., тем более сложную задачу он решает.

Каждый ур. предоставляет несколько услуг, подготавливающих дан-е для доставки по сети на др. компьютер. Уровни отделяются друг от друга границами — интерфейсами. Все запросы от 1 ур. к другому передаются через интерфейс.

Задача каждого ур. - предоставление услуг вышележащему уровню, «маскируя» детали реализации этих услуг. При этом каждый ур. на одном комп-ре работает так, будто он напрямую связан с таким же уровнем на другом компьютере. Эта логическая (Вирт.) связь м/у одинаковыми ур. показана на рис. ниже. Однако в действительности связь осущ-ся м/у смежными ур. 1 комп-ра — ПО, работающее на каждом ур., реализует определенные сетев. фун-ии в соотв-ии с набором протоколов.

Перед подачей в сеть данные разбиваются на пакеты. Пакет (packet) — это ед. инф-ии, передаваемая м/у устр-вами сети как



добавляется нек-рая инф-я, форматирующая или адресная, к-рая необходима для успешной передачи данных по сети. На принимающей стороне пакет проходит ч/з все уровни в обратном порядке. ПО на каждом ур. читает инф-ю пакета, затем удаляет инф-ю, добавленную к пакету на этом же ур. отправляющей стороной, и передает пакет следующему ур. Когда пакет дойдет

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

до Прикладного ур., вся адресная инф-я будет удалена и дан-е примут свой исходный вид.

Вз/дей-е смежных ур. осущ-ся через интерфейс (И). И опред. услуги, к-рые нижний ур. предоставляет верхнему, и способ доступа к ним. Поэтому каждому ур. 1 компьютера «кажется», что он непосредственно вз/дей-ет с таким же ур. др. компьютера.

Когда сообщение достигает физического ур.. оно "обрастает" заголовками всех ур.

В стандартах ISO для обозначения единиц данных, с к-рыми имеют дело протоколы разных ур., используется общ. название протокольный блок данных (PDU). Для обознач-я блоков данных определенных ур. исп-ся специальные названия: кадр (frame), пакет (packet), дейтаграмма (datagram), сегмент (segment).

Протоколы 2 удаленных объектов взаимодействуют в три фазы:

- 1) фаза соединения – протоколы обмен-ся парам-ми будущей передачи (может обговариваться максимальный размер пакета, необходимость ответа на правильно принятые кадры, окно ответа). Устан-ся логич. или физич. соединение
- 2) фаза передачи – происходит передача данных, обнаружение и исправление ошибок
- 3) фаза разъединения – протоколы догов-ся о корректном разъединение, без потери дан-х

Смежные протоколы 1 станции образуют набор объектов. Это м.б. программы, драйвера, либо аппаратура. Смежные объекты взаимодействуют через точки доступа (ТДУ) к услугам. ТДУ – это либо программный, либо аппаратный порт.

Для смежных протоколов 1 станции им-ся 3 смежных взаимодействия:

- 1) одна точка доступа к одной услуге
- 2) 1:M
- 3)

Кроме ПБД, смежные протоколы обмен-ся спец. ком-ми (примитивы). Название примитива определяет группу команд:

- 1) запрос
- 2) индикация
- 3) ответ
- 4) подтверждение

Передавая примитив, протокол вышестоящего уровня, указывает нижестоящему, какие действия выполнить.

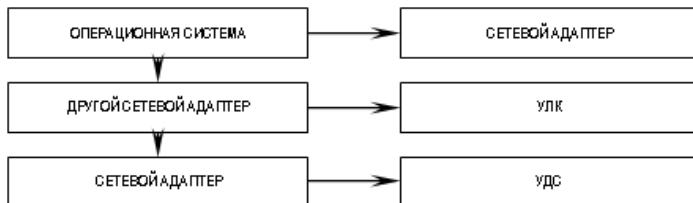
## **Дополнительно**

Смежные протоколы одной станции образуют набор объектов. Это могут быть прикладные программы, драйвера, либо аппаратура (сетевые адаптеры, либо модемы). Смежные объекты

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

взаимодействуют через точки доступа к услугам (рис 6.2). Точка доступа к услугам – это либо программный, либо аппаратный порт. Для смежных протоколов одной станции существует 3 смежных взаимодействия:

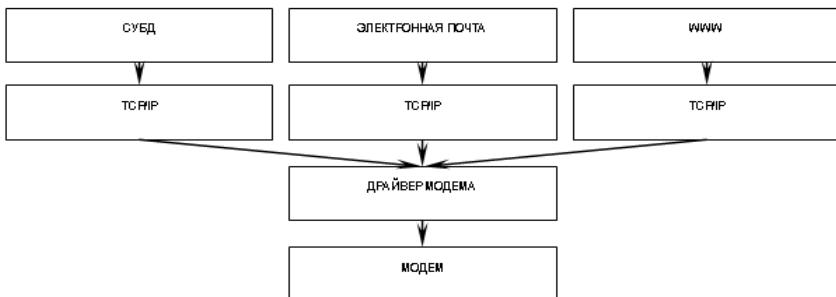
одна точка доступа к одной услуге



одна ко многим



многие к одной



Порция данных для протокола каждого уровня, называется протокольным блоком данных. При помощи протокольного блока данных передается, как данные пользователя, так и управляющие данные протокола. При переходе от одного протокола к другому, протокольный блок данных преобразуется в протокол данных услуги. К нему добавляется, управляющая информация протокола. Преобразования протокольного блока данных могут быть трех видов:

простое копирование

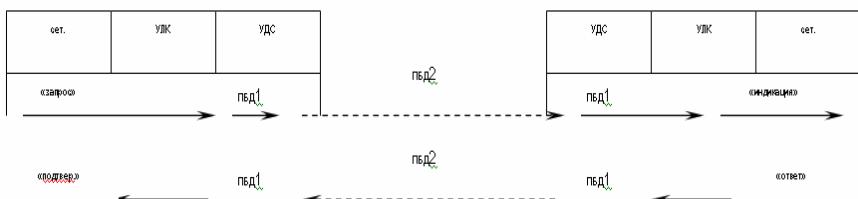
## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

разбиение протокольного блока данных на части  
сбор из частей протокольного блока данных, большого блока  
данных

Кроме протокольных блоков данных, смежные протоколы объединяются специальными командами, которые называются примитивами. Название примитива определяет группу команд: Запрос, индикация, ответ, подтверждение.

Передавая примитив, протокол вышестоящего уровня, указывает нижестоящему, какие действия выполнить.

Чтобы инициировать какие либо действия в сети, протокол сетевого уровня передает протоколу УЛК примитив «запрос», по этой команде протокол УЛК формирует протокольный блок данных, в котором содержатся либо команды протокола, либо данные пользователя. Затем этот протокольный блок данных посредством УДС передается в сеть. На приемной стороне протокол УЛК принял кадр, обязан сообщить протоколу сетевого уровня об изменении в сети. Для этого он передает этому



примитив «индикация». Протокол сетевого уровня анализирует состояние сети и формирует примитив «ответ», который протокол УЛК преобразует также в протокольный блок данных и передает ответ в сеть. На приемной стороне протокол УЛК должен завершить начатую фазу, для этого протоколу от уровня передается команда «подтверждение».

6. Зарисуйте структуру и назовите основные функциональные отличия повторителей, трансиверов и концентраторов ЛВС. На каком уровне эталонной модели ВОС функционирует каждое из этих устройств?

Для удлинения линии связи и восстановления сигналов в линии при передачи из одной физической среды в другую предназначены такие сетевые устройства, как трансиверы, репитеры и концентраторы. Все эти устройства работают на физическом уровне эталонной модели ВОС.



Рис. 1. Структура репитера

**Репитеры.** Репитер выполняет единственную функцию восстановления сигнала и передачи его в другие сегменты сети. Он не преобразует ни уровни сигналов сети, ни их физическую природу. Репитеры служат простыми двунаправленными ретрансляторами сигналов сети. Основная цель их применения – увеличение длины сети. Репитеры предназначены для соединения разных сегментов одной ЛВС, причем один репитер соединяет только два сегмента сети. Эти устройства предназначены для функционирования в таких типах физической среды, как витая пара, коаксиальный кабель, оптоволоконный кабель. На репитерах (повторителях) строятся в основном сети с кольцевой топологией, например Token Ring и FDDI.

Любой репитер не адресуется в сети. Структура репитера представлена на рис. 1.

**Трансиверы,** или приемопередатчики служат для двунаправленной передачи между сетевым адаптером и сетевым

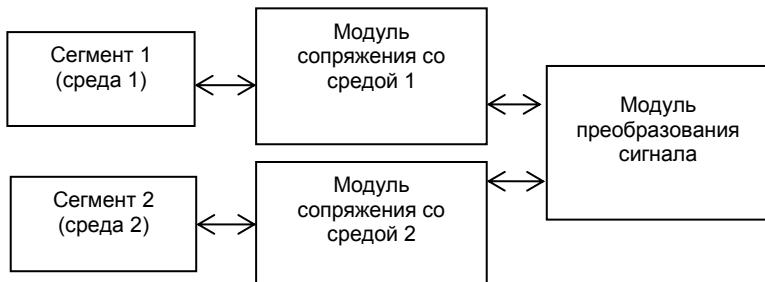


Рис. 2. Структура трансивера

кабелем или между двумя сегментами (отрезками сетевого кабеля). Основной функцией трансивера является усиление сигналов или преобразование их в другую форму для улучшения характеристик сети, например, для повышения

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

помехоустойчивости и/или увеличения расстояния между абонентами. Структура трансивера приведена на рис. 2.

Примером использования трансивера может служить подключение адаптеров сети Ethernet к «толстому» коаксиальному кабелю. В данном случае трансивер преобразует электрический сигнал для «тонкого» коаксиального кабеля в сигнал для «толстого» коаксиального кабеля и наоборот.

Более сложную функцию выполняет трансивер, преобразующий электрические сигналы сети в сигналы другой природы (оптические, радиосигналы и т.д.) с целью использования других сред передачи информации. Такие трансиверы также называют *конверторами среды*. Наиболее часто применяют оптоволоконные трансиверы, которые позволяют в несколько раз повысить допустимую длину кабеля сети.

Трансиверы, как и повторители, не выполняют никакой информационной обработки проходящих через них пакетов сообщений и не адресуются.

На трансиверах строятся в основном сети с шинной топологией и случайнym методом доступа, например Ethernet.

**Концентраторы.** К концентратору возможно подключение нескольких сегментов сети (обычно от 2 до 24) (рис. 3.), причем концентраторы работают с физической средой таких типов, как витая пара, коаксиальный кабель, оптоволоконный кабель. Концентраторы с точки зрения обработки информации можно условно разделить на активные и пассивные.

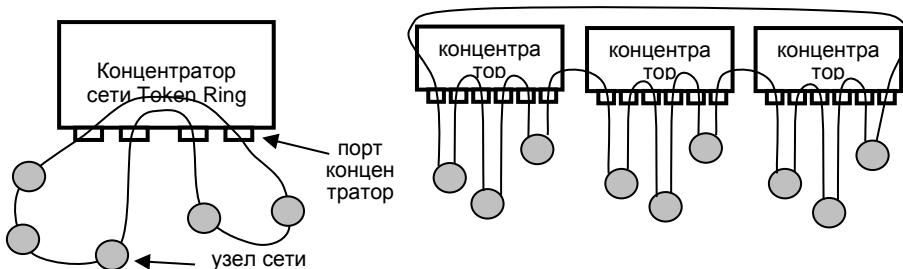


Рис. 3. Подключение узлов сети Token Ring через концентратор

**Пассивные** (репитерные) концентраторы выполняют функции нескольких повторителей (репитеров) или трансиверов, собранных в едином конструктиве. В связи с этим пассивные концентраторы никакой обработки информации не выполняют, а только восстанавливают и усиливают сигналы и могут также преобразовывать сигналы различной природы (например, электрические сигналы в оптические).

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

Пассивный концентратор должен принимать пакеты и отсыпать их во все сегменты сети, подключенные к нему, кроме того сегмента, откуда был принят пакет. Таким образом, концентратор выполняет функции усиления (функция репитера), преобразования природы среды (функция трансивера), а также соединяет сегменты сети.

К пассивному концентратору могут подключаться только части (сегменты) или отдельные абоненты одной и той же сети. Например, сегменты сети Ethernet, выполненные на тонком кабеле, на толстом кабеле, на оптоволоконном кабеле.

**Активные концентраторы** выполняют более сложные функции. В частности, они могут преобразовывать информацию и протоколы обмена, правда, это преобразование обычно очень простое. Структура активного концентратора представлена на рис. 4. Поскольку активный концентратор анализирует информацию, принятую из сети, т.е. распознает форматы пакетов данных, а для этого необходимо принять весь пакет, а не его отдельные биты, то модуль анализа и преобразования информации естественно должен содержать буфер для накопления данных. Активные концентраторы работают на **канальном уровне** модели ВОС, и являются разновидностью мостов.

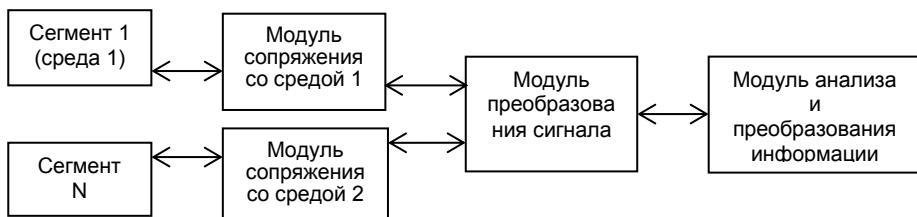


Рис. 4. Структура активного концентратора

## **7. Реализация маркерного метода доступа к моноканалу в ЛВС с кольцевой топологией. Особенности организации сети Token Ring на переключающих концентраторах.**

Из кольцевых ЛВС наиболее распространенными являются сети с передачей маркера по кольцу и среди них: 1) ЛВС типа Token Ring, которая стала основной для стандарта IEEE 802/5,

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

разработанная фирмой IBM; 2) сети FDDI на основе волоконно-оптической сети.

Сети Token Ring построены с использованием мостов, шлюзов, а

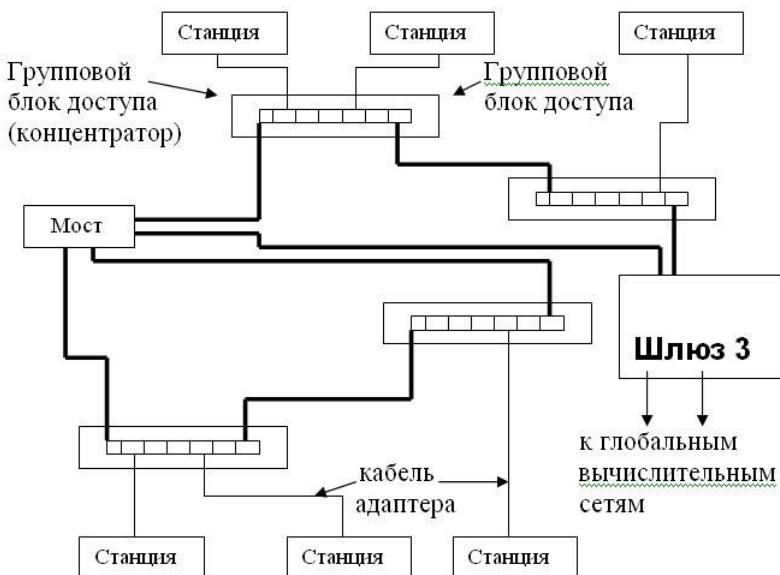


Рис. 1. Структура сети Token Ring.

также специальных схемных концентратов, управляющих конфигурацию сети и ее обслуживание (рис. 1.).

Сеть имеет комбинированную звездно-кольцевую конфигурацию и состоит из нескольких колец, работающих со скоростью 4М или 16Мбит/с и взаимодействующих через высокоскоростные мости. Область адресации кадра состоит из двух частей: первые два откола определяют адрес кольца, а следующие два – номер станции в кольце.

Операции в кольце могут выполняться в двух режимах: асинхронном и синхронном. Типичная реализация сети Token Ring определяет максимальное число станций 69; максимальное число концентратов 12.

Функционирование сети заключается в следующем. По сети циклирует маркер, имеющий структуру  
<ограничитель – Р – Т – М – Р – ограничитель>.

Если  $T \neq 0$ , то маркер свободен. Если свободный маркер проходит мимо станции, желающей передать данные, и приоритет станции не ниже приоритета маркера, записанного в Р, то станция преобразует маркер в информационный кадр:

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

устанавливает  $T = 1$  и записывает между полем  $R$  и конечным ограничи-телем адрес получателя, данные и другие сведения о соответствии с принятой структурой кадра. Информационный кадр проходит по кольцу и при этом: 1) каждая станция, готовая к передаче записывает значение своего приоритета в  $R$  (резервирует), если ее приоритет выше уже записанного в  $R$  значения; 2) станция-получатель, распознав свой адрес, считывает данные и отмечает в конце кадра (в бите "статус кадра") факт приема данных.

Совершив полный оборот по кольцу, кадр приходит к станции отправите-лю, которая организует состояние кадра. Если передача не произошла, то делается повторная попытка.

Если передача произошла, то кадр преобразуется в маркер с  $T = 0$ ,  $P \leftarrow R; R \neq 0$ , где  $P$  и  $R$  – трехбитовые коды. Здесь кадр данных из сети убирает-ся.

При следующем обороте маркер будет захвачен той станцией, у которой на предыдущем обороте оказался наивысший приоритет, записанный в  $R$ . Каждая станция имеет механизм обнаружения и устранения ошибок передачи.

Сеть Token Ring, рассчитанна на меньшии предельные расстояния и число станций, чем Ethernet, но она лучше приспособлена к повышенным нагрузкам.

Для увеличения скорости передачи в настоящее время кроме стандартного одно-маркерного доступа используется много маркерный доступ, а также вариант, при котором станция сразу после отправки кадра данных *отправляет кадр маркера*, не дожидаясь, когда ее кадр данных прейдет к ней для удаления после полного обхода кольца.

Для повышения надежности сети используют звездно-кольцевую топологию с переключающимися концентраторами (рис. 8.9).

Существует две разновидности Token Ring: четырех и шестнадцати Мбит.

В 4-х Мбит сети не используется резервирование приоритетов и используется стандартный маркерный метод доступа.

В 16-ти Мбит сетях используется процедура раннего высвобождения маркера и используется резервирование приоритетов по договоренности станций. Сеть собирается на

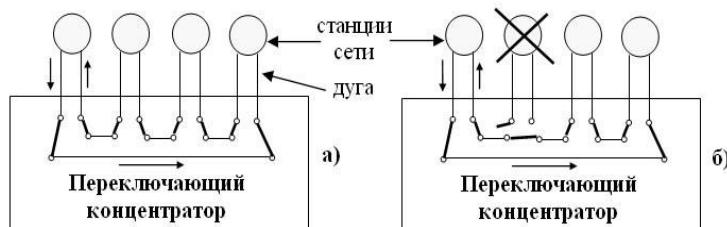


Рис. 8.9. Внутренняя структура переключающего концентратора при подключении всех дуг (а) и при отключении одной дуги (б)

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

переключающих концентраторах. Каждый порт концентратора питается от сети своего узла, и если узел отключен, тока в сети нет и концентратор автоматически отключает станцию из кольца. Концентратор также проверяет станцию на работоспособность. Если станция теряет маркер или передает постоянно искаженные кадры, то эту станцию из кольца исключает сам концентратор. Такие переключающие концентраторы два крайних разъема использую для подключения других концентраторов.

При появлении неисправности в какой-либо линии передачи или станции сети концентратор отключает от сети соответствующую дугу.

Конструктивно **концентратор** (8228 MAU) представляет собой автономный блок с восемью разъемами для подключения компьютеров с помощью адAPTERНЫХ кабелей и двумя (крайними) разъемами для подключения к другим концентраторам с помощью магистральных кабелей (рис. 8.10).

Сложное кольцо может иметь не только несколько переключающих концентраторов, но также содержать **ретрансляционные системы**, которые соединяют кольцо с другими локальными сетями.

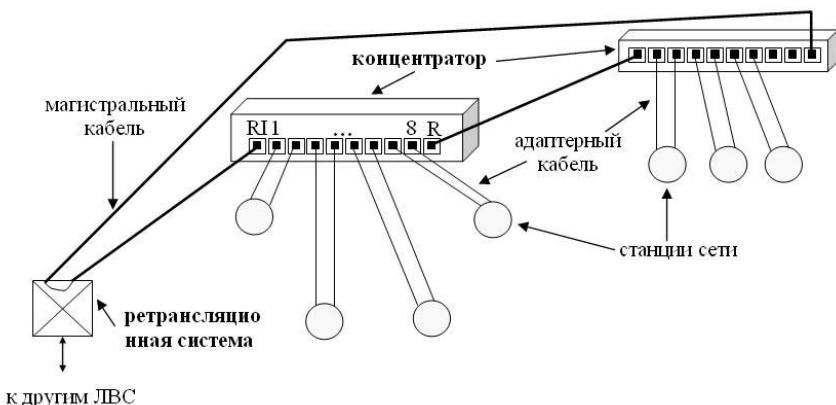


Рис. 8.10. Структура сложного кольца сети Token Ring

Станции подключаются к сети при помощи специальных двух входных **сетевых адAPTERов**. АдAPTERы Token Ring представляют собой платы расширения компьютера типа и ориентированы на системные шины ISA, EISA или PCI. Для

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

присоединения адаптерного кабеля адаптер имеет внешний 9-контактный разъем типа DIN. Также как и адаптеры Ethernet, адаптеры Token Ring имеют на своей плате переключатели или перемычки для настройки адресов и прерываний.

Для сети Token Ring на неэкранированной витой паре определены следующие ограничения:

- 1) максимальное число концентраторов типа IBM 8228 MAU - 12;
- 2) максимальное количество абонентов сети – 96;
- 3) максимальная длина кабеля между ПЭВМ и концентратором, а также между концентраторами – 45 м;
- 4) максимальная длина кабеля, соединяющего все концентраторы – 120 м;
- 5) скорость передачи данных – 4 Мбит/с и 16 Мбит/с.

**8. Объясните основные отличия в методе доступа для таких локальных сетей, как Token Ring и FDDI. Чем вызваны эти отличия. Синхронный и асинхронный режимы работы сети FDDI. Каким образом в сети FDDI определяется обрыв кабеля или отказ станции?**

Отличия метода доступа в том, что в сети FDDI время удержания маркера не яв-ся постоянной величиной, как в Token Ring. Это время зависит от загрузки кольца – при небольшой загрузке оно увеличивается, при перегрузках сильно уменьшается. Эти изменения в методе доступа касаются только асинхронного трафика, некритичного к большим задержкам передачи. Для синхронного трафика время удержания маркера постоянно. В остальном пересылка кадров между станциями на уровне MAC соответствует технологии Token Ring. Станции сети FDDI применяют алгоритм раннего высвобождения маркера, как и Token Ring со скоростью 16 Мбит/с

Механизм приоритетов Token Ring из 8 ступеней отсутствует. Трафик поделен на 2 класса: асинхронный (некритичный к задержкам) и синхронный (критичный к задержкам), последний обслуживается даже при перегрузках кольца. Тип определяется протоколами верхних уровней. Для передачи синхронных кадров станция всегда имеет право захватить маркер при его поступлении. Если же надо передать асинхронный кадр, то она замеряет время оборота маркера TRT, сравнивает его с максимальным временем оборота маркера по кольцу T\_Org и делает вывод о том, перегружена ли сеть. Если перегрузки нет ( $TRT < T_Org$ ), то станция может захватить маркер на время  $T_Org - TRT$  и передать столько асинхронных кадров, сколько успеет.

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

Если все станции хотят передать только асинхронные кадры, а маркер сделал оборот медленно, то на следующем обходе все станции пропускают маркер в режиме повторения, он быстро обходит кольцо, и на следующем цикле станции передают свои асинхронные кадры.

В FDDI на физическом уровне используется логическое кодирование 4B/5B. Которое высвобождает комбинации под служебные нужды. Самый важный служебный символ – Idle. Он постоянно передается между портами в паузах между передачами кадров. За счет этого станции и концентраторы постоянно получают информацию о состоянии соединений. Если этот символ не поступает, то фиксируется отказ физической связи и происходит реконфигурация сети.

Отличия в методах доступа вызваны 1. топологией сети FDDI (2 кольца), 2. скоростью (100 мбит) 3. и самое главное, назначением: Token ring предназначена для соединения рабочих станций и некрупных серверов, а FDDI предназначена для магистралей, подключения удаленных и мощных серверов и прочих задач, предъявляющих особые требования к качеству, скорости и надежности сети.

## **9. Основные функции транспортных и сетевых протоколов ИВС на примере протоколов TCP и IP. Взаимосвязь этих протоколов с другими протоколами ЭМ ВОС. Стратегии управления потоком данных.**

Взаимодействие между различными сетями, входящих в состав интегрированной сети, возложено на функции транспортного и сетевого уровней эталонной модели ВОС.

Функции транспортного уровня реализуются в конечных узлах и представляют собой следующие функции: разделение пакета на дейтограммы, если сеть работает без установления соединения; сборка сообщений из дейтограмм; обеспечение заданного уровня услуг, включающих заказ времени доставки, типа канала связи, возможности сжатия данных с частичной потерей информации; управление сквозными соединениями в сети с помощью специальных команд.

Протокол TCP – это дуплексный транспортный протокол с установлением соединения. Функциями протокола являются: а) упаковка и распаковка сегментов на концах транспортного соединения; б) установление вертикального канала путем обмена запросом и подтверждением на соединение; в) управление протоколом, который заключается в том, что получатель при подтверждении правильности передачи сообщает размер окна, т.е. диапазон номеров сегментов, которые получатель готов

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

принять; г) помещение срочных данных между специальными указателями , т.е. возможность управлять скоростью передачи.

Протокол IP – это дейтаграммный сетевой протокол без установления соединения. Назначение – это приспособление пакетов к особенностям промежуточных сетей и выбор направления передачи пакетов (т.е. маршрутизация). К функциям сетевых протоколов относятся: формирование пакетов с учетом требований промежуточных сетей (дополнение пакетов транспортного уровня заголовками, исключающими флаги, сетевые адреса получателя и отправителя, служебную информацию); управление потоками; маршрутизация; обнаружение неисправностей; ликвидация “заблудившихся” дейтаграмм.

### **Структура связей протокольных модулей**

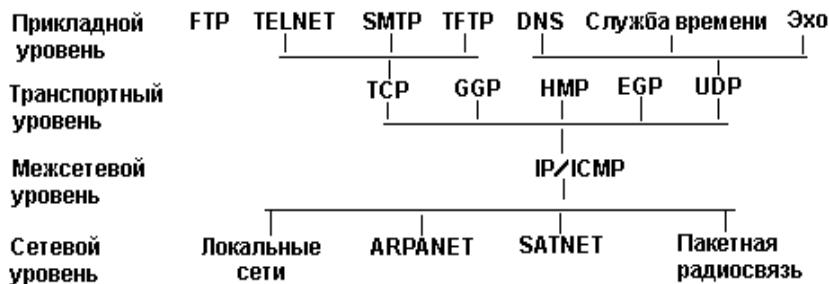
Рассмотрим потоки данных, проходящие через стек. В случае использования протокола TCP, данные передаются между прикладным процессом и модулем TCP. Типичным прикладным процессом, использующим протокол TCP, является модуль FTP. Стек протоколов в этом случае будет FTP/TCP/IP/Ethernet. При использовании протокола UDP, данные передаются между прикладным процессом и модулем UDP. Например, SNMP пользуется транспортными услугами UDP. Его стек протоколов выглядит так: SNMP/UDP/IP/Ethernet.

Модули TCP, UDP и драйвер Ethernet являются мультиплексорами  $n \times 1$ . Действуя как мультиплексоры, они переключают несколько входов на один выход. Они также являются демультиплексорами  $1 \times n$ . Как демультиплексоры, они переключают один вход на один из многих выходов в соответствии с полем типа в заголовке протокольного блока данных. Другими словами, происходит следующее:

1. Когда Ethernet-кадр попадает в драйвер сетевого интерфейса Ethernet, он может быть направлен либо в модуль ARP, либо в модуль IP. На то, куда должен быть направлен Ethernet-кадр, указывает значение поля типа в заголовке кадра.
2. Если IP-пакет попадает в модуль IP, то содержащиеся в нем данные могут быть переданы либо модулю TCP, либо UDP, что определяется полем "протокол" в заголовке IP-пакета.
3. Если UDP-датаграмма попадает в модуль UDP, то на основании значения поля "порт" в заголовке датаграммы определяется прикладная программа, которой должно быть передано прикладное сообщение.
4. Если TCP-сообщение попадает в модуль TCP, то выбор прикладной программы, которой должно быть передано

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

сообщение, осуществляется на основе значения поля "порт" в заголовке TCP-сообщения.



Мультиплексирование данных в обратную сторону осуществляется довольно просто, так как из каждого модуля существует только один путь вниз. Каждый протокольный модуль добавляет к пакету свой заголовок, на основании которого машина, принявшая пакет, выполняет демультиплексирование. Схема взаимозависимости протоколов семейства TCP/IP

Три стратегии управления потоком; они все направлены на борьбу с блокировками памяти в маршрутизаторах:

1) межузловое управление – основная функция (по умолчанию), заключается в том, что каждому порту отводиться одинаковое количество страниц в памяти, размер страницы равен максимальному размеру пакета локальной сети этого порта. Это приводит к ограничению длин канальных очередей.

2) Управление "вход-выход" направлено на предотвращение блокировок. Реализуется указанием в первом пакете сообщения его длины, что позволяет приемному узлу прогнозировать заполнение памяти и запрещать прием дейтаграмм определенных сообщений, если прогнозируется блокировка памяти.

3) управление внешними потоками осуществляется трем способами:

а) все потоки пакетов делятся на внутренние и внешние (внутренние – внутри одного домена, внешние – в другие домены). Разделение осуществляется по IP адресу. При переполнении памяти, наибольший приоритет имеют пакеты внутреннего потока, пакеты внешнего потока удаляются.

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

- b) каждый пакет должен иметь разделение доступа в другом сегменте сети, если такого разделения нет, то пакет удаляется. (широковещательные пакеты)
- c) если маршрутизатор обнаруживает узел, который создает перегрузку сети, он отправляет ему пакеты – заглушки. После отправки пакетов – заглушек, маршрутизатор удаляет пакеты от этого узла. После освобождения линии, узлу посыпается разрешение на передачу.

Протокол IP очень схож с протоколом УДС, поэтому протокол IP не осуществляет повторных передач искаженных кадров и не посыпает подтверждений правильности.

**10. Объясните понятие “окно конфликтов”. Как в сети Ethernet определяется эта величина и на что она влияет? Как в сети Ethernet на витой паре проводов уменьшить окно конфликтов?**

В сетях, где расстояния между станциями малы, время распространения сигнала по всем участкам сети невелико по сравнению с временем передачи кадра. Таким образом, период времени, в течении которого канал кажется свободным, хотя одна из станций передает информацию, очень короток. В этот период, называемый окном конфликтов, может быть передано более одного кадра, которые столкнутся друг с другом и будут испорчены.

Эта величина влияет на пропускную способность сети, а, следовательно, и на предел работоспособности. Причиной явления является недетерминированность алгоритма доступа к среде..



Длительность окна конфликтов определяется суммарным временем распространения сигналов по физическому уровню и по физической среде, то есть время распространения по кабелю + все задержки вносимые хабами, репитерами, сетевыми картами.. Только в этот промежуток времени в сети возможны конфликты.

## СЕТИ ЭВМ И ТЕЛЕКОММУНИКАЦИИ

Уменьшить это время можно путем сокращения длины кабеля между станциями, увеличением скорости распространения сигнала по кабелю (использование другого, более качественного кабеля), увеличением скорости модуляции, уменьшением количества хабов и репитеров между станциями.

### **Дополнительно.**

Станция, которая хочет передать кадр, должна сначала с помощью MAC-узла упаковать данные в кадр соответствующего формата. Затем для предотвращения смешения сигналов с сигналами другой передающей станции, MAC-узел должен прослушивать электрические сигналы на кабеле и в случае обнаружения несущей частоты 10 МГц отложить передачу своего кадра. После окончания передачи по кабелю станция должна выждать небольшую дополнительную паузу, называемую межкадровым интервалом (*interframe gap*), что позволяет узлу назначения принять и обработать передаваемый кадр, и после этого начать передачу своего кадра.

Одновременно с передачей битов кадра приемно-передающее устройство узла следит за принимаемыми по общему кабелю битами, чтобы вовремя обнаружить коллизию. Если коллизия не обнаружена, то передается весь кадр, поле чего MAC-уровень узла готов принять кадр из сети либо от LLC-уровня.

Если же фиксируется коллизия, то MAC-узел прекращает передачу кадра и посылает *jam*-последовательность, усиливающую состояние коллизии. После посылки в сеть *jam*-последовательности MAC-узел делает случайную паузу и повторно пытается передать свой кадр.

В случае повторных коллизий существует максимально возможное число попыток повторной передачи кадра (*attempt limit*), которое равно 16. При достижении этого предела фиксируется ошибка передачи кадра, сообщение о которой передается протоколу верхнего уровня.

Для того, чтобы уменьшить интенсивность коллизий, каждый MAC-узел с каждой новой попыткой случайным образом увеличивает длительность паузы между попытками. Временное расписание длительности паузы определяется на основе усеченного двоичного экспоненциального алгоритма отсрочки (*truncated binary exponential backoff*). Пауза всегда составляет целое число так называемых интервалов отсрочки.

Интервал отсрочки (*slot time*) - это время, в течение которого станция гарантированно может узнать, что в сети нет коллизии. Это время тесно связано с другим важным временным параметром сети - окном коллизий (*collision window*). Окно коллизий равно времени двухкратного прохождения сигнала

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

между самыми удаленными узлами сети - наихудшему случаю задержки, при которой станция еще может обнаружить, что произошла коллизия. Интервал отсрочки выбирается равным величине окна коллизий плюс некоторая дополнительная величина задержки для гарантии.

### V. ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

#### 1. Трансляция. Интерпретация и компиляция. Общие синтаксические критерии. Стадии трансляции.

Программа, написанная на языке программирования (ЯП), должна формально рассматриваться, во-первых, на соответствие синтаксическим правилам языка, а во-вторых, при синтаксической правильности проверяется на семантическую правильность, в-третьих (и это может сделать только человек) программа проверяется на смысловую правильность: является ли она осмысленной и если да, то какой смысл она несет. ЯП, следовательно, должен иметь ясные правила построения предложений (строгий синтаксис), и любое синтаксически верное предложение языка должно однозначно пониматься (однозначная семантика языка).

Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определённом смысле, равносильную первой. Цель трансляции — преобразовать текст с одного языка на другой, который понятен адресату текста. В случае программ-трансляторов, адресатом является техническое устройство (процессор) или программа-интерпретатор.

Транслятор обрабатывает файл с исходным текстом программы и осуществляет перевод программы с ЯП в понятную машине последовательность кодов.

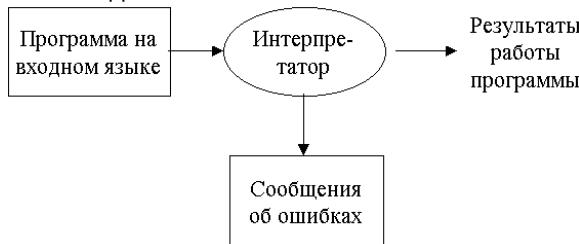
Трансляторы делятся на два класса: компиляторы и интерпретаторы.

Компилятор — это программа, которая обрабатывает программу на исходном языке и как результат выдает программу в некотором объектном коде.

Можно говорить о компиляторе как об отображении множества L1(исходная программа на ЯП L1) в множество L2(объектная программа на другом языке), т.е. K<sub>L3</sub>: L1->L2. L3 – язык на котором написан компилятор, возмож отличный от L1 и L2



Интерпретатор – это программа, которая переводит программу с исходного языка на некоторый внутренний язык и затем покомандно выполняет.



#### Достоинства интерпретатора:

- Передавать сообщения об ошибках пользователю часто оказывается легче в терминах оригинальной программы
- Версия программы на промежуточном языке нередко оказывается компактнее, чем машинный код, выдаваемый компилятором
- Изменение части программы не требует перекомпиляции всей программы (в качестве внутреннего языка используется польская запись)

#### Недостатки:

- Медленно работает, так как операторы промежуточного языка должны транслироваться каждый раз при выполнении
- Невозможно написать интерпретатор для некоторых языков программирования (Си, Паскаль и др.)

#### Общие синтаксические критерии языка:

- удобство чтения программы;
- удобство записи (противоречит п.1);
- простота трансляции;
- отсутствие разнотечений (Пример,  $A(i, j)$  – переменная массива или функция?);
- разные операторы должны выглядеть по-разному (Пример,  $DO I=1,5 \{цикл\}$  и  $DO I=1.5 \{присвоение\}$ );

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

- операторов в языке должно быть необходимое количество, желательно, чтобы разработчик ЯП доказал необходимость каждого оператора и полноту всей совокупности операторов;
- язык должен выполнять свое предназначение;
- язык должен быть легко расширяем;
- должно предполагаться множество внешних библиотек.

### Стадии трансляции:

Основным процессом при реализации любого языка программирования является трансляция программы, т. е. преобразование программы, представленной в исходном синтаксисе, в ее выполняемую форму. Процесс трансляции осложняется требованием эффективности выполнения программы (необходимо транслировать программу в эффективно выполняемые структуры, а именно в аппаратно интерпретируемый машинный код).

Логически трансляцию можно разбить на две основные части:

- I. Анализ исходной программы.
- II. Синтез выполняемой объектной программы.

Анализ программы можно разделить на следующие этапы:

1. Лексический анализ (ЛА).
2. Синтаксический анализ. (Син.А)
3. Семантический анализ. (Сем.А)

На этапе ЛА исходный текст подвергается лексической обработке. Программа разделяется на предложения, предложение делится на элементарные составляющие (лексемы). Каждая лексема распознаётся, определяется тип лексемы (имя, ключевое слово, литерал, символ операции или разделитель) и преобразуется в соответствующее двоичное представление. Блок транслятора, который соответствует ЛА, часто называется сканером или лексическим анализатором. Эта часть трансляции самая медленная, потому что необходимо сканировать и анализировать исходную программу, литера за литерой. Очень часто сканеры пишутся на ассемблере для того, чтобы они работали быстрее. В основе лексического разбора лежат регулярные грамматики.

Пример ЛА: исходный текст:  $A=(B+C)^*D$

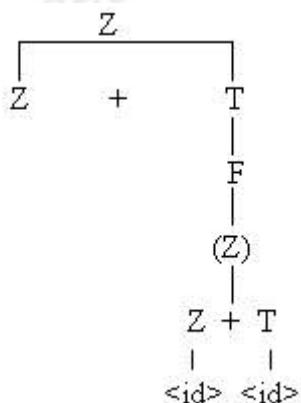
на выходе ЛА получим: <ид.1>=<(ид.2>+<ид.3>)\*<ид.4>

Син.А: лексемы, полученные на стадии ЛА, используются для идентификации более крупных программных структур: операторов, деклараций выражений и др. В ходе трансляции последовательности терминалных символов преобразуются в нетерминалы. Невозможность достижения очередного

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

нетерминала является признаком синтаксической ошибки в тексте исходной программы. Син.А на входе имеет набор лексем и пытается из этих лексем построить предложения языка. Фактически на этой стадии строятся синтаксические деревья (см. рисунок справа). В основе Син.А лежат КС-грамматики.

$$\begin{array}{l} Z \longrightarrow E + T | T \\ T \longrightarrow T \times F | F \\ F \longrightarrow F | (Z) | i \end{array}$$



Син.А обычно чередуется с Сем.А. Сначала Син. анализатор идентифицирует последовательность лексем, формируя синтаксическую единицу – выражение, оператор, вызов подпрограммы, декларацию. Затем вызывается семантический анализатор. Обычно для связи Син. анализатора с Сем. анализатором используется стек.

Сем.А служит соединением между фазой анализа и синтеза транслятора. На этой стадии выполняются следующие вспомогательные функции: ведение таблиц символов, обнаружение большинства ошибок, замена макросов макрорасширениями, выполнение инструкций, отнесенных ко времени компиляции.

При простой трансляции Сем. анализатор может генерировать объектный код, но, как правило, выходом этой стадии служит некоторая внутренняя форма выполняемой программы. В качестве внутреннего языка может использоваться прямая и обратная польские записи, запись триадами и тетрадами.,например A = B + C \* D; C D \* B + A = {постфиксная форма}

Обычно Сем. анализатор представляет собой набор отдельных процедур, связанных с определенной синтаксической конструкцией. Семантические процедуры взаимодействуют между собой посредством информации, которая хранится в различных таблицах (таблицах символов, типов, функций, меток).

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Синтез объектной программы - заключительная стадия трансляции. На этой стадии происходит оптимизация и генерация кода. На этапе генерации кода происходит замена операторов языка высокого уровня инструкциями ассемблера, а затем последовательностями машинных команд (т.е. внутренний код, сгенерированный Сем. анализатором заменяется на объектный). Оптимизация кода состоит из удаления неиспользованных переменных и пустых циклов, переноса наиболее используемых переменных в регистры, оптимизации логических выражений, просчета констант, вынесения из циклов независимых частей, оптимизации вычислений выражений.

Хотя в состав компилятора могут входить все компоненты, то есть ЛА, Син.А, Сем.А, генерация кода, их взаимодействие может осуществляться различными способами. Трехходовой компилятор – такая организация не может придать компилятору высокую скорость выполнения, так как обращение к файлам происходит три раза, но ее преимущество состоит в независимости фаз компиляции, т. е. переделки для новых машин коснутся только генерации кода. Любой блок может работать отдельно, и выгружаться из памяти после выполнения.

## **2. Грамматики и автоматы. Классификация Хомского**

Два наиболее распространенных способа конечного задания формального языка - это грамматики и автоматы. Грамматикой  $G[S]$  называется конечное непустое множество правил.  $S$  – это символ, который должен встретиться в левой части хотя бы одного правила. Он называется начальным символом.

**Алфавит** - это конечное множество символов. **Цепочкой символов в алфавите**  $V$  называется любая конечная последовательность символов этого алфавита. Цепочка, которая не содержит ни одного символа, называется **пустой цепочкой** ( $\varepsilon$ ). Более формально цепочка символов в алфавите  $V$  определяется следующим образом:

- (1)  $\varepsilon$  - цепочка в алфавите  $V$ ;
- (2) если  $\alpha$  - цепочка в алфавите  $V$  и  $a$  - символ этого алфавита, то  $\alpha a$  - цепочка в алфавите  $V$ ;
- (3)  $\beta$  - цепочка в алфавите  $V$  тогда и только тогда, когда она является таковой в силу (1) и (2).

**Язык** в алфавите  $V$  - это подмножество цепочек конечной длины в этом алфавите.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Обозначим через  $V^+$  множество, содержащее все цепочки в алфавите  $V$ , исключая пустую цепочку  $\epsilon$ . Обозначим через  $V^*$  множество, содержащее все цепочки в алфавите  $V$ , включая  $\epsilon$ . Следовательно,  $V^* = V^+ \cup \{\epsilon\}$ .

В заданной грамматике  $G$  символы, которые встречаются в левой части правил, называются нетерминалами или синтаксическими единицами языка. Символы, которые не входят во множество  $VN$ , называются терминальными символами или терминалами.

**Порождающая грамматика**  $G$  - это четверка  $(VT, VN, P, S)$ , где

$VT$  - алфавит *терминальных символов*,

$VN$  - алфавит *нетерминальных символов*, не пересекающийся с  $VT$ ,

$P$  - конечное подмножество множества  $(VT \cup VN)^+ \times (VT \cup VN)^*$ ; элемент  $(\alpha, \beta)$  множества  $P$  называется *правилом вывода* и записывается в виде  $\alpha \rightarrow \beta$ ,  $\square$  и  $\square$  - цепочки, построенные из букв алфавита  $V_T \cup V_N$ , который называют **полным алфавитом** (словарем) грамматики  $\Gamma$ .

$S$  - начальный символ грамматики,  $S \in VN$ .

Пусть  $G[Z]$ - грамматика. Цепочка  $x$  называется **сентенциальной формой**, если  $x$  выводима из начального символа  $Z$ , то есть если  $Z \Rightarrow^* x$ . **Предложение** – это сентенциальная форма, состоящая только из терминальных символов.

**Классификация Хомского** (грамматики класс-ся по виду их правил вывода)

**ТИП 0:**

Грамматика  $G = (VT, VN, P, S)$  называется **грамматикой типа 0**, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики).  $u ::= v$ , где  $u \in V^+$  и  $v \in V^*$ . То есть левая часть  $u$  может быть тоже последовательностью символов, а правая часть может быть пустой. Например,  $Aab \xrightarrow{} cBd$ . Грамматики нулевого типа не имеют практического применения. Так как в данной грамматике нет однозначности и они существуют только теоретически. Если ввести ограничение на правила подстановки, то получится языков типа 1.

**ТИП 1:**

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Грамматику  $t$  можно определить как *неукарачивающую*, или как *контекстно-зависимую*. Например,  $G[ S ]$ : 1)  $S \xrightarrow{} aSBa$  2)  $S \xrightarrow{} aba$  3)  $aB \xrightarrow{} Ba$  4)  $BB \xrightarrow{} bb$ .

Грамматика  $G = (VT, VN, P, S)$  называется *неукарачивающей грамматикой*, если каждое правило из  $P$  имеет вид  $\alpha ::= \beta$ , где  $\alpha \in (VT \cup VN)^+$ ,  $\beta \in (VT \cup VN)^+$  и  $|\alpha| \leq |\beta|$ .

Грамматика  $G = (VT, VN, P, S)$  называется *контекстно- зависимой (К3)*, если каждое правило из  $P$  имеет вид  $\alpha \square \beta$ , где  $\alpha = \xi_1 A \xi_2$ ;  $\beta = \xi_1 \gamma \xi_2$ ;  $A \in VN$ ;  $\gamma \in (VT \cup VN)^*$ ;  $\xi_1, \xi_2 \in (VT \cup VN)^*$ .

Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых неукарачивающими грамматиками, совпадает с множеством языков, порождаемых К3-грамматиками. Термин «контекстно-чувствительная» отражает тот факт, что можно заменить  $U$  на  $u$  лишь в контексте  $x...y$ .

### **ТИП2**

Дальнейшее ограничение дает класс грамматик, называемых *контекстно-свободными грамматиками* (КС-грамматики).

Грамматика называется контекстно-свободной, если все ее правила имеют вид:

$U ::= u$ , где  $U \in VN$  и  $u \in V^*$  (для укарачивающих,  $u \in V^+$  для неукарачивающих). Этот класс назван контекстно-свободным потому, что символ  $U$  можно заменить цепочкой  $u$ , не обращая внимание на контекст, в котором он встретился. Например,  $G[ R ]$ : 1)  $R \xrightarrow{} aa$  2)  $R \xrightarrow{} aAa$  3)  $A \xrightarrow{} b$  4)  $A \xrightarrow{} bA$ . В общем случае нельзя показать однозначность и безвозвратность КС – грамматик.

### **ТИП3**

Если мы ограничим правила еще раз, приведя их к виду  $U ::= x$  или  $U ::= Wx$ , или  $U ::= xW$ , где  $x \in T$ , а  $U$  и  $W \in VN$ , то получим грамматику типа 3, или *регулярную грамматику*. Регулярные грамматики играют основную роль как в теории языков, так и в теории автоматов. Например,  $G[ I ]$ : 1)  $I \xrightarrow{} aI$  2)  $I \xrightarrow{} bA$  3)  $A \xrightarrow{} bI$  4)  $A \xrightarrow{} a$ .

Грамматика  $G = (VT, VN, P, S)$  называется *праволинейной*, если каждое правило из  $P$   $t$ , где  $A \square tB$  либо  $A \square$  имеет вид  $A \in VN$ ,  $B \in VN$ ,  $t \in VT$ . Грамматика  $G = (VT, VN, P, S)$  называется *леволинейной*,  $t$ , где  $A \square Bt$  либо  $A \square$  если каждое правило из  $P$  имеет вид  $A \in VN$ ,  $B \in VN$ ,  $t \in VT$ . Грамматику типа 3 (регулярную,) можно определить как праволинейную либо как леволинейную.

Соотношения между типами грамматик:

- (1) любая регулярная грамматика является КС-грамматикой;
  - (2) любая регулярная грамматика является УКС-грамматикой;
  - (3) любая КС-грамматика является К3-грамматикой;
  - (4) любая КС-грамматика является неукорачивающей грамматикой;
  - (5) любая К3-грамматика является грамматикой типа 0.
- любая неукорачивающая грамматика является грамматикой типа 0.

**Замечание:** УКС-грамматика, содержащая правила вида

$A \rightarrow \epsilon$ , не является К3-грамматикой и не является неукорачивающей грамматикой.

Распознавателем для регулярной грамматики является конечный автомат (КА). Автоматами в данном контексте называют математические модели некоторых вычислительных устройств.

**Конечным автоматом (КА)** называют кортеж следующего вида:

$M(Q, V, \delta, H, S)$

$Q$  – конечное множество состояний автомата;

$V$  – конечное множество допустимых входных символов;

$H$  – начальное состояние автомата  $Q$ ;

$S$  – непустое множество конечных состояний автомата

$\delta$  – функция переходов. Находясь в некотором состоянии, функция  $\delta$  позволяет перейти в другое состояние по данному входному символу

Конечный автомат  $M(Q, V, \delta, H, S)$  называют **детерминированным конечным автоматом (ДКА)**, если в каждом из его состояний для любого входного символа есть однозначный переход из данного состояния в другое. В противном случае конечный автомат называют **недетерминированным (НКА)**

Для любого НКА можно построить эквивалентный ему ДКА. При построении компиляторов чаще всего используют полностью определенный (функция переходов  $\delta$  ДКА определена для каждого состояния автомата) ДКА.

На основе имеющейся регулярной грамматики можно построить эквивалентный ей КА, и наоборот, для заданного КА можно построить эквивалентную ему регулярную грамматику. Создав автомат на основе известной грамматики, мы получаем

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

распознаватель для лексических конструкций языка. Таким образом, удается решить задачу разбора для лексических конструкций языка, заданных произвольной регулярной грамматикой. Обратное утверждение также важно, поскольку позволяет узнать грамматику, цепочки языка которой допускает заданный автомат

Переход от регулярной грамматики к конечному автомату

Грамматика	Диаграмма состояний	КА																								
$S \Rightarrow C \perp$ $C \Rightarrow Ab Ba$ $B \Rightarrow b Cb$ $A \Rightarrow a Ca$	<pre> graph LR     H((H)) -- a --&gt; A((A))     H -- b --&gt; B((B))     A -- a --&gt; C((C))     A -- "⊥" --&gt; S((S))     B -- a --&gt; C     B -- "⊥" --&gt; S     C -- a --&gt; C     C -- "⊥" --&gt; S   </pre>	<table border="1"> <thead> <tr> <th></th> <th>a</th> <th>b</th> <th>⊥</th> </tr> </thead> <tbody> <tr> <td>H</td> <td>A</td> <td>B</td> <td></td> </tr> <tr> <td>A</td> <td></td> <td>C</td> <td></td> </tr> <tr> <td>B</td> <td>C</td> <td></td> <td></td> </tr> <tr> <td>C</td> <td>A</td> <td>B</td> <td>S</td> </tr> <tr> <td>S</td> <td>END</td> <td></td> <td></td> </tr> </tbody> </table>		a	b	⊥	H	A	B		A		C		B	C			C	A	B	S	S	END		
	a	b	⊥																							
H	A	B																								
A		C																								
B	C																									
C	A	B	S																							
S	END																									

$M(\{H, A, B, C, S\}, \{a, b, c, \perp\}, \delta, H, \{S\})$ , где  $\delta(H, a) = A$ ;  $\delta(H, b) = B$ ;  $\delta(A, a) = C$ ;  $\delta(B, a) = C$ ;  $\delta(C, a) = A$ ;  $\delta(C, b) = B$ ;  $\delta(C, \perp) = S$

### 3. Практические ограничения, налагаемые на грамматики. Отношения применимые к грамматикам.

Следующие ограничения налагаются на грамматики при их практическом использовании. Фактически эти условия не ограничивают множество языков, которые можно описать с помощью грамматик.

Правило, подобное  $U ::= U$ , очевидно, не является необходимым для грамматики; более того, оно приводит к неоднозначности грамматики. Поэтому далее полагается, что грамматика не содержит правил вида (1)  $U ::= U$

Кроме того, грамматика не должна содержать цепных правил, которые приводят к зацикливанию вывода. Например,

$A ::= B$      $B ::= C$      $C ::= A$      $A, B, C$  -нетерминалы.

Грамматики могут также содержать лишние правила, которые невозможно использовать в выводе хотя бы одного предложения. Например, в следующей ниже грамматике  $G[4]$  нетерминал  $<d>$  не может быть использован в выводе какого – либо предложения, так как он не встречается ни в одной из правых частей правил:

$Z ::= <b> e$   
 $<a> ::= <a> e | e$   
 $<b> ::= <c> e | <a> f$   
 $<c> ::= <c> f$

$\langle d \rangle ::= f$

Разбор предложений грамматики выполняется проще, если грамматика не содержит лишних правил. Можно с достаточным основанием утверждать, что если грамматика языка программирования содержит лишние правила, то она ошибочна. Следовательно, алгоритм, который обнаруживает лишние правила в грамматике, может оказаться полезным при проектировании языка.

Чтобы появиться в выводе какого-либо предложения, нетерминал  $U$  должен удовлетворять двум условиям. Во-первых, символ  $U$  должен встречаться в некоторой сентенциальной форме: (2)  $Z \Rightarrow^* xUy$  для некоторых цепочек  $x$  и  $y$ , где  $Z$  - начальный символ грамматики. Во-вторых, из  $U$  должна выводиться цепочка  $t$ , состоящая из терминальных символов: (3)  $U \Rightarrow^* t$  для некоторой  $t \in VT^*$ .

Если нетерминальный символ  $U$  не удовлетворяет этим условиям, то правила, содержащие  $U$  в левой части, не могут быть использованы ни в каком выводе. С другой стороны, если все нетерминалы удовлетворяют этим условиям, то грамматика не содержит лишних правил. Так, если  $U ::= u$  – правило, то, во – первых,  $Z \Rightarrow^* xUy \Rightarrow^* xu$  и  $y$ . Во-вторых, так как мы можем вывести цепочку терминальных символов для каждого нетерминала, содержащего в цепочке  $x$  и  $y$ , то  $x$  и  $y \Rightarrow^* t$  для некоторой  $t$  принадлежащей  $VT^*$ . В конечном итоге мы получаем  $Z \Rightarrow^* t$ , то есть вывод, в котором было использовано правило  $U ::= u$ .

( $\Rightarrow^+$  (“порождает”)- транзитивное замыкание(если  $aRb$  и  $bRc$ , то  $aRc$ ) отношения  $\Rightarrow$ .  $v \Rightarrow^* W$ , если  $v \Rightarrow^+ w$  или  $v=w$ ).

Условие (2) проверяет нетерминалы на недостижимость. Нетерминалы, которые не появляются ни в одной цепочке, выводимой из начального символа, называются недостижимыми нетерминалами.

Условие (3) проверяет нетерминалы на бесплодность. Нетерминалы, которые не порождают ни одной терминальной цепочки, называются бесплодными или мертвыми.

Нетерминалы, которые бесплодны или недостижимы, называются лишними.

В описанной выше грамматике G4 нетерминальный символ  $\langle c \rangle$  не удовлетворяет условию (3), так как при непосредственном выводе он должен быть заменен на  $\langle c \rangle f$ . Такая замена не может привести к цепочке из терминальных символов. Если отбросить все правила, которые нельзя использовать при порождении хотя бы одного предложения, то останутся правила  $Z ::= \langle b \rangle e$ ,  $\langle a \rangle ::= \langle a \rangle e$ ,  $\langle a \rangle ::= e$ ,  $\langle b \rangle ::= \langle a \rangle f$ .

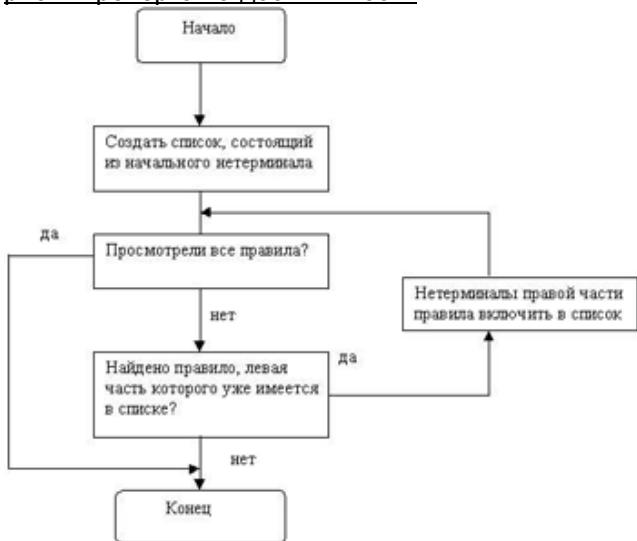
Грамматика  $G$  называется приведенной, если каждый нетерминал  $U$  удовлетворяет условиям (2) и (3), то есть не содержит лишних нетерминалов.

Нетерминал  $U$  удовлетворяет условию 2, тогда и только тогда, когда  $Z \text{ WITHIN}^+ U$ , где  $\text{WITHIN}^+$  является транзитивным замыканием отношения  $\text{WITHIN}$  (внутри).  $U \text{ WITHIN } S$  тогда и только тогда, когда существует правило  $U ::= \dots S \dots$ . Другими словами, если нетерминал в левой части правила является достижимым, то достижимы и все символы правой части этого правила. Это свойство выполняется, так как можно сначала вывести цепочку, содержащую символ, который является левой частью правила, и потом применить к ней это правило. Список, полученный в результате работы блок-схемы 1, содержит только достижимые нетерминалы, а все нетерминалы, не попавшие в этот список являются недостижимыми.

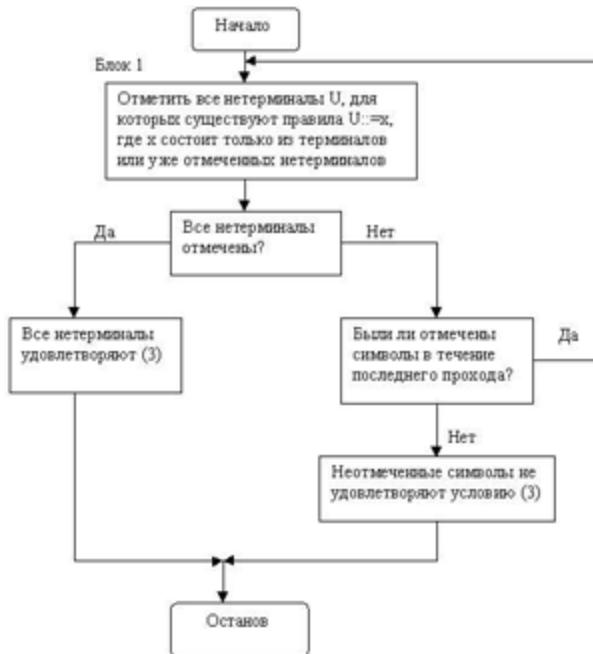
На рис.2 приведена блок схема алгоритма, который проверяет нетерминалы, встречающиеся в наборе правил на продуктивность (проверка условия (3)). Терминальный или нетерминальный символ называется продуктивным, если из него выводится какая-нибудь терминальная цепочка (то есть он не является бесплодным терминалом). Работа алгоритма начинается с того, что некоторым образом отмечаются те нетерминалы, для которых существует правило  $U ::= t$ , где  $t$  принадлежит  $VT^+$  (первое выполнение блока 1). Такие нетерминалы, очевидно удовлетворяют условию (3). Далее в алгоритме проверяются все неотмеченные нетерминалы  $U$  и для каждого из них делается попытка найти правило  $U ::= x$ , где  $x$  состоит только из терминальных символов или ранее отмеченных нетерминалов (повторное выполнение блока 1). Символы, для которых такое правило существует, также, очевидно удовлетворяют условию (3). Этот процесс продолжается до тех пор, пока либо все нетерминалы не будут отмечены, либо пока при выполнении блока 1 не будет отмечено ни одного символа. Работа алгоритма завершена. Неотмеченные нетерминалы не удовлетворяют условию (3).

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

### рис 1 Проверка на достижимость:



### рис 2 Проверка на продуктивность:



## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

### Отношения применимые к грамматикам.

Символы “=>” и “=> +” являются примерами отношений между цепочками символов.

Вообще говоря, (бинарным) отношением на множестве является любое свойство, которым обладают (или не обладают) любые два упорядоченных символа этого множества.

Другим примером является отношение МЕНЬШЕ ЧЕМ (<), определенное на множестве целых чисел:  $i < j$  тогда и только тогда, когда  $j - i$  является положительным ненулевым целым числом. Нематематическим является отношение ДРУГ, определенное на множестве людей. Некто А либо является другом В, либо нет.

Для отношений используются следующие обозначения: если между элементами с и d некоторого множества имеет место отношение, то мы пишем  $cRd$  (при этом важен порядок двух объектов: из  $cRd$  автоматически не следует  $dRc$ ).

Можно также рассматривать отношение как множество упорядоченных пар, для которых данное отношение справедливо:  $(c, d) \in R$  тогда и только тогда, когда  $cRd$ .

Говорят, что отношение  $R$  включает другое отношение  $R'$ , если из  $(c, d) \in R$  следует  $(c, d) \in R'$ .

Отношение, обратное отношению  $R$ , записывается как  $R^{-1}$  и определяется следующим образом:  $c R^{-1} d$  тогда и только тогда, когда  $dRc$ .

Отношение, обратное отношению БОЛЬШЕ ЧЕМ, есть МЕНЬШЕ ЧЕМ. Отношение, обратное отношению РЕБЕНОК, есть РОДИТЕЛЬ.

Свойства отношений.

Отношение  $R$  называется *рефлексивным*, если  $cRc$  справедливо для всех элементов с, принадлежащих множеству. Например, отношение МЕНЬШЕ ЧЕМ ИЛИ РАВНО ( $=<$ ) является рефлексивным ( $i = < j$  для всех действительных чисел  $i$ ), тогда как отношение МЕНЬШЕ ЧЕМ таковым не является.

Отношение  $R$  называется *симметричным*, если  $aRb$  и  $bRa$  справедливо для всех элементов с, принадлежащих множеству. Например, отношение РАВНО.

Отношение  $R$  называется *транзитивным*, если  $aRb$  и  $bRc$  влечет за собой  $aRc$ . Отношение МЕНЬШЕ ЧЕМ над целыми числами является транзитивным.

Например, рассмотрим отношения  $R$  и  $R'$  над целыми числами, определенные следующим образом:

$ARb$  тогда и только тогда, когда  $b=a+1$ ,

$APb$  тогда и только тогда, когда  $b=a+2$ .

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Очевидно, что  $aRpb$ , тогда и только тогда, когда есть такое с, что  $aRc$  и  $cRb$ , то есть тогда и только тогда, когда  $b=a+3$ .

Аналогично можно определить степень отношения  $R^n$ , как произведение п отношений R, т.е. степень отношения R формально выражается следующим образом:  $R^1=R$ ,  $R^2=RR$ ,  $R^n=R^{n-1}R=RR^{n-1}$  для  $n > 1$ , т.о.  $aR^nb$  означает, что существуют  $a_1, a_2, \dots, a_{n-1}$  такие, что  $a_1Ra_2, a_2Ra_3, \dots, a_{n-1}Rb$ . Определим  $R^0$  как единичное отношение:  $aR^0b$  тогда и только тогда, когда  $a=b$ .

И, наконец, транзитивное замыкание  $R^+$  отношения R определяется следующим образом:  $aR^+b$  тогда и только тогда, когда  $aR^nb$  для некоторого  $n>0$ .

Оно называется так потому, что включается в любое транзитивное отношение, которое включает в себя R. Другими словами, предположим, что P- транзитивное отношение, которое включает в себя R: и из того, что  $(c, d) \in R$  следует  $(c, d) \in P$ . Тогда P также включает в себя  $R^+$ .

Очевидно, если  $aRb$ , то  $aR^+b$ . Ясно, почему для транзитивного замыкания выбрано обозначение  $R^+$ ; когда отношения рассматриваются как множества упорядоченных пар, мы имеем  $R^+=R^{1U}R^{2U}R^{3U}\dots$

Определим рефлексивное транзитивное замыкание  $R^*$  отношения R как  $aR^*b$  тогда и только тогда, когда  $a=b$  или  $aR^+b$ .

Таким образом,  $R^*=R^{0U}R^{1U}R^{2U}\dots$

Легко доказать, что, если отношение R определено на конечном множестве A, то  $R^+=R^{1U}R^{2U}R^{3U}\dots R^n$ , где n мощность множества A. Это утверждение для нас будет важным в дальнейшем.

Если задана грамматика и нетерминальный символ U, то нам необходимо знать множество головных символов в цепочках, выводимых из U.

Определим отношение FIRST (первый) в конечном словаре V грамматики следующим образом:

U FIRST S тогда и только тогда, когда существует правило  $U ::= S\dots$

(Тремя точками (...) обозначается цепочка (возможно пустая), которая в данный момент нас не интересует.)

Затем, согласно определению транзитивного замыкания, имеем:

U FIRST<sup>+</sup> S тогда и только тогда, когда существует непустая последовательность правил  $U ::= S1\dots, S1 ::= S2\dots, \dots, Sn ::= Sn\dots$

Из этого, очевидно, вытекает, что U FIRST<sup>+</sup> S тогда и только тогда, когда  $U => +S\dots$

Пример. Чтобы проиллюстрировать оба отношения FIRST и FIRST<sup>+</sup>, выпишем несколько правил и справа от каждого из них укажем отношение FIRST, выведенное из этого правила.

Правило	Отношение		
A ::= Af	A	FIRST	A
A ::= B	A	FIRST	B
B ::= DdC	B	FIRST	D
B ::= De	B	FIRST	D
C ::= e	C	FIRST	e
D ::= Bf	D	FIRST	
			B

Таким образом, мы имеем следующие пары в  $\text{FIRST}^+$ : (A,A) (A,B) (A,D) (B,B) (B,D) (D,B) (D,D) (C,e).

Следовательно, головными символами цепочек, выводимых из A, будут A, B, D из B-B и D, из C-e.

Имеется еще три других множества, которые могут быть определены тем же путем, что и  $\text{FIRST}^+$ . Первое из них – множество символов, которыми оканчиваются цепочки, выводимые из некоторого символа U. Они определяются для всех символов U через отношение  $\text{LAST}^+$ , являющееся транзитивным замыканием отношения  $\text{LAST}$  (последний).

#### 4. Синтаксический анализ. Синтаксические деревья. Задача разбора. Однозначность разбора. Канонический разбор. Основа. Разбор сверху вниз, снизу вверх.

Синтаксический анализ – это процесс, который определяет, принадлежит ли некоторая последовательность лексем языку, порождаемому грамматикой.

Анализаторы реально используемых языков обычно имеют линейную сложность; это достигается, например, за счет просмотра исходной программы слева направо с заглядыванием вперед на один терминальный символ (лексический класс).

Цепочка принадлежит языку, порождаемому грамматикой, только в том случае, если существует ее вывод из цели этой грамматики. Процесс построения такого вывода (а, следовательно, и определения принадлежности цепочки языку) называется *разбором*.

С практической точки зрения наибольший интерес представляет разбор по контекстно-свободным грамматикам. Их порождающей мощности достаточно для описания большей части синтаксической структуры языков программирования, для различных подклассов КС-грамматик имеются хорошо разработанные практически приемлемые способы решения задачи разбора.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Рассмотрим основные понятия и определения, связанные с разбором по КС-грамматике.

Вывод цепочки  $\beta \in (VT)^*$  из  $S \in VN$  в КС-грамматике  $G = (VT, VN, P, S)$ , называется *левым (левосторонним)*, если в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого левого нетерминала.

Вывод цепочки  $\beta \in (VT)^*$  из  $S \in VN$  в КС-грамматике  $G = (VT, VN, P, S)$ , называется *правым (правосторонним)*, если в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого правого нетерминала.

В грамматике для одной и той же цепочки может быть несколько выводов, эквивалентных в том смысле, что в них в одних и тех же местах применяются одни и те же правила вывода, но в различном порядке.

Например, для цепочки  $a+b+a$  в грамматике

$G = (\{a,b,+\}, \{S,T\}, \{S \rightarrow T \mid T+S; T \rightarrow a \mid b\}, S)$

можно построить выводы:

$$(1) \quad S \rightarrow T+S \mid T+T+S \mid T+T+T \mid a+T+T \mid a+b+T \mid a+b+a$$

$$(2) \quad S \rightarrow T+S \mid a+S \mid a+T+S \mid a+b+S \mid a+b+T \mid a+b+a$$

$$(3) \quad S \rightarrow T+S \mid T+T+S \mid T+T+T \mid T+T+a \mid T+b+a \mid a+b+a$$

Здесь (2) - левосторонний вывод, (3) - правосторонний, а (1) не является ни левосторонним, ни правосторонним, но все эти выводы являются эквивалентными в указанном выше смысле.

Для КС-грамматик можно ввести удобное графическое представление вывода, называемое деревом вывода, причем для всех эквивалентных выводов деревья вывода совпадают.

Дерево называется *деревом вывода* (или *деревом разбора*) в КС-грамматике  $G = (VT, VN, P, S)$ , если выполнены следующие условия:

(1) каждая вершина дерева помечена символом из множества  $(VN \cup VT \cup \epsilon)$ , при этом корень дерева помечен символом  $S$ ; листья - символами из  $(VT \cup \epsilon)$ ;

(2) если вершина дерева помечена символом  $A \in VN$ , а ее непосредственные потомки - символами  $a_1, a_2, \dots, a_n$ , где каждое  $a_i \in (VT \cup VN)$ , то  $A \rightarrow a_1a_2\dots a_n$  - правило вывода в этой грамматике;

(3) если вершина дерева помечена символом  $A \in VN$ , а ее единственный непосредственный потомок помечен символом  $\epsilon$ , то  $A \rightarrow \epsilon$  - правило вывода в этой грамматике.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Синтаксические деревья помогают понять синтаксис предложений. В качестве примера рассмотрим следующую грамматику G1, содержащую 13 правил:

<число> ::= <чс>  
<чс> ::= <чс> <цифра>  
<чс> ::= <цифра>

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

Построим синтаксическое дерево для следующего вывода предложения грамматики G1:

<число> => <чс> =><чс>  
<цифра>=><цифра><цифра>=>2<цифра>=>22 (1)

Отправляемся от начального символа <число>, нарисуем его куст для того, чтобы указать первый непосредственный вывод (рис.1 а). Куст узла – это множество подчиненных ему узлов (символов). Символы куста образуют цепочку, которая заменяет имя куста в первом непосредственном выводе <число> ::= <чс>.

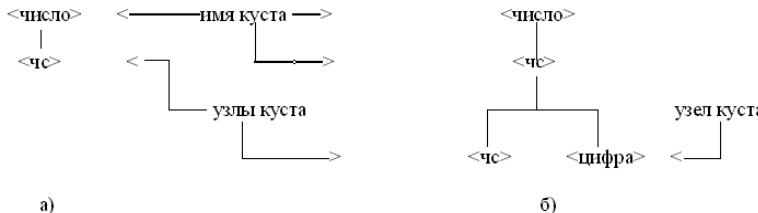


Рис. 1. Синтаксические деревья для двух выводов.

Чтобы показать второй вывод, из узла, представляющего заменяемый символ рисуется куст, узлы которого образуют цепочку, заменяющую этот символ (рис.1 б). Поступая таким же образом, мы построим одну за другой три синтаксических диаграммы, показанные на рис. 2.

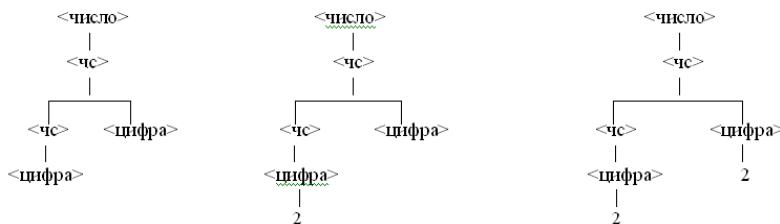


Рис. 2. Синтаксические деревья для вывода (1).

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

**Концевые (висячие) узлы** синтаксического дерева – это узлы, не имеющего подчиненного куста. При чтении слева направо концевые узлы образуют цепочку, вывод которой представлен деревом. Таким образом, после третьего непосредственного вывода в (1) цепочка концевых узлов – суть <цифра> <цифра> (см. рис.2а).

**Концевой куст** – это куст, все узлы которого концевые. На рис. 1б один концевой куст; его узлы <чс> и <цифра>. На рис.2в два концевых куста с узлами 2 и 2.

Когда речь идет о деревьях, часто используется следующая терминология. Пусть  $N$  – узел дерева. **Сыновьями**  $N$  называются узлы куста, подчиненного  $N$ .  $N$  - их отец. Сыновья называются братьями, самым младшим считается самый левый из них. На рис.2 б <чс> является единственным сыном узла <число>. Тот же самый <чс> имеет двух сыновей: <чс> и <цифра>. Отцом узла 2 является <цифра>.

**Поддерево синтаксического дерева** состоит из узла дерева (называемого корнем поддерева) вместе с той частью дерева (если она имеется), которая исходит от него. Поддеревья тесно связаны с фразами; концевые узлы образуют фразу для корня данного поддерева. Проследим это более подробно. Если  $U$  – корень поддерева и если  $u$  – цепочка из концевых узлов поддерева, то  $U \Rightarrow + u$ . Пусть  $x$ - цепочка концевых узлов слева от  $u$ , а  $y$ - цепочка концевых узлов справа от  $u$ , то есть  $xy$ -сентенциальная форма, заданная деревом. Тогда  $Z \Rightarrow^* xyUy$ , а это означает, что  $xy$  есть фраза для  $U$  в  $xy$ .

**Построение вывода по дереву.**

Можно восстановить вывод по синтаксическому дереву при помощи обратного процесса. Из рис. 2 с видно, что концевые узлы образуют цепочку 22. Самый правый концевой куст указывает непосредственный вывод:  $2 <\text{цифра}> \Rightarrow 22$ .

Чтобы пройти по синтаксическому дереву до  $2 <\text{цифра}>$ , мы отсекаем куст от дерева – удаляем его. Например, отсечение этого куста (на рис. 2 с) дает нам дерево на рис. 2 б. Этот процесс часто называют *непосредственной редукцией*.

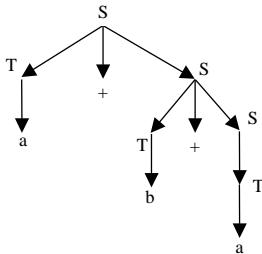
Рассматривая рис.2 б, можно видеть, что последним здесь должен быть вывод  $<\text{цифра}> <\text{цифра}> \Rightarrow 2 <\text{цифра}>$ . Это нам дает  $<\text{цифра}> <\text{цифра}> \Rightarrow 2 <\text{цифра}> \Rightarrow 22$ .

Продолжаем процесс, всегда восстанавливая последний непосредственный вывод, на который указывает концевой куст синтаксического дерева, и затем отсекая этот куст.

Подводя итог, сформулируем следующие положения о синтаксических деревьях:

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

- для каждого синтаксического дерева существует по крайней мере один вывод;
  - для каждого вывода есть соответствующее синтаксическое дерево (но несколько разных выводов могут иметь одно и тоже дерево);
  - куст дерева указывает на непосредственный вывод, в котором имя куста заменяется узлами куста;
  - Следовательно, в грамматике существует правило левой частью которого является имя куста, а правой частью – цепочка из узлов куста;
- концевые узлы дерева образуют выводимую сентенциальную форму;  
 пусть  $U$  – корень поддерева для сентенциальной формы  $w=xuy$ , где  $u$  образует цепочку концевых узлов этого поддерева. Тогда  $u$  – фраза сентенциальной формы  $w$  для  $U$ . Она является простой фразой, если поддерево представлено единственным кустом.



Неоднозначность.

Определение. Предложение грамматики **неоднозначно**, если для его вывода существуют два синтаксических дерева. Грамматика **неоднозначна**, если она допускает неоднозначные предложения, в противном случае она **однозначна**.

Если сентенциальная форма неоднозначна, то она имеет более чем одно синтаксическое дерево и поэтому более чем одну основу. Покажем это на примере, который в тоже время предоставит нам очень полезную грамматику для арифметических выражений. Рассмотрим следующую грамматику арифметических выражений, в которой множество терминальных символов представляют: единственный операнд-  $i$  (в качестве идентификатора), круглые скобки и бинарные операторы  $+$  и  $*$  :

$$\langle E \rangle ::= \langle E \rangle + \langle E \rangle \mid \langle E \rangle * \langle E \rangle \mid ( \langle E \rangle ) \mid i$$

Сентенциальная форма  $\langle E \rangle + \langle E \rangle * \langle E \rangle$  имеет два синтаксических дерева (рис.3) и две основы :  $\langle E \rangle + \langle E \rangle$  и  $\langle E \rangle * \langle E \rangle$ . Так как грамматика неоднозначна, разбор сентенциальной формы можно начать с любой из основ. Таким образом, мы не можем сказать, что выполняется раньше: умножение или

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

сложение.

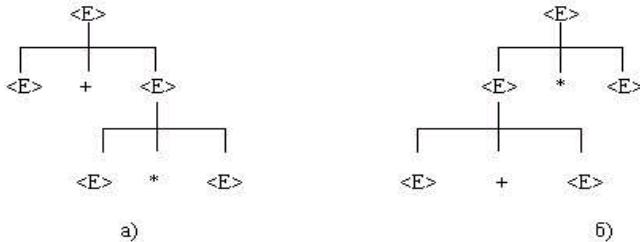


Рис. 3. Два синтаксических дерева для  $\langle E \rangle + \langle E \rangle * \langle E \rangle$ .

Рассмотрим теперь грамматику , состоящую из следующих правил:

**<врж>** ::= **<терм>** | **<врж> + <терм>** | **<врж> - <терм>**

<терм> ::= <множ> | <терм> \* <множ> | <терм> / <множ>

**<МНОЖ>** ::= (**<врж>**) | i

Единственное дерево для выражения  $i + i * i$  показано ниже (рис.4), и, таким образом, в соответствии с этой грамматикой предложение однозначно. В действительности все предложения G3 однозначны. Теперь определим, согласно G3, что в выражении  $i + i * i$  должно выполняться раньше: умножение или сложение. Операндами для +, согласно дереву, являются  $\langle \text{врж} \rangle$ , из которого получается  $i$ , и  $\langle \text{терм} \rangle$ , из которого в свою очередь получается  $i * i$ . Это означает, что умножение должно быть выполнено первым для того, чтобы образовать  $\langle \text{терм} \rangle$  для сложения, следовательно, умножение предшествует сложению.

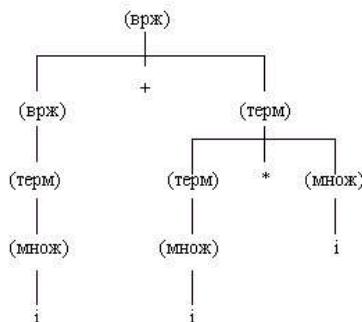


Рис.4. Дерево для выражения  $i + i * i$ .

## Задача синтаксического разбора.

Синтаксический разбор имеет дело с предложениями языка программирования или с сентенциальной формой.

Разбор сентенциальной формы означает построение вывода, и, возможно, синтаксического дерева для нее. Программу разбора

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ  
называют также распознавателем, так как она распознает только предложения рассм. грамматики.

Все алгоритмы разбора, которые здесь рассматриваются, называются алгоритмами разбора слева направо ввиду того, что они обрабатывают сначала самые левые символы рассматриваемой цепочки и продвигаются по цепочке только тогда, когда это необходимо.

Различают две категории алгоритмов разбора: нисходящий (сверху вниз) и восходящий (снизу вверх). Эти термины соответствуют способу построения синтаксических деревьев. При нисходящем разборе дерево строится от корня (начального символа) вниз к концевым узлам.

Основой всякой сентенциальной формы называется самая левая простая фраза.

Определение. Непосредственный вывод  $xUy \rightarrow x$  и  $y$  называется каноническим и записывается  $xUy \rightarrow x$  и  $y$ , если  $y$  содержит только терминалы. Вывод  $w \rightarrow +v$  называется каноническим и записывается  $w \rightarrow +v$ , если каждый непосредственный вывод в нем является каноническим.

Каждое предложение, но не каждая сентенциальная форма имеет канонический вывод.

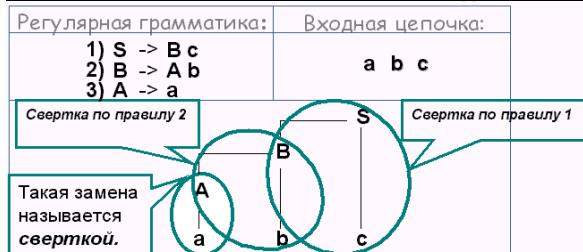
Пример. Рассмотрим в качестве примера сентенциальную форму 3 D грамматики (1). Ее единственным выводом является  $\langle\text{число}\rangle \rightarrow N D \rightarrow D D \rightarrow 3 D$ . И второй, и третий непосредственные выводы не являются каноническими.

Сентенциальная форма, которая имеет канонический вывод, называется канонической сентенциальной формой.

Рассмотрим пример построения дерева снизу-вверх:

- 1) Входная цепочка рассматривается слева направо.
- 2) Если существует правило регулярной грамматики, из которого выводится рассматриваемый символ (последовательность символов) входной цепочки, то данный символ (последовательность символов) заменяется на нетерминал, стоящий в левой части данного правила.
- 3) Если на каком-то шаге не найдется правила, позволяющего произвести свертку, то данное предложение не принадлежит регулярной грамматике.

В данном случае цепочка принадлежит заданной грамматике.



Рассмотрим пример **построения дерева сверху-вниз**:

- 1) Построение дерева начинается с аксиомы S, которая помещается в корень дерева.
- 2) В грамматике выбирается необходимое правило, помеченное рассматриваемым нетерминальным символом. В данном случае на первом шаге выбираем первое правило, т.к. по второму правилу мы не достигнем второго символа входной цепочки b.
- 3) Рассматриваемый нетерминал на дереве разбирается на символы, стоящие в правой части выбранного правила грамматики.
- 4) Построение продолжается до тех пор, пока все концевые вершины дерева (вершины не имеющие в своем подчинении каких либо вершин) не будут обозначены терминальными символами, в противном случае необходимо вернуться ко второму шагу и продолжить построение по другому правилу.
- 5) Если для входной цепочки можно построить дерево вывода, то она принадлежит данной регулярной грамматике.

Рассматриваемая цепочка **a b d** не принадлежит грамматике, т.к. не существует концевой вершины для входного символа d.

**построения дерева сверху - вниз:**



## 5. Сканер. Принципы построения.

**Лексический анализатор (сканер)** представляет ту часть компилятора, которая читает литеры первоначальной исходной программы и строит слова, или иначе символы, исходной программы :

идентификаторы,  
служебные слова,  
целые числа,

одно - или двулитерные разделители, такие как \*, +, \*\*, /\*

Иногда символы называются *атомами* или *лексемами*.

В чистом виде сканер выполняет посторой лексический анализ исходной программы, и поэтому сканер называют также **лексическим анализатором**.

**Результат работы сканера.** Сканер строит внутреннее представление для каждого символа. В большинстве случаев это целое число фиксированной длины (байт, полуслово, слово и т.д.). В других частях компилятора гораздо эффективнее обрабатываются эти целые числа, чем цепочки переменной длины, которыми фактически представляются символы.

Во внутреннем представлении некоторый номер обозначает «идентификатор», другой номер – «целое число». Таким образом, во внутреннем представлении все идентификаторы обозначаются одним и тем же числом. Это естественно, поскольку «идентификатор» для синтаксического анализатора является терминальным символом, и поэтому безразлично, какой идентификатор встречается в каждом конкретном случае. Однако при семантическом анализе приходится иметь дело с самим идентификатором, поэтому его необходимо сохранить. Вопрос исчерпан, если сканером выдается две величины: первая – внутренне представление, вторая – фактический символ или ссылка на него.

Покажем, как проектируется сканер для небольшого и простого языка. Нас интересуют лишь символы языка, и именно их построение является целью лексического анализа.

### **Символы исходного языка.**

Символами в языке являются:

разделители (/ , " , : = , + , \* , ( , ) , ; , // , /\* , \*/),

служебные слова (BEGIN, REAL и END),

идентификаторы (которые не могут быть служебными словами) и целые числа.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

По крайней мере, один пробел должен отделять смежные идентификаторы, целые числа и служебные слова. Ни одного пробела не должно появляться между литерами в слове.

Идентификаторы имеют вид: буква {буква | цифра}

Целые числа имеют вид: цифра {цифра}

В добавлении ко всему сканер должен распознавать и исключать комментарий. Комментарий начинается с двухлитерного символа /\* и заканчивается при первом появлении двухлитерного символа \*/.

Внутреннее представление	Символ	Представление
0	не определен	
1	идентификатор	
2	целое	
3	BEGIN	
4	REAL	
5	END	
6	/	
7	+	
8	,	
9	*	
10	(	
11	)	
12	//	
13	:=	
14	;	

Рис.1. Внутреннее представление символов.

Рассмотрим, например, сегмент программы

BEGIN REAL A,B; A:=B+5/C; /\*КОМЕТАРИЙ\*/ END //

Сканер передаст вызывающей его программе следующее:

Шаг	Результат	Смысл
1	2, "BEGIN"	BEGIN
2	4, "REAL"	REAL
3	1, "A"	идентификатор А
4	8, "	",
5	1, "B"	идентификатор В
6	14, ";"	
7	1, "A"	идентификатор А

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

8	13, “:=”	:=
9	1, “B”	идентификатор В
10	7, “+”	+
11	2, “5”	целое
12	1, “C”	идентификатор С
13	14, “,”	
14	5, “END”	END
15	2, “//”	//

Второстепенные функции сканера: удаление из текста исходной программы комментариев и не несущих смысловой нагрузки пробелов (табуляций и символов новой строки); согласование сообщений об ошибках компиляции и текста исходной программы.

### Принцип построения:

Лексический анализатор обычно реализуется в виде подпрограммы синтаксического анализатора или подпрограммы вызываемой им. При получении запроса на следующий токен лексический анализатор считывает входной поток символов до точной идентификации следующего токена.



*Взаимодействие лексического и синтаксического анализаторов*

Часто токены имеют внутреннее представление в виде целых чисел фиксированной длины. В других частях компилятора гораздо эффективнее обрабатываются эти целые числа, чем цепочки переменной длины.

### Термины:

Лексема – последовательность символов исходной программы.

Токен – 1) терминальный символ грамматики исходного языка; 2) тип лексемы. Лексемы, соответствующие шаблонам токенов, представляются в исходной программе в виде строки символов, которые рассматриваются вместе как лексическая единица.

Шаблон – правило, описывающее набор лексем, которые могут представлять определенный токен в исходной программе. Так,

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ  
шаблон токена **const** представляет собой строку `const`, являющуюся ключевым словом.

Токен	Пример лексем	Неформальное описание шаблона
<b>const</b>	<code>const</code>	<code>const</code>
<b>id</b>	<code>pi, count, d2</code>	Буква, за которой следуют буквы и цифры
<b>num</b>	<code>3.14, 0, 5, 6E2</code>	Любая числовая константа

## 6. Синтаксический анализ. Нисходящий разбор, рекурсивный спуск. Проблемы нисходящего разбора.

### Нисходящий разбор

Большинство известных методов анализа принадлежат одному из двух классов, один из которых объединяет *нисходящие* (*top-down*) алгоритмы, а другой – *восходящие* (*bottom-up*) алгоритмы. Происхождение этих терминов связано с тем, каким образом строятся узлы синтаксического дерева: либо от корня (аксиомы грамматики) к листьям (терминальным символам), либо от листьев к корню.

Нисходящие анализаторы строят вывод, начиная от аксиомы грамматики и заканчивая цепочкой терминальных символов. С нисходящими анализаторами связаны так называемые LL-грамматики, которые обладают следующими свойствами:

- Они могут быть проанализированы без возвратов
- Первая буква L означает, что мы просматриваем входную цепочку слева направо (*left-to-right scan*)
- Вторая буква L означает, что строится левый вывод цепочки (*leftmost derivation*).

Популярность нисходящих анализаторов связана с тем, эффективный нисходящий анализатор достаточно легко может быть построен вручную, например, методом рекурсивного спуска. Кроме того, LL-грамматики легко обобщаются: грамматики, не являющиеся LL-грамматиками, обычно могут быть проанализированы методом рекурсивного спуска с возвратами.

## Рекурсивный спуск

### **Основные положения**

Процессор грамматического разбора, основанный на методе рекурсивного спуска, состоит из отдельных процедур.

- Процедура разбора старается во входном потоке найти подстроку, которая может быть интерпретирована, как **правая часть правила** для нетерминала, связанного с данной процедурой.
- В процессе работы она может вызывать **другие подобные процедуры** или даже **рекурсивно саму себя**, для поиска других нетерминальных символов.
- Если процедура находит **соответствие нетерминальному символу**, то она заканчивает свою работу, и передает в вызывающую ее программу **признак успешного завершения** и устанавливает указатель текущей лексемы на первый символ после распознанной подстроки.

Иначе она заканчивается **признаком неудачи**, или же вызывает процедуру выдачи диагностического сообщения и процедуру восстановления.

#### **Пример:**

Рассмотрим процедуру для оператора **READ**.

Грамматика для оператора:

**READ::=**

- обнаружили лексему **READ**
- следующая лексема должна быть **(**,
- **<id-list>** вызов процедуры,
- если успешно, то лексема должна быть **)**,
- должна быть **;**,
- успех. end.

### **Достоинства**

- простота и скорость написания транслятора;
- соответствие грамматики и анализаторов. Это увеличивает вероятность правильности написания программы.

### **Недостатки**

- большое число вызовов процедур, отсюда относительно медленный анализатор;
- большой объем полученного анализатора;
- данный метод способствует включению в синтаксический анализ процедур семантического анализа и генерации кода. С одной стороны, это хорошо, так как оператор разбирается

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

полностью в одном месте. А с другой стороны, это плохо, так как транслятор становится машинозависимым, и это делает его не мобильным.

### **Условия использования метода рекурсивного спуска**

Метод рекурсивного спуска без возвратов можно использовать только для грамматик, правила которых удовлетворяют следующему условию: первого символа каждого правила должно быть достаточно для того, чтобы определить, какое правило применимо в данном случае. Более точно это условие можно формализовать путем определения множества FIRST.

### **Основные положения**

**Правила факторизации (левой факторизации)** заключаются в том, что общие части грамматических правил выносятся за скобку. Эти правила называют также и "вынесением левого множителя".

Принцип левой факторизации можно выразить в символической форме следующим образом: если грамматика содержит  $n$  правил  $\langle A \rangle \Rightarrow a_1 b_1 \dots \langle A \rangle \Rightarrow a_n b_n$

где  $\langle A \rangle$ - нетерминал, а  $a_i$  и  $b_i$  для  $1 \leq i \leq n$  - цепочки нетерминалов и терминалов, то эти правила можно заменить на следующие  $n+1$  правила:

$$\begin{aligned} &\langle A \rangle \Rightarrow a \langle B \rangle \\ &\langle B \rangle \Rightarrow b_1 \dots \langle B \rangle \Rightarrow b_n \end{aligned}$$

где  $\langle B \rangle$ - нетерминальный символ, не входящий в исходную грамматику.

Грамматика, полученная такой заменой, порождает тот же язык, что и исходная грамматика, и о ней говорят, что она получена левой факторизацией.

Нисходящий разбор — класс алгоритмов грамматического анализа, где правила формальной грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.

Проблемы:

1. Сам по себе метод не исключает возвратов, т.е. для нетерминала, имеющего несколько правых частей, приходится решать, какую альтернативу необходимо выбрать. Пример,  $\langle idlist \rangle = id \mid id, \langle idlist \rangle$ .

Для того, чтобы избавиться от возвратов, в компиляторах в качестве контекста обычно используется следующий "незакрытый" символ исходной программы. Тогда на грамматику налагается следующее требование: если есть альтернативы

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

$x|y|...|z$ , то множества символов, которыми могут начинаться выводимые из  $x,y,...,z$  слова, должны быть попарно различны. То есть если  $x \Rightarrow *Aa$  и  $y \Rightarrow *Bb$  то  $A \neq B$ . если это требование выполнено, можно довольно просто определить, какая из альтернатив  $x,y$  или  $z$  - наша цель. Заметим, что факторизация оказывает здесь большую помощь. Если есть правило  $U ::= xy|xz$ , то преобразование этого правила к виду  $U ::= x(y|z)$  помогает сделать множества первых символов для разных альтернатив непересекающимися.

2. Второй проблемой является левая рекурсия.

Рассмотрим предложение  $\langle idlist \rangle = id \mid \langle idlist \rangle, id$ . Если процедура выберет вторую альтернативу  $\langle idlist \rangle$ ,  $id$ , то она немедленно вызовет рекурсивно саму себя для поиска нетерминала  $\langle idlist \rangle$ . В результате будет зацикливание.

В связи с этим недостатком рекурсивного спуска, в процессе грамматического разбора часто используется замена левой рекурсии на правую.

Правила факторизации (левой факторизации) заключаются в том, что общие части грамматических правил выносятся за скобку. Принцип левой факторизации можно выразить в символической форме следующим образом:

если грамматика содержит  $n$  правил  $\langle A \rangle \Rightarrow \alpha\beta_1 \dots \langle A \rangle \Rightarrow \alpha\beta_n$ ,

где  $\langle A \rangle$  - нетерминал, а  $\alpha$  и  $\beta_i$  для  $1 \leq i \leq n$  - цепочки нетерминалов и терминалов, то эти правила можно заменить на следующие  $n+1$  правила:

$$\langle A \rangle \Rightarrow \alpha\langle B \rangle$$

$$\langle B \rangle \Rightarrow \beta_1 \dots \langle B \rangle \Rightarrow \beta_n$$

где  $\langle B \rangle$  - нетерминальный символ, не входящий в исходную грамматику.

Пример,  $A \Rightarrow BC$

$$A \Rightarrow BCD$$

$$A \Rightarrow axy$$

$$A \Rightarrow axz$$

Пример,  $A \Rightarrow BCM$

$$A \Rightarrow axN$$

$$M \Rightarrow \epsilon$$

$$M \Rightarrow D$$

$$N \Rightarrow y$$

$$N \Rightarrow z$$

Нетерминал называется леворекурсивным, если, применяя к нему одно или более правил, можно вывести цепочку, начинающуюся этим нетерминалом. Правило называется леворекурсивным, если оно используется на первом шаге такого вывода. Пример,  $\langle A \rangle \Rightarrow a\langle B \rangle$ ;  $\langle A \rangle \Rightarrow \langle B \rangle b$ ;  $\langle B \rangle \Rightarrow \langle A \rangle c$ ;  $\langle B \rangle \Rightarrow d$

Правило называется **самолеворекурсивным**, если его первый символ правой части и левая часть – это один и тот же символ.

Пример,  $\langle S \rangle \Rightarrow \langle S \rangle a; \langle S \rangle \Rightarrow b$

Грамматику с левой рекурсивностью всегда можно переделать в грамматику с правой рекурсией. Основная идея при этом состоит в том, чтобы смотреть на рекурсивный нетерминал как на порождающий некоторую цепочку, за которой следует список из одного или более элементов.

Пример, грамматика:  $\langle idlist \rangle \Rightarrow id \mid \langle idlist \rangle , id$  заменяется на  
 $\langle idlist \rangle \Rightarrow id \langle A \rangle$   
 $\langle A \rangle \Rightarrow \varepsilon \mid , id \langle A \rangle$

## **7. LL(K)-грамматики. Направляющие символы. Идея разбора.**

LL(k) – грамматики, основанные на принципе выборе одной альтернативы из множества возможных на основе нескольких очередных символов в цепочке. Грамматика обладает свойством LL(k),  $k > 0$ , если на каждом шаге вывода для однозначного выбора очередной альтернативы МП-автомату достаточно знать символ на верхушке стека и рассмотреть первые k символов от текущего положениячитывающей головки во входной цепочке символов. Грамматика называется LL(k)-грамматикой, если она обладает св-вом LL(k) для нек.  $k > 0$ .

Название «LL(k)» несет определенный смысл. Первая литера «L» происходит от слова «left» и означает, что входная цепочка символов читается в направлении слева направо. Вторая литера «L» также происходит от слова «left» и означает, что при работе распознавателя используетсялевосторонний вывод. Вместо «k» в названии класса грамматики стоит некоторое число, которое показывает, сколько символов надо рассмотреть, чтобы однозначно выбрать одну из множества альтернатив. Так, существуют LL(1)-грамматики, LL(2)-грамматики и другие классы.

Для LL(k)-грамматик известны следующие полезные свойства:

- всякая LL(k)-грамматика для любого  $k > 0$  является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LL(k)-граммикой для строго определенного числа k.

Кроме того, известно, что все грамматики, допускающие разбор по методу рекурсивного спуска, являются подклассом LL(1)-граммик. То есть любая грамматика, допускающая разбор по методу рекурсивного спуска, является LL(1)-граммикой (но не наоборот!).

Есть, однако, неразрешимые проблемы для произвольных КС-граммик:

- не существует алгоритма, который бы мог проверить, является ли заданная КС-грамматика LL( $k$ )-грамм. для нек. произвольного числа  $k$ ;
- не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику к виду LL( $k$ )-грамматики для некоторого  $k$  (или доказать, что преобразование невозможно).

Множество символов, позволяющих сделать правильный выбор правила, называется **множеством выбора** или **множеством направляющих символов**. Направляющие символы являются единственно допустимыми символами на каждом шаге анализа.

### Идея разбора

- Для каждого нетерминального символа, находящегося в левой части строится строка таблицы, в которую вносится имя нетерминала, множество направляющих символов для этого нетерминала и указатель на группу строк соответствующей правой части правила.
- Если нетерминал имеет несколько альтернатив, то для него строятся соответствующие альтернативам строки, в каждую вносится имя нетерминала, множество направляющих символов для этого нетерминала и альтернативы, указатель на группу строк соответствующей альтернативы.
- Каждому элементу из правой части отводится строка таблиц:

Если символ терминальный, то в строку вносится символ, и указатель на следующую строку таблицы, если символ не последний в альтернативе, иначе указателю присваивается нулевое значение.

Для нетерминала правой части строится строка таблицы, в которую вносится имя нетерминала, множество направляющих символов для этого нетерминала и указатель на строку, помеченную этим нетерминалом для левой части правила.

Для пустого символа тоже создается строка, в которую вносится пустой символ, множество направляющих символов и указателю присваивается нулевое значение.

**8. Построение LL(1)таблицы разбора.Разбор по LL(1)таблице.Проблемы LL(1)-разбора. Достоинство и недостатки метода.**

LL( $k$ ) – грамматики, основанные на принципе выборе одной альтернативы из множества возможных на основе нескольких очередных символов в цепочке.

Название «LL( $k$ )» несет определенный смысл. Первая литера «L» происходит от слова «left» и означает, что входная цепочка символов читается в направлении слева направо. Вторая литера «L» также происходит от слова «left» и означает, что при работе распознавателя используется левосторонний вывод. Число 1 показывает, сколько символов надо рассмотреть, чтобы однозначно выбрать одну из множества альтернатив.

**Основные положения:**

- Для каждого нетерминального символа, находящегося в левой части строится строка таблицы, в которую вносится имя нетерминала, множество направляющих символов для этого нетерминала и указатель на группу строк соответствующей правой части правила.
- Если нетерминал имеет несколько альтернатив, то для него строятся соответствующие альтернативам строки, в каждую вносится имя нетерминала, множество направляющих символов для этого нетерминала и альтернативы, указатель на группу строк соответствующей альтернативы.
- Каждому элементу из правой части отводится строка таблицы;
- Если символ терминальный, то в строку вносится символ, и указатель на следующую строку таблицы, если символ не последний в альтернативе, иначе указателю присваивается нулевое значение.
- Для нетерминала правой части строится строка таблицы, в которую вносится имя нетерминала, множество направляющих символов для этого нетерминала и указатель на строку, помеченную этим нетерминалом для левой части правила.
- Для пустого символа тоже создается строка, в которую вносится пустой символ, множество направляющих символов и указателю присваивается нулевое значение.

Пример построения LL(1) :

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

```

< PROG > ⇒ PROG id < VAR > begin < listst > END
< VAR > ⇒ VAR < idlist >
< idlist > ⇒ id < A >
    < A > ⇒ , < idlist > | E
    < listst > ⇒ st < B >
    < B > ⇒ < listst > | E
  
```



1	PROG	2	2
	id	4	3
	<VAR>	VAR	4
	<listst>	st	19
		END	NULL



№	Направляющее множество	Старт	Переход	Стек	Ошибка	Конечное состояние
1	PROG	нет	2	нет	да	нет
2	PROG	да	3	нет	да	нет
3	id	да	4	нет	да	нет
4	var	нет	8	да	да	нет
5	begin	да	6	нет	да	нет
6	st	нет	19	да	да	нет
7	end	нет	-	нет	да	да
8	var	нет	9	нет	да	нет
9	var	да	10	нет	да	нет
10	id	нет	11	нет	да	нет
11	id	нет	12	нет	да	нет
12	id	да	13	нет	да	нет
13	begin ,	нет	14	нет	да	нет
14	,	нет	16	нет	нет	нет
15	begin	нет	18	нет	да	нет
16	,	да	17	нет	да	нет
17	id	нет	11	нет	да	нет
18	begin	нет	- ( из стека)	нет	да	нет
19	st	нет	20	нет	да	нет
20	st	да	21	нет	да	нет
21	st, end	нет	22	нет	да	нет
22	st	нет	24	нет	нет	нет
23	end	нет	25	нет	да	нет
24	st, end	нет	19	нет	да	нет
25	end	нет	- ( из стека)	нет	да	нет

LL(k) – грамматики, основанные на принципе выборе одной альтернативы из множества возможных на основе нескольких очередных символов в цепочке.

Название «LL(k)» несет определенный смысл. Первая литера «L» происходит от слова «left» и означает, что входная цепочка символов читается в направлении слева направо. Вторая литера «L» также происходит от слова «left» и означает, что при работе распознавателя используется левосторонний вывод. Число 1 показывает, сколько символов надо рассмотреть, чтобы однозначно выбрать одну из множества альтернатив.

**Для левой части правила:**

- проверка входного символа на попадание в направляющее множество данного нетерминала и переход по указателям в случае, если входной символ совпадает с одним из направляющих символов. В противном случае – ошибка (error).

**Для терминала:**

- Проверка на совпадение входного символа с терминальным, если нет, то – ошибка (error);
- Сдвиг по входной цепочке;
- Переход по указателю, если указатель имеет нулевое значение, то переход по адресу, находящемуся в верхушке стека.

**Для нетерминала:**

- Проверка входного символа на совпадение с одним из символов направляющего множества;
- В стек заносится адрес следующей строки, соответствующей следующему за нетерминалом символу правой части правила. Если такого символа нет, т.е. символ последний во входной цепочке, то в стек ничего не заносится.
- Переход по указателю.

**Для пустого символа:**

- проверка входного символа на совпадение с одним из символов направляющего множества;
- переход по указателю, находящемуся в верхушке стека.

**Для левой части правила, имеющей несколько альтернатив:**

- входной символ проверяется на вхождение в направляющее множество сначала первой альтернативы;
- если входной символ не входит в это направляющее множество, то переход на следующую строчку, соответствующей следующей альтернативе и т.д.;
- если входной символ не входит в направляющее множество последней альтернативы, тогда ошибка (error);
- если входной символ совпал с одним из символов направляющего множества одной из альтернатив, то переход по указателю соответствующей строки.

**Модификация LL(1)-таблицы:**

- Модификация LL(1)-таблицы производится в ходе ее обработки.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

- Основные поля новой таблицы: номер строки, направляющее множество (для каждого символа), сдвиг по входной цепочке (есть он или нет), переход (номер строки, на который происходит переход), стек (вносится адрес строки символа), ошибка (необходима для определения перехода по альтернативе), конечное состояние (конец разбираемой цепочки).

№	направляющее множество	сдвиг	переход	стек	ошибка	конечное состояние
1	PROG	нет	2	нет	да	нет
2	PROG	да	3	нет	да	нет

Алгоритм заканчивается успешно, если входная цепочка разобрана и признана принадлежащей разбираемому языку, если стек пустой, входной символ соответствует концу цепочки и разбор заканчивается на строке, соответствующей конечному состоянию. В противном случае разбор заканчивается с сообщением об ошибке.

Пример разбора для таблицы из прошлого вопроса:

program work var A,B,C begin st st end

1. Считывается program, проверяется и переход к 2.
2. program принимается, переход к 3.
3. Считывается work(id), принимается, переход к 4.
4. Считывается var, переход к 8.
8. Проверяется var, переход к 9.
9. Принимается var, переход к 10.
10. Считывается A(id), переход к 11.
11. Проверяется id, переход к 12
12. Принимается id, переход к 13
13. Считывается ",", переход к 14
14. Выбор по ",", переход к 15-16
16. Принимаем ",", переход к 18
18. Считываем B(id), переход к 11
11. Проверяется id, переход к 12
12. Принимается id, переход к 13
13. Считывается ",", переход к 14, по "," к 15-17
17. Принимаем ",", переход к 18
18. Считываем C(id), переход к 11
11. Проверяется id, переход к 12
12. Принимается id, переход к 13
13. Считывается begin, переход к 14, выбор 16
16. Переход по адресу из стека (5)
5. begin принимается, переход к 6

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

6. Считывается st, переход к 19, в стек 7
19. Opr, переход к 20.
20. Принимается st, переход к 21
21. Считывается st, переход к 22
22. по st, переход к 25
25. переход к 19
19. переход к 20
20. Принимается st, переход к 21
21. Считывается end, переход к 22
22. по end, переход к 24
23. переход по адресу стека
25. Принимаем end, в стеке 0, в цепочке ~

Проблемы LL(1) разбора: см вопрос 9: “Проблемы нисходящего разбора”.

### Достоинства распознавателей, основанных на LL(1) грамматике

- в отличие от рекурсивного спуска в методе нет возвратов, это детерминированный метод;
- время разбора пропорционально длине входной программы;
- имеются хорошие диагностические характеристики, и существует возможность исправления ошибок, так как распознавание ведется по одному символу;
- таблица разбора меньше, чем соответствующая таблица разбора в других методах.

### Недостаток распознавателей, основанных на LL(1) грамматиках

- не все классы языков описываются LL(1) грамматиками.

## **9. Восходящий разбор. Проблемы. Общий метод разбора.LR(K)-грамматики. Идея разбора.**

LR-грамматики основаны на восходящем методе разбора, то есть разборе снизу - вверх, при котором промежуточные выводы перемещаются по дереву по направлению к корню. Разбор идет справа налево.

При **восходящем разборе** дерево начинает строиться от терминальных листьев путем подстановки правил, применимых к входной цепочке, в общем случае, в произвольном порядке. На следующем шаге новые узлы полученных поддеревьев используются как листья во вновь применяемых правилах. Процесс построения дерева разбора завершается, когда все символы входной цепочки будут являться листьями дерева, корнем которого окажется начальный нетерминал. Если, в результате полного перебора всех возможных правил, мы не сможем построить требуемое дерево разбора, то рассматриваемая входная цепочка не принадлежит данному языку.

При использовании этого метода в текущей сентенциальной форме повторяется поиск основы (самой левой простой фразы), которая в соответствии с правилом  $U ::=$  и приводится к нетерминалу  $U$ . При восходящем разборе возникает проблема – как найти основу и выяснить к какому нетерминалу нужно ее приводить и как найти основу, если задана сентенциальная форма (последовательность символов (терминалов и нетерминалов), выводимых из аксиомы)  $x$ . Хотелось бы, двигаясь слева направо и рассматривая только два соседних символа одновременно, суметь определить, нашли ли мы хвост основы. А затем, продвигаясь назад к левому концу сентенциальной формы, найти голову основы, принимая каждый раз решение только по двум соседним символам. То есть мы сталкиваемся с такой проблемой: если задана цепочка ...RS..., то всегда ли  $R$  является хвостом основы, или  $RS$  вместе могут встретиться в основе, или возможны другие варианты? Хотелось бы, не приступая еще к разбору, исследовать грамматику и принять решение относительно каждой пары символов  $R$  и  $S$ .

Рассмотрим два символа  $R$  и  $S$  из словаря  $V$  грамматики  $G$ . Предположим, что существует (каноническая) сентенциальная форма ...RS... На некотором этапе канонического разбора либо  $R$ , либо  $S$  (либо оба символа одновременно) должны войти в основу. При этом возникают следующие три возможности:

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

R- часть основы, а S-нет (рис.1 а). Эту ситуацию мы записываем, как  $R \succ S$  и говорим, что R больше S или что R предшествует S, поскольку символ R будет редуцирован раньше, чем S. Заметим, что R должен быть последним (хвостовым) символом в правой части некоторого правила  $U ::= ...R$ . Заметим, что поскольку основа находится слева от S, S должен быть терминалом.

Оба символа R и S входят в основу (рис.1 б). Запишем это как  $R \sim S$ . У них одинаковое значение предшествования, и они должны редуцироваться одновременно. Очевидно в грамматике должно быть правило  $U ::= ...RS...$

S- часть основы, а R- нет (рис.1 с). Отношение между ними записывается как  $R < S$ ; можно говорить, что R меньше, чем S. Символ S должен быть первым (головным) в правой части некоторого правила  $U ::= S...$

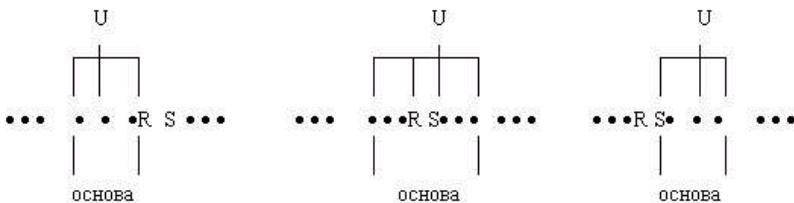


Рис.1. Примеры отношений предшествования.

Если (канонической) сентенциальной формы ...RS... не существует, мы считаем, что между упорядоченной парой символов (R, S) не определено никакое отношение. Заметим, что ни одно из трех определенных выше отношений предшествования  $<$  ,  $\sim$  и  $\succ$  не является симметричным. Например, из  $R < S$  вовсе не следует  $S \succ R$ .

Рассмотрим в качестве примера грамматику G5(Z)

$$(1) \quad \begin{aligned} Z &::= bMb \\ M &::= (L \mid a \\ L &::= Ma) \end{aligned}$$

Языку L(G5) принадлежат такие цепочки, как bab, b(aa)b, b((aa)a)b, b(((aa)a)a)b. В каждом столбце таблицы1 показана сентенциальная форма, ее синтаксическое дерево, основа дерева и отношения, которые можно из него получить.

**Таблица 1.**

Сентенциальная форма	b a b	b ( L b )	b ( M a ) b
Синтаксическое дерево	<pre>       Z               b   M   b                       a       b     </pre>	<pre>       Z               b   M   b                       (     L                       L     b     </pre>	<pre>       Z               b   M   b                       (     L                       M   a   b                       a   b     </pre>
Основа	a	( L	M a )
Отношения, определенные деревом	b <• a a •> b	b <• ( L ( L > b	( <• M M > a a > ) ) •> b

На рис.2 представлена матрица предшествования для грамматики G5- матрица, в которой указываются все отношения предшествования. Элемент этой матрицы  $B_{[i, j]}$  содержит отношение между парой символов  $(S_i, S_j)$ . Пустой элемент матрицы свидетельствует о том, что между соответствующими двумя символами отношение предшествования не определено.

	Z	b	M	L	a	(	)
Z							
b			≡		<•	<•	
M		≡			≡		
L		•>			•>		
a		•>			•>		≡
(			<•	≡	<•	<•	
)		•>			•>		

Рис.2. Матрица предшествования для грамматики G5.

Как же воспользоваться отношениями предшествования при разборе предложений? Если между какой – либо парой символов  $(R, S)$  определено более чем одно отношение, они бесполезны. Если же между любой парой символов определено не более одного отношения, отношения предшествования позволяют найти основу любой сентенциальной формы. Тогда можно сказать, что основой любой сентенциальной формы  $S_1\dots S_n$  является самая левая подцепочка  $S_j\dots S_i$ , такая, что

$$(2) \quad S_{j-1} \cdot S_j \\ S_{j-1} \cdot S_{j+1} \cdot S_{j+2} \cdot \dots \cdot S_i \\ S_i > S_{i+1}$$

Из определения отношений вытекает, что основа  $S_1 \dots S_n$  удовлетворяет (2). Не столь очевидным представляется тот факт, что самая левая подцепочка, удовлетворяющая (2), является основой.

**Пример.** Проведем канонический разбор предложения в (aa) в грамматики G5 с использованием матрицы предшествования, изображенной на рис.2. Рисунок 3 иллюстрирует разбор. На каждом шаге показаны сентенциальная форма и отношения, определенные между символами в соответствии с рис.2.

Шаг	Сентенциальная форма	Основа	Привести основу к	Построенный непосредственный вывод
1	$b \ ( \ a \ a \ ) \ b$ $\Leftrightarrow \Leftrightarrow \Rightarrow \equiv \Rightarrow$	a	M	$b(Ma)b \Rightarrow b(aa)b$
2	$b \ ( \ M \ a \ ) \ b$ $\Leftrightarrow \Leftrightarrow \equiv \equiv \Rightarrow$	M a )	L	$b(Lb \Rightarrow b(Ma)b$
3	B ( L b $\Leftrightarrow \equiv \Rightarrow$	( L	M	$BMb \Rightarrow b(Lb$
4	B M b $\equiv \equiv$	b M b	Z	$Z \Rightarrow bMb$

### Рис.3. разбор сентенциальной формы b (aa) b.

**Определение.** Если задана грамматика  $G$ , то отношения предшествования между символами из словаря  $V$  определяются следующим образом:

$R \cdot S$  тогда и только тогда, когда в  $G$  есть правило  $U ::= \dots RS\dots$

R  $\leftarrow$  S тогда и только тогда, когда существует правило  $U \vdash_{\text{FIRST}^+} RV$ , такое, что справедливо отношение  $V \text{ FIRST}^+ S$ .

$R \succ S$  тогда и только тогда, когда  $S$ -терминал, и существует правило  $U ::= \dots V W \dots$ , такое., что справедливы соотношения  $V \vdash_{LAST^+} R$  и  $W \vdash_{FIRST^-} S$

Определим еще два отношения:

$R \leq S$  тогда и только тогда, когда  $R : S$  или  $R \leq S$ .

$R \geq S$  тогда и только тогда, когда  $R \cdot S$  или  $R \geq S$ .

**Определение.** Грамматику G называют грамматикой (простого) предшествования или просто грамматикой предшествования, если:

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

- между любыми двумя символами из словаря определено не более чем одно отношение;
- ни у каких двух продукции нет одинаковых правых частей.

Если G- грамматика предшествования, то отношения позволяют найти основу любой сентенциальной формы; при этом соблюдение второго условия гарантирует нам то, что основу можно привести к единственному нетерминалу.

### Теорема.

Грамматика предшествования однозначна. Более того, единственная основа любой сентенциальной формы  $S_1\dots S_n$ - это самая левая подцепочка  $S_j\dots S_i$ , такая, что

$$S_{j-1} \cdot S_j : S_{j+1} \dots : S_{i-1} \cdot S_i + 1$$

### **При восходящем методе разбора:**

- 1) все правила грамматики обязательно нумеруются;
- 2) рассматривается пополненная грамматика (то есть грамматика, для которой из аксиомы выводится только одна альтернатива).

### Пример:

real A,B,C

- 1)  $S \rightarrow \text{real} <\text{idlist}>$
- 2)  $<\text{idlist}> \rightarrow <\text{id}>, <\text{idlist}>$
- 3)  $<\text{idlist}> \rightarrow <\text{id}>$
- 4)  $<\text{id}> \rightarrow A \mid B \mid C$

**1 шаг:** В стек помещается первый символ предложения **real**.

**2 шаг:** В стек помещается второй символ **A**.

**3 шаг:** Происходит замена символа **A** на нетерминал **<id>** по 4 правилу.

**4 шаг:** В стек помещается следующий символ «,».

**5 шаг:** В стек помещается символ **B**.

**6 шаг:** Символ **B** по правилу 4 заменяется на нетерминал **<id>**.

**7 шаг:** В стек помещается следующий символ.

**8 шаг:** в стек заносится символ **C**.

**9 шаг:** Символ **C** заменяется на нетерминал.

**10 шаг:** В стеке происходит замена нетерминала **<id>** на **<idlist>** по правилу 3.

**11 шаг:** В стеке происходит замена верхних символов по правилу 2.

**12 шаг:** Применяем правило 2.

**13 шаг:** Используем правило 1. В стеке осталась аксиома **S**.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

В данном методе восходящего разбора использовались две операции:

- 1 операция – это сдвиг.

Она состоит в том, что в стек заносится символ входной цепочки, и параллельно с этим происходит сдвиг по входной цепочке.

- 2 операция – операция свертки.

Она заключается в следующем: из верхушки стека достаются символы, которые сворачиваются по некоторому правилу.

*Основной проблемой восходящего разбора является детерминированность, то есть на каждом шаге алгоритма хотелось бы знать, что производит сдвиг или свертку и по какому правилу. Говорят о конфликте сдвиг-свертка, если в некоторый момент разбора для одной цепочки допустимы и сдвиг, и свертка. И говорят о конфликте свертка-свертка, если в некоторый момент разбора допустимы свертки по различным правилам.*

### Проблемы:

При восходящем разборе, при построении дерева снизу-вверх на каждом шаге редуцируется (прием сведения сложного к простому) **основа (самая левая простая фраза)** текущей сентенциальной формы и поэтому цепочка справа от основы всегда содержит только терминальные символы. Проблема восходящего разбора состоит в следующем: т.к. при восходящем разборе на каждом шаге редуцируется основа найти основу сентенциальной формы и то, к чему она должна приводиться становится сложно.

Если в процессе LR-разбора принять детерминированное решение о сдвиге/свертке удается, рассматривая только цепочку  $x$  и первые  $k$  символов непросмотренной части входной цепочки  $u$  (эти  $k$  символов называют аванцепочкой), говорят, что грамматика обладает LR( $k$ )-свойством.

-S--  
/ \ \/  
/-x^ \/  
--w-+-u--

Название «LR( $k$ ) - грамматика» указывает на то, что для нее существует МП автомат (автомат с магазинной памятью), который

- начинает просмотр слева направо (Left),

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

- распознает правило, когда добирается до самого правого (Rightmost) символа, выводимого из этого правила
- и может обнаружить любую основу просмотром k-го количества символов, расположенных правее последнего входного символа, выводимого из основы.

На практике чаще всего k=1 (LR(1) грамматика) или k=0 (LR(0) грамматика).

Используются также промежуточные между LR(0) и LR(1) методы, известные под названиями SLR(1) и LALR

При разборе строки **LR(0)-языка** можно вообще не использовать аванцепочку - выбор между сдвигом и сверткой делается на основании цепочки x (см.картинку). Так как в процессе разбора она изменяется только с правого конца, ее называют стеком. Будем считать, что в грамматике нет бесполезных символов и начальный символ не встречается в правых частях правил - тогда свертка к начальному символу сигнализирует об успешном завершении разбора. В LR(0) грамматике для всех состояний стека в процессе LR-вывода нет конфликтов сдвиг-свертка или свертка-свертка.

**SLR(1) и LALR(1) грамматики.** В основе этих двух методов лежит одна и та же идея. Допустим, у нас есть множество канонических LR(0)-состояний грамматики. Если это множество не содержит конфликтов, то можно применить LR(0)-парсер. Иначе разрешаем возникшие конфликты, рассматривая односимвольную аванцепочку, т.е. строится LR(1) парсер с множеством LR(0)-состояний.

Отличие LR(k) грамматик от LL(k) состоит в том, что LL(k) грамматики по считанному символу определяют правую часть, а LR(k) грамматики считывают в стек всю правую часть и еще один символ. LR грамматики шире LL грамматик. Можно решить обладает ли грамматика свойством LR(k) для заданного k. Но нельзя решить существует ли k, для которого заданная грамматика, будет LR(k) грамматикой.

LR грамматики основаны на восходящем методе разбора, то есть разборе снизу-вверх, при котором промежуточные выводы перемещаются по дереву по направлению к корню.

При восходящем методе разбора все правила грамматики обязательно нумеруются. Рассматривается, как правило, пополненная грамматика, то есть грамматика, для которой из аксиомы выводится только одна альтернатива.

Суть всех методов разбора заключается в следующем: сначала в стек символ за символом помещают входную цепочку, до тех пор,

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

пока в стеке не будет находиться некоторое правило. Символы этого правила извлекаются из стека, заменяются на соответствующий нетерминал, затем процесс продолжается до тех пор, пока в стеке не окажутся аксиомы, а во входной цепочке не останется символ конца.

**Рассмотрим грамматику со следующими правилами:**

1.  $S \Rightarrow \text{real} <\!\! \text{idlist} \!\!>$
2.  $<\!\! \text{idlist} \!\!> \Rightarrow \text{id}, <\!\! \text{idlist} \!\!>$
3.  $<\!\! \text{idlist} \!\!> \Rightarrow <\!\! \text{id} \!\!>$
4.  $<\!\! \text{id} \!\!> \Rightarrow A \mid B \mid C$

В качестве примера разберем следующее предложение  $\text{real}$   $A, B, C \perp$ , где  $\perp$  обозначает конец цепочки. Восходящий разбор этого предложения будет состоять из 13 шагов:

**1 шаг.** В стек помещается первый символ предложения  $\text{real}$ .



**2 шаг.** В стек помещается второй символ  $A$ .



**3 шаг.** Происходит замена символа  $A$  на нетерминал  $<\!\! \text{id} \!\!>$  по

4 правилу.



**4 шаг.** В стек помещается следующий символ  $,$ .



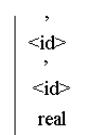
**5 шаг.** В стек помещается символ  $B$ .



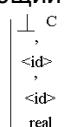
**6 шаг.** Символ  $B$  по правилу 4 заменяется на нетерминал  $<\!\! \text{id} \!\!>$ .



**7 шаг.** В стек помещается следующий символ.



**8 шаг.** В стек заносится символ  $C$ .



<id>
,
<id>
,
<id>
real

**9 шаг.** Символ C заменяется на нетерминал.

**10 шаг.** В стеке происходит замена нетерминала  $\langle id \rangle$  на  $\langle idlist \rangle$

<idlist>
,
<id>
,
<id>
real

по правилу 3.

**11 шаг.** В стеке происходит замена верхних символов по правилу

2.

<idlist>
,
<id>
real

<idlist>
real

**12 шаг.** Применяем правило 2.

**13 шаг.** Используем правило 1. В стеке осталась аксиома S.

S
---

Разбор предложения закончен.

В данном методе восходящего разбора использовались две операции. Первая операция – это *сдвиг*. Она состоит в том, что в стек заносится символ входной цепочки, и параллельно с этим происходит сдвиг по входной цепочке. Вторая операция – операция *свертки*. Она заключается в следующем: из верхушки стека достаются символы, которые сворачиваются по некоторому правилу.

Будем считать, что в грамматике нет бесполезных символов, и начальный символ не встречается в правых частях правил – тогда свертка к начальному символу сигнализирует об успешном завершении разбора. Говорят о конфликте сдвиг-свертка, если для одной цепочки и допустимы и сдвиг, и свертка. Говорят о конфликте свертка-свертка, если допустимы свертки по различным правилам.

Основной проблемой восходящего разбора является детерминированность, то есть на каждом шаге алгоритма хотелось бы знать, что производить сдвиг или свертку и по какому правилу. Все методы восходящего разбора отличаются

друг от друга тем, как они определяют, что надо делать на каждом шаге: сдвиг или свертку.

## 10. Построение таблиц разбора(LR(0), SLR(1),LALR(1)).

LR грамматики основаны на восходящем методе разбора, то есть разборе снизу-вверх, при котором промежуточные выводы перемещаются по дереву по направлению к корню. При восходящем методе разбора все правила грамматики обязательно нумеруются. Рассматривается, как правило, пополненная грамматика, то есть грамматика, для которой из аксиомы выводится только одна альтернатива.

Суть всех методов разбора заключается в следующем: сначала в стек символ за символом помещают входную цепочку, до тех пор, пока в стеке не будет находиться некоторое правило. Символы этого правила извлекаются из стека, заменяются на соответствующий нетерминал, затем процесс продолжается до тех пор, пока в стеке не окажутся аксиомы, а во входной цепочке не останется символ конца.

### **LR(0)**

Простейшим случаем LR( $k$ )-грамматик являются LR(0)-грамматики. При  $k = 0$  распознающий расширенный МП-автомат совсем не принимает во внимание текущий символ, обозреваемый егочитывающей головкой. Решение о выполняемом действии принимается только на основании содержимого стека автомата. При этом не должно возникать конфликтов между выполняемым действием (сдвиг или свертка), а также между различными вариантами при выполнении свертки. Управляющая таблица для LR(0)-грамматики строится на основании понятия «левых контекстов» для нетерминальных символов: очевидно, что после выполнения свертки для нетерминального символа  $A$  в стеке МП-автомата ниже этого символа будут располагаться только те символы, которые могут встречаться в цепочке вывода слева от  $A$ . Эти символы и составляют «левый контекст» для  $A$ . Поскольку выбор между сдвигом или сверткой, а также между типом свертки в LR(0)-грамматиках выполняется только на основании содержимого стека, то LR(0)-грамматика должна допускать однозначный выбор на основе левого контекста для каждого символа.

Грамматика называется  $LR(0)$ , если для всех состояний стека в процессе LR-вывода нет конфликтов сдвиг-свертка или свертка-свертка.

LR(0)-анализатор принимает решение о своих действиях только на основании содержимого магазина, не учитывая символы входной цепочки.

**Определение.** Пусть  $G = (VT, VN, P, S)$  – КС-грамматика.

*Положенной грамматикой* (augmented grammar) будем называть грамматику  $G' = (VT, VN + \{S'\}, P + \{S' \rightarrow S\}, S')$ , где  $S'$  – нетерминал, непринадлежащий множеству  $N$ .

В начале работы магазин пуст (на самом деле, на вершине магазина находится маркер конца  $\$$ ), и указатель входной цепочки находится перед ее первым символом. Этому состоянию соответствует ситуация  $[S' \rightarrow .S]$ . Значит, входная цепочка может начинаться с любого терминального символа, с которого начинается правая часть любого правила с левой частью  $S$ .

**Построение таблицы разбора.**

1. все правила грамматики пронумерованы;
2. рассматривается положенная грамматика.

*Грамматическим вхождением* символа называется позиция этого символа в правой части правила. Грамматические вхождения обозначаются индексами: номер правила, номер позиции символа в этом правиле, например:

$A \rightarrow a b a <B> c e c$

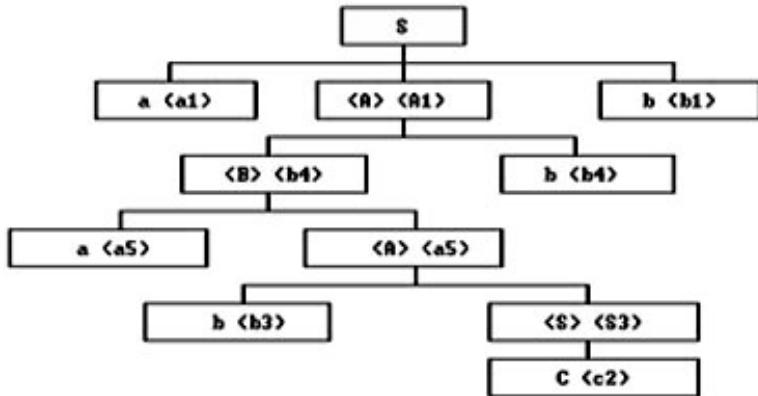
a51, a53, b52

**Основная идея.**

При построении дерева разбора предложения языка, всегда однозначно можно установить соответствие вершинам дерева и грамматическим вхождениям. Например, в грамматике

1.  $S \Rightarrow a <A> b$
2.  $S \Rightarrow c$
3.  $A \Rightarrow b <S>$
4.  $A \Rightarrow <B> b$
5.  $a \Rightarrow B <A>$
6.  $c \Rightarrow B$

Разбор предложения **aabcbb** можно представить следующим деревом. Каждой вершине в соответствие поставлено грамматическое вхождение.



Таким образом, при грамматическом разборе предложения языка, зная соответствие очередного рассматриваемого символа предложения грамматическому вхождению (т.е. правилу и позиции в этом правиле) можно определить, какой символ может быть следующим в предложении и соответствующее грамматическое вхождение.

Поэтому можно определить МП-автомат и соответствующую таблицу разбора следующим образом: строки (состояния автомата) - помечаются грамматическими вхождениями а столбцы - символами алфавита  $V$  (терминальными и нетерминальными).

На пересечении строки грамматического вхождения и столбца допустимого входного символа ставится переход к следующему грамматическому вхождению или, если грамматическое вхождение соответствует последнему символу правила, то в строке ставится свертка.

$Q \leq Y_j$  грам. вход. тогда и только тогда, когда:

1. существует  $Z_j$  грам. вход. : найдется такое  $U ::= ... Q Z_i$  и  $Z_i FIST^* Y_j$
2. если  $Q = S_0$  то  $S_0 FIST^* Y_j$

#### *Алгоритм построения LR(0) таблицы разбора.*

Строим таблицу: столбцы соответствуют символам грамматики ( $V_t, V_n$ ), строки - грамматическим вхождениям.

Причем, если грамматические вхождения неразличимы, т.е.

$A \leq Y_j$  и  $A \leq Y_i$ , то строится одна строка таблицы и помечается  $(Y_i, Y_j)$ .

Таблица строится и заполняется соответствующим образом

Первая строка таблицы соответствует

- a) пустому стеку

б)началу разбора, т.е. входной символ - это первый возможный символ входной цепочки.

	<b>S</b>		
<b>1</b>	<b>OK</b>		<b>Yj</b>

**S** —> .....

**S FIRST \* Yj**

Элементы - это грамматические вхождения. Причем S0 - соответствует входному символу S. Следующие строки таблицы создаются таким образом - для каждого грамматического вхождения взятого из предыдущей строки создается строка помеченная этим грамматическим вхождением.

Элементы строки при построении таблицы заполняются таким образом. Пусть Q - грамматическое вхождение, которым помечена строка, Z - входной символ.

а) если Q не самый правый символ любого правила, то для любого Z, такого что

Q <= Z - клетки строки таблицы заполняются соответствующими грамматическими вхождениями этих символов.

б) если Q - самый правый символ правила L, эта ситуация соответствует свертке по правилу L. Во все клетки строки ставится свертка по правилу L.

### ПРЕОБРАЗОВАНИЕ ТАБЛИЦЫ

Строки таблицы пронумеровать и заменить переход по грамматическому вхождению в клетках таблицы номером соответствующей строки.

Если таблица строится по этому алгоритму, то грамматика является LR(0).

Название "LR(0) грамматика" указывает на то, что просмотр предложения происходит слева направо, любую основу можно обнаружить без просмотра входных символов, расположенных правее последнего входного символа основы.

В LR(0)грамматиках свертка помещается во все клетки строки, соответствующей окончанию правила.

	a	b	c	s	A	B
нач	a11		c2	s0		
s0						
a11	a5	b3	c6		A1	B4
A1		b1				
b1	R1	R1	R1	R1	R1	R1
c2	R2	R2	R2	R2	R2	R2
b3	a1		C2	S3		
S3	R3	R3	R3	R3	R3	R3
B4		b4				
b4	R4	R4	R4	R4	R4	R4
a5	a5	b3	c6		A5	B4
A5	R5	R5	R5	R5	R5	R5
C6	R6	R6	R6	R6	R6	R6

## **SLR(1)**

Если для грамматики невозможно построить LR(0) таблицу разбора, т.е. некоторые состояния получаются неадекватными, т.к. в строке необходимо разместить и сдвиг и свертку, переходят на следующий шаг, строят SLR(1)-таблицу разбора. SLR-анализатор иначе называется анализатор с предварительным просмотром входных символов.

Основная идея SLR – метода состоит в том, чтобы вначале построить на базе грамматик детерминированный конечный автомат для распознавания активных префиксов. Если  $G'$  – грамматика со стартовым символом  $S'$ , то  $G'$ , расширенная грамматика грамматики  $G$ , представляет собой  $G$  с новым стартовым символом  $S'$  и продукцией  $S' \rightarrow S$ . Назначение этой новой стартовой продукции – указать синтаксическому анализатору, когда он должен прекратить разбор и объявить о допущении входной строки. Т.о., допуск строки происходит тогда и только тогда, когда синтаксический анализатор выполняет свертку, соответствующую продукции  $S' \rightarrow S$ .

### **Алгоритм построения:**

Вход: Расширенная грамматика  $G'$

Выход: Функция *action* и *goto* таблицы SLR-Анализа для грамматики  $G'$ .

Метод:

- Построим  $C = \{I_0, I_1, \dots, I_n\}$  - систему множества LR(0)-пунктов для грамматики  $G'$ .
- Состояние  $I_i$  строится на основе  $I_i$ . Действия синтаксического анализа для состояния  $i$  определяется следующим образом:
  - если  $[A \rightarrow \alpha \cdot a \beta] \in I_i$  и  $\text{goto}(I_i, a) = I_j$ , то определить  $\text{action}[i, a]$  как “перенос  $j$ ”; здесь  $a$  должно быть терминалом.
  - если  $[A \rightarrow \alpha \cdot] \in I_i$ , то определить  $\text{action}[i, a]$  как “свертка  $A \rightarrow \alpha$ ” для всех  $a$  из  $\text{FOLLOW}(A)$ ; здесь  $A$  не должно быть  $S'$ .
  - если  $[S' \rightarrow S \cdot] \in I_i$ , то определить  $\text{action}[i, \$]$  как “допуск”. Если по этим правилам генерируются конфликтующие действия, то грамматика на является SLR(1). Алгоритм не в состоянии построить синтаксический анализатор для нее.
- Переходы *goto* для состояния  $i$  и всех нетерминалов  $A$  строятся по правилу: если  $\text{goto}(I_i, A) = I_j$ , то  $\text{goto}(i, A) = j$ .
- Все записи не определенные по правилам (2) и (3), указываются как ошибка.

- Начальное состояние синтаксического анализатора представляет собой состояние, построенное из множества пунктов, содержащих  $[S' \rightarrow \cdot S]$ .

Таблица синтаксического анализа, состоящая из функций *goto* и *action*, определяемых алгоритмом, называется SLR(1) – таблицей грамматики G. LR – анализатор, использующий SLR(1) – таблицу грамматики G, называется SLR(1) – анализатором для G, а соответствующая грамматика - SLR(1) – грамматикой.

	<b>S</b>	<b>idlist</b>	<b>id</b>	<b>real</b>	<b>*</b>	<b>A B C</b>	<b>+</b>
<b>1</b>	OK			S2			
<b>2</b>		S5	S4			S3	
<b>3</b>					R4		R4
<b>4</b>					R3		R3
<b>5</b>					S6		R1
<b>6</b>			S7			S3	
<b>7</b>					R2		R2

**Рассмотрим пример:**

1.  $\text{real} \Rightarrow S <\text{idlist}>$
2.  $<\text{idlist}> \Rightarrow <\text{idlist}>, <\text{id}>$
3.  $<\text{idlist}> \Rightarrow <\text{id}>$
4.  $<\text{id}> \Rightarrow A | B | C | D$

real (1,1) A (4,1) B (4,1) ,(2,2) в скобках показаны правила

Строка 5 соответствует неразличимым грамматическим вхождениям IDLIST1 ,IDLIST2 , в строке должна быть и свертка и сдвиг, поэтому строка 5 соответствует неадекватному состоянию МП-автомата. Алгоритм построения LR(0) таблицы не разрешает эту неадекватность, необходимо дополнительная информация. Для разрешения конфликта предлагается делать анализ входного символа.

#### Алгоритм построения таблицы

Элементы строки при построении таблицы заполняются таким образом. Пусть Q - грамматическое вхождение, - которым помечена строка, Z - входной символ.

а) если Q не самый правый символ любого правила для любого Z : Q  $\leq Z$  - элементы таблицы соответствий входным символом удовлетворяется условием  $Z \text{ FIRST}^* Y$  заполняются соответствующими грамматическими вхождениями этих символов (т.е. отмечается номер правила)

б) если Q- самое правое вхождение в правиле  $<L>$  , и существует Z :  $<L> = Z$  (т.е. существует правило  $U ::= .. LZ ..$ ), то в соответствующую Z, клетку вставляем элемент свертки по правилу  $<L>$

## LALR(1)

### Алгоритм построения:

Определение. Символы следователи - это символы, которые могут следовать за левой частью правила при выполнении операций приведения.

Строим символы следователи для всех символов исходя из вывода

A ::= a b c D

.... A e |

e - символ следования для a b c D;

Символы следователи приписываются грамматическим вхождениям. Если состояния имеющие одинаковые грамматические вхождения, но различных следователей, объединены в единое состояние не порождают неадекватных состояний (противоречий) то они называются LALR(1) - LR(1) с предварительным рассмотрением символов.

1. Построить SLR(1) таблицу разбора.

2. Объединить идентичные состояния, если символы-следователи игнорируются, то есть у них одинаковое количество элементов — одинаковые переходы и свертки (одинаковые переходы различаются символами-следователями). Символы-следователи объединенных элементов это объединение символов-следователей из исходных состояний.

3. Функции свертки нового LALR(1) состояния это объединение функций свертки исходных состояний.

## 11. Разбор по LR(1)таблице

LR(1) - Класс контекстно-свободных грамматики, в которых, чтобы разрешить неоднозначность разбора любого куска входного текста, необходимо не более одной предпросмотренной лексемы.

**LR(1)** грамматика – это грамматика, которая разрешает конфликты LR(0) грамматики. Допустим, у нас есть множество канонических LR(0)-состояний грамматики. Если это множество не содержит конфликтов, то можно применить LR(0)-парсер.

Иначе разрешаем возникшие конфликты, рассматривая односимвольную аванцепочку, т.е. строится LR(1) парсер с множеством LR(0)-состояний.

Символы следователи (символы, которые могут следовать за левой частью правила при выполнении операций приведения) приписываются грамматическим вхождениям. Если состояния имеющие одинаковые грамматические вхождения, но различных следователей объединены в единое состояние не порождают

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ  
 неадекватных состояний (противоречий), то они называются LALR(1) или LR(1) с предварительным рассмотрением символов.

**Рассмотрим пример:**

1.  $S \Rightarrow \text{real } \langle \text{idlist} \rangle$
  2.  $\langle \text{idlist} \rangle \Rightarrow \langle \text{idlist} \rangle, \langle \text{id} \rangle$
  3.  $\langle \text{idlist} \rangle \Rightarrow \langle \text{id} \rangle$
  4.  $\langle \text{id} \rangle \Rightarrow A \mid B \mid C \mid D$
- real (1,1) A (4,1) B (4,1) , (2,2) в скобках показаны правила

	<b>S</b>	<b>idlist</b>	<b>id</b>	<b>real</b>	-	<b>A</b>	<b>B</b>	<b>C</b>	<b>+</b>
<b>1</b>	<b>OK</b>			<b>S2</b>					
<b>2</b>		<b>S5</b>	<b>S4</b>				<b>S3</b>		
<b>3</b>					<b>R4</b>			<b>R4</b>	
<b>4</b>					<b>R3</b>			<b>R3</b>	
<b>5</b>					<b>S6</b>			<b>R1</b>	
<b>6</b>			<b>S7</b>			<b>S3</b>			
<b>7</b>					<b>R2</b>			<b>R2</b>	

Строка 5 соответствует неразличимым грамматическим вхождениям IDLIST1 ,IDLIST2 , в строке должна быть и свертка и сдвиг, поэтому строка 5 соответствует неадекватному состоянию МП-автомата. Алгоритм построения LR(0) таблицы не разрешает эту неадекватность, необходимо дополнительная информация. Для разрешения конфликта предлагается делать анализ входного символа.

Алгоритм построения таблицы в пункте *заполнения строки таблицы* должен быть изменен.

Элементы строки при построении таблицы заполняются таким образом. Пусть Q - грамматическое вхождение, - которым помечена строка, Z - входной символ.

а) если Q не самый правый символ любого правила для любого Z : Q <= Z - элементы таблицы соответствий входным символом удовлетворяется условием  $Z \text{ FIRST}^*$  Y заполняются соответствующими грамматическими вхождениями этих символов (т.е. отмечается номер правила)

б) если Q- самое правое вхождение в правиле  $\langle L \rangle$  , и существует Z :  $\langle L \rangle = Z$  (т.е. существует правило  $U ::= .. LZ ..$ ), то в соответствующую Z, клетку вставляем элемент свертки по правилу  $\langle L \rangle$

Если таблица строится по этому алгоритму, то грамматика SLR(1) (простая LR(1) грамматика).

**Алгоритм построения LALR(1)-грамматики.**

- (1)  $S \Rightarrow T \text{ else } F;$
- (2)  $T \Rightarrow E$
- (3)  $T \Rightarrow i;$
- (4)  $F \Rightarrow E$
- (5)  $E \Rightarrow E + i$
- (6)  $E \Rightarrow i$

Попытаемся строить SLR(1) таблицу

Состояние	S	T	F	E	else	«;»	«+»	i	«#»
1	<u>о</u> к 1	T1		E21,E 51				i31,i 6	
i31,i6						«+»32. R6	«+»5 2		

В строке (i31,i6) в столбце («;») получаем неадекватность, конфликт свертка – сдвиг.

Необходима дополнительная информация, для решения конфликта.

Строим символы следователи для всех символов исходя из вывода

A ::= a b c D

.... A e |

е - символ следования для a b c D;

**Рассмотрим предыдущий пример:**

- (3)  $S \Rightarrow T \text{ else } F;$
- (4)  $T \Rightarrow E$
- (3)  $T \Rightarrow i;$
- (4)  $F \Rightarrow E$
- (5)  $E \Rightarrow E + i$
- (7)  $E \Rightarrow i$

**ПЕРЕХОДЫ И СВЕРТКИ**

Состояние	S	T	F	E	else	«;»	«+»	i	«#»
1	<u>о</u> к 1	T1		E21,E 51				i31,i 6	
i31,i6						«+»32. R6	«+»5 2		

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

S1:	T (1), <<#>>	- S2
	E (2), else	- S5
	E (5), else,<<+>>	
	i (3), else	- S6
	i (6), else, <<+>>	
S2:	ELSE (1), <<#>>	- S3
S3:	F (1), <<#>>	- S4
	E (4), <<;>>	- S7
	E (5), <<;>>,<<+>>	
	i (6), <<;>>,<<+>>	- S12
S4:	F(1) ;(1), <<#>>	- S11
S5:	E(2), else	R2
	E(5), else,<<+>>	- S9
S6:	i (3), else	- S8
	i (6), else,<<+>>	R6
S7:	E(4), <<;>>	R4
	+ (5), <<;>>,<<+>>	- S9
S8:	else	R3
S9:	i (5), <<;>>, else,<<+>>	- S10
S10	(5), <<;>>, else,<<+>>	R5
S11	# (1), <<#>>	R1
S12	(6), <<;>>,<<+>>	R6

В полном алгоритме LR(1) состояния, соответствующие одинаковым грамматическим вхождениям по различным символам следователям, считаются различными. Например, состояние 10

Состояние	S	T	F	E	else	<<;>>	<<+>>	i	<<#>>
1	END	S2		S5				S6	
2					S3				S12
3				S4	S7				
4							S11		
5						R2		S9	
6						R6	S8	R6	
7							R4	S9	
8						R3			
9						R5	R5	R5	S10
10									R11
11									
12							R6	R6	

S10 (5), <<;>>, else,<<+>> R5

i5 else +

i5 ; =

надо разбить на два состояния.

В LR(1) определение состояния требует учета символов следователей.

## 12. Включение действий в синтаксис. Транслирующие грамматики.

Анализ и синтез в компиляции удобно рассматривать отдельно, но на практике это часто происходит параллельно. Это совершенно очевидно для однопроходного компилятора, в много проходных – параллельно с синтаксическим разбором создается промежуточный код.

**Пример перевода арифметического выражения в тетрады:**

Параллельно с грамматическим разбором выполняются действия – генерируются тетрады(четверки)

Грамматика + действие для арифметических выражений выглядит так:

```

<S> ----> <E> A4
<E> ----> <T> | <E> + A1 <T> A3
<T> ----> <F> | <T> * A1 <F> <A3>
<F> ----> '-' A1 <F> A2 | id A1 | (E)

```

**A1** - поместить в стек для любого идентификатора

**A3** - взять три элемента из стека, напечатать их припечатать знак «=» и очередной номер (целое число) поместить полученное целое число в стек.

**A2** - взять два элемента из стека напечатать их припечатать знак «=» и очередной номер (целое число) поместить полученной целое число в стек.

**A4** - взять один элемента из стека

**Грамматика для четверок**

```

<S>     ---> <OPER> <OP1> <OPER> = <INT>
          ---> <OP2> <OPER> = <INT>
<OPER> ---> INT | ID
<INT>   ---> DIGIT | DIGIT <INT>
<OP1>   ---> + | x
<OP2>   ---> -

```

**Пример  $(-a + b)^*(c + d)$**

Литера	Действия	Выход
(		
- (минус)	A1 поместить в стек «а»	
a	A1 поместить в стек «а»	

	A2 удалить два элемента «1» в стек	$-a = "1"$
+	A1 поместить «+» в стек	
b	A1 поместить в стек «b»	
	A3 удалить из стека три элемента; «2» в стек	$1 + b = "2"$
)		
*	A1 «*» в стек	
(		
c	A1 поместить в стек «c»	
+	A1 поместить в стек «+»	
d	A1 поместить в стек «d»	
	A3 удалить из стека три элемента; «3» в стек	$c + d = "3"$
)		
	A3 удалить из стека три элемента; «4» в стек	$"2" **$ $3" = "4"$
	A4 удалить из стека элемент	

Грамматики могут содержать вызов действий не только для генерации кода, но и для выполнения других задач.

### Работа с таблицей символов

Поскольку синтаксический анализатор обычно использует контекстно-свободную грамматику, необходимо найти метод определения контекстно-зависимых частей языка. Например, во многих языках идентификаторы не могут применяться, если они не описаны, и имеются ограничения в отношении способов употребления в программе значений различных типов. Для запоминания описанных идентификаторов и их типов большинство компиляторов пользуются таблицей символов.

### **Определяющая и прикладная реализация**

Когда описывается идентификатор, например

int a

Мы говорим, что имеется определяющая реализация «а».

Однако «а» может встречаться и в другом контексте:

a:=4 или a+b или read(a),

здесь будут прикладные реализации «а» .

При обработке :

- определяющей реализации идентификатора компилятор помещает объект в таблицу символов,
- прикладной реализации - в таблице символов осуществляется поиск элемента, соответствующего определяющей реализации объекта, чтобы узнать его тип и (возможно) другие признаки, требующиеся во время компиляции.

### **Блочная структура программы**

Во многих языках один и тот же идентификатор может использоваться для представления в разных частях программы различных объектов. В таких случаях структура программы помогает различать эти объекты, например

```
begin int a;  
...  
end;  
begin char a  
...  
end;
```

Таблица символов должна иметь ту же блочную структуру, что и программа, чтобы различить виды употребления одного и того же идентификатора.

Основой построения СУ -схем перевода является использование двух грамматик, с помощью которых осуществляется синхронный вывод входной и выходной цепочек. Построение транслирующих грамматик предполагает применение другого подхода, который предусматривает использование одной грамматики и разрешает включение как входных, так и выходных символов в каждое правило такой грамматики.

**Определение.** Транслирующей грамматикой (Т -грамматикой) называется КС-грамматика, множество терминальных символов которой разбито на множество входных символов и множество выходных символов, которые называются также символами действия.

Примером Т - грамматики может служить следующая грамматика:

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

$\Gamma: V_{\text{вх}} = \{a, b, c\}, V_{\text{вых}} = \{x, y, z\}, V_a = \{I, A\}$

$$R = \{ \begin{aligned} & <|> \rightarrow a <|>x <A>, \\ & <|> \rightarrow z, \\ & <A> \rightarrow <A><C>, \\ & <A> \rightarrow by \\ \end{aligned} \}.$$

Чтобы не возникало путаницы в случае использования одинаковых символов во входном и выходном алфавитах, условимся выделять выходные символы фигурными скобками. С использованием таких обозначений правила грамматики ГТ4.1 имеют вид:

$$R = \{ \begin{aligned} & <|>\rightarrow a <|>\{x\} <A>, \\ & <|>\rightarrow \{z\}, \\ & <A>\rightarrow <A>c, \\ & <A>\rightarrow b\{y\} \\ \end{aligned} \}.$$

Вывод в транслирующих грамматиках выполняется по тем же правилам, что и в обычных КС - грамматиках. Например, в рассматриваемой грамматике из начального символа может быть выведена следующая цепочка:

$$<|> \implies a <|>\{x\} <A> \implies a\{z\}\{x\} <A> \implies a\{z\}\{x\}b\{y\}$$

Каждый символ или цепочка символов, заключенные в фигурные скобки, должны рассматриваться как единый символ, называемый символом действия. В общем случае цепочки символов, заключенные в фигурные скобки, можно интерпретировать как имена процедур, выполнение которых производит требуемый эффект на выходе. При описании перевода обычно предусматривают, что каждый символ действия представляет собой процедуру, осуществляющую передачу символа, заключенного в фигурные скобки, на выход. Когда нужно подчеркнуть, что используется такая интерпретация символов действия, то Т - грамматику называют грамматикой цепочного перевода.

### **Входная и выходная грамматики заданной транслирующей грамматики.**

Из каждой Т - грамматики можно получить две обычных грамматики, одна из которых позволяет строить входные цепочки, а другая - выходные. Правила построения таких грамматик можно сформулировать следующим образом.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

**Определение.** Если из правил транслирующей грамматики ГТ удалить выходные символы, то получим входную грамматику ГТвх для заданной грамматики. Если из правил заданной транслирующей грамматики удалить входные символы, то получим выходную грамматику ГТвых заданной транслирующей грамматики ГТ. Язык, порождаемый грамматикой ГТвх, называется входным языком заданной транслирующей грамматики, а язык, порождаемый ГТвых, называется выходным языком заданной транслирующей грамматики ГТ.

Цепочки символов, получаемые путем вывода в Т - грамматике, содержат как символы входного алфавита, так и символы выходного алфавита - символы действия. Каждую такую цепочку можно представить как пару, состоящую из входной и выходной цепочки.

**Определение.** Если из цепочки символов  $a_1$ , полученной путем вывода в заданной Т - грамматике, исключим все выходные символы, то получим цепочку  $a_2$ , которую назовем входной цепочкой. Если же из цепочки  $a_1$  исключим все символы входного алфавита, то в результате получим цепочку  $a_2$ , которую назовем выходной цепочкой, порождаемой Т - грамматикой. Цепочки  $a_1$  и  $a_2$  образуют пару, выводимую в заданной Т - грамматике.

### **Построение транслирующей грамматики по СУ - схеме.**

**Определение.** Множество пар цепочек, выводимых с помощью правил заданной Т - грамматики, образуют перевод, определяемый этой грамматикой.

Последнее определение позволяет нам сделать вывод, что один и тот же перевод может быть задан как с помощью СУ - схемы, так и с помощью Т - грамматики. Эти два способа задания являются равноправными и, более того, они допускают преобразование друг в друга.

Возможность одного из таких преобразований устанавливается следующим утверждением.

**Утверждение.** Для каждой простой СУ-схемы Т можно построить транслирующую грамматику ГТ такую, что переводы, порождаемые СУ - схемой и Т - грамматикой, совпадают.

$$C(T) = C(\Gamma T)$$

Чтобы показать справедливость этого утверждения, опишем способ построения Т - грамматики по заданной СУ - схеме.

Допустим, что задана СУ - схема

$$T = \{V_{T^H}, V_{T^H}, V_A, Q, I\}$$

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

и требуется построить Т - грамматику

$$\Gamma \sim = \{V'tbx, V'tvых, V'a, R, I\}.$$

Для того чтобы получить тот же самый перевод, необходимо, чтобы

$$Vtbx = Vtbx', Vtvых = Vtvых', Va = Va'.$$

Рассмотрим преобразование правила из множества Q СУ - схемы A->a ,b , где цепочки

$$a = xoAox1A1...xnAn \text{ и } b = yoAoy1A1...ynAn,$$

и поставим в соответствие этому правилу правило грамматики в виде:

$$A \rightarrow xoAox1y1A1...xlypAn.$$

Это можно сделать всегда, поскольку СУ - схема - простая и в каждом ее правиле используются одни и те же нетерминалы в одном и том же порядке. Рассмотренное построение обеспечивает включение во входную цепочку выходных символов цепочки, порождаемой СУ-схемой. Следовательно, каждый шаг вывода в Т - грамматике будет добавлять к выводимой цепочке те же символы, что и СУ - схема добавляет к выходной цепочке.

Применение описанных положений рассмотрим на примере построения Т - грамматики по СУ - схеме T4.4.

Используя фигурные скобки для представления выходных символов и выполняя пре-образование, получаем грамматику

$$\begin{aligned}
 R = & \{ \langle A \rangle \rightarrow x\{x\}, \\
 & \langle A \rangle \rightarrow (\langle B \rangle), \\
 & \langle B \rangle \rightarrow \langle A \rangle \langle C \rangle, \\
 & \langle C \rangle \rightarrow + \langle A \rangle \{+ \} \langle C \rangle, \\
 & \langle C \rangle \rightarrow \$ \\
 \}.
 \end{aligned}$$

Перевод цепочки  $((x+x)+x)$  с применением правил построенной грамматики может быть получен с помощью следующего вывода:

$$\begin{aligned}
 & \langle A \rangle ==> \langle A \rangle \langle C \rangle ==> (\langle B \rangle) \langle C \rangle ==> (\langle A \rangle \langle C \rangle) \langle C \rangle ==> \\
 & ((\langle B \rangle) \langle C \rangle) \langle C \rangle ==> ((\langle A \rangle \langle C \rangle) \langle C \rangle) \langle C \rangle ==> ((x\{x\} \langle C \rangle) \langle C \rangle ==> \\
 & ((x\{x\} + \langle A \rangle \{+ \} \langle C \rangle) \langle C \rangle) \langle C \rangle ==> \\
 & ((x\{x\} + x\{x\} \{+ \}) \langle C \rangle) \langle C \rangle ==> ((x\{x\} + x\{x\} \{+ \}) \langle C \rangle) \langle C \rangle ==> \\
 & ((x\{x\} + x\{x\} \{+ \}) + \langle A \rangle \{+ \} \langle C \rangle) \langle C \rangle ==> \\
 & ((x\{x\} + x\{x\} \{+ \}) + x\{x\} \{+ \} \langle C \rangle) \langle C \rangle ==> \\
 & ((x\{x\} + x\{x\} \{+ \}) + x\{x\} \{+ \}) \langle C \rangle ==> ((x\{x\} + x\{x\} \{+ \}) + x\{x\} \{+ \}).
 \end{aligned}$$

Исключая из полученной цепочки вначале выходные, а затем входные символы, получаем выводимую пару  $((x+x)+x, \{x\}\{+ \}\{x\}\{+ \})$ ,

которая совпадает с результатом вывода в заданной СУ - схеме.

### 13. Атрибутивные грамматики. Синтезируемый и наследуемый атрибуты.

**Атрибутивная транслирующая грамматика** – это транслирующая грамматика, к которой добавляются следующие определения:

1. Каждый входной символ, символ действия или нетерминальный символ имеет конечное множество атрибутов, и каждый атрибут имеет (возможно, бесконечное) множество допустимых значений.
2. Все атрибуты нетерминальных символов и символов действия делятся на наследуемые и синтезируемые.
3. Правила вычисления наследуемых атрибутов определяются следующим образом:
  - каждому вхождению наследуемого атрибута в правую часть данной продукции сопоставляется правило вычисления значения этого атрибута как функции некоторых других атрибутов символов, входящих в правую или левую часть данной продукции
  - задается начальное значение каждого наследуемого атрибута начального символа
4. Правила вычисления синтезируемых атрибутов определяются так:
  - каждому вхождению синтезируемого нетерминального атрибута в левую часть данной продукции сопоставляется правило вычисления значения этого атрибута как функции некоторых других атрибутов символов, входящих в левую или правую часть данной продукции;
  - каждому синтезируемому атрибуту символа действия сопоставляется правило вычисления значения этого атрибута как функции некоторых других атрибутов этого символа действия.

Главная идея, лежащая в основе понятия атрибутивной грамматики, состоит в том, что значения символов сопоставляются всем вершинам дерева вывода, как терминальным, так и нетерминальным. Отношения между входными и выходными значениями выражаются по принципу “от правила к правилу”, при этом значения находятся в вершинах дерева. Атрибуты нетерминалов могут передаваться и вычисляться по дереву вывода сверху вниз (синтезируемые атрибуты) и снизу вверх (наследуемые атрибуты).

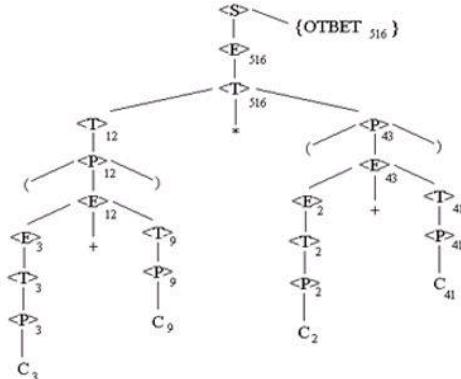
Пример синтезирующего атрибута.

Предположим существует лексический блок, задающий входное множество  $\{(., +, *, C)\}$ ,  $C\text{-const}$  для синтаксического блока допускающего выражения

1.  $S ::= <E> \{OTBET\}$
2.  $E ::= <E> + <T>$
3.  $E ::= <T>$
4.  $T ::= <T> * <P>$
5.  $T ::= <P>$
6.  $P ::= (<E>)$
7.  $P ::= C$

Значение выходного символа  $\{OTBET\}$  должно быть числом.

Требуемое отношение между значениями входных лексем и значением выходного OTBET можно выразить словами "OTBET - это числовое значение входного выражения". Найдем математическое выражение этой фразы. Рассмотрим конкретную входную цепочку:  $(C_3 + C_9) * (C_2 + C_{41})$ , где значения входных лексем, выданных лексическим блоком, указаны индексами. Этой входной цепочке должна соответствовать выходная цепочка  $OTBET_{516}$ . Построим дерево вывода, соответствующее данной входной цепочке.



Каждый нетерминал этого дерева  $<E>$ ,  $<T>$ ,  $<P>$  помечен значением соответствующего подвыражения, а выходной символ – требуемым выходным значением.

Чтобы определить, как расставлять значения на дереве вывода, получаемом по данной грамматике, вначале определим, как получить значение любой нетерминальной вершины, если даны значения ее прямых потомков. С этой целью сопоставим каждому правилу грамматики для  $<E>$ ,  $<T>$  и  $<P>$  правило вычисления значения вершины, соответствующей нетерминалу

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

левой части продукции, по данным значениям ее прямых потомков, соответствующих символам правой части грамматики.

Например, правило вычисления значений для продукции  $E \sqsubset E + T$  состоит в том, что значения нетерминала  $E$  в левой части равно сумме значения нетерминала  $E$  в правой части и значения нетерминала  $T$ . Чтобы применить это правило к левому, заключенному в скобки вхождению  $E$ , в дереве вывода примера, значение  $E$  в правой части равно 3, а значение  $T$  равно 9. Отсюда следует, что значение  $E$  в левой части равно 12 и можно приписать значение 12 соответствующей вершине дерева.

Правило для продукции  $E \sqsubset T$  состоит в том, что значение  $E$  равно значению  $T$ . Правило для продукции  $T \sqsubset T^*P$  состоит в том, что значение  $T$  в левой части равно произведению значения  $T$  в правой части и значения  $P$ . Правило для  $T \sqsubset P$  состоит в том, что значение  $T$  равно значению  $P$ . Правило для  $P \sqsubset (E)$  состоит в том, что значение  $P$  равно значению  $E$ . Правило для  $P \sqsubset C$  состоит в том, что значение  $P$  равно значению  $C$ .

И, наконец, продукцию  $S \sqsubset E\{\text{OTBET}\}$  снабдим правилом, что значение символа ОТБЕТ равно значению  $E$ .

Для того, чтобы выразить приведенные выше правила математически более строго, можно в каждой продукции дать разные имена разным встречающимся в ней значениям, а затем сформулировать соответствующие ей правила при помощи этих имен.

Тогда продукции и соответствующие им правила можно записать так:

1.  $S \sqsubset E_q \{\text{OTBET}_r\}$   
 $r \sqsubset q$
2.  $E \sqsubset E_q + T_r$   
 $p \sqsubset q + r$
3.  $E_p \sqsubset T_q$   
 $p \sqsubset q$
4.  $T_p \sqsubset T_q * P_r$   
 $p \sqsubset q * r$
5.  $T_p \sqsubset P_q$   
 $p \sqsubset q$
6.  $P_p \sqsubset (E_q)$   
 $p \sqsubset q$
7.  $P_p \sqsubset C_q$   
 $p \sqsubset q$

Использование имен переменных локально для каждой продукции, и, если одна и та же буква  $v$  встречается в двух разных продукциях, это не имеет значения.

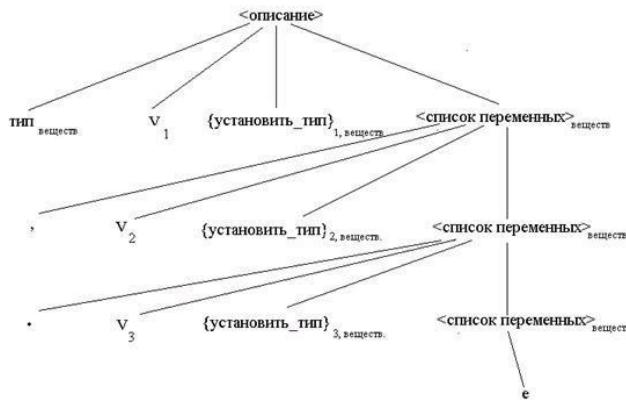
Приведенные выше продукции вместе с соответствующими правилами вычисления значений являются примерами "атрибутных правил", и взятые вместе с начальным нетерминалом  $\langle S \rangle$  образуют "атрибутную грамматику". Значения вышеприведенных правил называются "атрибутами".

В этом примере значение атрибута каждого нетерминала определяется символами, расположенными в дереве вывода под этим нетерминалом. Такое "восходящее" вычисление выражается в том, что правила вычисления атрибутов нетерминалов, ассоциированные с продукциями, указывают, как вычислять атрибуты в левой части продукции по данным атрибутам символов правой части. Атрибуты, значения которых получаются таким восходящим способом, то есть снизу вверх, называются "синтезируемыми" атрибутами.

### Пример наследуемого атрибута.

Рассмотрим следующую грамматику:

1.  $\langle \text{описание} \rangle ::= \text{type } \text{id} \langle \text{список переменных} \rangle$
2.  $\langle \text{список переменных} \rangle ::= , \text{id} \langle \text{список переменных} \rangle$
3.  $\langle \text{список переменных} \rangle ::= \epsilon / \text{пустое правило}$



Предположим, что существует лексический блок, задающий три лексемы  $\text{id}$  ТИП, где лексема  $\text{id}$  обозначает идентификатор и ее значение является указателем на соответствующий этому идентификатору элемент таблицы имен. ТИП – лексема со значением, определяющим, какой из типов, ВЕЩЕСТВЕННЫЙ,

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ  
ЦЕЛЫЙ или ЛОГИЧЕСКИЙ, должен быть поставлен в соответствии идентификаторам из данного списка.

При обработке описания каждой переменной синтаксический блок вызывает процедуру УСТАНОВИТЬ\_ТИП, которая помещает один из типов ВЕЩЕСТВЕННЫЙ, ЦЕЛЫЙ или ЛОГИЧЕСКИЙ в надлежащее поле таблицы символов. Вызов процедуры УСТАНОВИТЬ\_ТИП лучше всего осуществлять сразу после того как, идентификатор поступил на блок синтаксического анализатора. Это можно описать следующей грамматикой, использующей символ действия:

1. <описание> ::= type id {УСТАНОВИТЬ\_ТИП} <список переменных>
2. <список переменных> ::=, id {УСТАНОВИТЬ\_ТИП} <список переменных>
3. <список переменных> ::= e /пустое правило/

Пусть процедура УСТАНОВИТЬ\_ТИП имеет два аргумента: это идентификатор (или его адрес) и тип. Тогда вызов процедуры УСТАНОВИТЬ\_ТИП можно записать УСТАНОВИТЬ\_ТИП (id, type). Введем в вышеприведенную грамматику атрибуты и правила их вычисления, чтобы в последовательностях актов входные символы были представлены вместе с их значениями, играющими роль атрибутов, а вхождения символов действия {УСТАНОВИТЬ\_ТИП} имели по два атрибута, представляющих аргументы соответствующего вызова процедуры УСТАНОВИТЬ\_ТИП. Тогда вхождения УСТАНОВИТЬ\_ТИП будут иметь такой вид: {УСТАНОВИТЬ\_ТИП}<sub>id, type</sub>.

Рассмотрим, как УСТАНОВИТЬ\_ТИП получает свои атрибуты. В правиле 1 это делается просто, так как атрибуты можно получить используя входные символы type и id, входящие в это правило. В правиле 2 type не доступен, его нужно передать используя атрибуты нетерминала. Поэтому нетерминал <список переменных> должен иметь атрибут, который будет представлять тип.

Таким образом, грамматика перепишется :

1. <описание> ::= type<sub>t</sub> id<sub>p</sub> {УСТАНОВИТЬ\_ТИП}<sub>p1, t1</sub> <список переменных><sub>t2</sub>  
(t2, t1) □□ t , p1 □□ p
2. <список переменных><sub>t</sub> ::=, id<sub>p</sub> {УСТАНОВИТЬ\_ТИП}<sub>p1, t1</sub>  
<список переменных><sub>t2</sub>  
(t2, t1) □□ t , p1 □□ p
3. <список переменных><sub>t</sub> ::= e /пустое правило/

Запись означает, что  $t$  присваивается одновременно  $t_1$  и  $t_2$ . На рис.2. показано атрибутное дерево вывода последовательности ТИП<sub>вещественный</sub>  $V_1, V_2, V_3$  определяемое данной грамматикой. Чтобы получить значения атрибутов, соответствующие вхождениям нетерминала (список переменных), используются символы, расположенные выше в дереве вывода, или символы, входящие в ту же часть правила. Входной символ ТИП определяет значение ВЕЩЕСТВЕННЫЙ, которое передается самому верхнему вхождению нетерминала (список переменных), а затем передается вниз по дереву другим вхождениям.

Такой нисходящий характер вычисления значений атрибутов отражается в том, что каждое правило вычисления атрибутов нетерминалов, сопоставленное продукциям, указывает, как вычислять атрибуты нетерминала, входящего в правую часть продукции. Атрибуты, значения которых задаются таким нисходящим способом, называются “наследуемыми” атрибутами.

#### **14.Таблица символов. Назначение, структура.**

В процессе работы компилятор хранит информацию об объектах программы. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и его свойств. Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрей, а требуемая память по возможности меньше.

Кроме того, со стороны языка программирования могут быть дополнительные требования. Имена могут иметь определенную область видимости. Например, поле записи должно быть уникально в пределах структуры (или уровня структуры), но может совпадать с именем объектов вне записи (или другого уровня записи). В то же время имя поля может открываться оператором присоединения, и тогда может возникнуть конфликт имен (или неоднозначность в трактовке имени). Если язык имеет блочную структуру, то необходимо обеспечить такой способ хранения информации, чтобы, во-первых, поддерживать блочный механизм видимости, а во-вторых - эффективно освобождать память по выходе из блока. В некоторых языках (например, Аде) одновременно (в одном блоке) могут быть видимы несколько объектов с одним именем, в других такая ситуация недопустима.

### **Таблицы идентификаторов и таблицы символов**

Как уже было сказано, информацию об объекте обычно можно разделить на две части: имя (идентификатор) и описание. Удобно эти характеристики объекта хранить по отдельности. Это обусловлено двумя причинами: 1) символьное представление идентификатора может иметь неопределенную длину и быть довольно длинным; 2) как уже было сказано, различные объекты в одной области видимости и/или в разных могут иметь одинаковые имена и незачем занимать память для повторного хранения идентификатора. Таблицу для хранения идентификаторов называют таблицей идентификаторов, а таблицу для хранения свойств объектов - таблицей символов. В таком случае одним из свойств объекта становится его имя и в таблице символов хранится указатель на соответствующий вход в таблицу идентификаторов.

### **Таблицы идентификаторов**

Если длина идентификатора ограничена (или имя идентифицируется по ограниченному числу первых символов идентификатора), то таблица идентификаторов может быть организована в виде простого массива строк фиксированной длины. Некоторые входы могут быть заняты, некоторые —

Имя объекта	Описание объекта			
s	o	r	t	
a				
r	e	a	d	
i				

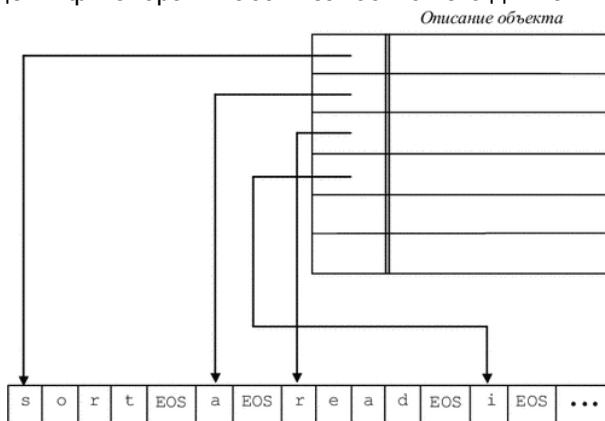
свободны.

Ясно, что, во-первых, размер массива должен быть не меньше числа идентификаторов, которые могут реально появиться в программе (в противном случае возникает переполнение таблицы); во-вторых, как правило, потенциальное число различных идентификаторов существенно больше размера таблицы.

Заметим, что в большинстве языков программирования символьное представление идентификатора может иметь

произвольную длину. Кроме того, различные объекты в одной или в разных областях видимости могут иметь одинаковые имена, и нет большого смысла занимать память для повторного хранения идентификатора. Таким образом, удобно имя объекта и его описание хранить по отдельности.

В этом случае идентификаторы хранятся в отдельной таблице - таблице идентификаторов. В таблице символов же хранится указатель на соответствующий вход в таблицу идентификаторов. Таблицу идентификаторов можно организовать, например, в виде сплошного массива. Идентификатор в массиве заканчивается каким-либо специальным символом EOS (рис.2). Второй возможный вариант - в качестве первого символа идентификатора в массив заносится его длина.



### Таблицы расстановки

Одним из эффективных способов организации таблицы символов является таблица расстановки (или хеш-таблица). Поиск в такой таблице может быть организован методом повторной расстановки. Суть его заключается в следующем.

Таблица символов представляет собой массив фиксированного размера  $N$ . Идентификаторы могут храниться как в самой таблице символов, так и в отдельной таблице идентификаторов. Определим некоторую функцию  $h1$  (первичную функцию расстановки), определенную на множестве идентификаторов и принимающую значения от 0 до  $N - 1$  (т.е.  $0 \leq h1(id) \leq N - 1$ , где  $id$  - символьное представление идентификатора). Таким образом, функция расстановки сопоставляет идентификатору некоторый адрес в таблице символов.

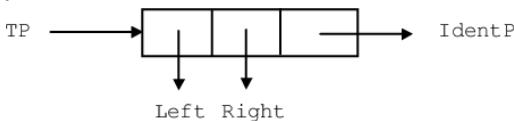
Пусть мы хотим найти в таблице идентификатор  $id$ . Если элемент таблицы с номером  $h1(id)$  не заполнен, то это означает, что

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

идентификатора в таблице нет. Если же занят, то это еще не означает, что идентификатор  $id$  в таблицу занесен, поскольку (вообще говоря) много идентификаторов могут иметь одно и то же значение функции расстановки. Для того чтобы определить, нашли ли мы нужный идентификатор, сравниваем  $id$  с элементом таблицы  $h1(id)$ . Если они равны - идентификатор найден, если нет - надо продолжать поиск дальше.

Для этого вычисляется вторичная функция расстановки  $h2(h)$  (значением которой опять таки является некоторый адрес в таблице символов). Возможны четыре варианта:

- элемент таблицы не заполнен (т.е. идентификатора в таблице нет),
- идентификатор элемента таблицы совпадает с искомым (т.е. идентификатор найден),
- адрес элемента совпадает с уже просмотренным (т.е. таблица вся просмотрена и идентификатора нет)
- предыдущие варианты не выполняются, так что необходимо продолжать поиск.

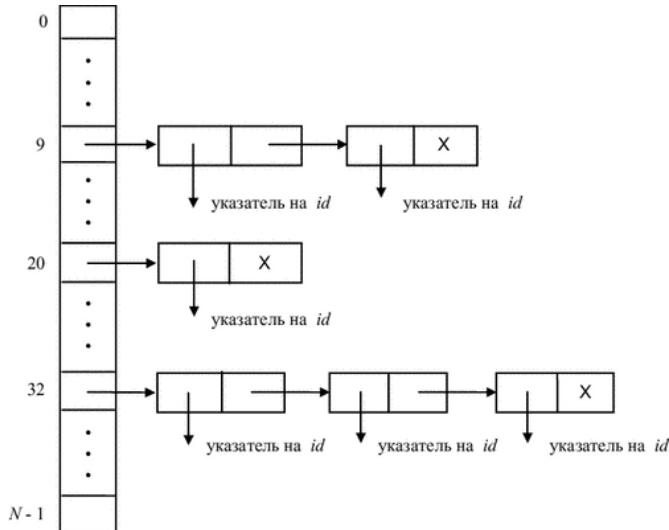


### **Таблицы расстановки со списками**

Только что описанная схема страдает одним недостатком - возможностью переполнения таблицы. Рассмотрим ее модификацию, когда все элементы, имеющие одинаковое значение (первой) функции расстановки, связываются в список (при этом отпадает необходимость использования функций  $h_i$  для  $i \geq 2$ ). Таблица расстановки со списками - это массив указателей на списки элементов (рис. 3).

Вначале таблица расстановки пуста (все элементы имеют значение NULL). При поиске идентификатора  $Id$  вычисляется функция расстановки  $h(Id)$  и просматривается соответствующий линейный список.

Рассмотрим еще один способ организации таблиц символов с использованием **двоичных деревьев**.



Пусть на множестве идентификаторов задан некоторый линейный (например, лексикографический) порядок , т.е. некоторое транзитивное, антисимметричное и антирефлексивное отношение. Таким образом, для произвольной пары идентификаторов  $id_1$  и  $id_2$  либо  $id_1 \prec id_2$ , либо  $id_2 \prec id_1$ , либо  $id_1$  совпадает с  $id_2$ .

Рис. 5:

Каждой вершине двоичного дерева, представляющего таблицу символов, сопоставим идентификатор. При этом, если вершина (которой сопоставлен  $id$ ) имеет левого потомка (к которому сопоставлен  $id_L$ ), то  $id_L \prec id$ ; если имеет правого потомка ( $id_R$ ), то  $id \prec id_R$ . Элемент таблицы изображен на рис. 5.

Среднее время поиска в таблице размера  $n$ , организованной в виде двоичного дерева, при равной вероятности появления каждого объекта равно  $(2 \ln 2) \log 2n + O(1)$ . Однако, на практике случай равной вероятности появления объектов встречается довольно редко. Поэтому в дереве появляются более длинные и более короткие ветви, и среднее время поиска увеличивается.

## **Реализация блочной структуры**

С точки зрения структуры программы блоки (и/или процедуры) образуют дерево. Каждой вершине дерева этого представления, соответствующей блоку, можно сопоставить свою таблицу символов (и, возможно, одну общую таблицу идентификаторов). Работу с таблицами блоков можно организовать в магазинном режиме: при входе в блок создавать таблицу символов, при выходе - уничтожать. При этом сами таблицы должны быть связаны в упорядоченный список, чтобы можно было просматривать их в порядке вложенности. Если таблицы организованы с помощью функций расстановки, это означает, что для каждой таблицы должна быть создана своя таблица расстановки.

## **Сравнение методов реализации таблиц**

Использование динамической памяти, как правило, довольно дорогая операция, поскольку механизмы поддержания работы с динамической памятью могут быть достаточно сложны. Необходимо поддерживать списки свободной и занятой памяти, выбирать наиболее подходящий кусок памяти при запросе, включать освободившийся кусок в список свободной памяти и, возможно, склеивать куски свободной памяти в списке.

С другой стороны, использование массива требует отведения заранее довольно большой памяти, а это означает, что значительная память вообще не будет использоваться. Кроме того, часто приходится заполнять не все элементы массива (например, в таблице идентификаторов или в тех случаях, когда в массиве фактически хранятся записи переменной длины, например, если в таблице символов записи для различных объектов имеют различный состав полей). Обращение к элементам массива может означать использование операции умножения при вычислении индексов, что может замедлить выполнение.

Наилучшим, по-видимому, является механизм доступа по указателям и использование факта магазинной организации памяти в компиляторе. Для этого процедура выделения памяти выдает необходимый кусок из подряд идущей памяти, а при выходе из процедуры вся память, связанная с этой процедурой, освобождается простой перестановкой указателя свободной памяти в состояние перед началом обработки процедуры. В чистом виде это не всегда, однако, возможно. Например, локальный модуль в Модуле-2 может экспорттировать некоторые

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

объекты наружу. При этом схему реализации приходится «подгонять» под механизм распределения памяти. В данном случае, например, необходимо экспорттированные объекты вынести в среду охватывающего блока и свернуть блок локального модуля.

### **15. Распределение памяти. Статическая и динамическая память.**

При создании компилятора необходимо различать два важных класса информации о программе:

**Статическая информация**, т.е. информация, известная во время компиляции

**Динамическая информация**, т.е. сведения, неизвестные во время компиляции, но которые станут известны во время выполнения программы

В приложении к управлению памятью разделение всей информации на статическую и динамическую позволяет определить, каким механизмом распределения памяти необходимо пользоваться для той или переменной, структуры или процедуры. Например, размер памяти, необходимой под простые переменные, можно вычислить (и, соответственно, выделить необходимую память) уже во время компиляции, а вот память, запрашиваемую пользователем с размером, заданным с помощью переменной, придется выделять уже во время выполнения программы. Понятно, что статическое распределение памяти при прочих равных условиях предпочтительнее ("дешевле").

Особенно интересны "пограничные" случаи, такие, как выделение памяти под массивы. Дело в том, что размер памяти, необходимой под массивы фиксированного размера, в большинстве современных языках программирования можно посчитать статически. И тем не менее, иногда распределение памяти под массивы откладывают на этап выполнения программы. Это может быть осмысленно, например, для языков, разрешающих описание динамических массивов, т.е. массивов с границей, неизвестной во время компиляции

Можно выделить три основных метода управления памятью:

- Статическое распределение памяти
- Стековое распределение памяти
- Представление памяти в виде кучи (heap)

Ранее многие языки проектировались в расчете на какой-то один из этих механизмов управления памятью. Сегодня большинство языков программирования требуют от компиляторов

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

использования всех трех механизмов управления памятью - обычно подразумевается применение самого "дешевого" из пригодных способов.

**Статическое управление памятью** - простейший способ распределения памяти и было достаточно распространено на заре развития языков программирования. Если структура языка позволяет обойтись статическим распределением памяти, то удается достичь максимальной эффективности программы в целом, так как во время выполнения не приходится выделять и освобождать память, делать какие-либо проверки и прочие дорогостоящие и часто возникающие операции.

Желание свести все к статическому управлению памятью заставляет разработчиков отказаться от многих конструкций, привычных для программиста. Приведем некоторые примеры: рекурсивные процедуры (так как неизвестно, сколько раз будет вызвана процедура, и непонятно, как различать экземпляры процедуры), массивы с неконстантными или изменяющимися границами и вложенные процедуры/подпрограммы. Подводя краткие итоги, в распоряжении программиста оказываются только простые переменные, структуры и массивы фиксированного размера.

Рассмотрим статическое управление памятью на небольшом примере на Фортране. В этом языке вся память может быть выделена статически и во время выполнения программы будут меняться только значения простых переменных и элементы массива. Для этого каждая функция транслируется в статически выделенную область памяти, которая содержит сам код и связанные с ним данные, а связь между подпрограммами осуществляется через блоки данных, общие для нескольких подпрограмм ( COMMON ) или путем передачи параметров и передачи управления при нерекурсивных вызовах. Типы данных могут быть только одного из пяти заранее заданных видов, а переменные никогда не освобождаются.

Все это позволяет прибегнуть к максимально простому способу представления данных в памяти - одномерному массиву переменных. При этом главная функция компилируется независимо от функции SUM и потому для них создаются два различных адресных пространства. Странслированные подпрограммы объединяются уже только во время загрузки.

Практически во всех языках управление памятью включает в себя статическую компоненту, так как этот способ распределения памяти наиболее дешев и не требует накладных расходов во время исполнения. Статически можно распределять константы,

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

переменные и фиксированные массивы. В более сложных языках программирования для распределения памяти нам придется отталкиваться от значений, которые станут известны только во время исполнения.

### **Стековое управление памятью**

Так как статическое распределение памяти чрезмерно ограничивает программиста, проектировщикам языков программирования со временем пришлось перейти к более сложным средствам управления памятью, работающим во время выполнения программы. Самым простым из таких методов является стековое управление памятью. Его идея заключается в том, что при входе в блок или процедуру на вершине специального стека выделяется память, необходимая для размещения переменных, объявленных внутри этого блока. При выходе же из блока память снимается не "вразнобой", а всегда только с вершины стека. Понятно, что задачи утилизации и повторного использования становятся тривиальными, а проблемы уплотнения просто не существует.

При таком подходе все значения каждого блока или процедуры объединяются в единую часть стека, называемую рамкой . Для управления памятью нам потребуется указатель стека, показывающий на первый свободный элемент, и указатель рамки, хранящий адрес дна рамки (это потребуется при выходе из блока или процедуры).

Стековое управление памятью особенно выгодно для языков со строгой вложенной структурой входов в процедуры и выходов из них. Дополнительно необходимо потребовать, чтобы структуры данных имели фиксированный размер, могли создаваться программистом только при входе в процедуру и обязательно уничтожались при выходе. Тогда всю информацию, необходимую для работы данной процедуры или подпрограммы, можно собрать в так называемую активационную запись , полностью определяющую данный экземпляр процедуры. В таком случае стекового управления памятью достаточно для всех элементов данных, используемых в программе.

### **Управление кучей**

Этот механизм предназначен для работы со всеми структурами данных, которые по тем или иным причинам не пригодны для статического или стекового распределения памяти. Потребность в куче возникает всякий раз, когда выделение и освобождение памяти может потребоваться в непредсказуемый момент времени. Более того, большинство современных объектно-

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

ориентированных языков программирования попросту не могут обойтись без динамического выделения и уничтожения объектов. При управлении кучей проблемы утилизации и уплотнения памяти становятся весьма серьезными: моменты возврата памяти не всегда очевидны, порядок возврата памяти просто непредсказуем, а память, отведенная для распределения под новые объекты, обычно напоминает решето. Решение этих проблем возлагается на механизм сборки мусора . В этом процессе выделенная ранее память пересматривается с целью обнаружения неиспользуемых фрагментов, а высвобожденная память передается для повторного использования.

Наверное, самой критичной проблемой управления кучей является отслеживание активных элементов в памяти: как только возникнет потребность в освобождении дополнительной памяти, процесс сборки мусора должен будет утилизировать все неиспользуемые более фрагменты памяти. Определить, используется ли данный момент в программе, можно несколькими способами. Наиболее распространенными из них являются счетчики ссылок и различные алгоритмы разметки памяти.

Самый простой способ отслеживания свободной памяти заключается в приписывании каждому объекту в памяти специального счетчика ссылок, показывающего количество "живых" переменных, использующих данный объект. При первичном выделении памяти в программе счетчику присваивается значение, равное единице; при создании новых указателей на данный фрагмент памяти, значение счетчика увеличивается на единицу. Если какая-то из переменных, указывающих на данный фрагмент памяти, перестает существовать, значение счетчика ссылок уменьшается на единицу. При достижении счетчиком нуля фрагмент памяти считается более не используемым и может быть утилизирован. Отметим также, что в эту схему легко вписывается и явное управление памятью: оператор free приводит к простому уменьшению счетчика ссылок на единицу.

Этот метод прост, понятен и легко реализуется, но, к сожалению, у этого метода есть серьезные недостатки. Во-первых, приведенный выше алгоритм в некоторых случаях не справляется с определением свободной памяти. Например, для циклического списка уничтожение внешнего указателя на эту структуру делает ее мусором, и в то же время счетчики ссылок циклического списка не становятся равными нулю, т.к. элементы списка указывают друг на друга.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Во-вторых, использование механизма счетчиков ссылок связано со значительной потерей эффективности во время исполнения, так как при каждом присваивании в программе необходимо производить соответствующие арифметические операции. Все эти действия могут оказаться совершенно бесполезными, если программе хватит исходного объема памяти, а это противоречит одному из принципов трансляции, согласно которому программы не должны терять в производительности из-за языковых или компиляторных механизмов, которыми они не пользуются.

Из-за этих проблем механизм счетчиков ссылок не получил широкого распространения.

Другим методом отслеживания свободной памяти является механизм **разметки памяти**. При этом подходе все действия по поиску неиспользуемых переменных откладываются до возникновения недостатка памяти, т.е. до того момента, когда программа требует выделить фрагмент слишком большого размера. В этот момент стартует процесс сборки мусора, начинающийся именно с разметки памяти.

Разметка памяти начинается с обнаружения всех заведомо живых элементов программы. К таким причисляются все объекты за пределами кучи (на стеке, в регистрах процессора и т.д.), а также все объекты в куче, на которые они указывают. Все эти элементы помечаются как используемые. Затем мы перебираем все используемые элементы и помечаем все прочие объекты, на которые они ссылаются. Этот процесс повторяется рекурсивно до тех пор, пока мы не перестаем находить новые используемые элементы.

Следующий просмотр сборки мусора утилизирует все элементы, не помеченные как живые, а также уплотняет все живые элементы, сдвигая их в начало кучи. Очевидно, что затем сборщику мусора придется поменять все значения указателей, используемых в программе; в связи с этим на время сборки мусора все остальные процессы приостанавливаются.

### **16. Распределение памяти. Адреса времени компиляции.**

К адресам времени компиляции относятся: адреса процедур, меток, другие любые адреса в исполняемом коде, а также статические переменные и массивы. При компиляции вызова процедуры или goto на метку просто генерируется переход (или call) на метку, объявленную в таблице символов. При неявных переходах (например, оператор break или continue в цикле) создаются метки для начала и конца цикла, на них и происходит переход.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

В общем случае, в процессе компиляции адреса переменных неизвестны. Укажем некоторые причины, почему это происходит.

- Во время выполнения программы расположение стекового фрейма, который соответствует конкретной функции или процедуре, зависит от порядка вызова функций (процедур).
- В процессе компиляции значение индексов массива обычно неизвестно и будет вычисляться при выполнении программы.
- Доступ к некоторым переменным осуществляется посредством указателей, значения которых в процессе компиляции неизвестны.
- Хотя в процессе компиляции адреса неизвестны, часть информации о них обычно имеется. Например, известны следующие параметры.
- Смещение простого значения относительно основания стекового фрейма.
- Смещение начала массива относительно основания стекового фрейма.
- Статическая глубина функции, в которой объявлена переменная. Статическая глубина (пункт 3) относится к языкам Pascal и Ada (в С такого понятия нет).

В языке С адрес простой переменной в процессе компиляции представляет собой смещение по отношению к основанию стекового фрейма. Это же относится и к полю записи, так как поля записи всегда запоминаются последовательно, и предполагается, что объем требуемой памяти для каждого из полей известен. Для языка Pascal или Ada адрес времени компиляции простой переменной или поля записи будет состоять из пары: (*номер уровня, оффсет*)

Здесь *номер уровня* — номер статического уровня функции или процедуры, в котором была объявлена переменная или запись, а термин "оффсет" употребляется с тем же значением, что и в языке С (смещение от начала фрейма).

Для массивов со статическими границами (значение границ известно в процессе компиляции) адрес элемента массива, в зависимости от применяемого языка, можно также выразить через номер уровня и оффсет или просто через оффсет. Смещение элемента массива по отношению к основанию стекового фрейма состоит из двух частей.

- Смещение начала массива по отношению к основанию стекового фрейма.

- Смещение элемента массива по отношению к началу массива.

Для массивов со статическими границами значение первой части известно в процессе компиляции, а второй, в общем случае, — нет, поскольку, повторимся, в процессе компиляции обычно неизвестно значение индексов массива.

При нахождении адресов элементов массива часть вычислений осуществляется во время выполнения программы с использованием информации, известной при компиляции. Как будет показано далее, объем вычислений зависит от размерности массива. Проиллюстрируем сказанное с помощью следующего примера на языке Pascal.

Рассмотрим объявление массива.

```
var table: array [1..10,1..20] of integer
```

Элементы массива обычно записывают построчно или, точнее, согласно лексикографическому порядку индексов. Например, значения элементов приведенной таблицы будут занесены в память в следующем порядке.

```
table [1,1],      table [1,2],.. table [1,20],  
table [2,1],      table [2,2], table [2,20],  
table [10,1],     table [10,2] table [10,20]
```

Адрес конкретного элемента массива вычисляется как смещение от адреса первого элемента массива.

$\text{адрес}(\text{table}[i, j]) = \text{адрес}(\text{table}[1,1]) + (U2 - I2 + 1)^* (i - I1) + (j - I2)$

Здесь  $I1$ , и  $U1$ , — нижняя и верхняя границы первого измерения и т.д., а каждый элемент массива предполагается размером в одну ячейку памяти. В приведенном выше примере нижние границы в каждом случае равны 1, а верхние — 10 и 20 соответственно. Для трехмерного массива arr3, объявленного как Var Arr: array [1..10, 1..10, 1..10] of integer;

общая формула для адреса элемента массива arr3 [x,y,z] имеет следующий вид.

$\text{офсет}(\text{arr}[i,j,k]) = \text{адрес}(\text{arr}[1,1,1]) + (U2 - I2 + 1)^* (U3 - I3 + 1)^* (i - I1) + (U3 - I3 + 1)^* (j - I2) + (k - I3)$

Из приведенной выше формулы для нахождения смещения элемента массива относительно адреса первого элемента массива понятно, что вычисления становятся достаточно простыми, если известны шаги по индексам. Например, для арг адрес элемента arr[i,j,k] выражается следующим образом:

$\text{адрес}(\text{arr}[i,j,k]) = \text{адрес}(\text{arr}[1,1,1]) + (U2 - I2 + 1)^* (U3 - I3 + 1)^* (i - I1) + (U3 - I3 + 1)^* (j - I2) + (k - I3)$

S1, S2, S3 — шаги по соответствующим индексам, равные следующему.

$$\begin{aligned} & (u_2 - l_2 + 1)^* (u_3 - l_3 + 1) \\ & (u_3 - l_3 + 1) \\ & 1 \end{aligned}$$

Для языков, в которых границы массива известны во время компиляции, значения шагов по индексам могут вычисляться сразу же (во время компиляции), что сокращает количество вычислений времени выполнения программы при каждом обращении к массиву. В то же время, дальше упростить приведенную формулу уже невозможно, поскольку разность ( $i - l_1$ ), в общем случае, во время компиляции неизвестна. Для языков с динамическими границами (до выполнения программы они неизвестны) шаги по индексам можно найти после объявления массива и занесения его в стек, что опять же уменьшит количество вычислений, выполняемых при каждом обращении к массиву. Хотя значения шагов по индексам в процессе компиляции могут быть неизвестны, практически всегда будет известен объем памяти, которую будут занимать шаги по индексам, и память для них может быть выделена в процессе компиляции. В то же время память для самих элементов массива может выделяться только при выполнении программы, поскольку при компиляции значения границ могут быть неизвестны.

Для рассмотрения динамических массивов (массивов с динамическими границами) требуется более общая модель стека времени выполнения, чем рассмотренная ранее. В общем случае неизвестно расположение начала массива в стековом фрейме. Поэтому каждый стековый фрейм удобно разбить на две части: статическую часть, в которой содержатся значения, известные во время компиляции, и динамическую часть, содержащую значения, неизвестные в процессе компиляции. Все значения динамической части можно будет получить (с помощью указателей) из значений статической части. Следовательно, в статической части фрейма будут содержаться следующие значения.

- Все простые значения (типы `integer`, `float` и т.д.).
- Статические части массивов (границы, шаги по индексам, указатели на элементы массива).
- Статические части записей ( поля, размеры которых известны во время компиляции).
- Указатели на глобальные значения — хотя глобальные значения будут храниться не в стеке, а в куче.

С другой стороны, в динамической части фрейма будут находиться элементы массива. При использовании этой модели на практике даже элементы массива со статическими границами будут храниться в динамической части фрейма. Описанная более общая модель стекового фрейма изображена на рис. 7.10.

В этой модели для доступа к элементам массива (по сравнению с доступом к элементам, не входящим в массив) необходимы дополнительный указатель и оффсет. Значение номера уровня фрейма дает первый указатель с дисплея. К этому добавляется оффсет указателя (в статической части массива) относительно начала массива, кроме того, во время выполнения

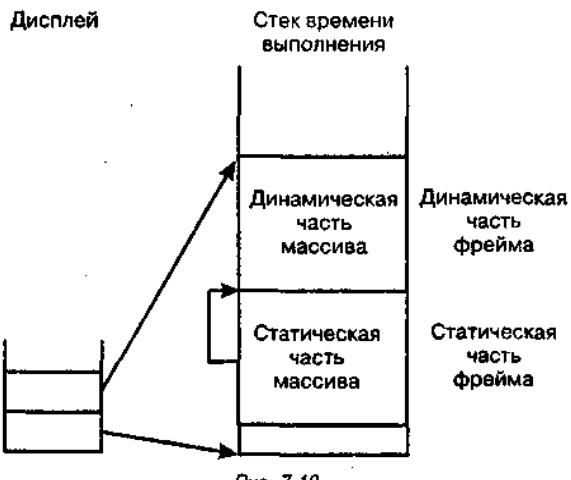


Рис. 7.10.

программы данный указатель увеличивается, чтобы представлять адрес конкретного элемента массива

В процессе компиляции адрес массива в целом — это просто уровень и оффсет, соответствующий началу статической части массива. Для нахождения адреса во время выполнения требуются вычисления, общий вид которых приведен выше. Очевидно, что доступ к элементам массива занимает много времени, в особенности для многомерных массивов. Это время можно уменьшить, если производить вычисление шагов по индексам только один раз. Для массивов с динамическими границами (в отличие от массивов со статическими границами) при выполнении программы время также тратится на каждое обращение к дополнительному указателю.

## **17. Организация памяти во время выполнения. Области данных при статическом и динамическом распределении памяти.**

**Память** — это один из самых важных ресурсов компьютера. Так как современные языки программирования не обязывают программиста работать напрямую с физическими ячейками памяти, на компилятор языка программирования возлагается ответственность за обеспечение доступа к физической памяти, ее распределение и утилизацию. В качестве ресурса могут выступать самые разные логические и физические единицы: обычные переменные примитивного типа, массивы, структуры, объекты, файлы и т.д. Со всеми этими объектами необходимо работать и, следовательно, обеспечить выделение памяти под связанные с ними переменные в программах.

Для этого компилятор должен последовательно выполнить следующие задачи:

- выделить память под переменную;
- инициализировать выделенную память
- некоторым начальным значением;
- предоставить программисту возможность использования этой памяти;
- как только память перестает использоваться, необходимо ее освободить;
- наконец, необходимо обеспечить возможность последующего повторного использования освобожденной памяти.

С точки зрения программиста, описанная выше схема кажется очень простой. Тем не менее, аккуратно реализовать эти действия в компиляторе и добиться корректной работы программ, использующих этот механизм, достаточно сложно из-за различных проблем. Самая большая неприятность управления памятью заключается в том, что память не бесконечна и потому приходится постоянно учитывать возможность исчерпания свободной памяти.

С точки зрения программиста, память становится свободной как только выполняется оператор явного освобождения памяти (`free/delete`) или в момент окончания времени жизни последней переменной, использующей данную область памяти. Эти операции, делающие структуру данных логически недоступной, называются уничтожением памяти. Затем освобожденную память необходимо вернуть системе

как свободную — утилизировать. Операции уничтожения памяти и утилизации могут быть сильно разнесены по времени.

При создании компилятора необходимо различать два важных класса информации:

- **Статическая информация**, т.е. информация, известная во время компиляции
- **Динамическая информация**, т.е. сведения, неизвестные во время компиляции, но которые станут известны во время выполнения программы

Например, значения констант в строго типизированных языках известны уже во время компиляции, в то время как значения переменных в общем случае становятся известными уже только во время выполнения программы

В приложении к управлению памятью разделение всей информации на статическую и динамическую позволяет определить, каким механизмом распределения памяти необходимо пользоваться для той или переменной, структуры или процедуры. Например, размер памяти, необходимой под простые переменные, можно вычислить (и, соответственно, выделить необходимую память) уже во время компиляции, а вот память, запрашиваемую пользователем с размером, заданным с помощью переменной, придется выделять уже во время выполнения программы.

Можно выделить три основных метода управления памятью:

- Статическое распределение памяти
- Стековое распределение памяти
- Представление памяти в виде кучи (heap)

Статическое управление памятью представляет собой простейший способ распределения памяти и было достаточно распространено на заре развития языков программирования. Если структура языка позволяет обойтись статическим распределением памяти, то удается достичь максимальной эффективности программы в целом.

Стековое управление памятью. Его идея заключается в том, что при входе в блок или процедуру на вершине специального стека выделяется память, необходимая для размещения переменных, объявленных внутри этого блока. При выходе же из блока память снимается не "вразнобой", а всегда только с вершины стека. Понятно, что задачи утилизации и повторного использования становятся тривиальными, а проблемы уплотнения просто не существует.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Наконец, последним механизмом управления памятью является управление кучей. Куча - это блок памяти, части которого выделяются и освобождаются способом, не подчиняющимся какой-либо структуре. Куча требуется в тех языках, где выделение и освобождение памяти требуется в произвольных местах программы. Возникают серьезные проблемы выделения, утилизации, уплотнения и повторного использования памяти – это все возложено на самую сложную часть управления кучей – сборку мусора.

Адреса переменных, в общем случае, в процессе компиляции неизвестны, т.к.:

1. Во время выполнения программы расположение стекового фрейма зависит от порядка вызова функций, а это не всегда известно
2. Индекс массива обычно неизвестен и будет вычисляться при выполнении программы.
3. Доступ к некоторым переменным – через указатели, значение которых также неизвестно в процессе компиляции:
  1. Смещение простого значения относительно основания стекового фрейма
  2. Смещение основания массива относительно основания стекового фрейма

В принципе, для простой переменной в процессе компиляции адрес представляется как смещение к основанию стека. Это же относится и к полям записи и предполагает, что объем каждого из полей известен.

Для массива со статичными границами необходимо знать смещение начала массива по отношению к основанию стекового фрейма и смещение элементов стека по отношению к началу массива:

LINE array [l..n] of int;

Хотим найти: LINE [i] = 5;

ADDR(LINE) – адрес относительно начала стекового фрейма

ADDR(LINE[i]) = ADDR(LINE) + (i - l);

MATRIX array [l1..n1, l2..n2] of int;

Хотим найти: MATRIX [i, j] = 5;

ADDR(MATRIX) – адрес относительно начала стекового фрейма

ADDR(MATRIX [i, j]) = ADDR(MATRIX) + (i - l1)(n2 - l2) + (j - l2)  
= ADDR(MATRIX) + i(n2 - l2) + j - l1(n2 - l2) - l2;

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Выделим  $-l1(n2 - l2) - l2$  – в константу с, а  $(n2 - l2)$  – в константу s:

$$\text{ADDR}(\text{MATRIX}[i, j]) = \text{ADDR}(\text{MATRIX}) + i*s + j + c$$

Объемный массив:

MATRIX array [l1..n1, l2..n2, l3..n3] of int;

Хотим найти: MATRIX [i, j, k] = 5;

ADDR(MATRIX) – адрес относительно начала стекового фрейма

$$\text{ADDR}(\text{MATRIX}[i, j, k]) = \text{ADDR}(\text{MATRIX}) + (i - l1)(n2 - l2)(n3 - l3) + (j - l2)(n3 - l3) + (k - l3)$$

Аналогично предыдущему, можно выделить константы.

При динамическом массиве – массиве с динамическими границами – потребуется более обобщенная модель стека выполнения: выделим 2 части памяти: стековая статическая часть и стековая динамическая часть. Все значения динамической части можно получить через указатель и значение статической части.

Следовательно, в статической части:

1. Известные значения, простые типы (float, int ...)
2. Границы, шаги, указатели на элементы массива, статические записи
3. Указатель на глобальные значения

В динамической части будут находиться элементы массива (элементы статич. массива). Для доступа к элементам необходимо знать смещение относительно динамической части.

## **18. Генерация кода. Генерация кода на примере одного из операторов Паскаля.**

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

На этапе генерации кода происходит замена операторов языка высокого уровня инструкциями ассемблера, а затем последовательностями машинных команд. Результат преобразования исходного текста программы записывается в виде двоичного файла (его называют объектным модулем) с расширением ".obj". На этой стадии внутренний код, сгенерированный семантическим анализатором, заменяется на объектный.

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

В процессе трансляции программы используется промежуточное представление (ПП) программы, предназначенное для эффективной генерации кода и проведения различных оптимизаций программы. Сама форма ПП зависит от целей его использования. Наиболее часто используемыми формами ПП является ориентированный граф (иногда абстрактное синтаксическое дерево), тройки, четверки, префиксная или постфиксная запись, атрибутированное абстрактное дерево.

#### **Представление в виде ориентированного графа**

Простейшей формой промежуточного представления является синтаксическое дерево программы. Более полную информацию о входной программе дает ориентированный ациклический граф (ОАГ), в котором в одну вершину объединены вершины синтаксического дерева, представляющие одни и те же константы, переменные и общие подвыражения. На рис. 1 приведены оба ПП программы.

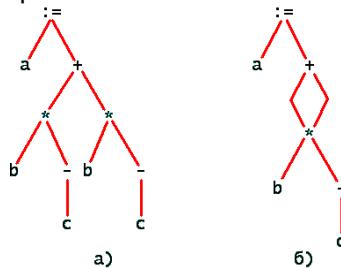


Рис. 1. Промежуточные представления программы.

#### **Трехадресный код**

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

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

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

$t1:=y^*z$

$t2:=x+t1,$

где  $t1$  и  $t2$  - имена, сгенерированные компилятором.

В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор  $if A>B then S1 else S2$  может быть представлен следующим кодом:

$t:=A-B$

$JGT\ t,S2$

.....

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

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

$t1 := -c$	$t1 := -c$
$t2 := b * t1$	$t2 := b * t1$
$t3 := -c$	$t5 := t2 + t2$
$t4 := b * t3$	$a := t5$
$t5 := t2 + t1$	
$a := t5$	
a)	б)

Рис. 2.

Представления синтаксического дерева и графа рис. 1 в виде трехадресного кода дано на рис. 2. а) и 2. б), соответственно.

Трехадресный код - это абстрактная форма промежуточного кода. В реализации трехадресный код может быть представлен записями с полями для операции и operandов.

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Рассмотрим три реализации трехадресного кода: четверки, тройки и косвенные тройки.

**Четверка** - это запись с четырьмя полями, которые будем называть op, arg1, arg2 и result. Поле op содержит код операции. В операторах с унарными операциями типа  $x:=-y$  или  $x:=y$  arg2 не используется. В некоторых операциях (типа "передать параметр") могут не использоваться ни arg2, ни result. Условные и безусловные переходы помещают в result метку перехода. Обычно содержимое полей arg1, arg2 и result - это указатели на входы таблицы символов для имен, представляемых этими полями. Временные имена вносятся в таблицу символов по мере их генерации.

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

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

### **Линеаризованные представления**

В качестве промежуточных представлений весьма распространены линеаризованные представления. Линеаризованное представление позволяет относительно легко хранить промежуточное представление на внешней памяти и обрабатывать его в порядке чтения. Самая распространенная форма линеаризованного представления - это запись дерева либо в порядке его обхода снизу-вверх (постфиксная запись, или обратной польской), либо в порядке обхода его сверху-вниз (префиксная запись, или прямой польской).

Таким образом, постфиксная запись (обратная польская) - это список вершин дерева, в котором каждая вершина следует непосредственно за своими потомками. Дерево на рис. 1 в постфиксной записи может быть представлено следующим образом: a b c - \* b c \* + :=

В постфиксной записи вершины синтаксического дерева явно не присутствуют. Они могут быть восстановлены из порядка,

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

в котором следуют вершины и из числа операндов соответствующих операций. Восстановление вершин аналогично вычислению выражения в постфиксной записи с использованием стека.

В префиксной записи сначала указывается операция, а затем ее operandы. Например, для приведенного выше выражения имеем := a + \* b - c \* b - c

### **Организация информации в генераторе кода**

Чисто синтаксическое дерево несет только информацию о структуре программы. На самом деле в процессе генерации кода требуется также информация о переменных (например, их адреса), процедурах (также адреса, уровни), метках и т.д. Для представления этой информации возможны различные решения. Наиболее распространены два. 1) всю эту информацию можно хранить в таблицах генератора кода; 2) информация хранится в вершинах дерева с соответствующими указателями.

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

### **Уровень промежуточного представления**

Промежуточное представление программы может в различной степени быть близким либо к исходной программе, либо к машине. Например, промежуточное представление может содержать адреса переменных, и тогда оно требует обработки перед перенесением на другую машину. С другой стороны, промежуточное представление может содержать раздел описаний программы, и тогда информацию об адресах можно извлечь из обработки описаний. В то же время ясно, что первое более эффективно, чем второе.

Операторы управления в промежуточном представлении могут быть представлены в исходном виде (в виде операторов языка if, for, while и т.д.), а могут содержаться в виде переходов. В первом случае некоторая информация может быть извлечена из самой структуры (например, для оператора for - информация о переменной цикла, которую, может быть, разумно хранить на регистре, для оператора case - информация о таблице меток и т.д.). Во втором случае возможно представление проще и унифицированней.

Некоторые формы промежуточного представления лучше годятся для различного рода оптимизаций (например, ориентированные графы), некоторые - хуже (например, префиксная запись для этого плохо подходит).

Генерация кода для операторов READ и WRITE языка ПАСКАЛЬ.

Напомним формат вывода и ввода для языка Паскаль, например: write[ln]('File has ','lines and ','words');

Естественно, мы должны определить некоторые соглашения по вызову таких функций - основной сложностью является то, что функция принимает переменное количество аргументов. Так как гораздо проще реализовать усложнённую функцию печати сообщений, чем усложнять компилятор, договоримся, что мы будем дополнительно передавать в функцию количество её аргументов, но непосредственно перед вызовом, чтобы вызываемая функция могла их вообще получить. Договоримся, что будем передавать каждый аргумент двумя фактическими параметрами: адресом аргумента и ярлыком типа аргумента. Независимо от операции WRITE или READ фактически списки аргументов не отличаются. Ещё условимся, что вызываемая функция очищает стек.

Рассмотрим генерацию кода для приведённого оператора:

str_1 db 9,"File has ";	Определим в сегменте данных три строки в формате Паскаль
str_2 db 10,"lines and"	
str_3 db 5,"words"	
...	
push offset str_3 push STRING	Адрес строки и ярлык строки
push offset w push INTEGER	Адрес целого числа, опции форматирования можно добавить к ярлыку
push offset str_2 push STRING	Адрес строки и ярлык
push offset l push FLOAT	Адрес плавающего числа и ярлык - пусть число строк будет плавающим
push offset str_1 push STRING	Адрес строки и ярлык
push 5	Число параметров
call _writeln_	Собственно вызов функции
...	

Генерация кода для условного оператора IF, вложенного оператора IF языка ПАСКАЛЬ.

Для генерации кода для любых вложенных операторов управления нам нужны дополнительные структуры. Условно

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ  
 назовём такую структуру стеком вложенных операторов.  
 Условимся, что в стеке будут храниться некие структуры  
 одинакового вида. Структура будет как минимум содержать ярлык  
 управляющей структуры и некоторые необходимые данные. Для  
 начала условимся, что будет как минимум ярлык - целое число.  
 Рассмотрим детально процесс генерации кода.

if (условие) then	test bx,bx jne метка_XXX	Генерируем код с уникальным именем метки, имя этой метки вместе с ярлыком _IF_ ложим на стек
... else ...	jmp метка_YYY метка_XXX:	Здесь мы достаём из стека ярлык _IF_ вместе с сгенерированным ранее именем метки, вставляем этую метку в ассемблеровский листинг, затем ложим на стек новую уникальную метку YYY с ярлыком _ELSE_
... ;	метка_YYY:	Достаём из стека ярлык _ELSE_ с меткой, которую вставляем в листинг

Дополнительные комментарии к примеру вряд ли  
 требуются. Ясно, что стек управляющих операторов позволяет  
 обрабатывать сколь угодно вложенные структуры.

### **Генерация кода для оператора цикла FOR, вложенного оператора FOR языка ПАСКАЛЬ.**

Продолжаем обсуждение, но уже рассматривая цикл FOR:

FOR var:=Value1	push dx mov dx, Value1	Инициализируем индексную переменную - регистр dx, сохраняя прежнюю в стеке
TO DOWNTO Value2	loop_XXX: cmp dx,Value2 ja jb loop_exit_XXX	Здесь мы генерируем уникальную метку (на самом деле только суффикс XXX), которую мы ложим на стек вместе с ярлыком _FOR_ и пометкой, увеличивается или уменьшается в цикле индексная переменная
DO ...	... ;тело цикла	в dx - индексная переменная
... ;	inc dec dx	Вынимаем из стека

	jmp loop_XXX loop_exit_XXX: pop dx	управляющих операторов ярлык _FOR_, генерируем инкремент или декремент индексной переменной, генерируем переход к началу цикла, метку окончания цикла и восстанавливаем регистр, содержащий индексную переменную
--	--	--

### Генерация кода для операторов цикла WHILE, REPEAT языка ПАСКАЛЬ.

Закончим обсуждение генерации кода примером для условных циклов. Первым будет цикл WHILE.

WHILE (условие) DO	w_loop_XXX: ...;код вычисления ...;условия test bx,bx jz w_loop_exit_XXX	Здесь мы генерируем уникальную метку (реально только уникальный суффикс), которую мы ложим на стек вместе с ярлыком _WHILE_, заметим, что между метками располагается не показанный здесь код вычисления условия, который возвращает булево значение на вершине стека
... ;	jmp w_loop_XXX w_loop_exit_XXX:	Вынимаем из стека управляющих операторов ярлык _WHILE_, генерируем переход к началу цикла в метку окончания цикла

Аналогичный пример для REPEAT:

REPEAT ...	r_loop_XXX: ...	Здесь мы генерируем уникальную метку, которую мы ложим на стек вместе с ярлыком _REPEAT_
UNTIL(условие);	...;код вычисления ...;условия test bx,bx jnz r_loop_XXX	Вынимаем из стека управляющих операторов ярлык _WHILE_, генерируем код вычисления условия, который возвращает булево значение на вершине стека и генерируем переход к началу

## 19. Свойства КС-грамматик. Лемма подкачки.

**Контекстно-свободная грамматика** (КС-грамматика, бесконтекстная грамматика) — частный случай формальной грамматики (тип 2 по иерархии Хомского), у которой левые части всех продукции являются нетерминалами. Смысл термина «контекстно-свободная» заключается в том, что возможность применить продукцию к нетерминалу, в отличие от общего случая грамматики Хомского, не зависит от контекста этого нетерминала. Язык, который может быть задан КС-грамматикой, называется контекстно-свободным языком или КС-языком. Следует заметить, что по сути КС-грамматика — другая форма БНФ. КС-грамматики находят большое применение в информатике. Ими задаётся грамматическая структура большинства языков программирования, структурированных данных и т.д. (см. грамматический анализ). Класс контекстно-свободных языков допускает распознавание с помощью недетерминированного КА со стековой (или магазинной) памятью.

### Типы КС грамматик:

- LL-грамматика
- LALR-грамматика
- LR-грамматика
- SLR-грамматика

Контекстно-зависимые языки - последний класс языков, которые можно эффективно распознать с помощью ЭВМ. Специалисты скажут, что они допускаются двусторонними недетерминированными линейно ограниченными автоматами. Для допуска цепочек языка без ограничений в общем случае требуется универсальный вычислитель (машина Тьюринга, машина с неогр. числом регистров и т.п.).

Если в КС-грамматике G существует нетерминал A, такой, что  $A \Rightarrow^* a_1 A a_2 | a_1 a_2 \neq 0$ , то говорят, что грамматика G содержит **самовложение**. Символ  $\Rightarrow^*$  обозначает существование вывода одной цепочки из другой (\* означает нуль или более применений отношения  $\Rightarrow$ ). Примеры грамм, содерж самовложения:

(G1)  $S \Rightarrow aSb$

$S \Rightarrow e$

(G2)  $S \Rightarrow \text{begin } A \text{ end}$

$S \Rightarrow \epsilon \quad A \Rightarrow [ S ]$

Теоретически любая КС-грамматика, не содержащая самовложений эквивалентна регулярной грамматике. Именно

самовложением позволяют отличать регулярные языки от нерегулярных.

### Лемма подкачки (Огдена)

Для любого КС-языка существует такая константа  $k$ , что  $|Z| > k$ , где  $Z$ -предложение языка  $L$ , и можно записать  $Z = uwxu$ ,  $u \neq \lambda$ ,  $|u| \leq k$  и  $|x| \neq 0$ , то строка  $uw^ixu$  ( $i \geq 0$ )  $\in L$ .

С пом. этой леммы можно показать, что определ. языки не являются КС языками:

$\{LL | L \in \{0, 1\}^*\}$  для него существует константа  $k$ .

Пусть  $L$  состоит из  $k$  нулей и  $k$  единиц  $\{0\dots 0, 1\dots 1\}$ , тогда  $Z = LL = 0\dots 0, 1\dots 1, 0\dots 0, 1\dots 1$ .

$$Z = uwxu. \text{ т.к. } |Z| > k, \text{ а } |uw| < k$$

Возможны два варианта:

1.  $u \neq \lambda$  и  $x$  находятся в первой половине  $Z$  или во второй, т.е. состоят из нулей или единиц.

2.  $u \neq \lambda$  и  $x$  содержат единички из первой половины строки.

Рассм. 1-ый случай. Так как длина  $uw \leq k$  и  $u \neq \lambda$ , то  $uw$  ( $i=0$ ) образуется исключением из  $Z$  0 и 1 первой половины, или второй, но не из обеих половин. Таким образом,  $uw \notin L$ .

Рассмотрим второй случай.  $uw$  ( $i=0$ ) исключает из  $Z$  единицы из первой части( $L$ ) и нули из второй, т.е. из середины слова  $Z$ . Полученное слово не принадлежит языку.

## 20. Автоматы с магазинной памятью.

Автомат с магазинной памятью (МП автомат) эквивалентен конечному автомату, к которому добавлена память (стек) магазинного типа. Основная особенность магазинной памяти состоит в том, что символы можно помещать в магазин и удалять из него по одному, причем удаляемый символ – это всегда тот, который был помещен в магазин последним. Магазин изображен на рис.1а. на дне магазина находится символ  $\square$ , а наверху символ  $C$ . символы расположены в том порядке, в каком они поступали в магазин. Сначала поступил символ  $\square$ , затем  $A$ , затем  $B$ , и наконец  $C$ . Если втолкнуть в магазин символ  $D$ , то магазин будет выглядеть , как показано на рис.1б. Если же наоборот, вытолкнуть из магазина верхний символ  $C$ , то верхним символом окажется  $B$ , и магазин будет выглядеть как показано на рис.1в. В обоих случаях изменениям подвергается только верх магазина, а остальные символы остаются неизменными. Символ  $\square$  - это специальный символ, который помечает дно магазина (начало) и называется маркером дна. Он используется как метка дна и никогда не выталкивается из магазина. Так, если  $\square$  - верхний символ магазина, как на рис.1г, то говорят, что магазин пуст.

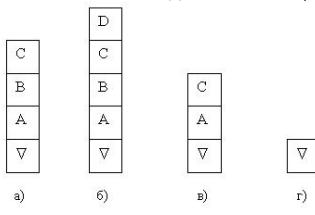


Рис. 1

Магазин на рис.1 можно изобразить в виде цепочки одним из следующих способов:

1. С В А □
2. А В С

МП автомат может находиться в одном из конечного числа состояний и имеет магазин, куда он может помещать и откуда может извлекать информацию. Как и в случае конечного автомата, обработка входной цепочки осуществляется за ряд мелких шагов. На каждом шаге действия автомата конфигурация его памяти может изменяться за счет перехода в новое состояние, а также втальчивания символа в магазин или выталкивания из него. Однако, в отличие от конечного автомата, МП автомат может обрабатывать один входной символ в течение нескольких шагов. На каждом шаге обработки управляющее устройство автомата решает, пора ли закончить обработку текущего входного символа и получить, если это так, новый входной символ или продолжить обработку текущего символа на следующем шаге.

Каждый шаг процесса обработки задается множеством правил, использующих информацию трех видов:

1. состояние,
2. верхний символ магазина,
3. текущий входной символ.

Это множество правил называется управляющим устройством или механизмом управления. В зависимости от получаемой информации, управляющее устройство выбирает либо выход из процесса (прекращает обработку), либо переход в новое состояние. Переход состоит из трех операций: над магазином, над состоянием, над входом.

#### Операции над магазином

1. втолкнуть в магазин определенный магазинный символ.
2. Вытолкнуть верхний символ магазина.
3. Оставить магазин без изменений.

#### Операция над состоянием

1. перейти в заданное новое состояние.

Операции над входом

1. перейти к следующему входному символу и сделать его текущим входным символом.
2. Оставить данный входной символ текущим, то есть держать его до следующего шага.

Будем считать, что цепочка, подаваемая на вход автомата, имеет концевой маркер. Пусть это будет символ  $-|$ . Обработку входной цепочки МП автомат начинает в некотором выделенном состоянии при определенном содержимом магазина, а текущим входным символом является первый символ входной цепочки. Затем автомат выполняет операции, задаваемые его управляющим устройством. Если происходит выход из процесса, обработка прекращается. Если происходит переход, то он дает новый верхний магазинный символ, новый текущий символ, автомат переходит в новое состояние и управляющее устройство определяет новое действие, которое нужно произвести.

Чтобы управляющие правила имели смысл, автомат не должен требовать следующего входного символа, если текущим символом является концевой маркер, и не должен выталкивать символ из магазина, если это маркер дна. Поскольку маркер дна может находиться исключительно на дне магазина, автомат не должен также выталкивать его в магазин.

**МП автомат задается следующими пятью объектами:**

1. Конечным множеством входных символов, в которое и входит и концевой маркер.
2. Конечным множеством магазинных символов, включающим маркер дна.
3. Конечным множеством состояний, включающим начальное состояние.
4. Управляющим устройством, которое каждой комбинации входного символа, магазинного символа и состояния ставит в соответствие выход или переход. Переход в отличие от выхода заключается в выполнении операций над магазином, состоянием и входом. Операции, которые запрашивали бы входной символ после концевого маркера или выталкивали из магазина, а также вталкивали в него маркер дна, исключаются.
5. Начальным содержимым магазина, которое представляет собой (при условии, что верхний символ считается расположенным справа) маркер дна, за которым следует (возможно, пустая) цепочка других магазинных символов.

МП автомат называется **МП распознавателем**, если у него два выхода – **ДОПУСТИТЬ** и **ОТВЕРГНУТЬ**. Говорят, что цепочка

## ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

символов входного алфавита (исключая концевой маркер) допускается распознавателем, если под действием этой цепочки с концевым маркером автомат, начавший работу в своем начальном состоянии и с начальным содержимым магазина, делает ряд переходов, приводящих к выходу ДОПУСТИТЬ. В противном случае цепочка отвергается.

При описании переходов МП автомата будем обозначать действия автомата словами **ВЫТОЛКНУТЬ**, **ВТОЛКНУТЬ**, **СОСТОЯНИЕ**, **СДВИГ** и **ДЕРЖАТЬ**, причем:

- ВЫТОЛКНУТЬ означает вытолкнуть верхний символ магазина;
- ВТОЛКНУТЬ (A), где A – магазинный символ, означает втолкнуть символ A в магазин;
- СОСТОЯНИЕ (s), где s – состояние, означает, что следующим состоянием становится s;
- СДВИГ означает, что текущим входным символом становится следующий входной символ;
- ДЕРЖАТЬ означает, что текущий входной символ надо держать до следующего шага, то есть оставить его текущим.

Рассмотрим применение МП распознавателя к проблеме скобок. Каждый раз, когда встречается левая скобка, в магазин будет вталкиваться символ A. Когда будет обнаружена соответствующая правая скобка, символ A будет выталкиваться из магазина. Цепочка отвергается, если на входе остаются правые скобки, а магазин пуст (то есть во входной цепочке есть лишние правые скобки) или если цепочка прочитана до конца, а в магазине остаются символы A (то есть входная цепочка содержит лишние левые скобки). Цепочка допускается, если к моменту прочтения входной цепочки до конца магазин опустошается.

### Полное определение:

1. Входное множество  $\{ (, ), - | \}$ .
2. Множество магазинных символов  $\{ A, \square \}$ .
3. Множество состояний  $\{ s \}$ , где s – начальное состояние.
4. Переходы:  
 $(, A, s = \text{ВТОЛКНУТЬ} (A), \text{СОСТОЯНИЕ} (s), \text{СДВИГ} (, \square, s = \text{ВТОЛКНУТЬ} (A), \text{СОСТОЯНИЕ} (s), \text{СДВИГ} (, A, s = \text{ВЫТОЛКНУТЬ}, \text{СОСТОЯНИЕ} (s), \text{СДВИГ} (, \square, s = \text{ДЕРЖАТЬ} -|, A, s = \text{ДЕРЖАТЬ} -|, \square, s = \text{ДОПУСТИТЬ} ) )$

Здесь комбинации входного символа, магазинного символа и состояния расположены слева от знака равенства, а переходы справа от него.

### 5. начальное содержимое магазина □ .

Для того чтобы продемонстрировать работу автомата, покажем, как он обрабатывает цепочку (( ) ( ) ).

Шаг а	▽	[s]	(( ) ( )) -
Шаг б	▽A	[s]	( ) ( ) ) -
Шаг в	▽AA	[s]	) ( ) ) -
Шаг г	▽A	[s]	( ) ) ) -
Шаг д	▽AA	[s]	) ) ) -
Шаг е	▽A	[s]	) ) -
Шаг ж	▽	[s]	-
Шаг з	допустить		

В этом представлении магазин изображен слева, состояние в середине, а необработанная часть входной цепочки – справа. Эта часть входной цепочки включает текущий входной символ и символы, которые следуют после него.

Управляющее устройство этого автомата с одним состоянием можно представить в виде управляющей таблицы, которая показана ниже, где показаны действия автомата для каждого сочетания входного символа и верхнего символа магазина. Столбцы таблицы обозначены входными символами, а на пересечении строк и столбцов обозначены соответствующие им действия. Так как этот конкретный автомат имеет лишь одно состояние, информация о состоянии опущена.

## VI. ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

### 1. Рекурсия и циклы в Лиспе

Циклические вычисления в Lispе выполняются или с помощью итерационных (циклических) предложений или рекурсивно.

Рекурсия в Lisp основана на математической теории рекурсивных функций. Рекурсия хорошо подходит для работы со списками, так как сами списки могут состоять из подсписков. То есть иметь рекурсивное строение. Для обработки рекурсивных структур совершенно естественно использование рекурсивных процедур.

Списки можно определить с помощью следующих правил Бэкуса-Наура:

**список ::= nil ; список либо пуст, либо это**

**список ::= (голова . хвост) ; точечная пара, хвост которой является списком**

**голова ::= атом**

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

**голова ::= список ;рекурсия "в глубину"**

**хвост ::= список ;рекурсия "в ширину"**

## РЕКУРСИЯ

Функция называется **рекурсивной**, если ее определение прямо или косвенно (через другие функции) содержит обращение к самой себе. Рекурсия основной и самый эффективный способ организации повторяющихся вычислений в функциональном программировании.

### ПРИМЕР

Определим функцию MEMBER :

```
(defun member(i l) (cond ((null l) nil) ((eql (car l) i) l) (t  
(member i (cdr l)))))
```

### Правила записи рекурсивной функции:

1. Терминальная ветвь необходима. Без терминальной ветви рекурсивный вызов был бы бесконечным. Терминальная ветвь возвращает результат, который является базой для вычисления результатов рекурсивных вызовов.
2. После каждого вызова функции самой себя, мы должны приближаться к терминальной ветви. Всегда должна быть уверенность, что рекурсивные вызовы ведут к терминальной ветви.
3. Проследить вычисления в рекурсии чрезвычайно сложно. Очень трудно мысленно проследить за действием рекурсивных функций. Это практически невозможно для сложных функций. Т.о. мы должны уметь писать рекурсивные функции, без того чтобы представлять точно порядок вычисления.

### Как писать рекурсивные функции:

При написании рекурсивных функций мы должны планировать терминальные и рекурсивные ветви.

1. Планирование терминальной ветви. При написании рекурсивной функции мы должны решить, когда функция может вернуть значение без рекурсивного вызова.
2. Планирование рекурсивной ветви. В этом случае мы вызываем функцию рекурсивно с упрощенным аргументом и используем результат для расчета значения при текущем аргументе. Таким образом, мы должны решить:

- 1) Как упрощать аргумент, приближая его шаг за шагом к конечному значению.
- 2) Кроме этого необходимо построить форму, называемую рекурсивным отношением, которая связывает правильное значение текущего вызова со значением рекурсивного вызова.

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Например: (sumall n) и (sumall (- n 1)). Иногда просто найти это отношение, а если не получается надо выполнить следующую последовательность шагов:

- а). Определить значение некоторого простого вызова функции и ее соответствующего рекурсивного вызова.
- б). Определить соотношение между парой этих функций.

## ОБЩАЯ ФОРМА ОПРЕДЕЛЕНИЯ РЕКУРСИВНОЙ ФУНКЦИИ:

```
(defun <имя> <параметры>
  (cond
    (терминальная ветвь 1)
    (терминальная ветвь 2)
    .....
    (терминальная ветвь n)
    (рекурсивная ветвь 1)
    (рекурсивная ветвь 2)
    .....
    (рекурсивная ветвь n)
  )
)
```

## ЦИКЛЫ:

LOOP реализует бесконечный цикл (LOOP форма1 форма2...), в котором формы вычисляются до тех пор, пока не встретится явный оператор завершения RETURN.

DO - это самое общее циклическое предложение.

```
( DO (( var1 знач1 шаг1) ( var2 знач2 шаг2)....)
      ( условие - окончания форма11 форма12...)
      форма21
      форма22
      ...)
```

Порядок вычисления цикла DO:

1. Начальные значения локальных переменных.
2. Проверка условие окончания.
3. Выполнение форм окончания вычислений. *Значение последней формы – значение циклического предложения.*
4. Выполнение форм продолжения вычислений.
5. Изменение локальных переменных для следующего цикла.

DOTIMES можно использовать вместо DO, если надо повторить вычисления заданное число раз.

(DOTIMES (var num форма-return)  
(форма-тело))

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

*var – переменная цикла,*

*пнт – форма, определяющая число циклов,*

*форма-return – результат, который должен быть возвращен.*

1. Вычисляется пнт-форма, результат – целое число-count.
2. var меняется от 0 до count, каждый раз вычисляется форма-тело.
3. Последним – вычисляется форма-return.
4. Если форма-return отсутствует, возвращается nil.

DOLIST выполняет итерацию по каждому из элементов списка. Вначале dolist оценивает значение формы listform, которое должно привести к формированию списка. Затем осуществляется оценка тела цикла для каждого его элемента, при этом переменная var связывается на каждой итерации с очередным элементом. После того, как будет просмотрен весь список, оценивается значение формы resultform (должна представлять собой простую форму, и не может быть неявной progn), которая возвращается как результат всего вызова dolist. Во время оценки значения resultform управляющая переменная цикла var по-прежнему является связанной и имеет значение nil. Если resultform опущена, функция возвращает NIL.

(dolist (var list-form)  
      body-form)

*list-form – список элементов, длина которого определяет число циклов*

*body-form – тела цикла*

### Примеры:

- Будет печатать последовательность чисел, в конце - end.  
`* ( do (( x 1 ( + 1 x))) (( > x 10) ( print 'end)) ( print x))`
- Может одновременно изменяться значения нескольких переменных.  
`*(do ((x1(+1x))(y1(+2y))(z3));значение не меняется  
((>x10)(print  
'end))(princ"x=")(prin1x)(princ"y=")(prin1y)(princ"z=")(prin1z)(terpri))`
- Можно реализовать вложенные циклы:  
`(*do((x1(+1x))) ((>x10)) (do((y1(+2y)))((>y4)) (princ"x=") (prin1x)  
(princ"y=")(prin1y)(terpri)))`
- Функция double-list принимает список чисел и возвращает новый список, в котором каждое число удвоено.

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

```
(defun double-list (lis)
  (let ((newlist nil))
    (loop
      (cond ((null lis) (return newlist)))
      (setq newlist (append newlist (list (* 2
                                             (car lis))))))
      (setq lis (cdr lis))
      )
    )
)
```

- После печати a b c d '' (не забудьте о пробеле)  
(dolist (x '(a b c d)) (prin1 x) (princ " "))

## **2. Внутреннее представление списков в Лиспсе**

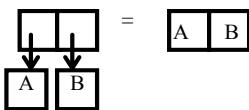
### СТРУКТУРА ПАМЯТИ.

Каждый атом занимает ячейку.

Списки, являются совокупностью атомов, и связываются вместе специальными элементами памяти, называемыми **списочными ячейками** или cons-ячейки.

Каждая списочная ячейка состоит из двух частей, полей. Первое поле - CAR, второе CDR. Поля типа списочная ячейка содержат указатели на другие ячейки, или nil.

Если обе ячейки содержат указатели на атомы, то можно записать:



Этому случаю соответствует структура (a.b) и называется точечная пара.

### ПРЕДСТАВЛЕНИЕ СПИСКОВ ЧЕРЕЗ СПИСОЧНУЮ ЯЧЕЙКУ

$$(a) = \boxed{a} \boxed{\text{nil}} = \boxed{a} \boxed{\backslash} \boxed{\text{nil}} = (a . \text{nil})$$

$\downarrow$   
a

Список из одного атома, Nil указывает на конец списка. Вместо nil пишут \.

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Список получается как операция(`cons 'a nil`). Список из двух элементов (`b a`)



Правое поле указывает на cdr списка (хвост). Левое поле, на саг списка (голову).

Каждому элементу списка соответствует списочная ячейка.

## ПРЕДСТАВЛЕНИЕ СПИСКОВ ЧЕРЕЗ ТОЧЕЧНЫЕ ПАРЫ

Любой список можно записать в точечной нотации.  
 $(a) \Leftrightarrow (a.\text{nil})$   
 $(a\ b\ c) \Leftrightarrow (a.(b.(c.\text{nil})))$

Выражение представленное в точечной нотации можно привести к списочной если cdr поле является списком.

$$\begin{aligned}(a.(b\ c)) &\Leftrightarrow (a\ b\ c) \\(a.(b.c)) &\Leftrightarrow (a\ b.c)\end{aligned}$$

## ФУНКЦИИ

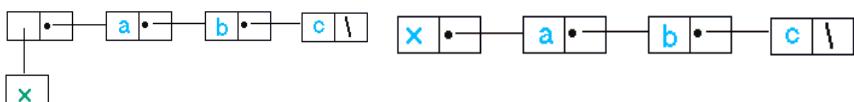
**CAR** – значение левого поля первой списочной ячейки

$$\begin{aligned}(\text{car } '(a\ (b\ c)\ d)) \\ a\end{aligned}$$

**CDR** – значение правого поля первой списочной ячейки.

$$\begin{aligned}(\text{cdr } '(a\ (b\ c)\ d)) \\ ((b\ c)\ d)\end{aligned}$$

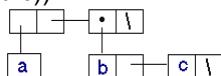
**CONS** создает новую списочную ячейку, саг-поле которой указывает на первый элемент, а cdr на второй (`cons 'x '(a b c)`).



## **LIST**

1. Создается списочная ячейка для каждого аргумента.
2. В car-поле ставится указатель на соответствующий элемент.
3. В cdr-поле ставится указатель на следующую списочную ячейку.

$$\begin{aligned}(\text{list } 'a\ '(b\ c)) \\ (a\ (b\ c))\end{aligned}$$



### 3. Декларативная и процедурная семантика Пролог-программ

В Прологе наиболее часто используются две семантические модели: декларативная и процедурная. Семантические модели предназначены для объяснения смысла программы.

В **декларативной модели** рассматриваются отношения, определенные в программе. Она определяет, является ли целевое утверждение истинным, исходя из данной программы, и если оно истинно, то для какой конкретизации переменных. Для этой модели порядок следования предложений в программе и условий в правиле не важен.

**Декларативная семантика** определяет, что должно быть результатом работы программы, не вдаваясь в подробности, как это достигается. Пусть задано

P:-Q, R.        где P, Q, R -термы.

Тогда с точки зрения декларативного смысла это предложение читается: " P-истинно, если Q R истинны. " Или " Из Q и R следует P." Т.е. определяются логические связи между головой предложения и целями в его теле. Таким образом, декларативный смысл программы определяет, является ли данная цель истинной (достижимой), и если - да, то при каких значениях переменных она достигается.

**Конкретизацией** I предложения C называется результат подстановки в него на место каждой переменной некоторого терма. Заметим, что это отличается от конкретизации переменной. Пример:

haschild( X ):-parent( X ,Y). Предложение C.

|              |

haschild(tom):-parent(tom,Y). Конкретизация I. Вариант 1. (X=tom)

haschild(bob):-parent(bob,ann). Конкретизация I. Вариант 2. (X=bob,Y=ann)

Пусть дана некоторая программа и цель G, тогда в соответствии с декларативной семантикой, можно утверждать, что: цель G истинна (т. е. достижима или логически следует из программы) тогда и только тогда, когда в программе существует предложение C, такое, что существует такая его (C) конкретизация I, что голова I совпадает с G и все цели в теле I истинны.

Пример

female(ann).  
C(I)      parent(ann, bob).  
C:  
            mother(ann):-parent(ann, Y), female(ann).

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

```
mother(X) :-parent(X, Y), female(Y).  
?- mother(ann).
```

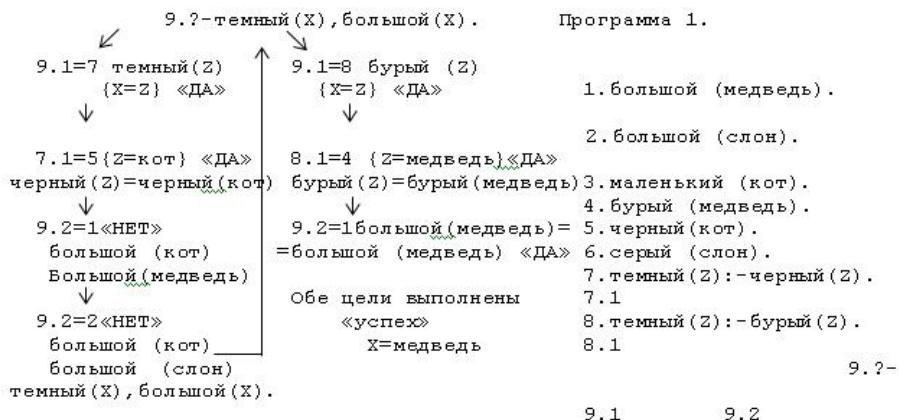
Это определение можно распространить на вопросы следующим образом. В общем случае вопрос - список целей, разделенных запятыми. Список целей называется истинным (достижимым), если все цели в этом списке истинны, достижимы, при одинаковых конкретизациях переменных. Запятая между целями означает конъюнкцию целей, и они должны быть все истинны.

Процедурная модель рассматривает правила как последовательность шагов, которые необходимо успешно выполнить для того, чтобы соблюдалось отношение, приведенное в заголовке правила. Это процедура достижения списка целей в контексте данной программы. Процедура выдает истинность или ложность списка целей и соответствующую конкретизацию переменных. Процедура осуществляет автоматический возврат для перебора различных вариантов.

Процедурная семантика (процедурный смысл) пролог - программы определяет, как пролог-программа отвечает на вопросы. Ответить на вопрос это значит удовлетворить цели. Поэтому процедурная семантика пролога - это процедура вычисления списка целей с учетом программы. Рассмотрим программу и на ее примере процедуру вычисления списка целей. Пример вычисления. Рассмотрим программу и на ее примере - процедуру вычисления списка целей.

Множество предложений, имеющих в заголовке предикат с одним и тем же именем и одинаковым количеством аргументов, трактуются как процедура. Для процедурной модели важен порядок, в котором записаны предложения и условия в предложениях. Поэтому порядок может повлиять на эффективность программы, неудачный порядок может даже привести к бесконечным рекурсивным вызовам.

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ



Таким образом, для вычисления целей потребовалось 7 сопоставлений и один откат.

Формальное описание процедуры вычисления целей. Пусть список целей  $G_1, G_2, \dots, G_m$ .

1. Если список целей пуст, вычисление дает успех, если нет, то выполняются пункт 2.

2. Берется первая цель  $G_1$  из списка. Пролог выбирает в базе данных, просматривая сначала первое предложение  $C$ ,

$C: H :- B_1, B_2, \dots, B_n$ . голова, которого, сопоставляется с целью  $G_1$ . Если такого предложения нет, то неудача. Если есть, то переменные конкретизируются и цель  $G_1$  заменяется на список целей  $B'_1, B'_2, \dots, B'_n$ , с конкретизированными значениями переменных.

3. Рассматривается рекурсивно через п.2 новый список целей.  $B'_1, B'_2, \dots, B'_n, G_2, \dots, G_m$ . Если  $C$  - факт, то новый список короче на одну цель. ( $n=0$ ) Если вычисление нового списка оканчивается успешно, то и исходный список целей выполняется успешно. Если нет, то новый список целей отбрасывается, снимается конкретизация переменных и происходит возврат к просмотру программы, но начиная с предложения следующего за предложением  $C$ . Описанный процесс возврата называется бэктрекингом. (backtracking).

Соотношение между процедурным и декларативным смыслом  
 Создавая пролог программы всегда надо помнить о процедурном и декларативном смысле. Декларативный смысл касается отношений, определенных в программе. Т.е. декларативный смысл определяет, что должно быть результатом программы. С другой стороны, процедурный смысл определяет, как этот

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

результат, может быть, достигнут, т.е., как реально отношения обрабатываются прологом.

Имея декларативно правильную программу, можно улучшить ее эффективность путем изменения порядка предложений и целей при сохранении ее декларативной правильности. Переупорядочивание - один из методов предотвращения зацикливания.

### 4. Отсечение. Графическая иллюстрация действия cut. Формальное описание действия отсечения

Все операторы CUT используются для уничтожения точек возврата, находящихся на стеке возвратов. Слово CUT: отмечает на стеке возвратов начало отсекаемой области. -CUT уничтожает все точки возврата, установленные после этой отметки. Конструкция выглядит так:

CUT: <действия> -CUT

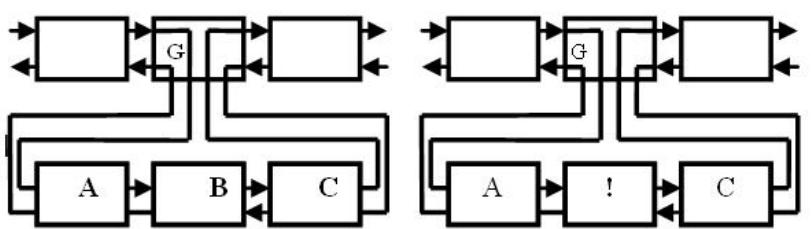
В процессе вычисления цели пролог проводит перебор вариантов, в соответствии с установленным порядком. Цели выполняются последовательно, одна за другой, а в случае неудачи происходит откат к предыдущей цели (backtracing)

Однако для повышения эффективности выполнения программы, часто требуется вмешаться в перебор, ограничить его, исключить некоторые цели. Для этой цели в пролог введена специальная конструкция cut - "отсечение", обозначаемая как "!"( Читается "cut").

Cut подсказывает прологу "закрепить" все решения предшествующие появлению его в предложении. Это означает, что если требуется бэктрекинг, то он приведет к неудаче (fail) без попытки поиска альтернатив.

Графическая иллюстрация действия cut.

Графически действие cut можно показать с помощью box-представления логического вывода. В обычном случае бэктрекинг для правила G:-A,B,C. выглядит следующим образом:



Т.е. поиск альтернатив производится для всех целей: G,A,B,C. Заменим цель B на cut.

#### ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Действие `cut` заключается в отмене поиска альтернатив для целей A,G, стоящих после "!".

Формальное описание действия отсечения. Рассмотрим предложение `H:-B1, B2,..., Bm, !,..., Bn.` Это предложение активизируется, когда некоторая цель G, будет сопоставляться с H. Тогда G называют целью-родителем. Если B1, B2,..., Bm, выполнены, а после `!`, например в `Bi, i>m`, произошла неудача и требуется выбрать альтернативные варианты, то для `B1, B2,..., Bm` такие альтернативы больше не рассматриваются и все выполнение окончится неудачей. Кроме этого G будет связана с головой H, и другие предложения процедуры во внимание не принимаются. Т.е. отсечение в теле предложения отбрасывает все предложения, расположенные после этого предложения. Формально действие отсечения описывается так: Пусть цель - родитель сопоставляется с головой предложения, в теле которого содержится отсечение. Когда при просмотре целей тела предложения встречается в качестве цели отсечение, то такая цель считается успешной и все альтернативы принятых решениям до отсечения отбрасываются и любая попытка найти новые альтернативы на промежутке между целью-родителем и `cut` оканчиваются неудачей. Процесс поиска возвращается к последнему выбору, сделанному перед сопоставлением цели родителя.

Т.е. отсечение в теле предложения отбрасывает все предложения, расположенные после этого предложения.

Формально действие отсечения можно описать еще так:

Пусть цель-родитель сопоставляется с головой предложения, в теле которого содержится отсечение. Когда при просмотре целей тела предложения встречается в качестве цели отсечение, то такая цель считается успешной, и все альтернативы принятых решениям до отсечения отбрасываются, и любая попытка найти новые альтернативы на промежутке между целью-родителем и `cut` оканчиваются неудачей. Процесс поиска возвращается к последнему выбору, сделанному перед сопоставлением цели родителя.

#### ПРИМЕР:

`max(X,Y,X):-`

`X>Y. /* если первое число больше второго,  
то первое число - максимум */`

`max(X,Y,Y):-`

`X<=Y./* если первое число меньше или равно  
второму, возьмем в качестве максимума  
второе число */`

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

В данной ситуации нам пригодится встроенный предикат, который по-английски называется **cut**, по-русски - отсечение, а в Прологе он обозначается восклицательным знаком "!". Этот предикат предназначен для ограничения пространства поиска, с целью повышения эффективности работы программ. Он всегда завершается успешно. После того, как до него дошла очередь, он устанавливает "забор", который не дает "откатиться назад", чтобы выбрать альтернативные решения для уже "сработавших" подцелей. То есть для тех, которые расположены левее отсечения. На цели, расположенные правее, отсечение не влияет. Кроме того, отсечение отбрасывает все предложения процедуры, расположенные после предложения, в котором находится отсечение.

С использованием отсечения наше решение будет еще короче:

```
max2(X,Y,X):-  
    X>Y,!/* если первое число больше второго,  
           то первое число - максимум */  
max2(_,Y,Y). /* в противном случае максимумом будет  
           второе число */
```

В случае, если сработает отсечение, а это возможно, только если окажется истинным условие  $X > Y$ , Пролог-система не будет рассматривать альтернативное второе предложение. Второе предложение "сработает" только в случае, если условие оказалось ложным. В этой ситуации в третий аргумент попадет то же значение, которое находилось во втором аргументе. Обратите внимание, что в этом случае нам уже не важно, чему равнялся первый аргумент, и его можно заменить анонимной переменной.

Все случаи применения отсечения принято разделять на "зеленые" и "красные". **Зелеными** называются те из них, при отбрасывании которых программа продолжает выдавать те же решения, что и при наличии отсечения. Если же при устранении отсечений программа начинает выдавать неправильные решения, то такие отсечения называются **красными**.

Пример "красного" отсечения имеется в реализации предиката max2 (если убрать отсечение, предикат будет выдавать в качестве максимума второе число, даже если оно меньше первого). Пример "зеленого" отсечения можно получить, если в запись предиката max добавить отсечения (при их наличии предикат будет выдавать те же решения, что и без них).

В принципе, с помощью отсечения в Прологе можно смоделировать такую конструкцию императивных языков, как ветвление.

Процедура

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

S:-

<условие>,! ,P.

S :-

P2.

будет соответствовать оператору if <условие> then P else P2, то есть если условие имеет место, то выполнить P, иначе выполнить P2. Например, в случае с максимумом, можно расшифровать нашу процедуру как "если X>Y, то M=X, иначе M=Y".

### **5. Сравнительная характеристика функционального, логического и процедурного подхода к программированию.**

#### **Процедурное программирование**

Процедурное (императивное) программирование является отражением архитектуры традиционных ЭВМ, которая была предложена фон Нейманом в 40-х годах. Теоретической моделью процедурного программирования служит алгоритмическая система под названием Машина Тьюринга (абстрактная вычислительная машина).

Программа на процедурном языке программирования состоит из последовательности операторов (инструкций), задающих процедуру решения задачи. Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, то есть значений исходных данных, в заключительное, то есть в результаты. Основным является оператор присваивания, служащий для изменения содержимого областей памяти.

Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

#### Процедурные языки характеризуются следующими особенностями:

- необходимостью явного управления памятью, в частности, описанием переменных;
- малой пригодностью для символьных вычислений;
- отсутствием строгой математической основы;
- высокой эффективностью реализации на традиционных ЭВМ.

Процедурный язык программирования предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

К процедурным языкам относятся: язык Ассемблера, С, Basic(версии начиная с QuickBasic до появления VisualBasic), Pascal.

### **Функциональное программирование**

Функциональное программирование — раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний. При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

Первым функциональным языком программирования был LISP. Роль основной конструкции в функциональных (аппликативных) языках играет выражение. К выражениям относятся скалярные константы, структурированные объекты, функции, тела функций и вызовы функций.

### Аппликативный язык программирования включает следующие элементы:

- классы констант, которыми могут манипулировать функции;
- набор базовых функций, которые программист может использовать без предварительного объявления и описания;
- правила построения новых функций из базовых;

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

- правила формирования выражений на основе вызовов функций.

Программа представляет собой совокупность описаний функций и выражения, которые необходимо вычислить. Данное выражение вычисляется посредством редукции, то есть серии упрощений, до тех пор, пока это возможно по следующим правилам:

- вызовы базовых функций заменяются соответствующими значениями;
- вызовы небазовых функций заменяются их телами, в которых параметры замещены аргументами.

Основной особенностью функционального программирования, определяющей как преимущества, так и недостатки данной парадигмы, является то, что в ней реализуется модель вычислений без состояний. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты, то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путём присваивания значений переменным, в функциональных достигается путём передачи выражений в параметры функций. Непосредственным следствием становится то, что чисто функциональная программа не может изменять уже имеющиеся у неё данные, а может лишь порождать новые путём копирования и/или расширения старых. Следствием того же является отказ от циклов в пользу рекурсии.

### Преимущества:

- повышение надёжности;
- удобство организации модульного тестирования;
- возможности оптимизации при компиляции;
- возможности параллелизма.

### Недостатки:

Недостатки функционального программирования вытекают из тех же самых его особенностей. Отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным

## ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

компонентом становится высокоэффективный сборщик мусора. Нестрогая модель вычислений приводит к непредсказуемому порядку вызова функций, что создает проблемы при вводе-выводе, где порядок выполнения операций важен. Кроме того, очевидно, функции ввода в своем естественном виде (например, getchar из стандартной библиотеки языка С) не являются чистыми, поскольку способны возвращать различные значения для одних и тех же аргументов, и для устранения этого требуются определенные ухищрения.

Для преодоления недостатков функциональных программ уже первые языки функционального программирования включали не только чисто функциональные средства, но и механизмы императивного программирования (присваивание, цикл, «неявный PROGN» были уже в LISPe). Использование таких средств позволяет решить некоторые практические проблемы, но означает отход от идей (и преимуществ) функционального программирования и написание императивных программ на функциональных языках.

## **Логическое программирование**

Логическое программирование — парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

Первым таким языком был язык Planner, в котором была заложена возможность автоматического вывода результата из данных и заданных правил перебора вариантов (совокупность которых называлась планом). Planner использовался для того, чтобы понизить требования к вычислительным ресурсам (с помощью метода backtracking) и обеспечить возможность вывода фактов, без активного использования стека. Затем был разработан язык Prolog, который не требовал плана перебора вариантов и был, в этом смысле, упрощением языка Planner.

Языки логического программирования, в особенности Пролог, широко используются в системах искусственного интеллекта. Центральным понятием в логическом программировании является отношение. Программа представляет собой совокупность определений отношений между объектами (в терминах условий или ограничений) и цели (запроса). Процесс выполнения программы трактуется как процесс

ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ  
общезначимости логической формулы, построенной из программы по правилам, установленным семантикой используемого языка. Результат вычисления является побочным продуктом этого процесса. В реляционном программировании нужно только специфицировать факты, на которых алгоритм основывается, а не определять последовательность шагов, которые требуется выполнить.

Языки логического программирования характеризуются:

- высоким уровнем;
- строгой ориентацией на символьные вычисления;
- возможностью инверсных вычислений, то есть переменные в процедурах не делятся на входные и выходные;
- возможной логической неполнотой, поскольку зачастую невозможно выразить в программе определенные логические соотношения, а также невозможно получить из программы все выводы правильные.

**Сравнительная характеристика языков программирования PASCAL, LISP, PROLOG:**

Характеристика	PASCAL	LISP	PROLOG
тип языка	процедурный	функциональный	логический
типы данных	скаляры, структуры	атомы, списки	атомы, структуры
обработка данных	присвоение, передача по значению, передача по ссылке	значение функции, передача по значению	связь переменных через унификацию
управление программой	последовательное, ветвление, циклы, рекурсия	вычисление функций, условные вычисления, рекурсии, циклы	рекурсия, бэктрекинг

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

<b>структур программы</b>	блоки, процедуры	функции, LET-блоки	правила, факты
<b>действия переменных</b>	глобальные, локальные	локальные, свободные	область действия переменной - одно предложение
<b>транслятор</b>	компилятор	интерпретатор, компилятор	интерпретатор
<b>длина программы</b>	5	3	1
<b>скорость</b>	1	2	5
<b>область</b>	программы общего назначения	символьная обработка, ИИ	ЭС, ИИ, прототипы

## VII. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

### 1. Определение класса в языке C++. Функции-члены класса в языке C++. Друзья класса в языке C++. Область видимости класса в языке C++. Инициализация класса в языке C++.

Механизм классов в C++ позволяет пользователям определять собственные типы данных. По этой причине их часто называют пользовательскими типами. Класс может наделять дополнительной функциональностью уже существующий тип. С помощью классов можно создавать абсолютно новые типы, например Screen (экран) или Account (расчетный счет). Как правило, классы используются для абстракций, не отражаемых встроенными типами адекватно.

#### **Определение класса**

Определение класса состоит из двух частей: заголовка, включающего ключевое слово class, за которым следует имя класса, и тела, заключенного в фигурные скобки. После такого определения должны стоять точка с запятой или список объявлений:

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

```
class Screen {  
public:  
    string      _screen; // string( _height * _width )  
    string::size_type _cursor; // текущее положение на экране  
    short       _height; // число строк  
    short       _width; // число колонок  
  
    void home();  
    void move( int, int );  
    char get();  
    char get( int, int );  
    void checkRange( int, int );  
    // ...  
} myScreen, yourScreen;
```

### **Функции-члены**

Функции-члены класса объявляются в его теле. Это объявление выглядит точно так же, как объявление функции в области видимости пространства имен.

Функцию-член можно объявить в любой из секций public, private или protected тела класса. Где именно это следует делать? Открытая функция-член задает операцию, которая может понадобиться пользователю. Множество открытых функций-членов составляет интерфейс класса. Например, функции-члены home(), move() и get() класса Screen определяют операции, с помощью которых программа манипулирует объектами этого типа.

Поскольку мы прячем от пользователей внутреннее представление класса, объявляя его члены закрытыми, то для манипуляции объектами типа Screen необходимо предоставить открытые функции-члены. Такой прием – скрытие информации – защищает написанный пользователем код от изменений во внутреннем представлении.

Внутреннее состояние объекта класса также защищено от случайных изменений. Все модификации объекта производятся с помощью небольшого набора функций, что существенно облегчает сопровождение и доказательство правильности программы.

Определение функции-члена также можно поместить внутрь тела класса:

```
class Screen {  
public:  
    // определения функций home() и get()
```

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

```
void home() { _cursor = 0; }
char get() { return _screen[_cursor]; }
// ...
};
```

### **Друзья**

Иногда удобно разрешить некоторым функциям доступ к закрытым членам класса. Механизм друзей позволяет классу разрешать доступ к своим неоткрытым членам. Объявление друга начинается с ключевого слова `friend` и может встречаться только внутри определения класса. Так как друзья не являются членами класса, то не имеет значения, в какой секции они объявлены. В примере ниже мы сгруппировали все подобные объявления сразу после заголовка класса:

```
class Screen {
    friend istream& operator>>( istream&, Screen& );
    friend ostream& operator<<( ostream&, const Screen& );
public:
    // ... оставшаяся часть класса Screen
};
```

Операторы ввода и вывода теперь могут напрямую обращаться к закрытым членам класса `Screen`.

Другом может быть функция из пространства имен, функция-член другого класса или даже целый класс. В последнем случае всем его функциям-членам предоставляется доступ к неоткрытым членам класса, объявляющего дружественные отношения.

### **Область видимости класса**

Тело класса определяет область видимости. Объявления членов класса внутри тела вводят их имена в область видимости класса.

Для обращения к ним применяются операторы доступа (точка и стрелка) и оператор разрешения области видимости (`::`). Когда употребляется оператор доступа, то предшествующее ему имя обозначает объект или указатель на объект типа класса, а следующее за ним имя должно находиться в области видимости этого класса. Аналогично при использовании оператора разрешения области видимости поиск имени, следующего за ним, идет в области видимости класса, имя которого стоит перед оператором.

Однако применение операторов доступа или оператора

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

разрешения области видимости нужно не всегда. Некоторые части программы сами по себе находятся в области видимости класса, и в них к членам класса можно обращаться напрямую. Одной из таких частей является само определение класса. Имя его члена можно использовать в теле после объявления:

```
class String {  
public:  
    typedef int index_type;  
    // тип параметра - это на самом деле String::index_type  
    char& operator[]( index_type )  
};
```

Порядок объявления членов класса в его теле важен: нельзя ссылаться на члены, которые будут объявлены позже. Однако из этого правила есть два исключения. Первое касается имен, использованных в определениях встроенных функций-членов, второе – имен, применяемых как аргументы по умолчанию.

### **Инициализация класса**

Рассмотрим следующее определение класса:

```
class Data {  
public:  
    int ival;  
    char *ptr;  
};
```

Мнемонические имена класса и обоих его членов сделали бы, конечно, его назначение более понятным для читателя программы, но не дали бы никакой дополнительной информации компилятору. Чтобы компилятор понимал наши намерения, мы должны предоставить одну или несколько перегруженных функций инициализации – конструкторов. Подходящий конструктор выбирается в зависимости от множества начальных значений, указанных при определении объекта. Например, любая из приведенных ниже инструкций представляет корректную инициализацию объекта класса Data:

```
Data dat01( "Venus and the Graces", 107925 );
```

```
Data dat02( "about" );
```

```
Data dat03( 107925 );
```

```
Data dat04;
```

Бывают ситуации (как в случае с dat04), когда нам нужен объект

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

класса, но его начальные значения мы еще не знаем. Возможно, они станут известны позже. Однако начальное значение задать необходимо, хотя бы такое, которое показывает, что разумное начальное значение еще не присвоено. Другими словами, инициализация объекта иногда сводится к тому, чтобы показать, что он еще не инициализирован. Большинство классов предоставляют специальный конструктор по умолчанию, для которого не требуется задавать начальных значений. Как правило, он инициализирует объект таким образом, чтобы позже можно было понять, что реальной инициализации еще не проводилось.

### **Конструктор класса**

Среди других функций-членов конструктор выделяется тем, что его имя совпадает с именем класса. Единственное синтаксическое ограничение, налагаемое на конструктор, состоит в том, что он не должен иметь тип возвращаемого значения, даже void. Количество конструкторов у одного класса может быть любым, лишь бы все они имели разные списки формальных параметров. C++ требует, чтобы конструктор применялся к определенному объекту до его первого использования.

### **Конструктор по умолчанию**

Конструктором по умолчанию называется конструктор, который можно вызывать, не задавая аргументов. Это не значит, что такой конструктор не может принимать аргументов; просто с каждым его формальным параметром ассоциировано значение по умолчанию:

### **2. Наследование в языке C++.**

Наследование — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.

Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса.

### **Типы наследования**

#### **1. Простое наследование**

Класс, от которого произошло наследование, называется базовым или родительским (англ. *base class*). Классы, которые

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

происходят от базового, называются потомками, наследниками или производными классами (англ. derived class).

В некоторых языках используются абстрактные классы. Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник ВУЗа», от которого наследуются классы «аспирант», «профессор» и т. д. Так как производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

### 2. Множественное наследование

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости. Множественное наследование реализовано в C++. Множественное наследование — потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках. В языках, которые позиционируются как наследники C++ (Java, C# и др.), от множественного наследования было решено отказаться в пользу интерфейсов. Практически всегда можно обойтись без использования данного механизма. Однако, если такая необходимость все-таки возникла, то, для разрешения конфликтов использования наследованных методов с одинаковыми именами, возможно, например, применить операцию расширения видимости — «::» — для вызова конкретного метода конкретного родителя.

Большинство современных объектно-ориентированных языков программирования (C#, Java, Delphi и др.) поддерживает возможность одновременно наследоваться от класса-предка и реализовать методы нескольких интерфейсов одним и тем же классом. Этот механизм позволяет во многом заменить множественное наследование — методы интерфейсов необходимо переопределять явно, что исключает ошибки при наследовании функциональности одинаковых методов различных классов-предков.

Наследование в языке C++

```
class A{ //базовый класс  
};
```

```
class B : public A{ //public наследование  
}  
  
class C : protected A{ //protected наследование  
}  
  
class Z : private A{ //private наследование  
}
```

В С++ существует три типа наследования: **public**, **protected**, **private**. Спецификаторы доступа членов базового класса меняются в потомках следующим образом:

- при public-наследовании все спецификаторы остаются без изменения.
- при protected-наследовании все спецификаторы остаются без изменения, кроме спецификатора public, который меняется на спецификатор protected (то есть public-члены базового класса в потомках становятся protected).
- при private-наследовании все спецификаторы меняются на private.

Одним из основных преимуществ public-наследования является то, что указатель на классы—наследники может быть неявно преобразован в указатель на базовый класс, то есть для примера выше можно написать: A\* a = new B;

### **3. Виртуальные функции в языке С++.**

С помощью виртуальных функций можно преодолеть трудности, возникающие при использовании поля типа. В базовом классе описываются функции, которые могут переопределяться в любом производном классе. Транслятор и загрузчик обеспечат правильное соответствие между объектами и применяемыми к ним функциями:

```
class employee {  
    char* name;  
    short department;  
    // ...  
    employee* next;  
    static employee* list;
```

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

```
public:  
    employee(char* n, int d);  
    // ...  
    static void print_list();  
    virtual void print() const;  
};
```

Служебное слово `virtual` (виртуальная) показывает, что функция `print()` может иметь разные версии в разных производных классах, а выбор нужной версии при вызове `print()` - это задача транслятора. Тип функции указывается в базовом классе и не может быть переопределен в производном классе. Определение виртуальной функции должно даваться для того класса, в котором она была впервые описана. Например:

```
void employee::print() const  
{  
    cout << name << '\t' << department << '\n';  
    // ...  
}
```

Мы видим, что виртуальную функцию можно использовать, даже если нет производных классов от ее класса. В производном же классе не обязательно переопределять виртуальную функцию, если она там не нужна. При построении производного класса надо определять только те функции, которые в нем действительно нужны:

```
class manager : public employee {  
    employee* group;  
    short level;  
    // ...  
public:  
    manager(char* n, int d);  
    // ...  
    void print() const;  
};
```

Место функции `print_employee()` заняли функции-члены `print()`, и она стала не нужна. Список служащих строит конструктор `employee`. Напечатать его можно так:

```
void employee::print_list()  
{  
    for ( employee* p = list; p; p=p->next) p->print();  
}
```

Данные о каждом служащем будут печататься в соответствии с типом записи о нем. Поэтому программа

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

```
int main()
{
    employee e("J.Brown",1234);
    manager m("J.Smith",2,1234);
    employee::print_list();
}

напечатает

J.Smith 1234
level 2
J.Brown 1234
```

Обратите внимание, что функция печати будет работать даже в том случае, если функция `employee_list()` была написана и оттранслирована еще до того, как был задуман конкретный производный класс `manager`! Очевидно, что для правильной работы виртуальной функции нужно в каждом объекте класса `employee` хранить некоторую служебную информацию о типе. Как правило, реализации в качестве такой информации используют просто указатель. Этот указатель хранится только для объектов класса с виртуальными функциями, но не для объектов всех классов, и даже для не для всех объектов производных классов. Дополнительная память отводится только для классов, в которых описаны виртуальные функции. Заметим, что при использовании поля типа, для него все равно нужна дополнительная память.

Если в вызове функции явно указана операция разрешения области видимости `::`, например, в вызове `manager::print()`, то механизм вызова виртуальной функции не действует. Иначе подобный вызов привел бы к бесконечной рекурсии. Уточнение имени функции дает еще один положительный эффект: если виртуальная функция является подстановкой (в этом нет ничего необычного), то в вызове с операцией `::` происходит подстановка тела функции. Это эффективный способ вызова, который можно применять в важных случаях, когда одна виртуальная функция обращается к другой с одним и тем же объектом. Пример такого случая - вызов функции `manager::print()`. Поскольку тип объекта явно задается в самом вызове `manager::print()`, нет нужды определять его в динамике для функции `employee::print()`, которая и будет вызываться.

#### 4. Полиморфизм. На примере C++.

Полиморфизм (polymorphism) (от греческого *polymorphos*) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Например для языка Си, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трёх различных функций: `abs()`, `labs()` и `fabs()`. Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В C++ каждая из этих функций может быть названа `abs()`. Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. В C++ можно использовать одно имя функции для множества различных действий. Это называется перегрузкой функций (function overloading).

В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор. Вам, как программисту, не нужно делать этот выбор самому. Нужно только помнить и использовать общий интерфейс. Пример из предыдущего абзаца показывает, как, имея три имени для функции определения абсолютной величины числа вместо одного, обычная задача становится более сложной, чем это действительно необходимо.

Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в Си, символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. В C++ вы можете применить эту концепцию и к другим, заданным вами, типам данных. Такой тип полиморфизма называется перегрузкой операторов.

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путём создания общего для них стандартного интерфейса для реализации похожих действий.

```
class Car {  
    public: double Benzin; // бензин  
    void Go( float dist ) {  
        Benzin -= dist / 10; // 1 литр бензина на 10 км  
        Rasst += dist; // проехали  
    }  
    protected: double Rasst; // пройденное расстояние  
};  
class Truck : public Car  
{  
    public: int Kuzov;  
    void Go( float dist ) {  
        Benzin -= dist / 5; // 1 литр бенза на 5 км  
        Rasst += dist; // проехали  
    }  
};
```

Стандартны следующие декларации объектов:

```
Car * car = new Car;  
Truck * truck = new Truck;
```

Однако принцип полиморфизма допускает и такую запись:

```
Car * car = new Truck;
```

Переменная ссылочного, более общего типа допускает хранение ссылки на дочерний тип.

А вот запись `Truck * truck = new Car;` будет ошибочной. Однако можно принудительно выполнить подобный оператор, если явно указать приведение типов: `Truck * truck = (Truck*)(new Car);`

Здесь создается новый экземпляр класса Car, и ссылка на него приводится к типу ссылки на объект класса Truck.

В результате в переменных `car` и `truck` будут храниться ссылки на объекты других классов! Переменая `car` хранит ссылку на объект-грузовик, наследник `Car`, а переменная `truck` хранит ссылку на экземпляр родительского класса `Car`.

Теперь зададим вызовы метода `Go()`, который имеется и в классе `Car`, и в классе `Truck`:

```
Car * car = new Truck;  
car->Go(100);  
Truck * truck = (Truck*)(new Car);  
truck->Go(100);
```

Статическое связывание - это связывание метода с родительским объектом на этапе компиляции. Так как компилятор не знает, как

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

конкретно будет работать программа, он всегда исходит из известного и явно заданного типа объекта-владельца метода. Если используется переменная `car` класса `Car`, то будет вызван метод `Go()` класса `Car`, хотя физически в переменной `car` хранится экземпляр класса `Truck`.

Так же и в случае с переменной `truck` - хотя мы принудительно записали в нее ссылку на объект класса `Car`, компилятор об этом не знает и исходит из типа переменной `truck` - вызывает метод `Go()` класса `Truck`.

При этом компилятор можно явно "обмануть" - выполнить приведение типов:

```
((Car*)truck)->Go(100);
```

Здесь уже будет вызван метод `Go()` класса `Car`, так как владелец метода `Go()` в левой части выражения явно приведен к типу `Car`.

Динамическое связывание - это связывание метода с объектом, которое происходит во время работы программы. При этом нужный метод определяется уже не типом переменной-владельца, а реальным типом объекта, ссылку на который она хранит. Но такое связывание возможно только для так называемых виртуальных методов, - разделение типов методов нужно, чтобы подсказать компилятору, где какое связывание реализовывать.

Виртуальный метод - это обычный метод, в заголовке которого используется ключевое слово `virtual`. Например:

```
class Car
{
public:
...
virtual void Go( float dist ) { ... }
```

Теперь метод `Go()` класса `Car` описан как виртуальный. Пусть имеются команды

```
Truck * truck = (Truck*)(new Car);
truck->Go(100);
```

Если бы метод `Go()` класса `Car` был невиртуальным, то выполнилось бы статическое связывание. Но в данном случае вызывается метод `Go()` класса `Car`, потому что переменная `truck` типа `Truck` хранит ссылку на экземпляр класса `Car`.

## **5. Инкапсуляция. На примере C++.**

Пусть члену класса (неважно функции-члену или члену, представляющему данные) требуется защита от "несанкционированного доступа". Как разумно ограничить

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

множество функций, которым такой член будет доступен? Очевидный ответ для языков, поддерживающих объектно-ориентированное программирование, таков: доступ имеют все операции, которые определены для этого объекта, иными словами, все функции-члены. Например:

```
class window
{
    // ...
protected:
    Rectangle inside;
    // ...
};

class dumb_terminal : public window
{
    // ...
public:
    void prompt ();
    // ...
};
```

Здесь в базовом классе `window` член `inside` типа `Rectangle` описывается как защищенный (`protected`), но функции-члены производных классов, например, `dumb_terminal::prompt()`, могут обратиться к нему и выяснить, с какого вида окном они работают. Для всех других функций член `window::inside` недоступен.

В таком подходе сочетается высокая степень защищенности (действительно, вряд ли вы "случайно" определите производный класс) с гибкостью, необходимой для программ, которые создают классы и используют их иерархию (действительно, "для себя" всегда можно в производных классах предусмотреть доступ к защищенным членам).

Неочевидное следствие из этого: нельзя составить полный и окончательный список всех функций, которым будет доступен защищенный член, поскольку всегда можно добавить еще одну, определив ее как функцию-член в новом производном классе. Для метода абстракции данных такой подход часто бывает мало приемлемым. Если язык ориентируется на метод абстракции данных, то очевидное для него решение - это требование указывать в описании класса список всех функций, которым нужен доступ к члену. В C++ для этой цели используется описание частных (`private`) членов.

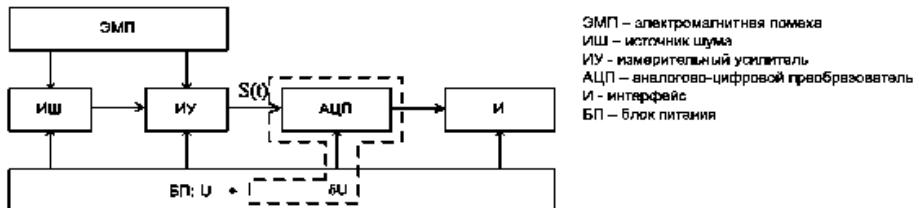
Важность инкапсуляции, т.е. заключения членов в защитную

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

оболочку, резко возрастает с ростом размеров программы и увеличивающимся разбросом областей приложения.

### VIII. МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

1. Системотехнические основы построения датчиков случайных чисел. Принципы аналого-цифрового преобразования. Причины выбора данного принципа аналого-цифрового преобразования.

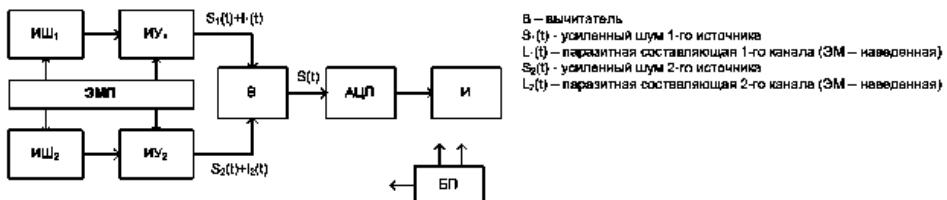


Функциональная схема генератора случайной последовательности (подход "в лоб")

S(t) оказывается не случайным, а модулированным рядом гармоник и не позволяет получить случайные числа.

Проблемы:

- 1) влияние ЭМП на случайность последовательности (50 Гц !!!);
  - 2) влияние качаний напряжений питания - вектор U (U на рис - это вектор питающих напряжений).
- Вторая проблема и отчасти первая могут быть решены подобающим выбором принципа аналого-цифрового преобразования.
  - Первая проблема принципиально может быть решена за счет схемотехнических и конструктивных решений



Функциональная схема реального генератора случайной последовательности

**Основные положения:**

- 1) Если ИШ<sub>1</sub> и ИШ<sub>2</sub> - источники на основе теплового шума, то S<sub>1</sub>(t) и L<sub>1</sub>(t), S<sub>2</sub>(t) и L<sub>2</sub>(t)- сравнимы по величине даже при хорошей экранировке.
- 2) Если S<sub>1</sub>(t<sub>i</sub>)и S<sub>1</sub>(t<sub>i</sub>), где t<sub>i</sub> = t<sub>0</sub> + iΔt, i=1,2,... – нормально распределенные случайные величины, то S<sub>1</sub>(t<sub>i</sub>) - S<sub>1</sub>(t<sub>i</sub>) – также нормально распределенная случайная величина
- 3) Если каналы конструктивно выполнены идентично, то обычно L<sub>1</sub>(t) ≈ L<sub>2</sub>(t). Для этого должны быть выполнены следующие требования:
  - a. Платы расположены в одной плоскости
  - b. Градиент плотности энергии электромагнитного поля в области расположения плат мал.

Если (1), (2) и (3) выполнены, то

$$s(t) = [s_1(t) + l_1(t)] - [s_2(t) + l_2(t)] = [s_1(t) - s_2(t)] + [l_1(t) - l_2(t)] \approx s_1(t) - s_2(t)$$

Где s<sub>1</sub>(t) – усиленный шум 1-го источника, l<sub>1</sub>(t) – паразитное составляющее 1-го канала

s<sub>2</sub>(t) – усиленный шум 2-го источника, l<sub>2</sub>(t) – паразитное составляющее 2-го канала

Таким образом ЭМП побеждена.

**Апробированные историей источники шума:**

1. Сцинтилляционные --- основан на случайному процессе радиоактивного распада. 1) Главный недостаток – временная нестабильность (ВН). Чем выше быстродействие, тем ниже стабильность. 2) Технические сложности защиты от радиации
2. Механические (лототроны) --- 1) неудобство использования, 2) ВН
3. Электрические
  - 3.1. Шум газового разряда --- ВН типа регресса Ток протекающий по люминесцентным лампам излучает случайность тем выше, чем ниже габаритные размеры.
  - 3.2. Шумовые вакуумные диоды --- ВН типа регресса. В криптографических приложениях используется, в т.ч для защиты информации от утечки побочным каналом.
  - 3.3. Шумовые полупроводниковые диоды --- Выдают шум в полурабочем состоянии. имеются паразитные процессы, но в целом обладают неплохими качествами. Именно они используется для генерации случайных последовательностей.

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

3.4. Использование теплового шума проволочных резисторов. Требуется высокая чувствительность усилителей

### **Аналогово-цифровое преобразование. Принадлежности.**

1 проблема – качания питающих напряжений

2 проблема – «нормальность» закона распределения исходных случайных величин.

Причины качания напряжения:

- Изменение нагрузки в зависимости от текущего сочетания решаемых на компьютере задач
- Изменение внешних напряжений (сезонные и принудительные со стороны злоумышленника)

Следствие: качание эталонных импульсов тока АЦП и т.о. неслучайное модулированная последовательность случайных чисел.

Для криптографических приложений нужны случайные числа с равномерным законом распределения, в то время как шум у полупроводниковых диодов и тепловой шум имеет нормальный закон распределения, т.е. необходим преобразователь. В качестве такого преобразователя, одновременно устраняющего проблему с качаниями напряжений, может быть использована АЦП принадлежность.

**Теорема.** Пусть для любого целого  $k$ ,  $x_{k+1} - x_k = b$ , тогда для непрерывной случайной величины с нормальной плотностью распределения

$$\varphi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\xi}{\sigma}\right)^2}$$

$$\bigcup_{x \in Z} (x_{2k-1}, x_{2k})$$

вероятности попадания в поле интервала  $\bigcup_{x \in Z} (x_{2k-1}, x_{2k})$  и

$$\bigcup_{x \in Z} (x_{2k}, x_{2k+1})$$

одинаковы.

Определение. АЦП, формирующей в той или иной форме номер поля интервала, к которому принадлежат измеряемый в данный момент аналоговый сигнал, назовем АЦП принадлежностью.

В качестве формирователей последовательностей случайных бит с равномерным распределением 0 и 1 на основе теплового шума и шума полупроводниковых диодов целесообразно использовать

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

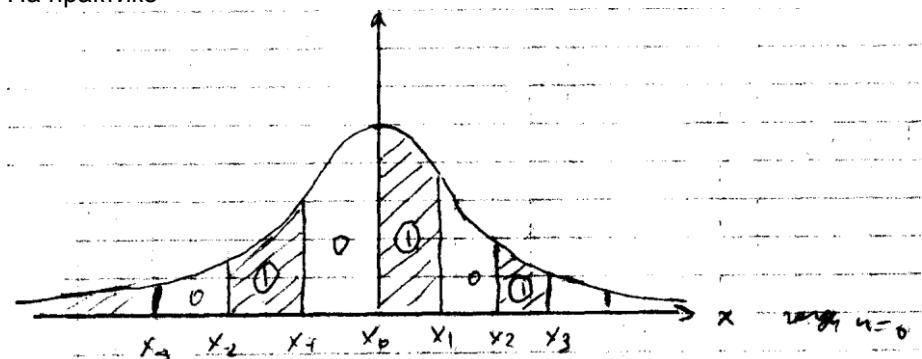
АЦП принадлежности, построены на основе результатов рассмотренной теоремы. Это объясняется 1) нечувствительностью таких формирователей к качаниям напряжений, 2) нечувствительностью таких формирователей их качаниям амплитуды теплового шума, 3) в случае зеркальной симметрии обоих интервалов такого рода формирователь будет правильно выполнять свою функцию для случайных сигналов с производимой четной функцией плотности распределения. На практике невозможно идеально соблюсти условия теоремы.

1) множество  $Z$  сужают до множества

$$Z_{\pm n} = \{-n, -n+1, \dots, -1, 0, 1, \dots, n-1, n\}$$

2)  $x_{k+n} - x_k = b$  - условие выполняется приближенно

На практике



Чем выше  $n$ , тем выше достигается точность преобразования, но тем сложнее настройка

Отсюда в коммерческих приложениях  $n = 2 \sim 3$ , а в приложениях систем гарантированной секретности  $n = 4 \sim 6$

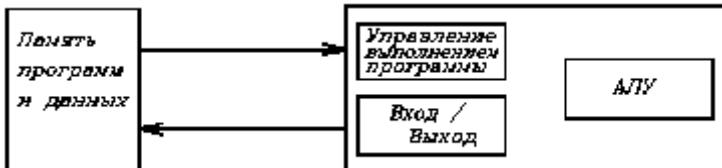
## **2. Понятие Фон Нэймановской архитектуры вычислительной системы. Базовые принципы. Проблема получения случайных чисел в рамках данной архитектуры. Основной вывод**

Практически все сигнальные процессоры имеют схожие базовые модули: вычислительное ядро, служащее для выполнения математических операций; память для хранения данных и программ; устройства преобразования аналоговых сигналов в цифровые и наоборот.

Адрес выполняемой команды отображается в адресном автомате вычислительного ядра сигнального процессора. Обычный цикл работы процессора состоит из выбора команды и данных из

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

памяти программ и данных и сохранения результатов обработки. По отношению к памяти программ и данных различают Фон Неймановскую и Гарвардскую архитектуры процессоров. Основные особенности типов архитектур показаны на рисунке 3.



Фон Неймановская архитектура является стандартом в развитии микропроцессорных систем. Описываемая архитектура наиболее проста, так как программа и данные располагаются в одной и той же памяти. Фон Неймановская архитектура используется для построения в основном универсальных процессоров, таких как процессоры семейства x86. Основная особенность такой архитектуры - наличие только одной шины, в результате за один цикл обращения процессор может получить доступ либо к памяти программ, либо к памяти данных.

### **Принципы фон Неймановской архитектуры:**

- 1) Совмещение памяти программ и памяти данных, т.е. программы в некоторых случаях могут рассматриваться как данные
  - 2) Строгая детерминированность действий в соответствии с программой, находящейся в памяти компьютера
- Вывод (следует из 2-го утверждения): в Фон-Неймановской архитектуре невозможна генерация случайных чисел, так как все определено, нет случайного характера.
- Аппаратные источники случайных чисел конечно-автоматного типа не возможны.
- 1) Нарушение 2-й характеристики фон Неймановской архитектуры возможно в истинно параллельных вычислительных структурах (отдельно независимые тактовые генераторы).

Реализация функции randomize в языках программирования. Случайность достигается за счет не «фон-нейманности» связанной с использованием взаимодействия двух истинно параллельных процессов: вычислительного (АЛУ со своим тактовым генератором) и векового таймера (со своим тактовым генератором). Случайность низкого качества. Повышение качества возможно за счет применения случайных вычислительных алгоритмов.

- 2) Не «фон-неймановость» поведений человека при взаимодействии с устройствами ввода (мышь, клавиатура).

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

- А) качество случайности – низкое, так как зона подбора может быть сужена  
 Б) не очень высокая скорость генерации

### **3. Системы гарантированной секретности. Теоретические основы.**

Получены были К. Шенноном.

Пусть  $R$  – алфавит символов ключа,  $A$  – алфавит символов исходного сообщения.

В случае, если мощность множества  $|R| \geq |A|$  и ключевая последовательность является равномерно распределенной случайно величиной на множестве  $R$  и каждая ключевая последовательность используется только 1 раз (т.к. 1 раз – шифр-блокнот), то атака на такой шифр невозможна в принципе. Связано это с тем, что доказано, что зашифрованное сообщение с равной вероятностью может быть декодировано в любое другое в принципе возможное.

#### **“Грабли” Цезаря.**

Пусть  $n$  – разрядность слова;

$\{ai\}_i = 1, 2, \dots, N$  – шифруемая последовательность слов;

$\{ri\}_i = 1, 2, \dots, N$  – ключ шифрования;

$\{a^*i\}_i = 1, 2, \dots, N$  – зашифрованная последовательность.

Положим:

$$ai^* = |ai + ri|m \quad (i = 1, 2, \dots, N), \text{ где } m = 2n. \quad (1)$$

Рассмотрим операцию:

$$|ai^* + ri|m, \text{ где } ri – \text{ противоположное } ri \text{ число по модулю } m.$$

$$|ai^* + ri|m = ||ai + ri|m + ri|m = |ai|m + |ri|m + |ri|m|m = |ai + |ri + ri|m|m = |ai|m = ai,$$

так как  $\forall i (i = 1, 2, \dots, N) ai < m = 2n$ .

$$\text{T. o.: } ai = |ai^* + ri|m = |ai^* + m - ri| \quad (i=1,2,\dots,N) \quad (2)$$

Схема шифрования (1) и дешифрования (2) называются схемой Цезаря. На рис. представлено условное изображение древнего механизма, реализующего схему (1) и (2).

В случае, когда последовательность  $\{ri\}_i = 1, 2, \dots, N$  является периодической с периодом  $k$ , то говорят о схеме “Граблей”



Цезаря с  $k$  “зубьями”.

В случае, когда последовательность  $\{ri\}_i=1,2,\dots,N$  является последовательностью случайных чисел, равномерно распределенных на отрезке  $[0, 2n-1]$  и мощность множества элементов, входящих в эту последовательность, не ниже мощности множества элементов, входящих в

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

последовательность  $\{a_i\}$ , и для каждого акта шифрования некоторой последовательности  $\{a_i\}_{i=1,2,\dots,N}$  и дешифрования  $\{a^*_i\}_{i=1,2,\dots,N}$ , используется отдельная реализация  $\{r_i\}_{i=1,2,\dots,N}$  (то есть  $\{r_i\}_{i=1,2,\dots,N}$  используется однократно), то говорят о схеме гарантированной секретности.

Однократное использование ключа в системе гарантированной секретности наз-ся использованием одноразового шифра-блокнота.

Схема, приближенная к данной модели называется *схемой Цезаря (Вижинера) с псевдослучайным длиннопериодическим ключом*.

На сегодня известны 2 изоморфных подхода генерации псевдослучайных последовательностей:

- сложение сдвиговых регистров
- на рекуррентных соотношениях, причем среди всех рекуррентных соотношений лучше дает результат следующее соотношение:

$$X_{i+1} = |ax_i + C|_M$$

$$0 < a < M, 0 \leq C < M, 0 \leq X_0 < M$$

Данный метод генерации псевдослучайно последовательности называется *линейным конгруэнтным методом*. При  $C = 0$  метод называется мультиплективным конгруэнтным методом,  $C \neq 0$  метод называется смешанным конгруэнтным методом. Качество получаемой псевдослучайной последовательности зависит от всех 4 параметров метода:  $X_0$  – начальное значение,  $a$  – множитель,  $C$  – приращение,  $M$  – модуль.

## **4. Длиннопериодические ключевые последовательности.**

**Датчики псевдослучайных чисел и их роль для создания длиннопериодических ключевых последовательностей.**  
**Анализ стойкости длиннопериодических ключевых последовательностей**

Псевдослучайный ключ (длиннопериодический) в некоторых случаях является практически достаточно стойким и используется в качестве вспомогательного алгоритма в современных криптографических алгоритмах.

Одним из самых лучших методов генерации случайной последовательности является метод Лемера 1948г. – линейный конгруэнтный метод генерации, который заключается в последовательности вычисления следующего регулярного соотношения:

$$X_{i+1} = |ax_i + C|_M, i=1,2,..$$

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

при известных  $M$  – модуль,  $C$  – приращение,  $a$  – множитель,  $x_0$  – начальное значение

эти пары д. б. специально подобраны ( $M, a, C, x_0$ )

В случае, если  $C=0$ , то метод называют мультипликативным конгруэнтным методом.

При  $C \neq 0$  – смешанным конгруэнтным методом.

Качественная псевдослучайная последовательность может быть получена только при правильном выборе параметров датчика.

Выбор параметров:  $x_{i+1} = |ax_i + C|_M$  датчик

Его описание  $0 \leq x_0 < M$

$0 < a < M$

$0 \leq C < M$

Как обеспечить стойкость? Действия:

- 1) выбор  $M$
- 2) выбор  $a$
- 3) выбор  $C$
- 4) выбор  $X_0$  (условный)
- 5) какую часть  $x_i$  использовать в качестве  $r_i$  ( $r_i$  – выбор из  $x_i$  – младшие, старшие байты)

$M$  должно быть достаточно большим. Чем больше  $M$ , тем выше вычислительная сложность. Выбор  $M$  обычно осуществляется из соображений вычислительной эффективности. Казалось бы наиболее эффективный выбор  $M=2^e$ , где  $e$  – разрядность вычислительной машины. Но качество псевдослучайной последовательности в этом случае окажется недопустимо низким.

Выбор модуля:

- 1) линейные конгруэнтные методы (линейные) последовательности, т. к.  $x_i < M$ ,
- 2) длина периода не м. б.  $> M$ ,
- 3) максимально большой период

В таких случаях просто произвести вычисления, - самый простой в вычисление со степенью 2.

Пример:  $M=2^{16}$

```

MOV AX,x
MOV BX,a
MOV CX,c
MUL BX; (DX;AX):=ax_i <=> (ax)=|ax_i|_M
ADD AX,CX; (ax):= |ax_i+c|_M
MOV x_{i+1}, ax

```

Также эффективные схемы могут быть реализованы при  $M=2^e+1$  и  $M=2^e-1$ , следует предположить, что  $C = 0$ . Датчики с таким модулем не следует применять в схемах, использующих вычеты

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

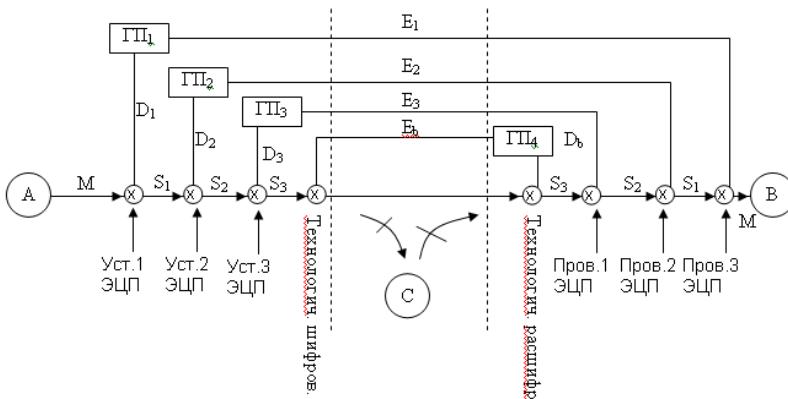
по модулю делителей М. Простое число – лучший выбор при выборе модуля.

### **5. Простейшие протоколы обеспечения многократной электронной цифровой подписи. Пример применения**

Криптографические системы с открытыми ключами шифрования позволяют пользователям осуществлять безопасную передачу данных по незащищенному каналу без какой-либо предварительной подготовки. Такие криптосистемы должны быть асимметричными в том смысле, что отправитель и получатель имеют различные ключи (алгоритмы), причем ни один из них не может быть выведен из другого при помощи вычислений.

Реализация ЭЦП при помощи асимметричных криптосистем возможна, только если операторы шифрования и дешифрования коммутируют  $DE = ED$ .

Процедура шифрования на секретном алгоритме отправляющей стороны означает однозначную идентификацию управляющей стороны и называется электронно-цифровой подписью (электронной сигнатурой).



D – секретный ключ

E – публичный(открытый) ключ

ГП – генерация пар

M – сообщение

Использование ЭЦП приведенной в схеме предполагает производственный порядок применения процедур шифрования и расшифрования.

Практическое применение: электронные платежи

Преимущества: отсутствие секретного канала

Недостатки: низкое быстродействие

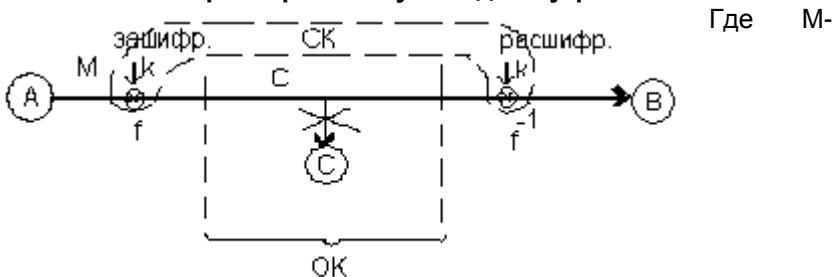
## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

### 6. Модель угроз «Несанкционированный доступ к передаваемой через открытый канал информации». Криптографические методы противодействия данной угрозе

Модель угроз «Несанкционированный доступ к передаваемой через открытый канал информации» предполагает доступ злоумышленника к передаваемой по открытому каналу информации с возможностью её дешифрования.

#### 1 метод.

##### Рассмотрим простейшую модель угроз



сообщение,

А и В – обмениваются сообщениями по открытому каналу (OK),

С – злоумышленник, подслушивающий сообщение.

$C=f(M,k)$  – функциональная нотация,

$M=f^{-1}(C,k)$  – функциональная нотация,

$C=F_k M$  – оператор нотации,

$M=F_k^{-1} C$  – оператор нотации,

$f$  – функция зашифровывания,

$f^{-1}$  – функция расшифровывания,

$F$  – оператор зашифровывания,

$F^{-1}$  – оператор расшифровывания,

$k$  – ключ – секретной информации, которая не передается по открытому каналу,

СК – секретный канал.

#### Преимущества:

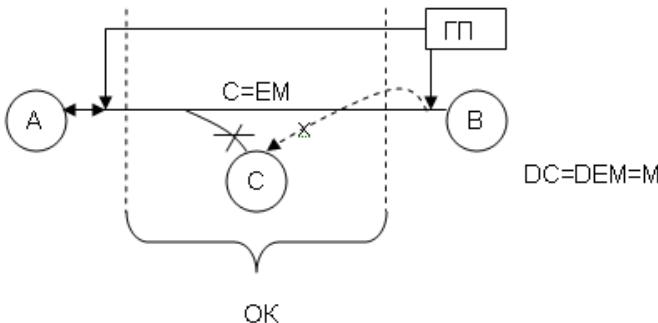
1. Высокое быстродействие
2. Простота алгоритма

#### Недостатки:

1. Наличие секретного канала (его недостаток – высокая стоимость организации)

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

**2 метод.** Для борьбы с упомянутой моделью угроз предложена следующая криптографическая схема: будем использовать оперативную нотацию, т.к. она более удобна



Основными идеями данной схемы для защиты от угрозы несанкционированного доступа является:

- 1) для зашифровывания и расшифровки используются различные алгоритмы  $E$  и  $D$  (различные ключи). При этом совместное порождение алгоритмов  $E$  и  $D$  – алгоритмически легко разрешимая задача. А вычисление по  $E$  алгоритма  $D$  вычислительно-трудноразрешимая (неразрешимая) в оригинале задача.

- 2) Совместная генерация алгоритмов  $E$  и  $D$  происходит на принимающей стороне (ГП-генератор пары на рис.)

$C$  – не может указать, что было в канале

ГП – генератор пары  $E$  и  $D$

$M$  – исходное сообщение

$C = EM$  – зашифрованное сообщение

**Преимущества:** Отсутствие секретного канала

**Недостаток:** Низкое быстродействие (на 2 порядка)

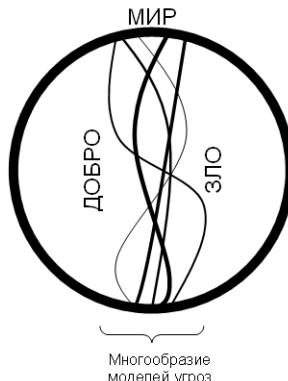
7. Модель угроз «Искажение передаваемой в открытом канале информации». Криптографические методы противодействия данной угрозе. Классификация методов. Пример.

**Определение:** Модель угроз – абстрактное формализованное или неформализованное описание методов реализации угроз и последствий от их реализации.

Образно модель угроз можно представить, как некоторое описание границы между добром и злом в мире. Поскольку, с

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

одной стороны, для любой точки зрения на соотношение добра и зла можно сформулировать множество описаний границ между ними, а с другой, существует множество точек зрения, то и различных моделей угроз может быть построено очень много.

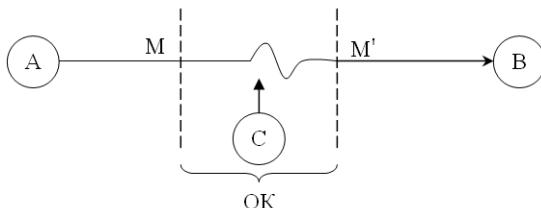


Модель угроз «Искажение передаваемой в открытом канале информации».

А передает информацию В по открытому каналу, которую злоумышленник С может исказить сообщение, т. е. модифицирует его.

Как устранить искажение?

О.К. – открытый канал, М - сообщение

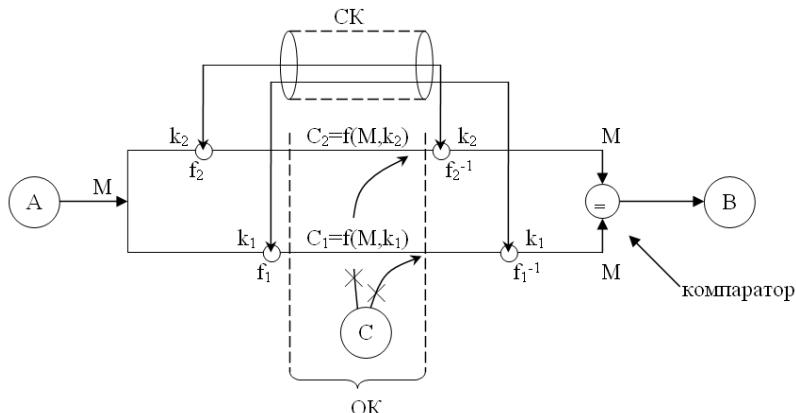


Протоколы(схемы), основанные на симметричном и асимметричном основании позволяют выявить искажения (рассматривая модель угроз) только в случае семантической избыточности М.

Есть три способа борьбы:

1. способ, основанный на алгоритмах симметричных криптографии;
2. способ, основанный на алгоритмах асимметричных криптографии;
3. способ, основанный на вычислении функции подтверждения целостности или криптографической хэш-функций;

*Метод симметрического шифрования:*



$$C_1 = f^{-1}(C_1, k_1); \quad C_2 = f(M, k_2); \quad M_1 = f^{-1}(C_1, k_1); \quad M_2 = f^{-1}(C_2, k_2); \\ M = M_1; \text{ если } M_1 = M_2$$

Решение о том, искажена ли информация в канале или нет принимает сторона на основе сравнения сообщения, полученных при расшифровке сообщения зашифрованного передающей стороной на двух различных ключах. При этом злоумышленник, чтобы исказить информацию в канале и убедить принимающую сторону в ее подлинности нужно знание обоих ключей, для передаче которых, каналом симметрической криптографии, используется СК – секретный канал. Т. о. атака на данную систему контроля целостности сообщения сводится к задаче критоанализа исходной криптографической системы.

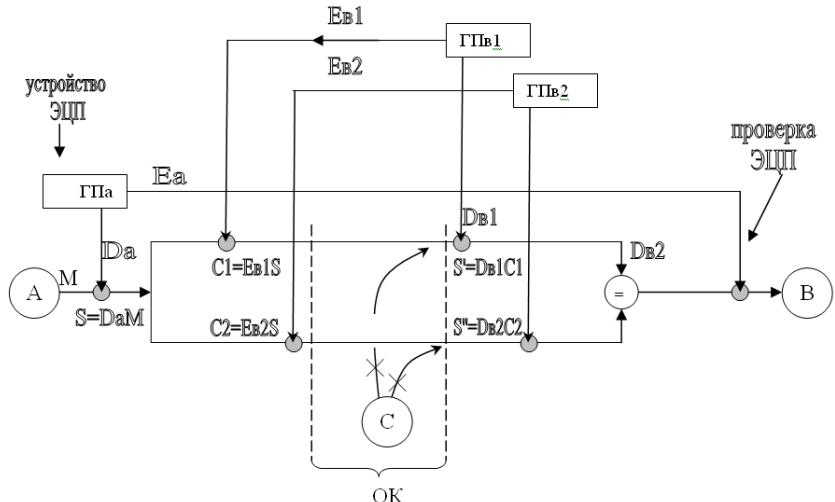
Данный протокол обеспечивает защиту от 2x видов сразу:

Несанкционированный доступ

Контроль целостности информации

Недостаток: наличие секретного канала.

*Метод асимметрического шифрования*



Принцип контроля целостности, используемый в данной схеме (протоколе) аналогичен предыдущему. Злоумышленник для осуществления искажения информации в канале и убеждения принимаемой стороны в ее достоверности и должен обладать 2-мя различными секретными ключами передающих сторон, т. е. атака на данный протокол сводится к атаке на асимметрическую криптографическую систему, использованную в протоколе. В то же время данный протокол обеспечивает защиту от угрозы несанкционированного доступа (за счет технологического шифрования на открытом канале принимающей стороны).

Преимущества схемы: отсутствие секретного канала

Недостаток: низкое быстродействие

Общий недостаток 2-х рассмотренных протоколов: емкость открытого канала д.б. в 2 раза больше емкости канала в случае передачи сообщения без контроля целостности.

Этого недостатка лишены подходы, основанные на использовании криптографической хэш-функций.

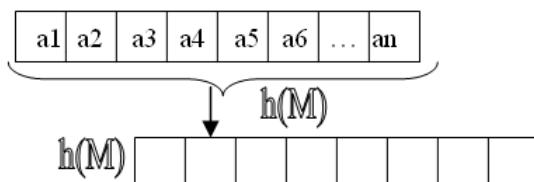
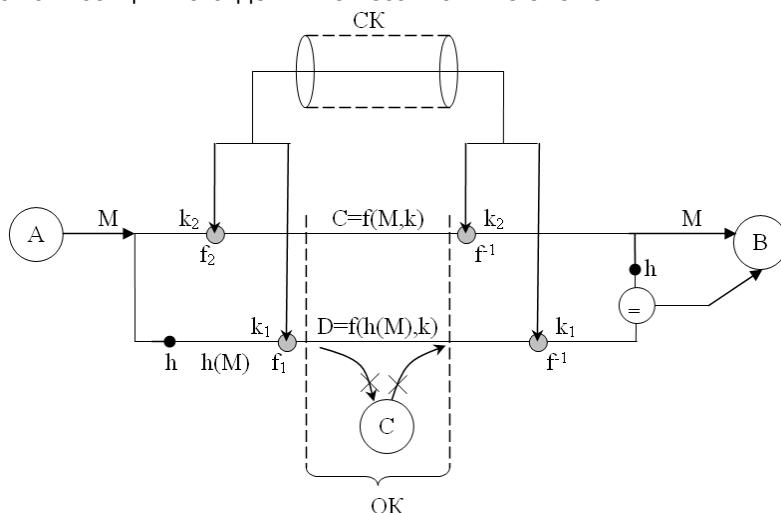
*Принципы формирования хэш-функций(х-ф):*

1) алгоритм вычисления хэш-функций не должен использовать секретную информацию, хотя сообщение или значение хэш-функции, либо то и другое вместе должны быть скрыты от злоумышленников.

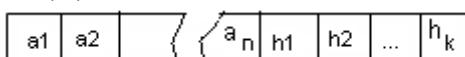
2) алгоритм должен эффективно выполняться как на микропроцессорах, так и на универсальных процессорах без использования специальных аппаратных средств защиты информации

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

- 3) если для 2-х различных сообщений произвольной длины вычислены значения хэш-функций, то вероятность их совпадений должна быть равномерной случайной величиной с математическим ожиданием  $2k$ ,  $k$  – количество значений битов контрольной суммы
- 4) хэш-функция должна быть чувствительна ко всем возможным перестановкам, переупорядоченным, также операции редактирования, удаления, включение текста
- 5) число, соответствующее значению хэш-функций, должно иметь длину не менее 120 бит, чтобы защитить информацию от вторжения быстрым подбором
- 6) хэш-функция должна быть необратимой, т.е. не допускающей до композиции на отдельные независимые элементы.



$k < h$ ,  $k \geq 128$  бит  
 $\langle M, h(M) \rangle$

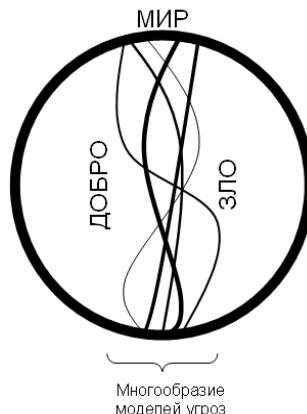


Преимущества х-ф: незначительная потеря в количестве информации пересылаемого значения.

**8. Модель угроз «Нарушение целостности программного обеспечения внутри периметра защиты». Формализация. Субъектно-объектный подход.**

**Определение:** Модель угроз – абстрактное формализованное или неформализованное описание методов реализации угроз и последствий от их реализации.

Образно модель угроз можно представить, как некоторое описание границы между добром и злом в мире. Поскольку, с одной стороны, для любой точки зрения на соотношение добра и зла можно сформулировать множество описаний границ между ними, а с другой, существует множество точек зрения, то и различных моделей угроз может быть построено очень много.



Модель угроз «Нарушение целостности программного обеспечения внутри периметра защиты».

При создании инфраструктуры корпоративной автоматизированной системы неизбежно встает вопрос о защищенности ее от угроз.

*Определение понятия защищенности Автоматизированных Систем*

Основой формального описания систем защиты традиционно считается модель системы защиты с полным перекрытием, в которой рассматривается взаимодействие:

1. «области угроз»;
2. «защищаемой области»;
3. «системы защиты».

Таким образом, имеем три множества:

1.  $T = \{t_i\}$  — множество угроз безопасности,
2.  $O = \{o_j\}$  — множество объектов (ресурсов) защищенной системы,

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

3.  $M = \{mk\}$  — множество механизмов безопасности АС.

Элементы этих множеств находятся между собой в определенных отношениях, собственно и описывающих систему защиты. Для описания системы защиты обычно используется графовая модель.

Множество отношений угроза-объект образует двухдольный граф  $\{<T, O>\}$ . Цель защиты состоит в том, чтобы перекрыть все возможные ребра в графе. Это достигается введением третьего набора  $M$ ; в результате получается трехдольный граф  $\{<T, M, O>\}$ .

Развитие модели предполагает введение еще двух элементов.

1.  $V$  — набор уязвимых мест, определяемый подмножеством декартова произведения  $T^*O$ :  $vg = <ti, oj>$ . Под уязвимостью системы защиты понимают возможность осуществления угрозы  $t$  в отношении объекта  $o$ . (На практике под уязвимостью системы защиты обычно понимают не саму возможность осуществления угрозы безопасности, а те свойства системы, которые либо способствуют успешному осуществлению угрозы, либо могут быть использованы злоумышленником для осуществления угрозы.)
2.  $B$  — набор барьеров, определяемый декартовым произведением  $V^*M$ :  $bl = <ti, oj, mk>$ , представляющих собой пути осуществления угроз безопасности, перекрытые средствами защиты.

В результате получаем систему, состоящую из пяти элементов:  $\{T, O, M, V, B\}$ , описывающую систему защиты с учетом наличия в ней уязвимостей.

Для системы с полным перекрытием для любой уязвимости имеется устраниющий ее барьер. Иными словами, в подобной системе защиты для всех возможных угроз безопасности существуют механизмы защиты, препятствующие осуществлению этих угроз.

Данное условие является первым фактором, определяющим защищенность автоматизированной системы(АС);

Второй фактор — прочность механизмов защиты.

В идеале каждый механизм защиты должен исключать соответствующий путь реализации угрозы. В действительности же механизмы защиты обеспечивают лишь некоторую степень сопротивляемости угрозам безопасности. Поэтому в качестве характеристик элемента набора барьеров  $bl = <ti, oj, mk>$ ,  $bl \in B$  может рассматриваться набор  $\{PI, LI, RI\}$ , где

$PI$  — вероятность появления угрозы;

$LI$  — величина ущерба при удачном осуществлении угрозы в

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

отношении защищаемых объектов (уровень серьезности угрозы); RI — степень сопротивляемости механизма защиты  $m_k$ , характеризующаяся вероятностью его преодоления.

Прочность барьера  $b_l = \langle t_i, o_j, m_k \rangle$  характеризуется величиной остаточного риска  $Risk_l$ , связанного с возможностью осуществления угрозы  $t_i$  в отношении объекта автоматизированной системы  $o_j$  при использовании механизма защиты  $m_k$ .

Знаменатель определяет суммарную величину остаточных рисков, связанных с возможностью осуществления угроз  $T$  в отношении объектов автоматизированной системы  $O$  при использовании механизмов защиты  $M$ . Суммарная величина остаточных рисков характеризует общую уязвимость системы защиты. А защищенность определяется как величина, обратная уязвимости. При отсутствии в системе барьеров  $b_k$ , перекрывающих определенные уязвимости, степень сопротивляемости механизма защиты  $R_k$  принимается равной нулю.

На практике получение точных значений приведенных характеристик барьеров затруднено, поскольку понятия угрозы, ущерба и сопротивляемости механизма защиты трудно формализовать. Так, оценку ущерба в результате несанкционированного доступа к информации политического и военного характера точно определить вообще невозможно, а определение вероятности осуществления угрозы не может базироваться на статистическом анализе.

Вместе с тем, для защиты информации экономического характера, допускающей оценку ущерба, разработаны стоимостные методы оценки эффективности средств защиты. Для этих методов набор характеристик барьера дополняет величина  $C_l$  затраты на построение средства защиты барьера  $b_l$ . В этом случае выбор оптимального набора средств защиты связан с минимизацией суммарных затрат  $W=\{w_l\}$ , состоящих из затрат  $C=\{c_l\}$  на создание средств защиты и возможных затрат в результате успешного осуществления угроз  $N=\{n_l\}$ .

Построение моделей системы защиты и анализ их свойств составляют предмет «теории безопасных систем», еще только оформляющейся в качестве самостоятельного направления.

*Формальные подходы к решению задачи оценки защищенности из-за трудностей, связанных с формализацией, широкого практического распространения не получили.* Значительно более действенным является использование неформальных классификационных подходов. Вместо стоимостных оценок используют категорирование:

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

- нарушителей (по целям, квалификации и доступным вычислительным ресурсам);
- информации (по уровням критичности и конфиденциальности);
- средств защиты (по функциональности и гарантированности реализуемых возможностей) и т.п.

Такой подход не дает точных значений показателей защищенности, однако позволяет классифицировать АС по уровню защищенности и сравнивать их между собой. Примерами классификационных методик, получивших широкое распространение, могут служить разнообразные критерии оценки безопасности ИТ, принятые во многих странах в качестве национальных стандартов, устанавливающие классы и уровни защищенности. Результатом развития национальных стандартов в этой области является обобщающий мировой опыт международный стандарт ISO 15408.

### *Нормативная база анализа защищенности*

Наиболее значимыми нормативными документами, определяющими критерии оценки защищенности и требования, предъявляемые к механизмам защиты, являются «Общие критерии оценки безопасности информационных технологий» и «Практические правила управления информационной безопасностью». В других странах их место занимают соответствующие национальные стандарты.

### *ISO 15408.*

«Общие критерии» определяют функциональные требования безопасности и требования к адекватности реализации функций безопасности. «Общие критерии» целесообразно использовать для оценки уровня защищенности с точки зрения полноты реализованных в ней функций безопасности и надежности реализации этих функций.

С точки зрения оценки защищенности АС особый интерес представляет класс требований по анализу уязвимостей средств и механизмов защиты (Vulnerability Assessment). Он определяет методы, которые должны использоваться для предупреждения, выявления и ликвидации следующих типов уязвимостей:

- наличия побочных каналов утечки информации;
- ошибки в конфигурации, либо неправильном использовании системы, приводящем к ее переходу в небезопасное состояние;
- недостаточной стойкости механизмов безопасности, реализующих соответствующие функции;
- наличие уязвимостей в средствах защиты информации,

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

позволяющих пользователям получать доступ к информации в обход существующих механизмов защиты. Соответствующие требования гарантированности оценки содержатся в следующих четырех семействах требований:

- Covert Channel Analysis (анализе каналов утечки информации);
- Misuse (ошибке в конфигурации, либо неправильном использовании системы, приводящем к переходу системы в небезопасное состояние);
- Strength of TOE Security Functions (стойкость функций безопасности, обеспечиваемая их реализацией);
- Vulnerability Analysis (анализ уязвимостей).

## *ISO 17799*

Наиболее полно критерии для оценки механизмов безопасности организационного уровня представлены в еще одном международном стандарте ISO 17799, принятом в 2000 году. Данный стандарт, являющийся международной версией британского стандарта BS 7799, содержит практические правила по управлению информационной безопасностью и может использоваться в качестве критериев оценки механизмов безопасности организационного уровня, включая административные, процедурные и физические меры защиты.

Практические правила разбиты на десять разделов:

1. Политика безопасности.
2. Организация защиты.
3. Классификация ресурсов и их контроль.
4. Безопасность персонала.
5. Физическая безопасность.
6. Администрирование компьютерных систем и сетей.
7. Управление доступом.
8. Разработка и сопровождение информационных систем.
9. Планирование бесперебойной работы организации.
10. Контроль выполнения требований политики безопасности.

В этих разделах содержится описание механизмов организационного уровня, реализуемых в настоящее время в государственных и коммерческих организациях во многих странах.

*Анализ конфигурации средств защиты внешнего периметра сети*

При анализе конфигурации средств защиты внешнего периметра сети и управления межсетевыми взаимодействиями особое внимание обращается на следующие аспекты:

- настройку правил разграничения доступа (фильтрация

#### МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

- сетевых пакетов);
- используемые схемы и настройку параметров аутентификации;
  - настройку параметров системы регистрации событий;
  - использование механизмов, обеспечивающих скрытие топологии защищаемой сети (например, трансляция сетевых адресов);
  - настройку механизмов оповещения об атаках и реагирования;
  - наличие и работоспособность средств контроля целостности;
  - версии используемого программного обеспечения и установленные обновления.

Анализ конфигурации средств защиты внешнего периметра локальной сети предполагает проверку правильности установки сотен различных параметров конфигурации межсетевых экранов, маршрутизаторов, шлюзов виртуальных частных сетей, прокси-серверов, серверов удаленного доступа и др. Для автоматизации этого процесса могут использоваться специализированные программные средства анализа защищенности, выбор которых в настоящее время достаточно широк.

Один из методов автоматизации процессов анализа и контроля защищенности распределенных компьютерных систем — использование технологий интеллектуальных программных агентов. На каждую из контролируемых систем устанавливается программный агент, который выполняет соответствующие настройки программного обеспечения, проверяет их правильность, контролирует целостность файлов, своевременность установки обновлений, а также решает другие задачи по контролю защищенности АС. Управление агентами осуществляется по сети программа-менеджер. Менеджеры, являющиеся центральными компонентами подобных систем, посыпают управляющие команды всем агентам контролируемого ими домена и сохраняют все полученные от агентов данные в центральной базе данных. Администратор управляет менеджерами при помощи графической консоли, позволяющей выбирать, настраивать и создавать политики безопасности, анализировать изменения состояния системы, осуществлять ранжирование уязвимостей и т.п. Все взаимодействия между агентами, менеджерами и управляющей консолью осуществляются по защищенному клиент-серверному протоколу. Такой подход, например, использован при построении комплексной системы управления безопасностью организации Symantec Enterprise Security Manager (ESM).

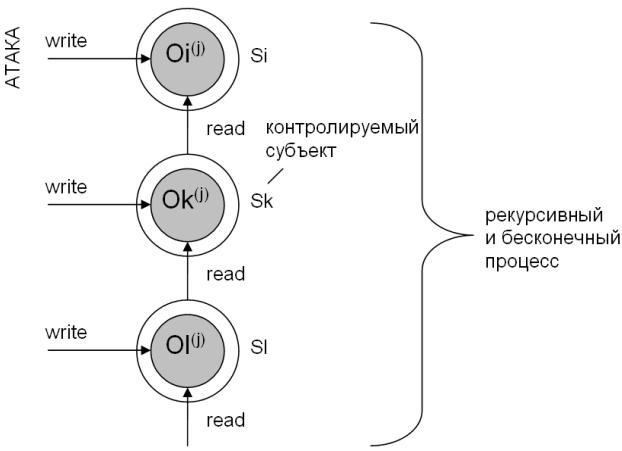
**9. Теорема о неразрешимости множества доверенных субъектов в вычислительной системе Фон Нэймановской архитектуры. Связь с одним из базовых принципов Фон Нэймановской архитектуры. Понятие доверенной аппаратной компоненты.**

**Определение:** Доверенный субъект – субъект с которым гарантирована целостность для всех ассоциированных с ним объектов.

**Теорема о неразрешимости множества доверенных субъектов в вычислительной системе Фон Нэймановской архитектуры.**

Множество доверенных субъектов в вычислительной системе(ВС), имеющих только оперативное запоминающее устройство(ОЗУ), не разрешимо.

Можно сказать и по другому, что В вычислительной системе(ВС), имеющей только оперативное запоминающее устройство(ОЗУ) не может быть доверенных субъектов.



**Определение:** множество является не разрешимым, если не существует алгоритма, способного за конечное время проверить, принадлежит ли данный элемент данному множеству.

**План доказательства:**

Предположим, что мы хотим обеспечить доверенность субъекта Si, т.е проверить отсутствие последствий от каких-либо доступов типа write, к объекту Oi(j) ассоциированным с этим субъектом.

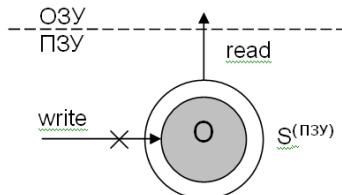
Для этого необходим субъект Sk, который осуществляет доступ read к объекту Oi(j) – проверяет его целостность.

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

Однако функции субъекта Sk могут быть нарушены в связи с доступом write, к ассоциированному к нему объекту Ok(j).

Таким образом целостность Ok(j) необходимо также контролировать при помощи субъекта Sl , для которого все предыдущие рассуждения могут быть повторены.

Т.е мы получим рекурсивную процедуру. Т.к. эта процедура бесконечна, то из этого следует, что подобный субъект создать нельзя.



Завершает порочный круг.

В теории изолированных программных сред(ИСП) доказывается, что для обеспечения существования доверенных субъектов в ВС необходим некий аппаратный компонент, целостность которого не требует доказательства.

**Определение:** Доверенная аппаратная компонента – это компонента, целостность которой не требует доказательства. Это понятие тесно связано с базовым принципом Фон-Неймановской архитектуры, по которому область программ и данных совмещена, и существует вероятность фальсификации любого субъекта. Поэтому нам нужна доверенная аппаратная компонента, в роли которой может выступать ROM.

## **10. Примеры аппаратных решений для создания изолированных программных сред.**

**Определение:** Изолированная(замкнутая) программная среда – среда, поведение которой может быть строго спрогнозировано формальными методами.

В изолированных программных средах не возможно существование разрушающих программных воздействий(Логических закладок, Троянских программ, Червей и Вирусов).

Но как же добиться создания изолированной программной среды. Задача обеспечения безопасного документооборота условно может быть поделена на две подзадачи:

1. создание безопасной среды для работы с документами.
2. защита документов на всех стадиях их жизненного цикла.

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

При этом в рамках решения первой подзадачи должны быть обеспечены:

- идентификация и аутентификация (ИА);
- контроль целостности состава компьютеров и ЛВС;
- контроль целостности ОС;
- контроль целостности прикладного ПО и данных;

Все это обеспечивается специализированными средствами защиты информации (СЗИ), относимыми к классу систем защиты от (несанкционированный доступ к информации) НСД.

В принципе, существует два вида подобного рода систем:

1. Программные
2. Аппаратные.

Можно строго доказать, что при использовании только программных средств защиты приемлемый уровень защищенности принципиально не может быть достигнут, а попытки применять программные средства для контроля программных же средств, по сути, аналогичны попытке барона Мюнхгаузена вытащить себя из болота за волосы.

А потому основной принцип построения защищенных систем - последовательный отказ от программных средств контроля как очевидно ненадежных и перенос наиболее критичных контрольных процедур на аппаратный уровень.

Пример, как должна создаваться изолированная программная среда (ИПС - среда, свободная от программных "закладок", позволяющих злоумышленникам преодолевать как парольную защиту, так и ЭЦП пакетов обмена) с использованием аппаратных СЗИ:

Процедура ИА должна выполняться до этапа загрузки ОС (идентификация должна осуществляться с применением отчужденного идентификатора, в качестве которого используются такие носители информации, как дискеты, "таблетки" Touch Memory и т. п.). При этом доступ средствами компьютера к БД ИА, хранимой в энергонезависимой памяти СЗИ, должен быть невозможен. Целостность ПО СЗИ (т.е. его защита от несанкционированных модификаций) должна обеспечиваться технологией изготовления СЗИ.

Контроль целостности аппаратной части компьютеров должен выполняться СЗИ до загрузки ОС (при этом должны контролироваться процессор, системный и дополнительный BIOS, вектора прерываний INT 13 и 40, КМОП-память).

Контроль целостности ЛВС должен обеспечиваться процедурой аутентификации сети с использованием рекомендованного варианта аппаратного датчика случайных

## МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

чисел, контролируемого системой рекомендованных (сертифицированных) тестов. Аутентификация должна выполняться на этапе подключения проверенного компьютера к сети и далее через заранее определенные администратором безопасности интервалы времени.

Контроль целостности системных областей и файлов ОС должен выполняться СЗИ до загрузки ОС, что обеспечивает чтение реальных данных. При этом должен выполняться разбор таких файловых систем, как FAT 12, FAT 16, FAT 32 (для DOS, Windows 3x, Windows 95/98), NTFS (для Windows NT), HPFS (для OS/2) и FreeBSD (для UNIX). Для контроля целостности должна применяться опубликованная хэш-функция, эталонное значение которой должно храниться в энергонезависимой памяти СЗИ, аппаратно защищенной от доступа из компьютера.

Для контроля целостности прикладного ПО и данных, которые при этом должны храниться в разных областях памяти, необходимо применять опубликованную хэш-функцию, эталонное значение которой должно аутентифицироваться с помощью отчужденного технического носителя информации. Только для данного вида контроля наряду с аппаратными средствами могут использоваться программные СЗИ, но при этом их целостность должна быть зафиксирована аппаратно на предыдущем этапе.

После того как создана ИПС, необходимо поддерживать ее изолированность, следя за тем, чтобы кроме проверенных программ в ней не запускалось никаких иных, и можно начинать работу с документами. При этом должны выполняться:

- аутентификация документа при его создании;
- защита документа при его передаче;
- аутентификация документа при обработке, хранении и исполнении;
- защита документа при доступе к нему из внешней среды.

Понятно, что проще и дешевле сразу создавать защищенный документооборот, правильно организуя сам процесс порождения документов с точки зрения безопасности информации, чем потом пытаться создавать разнообразные надстройки или менять технологию в уже работающих системах. Меры, способные обеспечить защищенность документа на всех этапах его жизненного цикла:

Для аутентификации документа при его создании должен аппаратно вырабатываться код аутентификации (КА) до того, как осуществляется запись копии документа на внешние

носители. При этом КА должен вырабатываться с привязкой либо к оператору, либо к соответствующей программной компоненте информационной системы в зависимости от того, кем из них он порождается.

Защита документов при их передаче по открытym каналам связи должна выполняться на основе применения сертифицированных криптографических средств.

При обработке, хранении и исполнении документа его защита должна осуществляться с применением двух КА - входного и выходного для каждого этапа (это означает, что в любой момент времени документ оказывается защищен двумя КА), причем КА должны вырабатываться аппаратно с привязкой к процедуре обработки документа. При этом должна реализовываться следующая последовательность действий: для поступившего на данную стадию обработки документа (снабженного КА и ЭЦП) вырабатывается КА2 и только после этого снимается ЭЦП. Далее на каждом следующем n-ом этапе вырабатывается КAn+1 и снимается КAn-1.

Поскольку принятие решения о доступе субъекта к документу связано с его правами доступа и очень слабо зависит от того, откуда субъект запрашивает сведения об объекте и с помощью каких средств он это делает, достижение необходимого уровня безопасности возможно только при реализации концепции функционально-распределенного межсетевого экрана с поддержкой семантического анализа данных на основе мандатного механизма.

## IX. ОПЕРАЦИОННЫЕ СИСТЕМЫ

### 1. Классификация ОС

Операционные системы могут различаться особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами), особенностями использованных методов проектирования, типами аппаратных платформ, областями использования и многими другими свойствами.

Ниже приведена классификация ОС по некоторым наиболее основным признакам.

#### **Особенности алгоритмов управления ресурсами**

От эффективности алгоритмов управления локальными ресурсами компьютера во многом зависит эффективность всей сетевой ОС в целом. Поэтому, характеризуя сетевую ОС, часто приводят важнейшие особенности реализации функций ОС по управлению процессорами, памятью, внешними устройствами

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

автономного компьютера. Так, например, в зависимости от особенностей использованного алгоритма управления процессором, операционные системы делят на многозадачные и однозадачные, многопользовательские и однопользовательские, на системы, поддерживающие многонитевую обработку и не поддерживающие ее, на многопроцессорные и однопроцессорные системы.

**Поддержка многозадачности.** По числу одновременно выполняемых задач операционные системы могут быть разделены на два класса:

- однозадачные (например, MS-DOS, MSX) и
- многозадачные (ОС EC, OS/2, UNIX, Windows 95).

Однозадачные ОС в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

Многозадачные ОС, кроме вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

**Поддержка многопользовательского режима.** По числу одновременно работающих пользователей ОС делятся на:

- однопользовательские (MS-DOS, Windows 3.x, ранние версии OS/2);
- многопользовательские (UNIX, Windows NT).

Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что не всякая многозадачная система является многопользовательской, и не всякая однопользовательская ОС является однозадачной.

**Вытесняющая и невытесняющая многозадачность.** Важнейшим разделяемым ресурсом является процессорное время. Способ распределения процессорного времени между несколькими одновременно существующими в системе процессами (или нитями) во многом определяет специфику ОС. Среди множества существующих вариантов реализации многозадачности можно выделить две группы алгоритмов:

- невытесняющая многозадачность (NetWare, Windows 3.x);
- вытесняющая многозадачность (Windows NT, OS/2, UNIX).

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Основным различием между вытесняющим и невытесняющим вариантами многозадачности является степень централизации механизма планирования процессов. В первом случае механизм планирования процессов целиком сосредоточен в операционной системе, а во втором - распределен между системой и прикладными программами. При невытесняющей многозадачности активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению процесс. При вытесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается операционной системой, а не самим активным процессом.

**Поддержка многонитевости.** Важным свойством операционных систем является возможность распараллеливания вычислений в рамках одной задачи. Многонитевая ОС разделяет процессорное время не между задачами, а между их отдельными ветвями (нитями).

**Многопроцессорная обработка.** Другим важным свойством ОС является отсутствие или наличие в ней средств поддержки многопроцессорной обработки - **мультипроцессирование**. Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами.

Многопроцессорные ОС могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой: асимметричные ОС и симметричные ОС. Асимметричная ОС целиком выполняется только на одном из процессоров системы, распределяя прикладные задачи по остальным процессорам. Симметричная ОС полностью децентрализована и использует весь пул процессоров, разделяя их между системными и прикладными задачами.

### **Особенности аппаратных платформ**

На свойства операционной системы непосредственное влияние оказывают аппаратные средства, на которые она ориентирована. По типу аппаратуры различают операционные системы персональных компьютеров, мини-компьютеров, мейнфреймов, кластеров и сетей ЭВМ. Среди перечисленных типов компьютеров могут встречаться как однопроцессорные варианты, так и многопроцессорные. В любом случае специфика аппаратных средств, как правило, отражается на специфике операционных систем.

Наряду с ОС, ориентированными на совершенно определенный тип аппаратной платформы, существуют операционные системы,

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

специально разработанные таким образом, чтобы они могли быть легко перенесены с компьютера одного типа на компьютер другого типа, так называемые *мобильные* ОС. Наиболее ярким примером такой ОС является популярная система UNIX. В этих системах аппаратно-зависимые места тщательно локализованы, так что при переносе системы на новую платформу переписываются только они. Средством, облегчающим перенос остальной части ОС, является написание ее на машинно-независимом языке, например, на C, который и был разработан для программирования операционных систем.

### **Особенности областей использования**

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности:

- системы пакетной обработки (например, ОС EC),
- системы разделения времени (UNIX, VMS),
- системы реального времени (QNX, RT/11).

*Системы пакетной обработки* предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени. Для достижения этой цели в системах пакетной обработки используются следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие отличающиеся требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины; так, например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом. Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается "выгодное" задание. Следовательно, в таких ОС невозможно гарантировать выполнение того или иного задания в течение определенного периода времени. В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит только в случае, если активная задача сама отказывается от процессора, например, из-за необходимости выполнить операцию ввода-вывода. Поэтому одна задача может надолго занять процессор, что делает невозможным выполнение

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

интерактивных задач. Таким образом, взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок снижает эффективность работы пользователя.

*Системы разделения времени* призваны исправить основной недостаток систем пакетной обработки - изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю системы разделения времени предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго, и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них единолично использует машину. Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая "выгодна" системе, и, кроме того, имеются накладные расходы вычислительной мощности на более частое переключение процессора с задачи на задачу. Критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя.

*Системы реального времени* применяются для управления различными техническими объектами, такими, например, как станок, спутник, научная экспериментальная установка или технологическими процессами, такими, как гальваническая линия, доменный процесс и т.п. Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная программа, управляющая объектом, в противном случае может произойти авария: спутник выйдет из зоны видимости, экспериментальные данные, поступающие с датчиков, будут потеряны, толщина гальванического покрытия не будет соответствовать норме. Таким образом, критерием эффективности для систем реального времени является их способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия). Это время называется временем реакции системы, а соответствующее свойство системы -

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

реактивностью. Для этих систем мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется исходя из текущего состояния объекта или в соответствии с расписанием плановых работ.

Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть задач может выполняться в режиме пакетной обработки, а часть - в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют фоновым режимом.

### **Особенности методов построения**

При описании операционной системы часто указываются особенности ее структурной организации и основные концепции, положенные в ее основу.

К таким базовым концепциям относятся:

- Способы построения ядра системы - монолитное ядро или микроядерный подход. Большинство ОС использует монолитное ядро, которое компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в пользовательский и наоборот. Альтернативой является построение ОС на базе микроядра, работающего также в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС - серверы, работающие в пользовательском режиме. При таком построении ОС работает более медленно, так как часто выполняются переходы между привилегированным режимом и пользовательским, зато система получается более гибкой - ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.
- Построение ОС на базе ООП дает возможность использовать все его достоинства, хорошо зарекомендовавшие себя на уровне приложений, внутри ОС, а именно: аккумуляцию удачных решений в форме стандартных объектов, возможность создания новых

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

- объектов на базе имеющихся с помощью мех-ма наследования, хорошую защиту данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкц. использования извне, структуризованность системы, сост. из набора хорошо определенных объектов.
- Наличие нескольких прикладных сред дает возможность в рамках одной ОС одновременно выполнять приложения, разработанные для нескольких ОС. Многие современные ОС поддерживают одновременно прикладные среды MS-DOS, Windows, UNIX (POSIX), OS/2 или хотя бы некоторого подмножества из этого популярного набора. Концепция множественных прикладных сред наиболее просто реализуется в ОС на базе микроядра, над которым работают различные серверы, часть которых реализуют прикладную среду той или иной ОС.
  - Распределенная организация операционной системы позволяет упростить работу пользователей и программистов в сетевых средах. В распределенной ОС реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера. Характерными признаками распределенной организации ОС являются: наличие единой справочной службы разделяемых ресурсов, единой службы времени, использование механизма вызова удаленных процедур (RPC) для прозрачного распределения программных процедур по машинам, многонитевой обработки, позволяющей распараллеливать вычисления в рамках одной задачи и выполнять эту задачу сразу на нескольких компьютерах сети, а также наличие других распределенных служб.

## **2. Структура сетевой операционной системы**

Сетевая операционная система составляет основу любой вычислительной сети. Каждый компьютер в сети в значительной степени автономен, поэтому под сетевой операционной системой в широком смысле понимается совокупность операционных систем отдельных компьютеров, взаимодействующих с целью обмена сообщениями и разделения ресурсов по единым правилам - протоколам. В узком смысле сетевая ОС - это операционная система отдельного компьютера, обеспечивающая ему возможность работать в сети.



Рис. 1.1. Структура сетевой ОС

В сетевой операционной системе отдельной машины можно выделить несколько частей (рисунок 1.1):

- Средства управления локальными ресурсами компьютера: функции распределения оперативной памяти между процессами, планирования и диспетчеризации процессов, управления процессорами в мультипроцессорных машинах, управления периферийными устройствами и другие функции управления ресурсами локальных ОС.
- Средства предоставления собственных ресурсов и услуг в общее пользование - серверная часть ОС (сервер). Эти средства обеспечивают, например, блокировку файлов и записей, что необходимо для их совместного использования; ведение справочников имен сетевых ресурсов; обработку запросов удаленного доступа к собственной файловой системе и базе данных; управление очередями запросов удаленных пользователей к своим периферийным устройствам.
- Средства запроса доступа к удаленным ресурсам и услугам и их использования - клиентская часть ОС (редиректор). Эта часть выполняет распознавание и перенаправление в сеть запросов к удаленным ресурсам от приложений и пользователей, при этом запрос поступает от приложения в локальной форме, а

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

передается в сеть в другой форме, соответствующей требованиям сервера. Клиентская часть также осуществляет прием ответов от серверов и преобразование их в локальный формат, так что для приложения выполнение локальных и удаленных запросов неразличимо.

- Коммуникационные средства ОС, с помощью которых происходит обмен сообщениями в сети. Эта часть обеспечивает адресацию и буферизацию сообщений, выбор маршрута передачи сообщения по сети, надежность передачи и т.п., то есть является средством транспортировки сообщений.

В зависимости от функций, возлагаемых на конкретный компьютер, в его операционной системе может отсутствовать либо клиентская, либо серверная части.

На рисунке 1.2 показано взаимодействие сетевых компонентов. Здесь компьютер 1 выполняет роль "чистого" клиента, а компьютер 2 - роль "чистого" сервера, соответственно на первой машине отсутствует серверная часть, а на второй - клиентская. На рисунке отдельно показан компонент клиентской части - редиректор. Именно редиректор перехватывает все запросы, поступающие от приложений, и анализирует их. Если выдан запрос к ресурсу данного компьютера, то он переадресовывается соответствующей подсистеме локальной ОС, если же это запрос к удаленному ресурсу, то он переправляется в сеть. При этом клиентская часть преобразует запрос из локальной формы в сетевой формат и передает его транспортной подсистеме, которая отвечает за доставку сообщений указанному серверу. Серверная часть операционной системы компьютера 2 принимает запрос, преобразует его и передает для выполнения своей локальной ОС. После того, как результат получен, сервер обращается к транспортной подсистеме и направляет ответ клиенту, выдавшему запрос. Клиентская часть преобразует результат в соответствующий формат и адресует его тому приложению, которое выдало запрос.

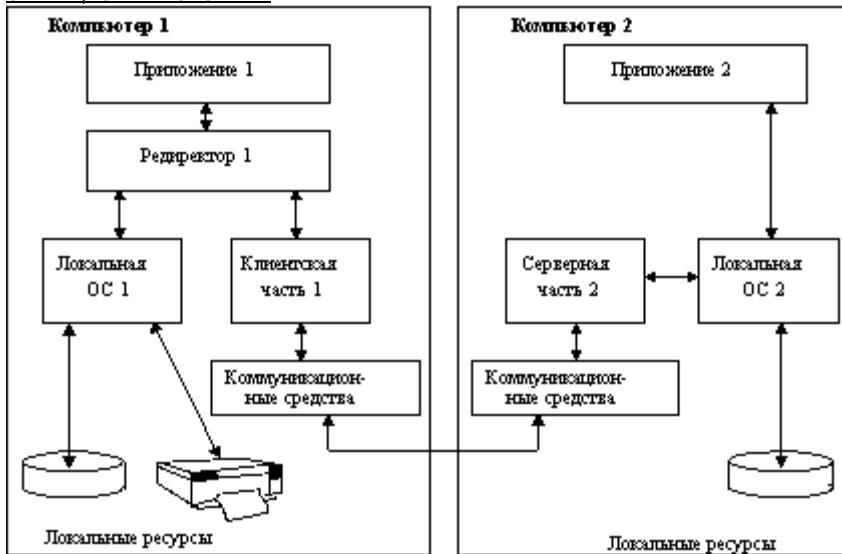


Рис. 1.2. взаимодействие компонентов операционной системы при взаимодействии компьютеров

На практике сложилось несколько подходов к построению сетевых операционных систем (рисунок 1.3).

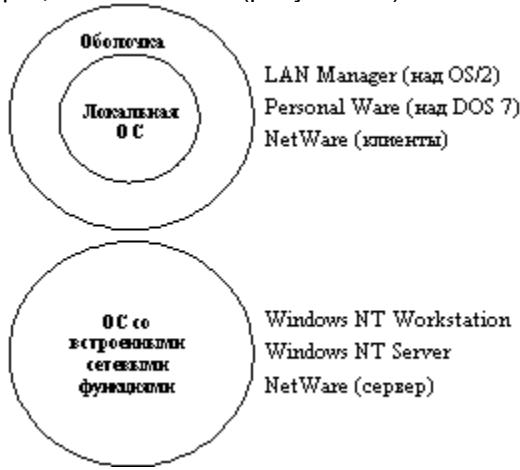


Рис. 1.3. Варианты построения сетевых ОС

Первые сетевые ОС представляли собой совокупность существующей локальной ОС и надстроенной над ней *сетевой оболочки*. При этом в локальную ОС встраивался минимум сетевых функций, необходимых для работы сетевой оболочки, которая выполняла основные сетевые функции. Примером такого

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

подхода является использование на каждой машине сети операционной системы MS DOS (у которой начиная с ее третьей версии появились такие встроенные функции, как блокировка файлов и записей, необходимые для совместного доступа к файлам). Принцип построения сетевых ОС в виде сетевой оболочки над локальной ОС используется и в современных ОС, таких, например, как LANtastic или Personal Ware.

Однако более эффективным представляется путь разработки операционных систем, изначально предназначенных для работы в сети. Сетевые функции у ОС такого типа глубоко *встроены* в основные модули системы, что обеспечивает их логическую стройность, простоту эксплуатации и модификации, а также высокую производительность. Примером такой ОС является система Windows NT фирмы Microsoft, которая за счет встроенности сетевых средств обеспечивает более высокие показатели производительности и защищенности информации по сравнению с сетевой ОС LAN Manager той же фирмы (совместная разработка с IBM), являющейся надстройкой над локальной операционной системой OS/2.

### **3. Управление процессами. Понятие процесса. Дескриптор и контекст процесса. Алгоритмы планирования процессов. Вытесняющая и не вытесняющая многозадачность.**

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами. *Процесс* (или по-другому, задача) - абстракция, описывающая выполняющуюся программу. Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

#### **Состояние процессов**

В многозадачной (многопроцессной) системе процесс может находиться в одном из трех основных состояний:

**ВЫПОЛНЕНИЕ** - активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

**ОЖИДАНИЕ** - пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам, он ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса;

**ГОТОВНОСТЬ** - также пассивное состояние процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он готов выполнятся, однако процессор занят выполнением другого процесса.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Типичный граф состояний процесса показан на рисунке 2.1.

В состоянии **ВЫПОЛНЕНИЕ** в однопроцессорной системе может находиться только один процесс, а в каждом из состояний **ОЖИДАНИЕ** и **ГОТОВНОСТЬ** - несколько процессов, эти процессы образуют очереди соответственно ожидающих и готовых процессов. Жизненный цикл процесса начинается с состояния **ГОТОВНОСТЬ**, когда процесс готов к выполнению и ждет своей очереди. При активизации процесс переходит в состояние **ВЫПОЛНЕНИЕ** и находится в нем до тех пор, пока либо он сам освободит процессор, перейдя в состояние **ОЖИДАНИЯ** какого-нибудь события, либо будет насильно "вытеснен" из процессора, например, вследствие исчерпания отведенного данному процессу кванта процессорного времени. В последнем случае процесс возвращается в состояние **ГОТОВНОСТЬ**. В это же состояние процесс переходит из состояния **ОЖИДАНИЕ**, после того, как ожидаемое событие произойдет.

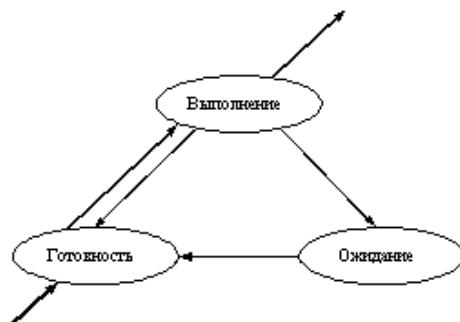


Рис. 2.1. Граф состояний процесса в многозадачной среде

### **Контекст и дескриптор процесса**

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того, чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды отображается состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и т.д. Эта информация называется *контекстом процесса*.

Кроме этого, операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и другая информация. В некоторых ОС (например, в ОС UNIX) информацию такого рода, используемую ОС для планирования процессов, называют *дескриптором процесса*.

Дескриптор процесса по сравнению с контекстом содержит более оперативную информацию, которая должна быть легко доступна подсистеме планирования процессов. Контекст процесса содержит менее актуальную информацию и используется операционной системой только после того, как принято решение о возобновлении прерванного процесса.

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки. Таким образом, каждый дескриптор, кроме всего прочего, содержит по крайней мере один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

Программный код только тогда начнет выполняться, когда для него операционной системой будет создан процесс. Создать процесс - это значит:

1. создать информационные структуры, описывающие данный процесс, то есть его дескриптор и контекст;
2. включить дескриптор нового процесса в очередь готовых процессов;
3. загрузить кодовый сегмент процесса в оперативную память или в область свопинга.

### **Алгоритмы планирования процессов**

Планирование процессов включает в себя решение следующих задач:

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

1. определение момента времени для смены выполняемого процесса;
2. выбор процесса на выполнение из очереди готовых процессов;
3. переключение контекстов "старого" и "нового" процессов.

Первые две задачи решаются программными средствами, а последняя в значительной степени аппаратно (см. раздел 2.3. "Средства аппаратной поддержки управления памятью и многозадачной среды в микропроцессорах Intel 80386, 80486 и Pentium").

Существует множество различных алгоритмов планирования процессов, по разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования. Среди этого множества алгоритмов рассмотрим подробнее две группы наиболее часто встречающихся алгоритмов: алгоритмы, основанные на **квантовании**, и алгоритмы, основанные на **приоритетах**.

В соответствии с алгоритмами, основанными на квантовании, смена активного процесса происходит, если:

- процесс завершился и покинул систему,
- произошла ошибка,
- процесс перешел в состояние ОЖИДАНИЕ,
- исчерпан квант процессорного времени, отведенный данному процессу.

Процесс, который исчерпал свой квант, переводится в состояние ГОТОВНОСТЬ и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени. Граф состояний процесса, изображенный на рисунке 2.1, соответствует алгоритму планирования, основанному на квантовании.

Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса. Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании. По разному может быть организована очередь готовых процессов: циклически, по

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

правилу "первый пришел - первый обслужился" (FIFO) или по правилу "последний пришел - первый обслужился" (LIFO).

Другая группа алгоритмов использует понятие "приоритет" процесса. *Приоритет* - это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии.

Приоритет может выражаться целыми или дробными, положительным или отрицательным значением. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может назначаться директивно администратором системы в зависимости от важности работы или внесенной платы, либо вычисляться самой ОС по определенным правилам, он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются динамическими.

Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты.

В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. По разному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ (или же произойдет ошибка, или процесс завершится). В системах с абсолютными приоритетами выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности. На рисунке 2.2 показаны графы состояний процесса для алгоритмов с относительными (а) и абсолютными (б) приоритетами.

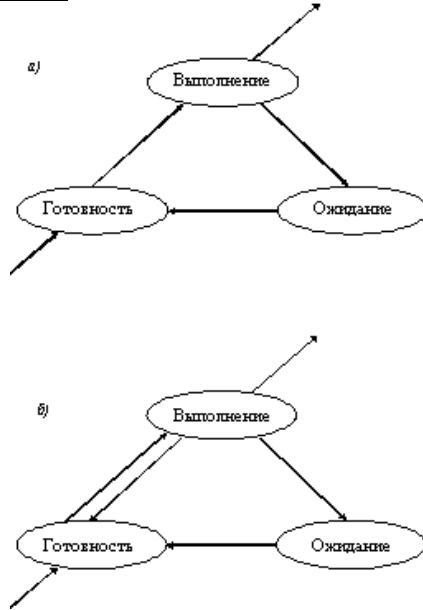


Рис. 2.2. Графы состояний процессов в системах  
(а) с относительными приоритетами; (б) с абсолютными приоритетами

Во многих ОС алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

#### Вытесняющие и невытесняющие алгоритмы планирования

Существует два основных типа процедур планирования процессов - вытесняющие (preemptive) и невытесняющие (non-preemptive).

*Non-preemptive multitasking* - невытесняющая многозадачность - это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

*Preemptive multitasking* - вытесняющая многозадачность - это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Понятия preemptive и non-preemptive иногда отождествляются с понятиями приоритетных и бесприоритетных дисциплин, что совершенно неверно, а также с понятиями абсолютных и относительных приоритетов, что неверно отчасти. Вытесняющая и невытесняющая многозадачность - это более широкие понятия, чем типы приоритетности. Приоритеты задач могут как использоваться, так и не использоваться и при вытесняющих, и при невытесняющих способах планирования. Так в случае использования приоритетов дисциплина относительных приоритетов может быть отнесена к классу систем с невытесняющей многозадачностью, а дисциплина абсолютных приоритетов - к классу систем с вытесняющей многозадачностью. А бесприоритетная дисциплина планирования, основанная на выделении равных квантов времени для всех задач, относится к вытесняющим алгоритмам.

Основным различием между preemptive и non-preemptive вариантами многозадачности является степень централизации механизма планирования задач. При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в ОС, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. При этом ОС выполняет следующие функции: определяет момент снятия с выполнения активной задачи, запоминает ее контекст, выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст. При невытесняющей многозадачности механизм планирования распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итерации и передает управление ОС с помощью какого-либо системного вызова, а ОС формирует очередь задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом приоритетов) следующую задачу на выполнение. Такой механизм создает проблемы как для пользователей, так и для разработчиков.

Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем). Если приложение тратит слишком много времени на выполнение какой-либо работы, например, на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу, например, на текстовый редактор, в то время как форматирование продолжалось бы в фоновом режиме. Эта

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу.

Поэтому разработчики приложений для non-preemptive операционной среды, возлагая на себя функции планировщика, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Например, программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить "дружественное" отношение своей программы к другим выполняемым одновременно с ней программам, достаточно часто отдавая им управление. Крайним проявлением "недружественности" приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения.

Однако распределение функций планировщика между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Так как разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в "неудобные" для них моменты времени. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные. Существенным преимуществом non-preemptive систем является более высокая скорость переключения с задачи на задачу.

## **4. Средства синхронизации взаимодействия процессов. Блокирующие переменные, семафоры**

### **Проблема синхронизации**

Процессам часто нужно взаимодействовать друг с другом, например, один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

синхронизации процессов, которая может решаться приостановкой и активизацией процессов, организацией очередей, блокированием и освобождением ресурсов.

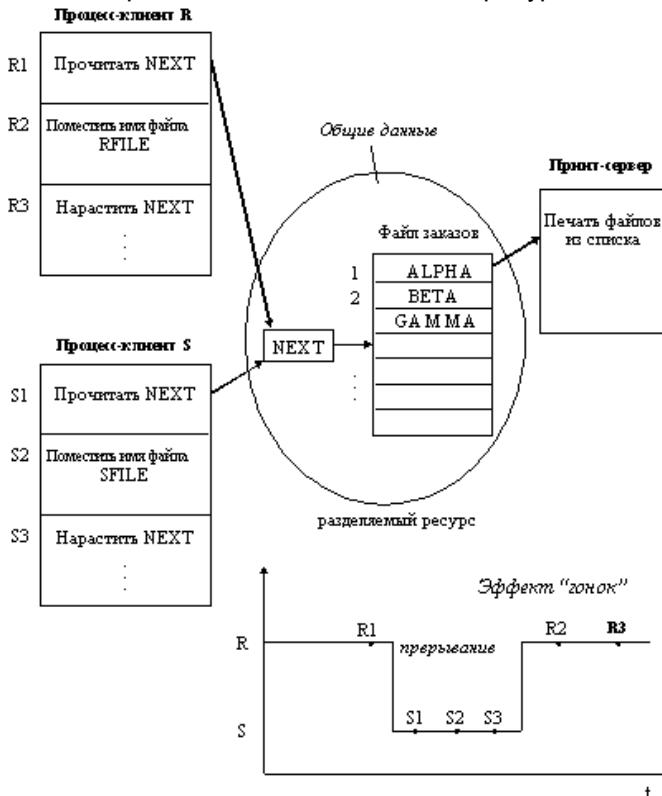


Рис. 2.3. Пример необходимости синхронизации

Пренебрежение вопросами синхронизации процессов, выполняющихся в режиме мультипрограммирования, может привести к их неправильной работе или даже к краху системы. Рассмотрим, например (рисунок 2.3), программу печати файлов (принт-сервер). Эта программа печатает по очереди все файлы, имена которых последовательно в порядке поступления записываются в специальный общедоступный файл "заказов" другие программы. Особая переменная NEXT, также доступная всем процессам-клиентам, содержит номер первой свободной для записи имени файла позиции файла "заказов". Процессы-клиенты читают эту переменную, записывают в соответствующую позицию файла "заказов" имя своего файла и наращивают значение NEXT на единицу. Предположим, что в некоторый момент процесс R решил распечатать свой файл, для этого он

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

прочитал значение переменной NEXT, значение которой для определенности предположим равным 4. Процесс запомнил это значение, но поместить имя файла не успел, так как его выполнение было прервано (например, в следствие исчерпания кванта). Очередной процесс S, желающий распечатать файл, прочитал то же самое значение переменной NEXT, поместил в четвертую позицию имя своего файла и нарастил значение переменной на единицу. Когда в очередной раз управление будет передано процессу R, то он, продолжая свое выполнение, в полном соответствии со значением текущей свободной позиции, полученным во время предыдущей итерации, запишет имя файла также в позицию 4, поверх имени файла процесса S.

Таким образом, процесс S никогда не увидит свой файл распечатанным. Сложность проблемы синхронизации состоит в нерегулярности возникающих ситуаций: в предыдущем примере можно представить и другое развитие событий: были потеряны файлы нескольких процессов или, напротив, не был потерян ни один файл. В данном случае все определяется взаимными скоростями процессов и моментами их прерывания. Поэтому отладка взаимодействующих процессов является сложной задачей. Ситуации подобные той, когда два или более процессов обрабатывают разделяемые данные, и конечный результат зависит от соотношения скоростей процессов, называются гонками.

### **Критическая секция**

Важным понятием синхронизации процессов является понятие "критическая секция" программы. *Критическая секция* - это часть программы, в которой осуществляется доступ к разделяемым данным. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы в каждый момент в критической секции, связанной с этим ресурсом, находился максимум один процесс. Этот прием называют взаимным исключением.

Простейший способ обеспечить взаимное исключение - позволить процессу, находящемуся в критической секции, запрещать все прерывания. Однако этот способ непригоден, так как опасно доверять управление системой пользовательскому процессу; он может надолго занять процессор, а при крахе процесса в критической области крах потерпит вся система, потому что прерывания никогда не будут разрешены.



Рис. 2.4. Реализация критических секций с использованием блокирующих переменных

Другим способом является использование блокирующих переменных. С каждым разделяемым ресурсом связывается двоичная переменная, которая принимает значение 1, если ресурс свободен (то есть ни один процесс не находится в данный момент в критической секции, связанной с данным процессом), и значение 0, если ресурс занят. На рисунке 2.4 показан фрагмент алгоритма процесса, использующего для реализации взаимного исключения доступа к разделяемому ресурсу D блокирующую переменную F(D). Перед входом в критическую секцию процесс проверяет, свободен ли ресурс D. Если он занят, то проверка циклически повторяется, если свободен, то значение переменной F(D) устанавливается в 0, и процесс входит в критическую секцию. После того, как процесс выполнит все действия с разделяемым ресурсом D, значение переменной F(D) снова устанавливается равным 1.

Если все процессы написаны с использованием вышеописанных соглашений, то взаимное исключение гарантируется. Следует заметить, что операция проверки и установки блокирующей переменной должна быть неделимой. Поясним это. Пусть в результате проверки переменной процесс определил, что ресурс свободен, но сразу после этого, не успев установить переменную

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

в 0, был прерван. За время его приостановки другой процесс занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому процессу, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом был нарушен принцип взаимного исключения, что потенциально может привести к нежелаемым последствиям. Во избежание таких ситуаций в системе команд машины желательно иметь единую команду "проверка-установка", или же реализовывать системными средствами соответствующие программные примитивы, которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Реализация критических секций с использованием блокирующих переменных имеет существенный недостаток: в течение времени, когда один процесс находится в критической секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной, бесполезно тратя процессорное время. Для устранения таких ситуаций может быть использован так называемый аппарат событий. С помощью этого средства могут решаться не только проблемы взаимного исключения, но и более общие задачи синхронизации процессов. В разных операционных системах аппарат событий реализуется по своему, но в любом случае используются системные функции аналогичного назначения, которые условно назовем WAIT(x) и POST(x), где x - идентификатор некоторого события. На рисунке 2.5 показан фрагмент алгоритма процесса, использующего эти функции. Если ресурс занят, то процесс не выполняет циклический опрос, а вызывает системную функцию WAIT(D), здесь D обозначает событие, заключающееся в освобождении ресурса D. Функция WAIT(D) переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его дескрипторе о том, что процесс ожидает события D. Процесс, который в это время использует ресурс D, после выхода из критической секции выполняет системную функцию POST(D), в результате чего операционная система просматривает очередь ожидающих процессов и переводит процесс, ожидающий события D, в состояние ГОТОВНОСТЬ.

Обобщающее средство синхронизации процессов предложил Дейкстра, который ввел два новых примитива. В абстрактной форме эти примитивы, обозначаемые P и V, оперируют над целыми неотрицательными переменными, называемыми *семафорами*. Пусть S такой семафор. Операции определяются следующим образом:

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

**V(S)** : переменная S увеличивается на 1 одним неделимым действием; выборка, инкремент и запоминание не могут быть прерваны, и к S нет доступа другим процессам во время выполнения этой операции.

**P(S)** : уменьшение S на 1, если это возможно. Если S=0, то невозможно уменьшить S и остаться в области целых неотрицательных значений, в этом случае процесс, вызывающий P-операцию, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также является неделимой операцией.

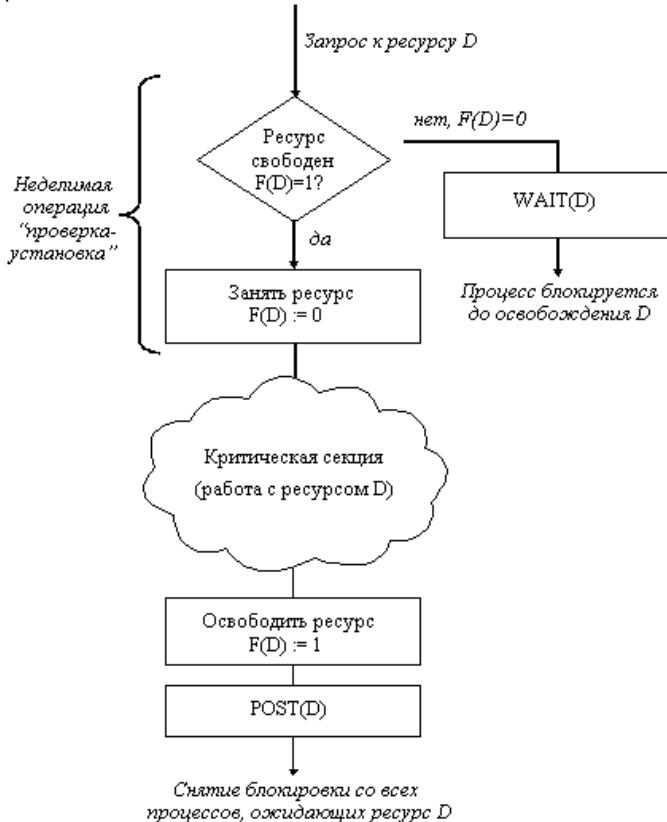


Рис. 2.5. Реализация критической секции с использованием системных функций *WAIT(D)* и *POST(D)*

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную. Операция P заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

ожидания, в то время как V-операция может при некоторых обстоятельствах активизировать другой процесс, приостановленный операцией P (сравните эти операции с системными функциями WAIT и POST).

### **5. Взаимные блокировки процессов. Тупики распознавание, рекомендации как избежать тупик, выход из тупика.**

Существует еще одна проблема синхронизации - **взаимные блокировки**, называемые также **дедлоками (deadlocks), клинчами (clinch)** или **тупиками**. При некотором стечении обстоятельств два процесса могут взаимно заблокировать друг друга. Действительно, пусть "писатель" первым войдет в критическую секцию и обнаружит отсутствие свободных буферов; он начнет ждать, когда "читатель" возьмет очередную запись из буфера, но "читатель" не сможет этого сделать, так как для этого необходимо войти в критическую секцию, вход в которую заблокирован процессом "писателем".

Рассмотрим пример тупика. Пусть двум процессам, выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например, принтер и диск. На рисунке 2.6,а показаны фрагменты соответствующих программ. И пусть после того, как процесс А занял принтер (установил блокирующую переменную), он был прерван. Управление получило процесс В, который сначала занял диск, но при выполнении следующей команды был заблокирован, так как принтер оказался уже занятым процессом А. Управление снова получил процесс А, который в соответствии со своей программой сделал попытку занять диск и был заблокирован: диск уже распределен процессу В. В таком положении процессы А и В могут находиться сколь угодно долго.

В зависимости от соотношения скоростей процессов, они могут либо совершенно независимо использовать разделяемые ресурсы (г), либо образовывать очереди к разделяемым ресурсам (в), либо взаимно блокировать друг друга (б). Тупиковые ситуации надо отличать от простых очередей, хотя и те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: процесс приостанавливается и ждет освобождения ресурса. Однако очередь - это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов. Она возникает тогда, когда ресурс недоступен в данный момент, но через некоторое время он освобождается, и процесс продолжает свое выполнение. Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

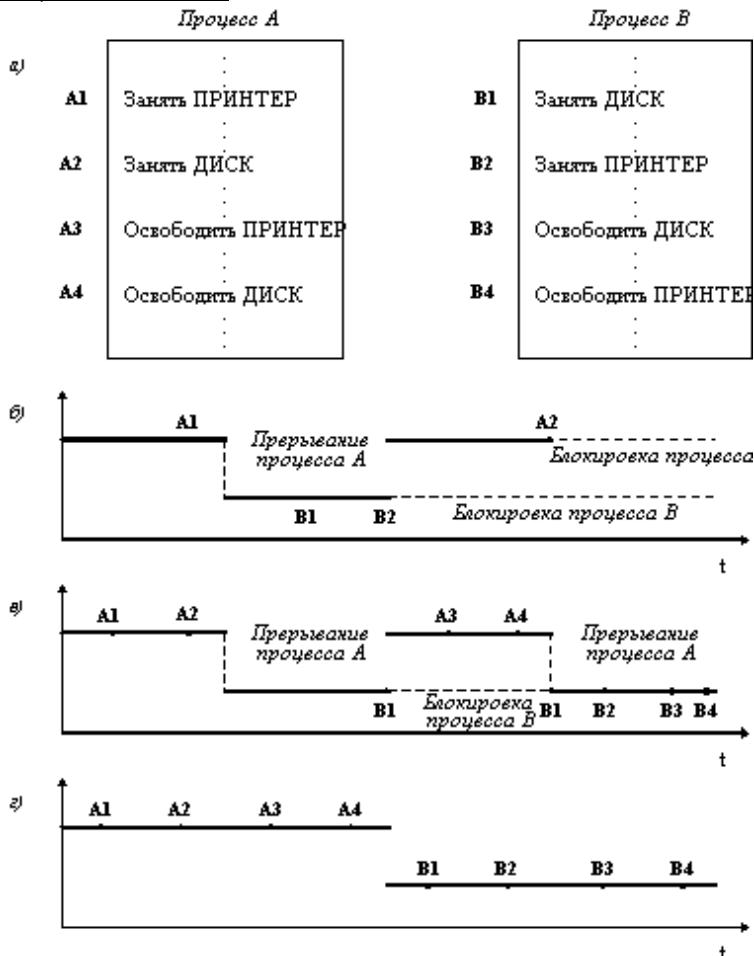


Рис. 2.6. (а) фрагменты программ A и B, разделяющих принтер и диск;  
 (б) взаимная блокировка (клинич); (в) очередь к разделяемому диску;  
 (г) независимое использование ресурсов

В рассмотренных примерах тупик был образован двумя

В рассмотренных примерах тупик был образован двумя процессами, но взаимно блокировать друг друга могут и большее число процессов.

Проблема тупиков включает в себя следующие задачи:

- предотвращение тупиков,
  - распознавание тупиков,
  - восстановление системы после тупиков.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Тупики могут быть предотвращены на стадии написания программ, то есть программы должны быть написаны таким образом, чтобы тупик не мог возникнуть ни при каком соотношении взаимных скоростей процессов. Так, если бы в предыдущем примере процесс А и процесс В запрашивали ресурсы в одинаковой последовательности, то тупик был бы в принципе невозможен. Второй подход к предотвращению тупиков называется динамическим и заключается в использовании определенных правил при назначении ресурсов процессам, например, ресурсы могут выделяться в определенной последовательности, общей для всех процессов.

В некоторых случаях, когда тупиковая ситуация образована многими процессами, использующими много ресурсов, распознавание тупика является нетривиальной задачей. Существуют формальные, программно-реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам. Анализ этих таблиц позволяет обнаружить взаимные блокировки. Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные процессы. Можно снять только часть из них, при этом освобождаются ресурсы, ожидаемые остальными процессами, можно вернуть некоторые процессы в область свопинга, можно совершить "откат" некоторых процессов до так называемой контрольной точки, в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места. Контрольные точки расставляются в программе в местах, после которых возможно возникновение тупика.

Из всего вышесказанного ясно, что использовать семафоры нужно очень осторожно, так как одна незначительная ошибка может привести к останову системы. Для того, чтобы облегчить написание корректных программ, было предложено высокоуровневое средство синхронизации, называемое монитором. *Монитор* - это набор процедур, переменных и структур данных. Процессы могут вызывать процедуры монитора, но не имеют доступа к внутренним данным монитора. Мониторы имеют важное свойство, которое делает их полезными для достижения взаимного исключения: только один процесс может быть активным по отношению к монитору. Компилятор обрабатывает вызовы процедур монитора особым образом. Обычно, когда процесс вызывает процедуру монитора, то первые несколько инструкций этой процедуры проверяют, не активен ли какой-либо другой процесс по отношению к этому монитору. Если да, то вызывающий процесс приостанавливается, пока другой

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

процесс не освободит монитор. Таким образом, исключение входа нескольких процессов в монитор реализуется не программистом, а компилятором, что делает ошибки менее вероятными.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, семафоры и мониторы оказываются непригодными. В таких системах синхронизация может быть реализована только с помощью обмена сообщениями.

## **6. Проблемы взаимодействия процессов. Основные задачи, возникающие при взаимодействии процессов.**

Проблемы синхронизации – см. вопрос №4.

Основные задачи:

- Описать критическую секцию (вопрос №4)
- Описать способы предотвращения тупиков и взаимные блокировки (вопрос №5)

## **7. Нити и процессы**

Многозадачность является важнейшим свойством ОС. Для поддержки этого свойства ОС определяет и оформляет для себя те внутренние единицы работы, между которыми и будет разделяться процессор и другие ресурсы компьютера. Эти внутренние единицы работы в разных ОС носят разные названия - задача, задание, процесс, нить. В некоторых случаях сущности, обозначаемые этими понятиями, принципиально отличаются друг от друга.

Говоря о процессах, мы отмечали, что операционная система поддерживает их обособленность: у каждого процесса имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы - файлы, окна, семафоры и т.д. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы машины, конкурируют с друг другом. В общем случае процессы принадлежат разным пользователям, разделяющим один компьютер, и ОС берет на себя роль арбитра в спорах процессов за ресурсы.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса). Однако задача, решаемая в рамках одного процесса, может обладать внутренним параллелизмом, который в принципе позволяет

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

ускорить ее решение. Например, в ходе выполнения задачи происходит обращение к внешнему устройству, и на время этой операции можно не блокировать полностью выполнение процесса, а продолжить вычисления по другой "ветви" процесса.

Для этих целей современные ОС предлагают использовать сравнительно новый механизм *многонитевой обработки* (*multithreading*). При этом вводится новое понятие "нить" (thread), а понятие "процесс" в значительной степени меняет смысл.

Мультипрограммирование теперь реализуется на уровне нитей, и задача, оформленная в виде нескольких нитей в рамках одного процесса, может быть выполнена быстрее за счет псевдопараллельного (или параллельного в мультипроцессорной системе) выполнения ее отдельных частей. Например, если электронная таблица была разработана с учетом возможностей многонитевой обработки, то пользователь может запросить пересчет своего рабочего листа и одновременно продолжать заполнять таблицу. Особенно эффективно можно использовать многонитевость для выполнения распределенных приложений, например, многонитевый сервер может параллельно выполнять запросы сразу нескольких клиентов.

Нити, относящиеся к одному процессу, не настолько изолированы друг от друга, как процессы в традиционной многозадачной системе, между ними легко организовать тесное взаимодействие. Действительно, в отличие от процессов, которые принадлежат разным, вообще говоря, конкурирующим приложениям, все нити одного процесса всегда принадлежат одному приложению, поэтому программист, пишущий это приложение, может заранее продумать работу множества нитей процесса таким образом, чтобы они могли взаимодействовать, а не бороться за ресурсы.

В традиционных ОС понятие "нить" тождественно понятию "процесс". В действительности часто бывает желательно иметь несколько нитей, разделяющих единое адресное пространство, но выполняющихся квазипараллельно, благодаря чему нити становятся подобными процессам (за исключением разделяемого адресного пространства).

Нити иногда называют облегченными процессами или минипроцессами. Действительно, нити во многих отношениях подобны процессам. Каждая нить выполняется строго последовательно и имеет свой собственный программный счетчик и стек. Нити, как и процессы, могут, например, порождать нити-потомки, могут переходить из состояния в состояние. Подобно традиционным процессам (то есть процессам, состоящим из одной нити), нити могут находиться в одном из следующих состояний: **ВЫПОЛНЕНИЕ**, **ОЖИДАНИЕ** и **ГОТОВНОСТЬ**. Пока одна нить

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так, как это делают процессы, в соответствии с различными вариантами планирования.

Однако различные нити в рамках одного процесса не настолько независимы, как отдельные процессы. Все такие нити имеют одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждая нить может иметь доступ к каждому виртуальному адресу, одна нить может использовать стек другой нити. Между нитями нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Все нити одного процесса всегда решают общую задачу одного пользователя, и аппарат нитей используется здесь для более быстрого решения задачи путем ее распараллеливания. При этом программисту очень важно получить в свое распоряжение удобные средства организации взаимодействия частей одной задачи. Кроме разделения адресного пространства, все нити разделяют также набор открытых файлов, таймеров, сигналов и т.п.

Итак, нити имеют собственные:

- программный счетчик,
- стек,
- регистры,
- нити-потомки,
- состояние.

Нити разделяют:

- адресное пространство,
- глобальные переменные,
- открытые файлы,
- таймеры,
- семафоры,
- статистическую информацию.

Многонитевая обработка повышает эффективность работы системы по сравнению с многозадачной обработкой. Например, в многозадачной среде Windows можно одновременно работать с электронной таблицей и текстовым редактором. Однако, если пользователь запрашивает пересчет своего рабочего листа, электронная таблица блокируется до тех пор, пока эта операция не завершится, что может потребовать значительного времени. В многонитевой среде в случае, если электронная таблица была разработана с учетом возможностей многонитевой обработки, предоставляемых программисту, этой проблемы не возникает, и пользователь всегда имеет доступ к электронной таблице.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Широкое применение находит многонитевая обработка в распределенных системах. Смотрите об этом в разделе "Процессы и нити в распределенных системах".

Некоторые прикладные задачи легче программировать, используя параллелизм, например задачи типа "писатель-читатель", в которых одна нить выполняет запись в буфер, а другая считывает записи из него. Поскольку они разделяют общий буфер, не стоит их делать отдельными процессами. Другой пример использования нитей - это управление сигналами, такими как прерывание с клавиатуры (del или break). Вместо обработки сигнала прерывания, одна нить назначается для постоянного ожидания поступления сигналов. Таким образом, использование нитей может сократить необходимость в прерываниях пользовательского уровня. В этих примерах не столь важно параллельное выполнение, сколь важна ясность программы.

Наконец, в мультипроцессорных системах для нитей из одного адресного пространства имеется возможность выполнять параллельно на разных процессорах. Это действительно один из главных путей реализации разделения ресурсов в таких системах. С другой стороны, правильно сконструированные программы, которые используют нити, должны работать одинаково хорошо как на однопроцессорной машине в режиме разделения времени между нитями, так и на настоящем мультипроцессоре.

## **8. Управление памятью. Типы адресов. Обзор методов распределения памяти.**

### **Управление памятью**

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Распределению подлежит вся оперативная память, не занятая операционной системой. Обычно ОС располагается в самых младших адресах, однако может занимать и самые старшие адреса. Функциями ОС по управлению памятью являются: отслеживание свободной и занятой памяти, выделение памяти процессам и освобождение памяти при завершении процессов, вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место, а также настройка адресов программы на конкретную область физической памяти.

**Типы адресов**

Для идентификации переменных и команд используются символьные имена (метки), виртуальные адреса и физические адреса (рисунок 2.7).

Символьные имена присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.

Виртуальные адреса вырабатывает транслятор, переводящий программу на машинный язык. Так как во время трансляции в общем случае не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что программа будет размещена, начиная с нулевого адреса. Совокупность виртуальных адресов процесса называется *виртуальным адресным пространством*. Каждый процесс имеет собственное виртуальное адресное пространство. Максимальный размер виртуального адресного пространства ограничивается разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

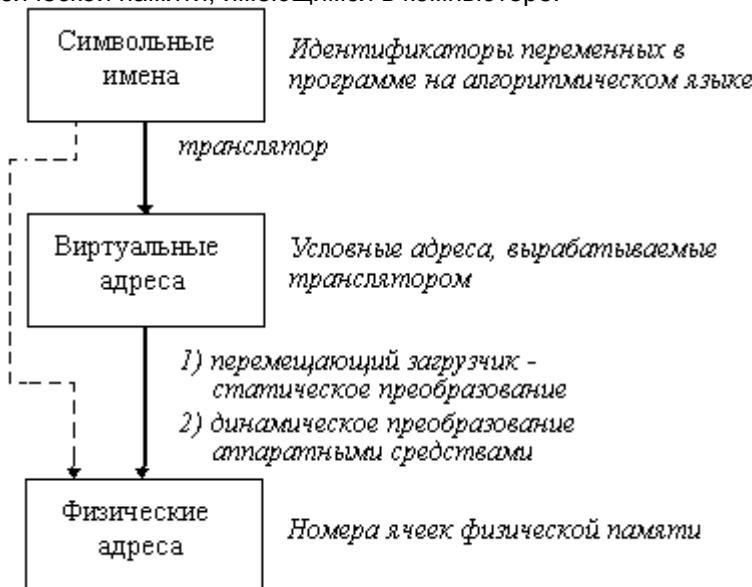


Рис. 2.7. Типы адресов

Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды. Переход от виртуальных адресов к физическим может осуществляться двумя способами. В первом случае замену виртуальных адресов на физические

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

делает специальная системная программа - перемещающий загрузчик. Перемещающий загрузчик на основании имеющихся у него исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, и информации, предоставленной транслятором об адресно-зависимых константах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, при этом операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Второй способ является более гибким, он допускает перемещение программы во время ее выполнения, в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти. Вместе с тем использование перемещающего загрузчика уменьшает накладные расходы, так как преобразование каждого виртуального адреса происходит только один раз во время загрузки, а во втором случае - каждый раз при обращении по данному адресу.

В некоторых случаях (обычно в специализированных системах), когда заранее точно известно, в какой области оперативной памяти будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

### **Обзор методов распределения памяти**

Все методы управления памятью могут быть разделены на два класса: методы, которые используют перемещение процессов между оперативной памятью и диском, и методы, которые не делают этого (рисунок 2.8).



Рис. 2.8. Классификация методов распределения памяти

## 9. Методы управления памятью без использования внешней памяти

### Распределение памяти фиксированными разделами

Самым простым способом управления оперативной памятью является разделение ее на несколько разделов фиксированной величины. Это может быть выполнено вручную оператором во время старта системы или во время ее генерации. Очередная задача, поступившая на выполнение, помещается либо в общую очередь (рисунок 2.9,а), либо в очередь к некоторому разделу (рисунок 2.9,б).

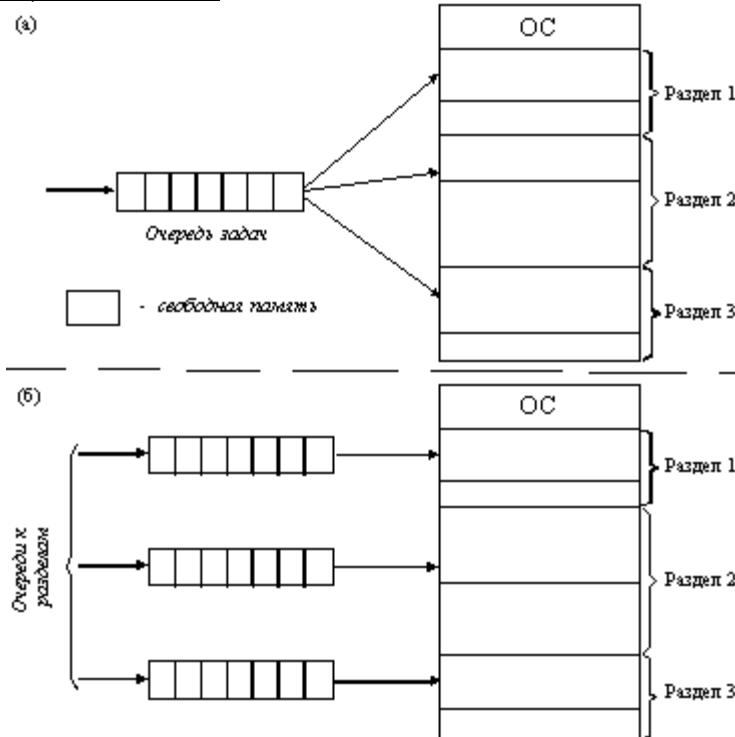


Рис. 2.9. Распределение памяти фиксированными разделами:  
а - с общей очередью; б - с отдельными очередями

Подсистема управления памятью в этом случае выполняет следующие задачи:

- сравнивая размер программы, поступившей на выполнение, и свободных разделов, выбирает подходящий раздел,
- осуществляет загрузку программы и настройку адресов.

При очевидном преимуществе - простоте реализации - данный метод имеет существенный недостаток - жесткость. Так как в каждом разделе может выполняться только одна программа, то уровень мультипрограммирования заранее ограничен числом разделов независимо от того, какой размер имеют программы. Даже если программа имеет небольшой объем, она будет занимать весь раздел, что приводит к неэффективному использованию памяти. С другой стороны, даже если объем оперативной памяти машины позволяет выполнить некоторую программу, разбиение памяти на разделы не позволяет сделать этого.

**Распределение памяти разделами переменной величины**

В этом случае память машины не делится заранее на разделы. Сначала вся память свободна. Каждой вновь поступающей задаче выделяется необходимая ей память. Если достаточный объем памяти отсутствует, то задача не принимается на выполнение и стоит в очереди. После завершения задачи память освобождается, и на это место может быть загружена другая задача. Таким образом, в произвольный момент времени оперативная память представляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера. На рисунке 2.10 показано состояние памяти в различные моменты времени при использовании динамического распределения. Так в момент  $t_0$  в памяти находится только ОС, а к моменту  $t_1$  память разделена между 5 задачами, причем задача П4, завершаясь, покидает память. На освободившееся после задачи П4 место загружается задача П6, поступившая в момент  $t_3$ .

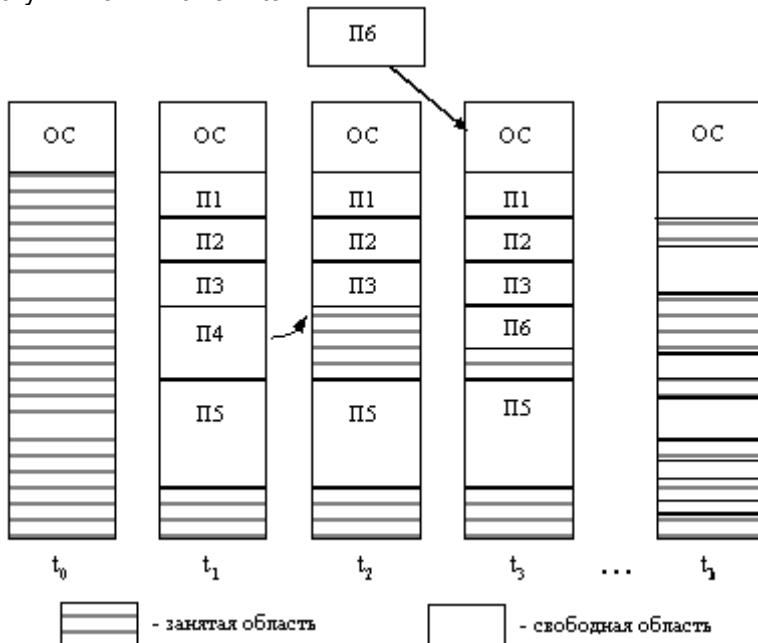


Рис. 2.10. Распределение памяти динамическими разделами  
Задачами операционной системы при реализации данного метода управления памятью является:

- ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти,
- при поступлении новой задачи - анализ запроса, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения поступившей задачи,
- загрузка задачи в выделенный ей раздел и корректировка таблиц свободных и занятых областей,
- после завершения задачи корректировка таблиц свободных и занятых областей.

Программный код не перемещается во время выполнения, то есть может быть проведена единовременная настройка адресов посредством использования перемещающего загрузчика.

Выбор раздела для вновь поступившей задачи может осуществляться по разным правилам, таким, например, как "первый попавшийся раздел достаточного размера", или "раздел, имеющий наименьший достаточный размер", или "раздел, имеющий наибольший достаточный размер". Все эти правила имеют свои преимущества и недостатки.

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток - *фрагментация памяти*. Фрагментация - это наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов). Настолько маленького, что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объем фрагментов может составить значительную величину, намного превышающую требуемый объем памяти.

#### *Перемещаемые разделы*

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших либо в сторону младших адресов, так, чтобы вся свободная память образовывала единую свободную область (рисунок 2.11). В дополнение к функциям, которые выполняет ОС при распределении памяти переменными разделами, в данном случае она должна еще время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется "сжатием". Сжатие может выполняться либо при каждом завершении задачи, либо только тогда, когда для вновь поступившей задачи нет свободного раздела

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц, а во втором - реже выполняется процедура сжатия. Так как программы перемещаются по оперативной памяти в ходе своего выполнения, то преобразование адресов из виртуальной формы в физическую должно выполняться динамическим способом.

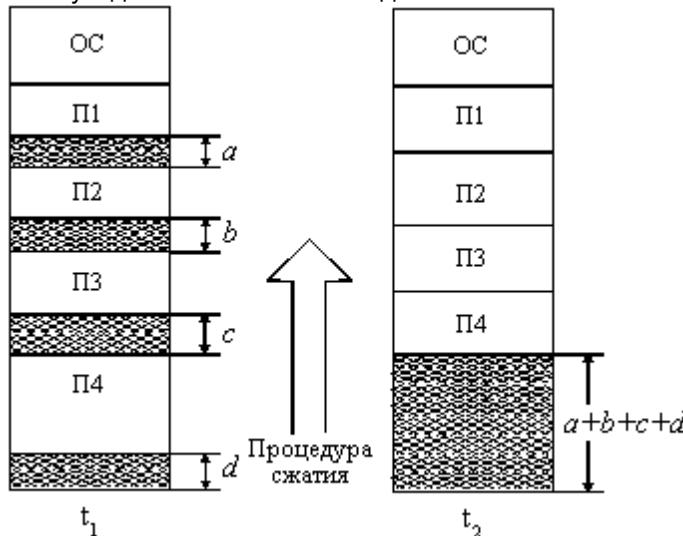


Рис. 2.11. Распределение памяти перемещаемыми разделами  
Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может потребовать значительного времени, что часто перевешивает преимущества данного метода.

## **10. Оверлеи. Виртуальная память. Способы организации виртуальной памяти**

### *Понятие виртуальной памяти*

Уже достаточно давно пользователи столкнулись с проблемой размещения в памяти программ, размер которых превышал имеющуюся в наличии свободную память. Решением было разбиение программы на части, называемые *оверлейами*. 0-ой оверлей начинал выполняться первым. Когда он заканчивал свое выполнение, он вызывал другой оверлей. Все оверлеи хранились на диске и перемещались между памятью и диском средствами операционной системы. Однако разбиение программы на части и планирование их загрузки в оперативную память должен был осуществлять программист.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Развитие методов организации вычислительного процесса в этом направлении привело к появлению метода, известного под названием **виртуальная память**. Виртуальным называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает. Так, например, пользователю может быть предоставлена виртуальная оперативная память, размер которой превосходит всю имеющуюся в системе реальную оперативную память. Пользователь пишет программы так, как будто в его распоряжении имеется однородная оперативная память большого объема, но в действительности все данные, используемые программой, хранятся на одном или нескольких разнородных запоминающих устройствах, обычно на дисках, и при необходимости частями отображаются в реальную память.

Таким образом, виртуальная память - это совокупность программно-аппаратных средств, позволяющих пользователям писать программы, размер которых превосходит имеющуюся оперативную память; для этого виртуальная память решает следующие задачи:

- размещает данные в запоминающих устройствах разного типа, например, часть программы в оперативной памяти, а часть на диске;
- перемещает по мере необходимости данные между запоминающими устройствами разного типа, например, подгружает нужную часть программы с диска в оперативную память;
- преобразует виртуальные адреса в физические.

Все эти действия выполняются *автоматически*, без участия программиста, то есть механизм виртуальной памяти является прозрачным по отношению к пользователю.

Наиболее распространенными реализациями виртуальной памяти является страничное, сегментное и странично-сегментное распределение памяти, а также свопинг.

### *Страницное распределение*

На рисунке 2.12 показана схема страницного распределения памяти. Виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые виртуальными страницами. В общем случае размер виртуального адресного пространства не является кратным размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками).

Размер страницы обычно выбирается равным степени двойки: 512, 1024 и т.д., это позволяет упростить механизм преобразования адресов.

При загрузке процесса часть его виртуальных страниц помещается в оперативную память, а остальные - на диск. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. При загрузке операционная система создает для каждого процесса информационную структуру - таблицу страниц, в которой устанавливается соответствие между номерами виртуальных и физических страниц для страниц, загруженных в оперативную память, или делается отметка о том, что виртуальная страница выгружена на диск. Кроме того, в таблице страниц содержится управляющая информация, такая как признак модификации страницы, признак невыгрузаемости (выгрузка некоторых страниц может быть запрещена), признак обращения к странице (используется для подсчета числа обращений за определенный период времени) и другие данные, формируемые и используемые механизмом виртуальной памяти.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Виртуальное адресное  
пространство процесса 1

0
1
2
3
4

Физическая  
область

Таблица страниц пр. 1

N в.с.	N ф.с.	Упр. инф.
0	5	
1	ВП	
2	ВП	
3	10	
4	2	

Виртуальное адресное  
пространство процесса 2

0
1
2
3
4
5

Таблица страниц пр. 2

N в.с.	N ф.с.	Упр. инф.
0	8	
1	ВП	
2	ВП	
3	ВП	
4	ВП	
5	11	

Физическая память	N физ. стр.
	0
	1
4 пр. 1	2
	3
	4
0 пр. 1	5
	6
	7
0 пр. 2	8
	9
	10
5 пр. 2	11
	12
	13
	14

$$V_{\text{вирт.стр.}} = V_{\text{физ.стр.}} = 2^k$$

Регистр адреса таблицы страниц



Страницочный обмен

Рис. 2.12. Страницочное распределение памяти

При активизации очередного процесса в специальный регистр процессора загружается адрес таблицы страниц данного процесса.

При каждом обращении к памяти происходит чтение из таблицы страниц информации о виртуальной странице, к которой произошло обращение. Если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое страницочное прерывание. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди готовых. Параллельно программа обработки страницочного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в оперативную память. Если в памяти имеется свободная

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то решается вопрос, какую страницу следует выгрузить из оперативной памяти.

В данной ситуации может быть использовано много разных критериев выбора, наиболее популярные из них следующие:

- дольше всего не использовавшаяся страница,
- первая попавшаяся страница,
- страница, к которой в последнее время было меньше всего обращений.

В некоторых системах используется понятие рабочего множества страниц. Рабочее множество определяется для каждого процесса и представляет собой перечень наиболее часто используемых страниц, которые должны постоянно находиться в оперативной памяти и поэтому не подлежат выгрузке.

После того, как выбрана страница, которая должна покинуть оперативную память, анализируется ее признак модификации (из таблицы страниц). Если выталкиваемая страница с момента загрузки была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то она может быть просто уничтожена, то есть соответствующая физическая страница объявляется свободной.

Рассмотрим механизм преобразования виртуального адреса в физический при страничной организации памяти (рисунок 2.13).

Виртуальный адрес при страничном распределении может быть представлен в виде пары ( $r, s$ ), где  $r$  - номер виртуальной страницы процесса (нумерация страниц начинается с 0), а  $s$  - смещение в пределах виртуальной страницы. Учитывая, что размер страницы равен 2 в степени  $k$ , смещение  $s$  может быть получено простым отделением  $k$  младших разрядов в двоичной записи виртуального адреса. Оставшиеся старшие разряды представляют собой двоичную запись номера страницы  $r$ .

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

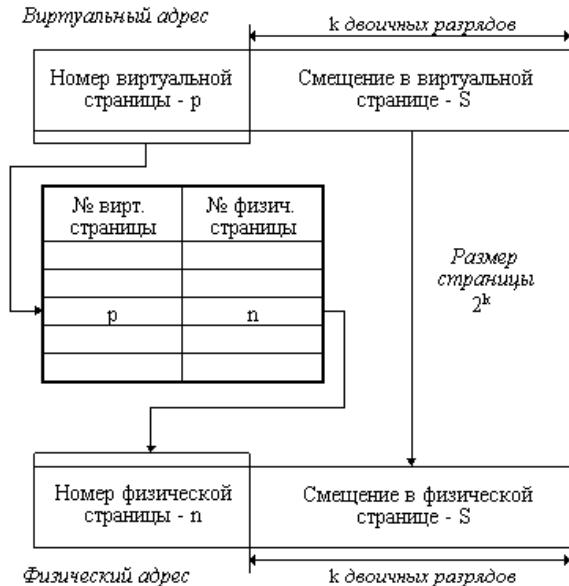


Рис. 2.13. Механизм преобразования виртуального адреса в физический

при страничной организации памяти

При каждом обращении к оперативной памяти аппаратными средствами выполняются следующие действия:

1. на основании начального адреса таблицы страниц (содержимое регистра адреса таблицы страниц), номера виртуальной страницы (старшие разряды виртуального адреса) и длины записи в таблице страниц (системная константа) определяется адрес нужной записи в таблице,
2. из этой записи извлекается номер физической страницы,
3. к номеру физической страницы присоединяется смещение (младшие разряды виртуального адреса).

Использование в пункте (3) того факта, что размер страницы равен степени 2, позволяет применить операцию конкатенации (присоединения) вместо более длительной операции сложения, что уменьшает время получения физического адреса, а значит повышает производительность компьютера.

На производительность системы со страничной организацией памяти влияют временные затраты, связанные с обработкой страничных прерываний и преобразованием виртуального адреса в физический. При часто возникающих страничных прерываниях система может тратить большую часть времени впустую, на свопинг страниц. Чтобы уменьшить частоту страничных

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

прерываний, следовало бы увеличивать размер страницы. Кроме того, увеличение размера страницы уменьшает размер таблицы страниц, а значит уменьшает затраты памяти. С другой стороны, если страница велика, значит велика и фиктивная область в последней виртуальной странице каждой программы. В среднем на каждой программе теряется половина объема страницы, что в сумме при большой странице может составить существенную величину. Время преобразования виртуального адреса в физический в значительной степени определяется временем доступа к таблице страниц. В связи с этим таблицу страниц стремятся размещать в "быстрых" запоминающих устройствах. Это может быть, например, набор специальных регистров или память, использующая для уменьшения времени доступа ассоциативный поиск и кэширование данных.

Страницное распределение памяти может быть реализовано в упрощенном варианте, без выгрузки страниц на диск. В этом случае все виртуальные страницы всех процессов постоянно находятся в оперативной памяти. Такой вариант страницной организации хотя и не предоставляет пользователю виртуальной памяти, но почти исключает фрагментацию за счет того, что программа может загружаться в несмежные области, а также того, что при загрузке виртуальных страниц никогда не образуется остатков.

### *Сегментное распределение*

При страницной организации виртуальное адресное пространство процесса делится механически на равные части. Это не позволяет дифференцировать способы доступа к разным частям программы (сегментам), а это свойство часто бывает очень полезным. Например, можно запретить обращаться с операциями записи и чтения в кодовый сегмент программы, а для сегмента данных разрешить только чтение. Кроме того, разбиение программы на "осмыслиенные" части делает принципиально возможным разделение одного сегмента несколькими процессами. Например, если два процесса используют одну и ту же математическую подпрограмму, то в оперативную память может быть загружена только одна копия этой подпрограммы.

Рассмотрим, каким образом сегментное распределение памяти реализует эти возможности (рисунок 2.14). Виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. Иногда сегментация программы выполняется по умолчанию компилятором.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система подыскивает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается начальный физический адрес сегмента в оперативной памяти, размер сегмента, правила доступа, признак модификации, признак обращения к данному сегменту за последний интервал времени и некоторая другая информация. Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

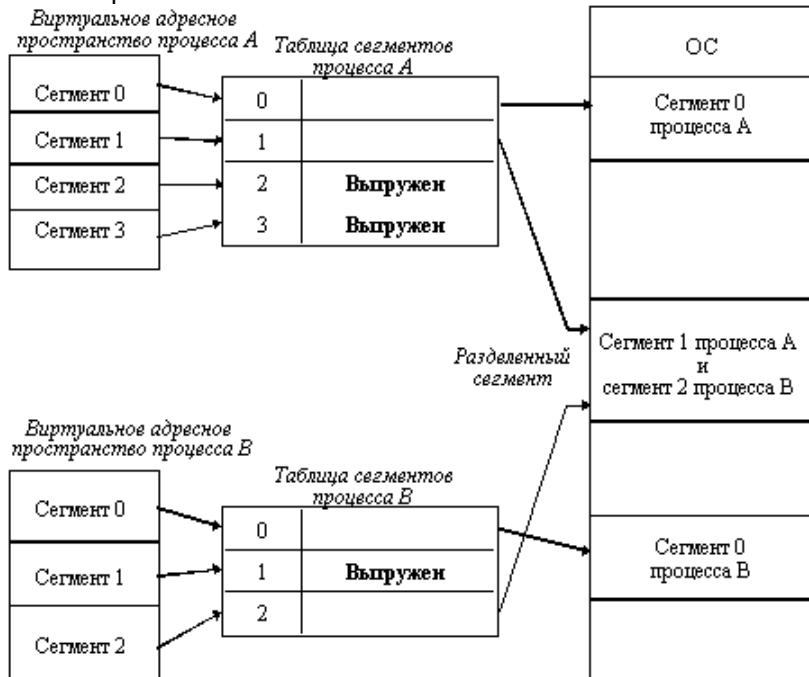


Рис. 2.14. Распределение памяти сегментами

Система с сегментной организацией функционирует аналогично системе со страничной организацией: времена от времени происходят прерывания, связанные с отсутствием нужных

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются, при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Кроме того, при обращении к памяти проверяется, разрешен ли доступ требуемого типа к данному сегменту.

Виртуальный адрес при сегментной организации памяти может быть представлен парой (g, s), где g - номер сегмента, а s - смещение в сегменте. Физический адрес получается путем сложения начального физического адреса сегмента, найденного в таблице сегментов по номеру g, и смещения s.

Недостатком данного метода распределения памяти является фрагментация на уровне сегментов и более медленное по сравнению со страничной организацией преобразование адреса.

### *Страницно-сегментное распределение*

Как видно из названия, данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс. На рисунке 2.15 показана схема преобразования виртуального адреса в физический для данного метода.

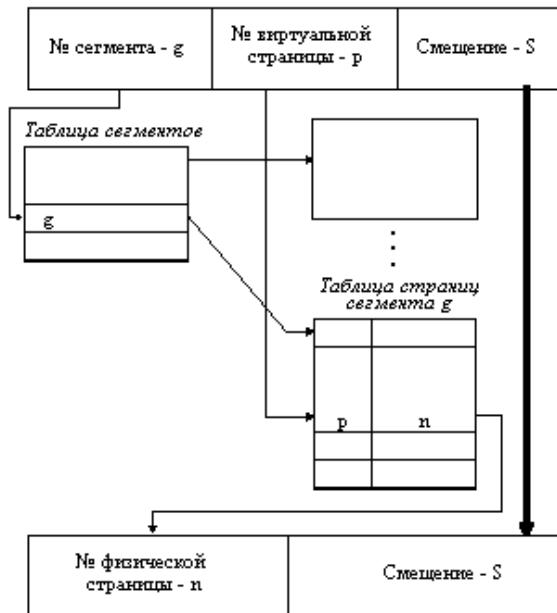
*Физический адрес*

Рис. 2.15. Схема преобразования виртуального адреса в физический для сегментно-страничной организации памяти

## 11. Свопинг и кэширование

### Свопинг

Разновидностью виртуальной памяти является свопинг.

На рисунке 2.16 показан график зависимости коэффициента загрузки процессора в зависимости от числа одновременно выполняемых процессов и доли времени, проводимого этими процессами в состоянии ожидания ввода-вывода.

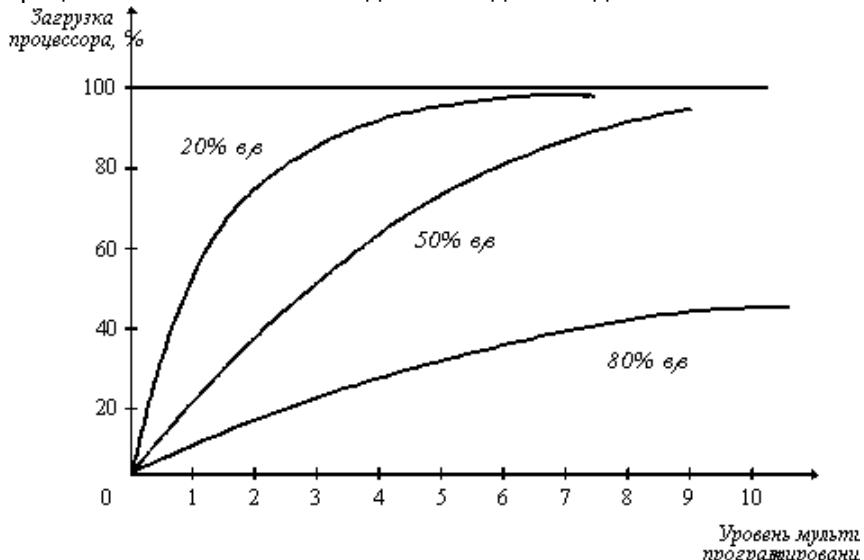


Рис. 2.16. Зависимость загрузки процессора от числа задач и интенсивности ввода-вывода

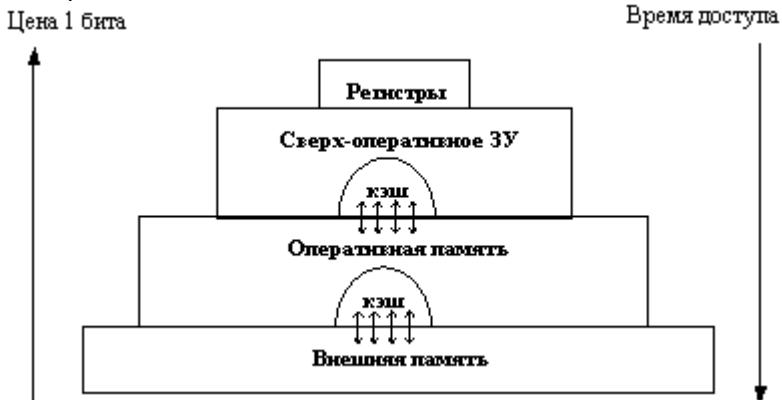
Из рисунка видно, что для загрузки процессора на 90% достаточно всего трех счетных задач. Однако для того, чтобы обеспечить такую же загрузку интерактивными задачами, выполняющими интенсивный ввод-вывод, потребуются десятки таких задач. Необходимым условием для выполнения задачи является загрузка ее в оперативную память, объем которой ограничен. В этих условиях был предложен метод организации вычислительного процесса, называемый свопингом. В соответствии с этим методом некоторые процессы (обычно находящиеся в состоянии ожидания) временно выгружаются на диск. Планировщик операционной системы не исключает их из своего рассмотрения, и при наступлении условий активизации некоторого процесса, находящегося в области свопинга на диске, этот процесс перемещается в оперативную память. Если свободного места в оперативной памяти не хватает, то выгружается другой процесс.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

При свопинге, в отличие от рассмотренных ранее методов реализации виртуальной памяти, процесс перемещается между памятью и диском целиком, то есть в течение некоторого времени процесс может полностью отсутствовать в оперативной памяти. Существуют различные алгоритмы выбора процессов на загрузку и выгрузку, а также различные способы выделения оперативной и дисковой памяти загружаемому процессу.

### **Иерархия запоминающих устройств. Принцип кэширования данных**

Память вычислительной машины представляет собой иерархию запоминающих устройств (внутренние регистры процессора, различные типы сверхоперативной и оперативной памяти, диски, ленты), отличающихся средним временем доступа и стоимостью хранения данных в расчете на один бит (рисунок 2.17). Пользователю хотелось бы иметь и недорогую и быструю память. Кэш-память представляет некоторое компромиссное решение этой проблемы.



*Рис. 2.17. Иерархия ЗУ*

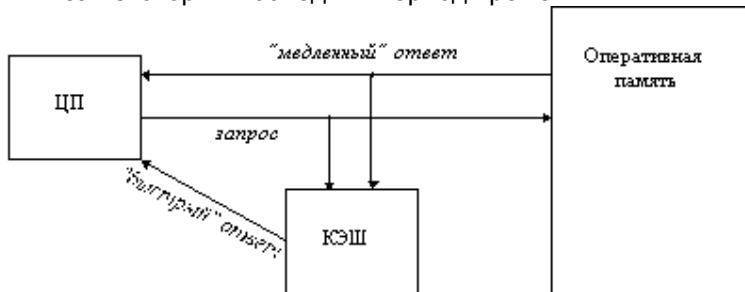
Кэш-память - это способ организации совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который позволяет уменьшить среднее время доступа к данным за счет динамического копирования в "быстрое" ЗУ наиболее часто используемой информации из "медленного" ЗУ.

Кэш-память часто называют не только способ организаций работы двух типов запоминающих устройств, но и одно из устройств - "быстрое" ЗУ. Оно стоит дороже и, как правило, имеет сравнительно небольшой объем. Важно, что механизм кэш-памяти является прозрачным для пользователя, который не должен сообщать никакой информации об интенсивности

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

использования данных и не должен никак участвовать в перемещении данных из ЗУ одного типа в ЗУ другого типа, все это делается автоматически системными средствами.

Рассмотрим частный случай использования кэш-памяти для уменьшения среднего времени доступа к данным, хранящимся в оперативной памяти. Для этого между процессором и оперативной памятью помещается быстрое ЗУ, называемое просто кэш-памятью (рисунок 2.18). В качестве такового может быть использована, например, ассоциативная память. Содержимое кэш-памяти представляет собой совокупность записей обо всех загруженных в нее элементах данных. Каждая запись об элементе данных включает в себя адрес, который этот элемент данных имеет в оперативной памяти, и управляющую информацию: признак модификации и признак обращения к данным за некоторый последний период времени.



Структура кэш-памяти

Адрес данных в ОП	Данные	Управл. информация	
		Бит модиф.	Бит обращ.

Рис. 2.18. Кэш-память

В системах, оснащенных кэш-памятью, каждый запрос к оперативной памяти выполняется в соответствии со следующим алгоритмом:

1. Просматривается содержимое кэш-памяти с целью определения, не находятся ли нужные данные в кэш-памяти; кэш-память не является адресуемой, поэтому поиск нужных

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

- данных осуществляется по содержимому - значению поля "адрес в оперативной памяти", взятому из запроса.
2. Если данные обнаруживаются в кэш-памяти, то оничитываются из нее, и результат передается в процессор.
  3. Если нужных данных нет, то они вместе со своим адресом копируются из оперативной памяти в кэш-память, и результат выполнения запроса передается в процессор. При копировании данных может оказаться, что в кэш-памяти нет свободного места, тогда выбираются данные, к которым в последний период было меньше всего обращений, для вытеснения из кэш-памяти. Если вытесняемые данные были модифицированы за время нахождения в кэш-памяти, то они переписываются в оперативную память. Если же эти данные не были модифицированы, то их место в кэш-памяти объявляется свободным.

В реальных системах вероятность попадания в кэш составляет примерно 0,9. Высокое значение вероятности нахождения данных в кэш-памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

- *Пространственная локальность.* Если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.
- *Временная локальность.* Если произошло обращение по некоторому адресу, то следующее обращение по этому же адресу с большой вероятностью произойдет в ближайшее время.

## **12. ОС. Управление вводом-выводом**

Одной из главных функций ОС является управление всеми устройствами ввода-вывода компьютера. ОС должна передавать устройствам команды, перехватывать прерывания и обрабатывать ошибки; она также должна обеспечивать интерфейс между устройствами и остальной частью системы. В целях развития интерфейс должен быть одинаковым для всех типов устройств (независимость от устройств).

### **Физическая организация устройств ввода-вывода**

Устройства ввода-вывода делятся на два типа: блок-ориентированные устройства и байт-ориентированные устройства. Блок-ориентированные устройства хранят информацию в блоках фиксированного размера, каждый из которых имеет свой собственный адрес. Самое распространенное блок-ориентированное устройство - диск. Байт-ориентированные устройства не адресуемы и не позволяют

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

производить операцию поиска, они генерируют или потребляют последовательность байтов. Примерами являются терминалы, строчные принтеры, сетевые адаптеры. Однако некоторые внешние устройства не относятся ни к одному классу, например, часы, которые, с одной стороны, не адресуемы, а с другой стороны, не порождают потока байтов. Это устройство только выдает сигнал прерывания в некоторые моменты времени.

Внешнее устройство обычно состоит из механического и электронного компонента. Электронный компонент называется контроллером устройства или адаптером. Механический компонент представляет собственно устройство. Некоторые контроллеры могут управлять несколькими устройствами. Если интерфейс между контроллером и устройством стандартизован, то независимые производители могут выпускать совместимые как контроллеры, так и устройства.

Операционная система обычно имеет дело не с устройством, а с контроллером. Контроллер, как правило, выполняет простые функции, например, преобразует поток бит в блоки, состоящие из байт, и осуществляют контроль и исправление ошибок. Каждый контроллер имеет несколько регистров, которые используются для взаимодействия с центральным процессором. В некоторых компьютерах эти регистры являются частью физического адресного пространства. В таких компьютерах нет специальных операций ввода-вывода. В других компьютерах адреса регистров ввода-вывода, называемых часто портами, образуют собственное адресное пространство за счет введения специальных операций ввода-вывода (например, команд IN и OUT в процессорах i86).

### **Организация программного обеспечения ввода-вывода**

Основная идея организации программного обеспечения ввода-вывода состоит в разбиении его на несколько уровней, причем нижние уровни обеспечивают экранирование особенностей аппаратуры от верхних, а те, в свою очередь, обеспечивают удобный интерфейс для пользователей.

Ключевым принципом является независимость от устройств. Вид программы не должен зависеть от того, читает ли она данные с гибкого диска или с жесткого диска.

Другим важным вопросом для программного обеспечения ввода-вывода является обработка ошибок. Ошибки следует обрабатывать как можно ближе к аппаратуре. Если контроллер обнаруживает ошибку чтения, то он должен попытаться ее скорректировать. Если же это ему не удается, то исправлением ошибок должен заняться драйвер устройства.

Еще один ключевой вопрос - это использование блокирующих (синхронных) и неблокирующих (асинхронных) передач.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Последняя проблема состоит в том, что одни устройства являются разделяемыми, а другие - выделенными. Диски - это разделяемые устройства, так как одновременный доступ нескольких пользователей к диску не представляет собой проблему. Принтеры - это выделенные устройства, потому что нельзя смешивать строчки, печатаемые различными пользователями. Наличие выделенных устройств создает для операционной системы некоторые проблемы.

Для решения поставленных проблем целесообразно разделить программное обеспечение ввода-вывода на четыре слоя (рисунок 2.30):

- Обработка прерываний,
- Драйверы устройств,
- Независимый от устройств слой операционной системы,
- Пользовательский слой программного обеспечения.



Рис. 2.30. Многоуровневая организация подсистемы ввода-вывода

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

### **Обработка прерываний**

Прерывания должны быть скрыты как можно глубже в недрах операционной системы, чтобы как можно меньшая часть ОС имела с ними дело. Наилучший способ состоит в разрешении процессу, инициировавшему операцию ввода-вывода, блокировать себя до завершения операции и наступления прерывания. Процесс может блокировать себя, используя, например, вызов DOWN для семафора, или вызов WAIT для переменной условия, или вызов RECEIVE для ожидания сообщения. При наступлении прерывания процедура обработки прерывания выполняет разблокирование процесса, инициировавшего операцию ввода-вывода, используя вызовы UP, SIGNAL или посыпая процессу сообщение. В любом случае эффект от прерывания будет состоять в том, что ранее заблокированный процесс теперь продолжит свое выполнение.

### **Драйверы устройств**

Весь зависимый от устройства код помещается в драйвер устройства. Каждый драйвер управляет устройствами одного типа или, может быть, одного класса.

В операционной системе только драйвер устройства знает о конкретных особенностях какого-либо устройства. Например, только драйвер диска имеет дело с дорожками, секторами, цилиндрами, временем установления головки и другими факторами, обеспечивающими правильную работу диска.

Драйвер устройства принимает запрос от устройств программного слоя и решает, как его выполнить. Типичным запросом является чтение п блоков данных.

### **Независимый от устройств слой операционной системы**

Типичными функциями для независимого от устройств слоя являются:

- обеспечение общего интерфейса к драйверам устройств,
- именование устройств,
- защита устройств,
- обеспечение независимого размера блока,
- буферизация,
- распределение памяти на блок-ориентированных устройствах,
- распределение и освобождение выделенных устройств,
- уведомление об ошибках.

### **Пользовательский слой программного обеспечения**

Системные вызовы, включающие вызовы ввода-вывода, обычно делаются библиотечными процедурами. Если программа, написанная на языке С, содержит вызов

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

count = write (fd, buffer, nbytes),

то библиотечная процедура write будет связана с программой. Набор подобных процедур является частью системы ввода-вывода.

Другой категорией программного обеспечения ввода-вывода является подсистема спуллинга (spooling). Спулинг - это способ работы с выделенными устройствами в мультипрограммной системе. Типичное устройство, требующее спуллинга - строчный принтер.

### **13. Файловая система. Основные функции. Общая схема.**

*Файловая система* - это часть операционной системы, назначение которой состоит в том, чтобы обеспечить пользователю удобный интерфейс при работе с данными, хранящимися на диске, и обеспечить совместное использование файлов несколькими пользователями и процессами.

В широком смысле понятие "файловая система" включает:

- совокупность всех файлов на диске,
- наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске,
- комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и другие операции над файлами.

#### **Типы файлов:**

- обычные файлы - текстовые и двоичные
- специальные файлы - это файлы, ассоциированные с устройствами ввода-вывода, которые позволяют пользователю выполнять операции ввода-вывода, используя обычные команды записи в файл или чтения из файла
- файлы-каталоги - это, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений (например, файлы, содержащие программы игр, или файлы, составляющие один программный пакет), а с другой стороны - это файл, содержащий системную информацию о группе файлов, его составляющих.

ОПЕРАЦИОННЫЕ СИСТЕМЫ



Рис. 2.31. Структура каталогов: а - структура записи каталога MS-DOS (32 байта); б - структура записи каталога ОС UNIX

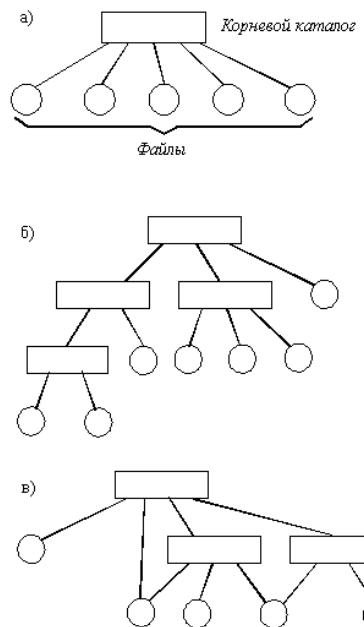


Рис. 2.32. Логическая организация файловой системы  
а - одноуровневая; б - иерархическая (дерево); в - иерархическая  
(сеть)

## Общая модель файловой системы

Функционирование любой файловой системы можно представить многоуровневой моделью (рисунок 2.36), в которой каждый уровень предоставляет некоторый интерфейс (набор функций) вышестоящему уровню, а сам, в свою очередь, для выполнения своей работы использует интерфейс (обращается с набором запросов) нижележащего уровня.

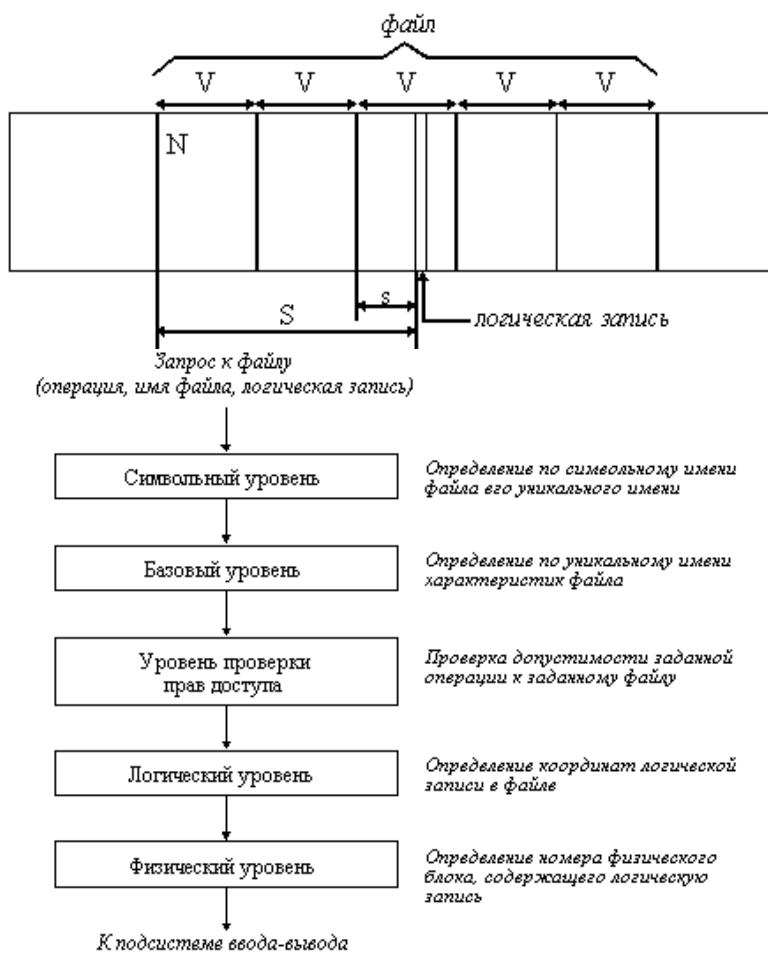


Рис. 2.36. Общая модель файловой системы

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Задачей символьного уровня является определение по символьному имени файла его уникального имени. В файловых системах, в которых каждый файл может иметь только одно символьное имя (например, MS-DOS), этот уровень отсутствует, так как символьное имя, присвоенное файлу пользователем, является одновременно уникальным и может быть использовано операционной системой. В других файловых системах, в которых один и тот же файл может иметь несколько символьных имен, на данном уровне просматривается цепочка каталогов для определения уникального имени файла. В файловой системе UNIX, например, уникальным именем является номер индексного дескриптора файла (i-node).

На следующем, базовом уровне по уникальному имени файла определяются его характеристики: права доступа, адрес, размер и другие. Как уже было сказано, характеристики файла могут входить в состав каталога или храниться в отдельных таблицах. При открытии файла его характеристики перемещаются с диска в оперативную память, чтобы уменьшить среднее время доступа к файлу. В некоторых файловых системах (например, HPFS) при открытии файла вместе с его характеристиками в оперативную память перемещаются несколько первых блоков файла, содержащих данные.

Следующим этапом реализации запроса к файлу является проверка прав доступа к нему. Для этого сравниваются полномочия пользователя или процесса, выдавших запрос, со списком разрешенных видов доступа к данному файлу. Если запрашиваемый вид доступа разрешен, то выполнение запроса продолжается, если нет, то выдается сообщение о нарушении прав доступа.

На логическом уровне определяются координаты запрашиваемой логической записи в файле, то есть требуется определить, на каком расстоянии (в байтах) от начала файла находится требуемая логическая запись. При этом абстрагируются от физического расположения файла, он представляется в виде непрерывной последовательности байт. Алгоритм работы данного уровня зависит от логической организации файла. Например, если файл организован как последовательность логических записей фиксированной длины  $l$ , то  $n$ -ая логическая запись имеет смещение  $l((n-1)$  байт. Для определения координат логической записи в файле с индексно-последовательной организацией выполняется чтение таблицы индексов (ключей), в которой непосредственно указывается адрес логической записи.

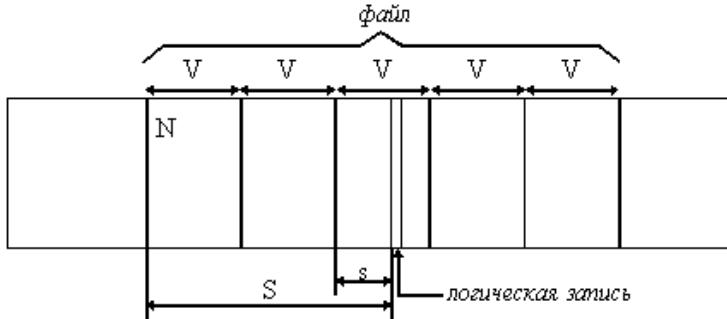


Рис. 2.37. Функции физического уровня файловой системы

#### 14. Логическая и физическая организация файлов. Права доступа к файлу. Кэширование файла. Отображение файла в оперативную память. Проблемы совместного использования.

##### Логическая организация файла

Программист имеет дело с логической организацией файла, представляя файл в виде определенным образом организованных логических записей. Логическая запись - это наименьший элемент данных, которым может оперировать программист при обмене с внешним устройством. Даже если физический обмен с устройством осуществляется большими единицами, операционная система обеспечивает программисту доступ к отдельной логической записи. На рисунке 2.33 показаны несколько схем логической организации файла. Записи могут быть фиксированной длины или переменной длины. Записи могут быть расположены в файле последовательно (последовательная организация) или в более сложном порядке, с использованием так называемых индексных таблиц, позволяющих обеспечить быстрый доступ к отдельной логической записи (индексно-последовательная организация). Для идентификации записи может быть использовано специальное поле записи, называемое ключом. В файловых системах ОС UNIX и MS-DOS файл имеет простейшую логическую структуру - последовательность однобайтовых записей.



Индекс	1	2	3	4	5	6
Адрес	21	201	315	661	670	715

Индекс = ключ

Рис. 2.33. Способы логической организации файлов

### Физическая организация и адрес файла

Физическая организация файла описывает правила расположения файла на устройстве внешней памяти, в частности на диске. Файл состоит из физических записей - блоков. Блок - наименьшая единица данных, которой внешнее устройство обменивается с оперативной памятью. Непрерывное размещение - простейший вариант физической организации (рисунок 2.34,а), при котором файлу предоставляется последовательность блоков диска, образующих единый сплошной участок дисковой памяти. Для задания адреса файла в этом случае достаточно указать только номер начального блока. Другое достоинство этого метода - простота. Но имеются и два существенных недостатка. Во-первых, во время создания файла заранее не известна его длина, а значит не известно, сколько памяти надо зарезервировать для этого файла, во-вторых, при таком порядке размещения неизбежно возникает фрагментация, и пространство на диске используется не эффективно, так как отдельные участки маленького размера (минимально 1 блок) могут остаться не используемыми.

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

Следующий способ физической организации - размещение в виде связанных списков блоков дисковой памяти (рисунок 2.34, б). При таком способе в начале каждого блока содержится указатель на следующий блок. В этом случае адрес файла также может быть задан одним числом - номером первого блока. В отличие от предыдущего способа, каждый блок может быть присоединен в цепочку какого-либо файла, следовательно фрагментация отсутствует. Файл может изменяться во время своего существования, наращивая число блоков. Недостатком является сложность реализации доступа к произвольно заданному месту файла: для того, чтобы прочитать пятый по порядку блок файла, необходимо последовательно прочитать четыре первых блока, прослеживая цепочку номеров блоков. Кроме того, при этом способе количество данных файла, содержащихся в одном блоке, не равно степени двойки (одно слово израсходовано на номер следующего блока), а многие программы читают данные блоками, размер которых равен степени двойки.

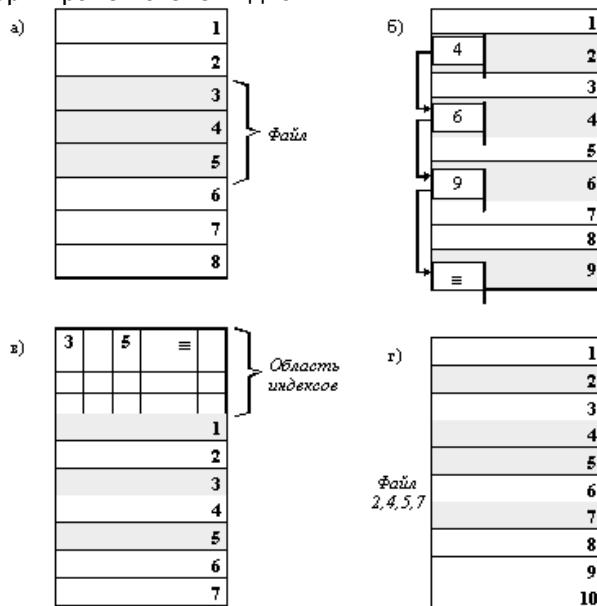


Рис. 2.34. Физическая организация файла

а - непрерывное размещение; б - связанный список блоков;  
в - связанный список индексов; г - перечень номеров блоков

### **Права доступа к файлу**

Определить права доступа к файлу - значит определить для каждого пользователя набор операций, которые он может применить к данному файлу. В разных файловых системах может

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

быть определен свой список дифференцируемых операций доступа. Этот список может включать следующие операции:

- создание файла,
- уничтожение файла,
- открытие файла,
- закрытие файла,
- чтение файла,
- запись в файл,
- дополнение файла,
- поиск в файле,
- получение атрибутов файла,
- установление новых значений атрибутов,
- переименование,
- выполнение файла,
- чтение каталога,

и другие операции с файлами и каталогами.

В самом общем случае права доступа могут быть описаны матрицей прав доступа, в которой столбцы соответствуют всем файлам системы, строки - всем пользователям, а на пересечении строк и столбцов указываются разрешенные операции (рисунок 2.35). В некоторых системах пользователи могут быть разделены на отдельные категории. Для всех пользователей одной категории определяются единые права доступа. Например, в системе UNIX все пользователи подразделяются на три категории: владельца файла, членов его группы и всех остальных.

*Имена файлов*

	modern.txt	win.exe	class.dbf	uchik.ppt
kira	читать	выполнять	-	выполнять
genya	читать	выполнять	-	выполнять читать
nataly	читать	-	-	выполнять читать
victor	читать писать	-	создать	-

*Рис. 2.35. Матрица прав доступа*

Различают два основных подхода к определению прав доступа:

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

- избирательный доступ, когда для каждого файла и каждого пользователя сам владелец может определить допустимые операции;
- мандатный подход, когда система наделяет пользователя определенными правами по отношению к каждому разделяемому ресурсу (в данном случае файлу) в зависимости от того, к какой группе пользователь отнесен.

### **Кэширование диска**

В некоторых файловых системах запросы к внешним устройствам, в которых адресация осуществляется блоками (диски, ленты), перехватываются промежуточным программным слоем-подсистемой буферизации. Подсистема буферизации представляет собой буферный пул, располагающийся в оперативной памяти, и комплекс программ, управляющих этим пулом. Каждый буфер пула имеет размер, равный одному блоку. При поступлении запроса на чтение некоторого блока подсистема буферизации просматривает свой буферный пул и, если находит требуемый блок, то копирует его в буфер запрашивающего процесса. Операция ввода-вывода считается выполненной, хотя физического обмена с устройством не происходило. Очевиден выигрыш во времени доступа к файлу. Если же нужный блок в буферном пуле отсутствует, то он считывается с устройства и одновременно с передачей запрашивающему процессу копируется в один из буферов подсистемы буферизации. При отсутствии свободного буфера на диск вытесняется наименее используемая информация. Таким образом, подсистема буферизации работает по принципу кэш-памяти.

### **Отображаемые в память файлы**

По сравнению с доступом к памяти, традиционный доступ к файлам выглядит запутанным и неудобным. По этой причине некоторые ОС, начиная с MULTICS, обеспечивают отображение файлов в адресное пространство выполняемого процесса. Это выражается в появлении двух новых системных вызовов: MAP (отобразить) и UNMAP (отменить отображение). Первый вызов передает операционной системе в качестве параметров имя файла и виртуальный адрес, и операционная система отображает указанный файл в виртуальное адресное пространство по указанному адресу.

Предположим, например, что файл  $f$  имеет длину 64 К и отображается на область виртуального адресного пространства с начальным адресом 512 К. После этого любая машинная команда, которая читает содержимое байта по адресу 512 К,

## ОПЕРАЦИОННЫЕ СИСТЕМЫ

получает 0-ой байт этого файла и т.д. Очевидно, что запись по адресу 512 К + 1100 изменяет 1100 байт файла. При завершении процесса на диске остается модифицированная версия файла, как если бы он был изменен комбинацией вызовов SEEK и WRITE.

В действительности при отображении файла внутренние системные таблицы изменяются так, чтобы данный файл служил хранилищем страниц виртуальной памяти на диске. Таким образом, чтение по адресу 512 К вызывает страничный отказ, в результате чего страница 0 переносится в физическую память. Аналогично, запись по адресу 512 К + 1100 вызывает страничный отказ, в результате которого страница, содержащая этот адрес, перемещается в память, после чего осуществляется запись в память по требуемому адресу. Если эта страница вытесняется из памяти алгоритмом замены страниц, то она записывается обратно в файл в соответствующее его место. При завершении процесса все отображенные и модифицированные страницы переписываются из памяти в файл.

Отображение файлов лучше всего работает в системе, которая поддерживает сегментацию. В такой системе каждый файл может быть отображен в свой собственный сегмент, так что k-ый байт в файле является k-ым байтом сегмента. На рисунке 2.38, а изображен процесс, который имеет два сегмента-кода и данных. Предположим, что этот процесс копирует файлы. Для этого он сначала отображает файл-источник, например, abc. Затем он создает пустой сегмент и отображает на него файл назначения, например, файл ddd.

С этого момента процесс может копировать сегмент-источник в сегмент-приемник с помощью обычного программного цикла, использующего команды пересылки в памяти типа mov. Никакие вызовы READ или WRITE не нужны. После выполнения копирования процесс может выполнить вызов UNMAP для удаления файла из адресного пространства, а затем завершиться. Выходной файл ddd будет существовать на диске, как если бы он был создан обычным способом.

Хотя отображение файлов исключает потребность в выполнении ввода-вывода и тем самым облегчает программирование, этот способ порождает и некоторые новые проблемы. Во-первых, для системы сложно узнать точную длину выходного файла, в данном примере ddd. Проще указать наибольший номер записанной страницы, но нет способа узнать, сколько байт в этой странице было записано. Предположим, что программа использует только страницу номер 0, и после выполнения все байты все еще установлены в значение 0 (их начальное значение). Быть может,

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

файл состоит из 10 нулей. А может быть, он состоит из 100 нулей. Как это определить? Операционная система не может это сообщить. Все, что она может сделать, так это создать файл, длина которого равна размеру страницы.



Рис. 2.38. (а) Сегменты процесса перед отображением файлов в адресное пространство; (б) Процесс после отображения существующего файла abc в один сегмент и создания нового сегмента для файла ddd

Вторая проблема проявляется (потенциально), если один процесс отображает файл, а другой процесс открывает его для обычного файлового доступа. Если первый процесс изменяет страницу, то это изменение не будет отражено в файле на диске до тех пор, пока страница не будет вытеснена на диск. Поддержание согласованности данных файла для этих двух процессов требует от системы больших забот.

Третья проблема состоит в том, что файл может быть больше, чем сегмент, и даже больше, чем все виртуальное адресное пространство. Единственный способ ее решения состоит в реализации вызова MAP таким образом, чтобы он мог отображать не весь файл, а его часть. Хотя такая работа, очевидно, менее удобна, чем отображение целого файла.

## X. ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

### 1. Операции над формальными языками

С развитием вычислительной техники появилась необходимость в теории формальных языков и грамматик - теории, которая бы позволяла описывать и анализировать синтаксические свойства языков программирования; теория, которая бы позволяла преобразовывать грамматики (КС) в автоматы (с магазинной памятью), чтобы автоматы распознавали и транслировали множества, задаваемые грамматиками.

Любой язык программирования (алгоритмический язык) можно понимать как множество цепочек, задаваемое некоторым

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

множеством правил. Множество цепочек, символов называется формальным языком.

Формальная грамматика - это набор грамматических правил, с помощью которых можно порождать и анализировать цепочки формального языка.

В грамматике имеются определенные правила, содержащие информацию о том, как из этих символов можно строить предложения языка.

В общем виде правила грамматики можно записать :

<нетерминал>: $\rightarrow$

<любая\_конечная\_цепочка\_терминальных\_и\_нетерминальных\_символов>

<нетерминал>: $\rightarrow$  <цепочка\_терминалов>

<нетерминал>: $\rightarrow \epsilon$  (Эпсилон - правило)

Контекстно – свободная грамматика (КСГ) задаётся:

- конечное множество терминальных символов;
- конечное множество нетерминальных символов;
- конечное множество правил вывода:

вида :  $\langle A \rangle \rightarrow a$ , где A- нетерминал, a – цепочка нетерминальных и терминальных символов (возможно пустая) или цепочка терминальных символов;

нетерминал A называется левой частью правила, а a – правой;

- аксиома грамматики – один нетерминальный символ, выделенный в качестве начального;

Правила грамматики задают способы подстановки цепочек. Подстановка осуществляется заменой нетерминального символа в заданной цепочке на правую часть правила, левой частью которого является такой нетерминал.

Пример:

$S \rightarrow aAbS$

$S \rightarrow b$

$A \rightarrow Sac$

$A \rightarrow \epsilon$ , где  $\langle S \rangle$  - начальный символ,  $\{A,S\}$  – множество нетерминальных символов,  $\{a,b,c\}$  - множество терминальных символов

*Основные понятия теории формальных языков и грамматики.*

Язык, задаваемый грамматикой, есть множество терминальных цепочек, которые можно вывести из начального символа грамматики.

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

Для каждого дерева существует единственный правый или левый выводы, т.е. вывод, когда на каждом шаге заменяется самый левый или правый нетерминальный символ.

Цепочке языков может соответствовать более чем одно дерево. Т.к. она может иметь разные выводы, порождающие разные деревья. Если одна цепочка имеет несколько деревьев вывода, следует, что соответствующая грамматика неоднозначна.

$V$  – алфавит терминальных символов.

$V^*$  – множество всех конечных слов или цепочек в алфавите  $V$ .

Формальный язык  $L$  над алфавитом  $V$  – это произвольное подмножество множества  $V^*$ , то есть  $L(V) \subseteq V^*$

Конструктивное описание формального языка осуществляется с помощью формальных систем, называемых формальными порождающими грамматиками.

1) Формальной порождающей грамматикой  $G$  – формальная система, описываемая с помощью четырёх формальных объектов  $\{V, W, P, S\}$

где  $V$  – словарь терминал. сим-в;  $W$  – словарь нетерминал. сим-в, причём  $V \cap W = \emptyset$ ;  $P$  – множество правил вида  $\varphi \rightarrow \psi$ , где  $\varphi$  и  $\psi \in (V \cup W)^*$ ;  $S$  – аксиомы грамматики.

2) Цепочка  $\beta$  выводится из цепочки  $\alpha$ , если они представимы в виде:

$\beta = \alpha \varphi \delta$   $\alpha = \lambda \psi \delta$  и в грамматике существует правило вида  $\psi \rightarrow \varphi$ .

3) Цепочка  $\beta$  называется выводимой из  $\alpha$ , если существует конечная цепочка вывода:  $\alpha \rightarrow \xi_0; \xi_0 \rightarrow \xi_1; \xi_1 \rightarrow \xi_2; \dots, \xi_n \rightarrow \beta$ , где  $\xi_i$  – цепочка нетерминальных символов  $\in (V^* \cup W^*)$ .

$\alpha \rightarrow G \beta$ :  $\beta$  выводима из  $\alpha$  в грамматике  $G$ .

4) Языком  $L$ , порождаемым грамматикой  $G$ , называется множество всех цепочек, выводимых из аксиомы грамматики.

5) Грамматики  $G_1$  и  $G_2$  эквивалентны, когда они порождают один и тот же язык.

### *Операции над языками.*

Пусть  $L_0$  - язык, заданный грамматикой  $G_0=\{V_0, W_0, R_0, I_0\}$ , а  $\in V_0$ ;

$L_1$ - язык, заданный грамматикой  $G_1=\{V_1, W_1, R_1, I_1\}$ ;

#### **1. Подстановка.**

Подстановка языка  $L$  в  $L_0$  вместо символа  $a$ –операция, сопоставляющая языкам  $L_0$  и  $L_1$  язык  $L=L_0$  ( $a \rightarrow L_1$ ), состоящий из всех цепочек языка  $L_0$ , в которых вместо символа  $a$  подставлена некоторая цепочка из  $L_1$ .

Пример:

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

$$L=L_0(a \rightarrow L_1)$$

$$L=L_0(a_1 \rightarrow L_1, a_2 \rightarrow L_2 \dots a_k \rightarrow L_k)$$

$$L_0=(a, (a+a), (a+a+a) \dots)$$

$$L_1=(b, bb, bbb \dots)$$

$$L=(b, bb, \dots, b+b, (b+b+b) \dots)$$

### **2. Конкатенация.**

Конкатенация языков  $L_0$  и  $L_1$ - это операция, сопоставляющая языки  $L_0$  и  $L_1$  языку  $L$ , который состоит из цепочек, представляющих собой парное сцепление цепочек языков  $L_0$  и  $L_1$ .

Пример:

$$L_0=(a, (a+a), \dots)$$

$$L_1=(b, bb, \dots)$$

$$L=L_0L_1=\{ab, abb, \dots, (a+a)b, (a+a)bb, \dots\}$$

Введем обозначение кратной конкатенации

$$L \& L = L^2;$$

$$L \& L \& L = L^3; \dots$$

### **3. Итерация.**

Язык  $L^*$ , его подмножество будет определяться равенством

$$L^* = [\varepsilon] \cup L \cup L^2 \cup \dots \cup L^n = \{\varepsilon\} \cup \bigcup_{i=1}^n L^i$$

Замечание: не следует смешивать язык, содержащий  $\varepsilon$  (пустую цепочку), с пустым языком, не содержащим ни одной цепочки.  $\varepsilon$  – не есть отсутствие правил. Язык, содержащий  $\varepsilon$  – не пустой.

## **2. Двоичное кодирование переменных и функций трехзначной логики**

Для описания дискретных устройств, наряду с булевой логикой применяются такие, у которых аргументы и сами функции принимают значения из множества, содержащего  $k$ -элементов.  $k: (0, 1, \dots, k-1)$

**Определение:** функция, принимающая значения от 0 до  $k-1$ , аргументы которой также принимают значения из этого множества, называется функцией  $k$ -значной логики. Булева функция - функция двухзначной логики.

В  $k$ -значной логике выделяется ряд элементарных функций, например:

$$1) \text{ квазиконъюнкция} \quad \dot{\&} = \min(x_1, x_2)$$

$$2) \text{ квазидизъюнкция} \quad \dot{\vee} = \max(x_1, x_2)$$

$$3) \text{ сумма по модулю } k \quad \{x_1 \oplus x_2\}_{\text{mod } k}$$

$$4) \text{ произведение по модулю } k \quad \{x_1 \otimes x_2\}_{\text{mod } k}$$

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

значение функции равно остатку от деления произведения  $x_1 x_2$  на  $k$

5) функция Вебба

$$\{\max(x_1, x_2)+1\}_{\text{mod } k}$$

6) цикл (циклическое отрицание)

$$\tilde{x} = \{x + 1\}_{\text{mod } k}$$

7) функция инверсии

$$\bar{x} = k - 1 - x$$

8) характеристические

функции:

$$\varphi_{\Omega}(x) = \begin{cases} k-1, \Omega = x \\ 0, \Omega \neq x \end{cases} \quad \Omega = (0 \dots k-1) \quad \psi_{\sigma} = \begin{cases} 1, x = \sigma \\ 0, x \neq \sigma \end{cases}$$

Построим таблицы, задающие все введенные элементарные функции 3-х значной логике.

В 3-х значной логике функциями const являются 0,1,2.

$x_1$	$x_2$	&	∨	$x_1 \downarrow x_2$	$\oplus$	$\otimes$
0	0	0	0	1	0	0
0	1	0	1	2	1	0
0	2	0	2	0	2	0
1	0	0	1	2	1	0
1	1	1	1	2	2	1
1	2	1	2	0	0	2
2	0	0	2	0	2	0
2	1	1	2	0	0	2
2	2	2	2	0	1	1

Закодируем аргументы следующим образом: 0 = 00, 1 = 01, 2 = 10.

Для записи и передачи любого троичного переменного необходимо использовать две двоичные переменные  $v_1, v_2$ . При этом функции  $\Psi_i(x)$  будут кодироваться следующим образом:

X	$v_1$	$v_2$	$\Psi_0'$	$\Psi_0''$	$\Psi_1'$	$\Psi_1''$	$\Psi_2'$	$\Psi_2''$
0	0	0	0	1	0	0	0	0
1	0	1	0	0	0	1	0	0
2	1	0	0	0	0	0	0	1
*	1	1	*	*	*	*	*	*

удобно доопределить  $\Psi_i$  на наборе  $\langle 1, 1 \rangle$  нулями, тогда получим:  $\Psi_0' = \Psi_1' = \Psi_2' = 0; \Psi_0'' = \neg v_1 \& \neg v_2; \Psi_1'' = \neg v_1 \& v_2; \Psi_2'' = v_1 \& \neg v_2$ ;

Один из способов моделирования трехзначной логики заключается в создании функциональных элементов с тремя устойчивыми состояниями, то есть с квантованием сигнала по трем уровням, при этом принята аналогия: положительный потенциал - 0, 0-й потенциал - 1, отрицательный потенциал - 2.

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

Практически в полупроводниковых схемах для трехзначной функции положительным потенциалом считается потенциал  $\geq 1.5$  В.

Нулевым потенциалом считается потенциал по модулю  $\leq 0.6$ .  
Отрицательным - потенциал  $\leq -1.5$  В.

Пример: рассмотрим кодирование ф-й  $X_1 \oplus X_2$ ,  $X_1 \otimes X_2$

X1		X2		$X_1 \oplus X_2$		$X_1 \otimes X_2$	
V1	V2	V3	V4	f1	f2	f3	f4
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	*	*	*	*
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	*	*	*	*
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	1
1	0	1	1	*	*	*	*
1	1	0	0	*	*	*	*
1	1	0	1	*	*	*	*
1	1	1	0	*	*	*	*
1	1	1	1	*	*	*	*

Таким образом, функцию  $f(x_1, x_2)$  можно представить следующим образом:

$$f(x_1, x_2) = \langle f_1(v_1, v_2, v_3, v_4), f_2(v_1, v_2, v_3, v_4) \rangle$$

$$f_1 = \neg v_1 \wedge \neg v_2 \wedge v_3 \wedge \neg v_4 \vee \neg v_1 \wedge v_2 \wedge \neg v_3 \wedge v_4 \vee v_1 \wedge \neg v_2 \wedge \neg v_3 \wedge \neg v_4$$

$$f_2 = \neg v_1 \wedge \neg v_2 \wedge \neg v_3 \wedge v_4 \vee \neg v_1 \wedge v_2 \wedge \neg v_3 \wedge \neg v_4 \vee v_1 \wedge \neg v_2 \wedge v_3 \wedge \neg v_4$$

Как следует из кодировки функции, логическая схема её реализующая должна иметь два выхода и четыре входа. Необходимо выполнить минимизацию сформированных функций  $f_1, f_2$ . Составим карту Карно для функции  $f_1$ :

	$\neg v_1$	$v_1$		
$v_2$				$\neg v_4$
		*	*	
	1	*	*	
		*	*	
$\neg v_2$				$v_4$
		1	1	
	$\neg v_3$	$v_3$	$\neg v_3$	$\neg v_4$

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

Для того чтобы минимизировать слабо определенную функцию в карте Карно проставляют специальный знак в местах характерных наборам, на которых функция не определена, затем \* меняют на 1 в тех клетках, составленные прямоугольники из которых уменьшили бы число конъюнкций, дизъюнкций и отрицаний.

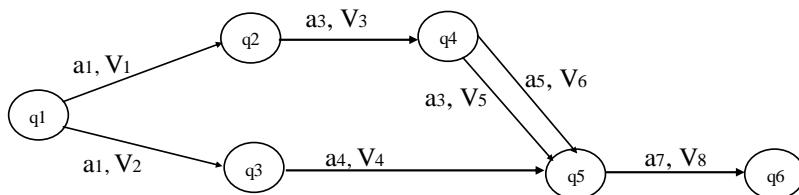
Для примера следует:  $f_1 = v_2 \& v_4 \vee v_3 \& \neg v_1 \& \neg v_2 \vee \neg v_3 \& \neg v_4 \& v_1$

Аналогично составляются функции  $f_2, f_3, f_4$ .

### 3. Перечислить способы представления конечного автомата

Конечный автомат – система объектов  $M = (Q, T, V, \delta, \lambda)$ , в которой  $Q = \{q_1, \dots, q_n\}$ ,  $T = \{a_1, \dots, a_m\}$ ,  $V = \{V_1, \dots, V_e\}$ ,  $Q, T, V$  – конечные множества (алфавиты),  $Q$  - алфавит состояний,  $T$  - входной алфавит,  $V$  - выходной алфавит,  $\delta$  - функция переходов (определяется как отображение множества  $Q \times T$  в множество  $Q$ , т.е.  $\delta: Q \times T \rightarrow Q$ ),  $\lambda$  - функция выходов  $\lambda: Q \times T \rightarrow V$ , т.е. отображается на множестве  $V$ .

1. **Ориентированный граф** (граф состояний), в котором состояния - вершины графа, дуги – переходы между состояниями. Вершины помечаются номерами состояний автомата. Дуги, соединяющие вершины, помечаются входным символом, который вызвал переход автомата из i-го состояния в j-тое, а также выходным символом, который устанавливается на выходе автомата в результате этого перехода.



$a_i$  - символы входного алфавита, вызывающие переход;

$V_i$  - символы выходного алфавита;

$q_i$  – состояния автомата.

Детерминированным называется автомат, граф перехода которого не содержит вершин, имеющих одинаково помеченные дуги.

### 2. Таблица переходов

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

Строки – состояния автомата. Столбцы - символы входного алфавита

Клетки таблицы заполняют состояния, в которые переходит автомат под действием входных символов, а также символы выходного алфавита, соответствующие реакции автомата на входной символ.

**Пример:**

	a1	a2	a3	a4	a5	a6
q1	q2, V1	q3, V2				
q2			q4, V3			
q3				q5, V4		
q4			q5, V5		q5, V6	
q5						q6, V7
q6						

### **3. Матрицей переходов**

Матрица переходов представляет собой квадратную матрицу, строки и столбцы которой соответствуют внутренним состояниям автомата. Клетки матрицы заполняются входными символами  $a_k$ , при которых автомат переходит из состояния  $q_i$  в состояние  $q_j$ , а также выходными символами, соответствующими паре  $(a_k, q_i)$ .

**Пример:**

	q1	q2	q3	q4	q5	q6
q1		a1, V1	a2, V2			
q2				a3, V3		
q3					a4, V4	
q4					a3, V5 a5, V6	
q5						a6, V7
q6						

**Детерминированным конечным автоматом** называется такой автомат, каждая клетка таблицы переходов которого не содержит состояний больше одного. В противном случае автомат называется **недетерминированным**.

#### 4. Определение недетерминированного и конечного автомата

Теория автоматов лежит в основе теории построения компиляторов. Конечный автомат – одно из основных понятий. Под автоматом подразумевается не реально существующее техническое устройство, хотя такое устройство может быть построено, а некоторая математическая модель, свойства и поведение которой можно имитировать с помощью программы на вычислительной машине. Конечный автомат является простейшей из моделей теории автоматов и служит управляющим устройством для всех остальных видов автоматов. Конечный автомат обладает рядом свойств:

- конечный автомат может решать ряд легких задач компиляции. Лексический блок почти всегда строится на основе конечного автомата.
- работа конечного автомата отличается высоким быстродействием.
- моделирование конечного автомата требует фиксированного объема памяти, что упрощает проблемы, связанные с управлением памятью.
- Существует ряд теорем и алгоритмов, позволяющих конструировать и упрощать конечные автоматы.

**Детерминированным конечным автоматом** называется такой автомат, каждая клетка таблицы переходов которого не содержит состояний больше одного. В противном случае автомат называется **недетерминированным**.

**Конечный автомат** называется полностью определенным, если его таблица переходов не содержит пустых клеток. Иначе автомат называют частично определенным.

Конечный автомат – это формальная система, которая создаётся с помощью следующих объектов:

- конечным множеством входных символов;
- конечным множеством состояний;
- функцией переходов, которая каждой паре (входной символ, текущее состояние) ставит в соответствие новое состояние;
- начальное состояние;
- подмножество состояний, выделенных в качестве допускающих или заключительных;

Итак, детерминированным конечным автоматом (ДКА) называется устройство, описываемое следующими параметрами:  
 $Q$  – конечное множество состояний.

$\Sigma$  – конечное множество входных символов.

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

δ – функция перехода. Аргументы – состояние и входной символ, результат – состояние.

$q_0$  – начальное состояние, принадлежит  $Q$ .

$F$  – множество допускающих состояний, является подмножеством  $Q$ .

И функционирующие следующим образом:

Автомат начинает работу в состоянии  $q_0$ .

Если автомат находится в состоянии  $q_i$ , а на вход поступает символ  $b$ , то автомат переходит в состояние  $\delta(q_i, b)$ .

Определение недетерминированного конечного автомата (НКА) практически полностью повторяет приведённое выше определение ДКА. Отличий всего два:

δ – функция перехода. Аргументы – состояние и входной символ, результат – множество состояний (возможно – пустое).

Если автомат находится в состоянии  $q_i$ , а на вход поступает символ  $b$ , то автомат переходит во множество состояний  $\delta(q_i, b)$ .

Если автомат находится во множестве состояний  $\{q_i\}$ , то он переходит во множество состояний, получаемое объединением множеств  $\delta(q_i, b)$ .

НКА тоже распознаёт цепочки символов, цепочка считается допустимой, если после её обработки множество состояний, в котором оказался автомат, содержит хотя бы одно допускающее. Таким образом, НКА также задаёт некоторый язык.

Важным аспектом является преобразование недетерминированного конечного автомата к детерминированному. Недетерминированные конечноавтоматные распознаватели могут быть двух типов: либо существует переход, помеченный пустой цепочкой  $\epsilon$ , либо из одного состояния выходят несколько переходов, помеченных одним и тем же символом (возможны оба случая).

Алгоритм построения эквивалентного детерминированного конечного автомата.

Приведение недетерминированного автомата к автомatu без  $\epsilon$ -переходов.

Определение:  $\epsilon$ -замыканием состояния  $s$  называется множество всех состояний, которые достижимы из  $s$  без подачи входного сигнала. Множеством состояний полученного автомата являются  $\epsilon$ -замыкания состояний автомата с  $\epsilon$ -переходами.

Построение по полученному автомату без  $\epsilon$ -переходов эквивалентного ему детерминированного автомата, допускающего тот же язык. В качестве начального (конечного)

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

состояния искового автомата выбрать множество начальных(конечных) состояний исходного автомата.

Примеры:

ДКА

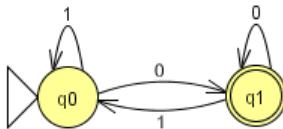


таблица переходов

		0	1
->	q0	q1	q0
*	q1	q1	q0

НедКА

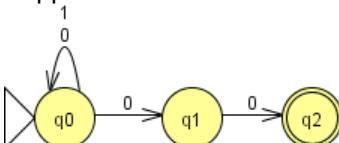


таблица переходов

	0	1	
->	q0	{ q0,q1 }	{ q0 }
	q1	{ q2 }	Ø
*	q2	Ø	Ø

## 5. Программная реализация логических функций

Представление автомата схемой, состоящей из логических элементов наиболее исследованный вид структурной реализации автомата. Другой её вид – реализация программ. Программа реализует логические функции  $f(x_1, x_2, \dots, x_n) = y$ , если для любого двоичного набора  $\delta = (\delta_1, \dots, \delta_n)$  при начальном состоянии элементов памяти  $x_1 = \delta_1, x_2 = \delta_2, \dots, x_n = \delta_n$  программа за конечное число шагов останавливается и в ячейке  $y$  лежит величина  $y = f(\delta_1, \delta_2, \dots, \delta_n)$ .

Если под сложностью схемы, реализующей автомат, обычно понимается число элементов схемы, то под сложностью программ можно понимать:

- число команд в тексте программы;
- объем промежуточной памяти;

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

- время вычисления программы, которое характеризуется двумя величинами:

$$t_{cp}^n = \frac{1}{2^n} * \sum_{\sigma} \tau_p(\sigma)$$

1. средним временем:

2. максимальным временем:  $t_{\max} = \max(\tau_p(\sigma))$

где сумма и максимум берется по всем  $2^n$  наборам, а  $\tau_p$  - время работы программы на одном наборе  $\sigma$ .

Рассмотрим 2 типа программ, которые реализуют логические функции:

- операторные
- бинарные.

*Операторные программы* не содержит условных переходов, порядок её команд в точности соответствует нумерации элементов в схеме, а система команд соответствует базису схемы. Элементы схемы нумеруются числами 1... n таким образом, чтобы на любом пути от входа к выходу номера элементов возрастили. При этом номер 1 получит один из входных элементов, а номер n - выходной элемент.

Пусть элемент схемы  $e_i$  реализует функцию  $\phi_i$  и к его входам присоединены выходы элементов  $e_j1, e_j2, \dots, e_jm$  (некоторые из них, возможно, являются входами схемы), тогда выход такого элемента можно записать:  $a_i = \phi_i(e_j1, e_j2, \dots, e_jm)$  при  $i \neq n$ , а выход схемы может быть записан:

$y = \phi_i(e_j1, e_j2, \dots, e_jm)$  при  $i = n$ . Такая программа будет реализовывать работу заданной схемы. Проблема синтеза операторных программ сводится к проблеме синтеза схем, то есть к вопросам функциональной полноты и минимизации схем. Поскольку операторная программа не содержит условных переходов, то время её выполнения на любом наборе одно и тоже, отсюда  $t_{\max} = t_{cp}$ .

*Бинарные программы* это программы, состоящие из команд типа:  $y:=b; b = \{0, 1\}$  и условные переходов.

Бинарные программы обладают двумя достоинствами по сравнению с операторными:

- отсутствие промежуточной памяти в процессе работы программы. Это позволяет реализовывать бинарную программу на постоянных элементах памяти.
- Более высоким быстродействием.

## ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

### **Пример:**

составить для функции  $f = (x_1 \vee \neg x_3) \& (x_5 \& \neg x_4 \vee x_2)$  бинарную и операторную программы.

### **Решение:**

```

Program binary;
{описание переменных f,x1,x2,x3,x4,x5}
Begin
  {ввод переменных x1,x2,x3,x4,x5}
Case x1 of
  0: Case x3 of
    0: f:=1;
    1: f:=0;
    End;
  1: f:=1;
End;
Case f of
  1: Case x5 of
    0: Case x2 of
      0: f:=0;
      1: f:=1;
      End;
    1: Case x4 of
      0: f:=0;
      1: f:=1;
      End;
    End;
  End;
{вывод f}
End.

```

Операторная программа пишется в базисе  $\{\& \neg\}$ . Для этого перепишем заданную функцию, используя формулы де Моргана.

$$f = \neg(\neg x_1 \& x_3) \& \neg(\neg(x_5 \& \neg x_4) \& \neg x_2)$$

```

Program operat;
{описание переменных a,b,c,f,x1,x2,x3,x4,x5}
Begin
  {вывод переменных x1,x2,x3,x4,x5}
  a:=1-x1; {\neg x1}
  b:=a*x3; {\neg x1 \& x3}
  b:=1-b; {\neg(\neg x1 \& x3)}
  a:=1-x4; {\neg x4}
  c:=a*x5; {x5 \& \neg x4}
  c:=1-c; {\neg(x5 \& \neg x4)}
  a:=1-x2; {\neg x2}
  c:=c*a; {\neg(x5 \& \neg x4)\& \neg x2}

```

```
c:=1-c; {¬(¬(x5 & ¬x4)& ¬x2)}  
f:=b*c; {¬(¬x1 & x3)) & (¬(¬(x5 & ¬x4)& ¬x2))}  
{вывод f}  
End.
```

## XI. АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

1. **Виды систем обработки данных. Режимы обработки данных.** Сформулируйте различия между многомашинными вычислительными комплексами и вычислительными сетями.

### **Виды систем обработки данных:**

Вычислительный комплекс (ВК) – это совокупность технических средств, включающих в себя несколько ЭВМ или процессоров и общесистемного программного обеспечения.

Основной задачей комплекса является повышение точности и надежности вычислений.

Многомашинный вычислительный комплекс (ММ) – несколько ЭВМ связаны между собой косвенно или прямой связью.

Многопроцессорный ВК (МП) – включает несколько процессоров с общей ОП и периферийными устройствами. Комплекс работает под управлением единой ОС, которая выполняет функции обеспечения работоспособности комплекса при выходе из строя оборудования.

Вычислительная система (ВС) – это система обработки данных, настроенная на решение задач конкретной области применения. Она включает в себя технические средства и специальное программное обеспечение.

Существует 2 способа ориентации ВС на решение задач:

1. С помощью ПО и периферийных устройств
2. За счет использования специализированных ЭВМ и вычислительных средств.

Система телеобработки (СТоб) – предназначена для обработки данных, передаваемых по каналам связи. Данные передаются в виде сообщений. Сообщение кроме непосредственной информации несет в себе служебную информацию, необходимую для управления процессами передачи данных и защиты их искажения.

Вычислительные сети (ВС) – это система взаимосвязанных и распределенных по фиксированной территории вычислительных

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

центров и ЭВМ, ориентированная на использование общих ресурсов.

Ядром является базовая сеть передачи данных (СПД) которая состоит из каналов и узлов сети.

Локальные вычислительные сети (ЛВС) – это совокупность близкорасположенных ЭВМ, которые связаны последовательными каналами оснащены программными средствами, обеспечивающие информационное взаимодействие между процессами в разных ЭВМ.

### **Режимы обработки данных.**

Режимы:

1. Однопрограммная обработка
2. Мультипрограммная обработка
3. Оперативная обработка
4. Пакетная обработка
5. Обработка в реальном масштабе времени
6. Режим телебработки

Мультипрограммная обработка – в системе обрабатываются сразу несколько задач на устройствах, которые способны функционировать параллельно.

Режим оперативной обработки:

Характеристики:

1. Малый объем входных/выходных данных и вычислений, приходящихся на взаимодействие с системой.
2. Высокая интенсивность взаимодействия с системой. (Режим запрос-ответ, режим диалоговый).

Пакетная обработка – характеризуется большим объемом входных/выходных данных и вычислений, приходящихся на одно взаимодействие с системой.

Обработка в реальном масштабе времени – здесь темп инициирования задач и время получения ответа определяется динамическими характеристиками управляемого объекта.

Режим телебработки – взаимодействие пользователей осуществляется через линии связи. Требует специальных методов доступа.

**Сформулируйте различия между многомашинными вычислительными комплексами и вычислительными сетями.**

Первое отличие - размерность. В состав многомашинного вычислительного комплекса входят обычно две, максимум три ЭВМ, расположенные преимущественно в одном помещении.

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Вычислительная сеть может состоять из десятков и даже сотен ЭВМ, расположенных на расстоянии друг от друга от нескольких метров до десятков, сотен и даже тысячи километров.

Второе отличие - разделение функций между ЭВМ. Если в многомашинном вычислительном комплексе функции обработки данных, передачи данных и управления системой могут быть реализованы в одной ЭВМ, то в вычислительных сетях эти функции распределены между различными ЭВМ.

Третье отличие - необходимость решения в сети задачи маршрутизации сообщений. Сообщение от одной ЭВМ к другой в сети может быть передано по различным маршрутам в зависимости от состояния каналов связи, соединяющих ЭВМ друг с другом.

**2. Уровни комплексирования устройств в вычислительных системах. Постройте структурную схему ПЭВМ, состоящей из двух процессоров. Покажите на ней используемые уровни комплексирования. Ответ поясните.**

**Уровни комплексирования устройств в вычислительных системах.**

Для построения вычислительных систем необходимо, чтобы модули были совместимы. Выделяют 3 уровня совместимости: аппаратный, программный, информационный.

Уровни совместимости:

1. аппаратный;

- подключаемая друг к другу аппаратура должна иметь единые стандартные средства соединения (кабели, число проводов в них, единое назначение проводов, разъемы, заглушки, адAPTERы, платы, перемычки);
- параметры электрических сигналов, которыми обмениваются технические средства, должны быть согласованы (амплитуда, длительность, способы модуляции);
- алгоритмы взаимодействия не должны вступать в противоречия друг с другом;

2. программный;

- программы, передаваемые от одного технического средства к другому, должны быть правильно поняты и выполнены.

3. информационный;

- передаваемые информационные массивы должны одинаково интерпретироваться в устройствах (алфавиты,

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

форматы, структуры и разметки файлов должны быть стандартизованы).

Уровни комплексирования (Все 5 используются в суперЭВМ):

1. прямого управления (интерфейс используется для передачи коротких сообщений и процессы прерываются внешними прерывателями, данные не пересекаются);
  2. общей оперативной памяти (для оперативного взаимодействия процессоров. Чем меньше устройств – тем больше взаимодействие);
  3. каналов ввода-вывода (для передачи больших объемов информации между блоками ОЗУ, обмен происходит через адаптер канал-канал; обычно сопрягаются селекторные каналы);
  4. устройства управления внешними устройствами (в устройстве внешнего управления используются двухканальные переключатели, которые позволяют подключать устройство внешнего управления одной машины к селекторным каналам других машин);
- общие внешние устройства (для подключения отдельных устройств, которые являются уникальными и дорогими).

### **Дополнительно**

В создаваемых ВС стараются обеспечить несколько путей передачи данных, что позволяет достичь необходимой надежности функционирования, гибкости и адаптируемости к конкретным условиям работы. Эффективность обмена информацией определяется скоростью передачи и возможными объемами данных, передаваемыми по каналу взаимодействия. Эти характеристики зависят от средств, обеспечивающих взаимодействие модулей и уровня управления процессами, на котором это взаимодействие осуществляется. Сочетание различных уровней и методов обмена данными между модулями ВС наиболее полно представлено в универсальных суперЭВМ и больших ЭВМ, в которых сбалансированное использование использовались все методы достижения высокой производительности. В этих машинах предусматривались следующие уровни комплексирования:

- 1)прямого управления (процессор - процессор);
- 2) общей оперативной памяти;
- 3) комплексируемых каналов ввода-вывода;
- 4) устройств управления внешними устройствами (УВУ);
- 5) общих внешних устройств.

На каждом из этих уровней используются специальные технические и программные средства, обеспечивающие обмен информацией.

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

*Уровень прямого управления* служит для передачи коротких однобайтовых приказов-сообщений. Последовательность взаимодействия процессоров сводится к следующему. Процессор-инициатор обмена по интерфейсу прямого управления (ИЛУ) передает в блок прямого управления байт-сообщение и подает команду «прямая запись». У другого процессора эта команда вызывает прерывание, относящееся к классу внешних. В ответ он вырабатывает команду «прямое чтение» и записывает передаваемый байт в свою память. Затем принятая информация расшифровывается и по ней принимается решение. После завершения передачи прерывания снимаются, и оба процессора продолжают вычисления по собственным программам. Видно, что уровень прямого управления не может использоваться для передачи больших массивов данных, однако оперативное взаимодействие отдельными сигналами широко используется в управлении вычислениями. У ПЭВМ типа IBM PC этому уровню соответствует комплексирование процессоров, подключаемых к системной шине.

*Уровень общей оперативной памяти* (ООП) является наиболее предпочтительным для оперативного взаимодействия процессоров. В этом случае ООП эффективно работает при небольшом числе обслуживаемых абонентов.

*Уровень комплексируемых каналов ввода-вывода* предназначается для передачи больших объемов информации между блоками оперативной памяти, сопрягаемых в ВС. Обмен данными между ЭВМ осуществляется с помощью адаптера «канал-канал» (АКК) и команд «чтение» и «запись». Адаптер - это устройство, согласующее скорости работы сопрягаемых каналов. Обычно сопрягаются селекторные каналы (СК) машин как наиболее быстродействующие. Скорость обмена данными определяется скоростью самого медленного канала. Скорость передачи данных по этому уровню составляет несколько Мбайт в секунду. В ПЭВМ данному уровню взаимодействия соответствует подключение периферийной аппаратуры через контроллеры и адAPTERы.

*Уровень устройств управления внешними устройствами* (УВУ) предполагает использование встроенного в УВУ двухканального переключателя и команд «зарезервировать» и «освободить». Двухканальный переключатель позволяет подключать УВУ одной машины к селекторным каналам различных ЭВМ. По команде «зарезервировать» канал - инициатор обмена имеет доступ через УВУ к любым накопителям на дисках НМД или на магнитных лентах НМЛ. На рис. 10.4 схематически показано, что они управляются одним УВУ. На самом деле УВУ магнитных дисков и

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

лент -совершенно различные устройства. Обмен канала с накопителями продолжается до полного завершения работ и получения команды «освободить». Только после этого УВУ может подключиться к конкурирующему каналу. Только такая дисциплина обслуживания требований позволяет избежать конфликтных ситуаций.

На четвертом уровне с помощью аппаратуры передачи данных (АПД) (мультплексоры, сетевые адаптеры, модемы и др.) имеется возможность сопряжения с каналами связи. Эта аппаратура позволяет создавать сети ЭВМ.

Пятый уровень предполагает использование общих внешних устройств. Для подключения отдельных устройств используется автономный двухканальный переключатель.

Пять уровней комплексирования получили название логических потому, что они объединяют на каждом уровне разнотипную аппаратуру, имеющую сходные методы управления. Каждое из устройств может иметь логическое имя, используемое в прикладных программах. Этим достигается независимость программ пользователей от конкретной физической конфигурации системы. Связь логической структуры программы и конкретной физической структуры ВС обеспечивается операционной системой по указаниям -директивам пользователя, при генерации ОС и по указаниям диспетчера-оператора вычислительного центра. Различные уровни комплексирования позволяют создавать самые различные структуры ВС.

Второй логический уровень позволяет создавать многопроцессорные ВС. Обычно он дополняется и первым уровнем, что позволяет повышать оперативность взаимодействия процессоров. Вычислительные системы сверхвысокой производительности должны строиться как многопроцессорные. Центральным блоком такой системы является быстродействующий коммутатор, обеспечивающий необходимые подключения абонентов (процессоров и каналов) к общей оперативной памяти.

Уровни 1, 3, 4, 5 обеспечивают построение разнообразных машинных комплексов. Особенно часто используется третий в комбинации с четвертым. Целесообразно их дополнять и первым уровнем.

Пятый уровень комплексирования используется в редких специальных случаях, когда в качестве внешнего объекта используется какое-то дорогое уникальное устройство. В противном случае этот уровень малоэффективен. Любое внешнее устройство - это недостаточно надежное устройство точной механики, а значит, выгоднее использовать четвертый

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

уровень комплексирования, когда можно сразу управлять не одним, а несколькими внешними устройствами, включая и резервные.

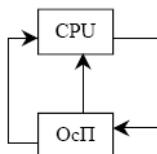
Сочетание уровней и методов взаимодействия позволяет создавать самые различные многомашинные и многопроцессорные системы.

**3. Методы улучшения ОКОД структуры. Степень, уровни и виды параллелизма. Какой из видов параллелизма реализуется в современных универсальных процессорах (например, в процессоре Pentium)? Ответ обоснуйте.**

### **Методы улучшения ОКОД структуры.**

1. ОКОД (SISD, одиночный поток команд одиночный поток данных). Такое структурное

построение характерно для классических машин фон Неймана.



Функционирование в виде линейного процессора.  
ОУ, OcП (основная память), УУ

Линейная организация вычислительного процесса обуславливает весьма низкую эффективность аппаратных средств (велик коэффициент простого). Для повышения работы такой структуры применяются методы локального параллелизма – совмещенная или опережающая выборка команд, расслоение памяти, но, как правило, это требует дополнительных аппаратных затрат.

### **Степень, уровни и виды параллелизма.**

Степень параллелизма – порядок числа параллельно работающих устройств при условии, что количество обрабатывающих устройств неограниченно.

Низкая степень – от 2 до 10 процессоров;

Средняя степень – 10 – 100 процессоров;

Высокая степень – 100 – 10000 процессоров;

Сверхвысокая степень –  $10^4$  –  $10^6$  процессоров (нейросистемы).

От степени параллелизма зависят:

Архитектура вычислительной машины, особенно система коммутации;

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Организация взаимодействия параллельно работающих процессоров;

Методы обмена данными между процессорами и памятью.

Уровень параллелизма – схемный аппаратный уровень, на котором осуществляется распараллеливание обработки данных и организация параллельных вычислений.

Уровни параллелизма:

На уровне логических вентилей и элементов памяти;

Уровень логических схем и простых автоматов с памятью;

Уровень регистров и интегральных схем памяти;

Уровень элементарных микропроцессоров;

Уровень микропроцессоров, реализующих крупные операции;

Уровень вычислительных машин, процессоров и программ;

Параллельные вычислительные системы строят по принципу модульного наращивания и расширения.

Виды параллелизма.

Среди способов параллельной обработки данных выделяют следующие направления:

1) Совмещение во времени различных этапов разных задач (мультипрограммная обработка);

2) Одновременное решение различных задач или частей одной задачи (конвейерная обработка).

Виды параллелизма:

**Естественный параллелизм.**

Задача обладает естественным параллелизмом, если в её исходной постановке она сводится к операциям над многомерными векторами, матрицами или решетчатыми функциями.

**Параллелизм множества объектов – частн. случай естественного парал-ма.**

Его смысл в том, что задача состоит в обработке информации о различных, но однотипных объектах, обрабатываемых по одной и той же программе.

Этот вид параллелизма характеризуется параметрами:

Суммарная длина программы;

$S_n=1..m_k$ , где  $m_k$  – количество вариантов программы на  $k$ -том

$L_{\Sigma} = \sum_{(k)} \sum_{S_k=1}^{m_k} l_{k_{Sk}}$  шаге.  
 $k_{Sk}$  – оператор, выполняемый по  $S_k$

ветви.

$|l_{k_{Sk}}|$  – длина операторов  $k_{Sk}$ .

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Средняя длина программы;

$$L_{CP} = \frac{1}{r} * \sum_{(k)} \sum_{S_k=1}^{mk} l_{k_{Sk}} * r_{k_{Sk}}$$

число классов объектов;  $r$  – ранг задачи.

$$\text{Величина расхождения задачи-} D = \frac{L_{\Sigma}}{L_{\bar{n}\delta}}$$

$r k_{Sk}$  – количество объектов, которые к-ом шаге относ-ся к  $S_k$  классу.  $m$  –

### **Параллелизм независимых ветвей:**

Суть состоит в том, что в программе могут быть выделены независимые части, которые называются ветвями. Ветвь X не зависит от ветви Y, если выполняются следующие условия:

ветви не зависят от данных, т.е. ни одна из входных переменных на ветви X не является выходной переменной на ветви Y;  
ветви выполняются по разным программам;  
ветви независимы по управлению.

Отличие параллелизма независимых ветвей от естественного параллелизма состоит в том, что выходные результаты ветвей используются для выполнения следующих операций.

Отличие параллелизма независимых ветвей от множества объектов состоит в том, что в параллелизме множества объектов для всех объектов используются копии одной и той же программы.

### **Параллелизм смежных операций или локальный параллелизм.**

Параллелизм смежных операций имеет место тогда, когда входные данные текущих операций получены на более ранних этапах и выполнение этих операций можно совместить во времени.

Характеристики:

Показатель связности смежных операций ( $\alpha$ ) – вероятность того, что результат некоторой операции будет использован в следующей за ней операции. Чем меньше  $\alpha$ , тем больше глубина параллелизма смежных операций;

Вероятность того, что начиная от данной операции имеется цепочка длиной не менее  $l$  операций, которые можно выполнять одновременно.

Гамма – вероятность того, что начиная от данной операции

$$\gamma_l = (1 - \alpha)^{\frac{(l-1)*l}{2}} + (1 - \alpha)^{\frac{(l+1)*l}{2}}$$

имеется цепочка из ровно  $l$  операций, которые можно

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

выполнять одновременно;

Глубина параллелизма смежных операций,

$$L_{nCO} = \sum_{l=1}^{\infty} (1 - \alpha)^{\frac{(l-1)*l}{2}}$$

т.е. математическое ожидание длины цепочки операций, которые можно выполнять одновременно.

Какой из видов параллелизма реализуется в современных универсальных процессорах (например, в процессоре Pentium)? Ответ обоснуйте.

Разделяя вычислительную работу, выполняемую в традиционных микропроцессорах одним ядром Pentium, между несколькими исполнительными ядрами Pentium, многоядерный процессор может выполнять больше работы за конкретный интервал времени и улучшать таким образом впечатления пользователей от работы с системой. Чтобы это улучшение стало возможным, ПО должно поддерживать распределение нагрузки между несколькими исполнительными ядрами. Эта функциональность называется параллелизмом на уровне потоков или организацией поточной обработки, а поддерживающие ее приложения и операционные системы (такие, как Microsoft Windows\* XP) называются многопоточными.

Процессор, поддерживающий параллелизм на уровне потоков, может выполнять полностью обособленные потоки кода, например, поток приложения и поток операционной системы или два потока одного приложения (особенно большую выгоду извлекают из параллелизма на уровне потоков мультимедийные приложения, потому что многие их операции могут выполняться параллельно).

Можно ожидать, что по мере увеличения числа многопоточных приложений, использующих достоинства этой архитектуры, многоядерные процессоры будут обеспечивать все новые и новые преимущества пользователям ПК как дома, так и на работе. Многоядерные процессоры могут также улучшить впечатления пользователей от работы в многозадачных средах, а именно при выполнении нескольких приложений переднего плана одновременно с несколькими фоновыми приложениями, такими, как антивирусное и защитное ПО, утилиты для беспроводной связи, управляющие программы и приложения, служащие для сжатия файлов, шифрования и синхронизации.

Как и другие аппаратные способы реализации многопоточности, разработанные и совершенствуемые в Intel, многоядерная архитектура отражает переход к параллельной обработке -

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

концепции, зародившейся в мире суперкомпьютеров. Например, технология Hyper-Threading (HT), представленная корпорацией Intel в 2002 году, обеспечивает возможность параллельного выполнения задач, объединяя несколько "потоков" в одноядерном процессоре. Но технология HT ограничена одним ядром, более эффективно использующим имеющиеся ресурсы для обеспечения лучшей поддержки многопоточности, тогда как многоядерная архитектура включает два (или более) полных набора исполнительных ресурсов.

Специалисты корпорации Intel считают, что многоядерная архитектура способна обеспечить несколько важных возможностей, улучшающих впечатления пользователей, в том числе увеличение числа выполняемых одновременно задач, выполнение требовательных к вычислительной мощности приложений и увеличение числа пользователей, работающих с одним ПК.

### **4. Подсистема памяти. Методы повышения быстродействия памяти. Виды ЗУ. Иерархическая организация памяти. Какие вычислительные системы, на каком уровне иерархической организации требуют организации пакетного доступа к памяти. Ответ поясните.**

Память любой ЭВМ состоит из нескольких видов памяти (оперативная, постоянная и внешняя - различные накопители). Память является одним из важнейших ресурсов. Поэтому операционная система управляет процессами выделения объемов памяти для размещения информации пользователей. В любых ЭВМ память строится по иерархическому принципу. Это обуславливается следующим:

Оперативная память предназначена для хранения переменной информации, так как она допускает изменение своего содержимого в ходе выполнения микропроцессором соответствующих операций. Поскольку в любой момент времени доступ может осуществляться к произвольно выбранной ячейке, то этот вид памяти называют также памятью с произвольной выборкой - RAM (Random Access Memory).

Все программы, в том числе и игровые, выполняются именно в оперативной памяти. Постоянная память обычно содержит такую информацию, которая не должна меняться в течение длительного времени. Постоянная память имеет собственное название - ROM (Read Only Memory), которое указывает на то, что ею обеспечиваются только режимы считывания и хранения.

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

С точки зрения пользователей желательно было бы иметь в ЭВМ единую сверх большую память большой производительности, однако емкость памяти и время обращения связаны между собой (чем больше объем тем больше время обращения к ней). Для упрощения все пересылки информации осуществляется не по вертикали, а через оперативную память. Кое-какие процедуры планирования теперь осуществляются компиляторами языков высокого уровня.

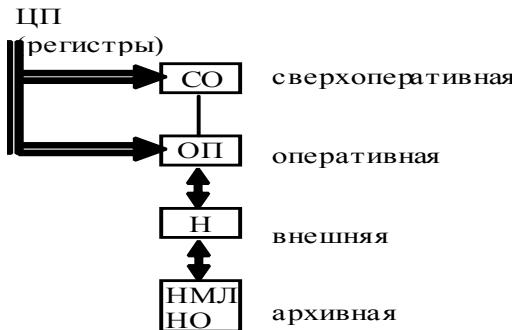
Существует противоречие, связанное с организацией ЗУ. Соблюдается тенденция увеличения объема памяти при медленном росте её быстродействия. Требуется, чтобы скоростные характеристики процессора и памяти были одинаковы. Поэтому структура памяти имеет иерархическую организацию. Она бывает оперативной и внешней. Оперативная память делится на сверхоперативную и главную. Быстродействие сверхоперативной памяти соизмеримо или превышает быстродействие процессора. Обмен между каскадами оперативной памяти может осуществляться через ЦП или специальное устройство обмена. Сверхоперативная память имеет специальную организацию со своей внутренней схемой управления. Сверхоперативная память может строиться по принципу многовходовой или ассоциативной памяти. Сверхоперативная память построенная по буферному принципу называется кэш-памятью. Кэш память, как правило, строится по многоблочной архитектуре. с использованием алгоритмов ассоциативной обработки.

Различают 4 типа ЗУ:

- адресные;
- ассоциативные (информация отыскивается по признаку);
- ортогональные (можно считывать данные как по ячейкам, так и по разрядам);
- стековая.

Функции системы памяти организуются (ограничиваются)

1. процессором
2. системным интерфейсом (он позволяет осуществить доступ к громадному адресному пространству)
3. Основанная память (Оперативная память)
4. ВЗУ (внешняя ЗУ)



Уровни памяти:

Существуют структурные и алгоритмические **методы повышения быстродействия памяти**:

Структурные:

1. метод иерархической памяти
2. создание новых технологий и организации микросхем памяти.

Алгоритмические:

1. Пакетный доступ
2. Расслоение памяти
3. Метод блочной пересылки.

См (рис 6.1.)

#### Пакетный доступ

Выборка широким словом. На входе порции данных одинакового объема. На выходе n-слов. За одно обращение к ОП производится запись и чтение нескольких команд и слов. Сложность организации.

#### Расслоение памяти

Используется для организации попаременного обращения к разным физическим модулям при одновременной сокращении числа обращений к каждому модулю

Способы:

1. Расслоение с соответствием младшими адресами (конвейерный способ) рис. 6.4
2. Разделение памяти на память данных и память программ. Используется в системах управления и обработки сигналов. (Гавардская архитектура)

#### Метод блочной пересылки

За один сеанс обмена передается не одна порция данных.

Используется при обращении к ВЗУ, а также подкачки

данных в кэш-память или локальную память процессора и данных.

### **Виды ЗУ**

#### Структура адресного ЗУ.

Существует 2 вида адресов запоминающих устройств: статический и динамический.

Динамические ЗУ используют цикл регенерации, т.е. подзарядка ЗУ. Чем больше объём ЗУ, тем сложнее дешифрация адреса. Основное время записи:

$$t_{0\text{зап}} = t_c + t_d + t_{\text{зап}}; \quad t_{0\text{чтен}} = t_d + t_{\text{чт}} + t_p$$

$t_c$  – время стирания

$t_d$  – время дешифрации

$t_{\text{зап}}$  – время записи

$t_{\text{чт}}$  – время чтения

$t_p$  – время регенерации

Время записи и чтения определяется технологией изготовления кристалла памяти. А время дешифрации как технологически, так и особенностями организации блока памяти, поэтому время дешифрации можно уменьшить.

#### Структура ассоциативной памяти.

Поиск информации осуществляется не по адресу, а по ассоциативному признаку. При этом поиск по ассоциативному признаку происходит параллельно во времени для всех ячеек памяти. Ассоциативный поиск позволяет упростить и ускорить обработку информации. Это достигается за счёт того, что одновременно с выборкой происходит некоторая логическая обработка.

Запоминающий массив содержит  $N$  ячеек разрядностью  $N+1$ . Для указания занятости ячейки используется  $N$ -ный разряд. Если он установлен в 0 – ячейка свободна. По входнойшине на регистр ассоциативного признака поступает  $N$ -разрядный ассоциативный запрос, а на регистр маски – код маски поиска.  $N$ -ный разряд регистра маски устанавливается в 0. Ассоциативный поиск производится для совокупности разрядов регистра ассоциативного признака, у которого регистр маски установлен в 1.

Для слов, у которых соответствующие разряды совпадали с незамаскированными разрядами регистров ассоциативного признака, комбинационная схема устанавливает 1 в соответствующие разряды регистра совпадения. Регистр результата поиска просматривает содержимое регистра совпадения и формирует 3 выходных сигнала:

$a_0 = 0$ ;  $a_1 = 1$ ;  $a_2$  – больше одного; (совпадения).

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Обрабатывающее устройство, которое послало ассоциативный запрос на память по этим сигналам количество циклов чтения из памяти. Поиск данных в ассоциативной памяти может производиться с учётом различных методов индексации и адресации. Алгоритм индексации реализуется аппаратно или микропрограммно, и заменить его на другой нельзя.

### **Иерархическая организация памяти**

Уровни иерархии памяти имеют каскадное включение. Обмен между каскадами осуществляется через ЦП (центральный процессор) или через специальное устройство обмена. Эффект повышения быстродействия от иерархической организации памяти будет больше, если данные, находящиеся на определенном уровне, будут многократно использоваться.

Пакетная организация требуется для взаимодействия между всеми уровнями памяти, за исключением регистры $\leftarrow\rightarrow$ кэш процессора. Например, организация взаимодействия между оперативной памятью и внешней памятью идет через контроллер, который работает в пакетном режиме.

### **5. Организация кэш-памяти. Зарисуйте структуру памяти (ОЗУ и кэш) для секторированного наборно-ассоциативного кэша, состоящего из трех банков. Поясните её.**

В современных процессорах используют 2 кэша: кэш команд, кэш данных.

Из всей памяти доступной процессору, кэшируется только динамическая память системной платы. И из этой памяти не кэшируется область, где хранятся общесистемные переменные и область для организации режима ПДП.

Рис. 6.3

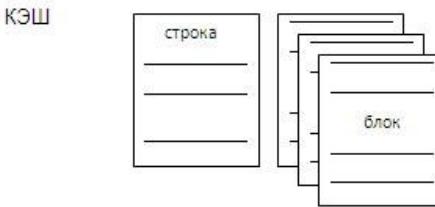
Кэш память включает собственное буферное запоминающее устройство, в котором хранятся информационные блоки.

Включает:

1. комбинационные схемы, необходимые для ассоциативного поиска признаков
2. запоминающие устройства для хранения таблиц адресов и таблиц активности

Процессор выполняет обмен только с оперативной памятью. Кэш контроллер перехватывает запрос и является посредником между процессором и оперативной памятью. Он должен обеспечивать обмен из кэш памяти данных когерентной памяти.

Рис 6.11

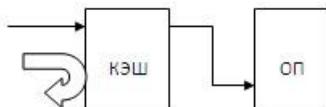


Если строка = блок, называется несекторированным.

Страницы ОП условно делятся на блоки. Кэш память разделена на строки. По системной шине данные передаются только блоками. Если размер строки = блоку, то кэш называется несекторированным. В каждой строке кэш памяти добавляется бит активности, который определяет занятость. Если строка КЭШа > блока, то кэш называют секторированным. Если строка КЭШа содержит несколько смежных блоков (секторов), такой кэш называют секторированным.

Существует 2 способа данных чтения из КЭШа.

1. Обращение к ОП начинается одновременно с кэш каталогом и в случае попадания прерывается.
2. Обращение к ОП начинается только после промаха.



### 15. 3 стратегии:

- 1) Стратегия псевдослучайного выбора места
- 2) Стратегия наиболее редко используемого места
- 3) Стратегия не модифицируемого места

Современные кэш - контроллеры организуют стратегию упреждающего чтения, т.е. свободные циклы памяти будут записываться в кэш строки, которые вероятнее всего понадобятся процессору. При этом учитывается направление процесса данных процессора.

Записи данных:

- 1) Сквозная запись WT – выполнение каждой операции записи производится одновременно и в строку КЭШа и в ОП.
- 2) Буферная сквозная запись BWT – между кэш контроллером и ОП располагается кэш буфер имеющий очередь на запись и на чтение (FIFO). Отложенная запись

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

выполняется во время свободных тактов шины. Если требуется прочитать блок который находится в буфере, то блок переписывается в очередь через чтение.

- 3) Данные заносятся в кэш и соответствующая строка помечается как модифицируемая. В ОП данные перемещаются только целой строкой в случае секторированного опроса или непосредственно перед её замещением новыми данными.
- 4) Буферная обратная запись.

В зависимости от способа определения, взаимного соответствия строки КЭШа и области основной памяти, выделяют 3 архитектуры кэш памяти:

- 1) КЭШ прямого отображения
- 2) Полностью ассоциативный КЭШ
- 3) Частично ассоциативный КЭШ (наборно - ассоциативный)

### **Наборно – ассоциативный КЭШ памяти**

Рис. 6.14

Реализован компромисс между ассоциативным КЭШом и КЭШом прямого отображения. Здесь каждый блок памяти может претендовать на одну из нескольких строк КЭШа объединённых в набор. Номер набора в котором модет отображаться требуемый блок однозначно определяется адресом блока. С каждым набором связан признак определяющий строку набора подлежащего замещению в случае КЭШ-промаха.

Рис. 6.15

### **6. Операционный и командный конвейер. Необходимые условия организации конвейеров этих типов. Режимы работы конвейеров. Объясните, почему при организации конвейера команд не целесообразно использовать Принстонскую архитектуру ЭВМ?**

Операционный конвейер (ОК) состоит из последовательности комбинационных схем, каждая из которых реализует определённый этап примитивной операции. Между комбинационными схемами располагаются регистры для хранения промежуточных результатов. Рис. 7.3

Такт работы определяется:

- 1) Задержка на распространение сигналов комбинационных схем.

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

$$\tau_{01}, \dots, \tau_{0n} \rightarrow \tau_{0m} = \max(\tau_{01}, \dots, \tau_{0n})$$

$$\tau_k = \tau_{0m}$$

2) Из задержки записи данных в регистр

$$\tau_r \quad \tau_k = \tau_{0m} + \tau_r$$

3) Время распространения сигнала по межсоединениям

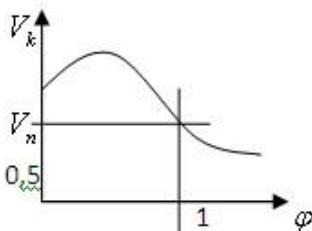
$$\tau_m \quad \tau_k = \tau_{0m} + \tau_r + \tau_m$$

4) Время задержки синхроимпульса

$$\tau_c \quad \tau_k = \tau_{0m} + \tau_r + \tau_m + \underbrace{\tau_c}_{\tau_s}$$

Определяет коэффициент эффективности использования конвейера. Второстепенные операции.

$$\varphi = \frac{\tau_s}{\tau_{0m}}$$



В современных процессорах параллельно устанавливают от 2-32 операционных конвейеров. Комбинационные конвейера могут соединяться последовательно, образуя линейный конвейер или по схеме с обратными связями.

Рис. 7.6

Рассмотрим векторный конвейер системы рис. 7.7 в состав которой входит векторный процессор рис. 7.8

Пример: Пусть необходимо выполнить следующую

$$C = \sum_{i=1}^L a_i * b_i$$

последовательность

1) LD L,A,V<sub>1</sub> загрузка

2) LD L,B,V<sub>2</sub> загрузка

регистр

} скалярный

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

- 3) MP  $V_1, V_2, V_3$  умножение  
 4) SUM  $V_3, S_1$  суммирование

См. рис. 7.9

Зацепление:

- 1) LD L,A, V<sub>1</sub>
- 2) ЗЦ V<sub>1</sub>,B,S<sub>1</sub>

На рис. 7.10 – приведена структура операционного конвейера

Пример конвейерных систем:

GREY (американская система) разрабатывается до сих пор

HEP

VP-200

S-810

### **Конвейер команд**

Для организации любого конвейера команд, необходимо выполнение 2-х условий:

- 1) Существование потока однотипных действий
- 2) Возможность разбиения каждого действия на ряд последовательных этапов.

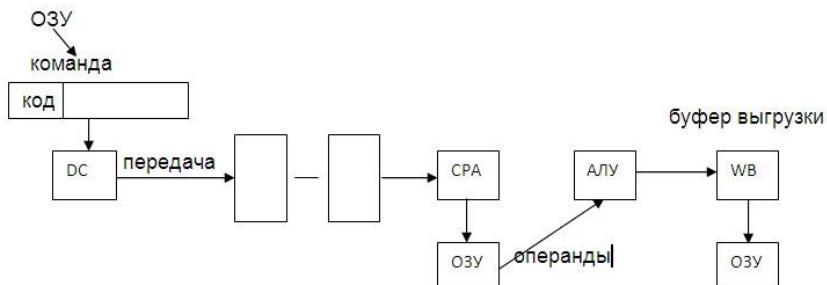
Каждая команда может быть разбита на несколько этапов и исполнения

Этап подготовки делится на:

- 1) Выборка кода команды из памяти
- 2) Декодирование
- 3) Передача команды на исполнение

Этап исполнения делится на:

- 1) Подготовка исполнительных адресов
- 2) Чтение команды с памяти
- 3) Исполнение команд



## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Для реализации каждого этапа должны быть аппаратные средства, которые освобождаются сразу после выполнения данного этапа текущей команды. В современных процессорах реализуется принцип суперскалярности – это принцип при котором команды не зависящие друг от друга могут обрабатываться в нескольких конвейерах. Чтобы не нарушался порядок следования результатов они помещаются в выходной буфер в соответствие с тэгами отражающими их порядковый номер в программе.

В составе структуры процессора Pentium можно увидеть 2 целочисленных конвейера команд. Устройство с плавающей точкой которое является продолжением главного конвейера и организована, как операционный конвейер.

- КЭШ памяти данных и команд
- устройство управления (управляющие ПЗУ)
- буфер предвыборки
- буфер предсказания переходов
- дешифратор команд
- устройство страничного преобразования
- шинное устройство

Оба конвейера аналогичны по структуре и порядку функционирования. Главный конвейер может выполнять все целочисленные команды и команды с плавающей точкой. Если две инструкции сразу запустить невозможно, то 1-я запускается в главном конвейере, а 2-я простояивает. Запуск команд производится одновременно в оба конвейера. Выполнение определенных ступеней конвейеров синхронизируется между собой.

подготовка	{	1-я ступень – ступень предвыборки PF
		2-я ступень – декодирования D1
		3-я - генерация адреса (адреса операндов)
D2	{	4-я - ступень исполнения EX
исполнение		5-я – обратная запись WB

Команда выбирается из КЭШ-памяти или из ВЗУ с учетом предсказаний переходов – позволяет продолжить выборку ID кодирования потоков инструкций, после выборки инструкция ветвления, не дожидаясь проверки самого условия.

Предсказание переходов 3-х типов:

- 1) Статические – она работает по схеме, считая что переходы по условию произойдут с большей вероятностью (не всегда эффективно).

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

- 2) Динамический – он упирается на предисторию вычислительного процесса. Эта информация собирается во время выполнения программы.
- 3) Исполнение по предположению. Предсказание после переходов инструкции не только декодируются, но и по возможности исполняются. По мимо предсказания переходов, выполняется алгоритм с изменениями последовательности.

Используется два независимых буфера предвыборки. Один применяется ... с предположением, что перехода нет, а другой, что переход есть.

На ступени D1 происходит декодирование и запуск двух команд. D2 – происходит вычисление операндов размещенных в памяти. EX – исполняются команды в АЛУ, происходит доступ к КЭШ-памяти данных.

WB – завершается выполнение команд, которое модифицирует состояние процессора. Реализуется с помощью 2-х буферов выгрузки и реализовано как устройство сдвига. В память будут записаны только те команды, где есть уверенность, что вычисление выбрано правильно. Устройство с плавающей точкой реализуется на 3-х ступенчатом командном конвейере. Основной конвейер может выполнять целочисленные инструкции независимо от инструкций с плавающей точкой.

Каждая КЭШ-память является частично – ассоциативной, она разбита на 8 банок, поэтому конвейеры могут обращаться к памяти одновременно.

Pentium 2 – 6 ступеней. Между 1-й и 2-й это выборка F

Pentium 3 – 12-17 - целочисленных, 25 – с плавающей точкой

Athlon – 10(15)

**7. Многопроцессорные вычислительные комплексы (МПВК).**

Типы структур. Проблемы организации. Способы распределения ресурсов в МПВК. Сравните типы структур МПВК по следующим критериям: а) быстродействию; в) аппаратным затратам на систему коммутации; г) надежности.

В ВС типа МКМД (множественный поток команд, множественный поток данных) множество вычислительных устройств реализует независимые потоки команд по обработке собственных локальных данных. В них используется параллелизм независимых ветвей и задач. Принцип параллельной обработки – пространственный.

Классические многопроцессорные комплексы выполняют крупнобlockчные операции, что позволяет уменьшить расходы на взаимодействие процессоров.

**Многопроцессорный вычислительный комплекс** – это комплекс, который состоит из нескольких процессоров, работающих с общей ОЗУ и общими периферийными устройствами под управлением единой ОС, которая организует весь процесс обработки. Единым ресурсом могут быть машины – посредники, ОЗУ, ВЗУ, общие шины, коммутаторы. Каноническая структура – объединение через общую память машины. Существует проблема повышения производительности:

1. Организация связи между элементами комплекса, т.к. практически каждый его элемент должен быть связан с остальными. При большом числе процессоров модули ОЗУ и каналов ввода-вывода – это достаточно сложно;

2. Организация вычислительного процесса в комплексе.

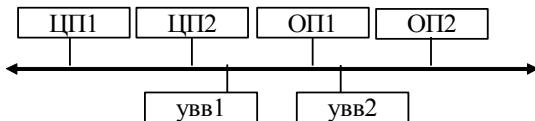
Задачи управления, которые решаются в комплексе:

1. Все задачи, которые встают при реализации мультипрограммного режима;
2. Распределение ресурсов и заданий между процессорами;
3. Синхронизация процессов при реализации несколькими процессорами одной задачи;
4. Разрешение конфликтных ситуаций при обращении нескольких процессоров к единому ресурсу;
5. Обеспечение работоспособности комплекса при выходе из строя какого-либо блока;
6. Планирование с учётом оптимизации загрузки всех процессоров.

Существует 3 типа структур многопроцессорного вычислительного комплекса:

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

1. С общим или раздельным во времени;
2. С перекрёстной коммутацией;
3. С многовходовой ОП.



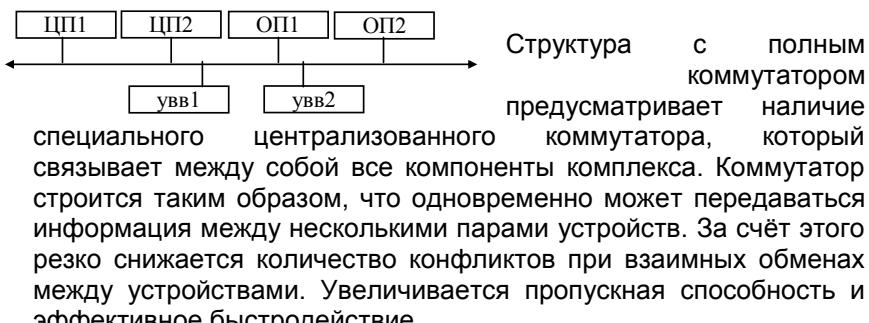
Все устройства соединяются общей совокупностью проводов, по которым передаются данные, адреса, команды и управляющие сигналы;

Достоинства:

- простота построения и использования;

Недостатки:

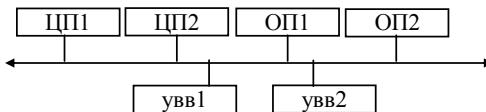
- при такой структуре одновременно обмениваться информацией могут 2 устройства, а остальные вынуждены ожидать освобождения шины;
- неизбежно возникновение конфликтов из-за совместного использования внешней шины;
- низкая эффективность использования оборудования, которое в данном случае больше простаивает, чем работает;
- при высокой интенсивности обмена между процессорами падает эффективная производительность комплекса. Путь комплекса. Прхитектурой делают 2-3 процессора;
- надёжность всего комплекса зависит от надёжности общей шины. Эту проблему решают резервированием шины.



Недостаток: сложность и высокая стоимость коммутатора. Для упрощения коммутатора и снижения его стоимости его делают состоящим из 2-х отдельных коммутаторов. Один является высокоскоростным и служит для обмена данными между

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

процессами, второй – низкоскоростной – для периферийного оборудования.



Использование многовходовой памяти в некоторой степени снижает недостатки

комплексной с перекрёстной коммутацией. Коммутация осуществляется в модулях ОЗУ, каждый из которых имеет число портов, равное числу остальных компонентов комплекса. Коммутатор распределён по всем модулям памяти

### **Способы организации вычислительного процесса в МПВК.**

Эффективность работы многопроцессорного комплекса в очень сильной степени зависит от способа распределения и использования аппаратных, программных и информационных ресурсов (т.е. от того, как будет организован вычислительный процесс). Наиболее простая организация вычислительного процесса в комплексах строится по принципу “ведущий-ведомый”. На один из процессоров возлагаются функции по управлению всеми остальными. Ведущий процессор распределяет задание и необходимые для их выполнения ресурсы. Такой подход исключает конфликты из-за ресурсов и уменьшает частоту внештатных ситуаций. При этом сам процессор является узким местом вычислительного комплекса.

Организация с раздельным выполнение заданий: все процессоры находятся в равных условиях. Они выполняют все функции, связанные с обработкой информации. Это достигается за счёт статического освобождения ресурсов одновременно с распределением заданий. В процессе выполнения заданий процессы перераспределяться не могут, что приводит к неравномерной загрузке процессора, каналов и памяти.

Симметричная обработка: при симметричной обработке устанавливается перечень задач и каждый процессор при освобождении 2-х предыдущих задач выбирает себе новую из общего перечня. При этом он выбирает необходимые ресурсы. Потенциально такая организация может обеспечить максимальную загрузку процессора. Недостатки: могут возникать конфликтные ситуации между процессорами (из-за общих ресурсами) и усложняется проблема синхронизации процессов.

- 8. Машины, управляемые потоком данных. Недостатки принципа управления потоком данных. Граф потока данных. Типы вершин графа потока данных. Можно ли использовать принцип управления потоком данных в конвейерных вычислительных системах? Ответ обоснуйте.**

В вычислительных системах, управляемых потоками данных, или машинах потоков данных (МПД) отсутствует понятие программы как последовательности команд, а следовательно, отсутствует понятие состояния процесса. Каждая инструкция передается на исполнение при создании условия для ее реализации; при наличии достаточных аппаратных средств одновременно может обрабатываться произвольное число готовых к исполнению инструкций. В инструкциях МПД параллелизм не задается явно, а аппаратные средства обработки должны его выявлять на этапе исполнения. Кроме того, в МПД используется метод передачи операндов между инструкциями по значению (а не по ссылке), что приводит к тому, что память не рассматривается как пассивное хранилище переменных.

Архитектура МПД отличается мелкоблочным характером и рассчитана на параллельное выполнение задач, плохо поддающихся векторизации, т.е. в алгоритмах которых отсутствует параллелизм объектов. Большинство архитектур МПД обладают свойством наращиваемости, позволяя увеличивать вычислительную мощность за счет добавления новых процессорных элементов. Однако реализация принципа управления потоком данных вызывает ряд трудностей, часть из которых носит принципиальный характер. К их числу необходимо отнести громоздкость программы, трудность обработки итерационных циклов, трудность работы со структурами данных.

Программа МПД. Хотя в различных МПД используются различные машинные языки, все они основаны на одинаковых принципах. Наиболее распространенной формой представления программы для МПД, которая носит название программы потока данных, является форма *графа потока данных* (ГПД). Граф потока данных является еще одной моделью вычислений; в его основе лежит информационный граф, а архитектура МПД является непосредственным отображением ГПД на аппаратуру. Используемая терминология заимствована из теории графов и сетей Петри.

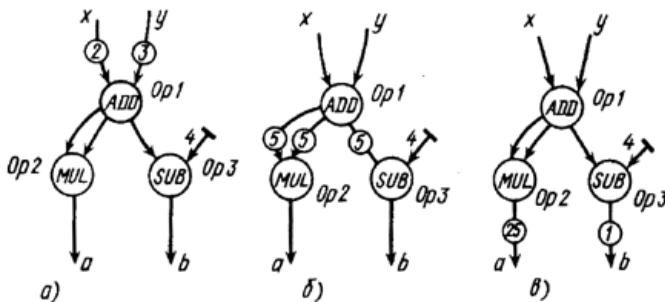
Инструкциям на ГПД соответствуют вершины, а дуги, обозначаемые стрелками, указывают на отношения предшествования. Точка вершины, в которую входит дуга, называется входным

## АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

портом (или входом), а точка, из которой она выходит,— выходным. По дугам передаются метки, называемые токенами данных.

Срабатывание вершины означает выполнение инструкции; оно

происходит в



произвольный момент времени при выполнении условий, соответствующих правилу запуска, или спусковой функции. Обычно используется строгое правило запуска, согласно которому срабатывание вершины происходит при наличии хотя бы одного токена на каждом из ее входных портов. Срабатывание вершины сопровождается удалением одного токена из каждого входного порта и размещением не более одного токена результата операции в выходные порты. В зависимости от конкретной архитектуры МПД порты могут хранить один или несколько токенов, причем они могут использоваться по правилу FIFO или в произвольном порядке.

На рис. 1, а приведен пример ГПД, включающего три вершины, обозначенные большими кружочками. Символы внутри этих кружочков обозначают код примитивной операции (ADD — сложения, MUL—умножения, SUB —вычитания). В двух входных портах вершины Op1 присутствуют токены, изображенные маленькими кружочками, внутри которых указаны значения переменных. Поскольку токены присутствуют на всех входных портах вершины Op1, операция может выполняться в произвольный момент, при этом токены данных из входных портов удаляются, а в выходные порты помещается токен результата (рис. 1, б). Константа, используемая при реализации вершины Op3, обозначается особым символом, так как она не удаляется из входного порта после срабатывания вершины. В результате срабатывания вершины Op1 создались условия для срабатывания вершин Op2

#### АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

и Ор3. Отметим, что если вершина формирует больше токенов, чем поглощает, то это, как правило, приводит к увеличению уровня параллелизма. В результате (срабатывания вершин Ор2 и Ор3 формируются токены результатов (рис. 1, в):

$$a = (x+y) = (2+3) = 25, b = (x+y)4 = (2+3) - 4 = 1.$$

Рассмотренные вершины являются функциональными, т.е. значения токена результата определяются только кодом операции Ор и значениями входных токенов.

**1. Разработка баз данных.**

**Пример №1.**

1. Проблемная область: «Заправочная станция»

2. Постановка задачи:

1) Обосновать для пользователя необходимость разработки базы данных в заданной проблемной области. Сформулировать требования к ней. Указать категории пользователей.

2) Разработать ER-диаграмму.

3) Описать состав и содержание таблиц базы (не менее 5 таблиц).

4) Указать первичные ключи и связи таблиц.

5) Обосновать, что таблицы находятся в третьей нормальной форме. При необходимости провести нормализацию.

6) Дать пример заполнения таблиц (не менее 5 записей в таблице).

7) Описать, какие пользователи и к каким таблицам должны иметь доступ. Какие виды доступа (чтение данных, включение записей, обновление, удаление)?

8) Как предполагается поддерживать целостность данных? Дать полные рекомендации с указанием Ваших таблиц.

9) Составить и проверить в FoxPro или SQL Explorer по 3 примера на каждый из следующих видов запросов SQL:

- к одной таблице;
- к нескольким с внутренними соединениями;
- к нескольким с внешними соединениями;
- сгруппированные запросы;
- запросы с подзапросами;
- запросы на включение, удаление, обновление

групп записей по заданным условиям;

– группы связанных между собой запросов, объединенных в транзакции.

10) Разработать подробное задание на программное обеспечение для работы с Вашей базой данных.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

### **Обоснование необходимости разработки базы данных в заданной проблемной области.**

Заданной проблемной областью является «Заправочная станция». Данное предприятие работает в сфере услуг и является посредником между крупными поставщиками топлива и розничными потребителями. В соответствии с этим можно предположить, что в данной сфере постоянно присутствует достаточно большой объем информации (о поставщиках, клиентах, закупках), требующий постоянного обновления как квалифицированным персоналом (менеджерами, координирующими поставки), так и менее квалифицированными работниками, непосредственно обслуживающими клиентов.

Наиболее оптимальным вариантом, позволяющим создать информационную систему, способную хранить, обеспечивать корректный доступ, ввод, модификацию, удаление данных, накопление статистической информации является разработка БД. Данное решение позволит создать гибкую систему управления информацией и ведения учета, обеспечит её сохранность и целостность, соблюдая требования удобства и простоты взаимодействия с системой.

**Основными требованиями** к данной базе можно считать удобство и безопасность доступа к информации. Для реализации этого принципа, логическая структура полномочий системы «Заправочная станция» будет отображена в структуре таблиц, доступ к которым будут иметь только определенные группы пользователей. Так, пользователи группы «Операторы» не смогут иметь доступ к данным о поставках, а среднее звено управления не сможет получить доступ к персональной информации поставщиков.

#### **Выделяемые группы пользователей:**

«Администраторы» – группа имеет полные права на операции с базой, также как и права передачи своих полномочий, заведения новых пользователей, таблиц и управления правами.

«Менеджеры» – группе предоставлены права вставки, удаления, модификации данных в таблицы поставок и поставщиков, может устанавливать цены и менять текущее количество топлива. Также возможны индивидуальные настройки прав отдельным менеджерам, с целью поддержки реально существующей организационной структуры системы.

«Операторы» – группа имеет ограниченные права: допускается вставка, корректировка и удаление (либо маркировка на

## ПРАКТИЧЕСКАЯ ЧАСТЬ

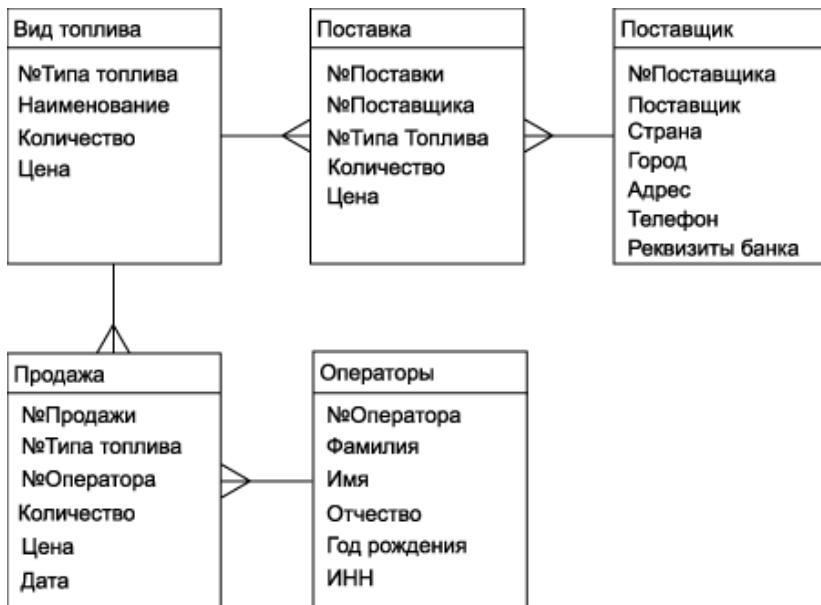
удаление) данных о продажах топлива (с изменением текущего количества топлива на станции), также просмотр информации о ценах.

### **ER-диаграмма.**

В базе данных «Заправочная станция» выделим следующие сущности:

[Вид топлива], [Поставщик], [Поставка], [Продажа], [Операторы]

На базе этих сущностей составим ER-диаграмму базы данных:



### **Описание состава и содержания таблиц базы.**

На базе ER-диаграммы составим структуру таблиц нашей БД (первичные ключи выделены жирным шрифтом):

Fuel (**Fuel\_id**, Fuel, Quantity\_available, Current\_price)

Suppliers(**Supplier\_id**, Supplier, Country, City, Address, Phone, Bank\_properties)

Delivery (**Delivery\_id**, Supplier\_id, Fuel\_id, Quantity\_delivered, Supplier\_price)

Employees (**Employee\_id**, LastName, FirstName, MiddleName, BirthDate, INN)

## ПРАКТИЧЕСКАЯ ЧАСТЬ

**Sale** (**Sale\_id**, **Fuel\_id**, **Employee\_id**, **Quantity\_sold**, **Sale\_price**, **Sale\_date**)

В каждой таблице присутствуют первичные ключи в виде автоинкрементных полей, позволяющих избежать дублирования текстовой информации и упростить процедуры транзакции. Таблицы 1, 2 и 4 предоставляют общую информацию о видах топлива, поставщиках и работниках «Заправочной станции». Таблицы 3 и 5 предоставляют сводную информацию о поставках (связывая данные о видах топлива и поставщиках) и продажах (связывая данные о проданных литрах топлива и продавцах, обслуживавших клиентов).

Таблицы 1 и 2 имеют связь один-ко-многим с таблицей 3.

Таблицы 1 и 4 имеют связь один-ко-многим с таблицей 5.

## **Нормализация.**

Данная организация БД отвечает требованиям третей нормальной формы, так как:

поля таблицы содержат неделимую информацию (*требование I НФ*)

отсутствуют повторяющиеся группы полей (*требование I НФ*) любое неключевое поле однозначно определяется первичным ключом таблицы (*требование II НФ*)

ни одно из ключевых полей не может однозначно идентифицироваться значением другого (неключевого) поля (*требование III НФ*)

## **Пример заполнения таблиц.**

Реализуем структуру БД в среде Delphi через DatabaseDesktop fuel.db

	Field Name	Type	Size	Key
1	Fuel_id	+		*
2	Fuel	A	25	
3	Quantity_available	N		
4	Current_price	\$		

suppliers.db

**ПРАКТИЧЕСКАЯ ЧАСТЬ**

	Field Name	Type	Size	Key
1	Supplier_id	+		*
2	Supplier	A	25	
3	Country	A	25	
4	City	A	25	
5	Address	A	50	
6	Phone	A	15	
7	Bank_properties	I		

delivery.db

	Field Name	Type	Size	Key
1	Delivery_id	+		*
2	Supplier_id	I		
3	Fuel_id	I		
4	Quantity_delivered	N		
5	Supplier_price	\$		

employees.db

	Field Name	Type	Size	Key
1	Employee_id	+		*
2	LastName	A	25	
3	FirstName	A	25	
4	MiddleName	A	25	
5	BirthDate	D		
6	INN	I		

sale.db

	Field Name	Type	Size	Key
1	Sale_id	+		*
2	Fuel_id	I		
3	Employee_id	I		
4	Quantity_sold	N		
5	Sale_price	\$		
6	Sale_data	D		

и приведем пример заполнения таблиц

(данные о поставщиках с сайта [http://www.riccom.ru/sale\\_market\\_r\\_np\\_11.htm](http://www.riccom.ru/sale_market_r_np_11.htm),

данные о ценах с сайта [http://www.au92.ru/msg/20040617\\_uyp/90x.html](http://www.au92.ru/msg/20040617_uyp/90x.html)):

Delivery_id	Supplier_id	Fuel_id	Quantity_delivered	Supplier_price
1	1	1	100	15,76р.
2	1	2	100	14,05р.
3	3	2	100	25,00р.
4	4	1	100	14,80р.
5	2	5	300	21,00р.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

Employee_id	Last Name	First Name	Middle Name	Birth Date	INN
1	Кукушкина	Елена	Николаевна		
2	Иванов	Иван	Васильевич		
3	Кугувалов	Анатолий	Сергеевич		
4	Андреев	Владимир	Владимирович		
5	Лисицина	Ольга	Сергеевна		
Fuel_id	Fuel	Quantity_available	Current price		
1	Бензин А-76	1000	13,97р.		
2	Бензин Аи-92	500	17,05р.		
3	Бензин Аи-95	750	18,33р.		
4	Бензин Аи-98	1000	21,80р.		
5	ДТ	450	16,86р.		
Sale_id	Fuel_id	Employee_id	Quantity_sold	Sale_price	Sale_data
1	1	1	15	16,00р.	
2	1	2	5	20,00р.	
3	3	5	50	22,00р.	
4	4	5	10	20,00р.	
5	5	1	12	30,00р.	
Supplier_id	Supplier	Country	City	Address	Phone
1	Хабаровский НПЗ	РФ	Хабаровск		
2	Ачинский НПЗ	РФ	Ачинск		
3	Омский НПЗ	РФ	Омск		
4	Новокуйбышевский НПЗ	РФ	Новокуйбышевск		
5	Лукойл-Волгограднефтеперек	РФ	Татьянка Прив.		

## **Права и полномочия пользователей.**

В целях обеспечения сохранности данных рекомендуется ввести принцип разделения полномочий – создать разные категории пользователей, имеющих ограниченный доступ к таблицам и определенные ограничения на права доступа. Более того, рекомендуется использовать правило «минимальных полномочий».

«Администраторы» – группа имеет полные права на операции с базой, также как и права передачи своих полномочий, заведения новых пользователей, таблиц и управления правами.

«Менеджеры» – группе предоставлены права вставки, удаления, модификации данных в таблицы поставок и поставщиков, может устанавливать цены и менять текущее количество топлива. Также возможны индивидуальные настройки прав отдельным менеджерам, с целью поддержки реально существующей организационной структуры системы.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

«Операторы» – группа имеет ограниченные права: допускается вставка, корректировка и удаление (либо маркировка на удаление) данных о продажах топлива (с изменением текущего количества топлива на станции), также просмотр информации о ценах. Просмотр и редактирование данных о поставщиках и поставках запрещено.

### **Поддержка целостности данных.**

Цель поддержки целостности данных – недопустить ввода некорректной информации (то есть проверять корректность связей с другими таблицами при вставке) и корректно обрабатывать удаление данных из таблиц, имеющих внешние связи. Для обеспечения целостности данных рекомендуется использовать механизм транзакций. При обнаружении нарушений целостности данных должны быть оповещены пользователи группы «Администраторы», а работа с базой прекращена до восстановления целостности данных.

Особое внимание следует уделить вставкам в таблицы поставок и продаж – имеющиеся в них ссылки на номера топлива, поставщика и оператора должны иметь соответствие с первичными ключами таблиц топлива, поставщиков и операторов.

Также особое внимание должно быть уделено удалению из таблиц Топлива, Поставщиков и Операторов. Через соответствующее программное обеспечение операторы базы должны быть уведомлены, что могут возникнуть неразрешимые внешние связи в таблицах Поставка и Продажа. Также должны быть предложены варианты действий (удаление соответствующих записей, переназначение, и вариант, в котором не будет делаться ничего).

### **Пример № 2.**

#### **Задание**

Имеется следующая проблемная область: столовая.

#### **Требуется:**

1. Обосновать для пользователя необходимость разработки базы данных в заданной проблемной области. Сформулировать требования к ней. Указать категории пользователей.
2. Разработать ER-диаграмму.
3. Описать состав и содержание таблиц базы (не менее 5 таблиц).

## ПРАКТИЧЕСКАЯ ЧАСТЬ

4. Указать первичные ключи и связи таблиц.
5. Обосновать, что таблицы находятся в третьей нормальной форме. При необходимости провести нормализацию.
6. Дать пример заполнения таблиц (не менее 5 записей в таблице).
7. Описать, какие пользователи и к каким таблицам должны иметь доступ. Какие виды доступа (чтение данных, включение записей, обновление, удаление) ?
8. Как предполагается поддерживать целостность данных? Дать полные рекомендации с указанием Ваших таблиц.
9. Составить и проверить в FoxPro или SQL Explorer по 3 примера на каждый из следующих видов запросов SQL:
  - 1) к одной таблице;
  - 2) к нескольким с внутренними соединениями;
  - 3) к нескольким с внешними соединениями;
  - 4) сгруппированные запросы;
  - 5) запросы с подзапросами;
  - 6) запросы на включение, удаление, обновление групп записей по заданным условиям;
  - 7) группы связанных между собой запросов, объединенных в транзакции.
10. Разработать подробное задание на программное обеспечение для работы с Вашей базой данных.

**Цель создания и требования к БД. Категории пользователей**  
Создание базы данных в данной проблемной области необходимо как для посетителей, так и для сотрудников столовой. Для посетителей – БД позволяет получить быстрый доступ к меню, узнать общую информацию о поставщиках продукции и сотрудниках столовой. Для работников столовой – БД предоставляет полную информацию о рецептах и продуктах, находящихся на складе. Для руководителей – БД помогает оптимизировать работу столовой, эффективно распоряжаться имеющимися продуктами и контролировать их использование.

Категории пользователей:

Guest – гость, случайный посетитель;

User – пользователь, постоянный клиент;

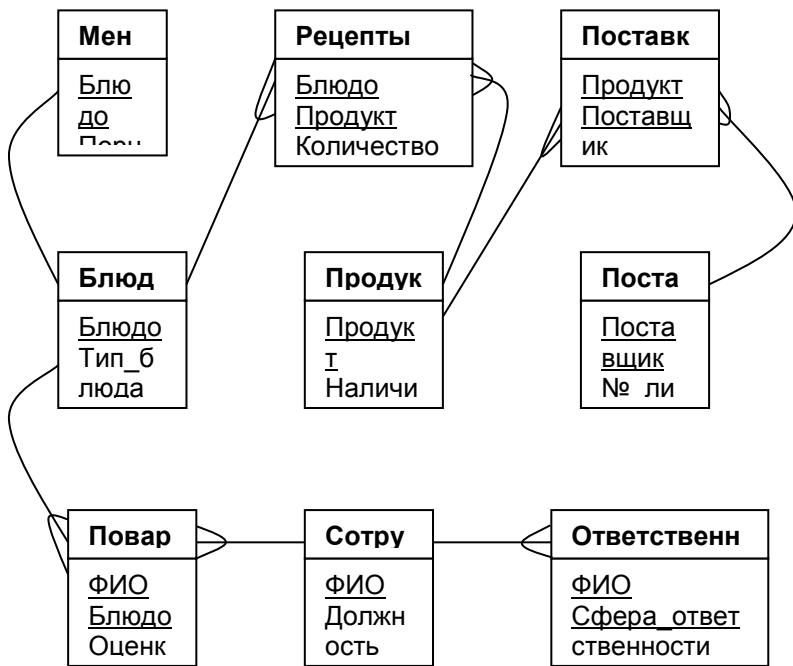
Worker – работник столовой (повар, буфетчица и т.п.);

Manager – руководитель (зав. производством, технолог);

Director – директор столовой;

Admin – администратор базы данных.

ПРАКТИЧЕСКАЯ ЧАСТЬ  
ER-диаграмма



Все отношения находятся в III нормальной форме, так как они находятся во II нормальной форме и в них нет транзитивных зависимостей.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Примеры заполнения таблиц

Меню

Блюдо	Порция (г.)	Цена (руб,коп)
Каша гречневая	200	7,50
Каша рисовая	200	9,20
Салат «Селедка под шубой»	150	12,30
Суп овощной	200	15,00
Чай	100	5,00

Блюда

Блюдо	Тип_блюда
Каша гречневая	Каши
Каша рисовая	Каши
Салат «Селедка под шубой»	Салаты
Суп овощной	Супы
Чай	Напитки

Продукты

Продукт	Наличие (кг.)	Срок_хранения (дней)
Картофель	2200	200
Капуста свежая	1000	100
Молоко	10	10
Морковь	200	150
Мясо (говядина)	500	50

Рецепты

Блюдо	Продукт	Кол-во_продукта	Способ_приготовления
Каша гречневая	Крупа гречневая	700 г. на 1 л. воды	Помыть, перебрать, высыпать в кипящую воду
Суп овощной	Капуста свежая	800 г. на 1 л. воды	Заложить в закипающий бульон
Суп овощной	Морковь	100 г. на 1 л. воды	Нарезать дольками, выложить в кипящий бульон
Суп овощной	Лук	10 г. на 1 л. воды	Нарезать дольками, выложить в кипящий

ПРАКТИЧЕСКАЯ ЧАСТЬ

			бульон
Суп овощной	Петрушка	5 г. на 1 л. воды	Добавить в конце варки

## Поставщики

Поставщик	№_лицензии	Телефон
ЗАО «ГлавРыба»	123-4567890	8-8302-999999
Мясокомбинат «Счастливая Буренка»	123-5567890	8-8302-556677
ОАО «МаслоСырВторПром»	123-6567890	8-8302-332211
ООО «Рога и копыта»	666-1300001	8-8302-666666
Совхоз «Светлый путь»	987-6453210	8-8301-443322

## Поставки

Продукт	Поставщик	Объем_поставки (кг.)
Картофель	Совхоз «Светлый путь»	5000
Капуста свежая	Совхоз «Светлый путь»	3000
Масло	ОАО «МаслоСырВторПром»	1000
Мясо (говядина)	Мясокомбинат «Счастливая Буренка»	1500
Мясо (говядина)	ООО «Рога и копыта»	1000

## Сотрудники

ФИО	Должность	Возраст	Стаж
Зыкова А.И.	Зав. производством	52	30
Иванов И.И.	Директор	50	20
Козлова Л.О.	Технолог	37	14
Петрова С.Г.	Повар	43	22
Сидоров А.А.	Грузчик	32	5

## ПРАКТИЧЕСКАЯ ЧАСТЬ

### Повара

ФИО	Блюдо	Оценка	Примечания
Волкова Е.С.	Чай	5	Только и знает, что чай гонять
Кузнецова Н.П.	Суп овощной	4	Иногда пересаливает
Орлов И.С.	Каша гречневая	5	Отличный кашевар
Орлов И.С.	Суп овощной	3	А вот суповар никудышный
Петрова С.Г.	Суп овощной	5	Спец по щам

### Ответственные лица

ФИО	Сфера ответственности
Зыкова А.И.	Прием продуктов
Зыкова А.И.	Качество блюд
Иванов И.И.	Переговоры с поставщиками
Иванов И.И.	Эвакуация в аварийных ситуациях
Козлова Л.О.	Качество блюд

### Доступ пользователей к таблицам

Пользователь	Виды доступа	Таблицы
Guest	READ	Меню, Ответственные_лица
User	READ	Меню, Сотрудники, Поставщики, Ответственные_лица
Worker	READ	Все таблицы
Manager	READ	Все таблицы
	WRITE	Меню, Продукты, Рецепты, Повара
Director	READ, WRITE	Все таблицы
Admin	READ, WRITE, CREATE, DROP	Все таблицы

READ – просмотр записей таблицы;

WRITE – добавление, удаление и изменение записей таблицы;

CREATE – создание новых таблиц;

DROP – удаление таблиц.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

SQL - запросы

Запрос к одной таблице.

Выбрать всех сотрудников столовой: `SELECT * FROM Сотрудники`  
Из всех блюд выбрать только каши:

`SELECT Блюдо FROM Блюда WHERE Тип_блюда='Каши'`

Выбрать сотрудников со стажем>10 лет и сортировать их по возрасту:

`SELECT * FROM Сотрудники WHERE Стаж>10 ORDER BY Возраст`

Запрос к нескольким таблицам с внутренними соединениями.

Получить номера телефонов поставщиков картофеля:

`SELECT Поставщики.Поставщик, Телефон, Продукт`

`FROM Поставщики, Продукты`

`WHERE (Поставщики.Поставщик=Продукты.Поставщик)`

`AND (Продукт='Картофель')`

Запрос к нескольким таблицам с внешними соединениями.

Получить рецепты с указанием наличия продуктов:

`SELECT Блюдо, Продукты.Продукт, Наличие`

`FROM Рецепты, Продукты WHERE`

`(Рецепты.Продукт*=Продукты.Продукт)`

Сгруппированный запрос.

Получить суммарные поставки для каждого продукта:

`SELECT Продукт, SUM(Объем_поставки) as Itogo`

`FROM Поставки GROUP BY Продукт`

Запрос с подзапросами.

Выбрать продукт, который никто не поставляет:

`SELECT Продукт, Наличие FROM Продукты`

`WHERE NOT EXISTS (SELECT * FROM Поставки`

`WHERE (Продукты.Продукт=Поставки.Продукт)`

`AND (Объем_поставки>0))`

Запрос на изменение групп записей по заданным условиям.

Установить порцию для любой каши – 300 г.:

`UPDATE Меню SET Порция=300`

`WHERE Блюдо IN (SELECT Блюдо FROM Блюда`

`WHERE Тип_блюда='Каши')`

Транзакции.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

Удаление сотрудника:

BEGIN TRANSACTION

DELETE FROM Повара WHERE ФИО='Петрова С.Г.'

DELETE FROM Ответственные\_лица WHERE ФИО='Петрова С.Г.'

DELETE FROM Сотрудники WHERE ФИО='Петрова С.Г.'

END TRANSACTION

## **Задание на разработку программного обеспечения для БД**

Необходимо создать систему, обеспечивающую управление базой данных «Столовая».

Эта система должна обеспечивать доступ пользователя к таблицам в соответствии с его категорией. Доступ на чтение позволяет просматривать записи таблицы, сортировать и фильтровать их, а также вести поиск по указанным значениям различных полей. Доступ на запись позволяет изменять, добавлять и удалять записи в таблице.

При изменении и добавлении записей необходимо контролировать правильность и целостность вводимой информации, а также проверять уникальность первичных ключей. При удалении записей из таблицы необходимо также удалять связанные с ней записи дочерних таблиц.

При изменении первичных ключей в таблице, необходимо изменять соответствующие внешние ключи во всех связанных с нею таблицах.

Необходимо реализовать удобный пользовательский интерфейс и обеспечить выдачу необходимых отчетов.

## **Примеры запросов к БД**

Приведем примеры запросов, которые могут быть реализованы к данной БД:

к одной таблице

select \* from sale;

результат:

Sale_id	Fuel_id	Employee_id	Quantity_sold	Sale_price	Sale_data
1	1	1	15	16,00р.	
2	1	2	5	20,00р.	
3	3	5	50	22,00р.	
4	4	5	10	20,00р.	
5	5	1	12	30,00р.	

select count(\*) from employees;

результат: 5

select fuel from fuel where Current\_price < 10;

## ПРАКТИЧЕСКАЯ ЧАСТЬ

результат: empty set

к нескольким с внутренними соединениями;

```
select employees.LastName,employees.FirstName, sale.sale_id from
employees, sale where sale.employee_id= employees.employee_id
and sale.quantity_sold*sale.sale_price>500;
```

результат:

LastName	FirstName	sale_id
Лисицына	Ольга	3

```
select distinct(fuel.fuel) from fuel, sale where sale.fuel_id=fuel.fuel_id;;
```

результат

fuel

Бензин А-76

Бензин Аи-95

Бензин Аи-98

ДТ

```
select suppliers.supplier, delivery.quantity_delivered from delivery,
suppliers where delivery.supplier_id=suppliers.supplier_id and
delivery.quantity_delivered >= 300;
```

результат:

Supplier	quantity_delivered
Ачинский НПЗ	300

к нескольким с внешними соединениями;

```
select * from fuel left join sale on fuel.fuel_id=sale.fuel_id;
```

fuel_id	Fuel	Quantity_available	Current_price	Sale_id	fuel_id_1	Employee_id	Quantity_sold	Sale_price	Sale_data
1	Бензин А-76	1000	13.97р.	1	1	1	15	16,00р.	
1	Бензин А-76	1000	13.97р.	2	1	2	5	20,00р.	
2	Бензин Аи-92	500	17,05р.						
3	Бензин Аи-95	750	18,33р.	3	3	5	50	22,00р.	
4	Бензин Аи-98	1000	21,80р.	4	4	5	10	20,00р.	
5	ДТ	450	16,86р.	5	5	1	12	30,00р.	

```
select * from fuel right join sale on fuel.fuel_id=sale.fuel_id;
```

fuel_id	Fuel	Quantity_available	Current_price	Sale_id	fuel_id_1	Employee_id	Quantity_sold	Sale_price	Sale_data
1	Бензин А-76	1000	13.97р.	1	1	1	15	16,00р.	
1	Бензин А-76	1000	13.97р.	2	1	2	5	20,00р.	
3	Бензин Аи-95	750	18,33р.	3	3	5	50	22,00р.	
4	Бензин Аи-98	1000	21,80р.	4	4	5	10	20,00р.	
5	ДТ	450	16,86р.	5	5	1	12	30,00р.	

```
select * from fuel full join sale on fuel.fuel_id=sale.fuel_id;
```

fuel_id	Fuel	Quantity_available	Current_price	Sale_id	fuel_id_1	Employee_id	Quantity_sold	Sale_price	Sale_data
1	Бензин А-76	1000	13.97р.	1	1	1	15	16,00р.	
1	Бензин А-76	1000	13.97р.	2	1	2	5	20,00р.	
2	Бензин Аи-92	500	17,05р.						
3	Бензин Аи-95	750	18,33р.	3	3	5	50	22,00р.	
4	Бензин Аи-98	1000	21,80р.	4	4	5	10	20,00р.	
5	ДТ	450	16,86р.	5	5	1	12	30,00р.	

сгруппированные запросы;

```
select count(fuel_id), fuel_id from sale group by fuel_id;
```

COUNT fuel\_id

2	1
---	---

1	3
---	---

1	4
---	---

## ПРАКТИЧЕСКАЯ ЧАСТЬ

```

1          5
select sum(quantity_sold), fuel_id from sale group by fuel_id;
SUM      fuel_id
20         1
50         3
10         4
12         5
select avg(quantity_delivered), supplier_id from delivery group by
supplier_id;
AVERAGE   supplier_id
100        1
300        2
100        3
100        4

```

запросы с подзапросами;

```

select * from suppliers where
supplier_id=(select supplier_id from delivery where
quantity_delivered=(select max(quantity_delivered) from delivery));

```

результат

Supplier_id	Supplier	Country	City	Address
2	Ачинский НПЗ	РФ	Ачинск	

```

select * from fuel where fuel_id=(select fuel_id from sale where
sale_id=(select max(sale_id) from sale));

```

результат

Fuel_id	Fuel	Quantity_available	Current_price
5	ДТ	450	16,86р.

```

select LastName, FirstName from employees where
employee_id=(select fuel_id from sale where sale_id=(select
max(sale_id) from sale));

```

результат

LastName	FirstName
Лисицина	Ольга

запрос на включение

```

insert into employees(LastName, FirstName,MiddleName) values
('Наумов', 'Константин', 'Леонидович');

```

запрос на удаление

```

delete from delivery where fuel_id=5;

```

запрос на обновление групп записей по заданным условиям;

```

update fuel set current_price=20 where fuel_id=1;

```

– группы связанных между собой запросов, объединенных в транзакции.

```

delete from suppliers where supplier_id=3;

```

```

delete from delivery where supplier_id=3;

```

#### ПРАКТИЧЕСКАЯ ЧАСТЬ

```
delete from suppliers where supplier_id=4;  
update delivery set supplier_id=5 where supplier_id=4;  
insert into employees(LastName, FirstName, MiddleName) values  
(‘Михеев’, ‘Сергей’, ‘Сергеевич’);  
select max(employe_id) from employees; (результат 5)  
insert into sale(Fuel_id, Employ_id, Quantity_sold, Sale_price) values  
(1,5,50,23.4);
```

#### **Руководство программисту.**

От программиста, обслуживающего данную БД, требуется создать устойчивое к сбоям, масштабируемое программное решение, позволяющее обеспечить удобный пользовательский интерфейс, с учетом предложенной структуры разделения прав и полномочий.

В программном обеспечении к базе должны присутствовать следующие компоненты:

Формы ввода данных в таблицы, с учетом требований целостности и проверкой прав доступа.

Формы просмотра и изменения существующих записей БД

Отчеты по каждой таблице в отдельности и по таблицам с учетом связей.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

### 2. UML диаграммы.

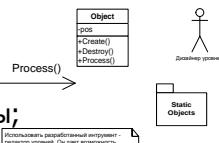
## | Типы диаграмм UML 1.X



## | Объекты в UML

В UML имеются четыре разновидности объектов(предметов) :

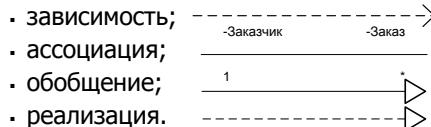
- структурные предметы;
- предметы поведения;
- группирующие предметы;
- поясняющие предметы.



Эти предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для описания моделей.

## Отношения в UML

- В UML имеются четыре разновидности отношений:



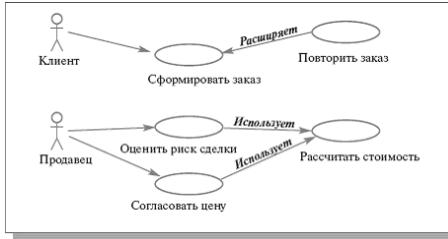
- Эти отношения являются базовыми строительными блоками отношений.

1. Диаграмма прецедентов.

## Пример use case diagram

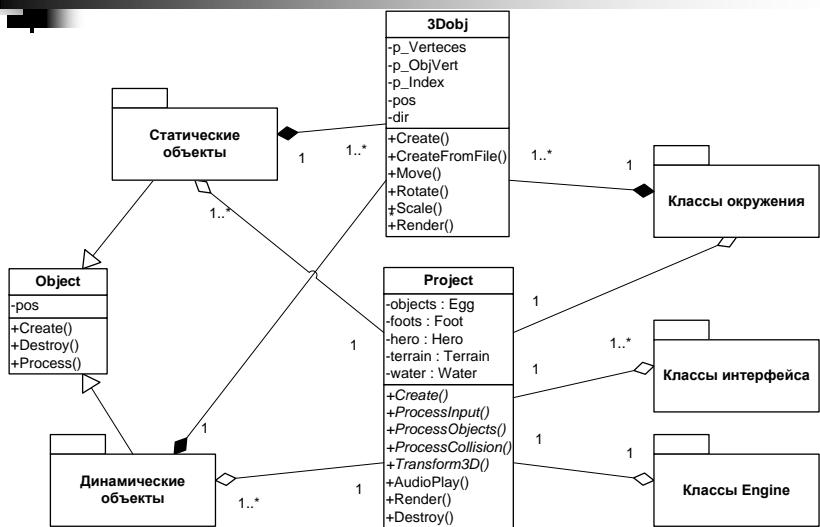


## ПРАКТИЧЕСКАЯ ЧАСТЬ

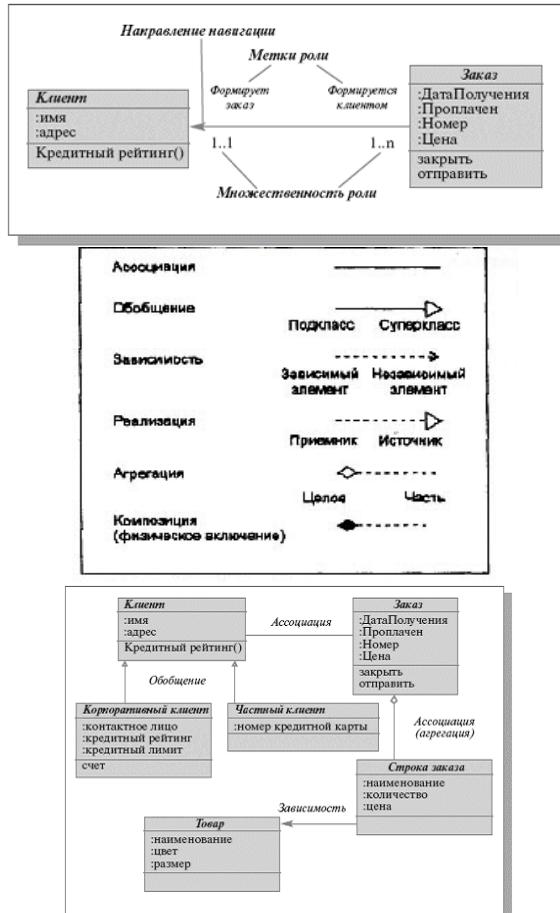


2. Диаграмма классов.

## Статическая структура с группами классов



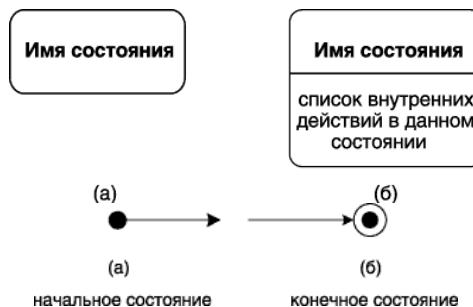
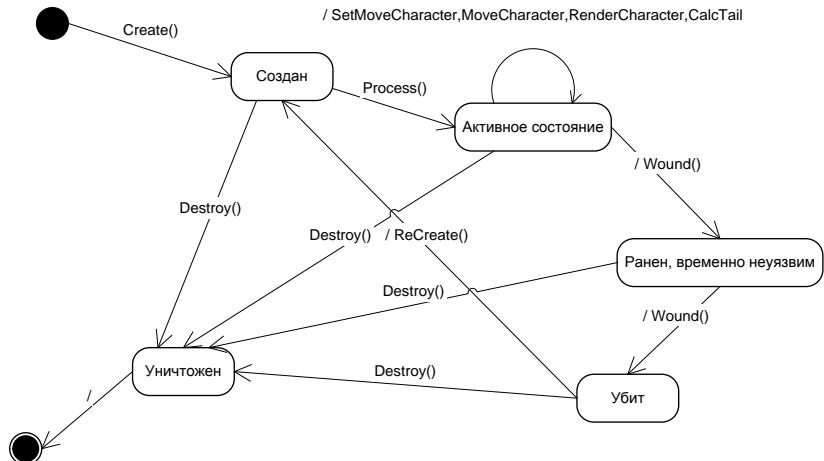
## ПРАКТИЧЕСКАЯ ЧАСТЬ



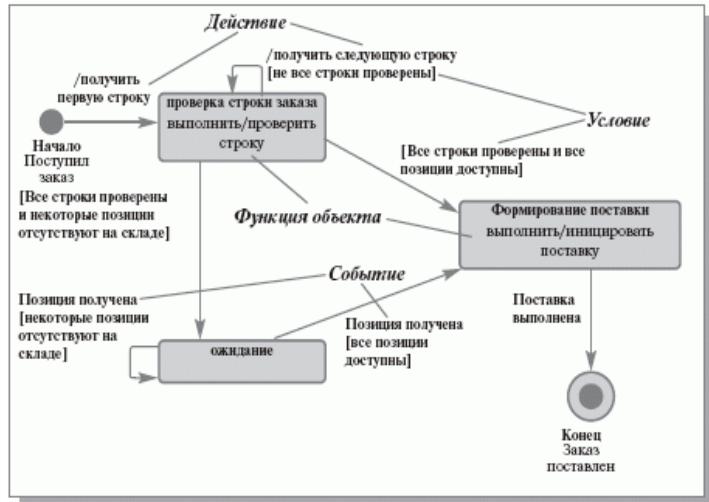
ПРАКТИЧЕСКАЯ ЧАСТЬ

3. Диаграмма состояний.

# StateChart для объекта Hero

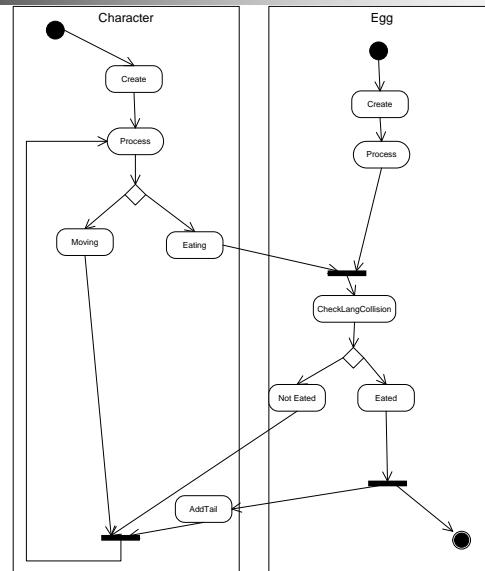


## ПРАКТИЧЕСКАЯ ЧАСТЬ

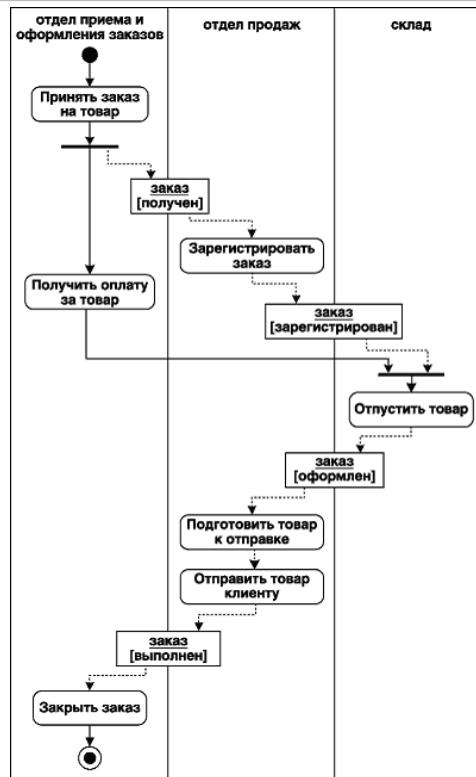
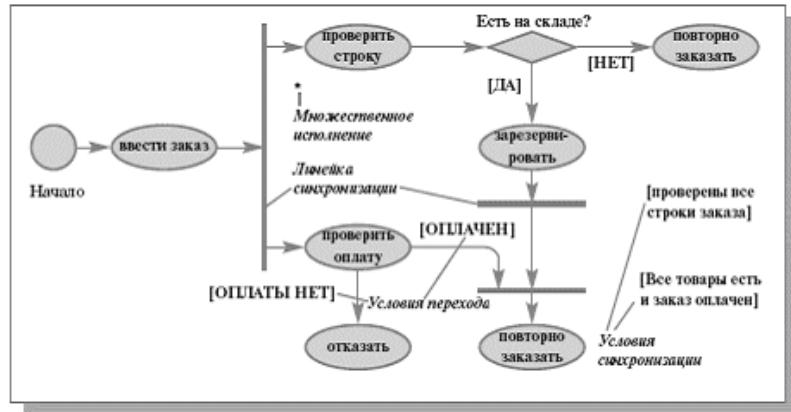


4. Диаграмма активности.

# Пример activity diagram



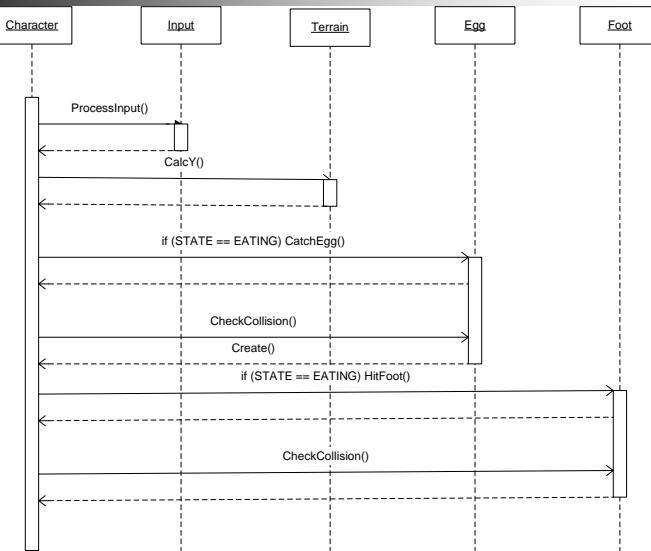
## ПРАКТИЧЕСКАЯ ЧАСТЬ



ПРАКТИЧЕСКАЯ ЧАСТЬ

5. Диаграмма последовательности.

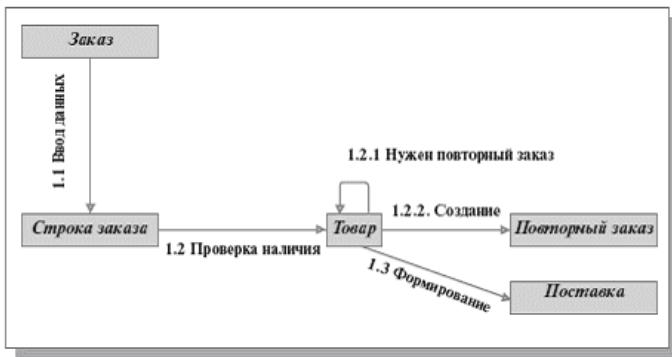
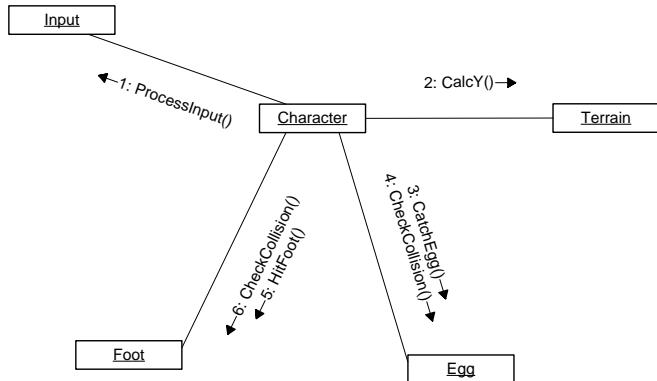
# Пример sequence diagram



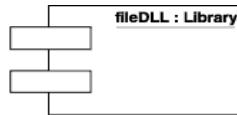
## ПРАКТИЧЕСКАЯ ЧАСТЬ

### 6. Диаграмма кооперации.

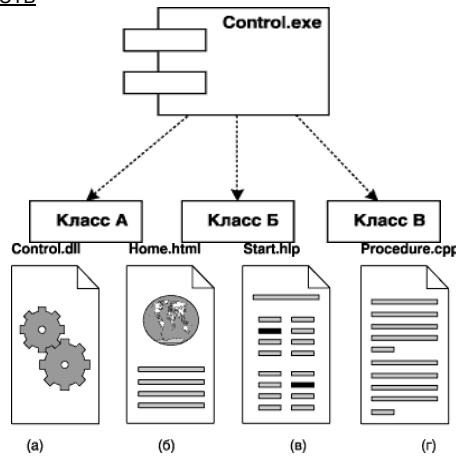
## Пример collaboration diagram



### 7. Диаграмма компонентов.



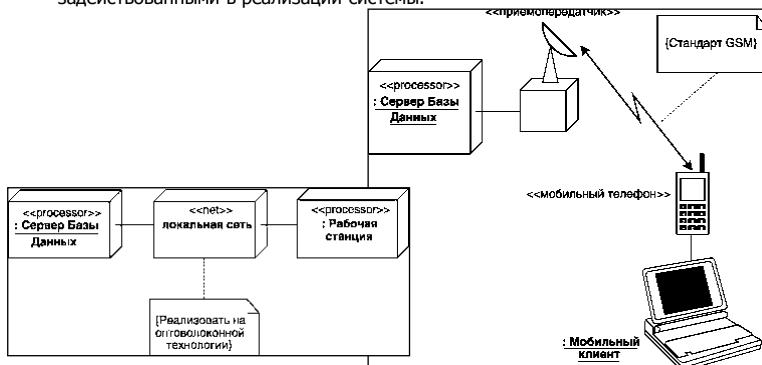
## ПРАКТИЧЕСКАЯ ЧАСТЬ



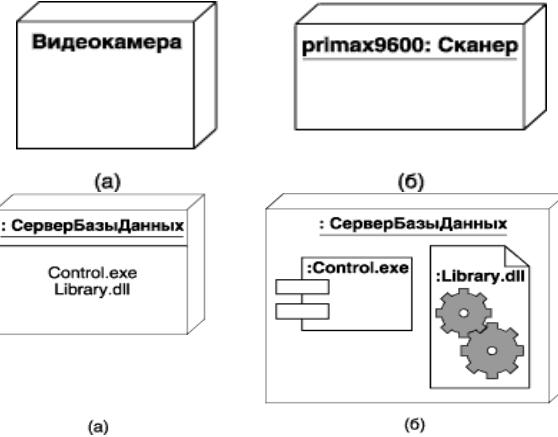
6. Диаграмма размещения.

# Пример deployment diagram

Диаграмма развертывания применяется для представления общей конфигурации и топологии распределенной программной системы и содержит изображение размещения компонентов по отдельным узлам системы. Кроме того, диаграмма развертывания показывает наличие физических соединений - маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.



## ПРАКТИЧЕСКАЯ ЧАСТЬ



### **3. Написание технического задания**

Техническое задание к подсистеме «Студенческий рейтинг»  
УДК 002:651.7/.78:006.354

#### **ВВЕДЕНИЕ**

Для определения лучших студентов необходимо построить их рейтинг.

Поэтому требуется разработать программу, которая автоматизировала бы работу по вводу и контролю рейтинга студентов. Данная программа должна интегрироваться в пакет информатизации управления учебного процесса, разрабатываемый для УМУ.

#### **ОСНОВАНИЕ ДЛЯ РАЗРАБОТКИ**

Данная разработка ведется на основании документа “Постановка задачи” от 3 февраля 2005 года. Задание выдал начальник отдела информатизации УМУ Нехорошкова Л.Г., разработка ведется при участии Матвеевой М.Н., исполнителем является программист отдела информатизации УМУ Семенов С.В. Условное название разработки – система “Рабочие планы”.

#### **НАЗНАЧЕНИЕ РАЗРАБОТКИ**

Система “Рейтинг” предназначена для следующих групп пользователей:

## ПРАКТИЧЕСКАЯ ЧАСТЬ

**“Секретари”.** В эту группу пользователей входят секретари деканатов. Пользователи этой группы вводят информацию о деятельности студентов, на основе которой строится рейтинг.

## **ТРЕБОВАНИЯ К ПРОГРАММЕ**

### **Общие требования**

Система “Рейтинг студентов” должна получать данные об успеваемости, научно-исследовательской работе, общественной работе, спортивной и художественной деятельности студента, а также деятельность в общественных организациях.

Программа интегрирована в систему деканат.

Должна обеспечиваться возможность сохранения на диске всех отчетов, генерируемых системой “Рейтинг студентов”. К отчетам системы относятся: рейтинг студентов по университету, по факультету и академическим группам, отчеты по эффективности работы деканатов, кафедр и кураторов групп.

### **Требования к функциональным характеристикам**

#### **Функции, предоставляемые системой “Рейтинг студентов”:**

Ввод, корректирование и удаление информации

об научно исследовательской работе (вводится балл)

об общественной работе студента: староста группы, дежурство по корпусу, общественные работы (в т.ч. социальная работа), добровольное выполнение общественных обязанностей

о спортивной жизни студент: участие, победы в соревнованиях, наличие спортивного разряда

о художественной самодеятельности

о членстве и работа в общественных организациях

Оперативное информирование пользователя

о балле студента на текущий момент

Вывод рейтинга студентов в табличном виде (Excel) для просмотра

#### **Требования к надежности**

Функционирование данной программы не должно нарушать целостности остального программного обеспечения, не должно нарушать работы остальных программ. В случае аварийного завершения программы целостность уже сохраненных в файлах учебных планов не должна нарушаться.

### **Условия эксплуатации**

## ПРАКТИЧЕСКАЯ ЧАСТЬ

Условия эксплуатации программной системы тождественны условиям эксплуатации аппаратного обеспечения, на котором будет установлен и эксплуатироваться данный программный комплекс. Эти требования можно узнать в документации, прилагаемой к компьютерам.

### **Требования к составу и параметрам технических и программных средств**

Система “Рейтинг студентов” должна работать на любой аппаратной платформе, требования к которой описаны в документации к ОС MS Windows 95.

Для просмотра планов и отчетов, генерируемых системой, необходимо наличие программы Microsoft Excel из пакета Microsoft Office 97 или выше.

### **Требования к информационной и программной совместимости**

Система “Рейтинг студентов” должна функционировать в любой из существующих на момент сдачи программной системы Win32 платформ, то есть должна удовлетворять всем требованиям компании Microsoft, предъявляемых к программным продуктам с логотипом **MS Windows compatible**.

Система должна работать с базой данных, разработанной ОИ УМУ для системы оптимизации управления учебным процессом.

Генерируемые системой отчеты должны быть совместимы по формату с программой Microsoft Excel 97 и выше.

### **ТРЕБОВАНИЯ К ПРОГРАММНОЙ ДОКУМЕНТАЦИИ**

Программа должна быть укомплектована следующей документацией:

Техническое задание

Документация по применению данного продукта

### **СТАДИИ И ЭТАПЫ РАЗРАБОТКИ**

№	Дата начала	Дата окончания	Работа	Исполнитель
1			Эскизный проект	Семенов С.В.
2			Доработка эскизного проекта	Семенов С.В.
3			Тестирование	Семенов

ПРАКТИЧЕСКАЯ ЧАСТЬ

		проекта	С.В. Матвеева М.Н.
4		Доработка проекта	Семенов С.В.
5		Внедрение проекта	Нехорошко ва Л. Г.

**ПОРЯДОК КОНТРОЛЯ И ПРИЕМКИ**

На каждом этапе разработки программа должна быть протестирована совместно разработчиком, заказчиком и третьим независимым лицом на предложенных заказчиком тестах. Результаты фиксируются в протоколе тестирования. Исправление ошибок должно быть закончено до даты начала следующей стадии проекта. Если программа выполняется на 100% тестов, она считается принятой заказчиком в опытную эксплуатацию.

**4. Написание постановки задачи.**

(для подсистемы «Студенческий рейтинг»)

**1. Основные требования:**

Требуется разработать программу, которая автоматизировала бы работу по вводу и контролю рейтинга студентов. Данная программа должна являться частью модуля подсистемы «Деканат». На основе системы:

- определяются лучшие студенты университета, факультетов и академических групп;
- представляются характеристики студентам по требованию сторонних организаций;
- оценивается эффективность работы деканатов, кафедр и кураторов по работе со студенческими группами по формированию их гражданской позиции;

Подсчет баллов проводится деканатами один раз в год на основе системы «Сессия» с учетом выполнения студентом графика учебного процесса и приказов (распоряжений) ректора и распоряжений декана факультета.

Шкала бальной оценки деятельности студентов находится в приложении.

**2. Требования к характеристикам системы:**

- Ввод, корректирование и удаление информации

#### ПРАКТИЧЕСКАЯ ЧАСТЬ

- Оперативное информирование пользователя о балле студента на текущий момент
- Вывод рейтинга студентов в табличном виде (Excel) для просмотра (не реализовано)

#### 3. Пользователи системы:

Система “Рейтинг” предназначена для группы пользователей “Секретари”. В эту группу пользователей входят секретари деканатов. Пользователи этой группы вводят и корректируют информацию, на основе которой формируется рейтинг студентов.

#### 4. Технические ресурсы:

1. Процессор Pentium III или совместимый
  2. 128 МБ оперативной памяти
  3. 100 МБ свободного места на жестком диске
  4. VGA дисплей
  5. Клавиатура, мышь
- Программа должна поддерживать работу на операционных системах семейства NT.

#### 6. Сроки:

На разработку предоставляется 1 месяц.

#### 7. Внедрение:

Программа является модулем системы «Деканат». Предполагаемое внедрение первоначально в деканате ФЛХиЭ, затем во всех деканатах университета.

#### 8. Предполагаемый эффект от внедрения

Внедрение позволит проводить оценку эффективности учебно-воспитательной работы в области формирования активной гражданской позиции студентов.

### **5. ГОСТ о стадиях разработки.**

ГОСТ 19.102-77

УДК 002:651.7/.78:006.354

ЕСПД. СТАДИИ РАЗРАБОТКИ.

ВЫПИСКА

1. Настоящий стандарт устанавливает стадии разработки программ и программной документации для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

2. Стадии разработки, этапы и содержание работ должны соответствовать

## ПРАКТИЧЕСКАЯ ЧАСТЬ

указанным в таблице.

Стадии Разработки	Этапы работ	Содержание работ
1. Техническое задание	Обоснование необходимости разработки программы	<p>Постановка задачи. Сбор исходных материалов. Выбор и обосн. критериев эффекта и качества разрабатываемой программы.</p>
	Научно-исследовательские работы	<p>Обоснование необх. проведения научно-исследовательских работ Опред. структуры входных и выходных данных.</p>
	Разработка и утверждение технического задания	<p>Предв. выбор методов решения задач. Обосн. целесообр-ти применения ранее разработанных программ. Определение требований к техническим средствам. Обосн. принципиальной возможности решения поставленной задачи. Определение требований к программе. Разработка технико-экономического обоснования разработки программы. Определение стадий, этапов и сроков разработки программы и документации на нее. Выбор языков программирования. Определение необходимости проведения научно-исследовательских работ на последующих стадиях. Согласование и утвержд. ТЗ</p>
2. Эскизный проект	Разработка эскизного проекта	<p>Предварительная разработка структуры входных и выходных данны. Уточнение методов решения задачи. Разработка общего описания алгоритма решения задачи. Разработка технико-экономического обоснования. Разработка пояснительной записки. Согласование и утверждение эскизного проекта.</p>
	Утверждение эскизного проекта	<p>Уточнение структуры входных и выходных данных. Разработка алгоритма решения задачи. Определ. формы представления входных и выходных данных. Определение семантики и синтаксиса языка. Разработка структуры программы. Окончательное определение конфигурации технических средств.</p>
3. Технический проект	Разработка технического проекта	<p>Разработка плана мероприятий по разработке и внедрению программ.</p>
	Утверждение технического	501

<u>ПРАКТИЧЕСКАЯ ЧАСТЬ</u>	
4. Рабочий проект	проекта Разработка программы Разработка программной документации Испытания программы
5. Внедрение	Подготовка и передача программы

Процесс разработки программы включает в себя следующие этапы:

- Разработка пояснительной записки. Согласование и утверждение технического проекта.
- Программирование и отладка программы.
- Разработка программных документов в соответствии с требованиями ГОСТ 19.101-77
- Разработка, согласование и утверждение программы и методики испытаний.
- Проведение предварительных государственных, межведомственных, приемо-сдаточных и других видов испытаний.
- Корректировка программы и программной документации по результатам испытаний.
- Подготовка и передача программы и программной документации для сопровождения и (или) изготовления.
- Оформление и утвержд. акта о передаче программы на сопровождение и (или) изготовление.
- Передача программы в фонд алгоритмов и программ.

#### Примечания:

1. Допускается исключать вторую стадию разработки, а в технически обоснованных случаях - вторую и третью стадии. Необходимость проведения этих стадий указывается в техническом задании.
2. Допускается объединять, исключать этапы работ и (или) их содержание, а также вводить другие этапы работ по согласованию с заказчиком.

## 6. ГОСТ о техническом задании.

ГОСТ 19.201-78 (СТ СЭВ 1627-79)

УДК 651.7/.78:002:006.354

ЕСПД. ТЕХНИЧЕСКОЕ ЗАДАНИЕ. ТРЕБОВАНИЯ К СОДЕРЖАНИЮ И ОФОРМЛЕНИЮ.  
ВЫПИСКА

Настоящий стандарт устанавливает порядок построения и оформления технического задания на разработку программы или программного изделия для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

### 1. Общие положения.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

1.1 ...

1.2 ...

1.3 ...

1.4 Техническое задание должно содержать следующие разделы:

введение;

основания для разработки;

назначение разработки;

требования к программе или программному изделию;

требования к программной документации;

технико-экономические показатели;

стадии и этапы разработки;

порядок контроля и приемки;

в техническое задание допускается включать приложения.

В зависимости от особенностей программы или программного изделия

допускается уточнять содержание разделов, вводить новые разделы или

объединять отдельные из них.

2. Содержание разделов.

2.1 В разделе "Введение" указывают наименование, краткую характеристику

области применения программы или программного изделия и объекта, в

котором используют программу или программное изделие.

2.2 В разделе "Основания для разработки" должны быть указаны:

документ (документы), на основании которых ведется разработка;

организация, утвердившая этот документ, и дата его утверждения.

наименование и (или) условное обозначение темы разработки.

2.3 В разделе "Назначение разработки" должно быть указано функциональное и эксплуатационное назначение программы или программного изделия.

2.4 Раздел "Требования к программе или программному изделию" должен

содержать следующие подразделы:

требования к функциональным характеристикам;

## ПРАКТИЧЕСКАЯ ЧАСТЬ

требования к надежности;

условия эксплуатации;

требования к составу и параметрам технических средств;

требования к информационной и программной совместимости;

требования к маркировке и упаковке;

требования к транспортированию и хранению;

специальные требования.

2.4.1 В подразделе "Требования к функциональным характеристикам"

должны быть указаны требования к составу выполняемых функций,

организации входных и выходных данных, временным характеристикам и т.п.

2.4.2 В подразделе "Требования к надежности" должны быть указаны

требования к обеспечению надежного функционирования (обеспечения

устойчивого функционирования, контроль входной и выходной информации,

время восстановления после отказа и т.п.).

2.4.3 В подразделе "Условия эксплуатации" должны быть указаны условия

эксплуатации (температура окружающего воздуха, относительная влажность

и т.п. для выбранных типов носителей данных), при которых должны

обеспечиваться заданные характеристики, а также вид обслуживания,

необходимое количество и квалификация персонала.

2.4.4 В подразделе "Требования к составу и параметрам технических

средств" указывают необходимый состав технических средств с указанием

их основных технических характеристик.

2.4.5 В подразделе "Требования к информационной и программной

совместимости" должны быть указаны требования к информационным

структурям на входе и выходе и методам решения, исходным кодам, языкам

программирования и программным средствам, используемым программой.

При необходимости должна обеспечиваться защита информации и программ.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

2.4.6 В подразделе "Требования к упаковке и маркировке" в общем случае

указывают требования к маркировке программного изделия, варианты и способы упаковки.

2.4.7 В подразделе "Требования к транспортированию и хранению" должны

быть указаны для программного изделия условия транспортирования, места

хранения, условия хранения, условия складирования, сроки хранения в

различных условиях.

2.5а В подразделе "Требования к программной документации" должен быть

указан предварительный состав программной документации и, при

необходимости, специальные требования к ней.

2.5 В разделе "Технико-экономические показатели" должны быть указаны:

ориентировочная экономическая эффективность, предполагаемая годовая

потребность, экономические преимущества разработки по сравнению с

лучшими отечественными и зарубежными образцами или аналогами.

2.6 В разделе "Стадии и этапы разработки" устанавливают необходимые

стадии разработки, этапы и содержание работ (перечень программных

документов, которые должны быть разработаны, согласованы и утверждены), а также, как правило, сроки разработки и

определяют

исполнителей.

2.7 В разделе "Порядок контроля и приемки" должны быть указаны виды

испытаний и общие требования к приемке работы.

2.8 В приложениях к техническому заданию, при необходимости, приводят:

перечень научно-исследовательских и других работ, обосновывающих

разработку;

схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие

документы, которые могут быть использованы при разработке;

## ПРАКТИЧЕСКАЯ ЧАСТЬ

другие источники разработки.

## **7. ГОСТ о видах программ и программных документов.**

ГОСТ 19.101-77 (СТ СЭВ 1626-79)

УДК 002:651.7/.78:006.354

ЕСПД. ВИДЫ ПРОГРАММ И ПРОГРАММНЫХ ДОКУМЕНТОВ.

ВЫПИСКА

### 1. Виды программ.

1.1 ...

1.2 ...

### 2. Виды программных документов.

2.1. К программным относят документы, содержащие сведения, необходимые для разработки, изготовления, сопровождения и эксплуатации программ.

#### 2.2. Виды программных документов и их содержание

Вид программного документа	Содержание программного документа
Спецификация	Состав программы и документации на нее
Ведомость держателей подлинников	Перечень предприятий, на которых хранят подлинники программных документов
Текст программы	Запись программы с необходимыми комментариями
Описание программы	Сведения о логической структуре и функционировании программы
Программа и методика испытаний	Требования, подлежащие проверке при испытании программы
Техническое задание	Назначение и область применения программы, технические, технико-экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний
Пояснительная записка	Схема алгоритма, общее описание алгоритма и (или) функционирования программы, а также обоснование принятых технических и технико-экономических решений
Эксплуатационные документы	Сведения для обеспечения функционирования и эксплуатации программы

#### 2.3. Виды эксплуатационных документов и их содержание

ПРАКТИЧЕСКАЯ ЧАСТЬ	Содержание эксплуатационного документа
Вид эксплуатационного документа	
Ведомость эксплуатационных документов	Перечень эксплуатационных документов на программу
Формуляр	Основные характеристики программы, комплектность и сведения об эксплуатации программы
Описание применения	Сведения о назначении программы, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств
Руководство системного программиста	Сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения
Руководство программиста	Сведения для эксплуатации программы
Руководство оператора	Сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программы
Описание языка	Описание синтаксиса и семантики языка
Руководство по техническому обслуживанию	Сведения для применения тестовых и диагностических программ при обслуживании технических средств

2.4. В зависимости от способа выполнения и характера применения программные документы подразделяются на подлинник, дубликат и корию (ГОСТ 2.102-68), предназначенные для разработки, сопровождения и эксплуатации программы.

2.5. Виды программных документов, разрабатываемых на разных стадиях и их коды

Код вида Документа	Стадии разработки				
	Вид док-та	Эскизный проект	Технический компонент	Рабочий проект	
		проект	компонент	комплекс	
--	Спецификация	--	--	+	++
05	Ведомость держателей подлинников	--	--	--	+-
12	Текст программы	--	--	++	+-
13	Описание прогр-мы	--	--	+-	+-
20	Ведомость эксплуата-	--	--	+-	+-

ПРАКТИЧЕСКАЯ ЧАСТЬ

	ционных документов				
30	Формуляр	--	--	+-	+-
31	Описание применения	--	--	+-	+-
32	Руководство системного программиста	--	--	+-	+-
33	Руководство программиста	--	--	+-	+-
34	Руководство оператора	--	--	+-	+-
35	Описание языка	--	--	+-	+-
46	Руководство по техническому обслуживанию	--	--	+-	+-
51	Программа и методика испытаний	--	--	+	+-
81	Пояснительная записка	+-	+-	--	--
90-99	Прочие документы	+-	+-	+-	+-

Условные обозначения:

++ - документ обязательный;

+ - документ обязательный для компонентов, имеющих самостоятельное применение;

+- - необходимость составления документа определяется на этапе разработки и утверждения технического задания;

-- - документ не составляют.

2.6. Допускается объединять отдельные виды эксплуатационных документов (за исключением ведомости эксплуатационных документов и формуляра).

Необходимость объединения этих документов указывается в техническом задании. Объединенному документу присваивают наименование и обозначение одного из объединяемых документов.

В объединенных документах должны быть приведены сведения, которые необходимо включать в каждый объединяемый документ.

2.7. На этапе разработки и утверждения технического задания определяют необходимость составления технических условий, содержащих требования к изготовлению, контролю и приемке программы.

Технические условия разрабатывают на стадии "Рабочий проект".

2.8. Необходимость составления технического задания на компоненты, не предназначенные для самостоятельного применения, и комплексы, входящие в другие комплексы, определяются по согласованию с заказчиком.

## **8. Принципы Юзабилити**

### **Research-Based Web Design & Usability Guidelines**

- 5.1 Доступ к главной странице
  - 5.2 Отображение всех главных функций на главной странице
  - 5.3 Позитивное первое впечатление о сайте
  - 5.4 Сообщайте о важности сайта и его цели
  - 5.5 Ограничьте несодержательный текст на главной странице сайта
  - 5.6 Гарантируйте, чтобы главная страница была похожа как главная
  - 5.7 Ограничьте длину главной страницы
  - 5.8 Объявляйте изменения сайта
  - 5.9 Уделяйте внимание ширине панели главной страницы
- 
- 6.1 Избегайте хаотичности отображения
  - 6.2 Располагайте важные элементы единообразно
  - 6.3 Располагайте важные элементы в центре верхней части страницы
  - 6.4 Структурирование для легкого сравнения
  - 6.5 Установите уровень значимости
  - 6.6 Оптимизируйте плотность отображения
  - 6.7 Выравнивайте элементы на странице
  - 6.8 Используйте гибкие расположения элементов
  - 6.9 Избегайте остановок при прокрутке
  - 6.10 Установите подходящие размеры страницы
  - 6.11 Используйте умеренное количества пустого пространства
  - 6.12 Выбирайте подходящую длину строк
  - 6.13 Используйте фреймы, когда функции должны оставаться доступными
- 
- 7.1 Обеспечьте навигационные опции
  - 7.2 Различайте и группируйте элементы навигации
  - 7.3 Используйте активируемый щелчком мыши «Список содержания» на длинных страницах
  - 7.4 Обеспечьте обратную связь с пользователем
  - 7.5 Располагайте основное навигационное меню на левой панели
  - 7.6 Используйте наглядные закладки лейбла
  - 7.7 Эффективное представление закладок
  - 7.8 Обеспечивайте навигацией только небольшие страницы
  - 7.9 Используйте подходящие типы меню
  - 7.10 Используйте карты сайта
  - 7.11 Используйте пояснения для облегчения навигации
  - 7.12 Иерархия страниц для навигации

## ПРАКТИЧЕСКАЯ ЧАСТЬ

- 8.1 Отсутствие горизонтального скrola
- 8.2 Обеспечьте быстрый скrol для чтения
- 8.3 Использование страниц со скrolом для чтения
- 8.4 Использование нумерации страниц вместо скrola
- 8.5 Скрол хуже заполненных страниц
  
- 9.1 Используйте ясные названия категорий
- 9.2 Давайте страницам осмысленные заголовки
- 9.3 Используйте информативные заголовки
- 9.4 Используйте уникальные и информативные заголовки
- 9.5 Выделяйте («подсвечивайте») важную информацию
- 9.6 Используйте информативные заголовки строк и столбцов
- 9.7 Используйте заголовки в соответствующем HTML-порядке
- 9.8 Предоставьте пользователям возможности сокращения выбора
  
- 10.1 Используйте выразительные заголовки для ссылок
- 10.2 Ссылки на связанную информацию
- 10.3 Называйте ссылки в соответствии с названием страницы назначения
- 10.4 Избегайте вводящих в заблуждение меток
- 10.5 Повторение важных ссылок
- 10.6 Использование текстовых ссылок
- 10.7 Пометка посещенных ссылок
- 10.8 Предоставление подходящих меток, обозначающих возможность нажатия
- 10.9 Ссылки в тексте понятны
- 10.10 Использование «наведение и нажатие»
- 10.11 Использование подходящих по длине текстовых ссылок
- 10.12 Различия между внутренними и внешними ссылками
- 10.13 Четкое обозначение границ нажимаемых полей картинки
- 10.14 Ссылки на вспомогательную информацию
  
- 11.1 Использование черного текста, высоко-контрастные фоны
- 11.2 Форматируйте схожих объектов одинаково
- 11.3 Используйте и прописные, и строчные буквы в тексте
- 11.4 Сохраняйте визуальное постоянство
- 11.5 Используйте жирный шрифт экономно
- 11.6 Используйте привлекающие внимание эффекты
- 11.7 Используйте знакомые шрифты
- 11.8 Используйте как минимум 12 размер шрифта
- 11.9 Выделение цветом
- 11.10 Выделение главного

## ПРАКТИЧЕСКАЯ ЧАСТЬ

### 11.11 Выделение (подсветка) информации

- 12.1 Расположите элементы так, чтобы максимально увеличить эффективность использования
  - 12.2 Размещайте важные элементы в начале списка
  - 12.3 Отформатируйте списки для более удобного просмотра
  - 12.4 Отображайте сходные элементы в списке
  - 12.5 Озаглавьте каждый список
  - 12.6 Используйте статическое меню
  - 12.7 Начинайте нумерацию с единицы
  - 12.8 Выбирайте соответствующий тип списка
  - 12.9 Первое слово в списке пишите с заглавной буквы
- 
- 13.1 Помечайте необходимые и необязательные поля ввода данных
  - 13.2 Понятно помечайте кнопки
  - 13.3 Согласованно помечайте поля ввода данных
  - 13.4 Не делайте вводимые пользователем символы чувствительными к регистру
  - 13.5 Понятно помечайте поля ввода данных
  - 13.6 Сводите к минимуму ввод данных пользователем
  - 13.7 Помещайте метки вблизи полей ввода данных
  - 13.8 Позвольте пользователям видеть введенную ими информацию
  - 13.9 Используйте селективные кнопки для выбора взаимно исключающих элементов
  - 13.10 Используйте привычные интерфейсные элементы
  - 13.11 Предупреждение о типичных ошибках пользователя
  - 13.12 Разделяйте длинные элементы данных
  - 13.13 Используйте метод единого ввода данных
  - 13.14 Реализованы приоритеты кнопок
  - 13.15 Использование кнопок-флажков для множественного выбора
  - 13.16 Подписаны единицы измерения
  - 13.17 Не ограничена видимость опций выпадающего списка
  - 13.18 Отображаются значения по умолчанию
  - 13.19 Расположение курсора в самом первом поле ввода
  - 13.20 Двойной щелчок не причина проблем
  - 13.21 Использование открытых списков для выбора одного из многих
  - 13.22 Использование полей ввода данных для ускорения быстродействия
  - 13.23 Использование как минимум двух радио кнопок-переключателей

## ПРАКТИЧЕСКАЯ ЧАСТЬ

13.24 Предусмотрена автоматическая табуляция

13.25 Минимизировано использование клавиши Shift

14.1 Использование простых фоновых картинок

14.2 Картинки-ссылки помечены

14.3 Картинки не загружаются долго

14.4 Использование видео, анимации, и аудио

14.5 Использование логотипа

14.6 Графика не должна выглядеть как баннер

14.7 Большие изображения ограничены

14.8 Проследи за пояснительными сообщениями к картинкам на веб-сайте

14.9 Ограничено использование изображений

14.10 Использование диаграмм с актуальными данными

14.11 Графическое отображение информации для мониторинга

14.12 Введение к анимации

14.13 Эмуляция объектов реального мира

14.14 Использование уменьшенных изображений в качестве предпросмотра для больших изображений

14.15 Использование изображений для обучения

14.16 Использование фотографий людей

15.1 Делайте понятный порядок действий

15.2 Избегайте жаргонизмов

15.3 Используйте знакомые слова

15.4 Определяйте акронимы и аббревиатуры

15.5 Используйте аббревиатуры разумно

15.6 Используйте смешанный регистр в непрерывном тексте

15.7 Ограничивайте число слов и предложений

15.8 Ограничивайте использование текста на страницах с навигацией

15.9 Используйте активный залог

15.10 Пишите инструкции в утвердительной форме

15.11 Делайте первые предложения исчерпывающими

ПРАКТИЧЕСКАЯ ЧАСТЬ  
**Общие принципы**

**Простота и естественность** для пользователя, оперирование его понятиями и представлениями.

**Обеспечение обратной связи.** При клике на ссылку или кнопку пользователь должен увидеть ожидаемую реакцию. Когда этой реакции нет (например, ссылка на главной странице на главную страницу) возникает раздражение за бесполезное действие;

**Единообразие.** Элементы управления (меню) должны всегда находиться в одном месте, и иметь идентичный дизайн на всех страницах сайта.

**Принцип распознавания.** Пользователь должен иметь возможность использовать введенные им сведения. Например, при регистрации, если пользователь ошибся при переходе на новую страницу, согласитесь, что не корректно просить пользователя повторно вводить данные.

**Устойчивость** (терпимость) к ошибкам.

**Принципы проектирования интерфейсов пользователя**

Принцип	Описание
Учет знаний пользователей	В интерфейсе необходимо использовать термины и понятия, взятые из опыта будущих пользователей системы
Согласованность	Интерфейс должен быть согласованным в том смысле, что однотипные операции должны выполняться одним и тем же способом
Минимум неожиданностей	Поведение системы должно быть прогнозируемым
Способность к восстановлению	Интерфейс должен иметь средства, позволяющие пользователям восстановить данные после ошибочных действий
Руководство пользователя	Интерфейс должен предоставлять необходимую информацию в случае ошибок пользователя и поддерживать средства контекстно-зависимой справки

## ПРАКТИЧЕСКАЯ ЧАСТЬ

<b>Учет разнородности пользователей</b>	В интерфейсе должны быть средства для удобного взаимодействия с пользователями, имеющими различный уровень квалификации и различные возможности
---	---

### **Принципы Usability (Нильсен, 2001)**

1. Видимость статуса системы
2. Связь между системой и реальным миром
3. Управление и свобода пользователя
4. Согласованность и стандарты
5. Помощь пользователю в обнаружении, диагностике и исправл. ошибки
6. Предотвращение ошибки
7. Узнавание, а не воспоминание
8. Гибкость и эффективность использования
9. Эстетический и минимальный дизайн
10. Помощь и документация

### **8 золотых правил дизайна интерфейса**

1. Стремитесь к согласованности. Однаковая последовательность действий для аналогичных ситуаций. Идентичная терминология(подсказка, меню, приглашение). Последовательный визуальный формат (шрифты, цвет и д.р.).
2. Обеспечьте универсальность использования
3. Предлагайте информационную обратную связь. Для каждого действия пользователя система должна предоставлять обратную связь. Важность задачи влияет на тип обратной связи
4. «Проект общается, чтобы давать закрытие». Проект спрашивает подтверждение перед выполнением действия
5. Предотвращение ошибок. Уменьшайте количество ошибок, которые может совершить пользователь
6. Простая отмена действий
7. «Поддержите внутреннее местоположение управления». Элементы управления всегда на 1ом месте. Встроенная автоматизация действий пользователя.
8. Уменьшите краткосрочную нагрузку памяти. Люди помнят 7+- 2 фрагмента информации. Делайте простые интерфейсы, Многостраничные окна должны быть похожи.

### **Семь стадий**

1. Формирование цели
2. Формирование задач
3. Определение действия
4. Выполнение действия

## ПРАКТИЧЕСКАЯ ЧАСТЬ

5. Восприятие состояния системы
6. Интерпретация состояния системы
7. Оценка результата

Модель Нормана концентрируется на взгляде пользователя на интерфейс

