

7. Организация взаимодействия процессов и средства их синхронизации. Классические задачи синхронизации.

Процесс — это совокупность машинных команд и данных, обрабатываемая в рамках вычислительной системы и обладающая правами на владение некоторым набором ресурсов ВС.

определения процесса (Рис. 81). Процесс (или **полновесный процесс**) — является объектом планирования и выполняется внутри защищённой области памяти. Альтернативой являются т.н. **легковесные процессы**, известные также как **нити** (или **потоки**), — это процессы, которые могут активироваться внутри полновесного процесса, могут быть объектами планирования, и при этом они могут функционировать внутри общей (т.е. незащищённой от других нитей) области памяти. Обратим внимание, что мы говорим о легковесных процессах, как о процессах другого иерархического уровня, нежели полновесных (т.е. легковесные процессы реализуются внутри полновесного процесса). Причём нити, которые функционируют внутри *одного* полновесного процесса, работают в *едином* адресном пространстве, и они незащищены друг от друга. При этом планированию подвергается каждая нить, т.е. планировщик может осуществлять переключение с нити на нить.

П.И.Л.М.И.

Тогда определение процесса можно обобщить — понятие «**процесса**» включает в себя следующее:

- исполняемый код;
- собственное адресное пространство, представляющее собой множество виртуальных адресов, которые может использовать процесс;
- ресурсы системы, которые назначены процессу операционной системой;
- хотя бы одну выполняемую нить.

Рассмотрим теперь, какая поддержка операционной системы необходима для функционирования процессов. Здесь в первую очередь следует обратить внимание на понятие **контекста процесса**. Под **контекстом процесса** будем понимать совокупность данных, характеризующих актуальное состояние процесса. Обычно контекст процесса состоит из трёх компонент:

- **пользовательская составляющая** — это текущее состояние программы (т.е. совокупность машинных команд и данных, размещённых в ОЗУ и характеризующих выполнение данного процесса);
- **аппаратная составляющая** — отражает актуальное состояние центрального процессора в момент выполнения данного процесса (т.е. это актуальное состояние регистров, настроек процессора и т.д.);
- **системная составляющая** — это структуры данных операционной системы, содержащие характеристики процесса. Эти структуры данных содержат информацию идентификационного характера (PID процесса, PID «родителя» и т.д.); информацию о содержимом регистров (РОН, индексные регистры, флаги и т.д.); а также информацию, необходимую для управления процессом (состояние процесса, приоритет и т.д.). Отметим, что системная составляющая процесса содержит копию аппаратной составляющей, если процесс остановлен.

С точки зрения понимания термина **процесса** в ОС Unix, данное понятие можно определить двояко. С одной стороны, **процесс** можно определить как объект, зарегистрированный в таблице процессов операционной системы. Таблица процессов — одна из специальных системных таблиц, которая является программной таблицей. Второе определение объявляет **процессом** объект, порожденный системным вызовом *fork()*. Оба определения являются корректными и равносильными (если учесть, что в системе существуют два особых процесса с нулевым и первым номерами, и об их особенностях речь пойдет ниже).

Одной из важных проблем, которые появились в современных операционных системах, является проблема взаимодействия процессов.

Будем говорить, что процессы называются **параллельными**, если их выполнение хотя бы частично перекрывается по времени. Т.е. можно говорить, что все процессы, находящиеся в буфере обрабатываемых процессов, являются параллельными, т.к. в той или иной степени времена их выполнения перекрываются друг с другом. Не стоит забывать, что, говоря о параллельных процессах, речь идет лишь о **псевдопараллелизме**, поскольку реально на процессоре может исполняться только один процесс.

Параллельные процессы могут быть независимыми и взаимодействующими. **Независимые процессы** используют независимые множества ресурсов; т.е. множества ресурсов, которые принадлежат независимым процессам, в пересечении дают пустое множество. Альтернативой независимым процессам являются взаимодействующие процессы. **Взаимодействующие процессы** совместно используют ресурсы, и выполнение одного процесса может оказывать влияние на результат другого процесса, участвующего в этом взаимодействии.

Совместное использование ресурса ВС двумя и более параллельными процессами, когда каждый из процессов некоторое время владеет этим ресурсом, называется **разделением ресурса**. Разделению подлежат как аппаратные, так и программные (виртуальные) ресурсы. Разделяемый ресурс, который в каждый момент времени может быть доступен только одному из взаимодействующих процессов, называется **критическим ресурсом**. (Таковыми ресурсами могут быть как внешнее устройство, так и некая переменная, значение которой может изменяться разными процессами.) Соответственно, часть программы (фактически набор операций), в рамках которой осуществляется работа с критическим ресурсом, называется **критической секцией (или критическим интервалом)**.

При организации многопроцессного взаимодействия и использования разделяемых ресурсов, возникает целый ряд новых проблем, по сравнению с однопроцессным программированием. Главным образом, эти проблемы связаны с обеспечением корректного доступа к разделяемым ресурсам. Выделяют две важнейшие задачи ОС: распределение ресурсов между процессами и организация корректного доступа (т.е. организация защиты ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов). В частности, под организацией корректного доступа может пониматься требование, декларирующее необходимость обеспечения независимости работы параллельных процессов от порядка и интенсивности доступа этих процессов к разделяемым ресурсам.

Независимости работы параллельных процессов нарушается. Такие ситуация, когда

процессы конкурируют за разделяемый ресурс, называются **гонкой процессов (race conditions)**.

Единственный способ избежать гонок при использовании разделяемых ресурсов — контролировать доступ к любым разделяемым ресурсам в системе. При этом необходимо организовать **взаимное исключение** — т.е. такой способ работы с разделяемым ресурсом, при котором постулируется, что в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ. Проблему организации взаимного исключения можно сформулировать в более общем виде: задача взаимного исключения сводится к тому, чтобы не допускать ситуации, когда два процесса одновременно находятся в критических секциях, связанных с одним и тем же ресурсом.

Для организации модели взаимного исключения используются различные модели синхронизации. Прежде чем рассматривать эти модели, необходимо отметить те проблемы, которые могут возникать при организации взаимного исключения, — это **блокировки** и **тупики**.

Блокировка — это ситуация, когда доступ к разделяемому ресурсу одного из взаимодействующих процессов не обеспечивается из-за активности других, более приоритетных процессов. Отметим следующее. Рассмотрим основанную на приоритетах модель доступа к разделяемому ресурсу, когда более приоритетный запрос на обращение к ресурсу будет обработан быстрее, чем менее приоритетный. И пусть в этой модели работают два процесса, у которых приоритеты доступа к критическому ресурсу разные. Тогда, если более приоритетный процесс будет «часто» выдавать запросы на обращение к ресурсу, может возникнуть ситуация, когда второй процесс будет «вечно» (или достаточно долго) ожидать обработки каждого своего запроса, т.е. этот менее приоритетный процесс будет блокирован.

Тупик, или *deadlock*, — это ситуация, когда (из-за некорректной организации доступа и разделения ресурсов) конкурирующие за критический ресурс процессы вступают в клинч — происходит взаимоблокировка. Рассмотрим *пример тупиковой ситуации* (Рис. 92).

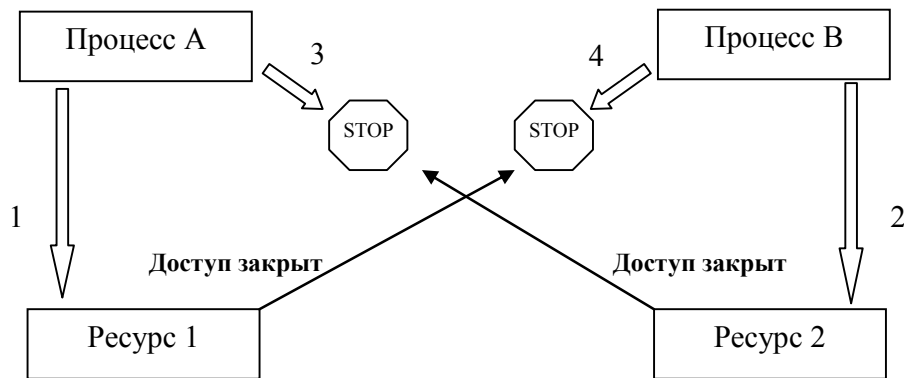


Рис. 92. Пример тупиковой ситуации (deadlock).

Предположим, что есть два процесса **А** и **В**, а также пара критических ресурсов. Пусть в некоторый момент времени процесс **А** вошел в критическую секцию работы с ресурсом 1. Это означает, что доступ любого другого процесса к данному ресурсу будет блокирован (пока процесс **А** не закончит с ним работать). Пусть также в это время процесс **В** войдет в критическую секцию ресурса 2. И этот ресурс также будет блокирован для доступа другим процессам (пока процесс **В** не закончит с ним работать). Пусть процесс **А**, не выходя из критической секции ресурса 1, пытается захватить ресурс 2. Но последний уже захвачен процессом **В**, и процесс **А** блокируется. Аналогично, пусть процесс **В**, не освобождая ресурс 2, пытается захватить ресурс 1 и также блокируется. Это пример простейшего тупика. В этой ситуации каждый из процессов ожидает освобождения недостающего ресурса, но оба ресурса никогда не будут освобождены, и процессы никогда не смогут выполнить необходимые действия. Соответственно, решением в данном случае может быть перезапуск системы или уничтожение обоих или одного из процессов.

В этом разделе речь пойдет о средствах, позволяющих организовать работу с критическими ресурсами, т.е. организовать такой способ работы с разделяемым ресурсом, при котором в любой момент времени к нему может иметь доступ не более одного процесса. В настоящий момент известно множество средств организации взаимного исключения, среди которых мы рассмотрим **семафоры Дейкстры**, **мониторы Хоара** и **аппарат передачи сообщений**.

Семафоры Дейкстры — это формальная модель организации доступа, предложенная в середине 60-х гг. голландским ученым Дейкстрой. Эта модель базируется на следующей концепции. Имеется специальный тип данных — т.н. **семафор**. Переменные типа **семафор** могут принимать целочисленные значения. Над этими переменными определены следующие атомарные (неделимые) операции: опустить семафор **down(S)** (или **P(S)**) и поднять семафор **up(S)** (или **V(S)**). Оригинальные обозначения **P** и **V**, данные Дейкстрой и получившие широкое распространение в литературе, являются сокращениями голландских слов *proberen* — проверить и *verhogen* — увеличить.

Операция **down(S)** проверяет значение семафора **S** и, если оно больше нуля, то уменьшает его на **1**. Если же это не так, процесс блокируется, причем связанная с заблокированным процессом операция **down** считается незавершенной.

Операция **up(S)** увеличивает значение семафора на **1**. При этом если в системе присутствуют процессы, заблокированные ранее при выполнении **down** на этом семафоре, то один из них разблокируется и завершает выполнение операции **down**, т.е. вновь уменьшает значение семафора. Увеличение значения семафора и, возможно,

разблокирование одного из процессов и уменьшение значения являются атомарной неделимой операцией. Выбор процесса для разблокирования никак не оговаривается.

Ещё раз отметим, что операции **up** и **down** являются атомарными (неделимыми), т.е. их выполнение не может быть прервано прерыванием.

Для иллюстрации рассмотренного механизма приведем следующий пример. Рассмотрим некий универсам. Вход в торговый зал магазина возможен лишь для посетителей, имеющих тележку. В магазине имеется **N** тележек. Итак, в начальный момент (когда магазин открывается) имеется **N** свободных тележек. Каждый очередной посетитель берет тележку и проходит в зал. Так продолжается, пока не появится (**N+1**)-ый посетитель, которому тележки уже не хватает. Он войти не может и ждет свободной тележки перед входом в торговый зал. Если приходят еще покупатели, то они также ожидают свободной тележки. Поскольку рассматриваемый формализм, как упоминалось выше, ничего не говорит о выборе очередного заблокированного процесса, то будем считать, что прибывающие в магазин покупатели не становятся в очередь, а стоят в некоем «беспорядке» (толпой). Как только один из покупателей с тележкой покидает торговый зал, происходит операция **up** — появляется одна свободная тележка. Эту тележку берет один из ожидающих посетителей и проходит в торговый зал. Это означает, что один из заблокированных клиентов разблокировался и продолжил работу, остальные же продолжают ждать в заблокированном состоянии.

Если тележка была бы одна, то это было бы иллюстрацией организации доступа режиме взаимного исключения, т.е. в любой момент времени в торговом зале может оказаться лишь один покупатель. Это пример т.н. **двоичного семафора** — семафор максимальное значение которого равно **1**. Этот тип семафоров и обеспечивает взаимное исключение.

системой программирования (компилятором). Монитор — это специализированный модуль, включающий в себя совокупность процедур и функций, а также структуры данные, с которыми работают эти процедуры и функции. При этом данный модуль обладает следующими свойствами:

1. структуры данных монитора доступны только через обращения к процедурам или функциям этого монитора (т.е. монитор представляет собой некоторый аналог объекта в объектно-ориентированных языках и реализует инкапсуляцию данных);
2. считается, что процесс занимает (или входит в) монитор, если он вызывает одну из процедур или функций монитора;
3. в каждый момент времени внутри монитора может находиться не более одного процесса.

Если процесс пытается попасть в монитор, в котором уже находится другой процесс, то (в зависимости от используемой стратегии) он либо получает отказ, либо блокируется, становясь в очередь. Таким образом, чтобы защитить разделяемые структуры данных, их достаточно поместить внутрь монитора вместе с процедурами, представляющими критические секции для обработки этих структур данных.

Монитор представляет собой конструкцию языка программирования и компилятору известно о том, что входящие в него процедуры и данные имеют особую семантику, поэтому первое условие может проверяться еще на этапе компиляции, кроме того, код для процедур монитора тоже может генерироваться особым образом, чтобы удовлетворялось третье условие. Поскольку организация взаимного исключения в данном случае возлагается на компилятор, количество программных ошибок, связанных с организацией взаимного исключения, сводится к минимуму.

Бытовой иллюстрацией монитора может служить cabina таксофонного аппарата.

Повторим, что монитор — это языковая конструкция с централизованным управлением (в отличие от семафоров, которые не обладают централизацией). Семафоры и мониторы являются, главным образом, средствами организации работы в однопроцессорных системах либо в многопроцессорных системах с общей памятью. Для многопроцессорных систем с распределенной памятью эти средства не очень подходят. Для них в настоящий момент наиболее часто используется механизм *передачи сообщений*.

Сразу отметим, что механизм передачи сообщений является универсальным в том смысле, что он предоставляет, как средства организации взаимодействия между процессами, так и средства синхронизации. Механизм *передачи сообщений* основан на двух функциональных примитивах: **send** (отправить сообщение) и **receive** (принять сообщение). Данные операции можно разделить по трем характеристикам: модель синхронизации, адресация и формат сообщения.

Синхронизация. Операции отправки/приема сообщений могут быть *блокирующими* и *неблокирующими*. Рассмотрим различные комбинации.

Блокирующий send: процесс-отправитель будет заблокирован до тех пор, пока посланное им сообщение не будет получено.

Блокирующий receive: процесс-получатель будет заблокирован до тех пор, пока не будет получено соответствующее сообщение.

Соответственно, неблокирующие операции, как следует из названия, происходят без блокировок.

Итак, комбинируя различные операции **send** и **receive**, мы получаем 4 различных модели синхронизации.

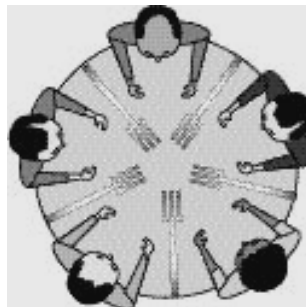
Адресация может быть *прямой* или *косвенной*. При прямой адресации указывается конкретный адрес получателя и/или отправителя (например, когда получатель ожидает сообщения от конкретного отправителя, игнорируя сообщения других отправителей).

В случае косвенной адресации не указывается адрес конкретного получателя при отправке или адрес конкретного отправителя при получении; сообщение «бросается» в некоторый общий пул, в котором могут быть реализованы различные стратегии доступа (FIFO, LIFO и т.д.). Этим пулом может выступать очередь сообщений (FIFO) или почтовый ящик, в котором может быть реализована любая модель доступа. Бытовым примером косвенной адресации может служить модель обслуживающей системы (сотруднику ателье всё равно, чей пиджак он чистит).

Итак, повторимся, что механизм передачи сообщений совмещает два средства: средство передачи информации и средство синхронизации. Этот аппарат является базовым средством организации взаимодействия процессов в многопроцессорных системах с распределенной памятью.

Механизм передачи сообщений реализуется на базе интерфейсов передачи сообщений MPI. На основе этих интерфейсов строятся почти все кластерные системы (т.е. системы с распределенной памятью), а также MPI может работать и в системах с общей памятью.

Обедающие философы (Рис. 94). Пусть существует круглый стол, за которым сидит группа философов: они пришли пообщаться и покушать. Кушают они спагетти, которое находится в общей миске, стоящей в центре стола. Для приема пищи они пользуются двумя вилками: одна в левой руке, другая — в правой. Вилки располагаются по одной между каждыми двумя философами. Каждый из философов некоторое время размышляет, затем берет две вилки и ест спагетти, затем кладёт вилки на стол и опять размышляет, и так далее. Каждый из них ведет себя независимо от других. Философы должны совместно использовать имеющиеся у них вилки (ресурсы). Задача состоит в организации доступа к вилкам.



```
#define N 5 /* количество философов */
#define LEFT (i-1)%N /* номер левого соседа для i-го философа */
#define RIGHT (i+1)%N /*номер левого соседа для i-го философа */
/* состояния философов:
«думает»,
«голоден»,
«кушает»
*/
#define THINKING 0
#define HUNGRY 1
#define EATING 2

/* определяем тип СЕМАФОР */
typedef int semaphore;
/*
массив состояний каждого из философов, инициализированный нулями
*/
int state[N];
/* семафор для доступа в критическую секцию */
semaphore mutex = 1;
/*
массив семафоров по одному на каждого из философов,
инициализированный нулями
*/
semaphore s[N];

/* Процесс-философ (i = 0..N) */
void Philosopher(int i)
```

```

{
    while(TRUE)
    {
        Think();
        /* философ берёт обе вилки или блокируется */
        TakeForks(i);
        Eat();
        PutForks(i);
    }
}

/* получение вилок */
void TakeForks(int i)
{
    /* вход в критическую секцию */
    down(&mutex);
    state[i] = HUNGRY;
    Test(i);
    /* выход из критической секции */
    up(&mutex);
    down(&s[i]);
}

/* освобождение вилок */
void PutForks(int i)
{
    /* вход в критическую секцию */
    down(&mutex);
    state[i] = THINKING;
    Test(LEFT);
    Test(RIGHT);
    /* выход из критической секции */
    up(&mutex);
}

/* функция проверки возможности получения вилок -
проверяется состояние соседей данного философа */
void Test(int i)
{
    if(state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING)
    {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

В этом решении каждый философ живет по аналогичному циклическому распорядку: размышляет некоторое время, затем берет вилки, кушает, кладет вилки. Рассмотрим процедуру получения вилок (TakeForks). Опускается семафор mutex, который используется для синхронизации входа в критическую секцию. Внутри критической секции меняем состояние философа (помечаем его состоянием HUNGRY). Затем предпринимается попытка начать есть (вызывается функция Test). Функция Test

проверяет, что если i -ый философ голоден, а его соседи в данный момент не едят (т.е. правая и левая вилки свободны), то этот философ начинает прием пищи (состояние EATING), а его семафор поднимается (заметим, что изначально этот семафор инициализирован нулем). После этого мы возвращаемся обратно в функцию TakeForks, в которой далее происходит выход из критической секции (поднимаем семафор mutex), а затем опускаем семафор этого философа. Если внутри функции Test философу удалось начать прием пищи, то семафор поднят, и операция down обнулит его, не блокируясь. Если же функция Test не изменит состояние философа, а также не поднимет его семафор, то операция down (в функции TakeForks) в этой точке заблокируется до тех пор, пока оба соседа не освободят вилки.

Внутри функции освобождения вилок PutForks сначала происходит опускание семафора mutex, происходит вход в критическую секцию. Затем меняется статус философа (на статус THINKING), после чего проверяем его соседей: если любой из них был заблокирован лишь из-за того, что наш i -ый философ забрал его вилку, то мы его разблокируем, и он начинает прием пищи. После этого происходит выход из критической секции путем подъема семафора mutex.

Заметим, что использование механизма *взаимоисключающего нахождения внутри критической секции* (за счет семафора mutex) гарантирует, что не возникнет ситуация, когда два процесса, соответствующие соседним философам, будут так спланированы на обработку на процессоре, что функция Test в каждом из них проработает и разрешит каждому из них начать прием пищи (что, конечно же, является ошибкой). Если же этого механизма не будет, то возможно, что один из процессов-соседей входит в Test, делает проверку на возможность начала приема пищи. Проверка дает истинное значение, управление переходит к первой команде внутри if-блока. После этого происходит смена процесса на процессоре, управление получает сосед этого философа. Тот тоже делает проверку внутри функции Test и также получает положительный ответ, и управление переходит к первой инструкции if-блока. Дальнейшая работа будет некорректной.

Задача «читателей и писателей». Представим произвольную систему резервирования ресурса. Например, это может быть система резервирования места в гостинице. В данной системе существует два типа процессов для работы с информацией. Одни процессы могут читать информацию, а другие — ее изменять, корректировать. Соответственно, возникает все тот же вопрос, как организовать корректную совместную работу этих процессов. Это означает, что в любой момент времени читать данные могут любое количество процессов-читателей, но если процесс-писатель начал свою работу, то все остальные процессы (и читатели, и писатели) будут блокированы на входе в систему. Задача заключается в планировании работы такой системы.

Рассмотрим модельную реализацию данной задачи при выбранной следующей стратегии: будем считать, что наиболее приоритетными являются читающие процессы. То есть процесс-писатель будет ожидать момента, когда все желающие процессы-читатели окончат свои действия в системе и покинут ее.

```
/* переопределение типа семафор */  
typedef int semaphore;  
/* семафор для доступа в критическую секцию - контроль за  
доступом к «rc» (разделяемый ресурс) */  
semaphore mutex = 1;  
/* семафор для доступа к базе данных */  
semaphore db = 1;  
/* количество читателей внутри хранилища */  
int rc = 0;  
  
/* процесс-читатель */
```

```

void Reader(void)
{
    while(true)
    {
        down(&mutex); /* получить эксклюзивный доступ к «rc»*/
        rc = rc + 1; /* еще одним читателем больше */
        /* если это первый читатель, нужно заблокировать
        эксклюзивный доступ к базе */
        if(rc == 1)
            down(&db);
        up(&mutex); /*освободить ресурс rc */
        ReadDataBase(); /* доступ к данным */
        down(&mutex); /*получить эксклюзивный доступ к «rc»*/
        rc = rc - 1; /* теперь одним читателем меньше */
        /*если это был последний читатель, разблокировать
        эксклюзивный доступ к базе данных */
        if(rc == 0)
            up(&db);
        up(&mutex); /*освободить разделяемый ресурс rc*/
        UseDataRead(); /* некритическая секция */
    }
}

/* процесс-писатель */
void Writer(void)
{
    while(TRUE)
    {
        ThinkUpData(); /* некритическая секция */
        down(&db); /* получить эксклюзивный доступ к данным*/
        WriteDataBase(); /* записать данные */
        up(&db); /* отдать эксклюзивный доступ */
    }
}

```

В приведенном решении процесс-читатель в каждом цикле своей работы входит в критическую секцию (опускает семафор `mutex`), увеличивает счетчик читателей, находящихся в хранилище, на 1. Затем проверяет, что является ли он первым читателем (т.е. в данный момент он единственный клиент в хранилище). Если да, то он опускает семафор `db`, тем самым, препятствуя писателям войти в систему, если они того пожелают. Если же семафор `db` уже был опущен, то это означает, что в данный момент в хранилище присутствует писатель, и этот первый читатель блокируется на этой операции, ожидая выхода писателя из системы. (Заметим, что эта блокировка происходит внутри критической секции, поэтому остальные читатели будут блокироваться на опускании семафора `mutex`.) После этого происходит выход из критической секции (подымаем семафор `mutex`), чтение информации из хранилища. Затем производятся обратные действия по выходу из хранилища, которые также происходят внутри критической секции. Итак, на выходе мы уменьшаем число читателей в хранилище, и если некоторый читатель является последним клиентом в библиотеке, то происходит поднятие семафора `db`, разрешая работу писателям (которые к этому моменту могли быть заблокированы на входе). В конце цикла работы читатель обрабатывает полученные данные из хранилища, после чего цикл повторяется.

Писатель в начале каждого цикла своей работы подготавливает данные для сохранения, затем пытается войти в хранилище, опуская семафор db. Если в хранилище кто-то есть, то он будет ожидать, пока последний клиент (независимо от того, читатель это или писатель) не покинет хранилище. После этого он производит корректировку данных в хранилище и покидает его, поднимая семафор db.

Заметим, что в данном решении если хотя бы один читатель находится внутри системы, то любой следующий читатель беспрепятственно в нее попадет, а писатель же будет ожидать, когда все посетители покинут хранилище, т.е. реализована стратегия приоритетности читателя перед писателем. Чтобы этого избежать, можно модифицировать алгоритм таким образом, чтобы в случае, если имеется хотя бы один ожидающий процесс-писатель, новые процессы-читатели не получали доступа к ресурсу, а ожидали, когда процесс-писатель обновит данные. Однако, обратная сторона данного решения в том, что оно несколько снижает производительность процессов-читателей, т.к. вынуждает их ждать в тот момент, когда ресурс не занят в эксклюзивном режиме.

Данная задача иллюстрирует модель доступа к разделяемому ресурсу процессов, имеющих разные приоритеты.

Задача о «спящем парикмахере». Рассмотрим парикмахерскую, в которой работает один парикмахер, имеется одно кресло для стрижки и несколько кресел в приемной для посетителей, ожидающих своей очереди. Если в парикмахерской нет посетителей, парикмахер засыпает прямо на своем рабочем месте. Появившийся посетитель должен его разбудить, в результате чего парикмахер приступает к работе. Если в процессе стрижки появляются новые посетители, они должны либо подождать своей очереди, либо покинуть парикмахерскую, если в приемной нет свободного кресла для ожидания (т.е. это стратегия обслуживания с отказами). Задача состоит в том, чтобы корректно запрограммировать поведение парикмахера и посетителей.

Данная задача является иллюстрацией модели клиент-сервер с ограничением на длину очереди клиентов.

Рассмотрим реализацию данной модели. Понадобится 3 семафора: *customers* — подсчитывает количество посетителей, ожидающих в очереди, *barbers* — статус парикмахера (0 - занят, 1 - свободен)¹ и *mutex* — используется для синхронизации доступа к разделяемой переменной *waiting*. Переменная *waiting*, как и семафор *customers*, содержит количество посетителей, ожидающих в очереди. Эта переменная используется в программе для того, чтобы иметь возможность проверить, имеется ли свободное кресло для ожидания, и при этом не заблокировать процесс, если кресла не окажется. Заметим, что как и в предыдущем примере, эта переменная является разделяемым ресурсом, и доступ к ней охраняется семафором *mutex*.

```
/* количество стульев в комнате ожидания */
#define CHAIRS 5
/* переопределение типа СЕМАФОР */
typedef int semaphore;
/* наличие посетителей, ожидающих парикмахера */
semaphore customers = 0;
/* количество парикмахеров, ожидающих посетителей (0 или 1)*/
semaphore barbers = 0;
/* семафор для доступа в критическую секцию - контроль за
доступом к переменной waiting */
semaphore mutex = 1;
/* количество ожидающих посетителей */
int waiting = 0;
```



```

/* Брадобрей */
void Barber(void)
{
    while(TRUE)
    {
        /* если customers == 0, т.е. посетителей нет, то
        заблокируемся до появления посетителя */
        down(&customers);
        down(&mutex); /* получаем доступ к waiting */
        /* уменьшаем кол-во ожидающих клиентов */
        waiting = waiting - 1;
        up(&barbers); /* парикмахер готов к работе */
        up(&mutex); /* освобождаем ресурс waiting */
        CutHair(); /* процесс стрижки */
    }
}

/* Посетитель */
void Customer(void)
{
    down(&mutex); /* получаем доступ к waiting */
    if(waiting < CHAIRS) /* есть место для ожидания */
    {
        /* увеличиваем кол-во ожидающих клиентов */
        waiting = waiting + 1;
        /* если парикмахер спит, это его разбудит */
        up(&customers);
        up(&mutex); /* освобождаем ресурс waiting */
        /* если парикмахер занят, переходим в состояние
        ожидания, иначе - занимаем парикмахера */
        down(&barbers);
        /* занять место и перейти к стрижке */
        GetHaircut();
    }
    else
    {
        /*нет свободного кресла для ожидания - придется уйти */
        up(&mutex);
    }
}

```

Процесс-парикмахер сначала опускает семафор *customers*, уменьшив тем самым количество ожидающих посетителей на 1. Если в комнате ожидания никого нет, то он «засыпает» в своем кресле, пока не появится клиент, который его разбудит. Затем парикмахер входит в критическую секцию, уменьшает счетчик ожидающих клиентов, поднимает семафор *barbers*, сигнализируя клиенту о своей готовности его обслужить, а потом выходит из критической секции. После описанных действий он начинает стричь волосы посетителю.

Посетитель парикмахерской входит в критическую секцию. Находясь в ней, он проверяет, есть ли свободные места в зале ожидания. Если нет, то он просто уходит (покидает критическую секцию, поднимая семафор *mutex*). Иначе он увеличивает счетчик ожидающих процессов и поднимает семафор *customers*. Если же этот посетитель является

единственным в данный момент клиентом брадобрея, то он этим действием разбудит брадобрея. После этого посетитель выходит из критической секции и «захватывает» брадобрея (опуская семафор *barbers*). Если же этот семафор опущен, то клиент будет дожидаться, когда брадобрей его поднимет, известив тем самым, что готов к работе. В конце клиент обслуживается (*GetHaircut*).

8. Виртуальная память. Модели организации оперативной памяти.

Перейдём теперь к проблеме организации, использования и управления оперативной памятью. Рассмотренные выше проблемы перемещаемости программы по ОЗУ и фрагментации памяти связаны с необходимостью наличия т.н. аппарата **виртуальной памяти**, т.е. аппаратного средства процессора, которое обеспечивает преобразование (установление соответствия) логических адресов, используемых внутри программы, в те адреса физической оперативной памяти, в которой размещается программа во время выполнения.

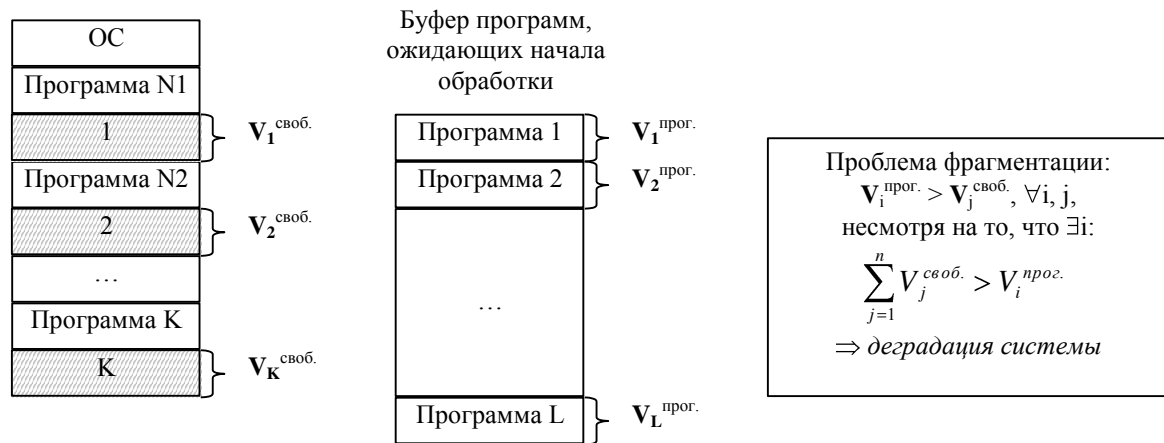


Рис. 48. Фрагментация памяти.

Что понимается под виртуальной памятью и виртуальным адресным пространством? Неформально *виртуальное адресное пространство* можно определить как то адресное пространство, которое используется внутри программ (написанных, например, на языках программирования высокого уровня). Ведь когда программист пишет программу, оперируя теми или иными адресами, он зачастую не задумывается, к каким физическим адресам эти адреса будут привязаны. В исполняемом модуле используется т.н. программная (логическая, виртуальная) адресация. Виртуальные адреса существуют «вне машины». Соответственно, стоит проблема установления соответствия между программной адресацией и физической памятью. И эта проблема решается за счет аппарата виртуальной памяти.

Реализацией одной из моделей аппарата виртуальной памяти является аппарат **базирования адресов**. Механизм базирования адресов основан на двоякой интерпретации (Рис. 55) получаемых в ходе выполнения программы исполнительных адресов ($A_{исп.}^{прог.}$). С одной стороны, его можно интерпретировать как **абсолютный исполнительный адрес**, когда физический адрес в некотором смысле соответствует исполнительному адресу программы ($A_{исп.}^{физ.} = A_{исп.}^{прог.}$). Например, требуется «прочитать ячейку с адресом (абсолютным адресом) 0», или «передать управление по адресу входа в обработчик прерывания». С другой стороны, исполнительный адрес программы можно проинтерпретировать как **относительный адрес**, т.е. адрес, зависящий от места дислокации программы в ОЗУ (адрес относительно точки загрузки программы). Иными словами, имеется оперативная память с ячейками с номерами от 0 до некоторого $L-1$, и, начиная с некоторого адреса K , расположена программа. Тогда адрес $A_{исп.}^{прог.}$ внутри

программы можно трактовать, как отступ от физической ячейки с адресом K на величину $A_{\text{исп.}}^{\text{прог.}}$. Для реализации модели базирования используется специальный *регистр базы*, в который в момент загрузки процесса в оперативную память операционная система записывает начальный адрес загрузки (т.е. K). Тогда реальный физический адрес получается, исходя из формулы $A_{\text{исп.}}^{\text{физ.}} = A_{\text{исп.}}^{\text{прог.}} + \langle R_{\text{базы}} \rangle$.

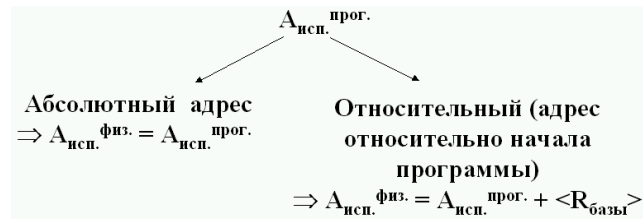


Рис. 55. Базирование адресов – решение проблемы перемещаемости программы по ОЗУ.

Таким образом, базирование адресов – это средство отображения виртуального адресного пространства программы в физическую память «один в один». Аппарат базирования позволяет разрешить проблему перемещаемости программ по ОЗУ, поскольку процесс можно загрузить в любую область памяти. Но при этом необходимо помнить, что программа представляется в виде непрерывной области виртуальной памяти, которая загружается в непрерывный фрагмент физической памяти. Поэтому для решения

проблемы фрагментации аппарата базирования недостаточно, и используются более развитые механизмы организации ОЗУ и виртуальной памяти.

Итак, *виртуальное адресное пространство* — это множество виртуальных страниц, доступных для использования в программе. *Физическое адресное пространство* — это оперативная память, подключенная к данному компьютеру. Физическая память может иметь произвольный размер по отношению к размеру виртуальной памяти (число физических страниц может быть меньше, больше или равно числу виртуальных страниц).

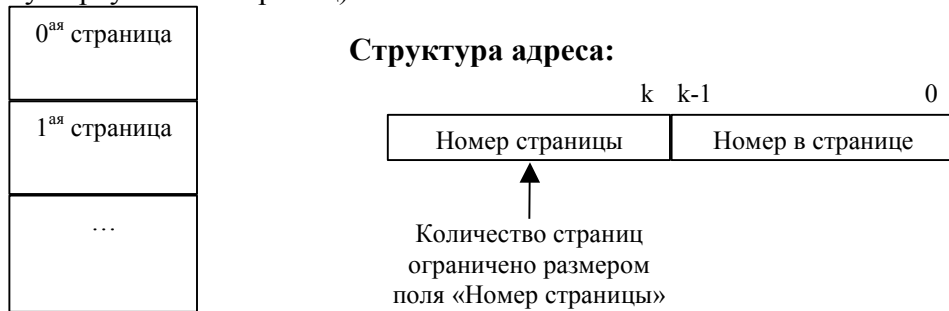


Рис. 56. Страничная организация памяти.

В центральном процессоре имеется аппаратная (регистровая) таблица, называемая **таблицей страниц**, предназначенная для организации отображения между виртуальными и физическими адресами при страничной организации памяти. Количество строк в этой таблице определяется максимальным числом виртуальных страниц, которое зависит от схем работы процессора и максимальной адресной разрядности процессора. Каждой виртуальной странице исполняемой программы ставится в соответствие строка таблицы страниц с тем же номером (нулевой странице соответствует нулевая строка и т.п.). Внутри каждой записи таблицы страниц находится номер физической страницы, в которой размещается соответствующая виртуальная страница программы. Соответственно, аппарат виртуальной страничной памяти позволяет **автоматически** (т.е. **аппаратно**) преобразовывать номер виртуальной страницы в номер физической страницы посредством обращения к таблице страниц (Рис. 57). Программных действий при таком подходе требуется минимально: при выборе операционной системой очередного процесса, который ставится на обработку на центральный процессор, она должна лишь корректно заполнить аппаратную таблицу страниц процессора для данного процесса.

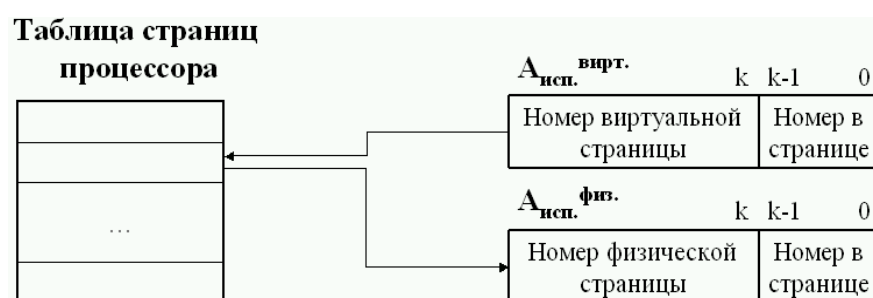


Рис. 57. Страничная организация памяти. Преобразование виртуального адреса в физический.

Но данный подход имеет и свои *недостатки*. Во-первых, это страничная фрагментация, или **внутренняя** (скрытая) **фрагментация**: если в странице используется хотя бы один байт, то вся страница отводится процессу и считается занятой (т.е., решив вопрос с т.н. **внешней фрагментацией**, мы в данном случае не используем память, размером со страницу минус один байт). Во-вторых, описанная выше модель является **вырожденной**: если таблица страниц целиком располагается на регистровой памяти, то в силу дороговизны последней размеры подобной таблицы должны быть слишком малы (а следовательно, будет невелико количество физических страниц). Реальные современные системы имеют более сложную логическую организацию, и речь о ней пойдет ниже. Кроме того, при смене процессов таблицу страниц сначала обязательно надо сохранить, а потом обновить – дополнительные накладные расходы.

Напоследок заметим, что данное нами определение аппарата виртуальной памяти расходится с определениями некоторых других источников. Повторим, что мы рассматриваем механизм виртуальной памяти как механизм преобразования виртуального адресного пространства в физическое. Во многих изданиях, посвященных рассмотрению операционных систем, виртуальной памятью считается то, что позволяет часть программы размещать на внешних устройствах, т.е. считают механизм виртуальной памяти как средство увеличения объема физической памяти. Мы считаем такое определение некорректным. Если рассматривать, например, виртуальную память как механизм увеличения объема памяти, то возникает вопрос: в случае большого объема физической памяти разве виртуальная память отсутствует? Соответственно, возникают проблемы с подобным определением.

И еще одно важное замечание. В компьютере имеется физическое адресное пространство и виртуальное. Физическое пространство — это та оперативная память, которая физически может быть подключена к компьютеру, а виртуальное адресное пространство — это то пространство, которое доступно программе. И возникает вопрос, что и каким способом задает максимальные размеры этих адресных пространств. На размер виртуального адресного пространства влияет разрядность исполнительных адресов, получаемых в ходе обработки программы на центральном процессоре. Размер физического пространства определяется характеристикой компьютера: зависит от того, сколько физически можно подключить памяти к машине, и какова разрядность внутренней аппаратной шины. Но и то, и другое являются аппаратными характеристиками компьютера.

Будем говорить о функциях управления оперативной памятью в контексте решения следующих основных задач. Во-первых, это осуществление контроля использования ресурсов, т.е. учет состояния каждой доступной в системе единицы памяти (свободна она или распределена).

Второй задачей является выбор стратегии распределения памяти. Иными словами, решается задача, какому процессу, в течение какого времени и в каком объеме должен быть выделен соответствующий ресурс. Стратегия распределения памяти является достаточно сложной задачей планирования.

5.1 Одинокое непрерывное распределение

Данная модель распределения оперативной памяти (Рис. 131) является одной из самых простых и основывается на том, что все адресное пространство подразделяется на два компонента. В одной части памяти располагается и функционирует операционная система, а другая часть выделяется для выполнения прикладных процессов.

При таком подходе не возникает особых организационных трудностей. С точки зрения обеспечения корректности функционирования этой модели, необходимо аппаратно обеспечить «водораздел» между пространствами, принадлежащими операционной системе и пользовательским процессом. Для этих целей достаточно иметь один регистр границы: если получаемый исполнительный адрес оказывается меньше значения этого регистра, то это адрес в пространстве операционной системы, иначе — в пространстве процесса. Такая реализация может сочетаться с аппаратной поддержкой двух режимов функционирования: пользовательского режима и режима ОС. Если в режиме пользователя происходит попытка обратиться в область памяти операционной системы, возникает прерывание. Алгоритмы, используемые при таком распределении, достаточно просты, и мы не будем их здесь обсуждать.

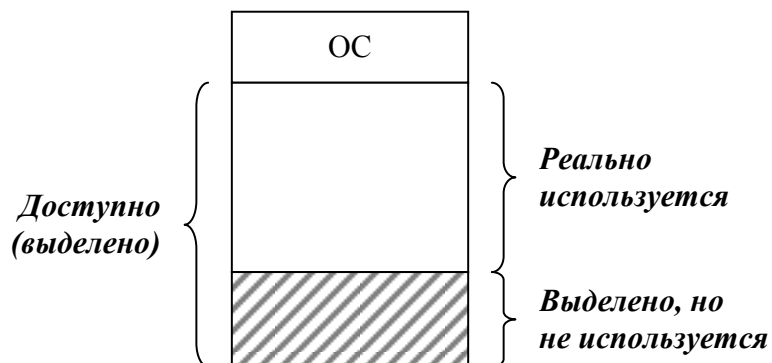


Рис. 131. Одинокое непрерывное распределение.

К достоинствам данной модели относится концептуальная простота во всех отношениях. В частности, минимальные аппаратные требования или отсутствие таковых,

Среди недостатков можно отметить, во-первых, неэффективное использование физического ресурса: часть памяти, выделяемой под процесс, никогда реально не используется. Во-вторых, процесс занимает всю память полностью на все время выполнения. Но реально оказывается, что зачастую обращения процесса к памяти

5.2 Распределение неподвижными разделами

Данная модель строится по следующим принципам (Рис. 132). Опять же, все адресное пространство оперативной памяти делится на две части. Одна часть отводится под операционную систему, все оставшееся пространство отводится под работу прикладных процессов, причем это пространство заблаговременно делится на N частей (назовем их *разделами*), каждая из которых в общем случае имеет произвольный фиксированный размер. Эта настройка происходит на уровне операционной системы. Соответственно, очередь прикладных процессов разделяется по этим разделам.

Существуют концептуально два варианта организации этой очереди. Первый вариант (вариант Б) предполагает наличие единственной сквозной очереди, которая по каким-то соображениям распределяется между этими разделами. Второй вариант (вариант А) организован так, что с каждым разделом ассоциируется своя очередь и поступающий процесс сразу попадает в одну из этих очередей.

Существуют несколько способов аппаратной реализации данной модели. С одной стороны, это *использование двух регистров границ*, один из которых отвечает за начало, а второй — за конец области прикладного процесса. Выход за ту или иную границу ведет к возникновению прерывания по защите памяти.

Альтернативной аппаратной реализацией может служить *механизм ключей защиты* (PSW — process[or] status word), которые могут находиться в слове состояния процесса и в слове состояния процессора. Данное решение подразумевает, что каждому разделу ОЗУ ставится в соответствие некоторый ключ защиты. Если аппаратура поддерживает, то в процессоре имеется слово состояния, в котором может находиться ключ защиты доступного в данный момент раздела. Соответственно, у процесса также есть некоторый ключ защиты, который тоже хранится в некотором регистре. Если при обращении к памяти эти ключи защиты совпадают, то доступ считается разрешенным, иначе возникает прерывание по защите памяти.

модификацией второго решения, основанной на следующем подходе. Для каждого процесса имеется счетчик дискриминации. Под *дискриминацией* будем понимать

ситуацию, когда в системе освободился раздел, достаточный для загрузки некоторого процесса, но система планирования ОС его пропустила. Соответственно, при каждой такой дискриминации из счетчика дискриминации данного процесса вычитается единица. Тогда при просмотре очереди планировщик сначала проверяет значение этого счетчика: если оно равно нулю и процесс помещается в освободившемся разделе, то планировщик обязан загрузить данный процесс в этот раздел.

К достоинствам данной модели распределения оперативной памяти можно отнести простоту аппаратных средств организации мультипрограммирования (например,

К недостаткам можно отнести, во-первых, внутреннюю фрагментацию в разделах, поскольку зачастую процесс, загруженный в раздел, оказывается меньшего размера, чем данный раздел. Во-вторых, это ограничение предельного размера прикладных процессов размером максимального физического раздела ОЗУ. И, в-третьих, опять-таки весь процесс

5.3 Распределение перемещаемыми разделами

Данная модель распределения (Рис. 133) позволяет загрузку произвольного (нефиксированного) числа процессов в оперативную память, и под каждый процесс отводится раздел необходимого размера. Соответственно, система допускает перемещение раздела, а, следовательно, и процесса. Такой подход позволяет избавиться от фрагментации.

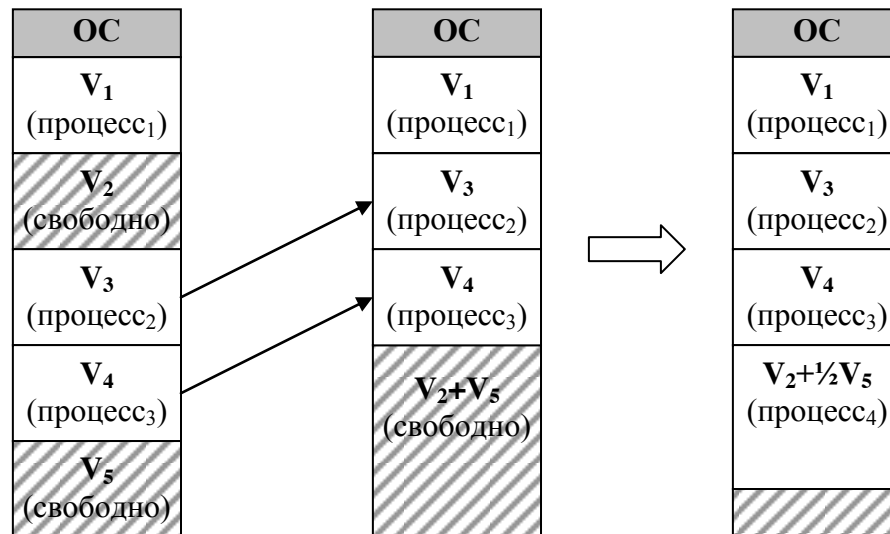


Рис. 133. Распределение перемещаемыми разделами.

По мере функционирования операционной системы после завершения тех или иных процессов пространство оперативной памяти становится все более и более

фрагментированным: в памяти присутствует множество небольших участков свободного пространства, суммарный объем которых позволяет поместить достаточно большой процесс, но каждый из этих участков меньше размера этого процесса. Для борьбы с фрагментацией используется специальный процесс компрессии. Данная модель позволяет использовать компрессию за счет того, что исполняемый код процессов может перемещаться по оперативной памяти.

Что касается аппаратной поддержки, то здесь она аналогична предыдущей модели: требуются аппаратные средства защиты памяти (регистры границ или же ключи защиты) и аппаратные средства, позволяющие осуществлять перемещение процессов (в большинстве случаев для этих целей используется регистр базы, который в некоторых случаях может совпадать с одним из регистров границ). Используемые алгоритмы также

5.4 Страничное распределение

Данная модель основывается на том, что все адресное пространство может быть представлено совокупностью блоков фиксированного размера (Рис. 134), которые называются *страницами*. Есть *виртуальное адресное пространство* — это то пространство, с адресами которого оперирует программа, и *физическое адресное пространство* — это то пространство, которое есть в наличии в компьютере. Соответственно, при страничном распределении памяти существуют программно-аппаратные средства, позволяющие устанавливать соответствие между виртуальными и физическими страницами. Механизм преобразования виртуального адреса в физический обсуждался ранее, он достаточно прост: берется номер виртуальной страницы и

заменяется соответствующим номером физической страницы. Также отмечалось, что для этих целей используется т.н. **таблица страниц**, которая целиком является аппаратной, что на самом деле является большим упрощением. Если рассмотреть современные машины с современным объемом виртуального адресного пространства, то окажется, что эта таблица будет очень большой по размеру. Соответственно, возникает важный вопрос, как осуществлять указанное отображение виртуальных адресов в физические.

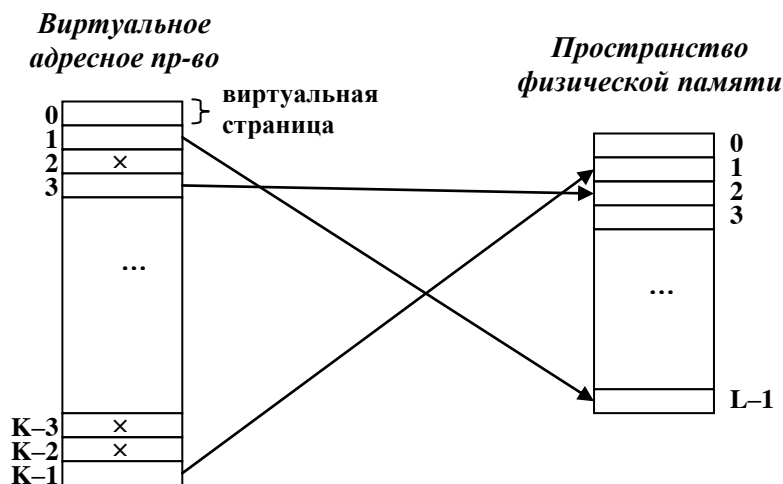


Рис. 134. Страничное распределение.

Ответ на поставленный вопрос, как всегда, неоднозначный и имеет несколько вариантов. Первое решение, приходящее на ум, — это полное размещение таблицы преобразования адресов в аппаратной части компьютера, но это решение применимо лишь в тех системах, где количество страниц незначительное. Примером такой системы может

Альтернативой служит решение, предполагающее хранение данной таблицы в оперативной памяти, тогда каждое преобразование происходит через обращение к ОЗУ, что совсем неэффективно. К аппаратуре предъявляются следующие требования: должен быть регистр, ссылающийся на начало таблицы в ОЗУ, а также должно аппаратно поддерживаться обращение в оперативную память по адресу, хранящемуся в указанном регистре, извлечение данных из таблицы и осуществление преобразования.

Возможно оптимизировать рассмотренный подход за счет использования кэширования L1 или L2. С одной стороны, поскольку к таблице страниц происходит постоянное обращение, странички из данной таблицы «зависают» в КЭШе. Но, если в компьютере используется всего один КЭШ и для потока управления, и для потока данных, то в этом случае через него направляется еще и поток преобразования страниц. Поскольку эти потоки имеют свои особенности, то добавление дополнительного потока со своими индивидуальными характеристиками приведет к снижению эффективности системы.

Типовая структура записи таблицы страниц (Рис. 135) содержит информацию о номере физической страницы, а также совокупность атрибутов, необходимых для описания статуса данной страницы. Среди атрибутов может быть атрибут присутствия/отсутствия страницы, атрибут режима защиты страницы (чтение, запись, выполнение), флаг модификации содержимого страницы, атрибут, характеризующий обращения к данной странице, чтобы иметь возможность определения «старения» страницы, атрибут блокировки кэширования и т.д. Итак, в каждой записи может присутствовать целая совокупность атрибутов, которые аппаратно интерпретируемы: например, при попытке записать данные в страницу, закрытую на запись, произойдет прерывание.

ε	δ	γ	β	α	Номер физической страницы
---------------	----------	----------	---------	----------	---------------------------

Рис. 135. Модельная структура записи таблицы страниц. Здесь: α — присутствие/отсутствие; β — защита (чтение, чтение/запись, выполнение); γ — изменения; δ — обращение (чтение, запись, выполнение); ε — блокировка кэширования.

В качестве одного из первых решений оптимизации работы с памятью стало использование т.н. **TLB-таблиц** (Translation Look-aside Buffer — таблица быстрого преобразования адресов, Рис. 136). Данный метод подразумевает наличие аппаратной таблицы относительно небольшого размера (порядка 8 – 128 записей). Данная таблицы концептуально содержит три столбца: первый столбец — это номер виртуальной страницы, второй — это номер физической страницы, в которой находится указанная виртуальная страница, а третий столбец содержит упомянутые выше атрибуты.

Виртуальный адрес состоит из номера виртуальной страницы (**VP**) и смещения в ней (**offset**). Страница изымает из этого адреса номер виртуальной страницы и осуществляет оптимизированный поиск (т.е. поиск не последовательный, а параллельный) этого номера по TLB-таблице. Если искомый номер найден, то система автоматически на уровне аппаратуры осуществляет проверку соответствия атрибутов, и если проверка успешна, то происходит подмена номера виртуальной страницы номером физической страницы, и, таким образом, получается физический адрес.

Если же при поиске происходит промах (номер виртуальной странице не найден), то в этом случае система обращается в программную таблицу, выкидывает самую старую запись из TLB, загружает в нее найденную запись из программной таблицы, и затем вычисляется физический адрес. Таким образом, получается, что TLB-таблица является некоторым КЭШем.

Модели отработки промаха могут быть различными. Возможна организация отработки промаха без прерываний, когда система самостоятельно, имея регистр начала программной таблицы страниц, обращается к этой таблице и осуществляет в ней поиск. Возможна модель с прерыванием, когда при промахе возникает прерывание, управление передается операционной системе, которая затем начинает работать с программной таблицей страниц, и т.д. Заметим, что вторая модель менее эффективная, поскольку прерывания ведут к увеличению накладных расходов.

Одним из решений, позволяющих снизить размер таблицы страниц, является модель **иерархической организации таблицы страниц** (Рис. 137). В этом случае информация о странице представляется не в виде одного номера страницы, а в виде совокупности номеров, используя которые, можно получить номер соответствующей физической страницы, посредством обращения к соответствующим таблицам, участвующим в иерархии (это может быть 2-х-, 3-х- или даже 4-хуровневая иерархия).

Пусть имеется 32-разрядный виртуальный адрес, который в свете рассмотренной ранее модели может, например, содержать 20-разрядный номер виртуальной страницы и 12-разрядное значение смещения в ней. Если же используется двухуровневая иерархическая организация, то этот же виртуальный адрес можно трактовать, к примеру, как 10-разрядный индекс во «внешней» таблице групп (или кластеров) страниц, 10-разрядное смещение в таблице второго уровня и, наконец, 12-разрядное смещение в физической странице. Соответственно, чтобы получить номер физической страницы необходимо по индексу во «внешней» таблице групп страниц найти необходимую ячейку, содержащую начальный адрес таблицы второго уровня, затем по этому адресу и по значению смещения в виртуальном адресе находится нужная запись в таблице страниц второго уровня, которая уже и содержит номер соответствующей физической страницы.

Используя данный подход, может оказаться, что всю таблицу страниц хранить в памяти вовсе необязательно: из-за принципа локализации будет достаточно хранить сравнительно небольшую «внешнюю» таблицу групп страниц и некоторые таблицы второго уровня (они также имеют незначительные размеры); все необходимые таблицы второго уровня можно подкачивать по мере надобности.

Подобные рассуждения можно распространить на большее число уровней иерархии, но, начиная с некоторого момента, эффективность системы начинает сильно падать с ростом числа уровней иерархии (из-за различных накладных расходов), поэтому обычно число уровней ограничено четырьмя.

Существует иное решение, позволяющее также обойти проблему большого размера таблицы страниц, - оно основано на использовании **хеширования** (на использовании т.н. **хеш-таблиц**). Это решение в свою очередь, базируется на использовании **хеш-функции**, или функции расстановки (Рис. 138). Эти функции используются в следующей задаче: пусть имеется некоторое множество значений, которое необходимо каким-то образом отобразить на множество фиксированного размера. Для осуществления этого отображения используют функцию, которая по входному значению определяет номер позиции (номер кластера, куда должно попасть это значение). Но эта функция имеет свои особенности: при ее использовании возможны коллизии, связанные с тем, что различные значения могут оказаться в одном и том же кластере.

Модель преобразования адресов, основанная на хешировании, достаточно проста. Из виртуального адреса аппаратно извлекается номер виртуальной страницы, который подается на вход некоторой хеш-функции, отображающей значение на аппаратную таблицу (т.н. хеш-таблицу) фиксированного размера. Каждая запись в данной таблице хранит начало списка коллизий, где каждый элемент списка является парой: номер виртуальной страницы — соответствующий ему номер физической страницы. Итак, перебирая соответствующий список коллизий, можно найти номер исходной виртуальной страницы и соответствующий номер физической страницы. Подобное решение имеет свои достоинства и недостатки: в частности, возникают проблемы с перемещением списков коллизий.

Еще одним решением, позволяющим снизить размер таблицы страниц, является модель использования т.н. **инвертированных таблиц страниц** (Рис. 139). Главной сложностью данного решения является требование к процессору на аппаратном уровне работать с идентификаторами процессов (их PID). Примерами таких процессоров могут служить процессоры серий SPARC и PowerPC.

В этой модели виртуальный адрес трактуется как тройка значений: PID процесса, номер виртуальной страницы и смещение в этой странице. При таком подходе используется единственная таблица страниц для всей системы, и каждая строка данной таблицы соответствует физической странице (с номером, равным номеру этой строки). При этом каждая запись данной таблицы содержит информацию о том, какому процессу принадлежит данная физическая страница, а также какая виртуальная страница этого процесса размещена в данной физической странице. Итак, имея пару PID процесса и номер виртуальной страницы, производится поиск ее в таблице страниц, и по смещению найденного результата определяется номер физической страницы.

К достоинствам данной модели можно отнести наличие единственной таблицы страниц, обновление которой при смене контекстов сравнительно нетрудоемкое: операционная система производит обновление тех строк таблицы, для которых в соответствующие физические страницы происходит загрузка процесса. Отметим, что «тонким местом» данной модели является организация поиска в таблице. Если будет

Революционным достоинством страничной организации памяти стало то, что исполняемый в системе процесс может использовать очень незначительную часть физического ресурса памяти, а все остальные его страницы могут размещаться во внешней памяти (быть откачанными). Очевидно, что и страничная организация памяти

5.5 Сегментное распределение

Недостатком страничного распределения памяти является то, что при реализации этой модели процессу выделяется единый диапазон виртуальных адресов: от нуля до некоторого предельного значения. С одной стороны, ничего плохого в этом нет, но это свойство оказывается неудобным по следующей причине. В процессе есть команды, есть статические переменные, которые, по сути, являются разными объединениями данных с различными характеристиками использования. Еще большие отличия в использовании иллюстрируют существующие в процессе стек и область динамической памяти, называемой также *кучей*. И модель страничной организации памяти подразумевает статическое разделение единого адресного пространства: выделяются область для команд, область для размещения данных, а также область для стека и кучи. При этом зачастую стек и куча размещаются в единой области, причем стек прижат к одной границе области, куча — к другой, и «растут» они навстречу друг другу. Соответственно, возможны ситуации, когда они начинают пересекаться (ситуация переполнения стека). Или даже если стек будет располагаться в отдельной области памяти, он может переполнить выделенное ему пространство. Таковы основные недостатки страничного распределения памяти.

Избавиться от указанных недостатков на концептуальном уровне призвана модель сегментного распределения памяти (Рис. 141). Данная модель представляет каждый процесс в виде совокупности сегментов, каждый из которых может иметь свой размер. Каждый из сегментов может иметь собственную функциональность: существуют сегменты кода, сегменты статических данных, сегмент стека и т.д. Для организации работы с сегментами может использоваться некоторая таблица, в которой хранится информация о каждом сегменте (его номер, размер и пр.). Тогда виртуальный адрес может быть проинтерпретирован как номер сегмента и величина смещения в нем.

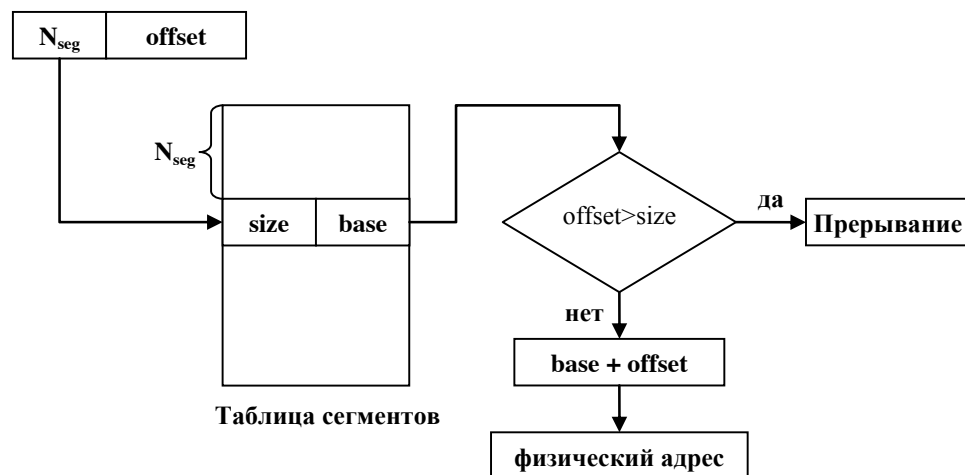


Рис. 141. Сегментное распределение.

Модель сегментного распределения может иметь достаточно эффективно работающую аппаратную реализацию. Существует аппаратная таблица сегментов с фиксированным числом записей. Каждая запись этой таблицы соответствует своему сегменту и хранит информацию о размере сегмента и адрес начала сегмента (т.е. адрес базы), а также тут могут присутствовать различные атрибуты, которые будут оговаривать права и режимы доступа к содержимому сегмента.

Итак, имея виртуальный адрес, система аппаратным способом извлекает из него номер сегмента i , обращается к i -ой строке таблицы, из которой извлекается информация о сегменте. После этого происходит проверка, не превосходит ли величина сегмента

размера самого сегмента. Если превосходит, то происходит прерывание; иначе, складывая базу со смещением, вычисляется физический адрес.

К достоинствам данной модели можно отнести простоту организации, которая, по сути, явилась развитием модели распределения разделов. Если модели распределения разделов каждому процессу выделяется только один сегмент (раздел), то при сегментной модели распределения процессу выделяется совокупность сегментов, каждый из которых будет иметь свои функциональные обязанности.

К недостаткам данной модели необходимо отнести то, что каждый сегмент должен целиком размещаться в памяти (возникает упоминавшаяся выше проблема неявной неэффективности, связанная с принципом локальности). Также возникают проблемы с откачкой/подкачкой: подкачка осуществляется всем процессом или, по крайней мере, целым сегментом, что зачастую оказывается неэффективно. И поскольку каждый сегмент так или иначе должен быть размещен в памяти, то возникает ограничение на предельный

5.6 Сегментно-страничное распределение

Естественным развитием рассмотренной модели сегментного распределения памяти стала модель сегментно-страничного распределения. Эта модель рассматривает виртуальный адрес как номер сегмента и смещение в нем. Имеется также аппаратная таблица сегментов, посредством которой из виртуального адреса получается т.н. **линейный адрес**, который, в свою очередь, представляется в виде номера страницы и величины смещения в ней. А затем, используя таблицу страниц, получается непосредственно физический адрес.

Итак, данный механизм подразумевает, что в процессе имеется ряд виртуальных сегментов, которые дробятся на страницы. Поэтому данная модель сочетает в себе, с одной стороны, логическое сегментирование, а с другой стороны, преимущества страничной организации (когда можно работать с отдельными страницами памяти, не требуя при этом полного размещения сегмента в ОЗУ).

Примером реализации может служить реализация, предложенная компанией Intel. Рассмотрим упрощённую модель этой реализации (Рис. 142). Виртуальный адрес в этой модели представляется в виде *селектора* (информации о сегменте) и смещения в сегменте.

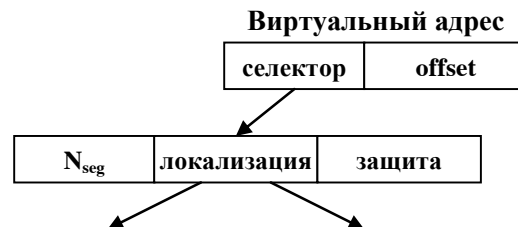


Рис. 142. Сегментно-страничное распределение. Упрощенная модель Intel.

Селектор содержит информацию о локализации сегмента. В модели Intel сегмент может быть одного из двух типов: *локальный сегмент*, который описывается в таблице локальных дескрипторов **LDT** (Local Descriptor Table) и который может быть доступен лишь данному процессу, или *глобальный сегмент*, который описывается в таблице глобальных дескрипторов **GDT** (Global Descriptor Table) и который может разделяться между процессами. Заметим, что каждая запись таблиц LDT и GDT хранит полную информацию о сегменте (адрес базы, размер и пр.). Итак, в селекторе хранится тип

сегмента, после которого следует номер сегмента (номер записи в соответствующей таблице). Помимо перечисленного, селектор хранит различные атрибуты, касающиеся режимов доступа к сегменту.

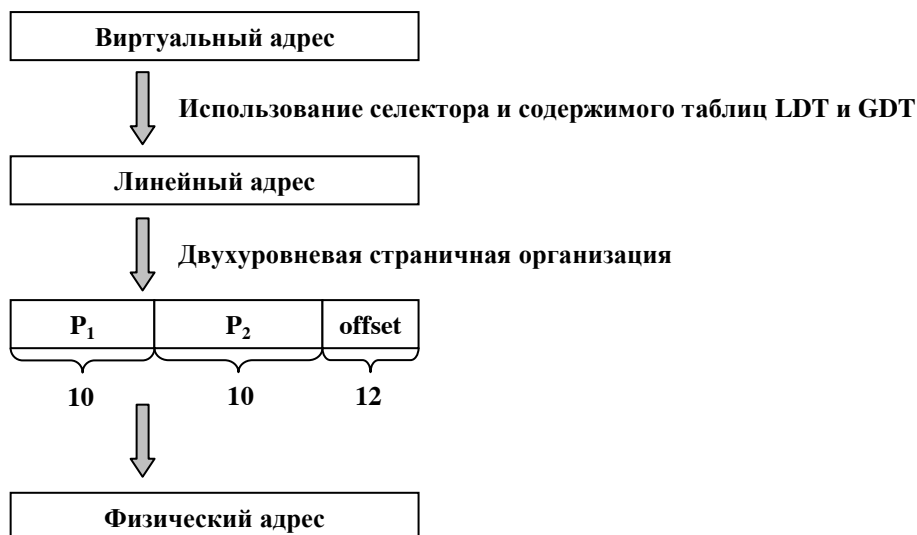


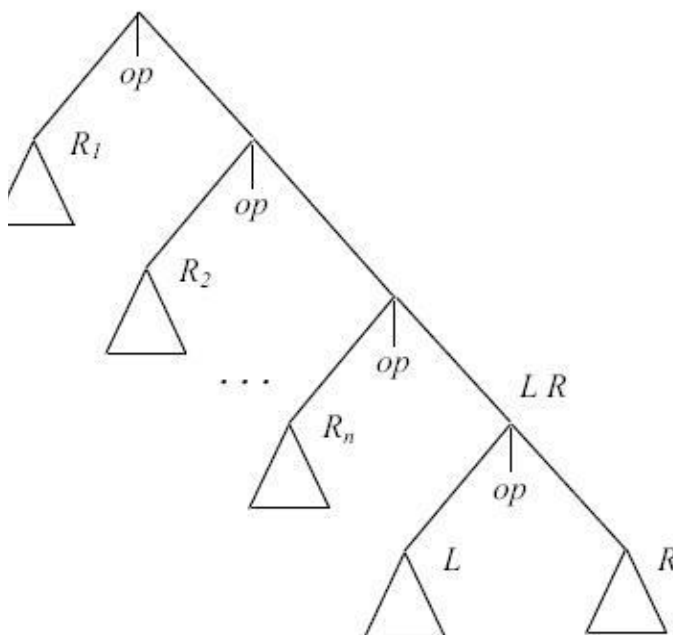
Рис. 143. Схема преобразования адресов.

Среди особенностей данной модели можно отметить, что можно «выключать» страничную функцию, и тогда модель Intel начинает работать по сегментному распределению. А можно не использовать сегментную организацию процесса, и тогда данная реализация будет работать по страничному распределению памяти.

9. Алгоритм Сети-Ульмана оптимального распределения регистров и его обоснование.

$op\ R_i, R_j \quad R_j := R_i\ op\ R_j$
(рез-т сохраняется во втором регистре)

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



Предположим сначала, что распределение регистров осуществляется по простейшей схеме слева-направо, как изображено на рис. 8.13. Тогда к моменту генерации кода для поддерева LR занято n регистров. Пусть поддерево L требует nl регистров, а поддерево R - nr регистров. Если $nl=nr$, то при вычислении L будет использовано nl регистров и под результат будет занят $n+1$ -й регистр. Еще nr ($=nl$) регистров будет использовано при вычислении R . Таким образом, общее число использованных регистров будет равно $n+nl+1$. Если $nl>nr$, то при вычислении L будет использовано nl регистров. При вычислении R будет использовано $nr<nl$ регистров, и всего будет использовано не более чем $n+nl$ регистров.

Если $n_l < n_r$, то после вычисления L под результат будет занят один регистр (предположим $n+1$ -й) и n_r регистров будет использовано для вычисления R . Всего будет использовано $n+n_r+1$ регистров. Видно, что для деревьев, совпадающих с точностью до порядка потомков каждой вершины, минимальное число регистров при распределении их слева-направо достигается на дереве, у которого в каждой вершине слева расположено более "сложное" поддерево, требующее большего числа

регистров. Таким образом, если дерево таково, что в каждой внутренней вершине правое поддерево требует меньшего числа регистров, чем левое, то, обходя дерево слева направо, можно оптимально распределить регистры. Без перестройки дерева это означает, что если в некоторой вершине дерева справа расположено более сложное поддерево, то сначала сгенерируем код для него, а затем уже для левого поддерева. Алгоритм работает следующим образом. Сначала осуществляется разметка синтаксического дерева по следующим правилам.

Правила разметки:

1) если вершина - правый лист или дерево состоит из единственной вершины, помечаем эту вершину числом 1, если вершина - левый лист, помечаем ее 0 (рис. 8.14).

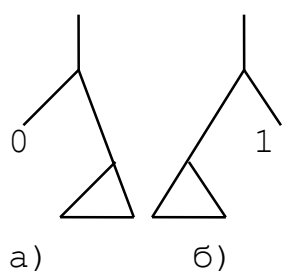


Рис. 8.14

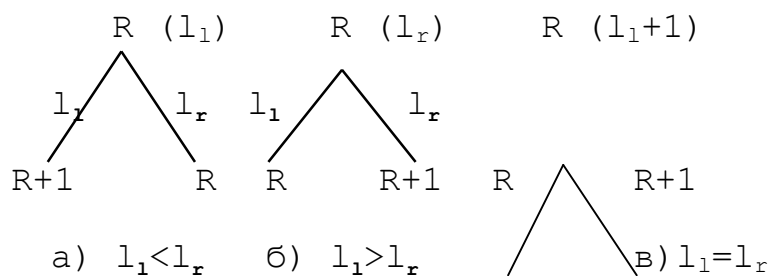


Рис. 8.15

2) если вершина имеет прямых потомков с метками l_1 и l_2 , то в качестве метки этой вершины выбираем большее из чисел l_1 или l_2 либо число l_1+1 , если $l_1=l_2$ (рис. 8.15).

Эта разметка позволяет определить, какое из поддеревьев требует большего количества регистров для своего вычисления.

Затем осуществляется распределение регистров для результатов операций.

Правила распределения регистров.

1) Корню назначается первый регистр.

2) Если метка левого потомка меньше метки правого, то левому потомку назначается регистр на единицу больший, чем предку, а правому - с тем же номером (сначала вычисляется правое поддерево и его результат

помещается в регистр R), так что регистры занимаются последовательно. Если же метка левого потомка больше или равна метке правого потомка, то наоборот, сначала вычисляется левое поддерево и его результат помещается в регистр R (рис. 8.15), а правому потомку – R+1. После этого формируется код по следующим правилам.

Правила генерации кода:

1) если вершина - правый лист с меткой 1, то ей соответствует код `LOAD X,R`, где R - регистр, назначенный этой вершине, а X - адрес переменной, связанной с вершиной (рис. 8.16.б);

2) если вершина внутренняя и ее левый потомок - лист с меткой 0, то ей соответствует код (рис. 8.16а)

Код правого поддерева
Op X, R

где снова R - регистр, назначенный этой вершине, X - адрес переменной, связанной с вершиной, а Op - операция, примененная в вершине (рис. 8.16.а);

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

Код правого поддерева
Код левого поддерева
Op R+1, R

где R - регистр, назначенный внутренней вершине, и операция Op, вообще говоря, не коммутативная (рис. 8.17 б)).

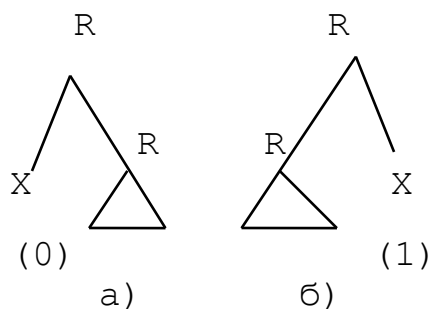


Рис. 8.16

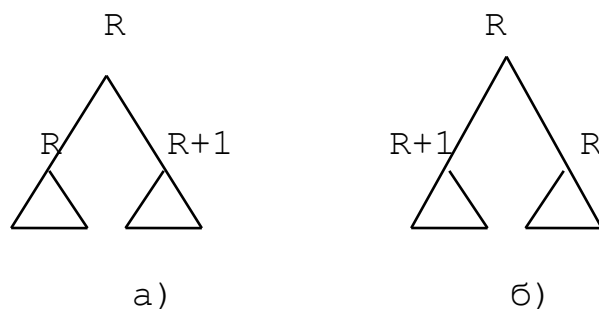


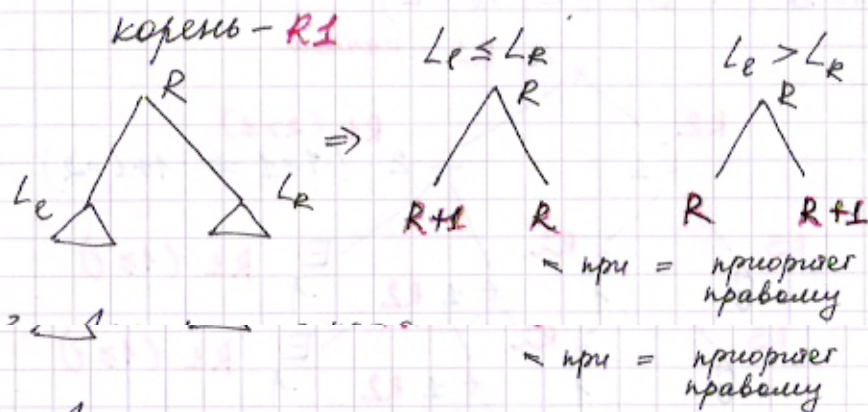
Рис. 8.17

Если метка правой вершины меньше или равна метке левой вершины, то вершине соответствует код

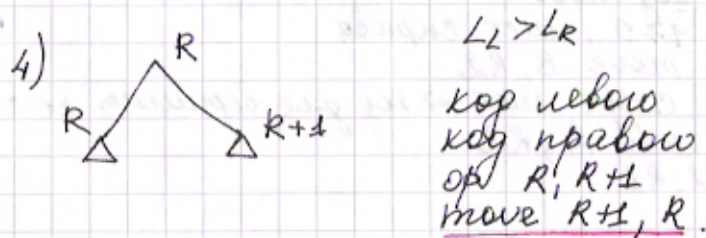
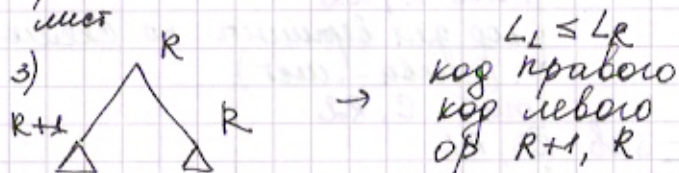
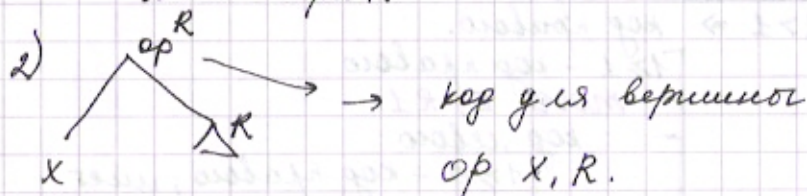
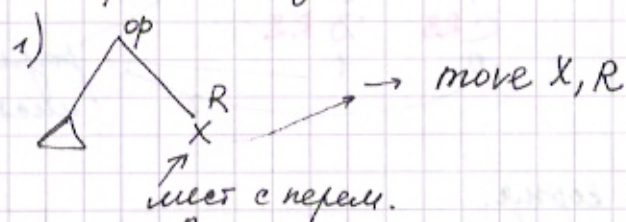
Код левого поддерева
Код правого поддерева
Op R, R+1
MOVE R+1, R

Последняя команда генерируется для того, чтобы получить результат в нужном регистре (в случае коммутативной операции операнды операции можно поменять местами и избежать дополнительной пересылки)(рис. 8.17 а)).

2. Распределение ролей

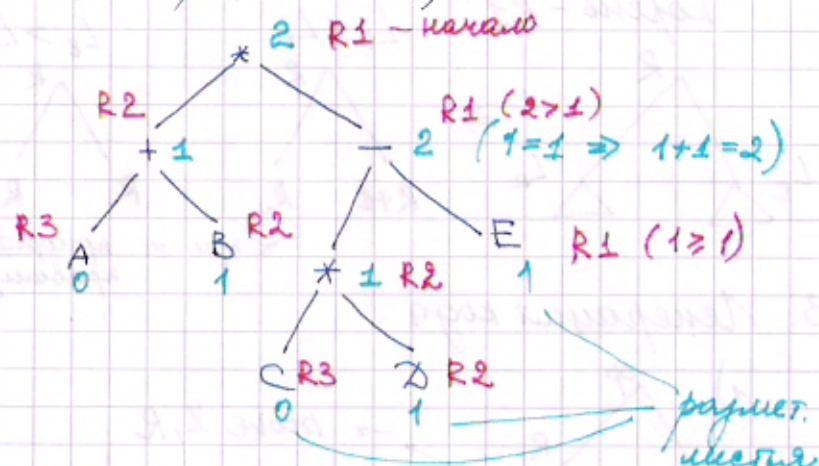


3. Генерация кода



Пример:

$$A * B (A+B) * (C * D - E)$$



Код:

начинаем с корня.

- $2 > 1 \Rightarrow$ код правую.

- $1 \geq 1$ - код правую

move E, R1

- код левую

- $1 \geq 0$ - код правую, имеет

move D, R2

код для вершины по схеме 2)

(слева - имеет)

mul C, R2

- sub R2, R1

код левую

- $1 \geq 0$, имеет справа

move B, R2

слева имеет $\Rightarrow$ код для вершины по сх. 1)

add A, R2

- mul R2, R1

10. Основные принципы объектно-ориентированного программирования.

Объектно-Ориентированное Программирование - это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследования.

Объектно-ориентированная парадигма, при которой программа рассматривается как совокупность фрагментов кода, обрабатывающих отдельные совокупности данных — **объекты**. Эти объекты взаимодействуют друг с другом посредством так называемых **интерфейсов**. При этом данные управляют доступом к коду.

В этом определении можно выделить три части

1. ООП использует в качестве элементов конструкции объекты, а не алгоритмы [как структурное программирование]
2. Каждый объект является реализацией какого-либо определенного типа (класса)
3. Классы организованы иерархически

При несоблюдении хотя бы 1 из указанных требований программа перестает быть ОО. (В частности при нарушении 3 имеем программирование на основе Абстрактных Типов Данных)

Центральной идеей ООП является реализация понятия «**абстракция**». Смысл абстракции заключается в том, что *сущность* произвольной сложности можно рассматривать, а также производить определенные действия над ней, как над **единым целым**, не вдаваясь в детали внутреннего построения и функционирования.

Одним из основных способов создания абстракции является использование концепции **иерархической классификации**. Ее суть заключается в том, что сложные системы разбиваются на более простые фрагменты.

Практически все сложные системы иерархичны, и уровни их иерархии отражают различные уровни абстракции. Для каждой конкретной задачи рассматривается соответствующий уровень. Выбор низшего уровня абстракции достаточно произволен. Выбранный уровень в одном случае в качестве низшего уровня может оказаться уровнем достаточно высокой абстракции в другом проекте.

Различают **типовую** иерархию и **структурную** иерархию, которые далее мы будем называть соответственно **структурой классов** и **структурой объектов**.

Во всех объектно-ориентированных языках программирования реализованы следующие **основные механизмы (постулаты) ООП**:

- **инкапсуляция,**
- **наследование,**
- **полиморфизм.**

Все эти механизмы важны для разработки и использования абстракций.

1) Инкапсуляция — механизм, связывающий вместе код и данные, которыми он манипулирует, и одновременно защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому. Доступ к коду и данным жестко контролируется интерфейсом.

Основой инкапсуляции при ООП является **класс**.

Механизма инкапсуляции позволяет оставлять скрытыми от пользователя некоторые детали реализации класса (то есть инкапсулировать их в классе), что упрощает работу с объектами этого класса.

2) Наследование — механизм, с помощью которого один объект (производного класса) приобретает свойства другого объекта (родительского, базового класса). При использовании наследования новый объект не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста. Наследование позволяет какому-либо объекту наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Наследование есть очень важное понятие, поддерживающее концепцию иерархической классификации.

Абстрагирование данных состоит в установлении жестких границ, защищающих состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных классов. В связи с этим интерфейсная часть класса может быть разделена на три части:

Обособленную (private) - видимая только для самого класса

Защищенную (protected) - видимую также и для подклассов

Общедоступную (public) - видимую для всех

Наиболее полный контроль видимости реализован в C++.

3) Полиморфизм — механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

Общая концепция полиморфизма: **один интерфейс — много методов**.

Выбор конкретного действия (метода) применительно к конкретной ситуации возлагается на компилятор. Программисту же достаточно запомнить и применить один интерфейс, вместо нескольких, что также упрощает работу.

Различаются **статический** (реализуется на этапе компиляции с помощью перегрузки функций и операций), **динамический** (реализуется во время выполнения программы с помощью механизма виртуальных функций) и **параметрический** (реализуется на этапе компиляции с использованием механизма шаблонов) полиморфизм.

Дополнительные возможности ОО языков.

Некоторые языки позволяют определять несколько специальных методов класса:

Конструктор - специальная процедура класса для создания и/или инициализации начального состояния объекта. В частности, конструктор может инициализировать таблицу виртуальных функций.

Деструктор - специальная процедура класса, которая делает состояние объекта неопределенным и (или) ликвидирует сам объект.

Некоторые преимущества ОО подхода:

1. Использование объектного подхода существенно повышает качество разработки в целом и ее фрагментов. ОО Системы часто получаются более компактными чем их не-ОО аналоги
2. Использование объектного подхода приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к ее полной переработке в случае существенных изменений исходных требований
3. Ориентирован на человеческое восприятие мира

Типы данных, создаваемые пользователем (программистом), называются **пользовательскими типами данных**. Пользовательский тип данных с полностью скрытой (инкапсулированной) внутренней структурой называется **абстрактным типом данных (АТД)**.

Ассоциация представляет смысловую связь между сущностями (объектами), создаваемыми на основе классов. Ассоциация (*association*) определяется некоторой связью между классами. Когда в системе создаются представители ассоциированных классов, они связываются так, как определяет данная ассоциация.

1. **Сущность** — класс однотипных объектов, информация о которых должна быть учтена в модели. Сущность в рамках ООП представляется классом.
2. **Экземпляр сущности** — объект, создаваемый на основе класса.
3. **Атрибут сущности** — именованная характеристика. в ООП — информационный член класса
4. **Ключ сущности** — совокупность атрибутов, однозначно определяющих объект.
5. **Связь** — ассоциация между сущностями.

Наследование — отношение между классами, при котором один класс повторяет структуру и поведение другого класса (*одиночное наследование*) или других (*множественное наследование*) классов.

Агрегация — это отношение между классами типа целое/часть. Агрегируемый класс в той или иной форме является частью агрегата. Объект класса-агрегата может хранить объект агрегируемого класса, или хранить ссылку (указатель) на него.

Композиция является специальным видом агрегирования (так называемое сильное агрегирование). Композиция объектов заключается

в использовании объектов типов разработанных классов в качестве информационных членов при описании других классов.

Использование — отношение между классами, при котором один класс в своей реализации использует в той или иной форме реализацию объектов другого класса.

Использование одним классом объектов другого класса может проявляться одним из следующих образов:

- Имя одного класса используется в профиле метода другого класса
- В теле метода одного класса создаётся локальный объект другого класса
- Метод одного класса обращается к методу другого класса (не совсем частный случай предыдущего способа использования, так как при вызове статических членов классов локальный объект не создаётся).

Для реализации механизма виртуальных функций используется специальный, связанный с полиморфным типом, массив указателей на виртуальные методы класса. Такой массив называется **Таблицей Виртуальных Методов (ТВМ)**. в каждый полиморфный объект компилятор неявно помещает указатель, условно обозначаемый как

```
vtbl* pvtbl;
```

на соответствующую TBM, хранящую адреса виртуальных методов.

В TBM типа-наследника имеющиеся адреса одинаковых методов замещаются, а новые — дописываются в конец. Так, если бы в классе *C* был описан метод с прототипом *int sa()*, то TBM для класса *C* имела бы вид:

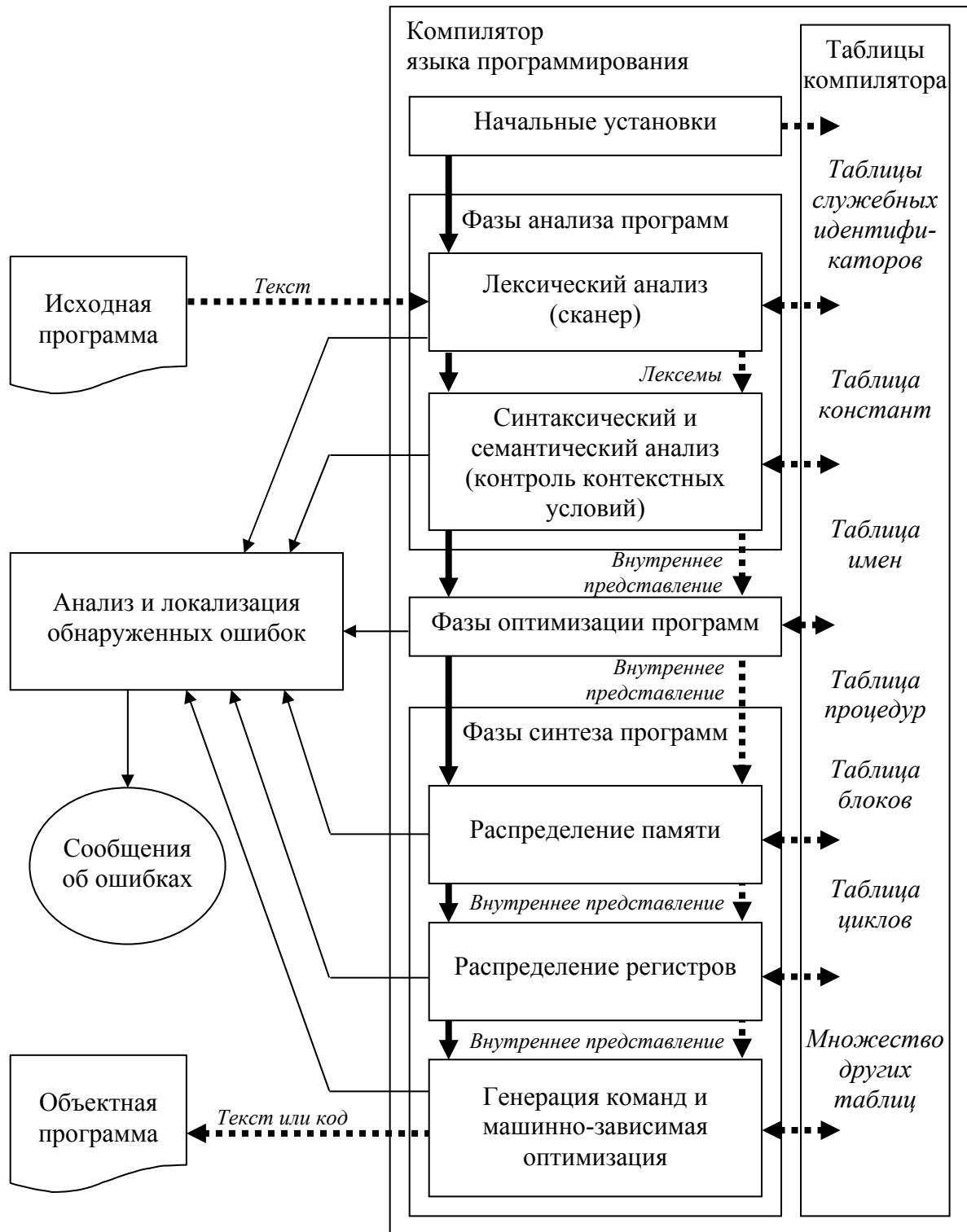
<i>A::</i>	<i>& ~A()</i>
<i>C::</i>	<i>& sa()</i>
<i>C::</i>	<i>& ~C()</i>
<i>C::</i>	<i>& sc()</i>

Так как указатель на TBM находится в самом начале объекта, то он доступен всегда, каким бы ни был тип указателя на объект. Конечно, при этом из TBM могут быть выбраны только те методы, которые имеются в структуре указателя (входят в так называемый интерфейс). То есть, как показано в примере, если объект производного типа обрабатывается через указатель базового типа, то из TBM данного объекта можно вызывать только виртуальные методы, перечисленные в базовом типе. Естественно, при этом будет выполняться алгоритм, определенный для данного объекта в соответствии с его типом.

Издержками при использовании виртуальных функций является дополнительная память для неявного хранения указателя на TBM в каждом полиморфном объекте.

Абстрактный класс — это класс, содержащий хотя бы одну чистую виртуальную функцию.

11. Основные этапы компиляции (лексический анализ, синтаксический анализ, семантический анализ, генерация кода и т.д.)



3.3.1.1. Основные компоненты компилятора и фазы компиляции

Информационные таблицы. При анализе программы из имеющихся в ней описаний, заголовков процедур, блоков, циклов и других структурных операторов извлекается информация, которая должна сохраняться для использования на последующих фазах компиляции. Вся эта информация размещается в информационных таблицах компилятора. Если при просмотре программы встречается некоторый идентификатор, в таблицах должны быть сведения о том, как он был описан и как использовался в программе. Конкретная информация зависит от исходного языка, от объектного языка и сложности транслятора. Кроме таблицы идентификаторов (“имен”) обычно строится таблица констант, в которую включаются все константы, использованные в программе, и заносятся их адреса в объектной программе.

Начальные установки. Перед началом работы с программой компилятору необходимо провести иницилирующие мероприятия. К таковым относятся ввод и обработка режимов запуска компилятора и первичное заполнение таблиц исходной информацией.

Фазы анализа программ. На этапе анализа в компиляторе выполняется распознавание текста исходной программы и заполнение информационных таблиц. В результате формируется некоторое внутреннее представление программы, понятное

следующим фазам компиляции. Получая на вход исходную программу как последовательность символов входного языка (“цепочку”), компилятор должен проверить, принадлежит ли она входному языку, а также определить правила, по которым эта последовательность строилась. Анализ часто подразделяется на лексический, синтаксический и семантический анализ.

Лексический анализатор (сканер). Основная задача лексического анализатора – просмотреть весь текст исходной программы и выделить в нем лексемы (*минимальные лексические единицы или элементы текста программы, обладающие смыслом в рамках данного языка*). В обычных языках программирования лексемами являются числа (десятичные, целые, вещественные), идентификаторы, служебные слова, разделители. Задачей лексического анализатора является замена разнообразных элементов текста стандартно выглядящими лексемами, которые в дальнейшем будет легче обрабатывать в других частях компилятора. Лексический анализ сопровождается исключением незначащих фрагментов текстов программ, например, комментариев. Для тех языков, в которых имеются макросредства, дополнительно выполняется расширение макровыводов. Подробнее задачи и проблемы лексического анализа рассматриваются в пособии “Формальные грамматики и языки. Элементы теории трансляции”.

Таким образом, общая схема работы лексического анализатора такова. Сначала выделяется отдельная лексема (при этом, возможно, используются символы-разделители). Ключевые слова распознаются явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов.

Если выделенная лексема является ограничителем, то этот ограничитель (точнее, некоторый его признак) выдается как результат лексического анализа. Если выделенная лексема является ключевым словом, то выдается признак соответствующего ключевого слова. Если выделенная лексема является идентификатором - выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Наконец, если выделенная лексема принадлежит какому-либо из других классов лексем (например, лексема

представляет собой число, строку и т.д.), то выдается признак соответствующего класса, а значение лексемы сохраняется отдельно.

Лексический анализатор может быть как самостоятельной фазой трансляции, так и подпрограммой, работающей по принципу "дай лексему". В первом случае (рис. 3.1, а) выходом анализатора является файл лексем, во втором - (рис. 3.1., б) лексема выдается при каждом обращении к анализатору (при этом, как правило, признак класса лексемы возвращается как результат функции "лексический анализатор", а значение лексемы передается через глобальную переменную). С точки зрения обработки значений лексем, анализатор может либо просто выдавать значение каждой лексемы, при этом построение таблиц объектов (идентификаторов, строк, чисел и т.д.) переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов. В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу. Работа лексического анализатора задается некоторым конечным автоматом. Однако, непосредственное описание конечного автомата неудобно с практической точки зрения. Поэтому для задания лексического анализатора, как правило, используется либо регулярное выражение, либо праволинейная грамматика.

Определение: порождающая грамматика G — это четверка $\langle T, N, P, S \rangle$, где

- T — алфавит терминальных символов (терминалов);
- N — алфавит нетерминальных символов (нетерминалов), $T \cap N = \emptyset$;
- P — конечное подмножество множества $(T \cup N)^+ \times (T \cup N)^*$; элемент (α, β) множества P называется *правилом вывода* и записывается в виде $\alpha \rightarrow \beta$; α называется *левой частью* правила, β — *правой частью*; левая часть любого правила из P обязана содержать хотя бы один нетерминал;
- S — начальный символ (цель) грамматики, $S \in N$.

Для записи правил вывода с одинаковыми левыми частями

$$\alpha \rightarrow \beta_1 \quad \alpha \rightarrow \beta_2 \quad \dots \quad \alpha \rightarrow \beta_n$$

будем пользоваться сокращенной записью

$$\alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n.$$

Каждое β_i ($i = 1, 2, \dots, n$) будем называть *альтернативой* правила вывода из цепочки α .

Грамматика называется праволинейной, если она содержит только правила вида $A \rightarrow a$, $A \rightarrow aB$.

Все три формализма (конечных автоматов, регулярных выражений и праволинейных грамматик) имеют одинаковую выразительную мощность. В частности, по регулярному выражению или праволинейной грамматике можно сконструировать конечный автомат, распознающий тот же язык.

Синтаксический и семантический анализаторы. Программа должна быть проверена на синтаксическую и семантическую правильность (должно быть проверено соблюдение контекстных условий), разделена на составные части, для каждой из которых должно быть сформировано внутреннее представление. В таблицы транслятора должна быть занесена вся информация, которую можно извлечь из обрабатываемой программы. Подробнее задачи и проблемы синтаксического и семантического анализа рассматриваются в разделе 3.3.2 и в пособии “Формальные грамматики и языки. Элементы теории трансляции”.

Результатом обычно является дерево разбора (синтаксическое дерево).

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

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

Метод рекурсивного спуска

LL-анализатор

Восходящий парсер (англ. bottom-up parser) — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.

LR-анализатор

GLR-парсер

Семантический анализ

Задачей контекстного анализа является установление свойств объектов и их использования. Наиболее часто решаемой задачей является определение существования объекта и соответствия его использования контексту, что осуществляется с помощью анализа типа объекта. Под контекстом здесь понимается вся совокупность свойств текущей точки программы, например множество доступных объектов, тип выражения и т.д.

Таким образом, необходимо хранить объекты и их типы, уметь находить эти объекты и определять их типы, определять характеристики контекста. Совокупность доступных в данной точке объектов будем называть средой.

В процессе работы компилятор хранит информацию об объектах программы в специальных таблицах символов. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и описания объекта. Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрее, а требуемая память по возможности меньше. Кроме того, со стороны языка программирования могут быть дополнительные требования к организации информации. Имена могут иметь определенную область видимости. Например, поле записи должно быть уникально в пределах структуры (или уровня структуры), но может совпадать с именем объекта 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 при 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.

Некоторые основные способы организации таблиц символов в компиляторе: таблицы идентификаторов, таблицы расстановки, двоичные деревья и реализация блочной структуры.

Внутреннее представление исходной программы. Внутреннее представление исходной программы в компиляторе в наибольшей степени зависит от той обработки, которой должна подвергнуться программа. Некоторые виды внутреннего представления больше подходят для фиксации структуры компилируемой программы, другие ориентированы на проведение оптимизирующих преобразований, третьи наиболее удобны при синтезе (генерации) результата компиляции. Более подробно внутреннее представление программ в компиляторах рассматривается в разделе 3.3.3 и в пособии “Формальные грамматики и языки. Элементы теории трансляции”.

Язык внутреннего представления программы

Основные свойства языка внутреннего представления программ:

- а) он позволяет фиксировать синтаксическую структуру исходной программы;
- б) текст на нем можно автоматически генерировать во время синтаксического анализа;
- в) его конструкции должны относительно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

Некоторые общепринятые способы внутреннего представления программ:

- а) постфиксная запись;
- б) префиксная запись;
- в) многоадресный код с явно именуемыми результатами;
- г) многоадресный код с неявно именуемыми результатами;
- д) связанные списочные структуры, представляющие синтаксическое дерево.

Выберем в качестве языка для представления промежуточной программы *постфиксную запись* (ее также называют **ПОЛИЗ** — польская инверсная запись).

- связанные списочные структуры, представляющие синтаксическое дерево;
- многоадресный код с явно именуемыми результатами (тетрады – *оператор, операнд, операнд, результат*);
- многоадресный код с неявно именуемыми результатами (триады – *оператор, операнд-результат, операнд*), иногда этот способ представления модифицируется и используется последовательность косвенных триад (сами триады хранятся в отдельной таблице, формируемая программа представляет собой последовательность ссылок на таблицу);
- инфиксная запись (операции записываются между своими операндами, как в обычной записи арифметических выражений);
- префиксная запись (операции записываются перед своими операндами, как в записи вызова процедуры или функции с параметрами);
- постфиксная запись (операции записываются после своих операндов);
- язык ассемблера целевой или абстрактной машины.

Связные списочные структуры. Списки – это структуры представления программ, которые можно наиболее просто и эффективно строить на этапе синтаксического анализа. Обычно списки используются для представления синтаксических деревьев – таких структур, вершинами которых являются операции, а листьями – операнды. Как правило, листья связаны с записями в таблицах идентификаторов и констант. Структура синтаксических деревьев наиболее точно отражает синтаксис языка программирования, на котором была написана исходная программа. Обычно перед построением синтаксических деревьев в грамматиках языка избавляются от цепных правил вида $A \rightarrow B$, где A и B – нетерминальные символы.

В тех случаях, когда синтаксическому дереву соответствует последовательность операций, порождающая в результате команды объектной программы, такое дерево называют деревом операций. Дерево операций строится непосредственно и автоматически из дерева вывода, порождаемого синтаксическим анализатором. Для этого из него удаляются цепочки нетерминальных символов, а также узлы, не несущие семантической нагрузки при генерации объектной программы (например, скобки, меняющие порядок операций):

Многоадресный код с явно именуемым результатом (тетрады). Тетрады представляют собой запись операций в форме четырех составляющих: операции, двух операндов и результата операции:

<операция>(<операнд1>,<операнд2>,<результат>)

Тетрады составляют линейную последовательность команд, как в следующей записи арифметического выражения $A=B*C+D-B*10$ в виде последовательности тетрад:

1	*	B	C	T1
2	+	T1	D	T2
3	*	B	10	T3
4	-	T2	T3	T4
5	=	T4	?	A

Многоадресный код с неявно именуемым результатом (триады). Триады представляют собой запись операций в форме из трех составляющих: операции и двух операндов:

<операция>(<операнд1>,<операнд2>)

Особенностью триад является то, что один или оба операнда в триаде могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады, например, так в виде триад можно записать арифметическое выражение $A=B*C+D-B*10$:

1	*	B	C
2	+	^1	D
3	*	B	10

Инфиксная запись. Запись операций и операндов в традиционном виде применяется только в программах, подаваемых на вход компиляторов. Это наиболее удобная для людей, но наименее удобная для автоматической обработки запись.

Префиксная запись. Префиксная запись иначе называется польской записью или прямой польской записью. В такой записи операторы (операции) предшествуют своим операндам. Некоторое неудобство прямой польской записи, которое привело к использованию обратной записи, состоит в том, что операторы в ней следуют не в том порядке, в каком они должны выполняться в вычислительной машине:

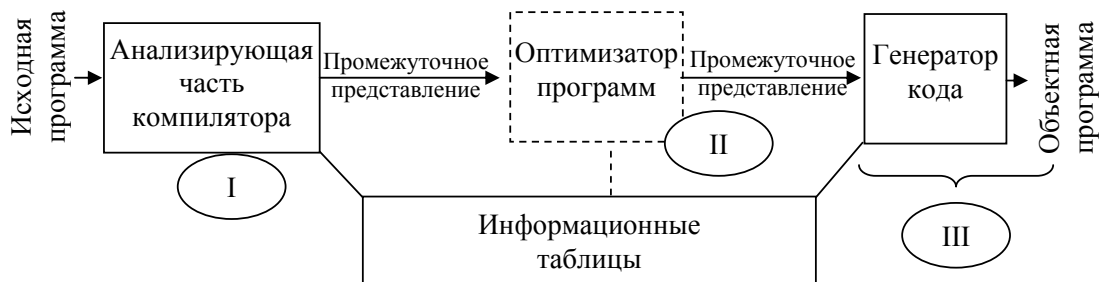
= A - + * B C D * B 10

Постфиксная (инверсная, обратная, суффиксная) запись. В отличие от прямой польской записи инверсная польская запись (ПОЛИЗ) обладает следующими важными свойствами:

- Операнды в обратной польской записи следуют в том же порядке, в каком они следуют в инфиксной записи.
- Операции в обратной польской записи следуют в том порядке, в каком они должны вычисляться (слева направо).
- Операции следуют непосредственно за своими операндами.

Язык ассемблера и машинные команды. Из определения компилятора, как частного случая транслятора с языка программирования высокого уровня на машинный язык или язык ассемблера, понятно, что представление программы в виде машинных команд или ассемблерной записи является обязательным. Однако некоторые компиляторы преобразованием программ к такому виду не заканчивают свою работу, а лишь продолжают ее. Именно в таком виде наиболее удобно проводить машинно-зависимую оптимизацию, при которой легче всего учесть семантические особенности выполнения отдельных связок команд и получить дополнительный выигрыш в производительности программы.

Фазы оптимизации программ. Оптимизация – важная задача компилятора. Языки высокого уровня, не связанные напрямую с особенностями конкретной аппаратуры, на которой должны выполняться программы, без оптимизации не могут использоваться для создания эффективных программ. Оптимизация программ может проводиться в интересах различных свойств программ. Обычно используют две стратегии оптимизации: оптимизация в целях повышения скорости работы программы и оптимизация в целях уменьшения размеров программ. Методы, используемые при реализации этих стратегий часто противоположны, хотя некоторые из них близки друг к другу. Более подробно проблемы оптимизации программ в компиляторах рассматриваются в разделе 3.3.4.



Первичную оптимизацию может проводить сам пользователь (пометка I). Некоторые системы программирования предлагают поддержку пользовательской оптимизации, в частности, имеют в своем составе профилировщики, помогающие выявить те фрагменты программ, которые для своего выполнения требуют максимальной доли времени работы программы. В целом оптимизация на уровне исходной программы может дать наибольший эффект для улучшения технических характеристик программы. К таким техническим характеристикам относят обычно две: объем памяти, необходимый для выполнения объектной программы (для хранения данных и самих команд программы), и скорость выполнения программы (ее быстродействие). Очень часто сокращение используемых в программе данных приводит к увеличению времени работы программы, а попытки повысить быстродействие приводят к увеличению используемой памяти. Поэтому часто для оптимизации выбирается один, главный критерий, либо некоторый интегрированный критерий, основанный на сбалансированном подходе к достижению многих целей одновременно.

При выборе используемых в компиляторе оптимизирующих преобразований руководствуются следующими критериями:

- все преобразования должны быть *эквивалентными* (для всех наборов данных, даже неправильных). Эквивалентные преобразования сохраняют семантику исходной программы;
 - “стоимость” преобразования должна быть сопоставима с затрачиваемыми усилиями и полученными эффектами. Время компиляции от включения оптимизирующих преобразований всегда растет, но иногда это не очень важно. Важнее, чтобы при оптимизации не вносились дополнительные ошибки, на исправление которых затрачиваются дополнительные усилия;
 - в результате преобразований программы в среднем должны “улучшаться” (почти для всех допустимых данных), лишь на каких-то (редко встречающихся) комбинациях данных допускается обратный эффект (ухудшение характеристик).
-
- *Машинно-независимая оптимизация*, то есть проведение преобразований исходной программы (в форме некоторого внутреннего представления), не зависящих от выходного языка компилятора (без учета конкретных свойств объектной машины). Эта оптимизация обычно выполняется на специально выделенной фазе компиляции (пометка II на рисунке),
 - *Машинно-зависимая оптимизация*, то есть преобразование программы на выходном языке компилятора. Этот вид оптимизации обычно проводится одновременно с генерацией объектной программы или уже после этой генерации на дополнительной стадии (пометка III на рисунке).

Обычно оптимизирующим преобразованиям подвергается внутреннее представление программы, а не текст на исходном языке. Во-первых, операции, необходимые для реализации высокоуровневых операций становятся на языках внутреннего представления программ более явными, что облегчает их обнаружение и оптимизацию. Например, исходный оператор $s = s + a[i] * b[i]$ скрывает, что вычисление адресов для элементов $a[i]$ и $b[i]$ содержит общие подвыражения $sizeof(\text{тип}) * i$. Во-вторых, внутреннее представление может оказаться относительно независимым от объектной машины, что делает оптимизатор достаточно устойчивым к изменениям при переносе на другую машину.

3.3.4.1. Машинно-независимая оптимизация

Основные преобразования машинно-зависимой оптимизации выполняются для отдельных выражений, линейных участков программ, циклов, вызовов процедур и функций.

Оптимизация однократно выполняемых участков программы практически не оказывает влияния на быстродействие программы и может сказываться только на объеме занимаемой программой памяти, поэтому наиболее тщательно всегда оптимизируется самый внутренний цикл программы. Для проведения оптимизации программы делят на **линейные участки**, то есть на *выполняемые по порядку последовательности операций*. Линейные участки имеют *один вход и один выход*. Линейные участки имеются в любой программе и чаще всего содержат последовательности вычислений, состоящие из арифметических выражений и операторов присваивания значений переменным программы. Ни одна операция линейного участка не может быть выполнена большее число раз, чем смежные с нею операции. Для **линейных участков** проводятся следующие преобразования:

- вычисление выражений из констант на стадии компиляции,
- арифметические преобразования,
- устранение общих подвыражений (избыточных вычислений),
- удаление ненужных присваиваний и других операций, распространение копий значений,
- перестановка независимых смежных участков программ,
- удаление недостижимых фрагментов программы,
- оптимизация вычисления логических выражений.

Арифметические преобразования. Компилятор может изменять характер и порядок следования операций на основании известных алгебраических и логических тождеств, например, заменять выражение $A=B*C+B*D$ выражением $A=B*(C+D)$. Некоторые операции могут заменяться более “простыми”, что делает их выполнение более эффективным:

$$\begin{array}{ll} x := y ** 2 & \Rightarrow x := y * y; \\ x := y * 2 & \Rightarrow x := y + y; \end{array}$$

Устранение общих подвыражений (избыточных вычислений). Операция линейного участка может оказаться избыточной, если ранее на этом же линейном участке уже выполнялась идентичная операция, и никакой операнд данной операции не был изменен в промежутке между двумя идентичными операциями.

Удаление ненужных присваиваний и других операций. Если на некотором линейном участке между двумя операциями присваивания какой-либо переменной значений (одинаковых или разных, не имеет значения) не было ни одного оператора, в котором использовалось бы первое значение переменной, это присваивание является бесполезным и может быть удалено из программы без изменения ее смысла.

Иногда по наличию операторов присваивания одним переменным значений других переменных удается исключить использование некоторых переменных, заменив

их использованием их копий (метод носит название “распространение копий”). В таких случаях происходит как экономия времени исполнения программ, так и экономия памяти, отведенной под хранение данных программы. Например, после присваивания $f := g$ можно вместо переменной f использовать переменную g , а присваивание просто исключить из программы. В наилучшем случае переменная f совсем станет ненужной, значит, и память для нее тоже распределять не придется. Подобные преобразования становятся особенно актуальными при компиляции автоматически сгенерированных программ, работающих с многочисленными переменными и “цепными” присваиваниями

Перестановка независимых смежных участков программ. Иногда компиляторам удается таким образом переставить следующие друг за другом операции, что без изменения смысла программы удастся применить какие-либо другие преобразования. Например, имея выражение $A = 2 * B * 3 * C$, можно преобразовать его перестановкой в $A = (B * C) * (2 * 3)$, а затем вычислить значение подвыражения из констант. Даже если выражение из констант получить не удастся, перестановка операций может привести к экономии временных переменных, которые порождаются компилятором для хранения промежуточных результатов вычислений. Например, непосредственное вычисление выражения $A = (B + C) + (D + E)$ может потребовать, по крайней мере, одной временной переменной для хранения промежуточного результата сложения B и C . Если же провести перестановку операций, эта переменная будет не нужна, а результат останется прежним: $A = ((B + C) + D) + E$.

В некоторых случаях перестановка операций может приводить к потере точности вычислений. Некоторые типы операндов могут сделать переупорядочение выражений невозможным. Например, перестановка целочисленных операций в выражении $I / J * K$ может привести к неверному вычислению выражения $10 * 3 / 8$.

Удаление недостижимых фрагментов программы часто требует глобального анализа программы для определения “достижимости”. В основе такого анализа лежат идеи теории графов, связанные с анализом потока управления и потока данных. Простейший вариант – удаление недостижимых фрагментов, выделяемых с помощью препроцессорных операторов.

Оптимизация вычисления логических выражений. Особенностью оптимизации логических выражений является то, что для получения их значений не всегда требуется проводить вычисление всего выражения до конца. Иногда по результату первых же операций можно заранее определить окончательный результат. Например, операцию логического сложения можно не проводить, если известно, что один из ее операндов имеет значение “истина”. Если это разрешается правилами языка, компиляторы так строят внутренние представления логических выражений, чтобы их вычисления прекращались сразу же, как только значение всего выражения становится предопределенным. Аналогичные рассуждения относятся и к арифметическим выражениям, но умножения на 0 встречаются гораздо реже, чем логическое “И” со значением “ложь”.

Оптимизация передачи параметров и вызовов функций проводится на основе двух подходов: прямой подстановки тел функций в основной текст программы и передачи параметров не с помощью общего стекового механизма, а через глобальные переменные, которые впоследствии связываются с регистрами центральных процессоров.

Прямая подстановка функций в основной цикл программы может привести к существенному увеличению скорости работы программы, но одновременно и к увеличению размеров программы (если функция вызывается из нескольких разных мест). Этот метод приводит к сокращению времени на передачу параметров и возвращаемого результата, на команды передачи управления, захвата памяти в стеке и другие вспомогательные операции. Некоторые языки (Си++) разрешают программистам явно указывать функции, которые желательно реализовать подстановкой (*inline*).

Передача параметров через регистры процессора относится к машинно-зависимой оптимизации. Отказ от универсального стекового механизма в определенных случаях может приводить к значительному снижению времени работы программы, но передача параметров через регистры зависит от количества доступных регистров в вычислительной системе и от используемого в компиляторе алгоритма их распределения.

Оптимизация циклов. Циклом в программе называется любая последовательность участков программы, которая может выполняться повторно. Циклы необязательно должны оформляться с помощью операторов цикла, и чтобы их обнаружить, используется граф управления программы. Обычно циклы содержат в себе один или более линейных участков, где производятся вычисления, поэтому для них могут применяться все методы оптимизации линейных участков. Для оптимизации циклов разработаны и специальные методы:

- вынесение инвариантных вычислений из тела цикла,
- замена операций с переменными цикла,
- слияние, расщепление и разворачивание циклов.

Замена операций с переменными цикла производится на основе понимания того, что с каждым шагом цикла значение переменной цикла меняется на один “шаг цикла”. Многие алгоритмы устроены так, что в ходе вычислений некоторые величины оказываются пропорциональными номеру итерации. При этом в программах производится умножение этих величин на значение переменной цикла. Такие переменные называются индуктивными. Все такие переменные заменяются компилятором на одну, значения других вычисляются с помощью коэффициентов. Например, последовательность операторов

Слияние и разворачивание циклов – это два различных варианта преобразований: слияния двух смежных или вложенных циклов в один и замена цикла на последовательность операций (часто линейную). Слияние смежных циклов с независимыми внутренними операторами S_1 и S_2 позволяет снизить накладные расходы на организацию циклической работы:

Расщепление цикла может оказаться полезным в случае наличия в теле цикла условных операторов:

$$\left. \begin{array}{l} \text{for } (i = 0; i < n; i++) \\ \quad \left\{ \begin{array}{l} \text{if } (x < y) \quad \{ S_1; \} \\ \text{else} \quad \{ S_2; \} \end{array} \right\} \end{array} \right\} \quad \begin{array}{l} \text{if } (x < y) \\ \quad \text{for } (i = 0; i < n; i++) \{ S_1; \} \\ \text{else} \quad \text{for } (i = 0; i < n; i++) \{ S_2; \} \end{array}$$

Развертывание цикла позволяет в определенных случаях уменьшить число проверок условий завершения цикла, а также создать предпосылки для последующего распараллеливания выполнения операций:

$$\left. \begin{array}{l} \text{for } (i = 0; i < n; i++) \\ \quad \{ A[i] = B[i] * C[i]; \} \end{array} \right\} \quad \begin{array}{l} \text{for } (i = 0; i < n; i += 2) \\ \quad \{ A[i] = B[i] * C[i]; \\ \quad \quad A[i+1] = B[i+1] * C[i+1]; \} \end{array}$$

3.3.4.2. Машинно-зависимая оптимизация

Машинно-зависимые методы оптимизации ориентированы на конкретную архитектуру вычислительной системы, то есть на совокупность аппаратных и программных составляющих, а также взаимосвязи между ними. Некоторые аспекты методов машинно-зависимой оптимизации имеют общий характер и применяются многими разработчиками. К таким аспектам относятся:

- учет регистровой структуры вычислительной аппаратуры,
- удаление излишних команд,
- оптимизация потока управления и удаление недостижимых участков программ,
- снижение “стоимости” программы,
- использование машинных идиом,
 - слияние, дробление и развертывание циклов, иногда требующееся из-за технических особенностей аппаратуры,
- учет векторных и конвейерных свойств архитектуры.

Одним из важнейших аспектов является учет распространенной особенности многих вычислительных архитектур, строящихся на программно доступных регистрах. Среди этих регистров одни могут быть специально выделены для выполнения определенных задач (управление стеком), а другие представляют собой *регистры общего назначения*. Выполнение операций над регистрами производится существенно быстрее, чем над элементами памяти, к тому же часто над элементами памяти, кроме операций пересылки, вообще нельзя выполнять никаких операций, а требуется предварительная загрузка их содержимого на регистры. Все это ставит перед разработчиками почти всех компиляторов **задачи распределения регистров и оптимизации их использования**. Эта задача в общем случае является NP-полной, но в каждом конкретном случае удается найти приемлемое решение.

Фазы синтеза программ. Второй главной работой компилятора является генерация результирующей программы. На выходе компилятора должна быть построена последовательность символов (“цепочка”) выходного языка по тем правилам, которые предлагаются языком машинных команд или языком ассемблера. В

случае машинных команд распознавателем этой последовательности символов будет выступать целевая вычислительная система, для которой создается результирующая программа.

Распределение памяти и регистров. Даже в ассемблерах можно встретить фрагменты, выполняющие в том или ином виде распределение памяти и регистров. Тем более, подобные действия по формированию зон или блоков памяти, определению смещений в этих зонах, приписке регистров некоторым элементам данных, необходимы в компиляторах. При проведении таких действий производится компоновка данных в блоки, выравнивание данных на границы физических элементов памяти (байтов, слов, страниц), а также по регистрам специального назначения (векторным регистрам, регистрам устройства работы с вещественными числами). Более детально распределение памяти рассматривается в разделе 3.3.5.

Генерация команд и машинно-зависимая оптимизация. Этап генерации команд (кода) проводится в компиляторах, однако иногда этот этап присутствует и в интерпретаторах. На этом этапе производится окончательное преобразование внутреннего представления транслируемой программы к записи на машинном языке или на языке ассемблера.

Работа генератора не зависит от того, имеется ли перед фазой генерации дополнительная фаза оптимизации, или она отсутствует. Математически проблема генерации эффективной объектной программы является неразрешимой, но на практике применяются достаточно эффективные эвристические подходы, дающие хорошие результаты, хотя и не абсолютно оптимальные.

На вход генератора кода поступает внутреннее, промежуточное представление компилируемой программы, полученное на начальных стадиях компиляции, осуществляющих анализ исходной программы и построение информационных таблиц компилятора. Обычно таблица информации используется генератором кода для определения тех адресов объектов программы, которые они будут иметь в процессе выполнения программы. Так как перед началом генерации кода уже выполнен синтаксический разбор, отсеяны очевидно ошибочные конструкции и получено полное внутреннее представление исходной программы, во внутреннем представлении программы эти объекты обозначаются специально построенными внутренними именами, по которым записи об объектах легко отыскать в информационных таблицах, а не теми именами, которые присвоил им программист.

12. Построение детерминированного конечного автомата по регулярному выражению.

Пусть T - конечный алфавит. **Регулярное множество** в алфавите T определяется рекурсивно следующим образом (знаком ' $\in$ ' будем обозначать принадлежность множеству, знаком ' $\subseteq$ ' включение):

1. $\{\}$ (пустое множество) - регулярное множество в алфавите T ;
2. $\{a\}$ - регулярное множество в алфавите T для каждого $a \in T$;
3. $\{e\}$ - регулярное множество в алфавите T (e - пустая цепочка);
4. если P и Q - регулярные множества в алфавите T , то таковы же и множества
 - $P \cup Q$ (объединение),
 - PQ (конкатенация, т.е. множество pq , $p \in P$, $q \in Q$),
 - P^* (итерация: $P^* = \{e\} \cup P \cup PP \cup \dots$);
5. ничто другое не является регулярным множеством в алфавите T .

Итак, множество в алфавите T регулярно тогда и только тогда, когда оно либо $\{\}$, либо $\{e\}$, либо $\{a\}$ для некоторого $a \in T$, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации. Приведенное выше определение регулярного множества одновременно определяет и форму его записи, которую будем называть регулярным выражением. Для сокращенного обозначения выражения PP^* будем пользоваться записью P^+ и там, где это необходимо, будем использовать скобки. В этой записи наивысшим приоритетом обладает операция $*$, затем конкатенация и, наконец, операция $\cup$, для записи которой иногда будем использовать значок '|'.
'|'

Детерминированный конечный автомат (ДКА) - это пятерка $M = (Q, T, D, q_0, F)$, где

1. Q - конечное множество состояний;
2. T - конечное множество допустимых входных символов;
3. D - функция переходов, отображающая множества $Q \times T$ в множество Q и определяющая поведение управляющего устройства;
4. $q_0 \in Q$ - начальное состояние управляющего устройства;
5. $F \subseteq Q$ - множество заключительных состояний.

Работа конечного автомата представляет собой некоторую последовательность шагов, или тактов. Такт определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в данный момент входной головкой. Сам шаг состоит из изменения состояния и сдвига входной головки на одну ячейку вправо.



Текущее состояние управляющего устройства, символ под головкой и цепочка символов вправо от головки называются конфигурацией автомата. Конфигурация (q_0, w) называется начальной, а пара (q, e) , где $q \in F$, называется заключительной (или допускающей).

Такт автомата M представляется бинарным отношением $|$ -, определенным на конфигурациях: отношение имеет место, если есть переход из конфигурации (q_1, w_1) в конфигурацию (q_2, w_2) . Отношения $|$ -+ и $|$ -* - это, соответственно, транзитивное и рефлексивно-транзитивное замыкание отношения $|$ -. Говорят, что автомат M допускает цепочку w , если $(q_0, w)|$ -(q, e) для некоторого $q \in F$. Языком, допускаемым (распознаваемым, определяемым) автоматом M , (обозначается $L(M)$), называется множество входных цепочек, допускаемых автоматом M . Т.е.

$$L(M) = \{w \mid w \in T^* \text{ и } (q_0, w)|$$

Построение детерминированного конечного автомата по регулярному выражению.

К регулярному выражению (сокращенно РВ) r добавим маркер конца: $(r)\#$. После построения ДКА для расширенного РВ легко построить ДКА для исходного РВ: все состояния ДКА из которых есть переход в конечное с чтением символа "#", можно считать конечными, а символ "#" и соответствующие переходы удалить.

Представим РВ в виде дерева, листья которого - терминальные символы, а внутренние вершины - операции "." (конкатенации), "U" (объединение), "*" (итерация). Каждому листу дерева (кроме ϵ -листьев) припишем уникальный номер и ссылаться на него будем, с одной стороны, как на позицию в дереве и, с другой стороны, как на позицию символа, соответствующего листу.

Теперь, обходя дерево T сверху-вниз слева-направо, вычислим четыре функции: nullable, firstpos, lastpos и followpos. Функции nullable, firstpos и lastpos определены на узлах дерева, а followpos - на множестве позиций. Значением всех функций, кроме nullable, является множество позиций. Функция followpos вычисляется через три остальные функции. Функция firstpos(n) для каждого узла n синтаксического дерева регулярного выражения дает множество позиций, которые соответствуют первым символам в подцепочках, генерируемых подвыражением с вершиной в n. Аналогично, lastpos(n) дает множество позиций, которым соответствуют последние символы в подцепочках, генерируемых подвыражениями с вершиной n. Для узлов n, поддеревья которых (т.е. дерево, у которого узел n является корнем) могут породить пустое слово, определим nullable(n)=true, а для остальных узлов false.

узел n	nullable(n)	firstpos(n)	lastpos(n)

листе	true	0	0
-----+-----+-----+-----			
листе i	false	{i}	{i}
-----+-----+-----+-----			
U	nullable(a)	firstpos(a)	lastpos(a)
/ \	or	U	U
a b	nullable(b)	firstpos(b)	lastpos(b)
-----+-----+-----+-----			
.	nullable(a)	if nullable(a)	if nullable(b)
/ \	and	then firstpos(a)	then lastpos(a)
		U firstpos(b)	U lastpos(b)
a b	nullable(b)	else firstpos(a)	else lastpos(b)
-----+-----+-----+-----			
*			
	true	firstpos(a)	lastpos(a)
a			

Слева от каждой вершины значение firstpos, справа - lastpos.

Заметим, что эти функции могут быть вычислены за один обход дерева. Если i - позиция, то followpos(i) есть множество позиций j таких, что существует некоторая строка ...cd..., входящая в язык, описываемый РВ, такая, что i - соответствует этому вхождению c , а j - вхождению d . Функция followpos может быть вычислена также за один обход дерева по следующим двум правилам

1. Пусть n - внутренний узел с операцией "." (конкатенация), a, b - его потомки. Тогда для каждой позиции i , входящей в lastpos(a), добавляем к множеству значений followpos(i) множество firstpos(b).
2. Пусть n - внутренний узел с операцией "*" (итерация), a - его потомок. Тогда для каждой позиции i , входящей в lastpos(a), добавляем к множеству значений followpos(i) множество firstpos(a).

Прямое построение ДКА по регулярному выражению.

Будем строить множество состояний автомата Dstates и помечать их. Состояния ДКА соответствуют множествам позиций. Начальным состоянием будет состояние firstpos(root), где root - вершина синтаксического дерева регулярного выражения, конечными - все состояния, содержащие позиции, связанные с символом "#". Сначала в Dstates имеется только одно непомеченное состояние firstpos(root).

Сначала в Dstates имеется только одно непомеченное состояние firstpos(root).

```
while (в Dstates есть непомеченное состояние R)
{пометить R;
  for (каждого входного символа a, такого, что в R
    имеется позиция, которой соответствует a)
    {пусть символ a в R соответствует позициям
       $p_1, \dots, p_i$ , и пусть  $S = \bigcup_i \text{followpos}(p_i)$ ;
      if (S не пусто и S не принадлежит Dstates)
        добавить непомеченное состояние S в Dstates
        (рис. 2.9);
      Функцию перехода Dtran для R и a определить как
      Dtran(R, a) = S;
    }
}
```

Алгоритм 2.3. Построение ДКА с минимальным числом состояний.

Шаг 1. Строим начальное разбиение Π множества состояний из двух групп: заключительные состояния Q и остальные $Q-F$.

Шаг 2. Применяем к Π следующую процедуру и получаем новое разбиение Π_{new} (рис. 2.12):

```

for (каждой группы  $G$  в  $\Pi$ )
    {разбиваем  $G$  на подгруппы так, чтобы
      состояния  $s$  и  $t$  из  $G$  оказались в одной
      группе тогда и только тогда, когда для каждого
      входного символа  $a$  состояния  $s$  и  $t$  имеют
      переходы по  $a$  в состояния из одной и той же
      группы в  $\Pi$ ;
      заменяем  $G$  в  $\Pi_{\text{new}}$  на множество всех
    }

```

29

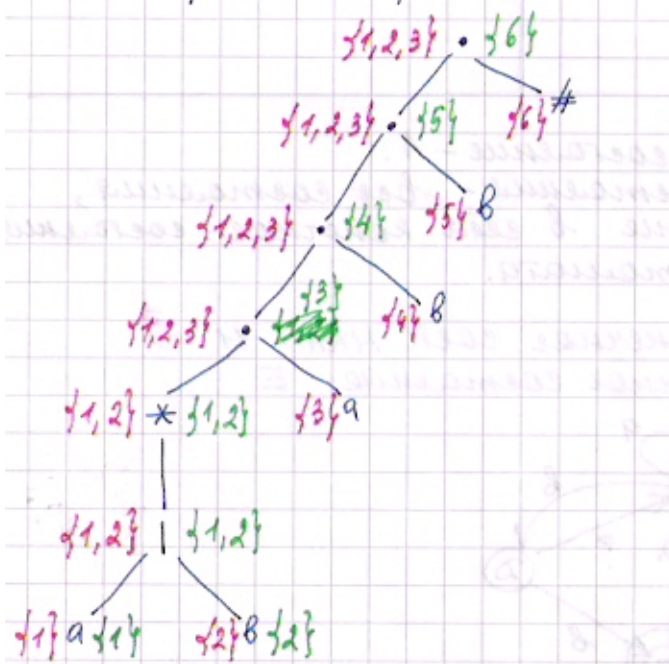
3. Построение ДКА по РВ.

1. ~~нужно~~ Добавляем в конец РВ #.

2. Нумеруем позиции (и # тоже).

Пример: $((a|b)^*abbb)^{\#}$
 1 2 3 4 5 6

3. Строим дерево.



i	$followpos(i)$
1	1, 2, 3
2	1, 2, 3
3	4
4	5
5	6
6	$\emptyset$

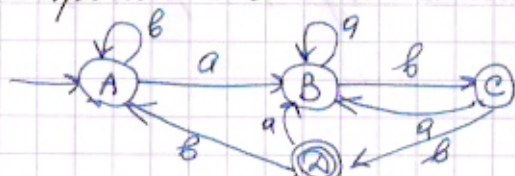
	$a \{1, 3\}$	$b \{2, 4, 5\}$
$\rightarrow A \{1, 2, 3\}$	$\{1, 2, 3, 4\} B$	A
$B \{1, 2, 3, 4\}$	B	$C \{1, 2, 3, 5\}$
$C \{1, 2, 3, 5\}$	B	$D \{1, 2, 3, 6\}$
$*D \{1, 2, 3, 6\}$	B	A

Заполняем таблицу: $A + a = \cup followpos(i)$,
где $i = A \cap a$

в нашем примере:

$$\{1, 2, 3\} \cap \{1, 3\} = \{1, 3\} \rightarrow \{1, 2, 3\} \cup \{4\} = \{1, 2, 3, 4\}$$

7. Строим автомат по таблице.



4. Определяем $firstpos(i) = \#$ первый вору. символ
перехода i

$$= \begin{cases} i, & \text{если } i - \text{мис} \\ \cdot firstpos(\text{лев}) \cup firstpos(\text{прав.}), & \text{если } i = | \\ \cdot \text{если nullable}(\text{лев}), \text{ то} \\ firstpos(\text{лев}) \cup firstpos(\text{прав.}), & \text{если } i = \cdot \\ \text{иначе } firstpos(\text{лев.}) \\ \cdot firstpos(\text{мис}), & \text{если } i = * \end{cases}$$

а вообще здравый смысл и внимательность
тот же алгоритм, но алгоритм еще учесть надо

$nullable(i)$ - обращается ли i в левый
когда-нибудь

- $\left\{ \begin{array}{l} \cdot \text{false, если } i = \text{терм. символ} \\ \cdot \text{true, если } i = * \\ \cdot \text{null(лев) or null(прав), если } i = ! \\ \cdot \text{null(лев) and null(прав), если } i = \cdot \end{array} \right.$

опять же зрабый символ не помешает.

Алгоритм по учебнику переписан - стр. 19

$lastpos(i)$ - последний вхожд. символ
поддерева i .

- $\left\{ \begin{array}{l} \cdot i, \text{ если } i - \text{лист} \\ \cdot \text{lastpos(лев)} \cup \text{lastpos(прав)}, \text{ если } i = ! \\ \cdot \text{lastpos(лист)}, \text{ если } i = * \\ \cdot \text{если nullable(прав), то} \\ \quad \text{lastpos(лев)} \cup \text{lastpos(прав)}, \text{ если } i = \cdot \\ \quad \text{иначе lastpos(прав)} \end{array} \right.$

5. Строим таблицу - для каждого i -
 $followpos(i)$ - позиции, которые могут
стать за i

опять же зрабый символ... а вот алгоритм:

- 1) Для операции $a \cdot b$:
для каждой позиции $i \in \text{lastpos}(a)$
добавим в таблицу $\text{firstpos}(b)$
- 2) Для операции a^* :
для каждой позиции $i \in \text{lastpos}(a)$
добавим в таблицу $\text{firstpos}(a)$.

4. Минимизация ДКА.

1. Убавиться от переходов в никуда:
(то есть отсутствующих)

$\delta(q, a) = \emptyset$ - ниль.

Добавим новое состояние q' и будем
переходить в него.

2. Строим множества:

$P_0: \{F\} \quad \{Q \setminus F\}$

← все оставшиеся.
все конечные сост.

Π_1 : если $A \cup B \in$ одному мн-ву и
 $A \xrightarrow{a} \{ \}$ в одной группе
 $B \xrightarrow{a} \{ \}$ в одной группе $\Rightarrow$
 $A \xrightarrow{b} \{ \}$ в одной группе
 $B \xrightarrow{b} \{ \}$ в одной группе

A и B остаются в одном мн-ве.

Повторять до стабилизации.

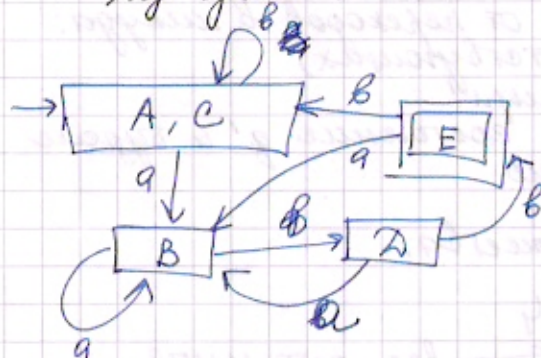
Пример:

	a	b
$\rightarrow A$	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

нет пустых мест
в таблице -
не нужно вводить
новое состояние.

Π_0 : $\{E\}$ $\{A, B, C, D\}$ ← есть переход $D \xrightarrow{b} E$,
 Π_1 : $\{E\}$ $\{D\}$ $\{A, B, C\}$ ← E из другой группы -
 Π_2 : $\{E\}$ $\{D\}$ $\{B\}$ $\{AC\}$ ← отделим D .
 Π_3 : $\{E\}$ $\{D\}$ $\{B\}$ $\{AC\}$ ← есть переход $B \rightarrow D$
 $= \Pi_2$ (остальное в эту
же группу) →
отделим B

3. Строим автомат, обходящее состояние из одного мн-ва



4. + нужно удалить неростижимые состоя-
ния и те, из которых нет пути
в конечное.

13. Построение канонической системы множеств LR(1) ситуаций и таблиц действий и переходов для LR(1) грамматик.

Пусть $G = \langle N, T, P, S \rangle$ - контекстно-свободная грамматика, где N - множество нетерминальных символов, T - множество терминальных символов, P - множество правил вывода и S - аксиома. Будем говорить, что u выводится за один шаг из uAv (и записывать это как $uAv \Rightarrow u'v$), если $A \rightarrow x$ - правило вывода и u и v - произвольные строки из $(N \cup T)^*$. Если $u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_n$, будем говорить, что из u_1 выводится u_n , и записывать это как $u_1 \Rightarrow^* u_n$. Т.е.:

- 1) $u \Rightarrow^* u$ для любой строки u ,
- 2) если $u \Rightarrow^* v$ и $v \Rightarrow^* w$, то $u \Rightarrow^* w$.

Аналогично, " $\Rightarrow^+$ " означает выводится за один или более шагов.

Если дана грамматика G с начальным символом S , отношение $\Rightarrow^+$ можно использовать для определения $L(G)$ - языка, порожденного G . Строки $L(G)$ могут содержать только терминальные символы G . Строка терминалов w принадлежит $L(G)$ тогда и только тогда, когда $S \Rightarrow^+ w$. Строка w называется предложением в G .

Если $S \Rightarrow^* u$, где u может содержать нетерминалы, то u называется сентенциальной формой в G . Предложение - это сентенциальная форма, не содержащая нетерминалов.

Рассмотрим выводы, в которых в любой сентенциальной форме на каждом шаге делается подстановка самого левого нетерминала. Такой вывод называется левосторонним. Если $S \Rightarrow^* u$ в процессе левостороннего вывода, то u - левая сентенциальная форма. Аналогично определяется правосторонний вывод.

Упорядоченным графом называется пара (V, E) , где V обозначает множество вершин, а E - множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид $((v, e_1), (v, e_2), \dots, (v, e_n))$. Этот элемент указывает, что из вершины v выходят n дуг, причем первой из них считается дуга, входящая в вершину e_1 , второй - дуга, входящая в вершину e_2 , и т.д.

Дерево вывода в грамматике $G = \langle N, T, P, S \rangle$ - это помеченное упорядоченное дерево, каждая вершина которого помечена символом из множества $N \cup T \cup \{e\}$. Если внутренняя вершина помечена символом A , а ее прямые потомки - символами $X_1, \dots, X_n$, то $A \rightarrow X_1 X_2 \dots X_n$ - правило этой грамматики.

Упорядоченное помеченное дерево D называется деревом вывода (или деревом разбора) в КС-грамматике $G(S) = \langle N, T, P, S \rangle$, если выполнены следующие условия:

- (1) корень дерева D помечен S ;
- (2) каждый лист помечен либо $a \in T$, либо e ;
- (3) каждая внутренняя вершина помечена нетерминалом;

(4) если N - нетерминал, которым помечена внутренняя вершина и $X_1, \dots, X_n$ - метки ее прямых потомков в указанном порядке, то $N \rightarrow X_1 \dots X_n$ - правило из множества P .

Автомат с магазинной памятью (сокращенно МП-автомат) - это семерка $P = (Q, T, \Gamma, D, q_0, Z_0, F)$, где

(1) Q - конечное множество символов состояний, представляющих всевозможные состояния управляющего устройства;

(2) T - конечный входной алфавит;

(3) Γ - конечный алфавит магазинных символов;

(4) D - функция переходов - отображение множества $Q \times (T \cup \{e\}) \times \Gamma$ в множество конечных подмножеств $Q \times \Gamma^*$, т.е. $d: Q \times (T \cup \{e\}) \times \Gamma \rightarrow \{Q \times \Gamma^*\}$;

(5) $q_0 \in Q$ - начальное состояние управляющего устройства;

(6) $Z_0 \in \Gamma$ - символ, находящийся в магазине в начальный момент (начальный символ);

(7) $F \subseteq Q$ - множество заключительных состояний

Конфигурацией МП-автомата называется тройка $(q, w, u) \in Q \times T^* \times \Gamma^*$, где

(1) q - текущее состояние управляющего устройства;

(2) w - неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если $w = e$, то считается, что вся входная лента прочитана;

(3) u - содержимое магазина; самый левый символ цепочки u считается верхним символом магазина; если $u = e$, то магазин считается пустым.

Такт работы МП-автомата P будем представлять в виде бинарного отношения $|$ -, определенного на конфигурациях. Будем писать $(q, aw, Zu) | (q', w, vu)$ если множество $d(q, a, Z)$ содержит (q', v) , где $q, q' \in Q$, $a \in T \cup \{e\}$, $w \in T^*$, $Z \in \Gamma$, $u, v \in \Gamma^*$.

Начальной конфигурацией МП-автомата P называется конфигурация вида (q_0, w, Z_0) , где $w \in T^*$, т.е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно распознать, а в магазине есть только начальный символ Z_0 . Заключительная конфигурация - это конфигурация вида (q, e, u) , где $q \in F$, $u \in \Gamma^*$.

Говорят, что цепочка w допускается МП-автоматом P , если $(q_0, w, Z_0) |^* (q, e, u)$ для некоторых $q \in F$ и $u \in \Gamma^*$. Языком, определяемым (или допускаемым) автоматом P (обозначается $L(P)$), называют множество цепочек, допускаемых автоматом P . Иногда

допустимость определяют несколько иначе: цепочка w допускается МП-автоматом P , если $(q_0, w, Z_0) |^* (q, e, e)$. Эти определения эквивалентны.

Он состоит из входа, выхода, магазина, управляющей программы и таблицы анализа, которая имеет две части - действий и переходов. Управляющая программа одна и та же для всех анализаторов, разные анализаторы различаются таблицами анализа. Программа анализатора читает символы из входного буфера по одному за шаг. В процессе анализа используется магазин, в котором хранятся строки вида $S_0X_1S_1X_2S_2\ldots X_mS_m$ (S_m - верхушка магазина). Каждый X_i - символ грамматики (терминальный или нетерминальный), а S_i - символ, называемый состоянием. Каждый символ состояния выражает информацию, содержащуюся в магазине ниже него, а комбинация символа состояния на верхушке магазина и текущего входного символа используется для индексации таблицы анализа и определяет решение о сдвиге или свертке.

4.0.2. LR(1)-анализаторы

В названии LR(1) символ L указывает на то, что входная цепочка читается слева-направо, R — на то, что строится правый вывод, наконец, 1 указывает на то, что анализатор видит один символ непрочитанной части входной цепочки.

LR(1)-анализ привлекателен по нескольким причинам:

- LR(1)-анализ — наиболее мощный метод анализа без возвратов типа сдвиг-свёртка;
- LR(1)-анализ может быть реализован довольно эффективно;
- LR(1)-анализаторы могут быть построены для практически всех конструкций языков программирования;
- класс грамматик, которые могут быть проанализированы LR(1)-методом, строго включает класс грамматик, которые могут быть проанализированы предсказывающими анализаторами (сверху-вниз типа LL(1)).

Схематически структура LR(1)-анализатора изображена на рис. 4.8. Анализатор состоит из входной ленты, выходной ленты, магазина, управляющей программы и таблицы анализа (LR(1)-таблицы), которая имеет две части —

функцию действий (Action) и функцию переходов (Goto). Управляющая программа одна и та же для всех LR(1)-анализаторов, разные анализаторы отличаются только таблицами анализа.

Анализатор читает символы на входной ленте по одному за шаг. В процессе анализа используется магазин, в котором хранятся строки вида $S_0X_1S_1X_2S_2\ldots X_mS_m$ (S_m — верхушка магазина). Каждый X_i — символ грамматики (терминальный или нетерминальный), а S_i — символ *состояния*.

Заметим, что символы грамматики (либо символы состояний) не обязательно должны размещаться в магазине. Однако, их использование облегчает понимание поведения LR-анализатора.

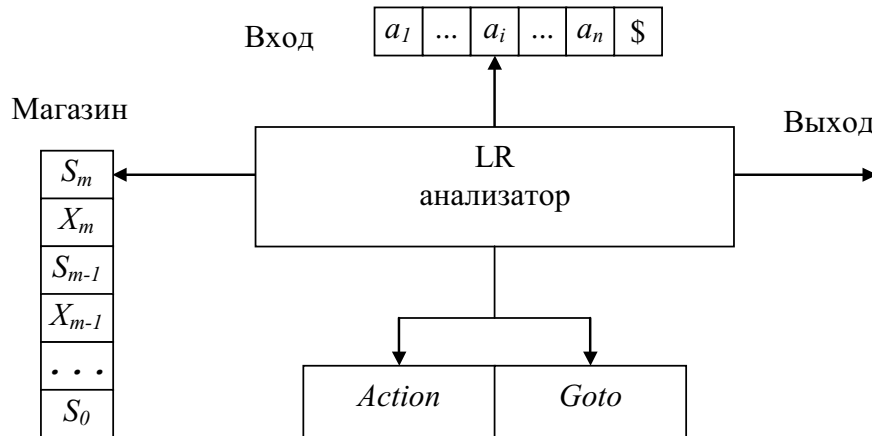


Рис. 4.8.

Элемент функции действий $Action[S_m, a_i]$ для символа состояния S_m и входа $a_i \in T \cup \{\$, \}$, может иметь одно из четырёх значений:

- 1) **shift** S (сдвиг), где S — символ состояния,
- 2) **reduce** $A \rightarrow \gamma$ (свёртка по правилу грамматики $A \rightarrow \gamma$),
- 3) **accept** (допуск),
- 4) **error** (ошибка).

Элемент функции переходов $Goto[S_m, A]$ для символа состояния S_m и входа $A \in N$, может иметь одно из двух значений:

- 1) S , где S — символ состояния,
- 2) **error** (ошибка).

Конфигурацией LR(1)-анализатора называется пара, первая компонента которой — содержимое магазина, а вторая — непросмотренный вход:

$$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$$

Когда анализатор начинает работу, в магазине находится только символ начального состояния S_0 , на входной ленте — анализируемая цепочка с маркером конца.

Каждый очередной шаг анализатора определяется текущим входным символом a_i и символом состояния на верхушке магазина S_m следующим ниже образом.

Пусть LR(1)-анализатор находится в конфигурации

$$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$$

Анализатор может проделать один из следующих шагов:

1. Если $Action[S_m, a_i] = \text{shift } S$, то анализатор выполняет сдвиг, переходя в конфигурацию

$$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$$

То есть, в магазин помещаются входной символ a_i и символ состояния S , определяемый $Action[S_m, a_i]$. Текущим входным символом становится a_{i+1} .

2. Если $Action[S_m, a_i] = \text{reduce } A \rightarrow \gamma$, то анализатор выполняет свёртку, переходя в конфигурацию

$$(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} A S, a_i a_{i+1} \dots a_n \$)$$

где $S = Goto[S_{m-r}, A]$ и r — длина γ , правой части правила вывода.

Анализатор сначала удаляет из магазина $2r$ символов (r символов состояния и r символов грамматики), так что на верхушке оказывается состояние S_{m-r} . Затем анализатор помещает в магазин A — левую часть правила вывода, и S — символ состояния, определяемый $Goto[S_{m-r}, A]$. На шаге свёртки текущий входной символ не меняется. Для LR(1)-анализаторов последовательность символов грамматики $X_{m-r+1} \dots X_m$, удаляемых из магазина, всегда соответствует γ — правой части правила вывода, по которому делается свёртка. После осуществления шага свёртки генерируется выход LR(1)-анализатора, то есть исполняются семантические действия, связанные с правилом, по которому делается свёртка, например, печатаются номера правил, по которым делается свёртка.

Заметим, что функция *Goto* таблицы анализа, построенная по грамматике G , фактически представляет собой функцию переходов детерминированного конечного автомата, распознающего активные префиксы G .

3. Если $Action[S_m, a_i] = \text{accept}$, то разбор успешно завершён.
4. Если $Action[S_m, a_i] = \text{error}$, то анализатор обнаружил ошибку, и выполняются действия по диагностике и восстановлению.

Рассмотрим алгоритм конструирования таблицы, управляющей LR(1) - анализатором.

Пусть $G = (N, T, P, S)$ — КС-грамматика. *Пополненной* грамматикой для данной грамматики G называется КС-грамматика

$$G' = (N \cup \{S'\}, T, P \cup \{S' \rightarrow S\}, S'),$$

то есть эквивалентная грамматика, в которой введён новый начальный символ S' и новое правило вывода $S' \rightarrow S$.

Это дополнительное правило вводится для того, чтобы определить, когда анализатор должен остановить разбор и зафиксировать допуск входа. Таким образом, допуск имеет

место тогда и только тогда, когда анализатор готов осуществить свёртку по правилу $S' \rightarrow S$.

LR(1)-*ситуацией* называется пара $[A \rightarrow \alpha.\beta, a]$, где $A \rightarrow \alpha\beta$ — правило грамматики, a - терминал или правый концевой маркер $\$$. Вторая компонента ситуации называется *аванцепочкой*.

Будем говорить, что LR(1)-ситуация $[A \rightarrow \alpha.\beta, a]$ *допустима* для активного префикса δ , если существует вывод $S \Rightarrow_r^* \gamma A w \Rightarrow_r \gamma \alpha \beta w$, где $\delta = \gamma \alpha$ и либо a — первый символ w , либо $w = e$ и $a = \$$.

Будем говорить, что ситуация *допустима*, если она допустима для какого-либо активного префикса.

Будем говорить, что LR(1)-ситуация $[A \rightarrow \alpha.\beta, a]$ *допустима* для активного префикса δ , если существует вывод $S \Rightarrow_r^* \gamma A w \Rightarrow_r \gamma \alpha \beta w$, где $\delta = \gamma \alpha$ и либо a — первый символ w , либо $w = e$ и $a = \$$.

Будем говорить, что ситуация *допустима*, если она допустима для какого-либо активного префикса.

вывода вида $B \rightarrow q$ имеется вывод $S \Rightarrow_r^* zBbw \Rightarrow_r zqbw$. Таким образом $[B \rightarrow .q, b]$ также допустима для z и ситуация $[A \rightarrow \alpha B.\beta, a]$ допустима для активного префикса zB . Здесь либо b может быть первым терминалом, выводимым из β , либо из β выводится ϵ в выводе $\beta ax \Rightarrow_r^* bw$ и тогда b равно a . То есть b принадлежит $FIRST(\beta ax)$. Построение всех таких ситуаций для данного множества ситуаций, то есть его замыкание, делает приведённая ниже функция **closure**.

Система множеств допустимых LR(1)-ситуаций для всевозможных активных префиксов пополненной грамматики называется *канонической системой* множеств допустимых LR(1)-ситуаций. Алгоритм построения канонической системы множеств приведён ниже.

Алгоритм 4.10. Конструирование канонической системы множеств допустимых LR(1)-ситуаций.

Вход. КС-грамматика $G = (N, T, P, S)$.

Выход. Каноническая система C множеств допустимых LR(1)-ситуаций для грамматики G .

Метод. Выполнить для пополненной грамматики G' процедуру **items**, которая использует функции **closure** и **goto**.

```
function closure(I){ /* I - множество ситуаций */
  do{
    for (каждой ситуации  $[A \rightarrow \alpha.B\beta, a]$  из  $I$ ,
        каждого правила вывода  $B \rightarrow \gamma$  из  $G'$ ,
        каждого терминала  $b$  из  $FIRST(\beta a)$ ,

        такого, что  $[B \rightarrow .\gamma, b]$  нет в  $I$ )
      добавить  $[B \rightarrow .\gamma, b]$  к  $I$ ;
  }
  while (к  $I$  можно добавить новую ситуацию);
  return I;
}
```

```

function goto( $I, X$ ){ /*  $I$  - множество ситуаций;
                                $X$  - символ грамматики */
    Пусть  $J = \{[A \rightarrow \alpha X.\beta, a] \mid [A \rightarrow \alpha.X\beta, a] \in I\}$ ;
    return closure( $J$ );
}

procedure items( $G'$ ){ /*  $G'$  - пополненная
                               грамматика */
     $I_0 = \text{closure}(\{[S' \rightarrow .S, \$]\})$ ;
     $C = \{I_0\}$ ;
    do{
        for (каждого множества ситуаций  $I$  из
                системы  $C$ , каждого символа грамматики  $X$ 
                такого, что goto( $I, X$ ) не пусто
                и не принадлежит  $C$ )
            добавить goto( $I, X$ ) к системе  $C$ ;
    }
    while (к  $C$  можно добавить новое множество
            ситуаций);

```

Если I — множество ситуаций, допустимых для некоторого активного префикса δ , то goto(I, X) — множество ситуаций, допустимых для активного префикса δX .

Работа алгоритма построения системы C множеств допустимых LR(1)-ситуаций начинается с того, что в C помещается *начальное* множество ситуаций $I_0 = \text{closure}(\{[S' \rightarrow .S, \$]\})$. Затем с помощью функции goto вычисляются новые множества ситуаций и включаются в C . По-существу, goto(I, X) — переход конечного автомата из состояния I по символу X .

Алгоритм 4.11. Построение LR(1)-таблицы.

Вход. Каноническая система $C = \{I_0, I_1, \dots, I_n\}$ множеств допустимых LR(1)-ситуаций для грамматики G .

Выход. Функции *Action* и *Goto*, составляющие LR(1)-таблицу для грамматики G .

Метод. Для каждого состояния i функции $Action[i, a]$ и $Goto[i, X]$ строятся по множеству ситуаций I_i :

- (1) Значения функции действия (*Action*) для состояния i определяются следующим образом:
 - а) если $[A \rightarrow \alpha.a\beta, b] \in I_i$ (a — терминал) и $goto(I_i, a) = I_j$, то полагаем $Action[i, a] = \mathbf{shift } j$;
 - б) если $[A \rightarrow \alpha., a] \in I_i$, причём $A \neq S'$, то полагаем $Action[i, a] = \mathbf{reduce } A \rightarrow \alpha$;
 - в) если $[S' \rightarrow S., \$] \in I_i$, то полагаем $Action[i, \$] = \mathbf{accept}$.
- (2) Значения функции переходов для состояния i определяются следующим образом: если $goto(I_i, A) = I_j$, то $Goto[i, A] = j$ (здесь A — нетерминал).
- (3) Все входы в *Action* и *Goto*, не определённые шагами 2 и 3, полагаем равными **error**.
- (4) Начальное состояние анализатора строится из множества, содержащего ситуацию $[S' \rightarrow .S, \$]$.

Таблица на основе функций *Action* и *Goto*, полученных в результате работы алгоритма 4.11., называется *канонической* LR(1)-таблицей. Работающий с ней LR(1)-анализатор, называется *каноническим* LR(1)-анализатором.

4.6.4. LR(1)-грамматики

Если для КС-грамматики G функция $Action$, полученная в результате работы алгоритма 4.11., не содержит неоднозначно определённых входов, то грамматика называется LR(1)-грамматикой.

Язык L называется LR(1)-языком, если он может быть порождён некоторой LR(1)-грамматикой.

Иногда используется другое определение LR(1)-грамматики. Грамматика называется LR(1), если из условий

1. $S' \Rightarrow_r^* uAw \Rightarrow_r uvw$,
2. $S' \Rightarrow_r^* zBx \Rightarrow_r uvu$,
3. $FIRST(w) = FIRST(y)$

следует, что $uAy = zBx$ (то есть $u = z$, $A = B$ и $x = y$).

$FIRST(w) = FIRST(y)$ и $A \rightarrow v$ — последнее правило, использованное в правом выводе цепочки uvw , то правило $A \rightarrow v$ должно применяться и в правом разборе при свёртке uvu к uAy . Так как A даёт v независимо от w , то LR(1)-условие означает, что в $FIRST(w)$ содержится информация, достаточная для определения того, что uv за один шаг выводится из uA . Поэтому никогда не может возникнуть сомнений относительно того, как свернуть очередную правовыводимую цепочку пополненной грамматики.

Можно доказать, что эти два определения эквивалентны.

Дадим теперь определение LR(k)-грамматики.

8. Построение LR(1) анализатора по КС-грамматике.

1. Дополнение грамматики.
добавим правило $S' \rightarrow S$, где S — наш начальный символ.

2. LR-1 ситуации:

$[A \rightarrow \alpha \cdot B\beta, a]$, $A \rightarrow \alpha B\beta$ — правило грам.,
 a — терминал

начальная ситуация: $[S \rightarrow \cdot S, \#]$.

3. Применение замыкания к мнот-ву: (I)

$$\forall [A \rightarrow \alpha. B\beta, a] \in I$$

$$\left\{ \begin{array}{l} \forall B \rightarrow \gamma \\ \forall b \in \text{FIRST}(\beta a) \end{array} \right\} \rightarrow$$

в I добавим правила вида:
 $[B \rightarrow \gamma, b]$

4. Переход к новому множеству:

$$\forall I: A \rightarrow \alpha. B\beta, b \Rightarrow$$

$$\forall I': A \rightarrow \alpha B. \beta, b.$$

5. Строим множества.

Пример:

$$A' \rightarrow A^*$$

$$A \rightarrow BS$$

$$B \rightarrow Aa | e$$

$$S \rightarrow Sb | b$$

$$I_0: \boxed{A' \rightarrow \cdot A^*, \#}$$

начальная ситуация.

замыкание:

можно
списать
в

$$A \rightarrow \cdot BS, \#$$

есть $A \rightarrow BS$, $\text{FIRST}(\#) = \#$

$$B \rightarrow \cdot Aa, b$$

есть $B \rightarrow Aa$, $\text{FIRST}(S\#) = b$

$$B \rightarrow \cdot, b$$

есть $B \rightarrow e$, $\text{FIRST}(S\#) = b$

$A \rightarrow BS, a\#$
но не есть
если путаница

$$A \rightarrow \cdot BS, a$$

есть $A \rightarrow BS$, $\text{FIRST}(ab) = a$

больше ничего нового.

(взять все правила
с буквой A после. и
сдвинуть ее за.)

$$I_1: \boxed{\begin{array}{l} A' \rightarrow A. \cdot, \# \\ B \rightarrow A. a, b \end{array}}$$

замыкание нечего
добавить, нет тер-
миналов сразу за.

$$I_3: \boxed{B \rightarrow Aa. \cdot, b}$$

$$I_4: \boxed{S \rightarrow b. \cdot, \# a b}$$

$$I_5: \boxed{\begin{array}{l} S \rightarrow S. b, \# a b \\ A \rightarrow BS. \cdot, \# a \end{array}}$$

$$I_2: \boxed{\begin{array}{l} A \rightarrow B. S, \# \\ A \rightarrow B. S, a \\ S \rightarrow \cdot Sb, \# \\ S \rightarrow \cdot Sb, a \\ S \rightarrow \cdot Sb, b \\ S \rightarrow \cdot b, a b \# \end{array}}$$

здесь 3 ситуации!
мы их склеим,
чтобы меньше писать

I_6
 $S \rightarrow Sb., ab\#$

В процессе могут появиться
срочковые множества! ~~не~~
у них должны быть свои коды!
не нужно разбивать их на несколько

большее можно
двигать за.

6. Строим таблицу

1) $I_i \rightarrow I_j$
N-нестермимый

добавим в таблицу $Goto(I_i, N) = I_j$

2) $I_i \xrightarrow{a \in T} I_j$

Action $(I_i, a) = \text{Shift } I_j$

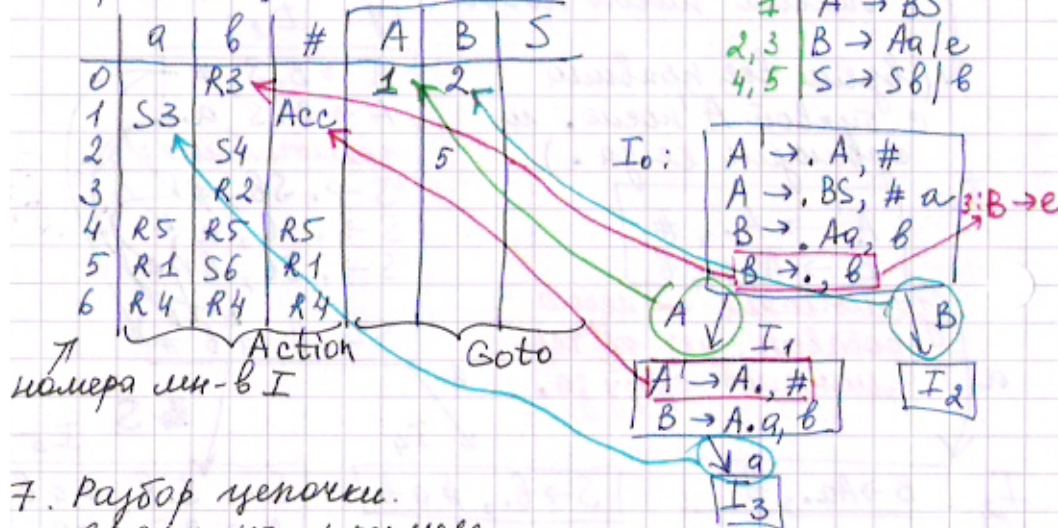
3) в I_i есть ситуация $A \rightarrow d., b$

Action $(I_i, b) = \text{Reduce } A \rightarrow d \rightarrow \text{номер правила}$

4) Reduce $S' \rightarrow S = \text{Акцепт}$

5) все остальное - E-точка.

Пример: терминалов нестерм.



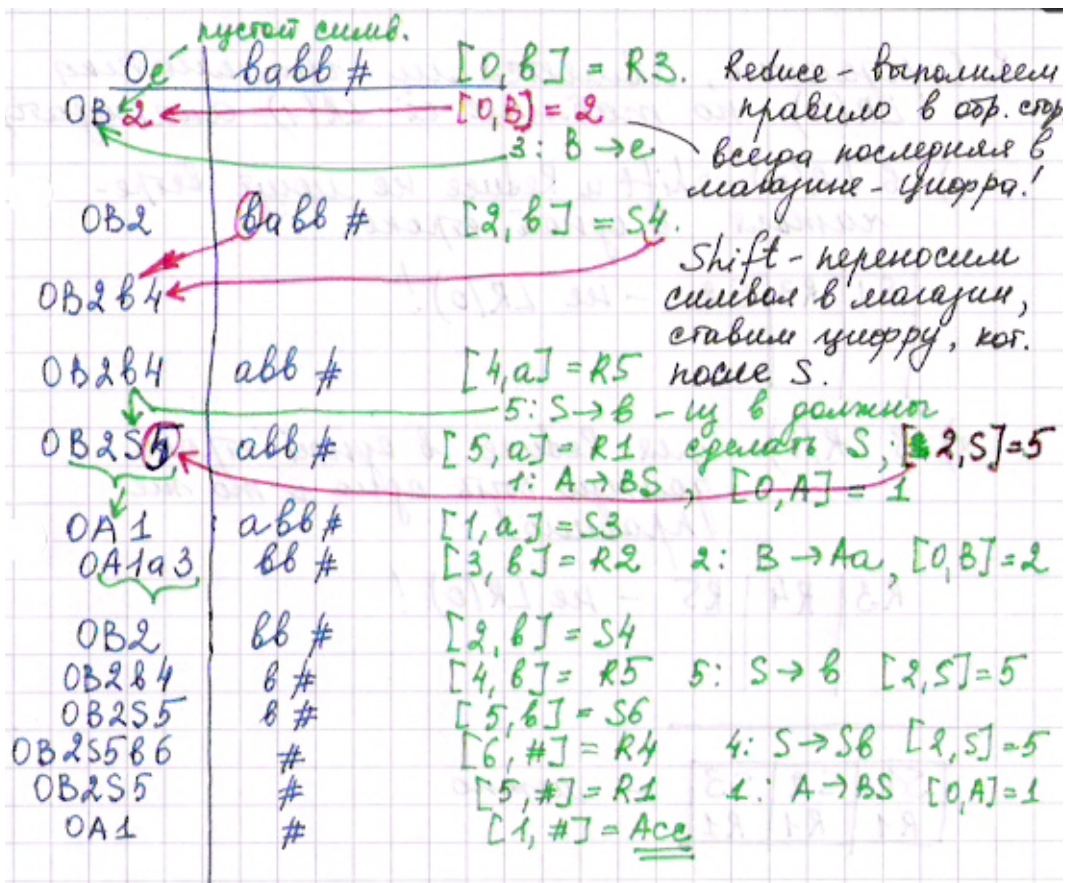
7. Разбор цепочки.

сразу на примере.

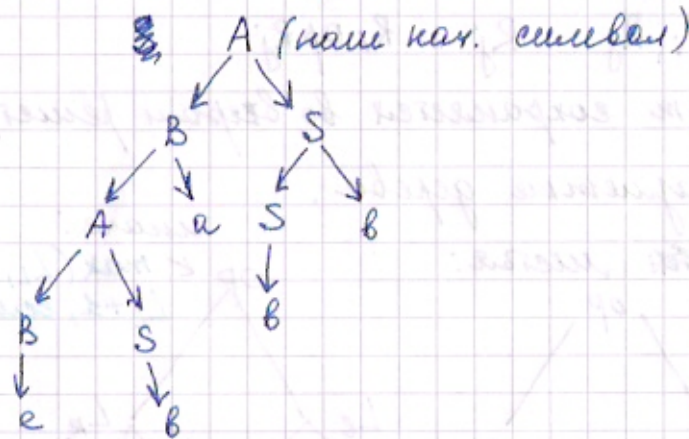
начальное состояние - 0

малю. & строка

$a_1 \dots a_n \#$



применяя правило R... снизу вверх к S, можем получить лек. цепочку



14.