ПРОГРАММИРОВАНИЕ И МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ для астрономов

> Я думаю, что computer science, достигшая зрелого возраста, становится кровосмесительной: людей обучают люди, думающие односторонне. В результате так называемые “ортогонально мыслящие” встречаются все реже и реже...
>
> *Кен Томпсон*

РАЗДЕЛ 1. ВВЕДЕНИЕ В ИНФОРМАТИКУ ДЛЯ АСТРОНОМОВ

Цели и задачи курса. Притча Дейкстры

Данный курс является введением в информатику. *Информатика* – фундаментальная естественная наука, изучающая структуру и свойства информации, а также методы ее обработки, хранения, поиска, передачи и т.д.

Информатика как наука сформировалась в середине прошлого века. Теперь это большая наука и мы коснемся в этом курсе лишь некоторой ее части. Суть предмета, который мы будем изучать, поясним на примере известной притчи Дейкстры, но сначала несколько слов об ее авторе.

Эдсгер Дейкстра (Edsger W. Dijkstra, 11.05.1930–06.08.2002) – один из тех людей, с именем которых связано превращение программирования из шаманства в науку. Работы Э. Дейкстры уже сегодня можно назвать классическими. Одной из форм научной деятельности Дейкстры являлись письма, которые он время от времени посылал своим корреспондентам, призывая распространять их дальше. Сборник, содержащий некоторые из этих писем, был опубликован в 1982 г. Когда взгляды Э. Дейкстры стали известны широкому кругу программистов, они вызвали сильную (и далеко не всегда положительную) реакцию. Теперь приведем полностью перевод письма Дейкстры, содержащего притчу.

Недавно среди старых моих бумаг я нашел следующий рукописный текст. Должно быть я написал его в середине 1973, но не думаю, что по прошествии трех лет его смысл хоть в чем-то изменился. Следовательно, я включаю его в EWD-серию.)

Притча о первом программисте

В незапамятные времена была организована железнодорожная компания. Один из ее руководителей, вероятно малый не промах, обнаружил, что начальные инвестиции могут быть значительно снижены, если снабжать туалетом не каждый железнодорожный вагон, а лишь половину из них. Так и порешили.

Однако вскоре после начала пассажирских перевозок посыпались жалобы. Провели расследование и обнаружили, что причина крайне проста: хотя компания была только что создана, неразберихи уже хватало, и о распоряжении дирекции о туалетах ничего не знали на сортировочных станциях, где все вагоны считались одинаковыми, и в результате в некоторых поездах туалетов почти совсем не было.

Чтобы исправить положение, каждый вагон снабдили надписью, говорящей, есть ли в нем туалет, и сцепщикам было велено составлять поезда так, чтобы около половины вагонов имели туалеты. Хотя это и осложнило работу сцепщиков, но проблему решили и вскоре ответственные за сцепку с гордостью сообщили, что тщательно выполняют новую инструкцию.

Хотя новый порядок сцепки выполнялся неукоснительно, тем не менее неприятности с туалетами продолжались. Новое расследование их причин показало, что хотя действительно половина вагонов в поезде снабжена туалетами, иногда выходит так, что все они оказываются в одной половине поезда. Чтобы спасти дело, были выпущены инструкции, предписывающие чередовать вагоны с туалетами и без них. Это добавило работы сцепщикам, однако, поворчав(!), они и с этим справились.

Жалобы, однако, продолжались. Как оказалось, причина в том, что поскольку туалеты располагаются в одном из концов вагона, расстояние между двумя соседними туалетами в поезде могло достигать трех длин вагонов и для пассажиров с детьми — особенно если коридоры были заставлены багажом — это могло привести к неприятностям. Тогда вагоны с туалетами были снабжены стрелкой, и были изданы новые инструкции, предписывающие, чтобы в каждом поезде(!) все стрелки были направлены в одну сторону. Нельзя сказать, чтобы эти инструкции были встречены на сортировочных станциях с энтузиазмом — количество поворотных кругов(!) было недостаточным, и по правде говоря, мы должны восхищаться тем, что несмотря на существующие стандарты и нехватку поворотных кругов, напрягшись, сцепщики сделали и это.

Теперь, когда все туалеты находились на равных расстояниях, компания была уверена в успехе, однако пассажиры продолжали беспокоиться: хотя до ближайшего туалета было не больше одного вагона, но не было ясно, с какой стороны он находится. Чтобы решить и эту проблему, внутри вагонов были нарисованы стрелки с надписью “ТУАЛЕТ”, сделавшие необходимым правильно ориентировать и вагоны без туалетов.

==

На сортировочных станциях новая инструкция вызвала шок: сделать требуемое вовремя было невозможно(!). В этот критический момент кто-то, чье имя сейчас невозможно установить, заметил следующее. Если мы сцепим вагон с туалетом и без оного так, чтобы туалет был посередине, и никогда их не будем расцеплять, то сортировочная станция будет иметь дело не с N ориентированными объектами, а с N/2 объектами, которые можно во всех отношениях

и со всех точек зрения считать симметричными. Это наблюдение решило проблему ценой двух уступок. Во-первых, поезда могли теперь состоять лишь из четного числа вагонов — недостающие вагоны могли быть оплачены за счет экономии от сокращения числа туалетов, и, во-вторых, туалеты были расположены на чуть-чуть неравных расстояниях. Но кого беспокоит лишний метр?

Хотя во времена, к которым относится наша история, человечество еще не было осчастливлено ЭВМ, неизвестный, нашедший это решение, заслуживает звания первого компетентного программиста.

***

Я рассказывал эту историю в разных аудиториях. Как правило программисты восхищались ею, а менеджеры неизменно становились тем более раздраженными, чем ближе подходил рассказ к концу; настоящим математикам, однако, не удавалось ухватить соль.

EWD. 1976?

Мораль притчи - нужно уметь не только кодировать верно записывать операторы программы), но и правильно подходить к решению задачи в целом, т.е. адекватно выбирать структуры данных и алгоритмы работы с ними.

Литература

1. Д.Кнут. Искусство программирования, том 1. Основные алгоритмы. 3-е изд. М.: Издательский дом "Вильямс", 2000.

2. Н.Вирт. Алгоритмы + структуры данных = программы. М.: Мир, 1985.

3. Н.Вирт. Алгоритмы и структуры данных. СПб: Невский диалект, 2001.

4. Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 2001.

5. http://www.astro.spbu.ru/staff/ilin2/EDU/kurs1.html

1 Информация

1.1 Теория Шеннона

1.1.1 Введение. Сообщение и информация

Информатика была определена нами с использованием понятия информация. Термин *информатика* (от лат. informatio – разъяснение, изложе-

ние) был введен в середине прошлого века Клодом Шенноном применительно к теории связи (передачи кодов) и в настоящее время получил очень широкое распространение.

Строгого определения понятия информация нет, и обычно считается, что информация — это передаваемые сведения, при этом информация (нечто абстрактное) передается от одного объекта другому в виде конкретных сообщений.

Отсюда следует, что:

1. При передаче информации всегда существует объект передающий ее (например, лектор) и принимающий объект (в данном случае слушатель);

2. Информация преобразуется передающим объектом в сообщение (в данном случае в текст).

3. Сообщение, поступившее принимающему объекту (читателю) преобразуется им обратно в информацию.

Сообщение без информации, т.е. само по себе, не существует. Любое сообщение несет информацию (отрицание чего-то в сообщении — тоже информация).

Информация может преобразовываться передающим объектом в сообщения разными способами. Так в нашем случае текст мог быть также произнесен и записан на магнитную ленту. Аналогично и принимающий объект, может получать сообщения по-разному: прочитать написанную текст, воспринять его на слух и т.д.

Как в роли передающего объекта, так и в роли и принимающего объекта может выступать живое существо или прибор. Если передающим и принимающим объектами являются приборы, рассмотрение сообщений и информации не представляет сложностей, т.к. происходящие при этом физические процессы хорошо известны. Сложнее, когда передающим/принимающим объектом является орган чувств живого существа. Но и здесь при передачи информации всегда присутствует физический носитель.

1.1.2 Связь сообщения и информации, их интерпретация и обработка

Связь между информацией и сообщением не является однозначной. Одна и та же информация может передаваться разными сообщениями (например, на разных языках). Сообщение может содержать ненужную информацию. Одно и тоже сообщение может нести разную информацию.

Пример — известная фраза “Над всей Испанией безоблачное небо”, переданная в сообщении о погоде, но означавшая начало военных действий.

Следовательно, для однозначного восприятия информации, объект, передающий сообщение, и объект, ее получающий, должны действовать в соответствии со строго определенными *правилами интерпретации сообщений*, т.е. необходимо определить некоторое взаимно-однозначное отображение α сообщения N в информацию I

$$I = \alpha(N)$$

Например, в фильме “Бриллиантовая рука” сообщения для N, равного фразе “Черт побери”, было определено правило α, согласно которому $\alpha(N)$ означало, что нужно прятать бриллианты в гипс. Непосвященный человек интерпретировал бы эту фразу по-другому.

Обработка сообщений, в случае если один из объектов человек, происходит в коре головного мозга, и процессы происходящие в нем далеко не все изучены. Поэтому упрощенно будем считать, что при передаче информации происходит изменение некой физической величины, называемой *сигналом*. Характеристику сигнала, используемую для представления сообщения, будем называть *параметром сигнала*. Если, например, рассмотреть светофор, то сигнал — световые волн, а параметр сигнала — длина волны (соответствующая цвету — красный, желтый, зеленый).

1.1.3 Дискретные сообщения. Знаки. Алфавит

Сигнал называется дискретным, если параметр сигнала может принимать лишь конечное число значений. Сообщения, переданные с помощью такого сигнала, называются *дискретными сообщениями*.

При обмене сообщениями обычно используются некоторые соглашения относительно их формы. Будем рассматривать *языковую форму*, то есть случай, когда сообщения составлены на некотором языке. Важно, что такие сообщения можно записать на некоторый *носитель информации* и впоследствии воспроизвести.

Языковые сообщения состоят из последовательности знаков. При этом под знаками понимаются не только буквы и цифры.

Знак — элемент некоторого конечного множества, которое называется *набором знаков*.

Набор знаков, в котором определен некоторый порядок, называется *алфавитом*.

Приведем несколько примеров наборов знаков:

1. Десятичные цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

2. Шестнадцатеричные цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

3. Знаки мужского и женского начала: ♀, ♂

4. Знаки планет (и Солнца): ☉, ☿, ♀, ♁, ♂, ♃, ♄, ♅, ♆, ♇

5. Знаки зодиака: ♈, ♉, ♊, ♋, ♌, ♍, ♎, ♏, ♐, ♑, ♒, ♓

6. Фазы Луны: ○, ☾ , ●, ☽

7. Греческие буквы: $\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, o, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega$.

8. Наконец, наборы двоичных знаков (“0”-“1”, “да”-“нет” и т.п.), имеющие в информатике большое значение.

Заметим, что алфавитами не являются наборы 3 и 6 (в них не определен порядок), а также то, что некоторые знаки входят в разные наборы (алфавиты).

Как правило дискретные сообщения из технических соображений либо из соображений чувственного восприятия разбиваются на конечные последовательности знаков — *слова*. В свою очередь каждое слово можно рассматривать тоже как знак, при этом набор знаков, элементы которого слова, будет шире первоначального набора знаков, из которого составлены слова.

1.1.4 Двоичные наборы знаков. Слова. Коды. Символы

Слова над двоичным набором знаков называются двоичными словами. Вообще говоря, они не обязаны иметь постоянную длину (азбука Морзе тому пример, хотя и с некоторой оговоркой), но если длина слов постоянна, то говорят об n-разрядных двоичных словах.

Кодом или *кодировкой* называется правило, описывающее отображение одного набора знаков в другой набор знаков (или слов). Так же называют и множество образов этого отображения. В частности, к таким отображениям относятся *шифры*, и в этом случае образы – это шифровки.

Поясним на примере кодирования десятичных цифр двоичными словами. Здесь мы имеем два набора знаков {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} и {0, 1} и следующее правило отображения первого набора в слова над вторым набором:

0	1	2	3	4	5	6	7	8	9
0	1	10	11	100	101	110	111	1000	1001

Очевидно, при кодировании должны учитываться свойства кодируемых объектов, при кодировании чисел должны учитываться возможные действия с ними, при кодировании букв — их порядок и т.д. Например, если действие, которые предполагается производить с числами, состоит только в их сложении, римские цифры (с некоторой оговоркой) — самое простое представление. Сложение во многих случаях сводится к механическому приписыванию слагаемых: I+II=III.

Знак вместе с его смыслом называется *символом*. Например, знак * часто используется в литературе как символ звездочка, а в математике иногда как символ умножения, в программировании как символ разыменования (C), или квантификатор (регулярные выражения).

1.1.5 Шенноновские сообщения. Количество информации. Теорема кодирования Шеннона

Как же измерить количество информации, содержащееся в том или ином сообщении?

Пусть сообщение состоит из знаков некоторого алфавита и знаки появляются с некоторой вероятностью: p_i – вероятность появления знака Z_i.

Так, например, если сообщение состоит из знаков кириллицы, то частоты появления букв следующие (отн. частота прямопропорциональна вероятности) (Яглом А.М., Яглом И.М. Вероятность и информация. М.: ГИТТЛ, 1957. 160 с.).

Буква	о	е	а	и	т	н	с	р
Отн.частота	0,110	0,087	0,075	0,075	0,065	0,065	0,055	0,048
Буква	в	л	к	м	д	п	у	я
Отн.частота	0,046	0,042	0,034	0,031	0,030	0,028	0,025	0,022
Буква	ы	з	ь,ъ	б	г	ч	й	х
Отн.частота	0,019	0,018	0,017	0,017	0,016	0,015	0,012	0,011
Буква	ж	ю	ш	ц	щ	э	ф	
Отн.частота	0,009	0,007	0,007	0,005	0,004	0,003	0,002	

Естественно, это будет выполняться не во всех сообщениях. Например в сообщении "Саша пошел в шалаш", видно без подсчета, что относительная частота буквы "ш"больше чем 0,007. Но в достаточно больших это будет выполняться.

Сообщения, в которых вероятность появления знака не меняется с течением времени, называются *шенноновскими сообщениями*. Именно такие сообщения мы и будем рассматривать в дальнейшем.

По существу, количество информации – это мера затрат, необходимых для того, чтобы раскодировать все знаки сообщения. Это несколько отличается от интуитивных представлений об информации, но дает удобную для анализа модель.

Наиболее простой алфавит — двоичный, т.е. состоящий из двух знаков (например, “0” и “1” или точка и тире). Количество информации, содержащееся в сообщении, состоящем из одного двоичного знака примем за единицу и будем называть *битом*. Очевидно, что при раскодировании такого сообщения требуются минимальные затраты — следует сделать только один выбор между двумя альтернативными вариантами.

Рассмотрим, как может производиться анализ более сложных сообщений и как при этом измерить затраты (количество информации). Для простоты будем считать, что информация это текст, состоящий только из букв {a, e, f, c, b, d, g}, которые кодируются в сообщениях двоичными словами следующим образом:

a	e	f	c	b	d	g
00	01	100	101	110	1110	1111

Разделим данное множество на два: {a, e} и {f, c, b, d, g}. Полученные подмножества снова разделим надвое: {a} {e} {f, c} {b, d, g}, и для подмножеств, содержащих более одного элемента, будем повторять процедуру деления, пока не получим одноэлементные подмножества. Результаты можно представить в виде дерева, состоящего из подмножеств:

```
      {a, e, f, c, b, d, g}
    {a, e}    {f, c, b, d, g}
   {a}  {e}  {f, c}   {b, d, g}
           {f}  {c}  {b}  {d, g}
                            {d}  {g}
```

Подставим в это дерево, двоичные слова соответствующих знаков:

```
{  00,   01,  100,  101,  110, 1110, 1111}
{  00,   01} {100,  101,  110, 1110, 1111}
  {00}  {01} {100,  101} {110, 1110, 1111}
   a     e   {100} {101} {110}{1110, 1111}
               f     c     b  {1110}{1111}
                                d     g
```

Предположим, что к нам поступило некоторое сообщение состоящее из нулей и единиц:

001011111

и его необходимо раскодировать.

Т.к. первый двоичный знак сообщения – 0, то следовательно отождествлять нужно в множестве { 00, 01} ({a, e}). Из следования следующего знака, а он тоже – 0, вытекает, что отождествление производится при помощи множества {00} ({a}), а это множество одноэлементное и соответствует a. Дальнейшее исследование двоичных знаков сообщения приведет соответственно к отождествлению знаков c и g.

Таким образом, чтобы отождествить знак a, нам потребовалось сделать два шага, т.е. произвести два альтернативных выбора между его подмножествами. а при раскодировании всего сообщения 9 альтернативных выборов (2 для отождествления буквы a, 3 — e и 4 — g).

Из определения единицы информации (бита), видно, что один альтернативный выбор — соответствует 1 биту информации, т.к. каждое множество на шаге отождествление можно рассматривать как один знак и соответственно кодировать его как “0” или “1”. Следовательно, для отождествления знака a, нам потребовалось 2 бита информации. Аналогичные рассуждая, получим следующий результат:

a	e	f	c	b	d	g
2 бита	2 бита	3 бита	3 бита	3 бита	4 бита	4 бита

Оччевидно, что отождествление — поиск определенного слова в множестве из n слов ($n \geq 2$) — всегда можно представить посредством конечного числа следующих друг за другом альтернативных выборов. Вопрос в том, сколько потребуется альтернативных выборов для выбора какого-нибудь определенного знака. Если символ встречается часто, то разумно количество выборов, требующихся для его распознавания, сделать по возможности меньшим. Этого можно достичь, разбивая множество знаков на равновероятные подмножества. В нашем примере это имело бы место при следующих вероятностях появления букв в исходном тексте или, что то же самое, соответствующих двоичных слов в сообщениях:

a	e	f	c	b	d	g
1/4	1/4	1/8	1/8	1/8	1/16	1/16

Наблюдающиеся в реальных случаях вероятности не всегда позволяют разбивать множества точно на равновероятные подмножества. Тем не менее, рассмотрим более детально множества знаков, для которых это возможно. Если i-й знак в них выделяется после k_i альтернативных выборов, то вероятность его появления в сообщениях $p_i = (1/2)^{k_i}$ (например, для знака g $i = 7$, $k_i = 4$, $p_i = 1/16 = (1/2)^4$). И соответственно, наоборот для отождествления знака, вероятность появления которого p_i, требуется $k_i = \log_2(\frac{1}{p_i})$ альтернативных выборов.

По определению количество информации (в битах), даваемой знаком Z_i в сообщении, равно числу выборов k_i, т.е.

$$I_i = \log_2(\frac{1}{p_i}).$$

Среднее количество информации, приходящееся на один знак равно произведению вероятности появления знака p_i на указанное выше количество информации

$$H = \Sigma_i p_i I_i = -\Sigma_i p_i \log_2 p_i$$

Эту величину также называют *энтропией* источника сообщений. Термин "энтропия" был введен Шенноном по совету фон Неймана, который заметил, что формулы, полученные Шенноном для этой величины, совпали с соответствующими формулами для энтропии в физике.

Если каждый выбор представить в виде знака "0" или "1", то отождествлению каждого знака в сообщении будет соответствовать последовательность двоичных знаков, то есть двоичное слово, которое можно рассматривать как кодировку знака. Такие двоичные слова, вообще говоря, имеют разную длину, и знак, вероятность появления которого p_i, кодируется словом длины $N_i = \log_2(1/p_i)$, совпадающей с количеством сделанных альтернативных выборов. Поскольку отсюда следует, что

$$H = \Sigma_i p_i N_i,$$

то становится ясным "физический смысл" H как средней длины слов при двоичном кодировании в рассматриваемом (идеальном) случае.

В нашем пример кодировка 4-разрядными двоичными словами имела бы среднюю длину, равную 4, тогда как используемая кодировка (двоичными числами разной длины) — 2.625 ($H = 1/4^2 + 1/4^2 + 1/8^3 + 1/8^3 + 1/8^3 + 1/16^4 + 1/16^4$), что в 1.5 раза меньше.

Обратимся теперь к более реальной ситуации, когда вероятности появления знаков не позволяют разбить множество на равновероятные подмножества. При кодировании в этом случае код i-го знака имеет некоторую длину $\tilde{N}_i$ и средняя длина двоичного слова равна

$$L\Sigma_i p_i \tilde{N}_i.$$

При известных вероятностях появления знаков p_i ничто не мешает нам вычислить здесь и энтропию источника H по приведенной ранее формуле. Соотношение между энтропией H и средней длиной слова L для любых наборов знаков определяется теоремой Шеннона.

Теорема. 1. Имеет место неравенство $H \leq L$. 2. Всякий источник сообщений можно закодировать так, что разность $L - H$ будет сколь угодно мала.

Разность $L - H$ называется *избыточностью кода.* Если набор знаков можно точно разбить на равновероятные подмножества, то, как мы видели, можно выбрать код с $tildeN_i = N_i$, и следовательно с $L = H$.

Заметим, что в на практике отдельные знаки редко встречаются одинаково часто, и, поэтому кодирование с постоянной длиной кодовых слов, часто применяемое из технических соображений, в большинстве случаев заведомо избыточно.

1.1.6 Обработка сообщений

Очевидно, что всякое правило обработки сообщений можно трактовать, как отображение ν, которое сообщениям N из некоторого множества сообщений S ставит в соответствие новые сообщения N' из множества S'. Сообщения N и N' — это последовательности знаков, которые можно трактовать как последовательность букв, слов или предложений. Таким образом обработку сообщений можно представить как некоторое кодирование.

Чтобы правило служило основой для обработки сообщений, оно должно задавать определенный способ построения сообщения $N' = \nu(N)$. Если множество S велико и задавать отображение ν посредством перечисления всех соответствий неэффективно, то необходимо задать конечное множество операций (элементарных шагов) так, чтобы каждый переход можно было осуществить с помощью конечного числа таких шагов. Таким элементарным шагом может, например, быть следующий

заменить подслово $a_1a_2\ldots a_{k-1}a_ka_{k+1}\ldots a_n$ на $a_1a_2\ldots a_{k-1}ba_{k+1}\ldots a_n$.

Кроме того, нужно задать порядок выполнения элементарных шагов и условия завершения преобразования.

1.2 Простейшие данные

Сведения, информацию, передаваемую компьютеру или получаемую от него, будем называть *данными.* Сначала мы рассмотрим *простейшие типы данных.* Как правило это данные, работа с которыми в той или иной степени поддерживается на уровне инструкций, непосредственно выполняемых процессором. Последнее накладывает определенные ограничения на представление этих данных или, иными словами, на используемые правила интерпретации.

По техническим причинам для представления данных при их хранении и обработке в компьютере используются двоичные знаки “0” и “1”. Напомним, что любую информацию можно закодировать с помощью двоичного алфавита (набора двоичных знаков). При этом, как мы знаем, необходимо установить правила интерпретации. Эти правила будут разными для данных разного типа. Как следствие, одна и та же последовательность двоичных знаков будет интерпретироваться по-разному. Например, слово 0100 0001 может означать и десятичное число 65, и символ латинского алфавита “A”.

1.2.1 Биты. Булевы алгебры. Операции

> “Да будет слово ваше: да, да; нет, нет; а что сверх этого, то от лукавого”
>
> *(Евангелие от Матфея 5,37)*

Первое, что мы рассмотрим – это *биты*. Слово “бит” уже встречалось нам ранее. Оно означало единицу количества информации, содержащейся в сообщении. Теперь же мы сузим это понятие и будем подразумевать под битом объект, который может принимать только два значения “0” или “1”. В англоязычной терминологии эти понятия различают: Bit и bit соответственно. Еще раз подчеркнем, что алфавит, используемый для представления информации, один и тот же (набор двоичных знаков), а вот правила интерпретации для данных разного типа различны. Разными будут также и операции, применяемые к данным того или иного типа. Применительно к битам это логические операции. Свяжем значение “1” с истиной (“TRUE”), а “0” — с ложью (“FALSE”). Над данными этого (логического) типа можно определить следующие операции:

“И” (умножение)	$1\ and\ 1 = 1,\ 1\ and\ 0 = 0,\ 0\ and\ 1 = 0,\ 0\ and\ 0 = 0;$
“ИЛИ” (сложение)	$1\ or\ 1 = 1,\ 1\ or\ 0 = 1,\ 0\ or\ 1 = 1,\ 0\ or\ 0 = 0;$
“НЕ” (дополнение)	$not\ 1 = 0,\ not\ 0 = 1$
“исключающее ИЛИ” (eXcluding OR)	$1\ xor\ 1 = 0,\ 1\ xor\ 0 = 1,\ 0\ xor\ 1 = 1,\ 0\ xor\ 0 = 1.$

Множество элементов с определенными таким образом тремя первыми операциями называются *булевой алгеброй* в честь Джорджа Буля, издавшего в 1854 году сочинение “Исследование законов мысли, на которых основаны математические теории логики и теории вероятностей”.

Булевой функцией является выражение, полученное из ее аргументов (элементов некоторой булевой алгебры) путем сложения, умножения и взятия дополнения. Известно, что для каждой булевой алгебры существует ровно 2^{2n} различных булевых функций n аргументов. В нашем случае возможны 4 булевы функции одного аргумента – две постоянные,

равные 0 и 1, тождественная $f(x) = x$ и единственная нетривиальная функция, представляющая собой операцию “НЕ”. Это легко видеть из следующей таблицы, показывающей аргумент и значения этих функций:

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	1	0	1
1	0	1	1	0

Если рассмотреть булевы функции двух аргументов, то мы найдем только 10 нетривиальных функций. Интересно, что все булевы функции одного и двух аргументов можно выразить через одну булеву функцию. Такой функцией может быть и так называемый штрих Шеффера $x_1'x_2 = not(x_1\, and\, x_2)$ (неверно, что x_1 и x_2), и функция Пирса, равная $(not\, x_1)\, and\, (not\, x_2)$ (ни x_1 и ни x_2). Выражения для основных функций через штрих Шеффера таковы:

$$\begin{aligned} x_1\ and\ x_2 &= (x_1'x_2)'(x_1'x_2), \\ x_1\ or\ x_2 &= (x_1'x_1)'(x_2'x_2), \\ not\ x_1 &= x_1'x_1. \end{aligned}$$

Это означает, что для представления всех логических операций в компьютере достаточно реализовать только одну, например, только штрих Шеффера или функцию Пирса, а использование четырех приведенных выше функций — явная избыточность. Это обусловлено простотой и близостью этих четырех функций человеческому мышлению.

1.2.2 Байты. Символы. Кодирование (отображение одного набора знаков в другой)

Любые данные можно представить в виде последовательности битов. Однако на каждом этапе решения реальной задачи мы должны использовать типы данных, для этого наиболее подходящие. Для элементов булевых алгебр это, конечно, биты, а для описания, скажем, какой-то внесолнечной планеты использование только бит затруднило бы дело. Уровень абстрагирования здесь должен быть много выше.

Предположим, мы обрабатываем тексты, написанные на каком-то языке. Если это английский язык, то соответствующий алфавит должен содержать 62 различных кода (26 для латинских строчных букв, 26 для заглавных и 10 арабских цифр), а также знаки препинания и некоторые другие знаки.

Не все символы равновероятны, а некоторые и вовсе могут отсутствовать в текстах, и чтобы уменьшить избыточность кодирования, было бы

правильно кодировать символы, встречающиеся чаще, меньшим количеством битов (см. подробнее п. 2.1.4).

Однако практически все современные компьютеры “работают” только с данными фиксированного размера, и наименьший размер –8 битов или 1 байт. *Байт* является также наименьшей адресуемой частью памяти компьютера. Если бы адресовался каждый бит, адреса были бы чрезмерно длинными.

Итак, технические требования диктуют нам фиксированный размер кода “символа” (8 битов), и поэтому мы затрачиваем для представления текстовой информации больше битов, чем это необходимо, вынужденно считая, что все символы равновероятны. В этом затрачиется резерв для “сжатия” данных (точнее файлов), которое мы рассмотрим ниже.

Остается только смириться с навязанной “равновероятностью” всех символов. Для представления латинских символов достаточно 7 битов (так называемый ASCII код), остальные 128 значений (возникающие благодаря использованию 8-го бита) применяются для кодирования дополнительных символов латинского алфавита (умляуты и т.п.), символов псевдографики или символов кириллицы.

Для того, чтобы закодировать стандартные и дополнительные символы латинского алфавита и символы кириллицы одновременно 256 символов (8 битов) недостаточно. А ведь есть еще и большое количество иероглифов, букв древних алфавитов и т.д.

Развитие компьютеров привело к возможности выделять под символы 2 байта (16 битов), что позволяет кодировать как минимум 65536 знаков. Первая кодировка, использующая 2 байта, получила название Unicode. При этом для совместимости со старым программным обеспечением были разработаны различные методы представления номеров символов в Unicode: UTF8, UTF16, UTF16LE, UTF16BE. Сейчас разрабатывается четырехбайтовая кодировка UCS-4 (Universal Character Set).

1.2.3 Целые числа. Позиционные системы счисления. Дополнительный и обратный коды. Операции. Переполнение

Вспомним, что такое позиционная система счисления. Обычные (десятичные) числа, как известно, можно записать в виде (для простоты рассмотрим трехзначное число):

$$abc = a \cdot 10^2 + b \cdot 10^1 + c,$$

где $0 \leq a, b, c < 10$, а 10 — это основание системы счисления.

Вместо 10 в качестве основания можно взять любое натуральное число (кроме 1), например 8:

$$127_8 = 1 \cdot 8^2 + 2 \cdot 8^1 + 7 = 87_{10}$$

или 16

$$57_{16} = 5 \cdot 16^1 + 7 = 87_{10}$$

Разумеется, если мы рассматриваем позиционную систему с основанием N, то используются цифры $[0...N-1]$. Так, в шестнадцатеричной системе счисления пятнадцать цифр: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$. Пример не позиционной системы — римские цифры.

Для перевода чисел из одной системы счисления в другую существуют простые правила. Проще всего переводить числа в десятичную систему. Действуем прямо по определению, выполняя все действия по правилам десятичной арифметики:

$$1357_8 = 1 \cdot 8^3 + 3 \cdot 8^2 + 5 \cdot 8^1 + 7 = 512 + 3 \cdot 64 + 5 \cdot 8 + 7 = 512 + 192 + 40 + 7 = 751$$

Для перевода десятичного числа N в систему счисления с основанием r разделить нацело N на r (тоже записанное в десятичной системе), полученное частное снова разделить на r и так делить пока последнее частное не станет равным нулю. Представлением числа N в системе счисления q будет обратная последовательность остатков деления. Можно было бы поступить точно также как и при переводе в десятичную систему, если бы мы знали таблицу умножения, скажем, восьмеричной системы также хорошо, как десятичной. Но нам легче делить в десятичной системе, чем умножать в восьмеричной.

Компьютеры используют двоичную систему счисления. Для этого есть ряд причин.

- Технические устройства с двумя устойчивыми состояниями наиболее надежны;
- Двоичная арифметика намного проще десятичной в реализации на аппаратном уровне;
- Для обработки (для выполнения логических преобразований) информации возможно применение аппарата булевых алгебр.

Заметим, что и в обычной жизни мы используем десятичную систему счисления, потому что привыкли считать по пальцам. Однако десятичная арифметика используется далеко не всегда. В Китае, например,

долгое время пользовались пятеричной системой. Да и сейчас в минуте 60 секунд, а в часе – 60 минут.

Насколько проще было бы нам жить, если бы у нас было восемь пальцев (шестнадцать на руках и ногах). Перевод из двоичной системы, используемой компьютером, в шестнадцатеричную (или восьмеричную) выполняется элементарно (обратный перевод не сложнее). Для этого надо разбить число на тетрады (или триады) и записать каждую тетраду (триаду) соответствующей шестнадцатеричной (или восьмеричной) цифрой, например

```
0100 1000  0101_2 = 408A_16
010 010 000 101_2 = 40205_8
```

Если мы рассматриваем целые числа, то предположение об их равновероятности в большинстве случаях приемлемо. Здесь есть другие ограничения. Как правило архитектура ЭВМ поддерживает представление целых чисел только нескольких фиксированных размеров. Скажем, короткие целые, которые занимают (для Intel) 2 байта, и длинные целые – 4 байта. Впрочем, мы можем ограничимся и одним байтом, в этом случае можно представить только 256 различных чисел. В случае же двух байт мы получаем 65536 различных целых чисел, а в случае четырех – 4294967296. В большинстве случаев этого достаточно, но нетрудно придумать и задачи, где такое ограничение будет весьма существенным.

Итак рассмотрим, однобайтовые целые. Они принимают значения от 00000000_2 до 11111111_2, т.е. от 0 до 255. Если же мы хотим представить и отрицательные числа, то необходимо выделить бит для информации об знаке числа (отрицательное/неотрицательное). Для этого используется самый левый (старший) бит.

Таким образом, при представлении целого числа со знаком можно работать с числами в диапазоне $-128 \ldots 127 (-2^7 \ldots 2^7 - 1)$, если используется один байт; $-32768 \ldots 32767 \, (-2^{15} \ldots 2^{15} - 1)$, если два байта; и $-2147483648 \ldots 2147483647 \, (-2^{31} \ldots 2^{31} - 1)$, если четыре байта. Положительных чисел на одно меньше, чем отрицательных, поскольку ноль здесь считается положительным числом.

Архитектура большинства компьютеров поддерживает две из трех форм кодирования целых чисел со знаком — так называемый прямой код и обратный или дополнительный коды.

Положительные числа представляются во всех трех кодах одинаково, а отрицательные числа имеют разное представление.

Прямой код. В знаковом разряде 1, а в разряды цифровой части помещен двоичный код абсолютной величины.

Обратный код. Двоичный код абсолютной величины числа поразрядно инвертируется (при этом в знаковом разряде получается 1).

Дополнительный код. К обратному коду прибавляется единица.

Как правило отрицательное число преобразуется в нужный код при вводе в компьютер автоматически, и далее именно в таком виде хранится и участвует в операциях. При выводе происходит обратное преобразование.

Рассмотрим операцию сложения двух целых чисел, представленных обратными кодами, на простом примере

$$1 + (-3) = 0000\,0001_2 + 1111\,1100_2 = 1111\,1101_2 = -2$$

Вычитание заменяется сложением с заменой знака второго слагаемого на противоположенный.

Поскольку умножение на 2 эквивалентно смещению цифр двоичного числа на одну позицию влево, умножение реализуется как последовательность сложений и сдвигов. При этом используется аппарат булевых алгебр. Деление обычно происходит путем многократного прибавления к делимому кода делителя.

Очевидно, что представление объектов (т.е. правила интерпретации) выбирается так, чтобы облегчить реализацию в компьютере операций, которые предполагается над ними выполнять. Для целых чисел, кроме операций сравнения (равенство/неравенство, больше/меньше), это рассмотренные выше арифметические операции.

Ограниченное число байтов, выделяемых для представления целых чисел, может привести к тому, что при операциях с ними случится *переполнение* – ситуация, когда полученное в результате целое число нельзя представить, поскольку оно выходит за пределы указанных диапазонов. Например, при вычислении факториалов двухзначных чисел ($9! = 362880$ невозможно представить, используя два байта, а $13! = 6227020800$ — четыре байта). В этих случаях для представления целых чисел применяется вещественный тип данных или моделируется целочисленная арифметика с числами произвольной длины, представленными, например, в виде списка из стандартных целых чисел.

1.2.4 Вещественные числа. Способы представления, стандарт IEEE754

> Бог создал целые числа; все остальные — дело рук человеческих
>
> *Leopold Kronecker*

Если множество целых чисел — счетное, и мы можем представить в том или ином виде ВСЕ целые числа из некоторого фиксированного

диапазона, то с вещественными числами все значительно сложнее, поскольку множество вещественных чисел имеет мощность континуум, и сколько бы мы не выделяли бит под вещественные числа, нам все равно не удастся представить ВСЕ вещественные числа, даже ограничившись малым диапазоном.

Именно поэтому представление целых чисел в архитектурах компьютеров различных производителей с самого начала было практически стандартным (либо дополнительный, либо обратный код), а представление вещественных чисел, или чисел с плавающей точкой, еще пару десятков лет назад было своим практически у каждого производителя. Различались и количество бит, и точность, и алгоритмы округления, и поведение при переполнении. Можно было насчитать более дюжины различных подходов, что значительно затрудняло переносимость программного обеспечения, и сравнение результатов, полученных на компьютерах различных архитектур, особенно это касается научных приложений.

Лишь в 1985 году был принят стандарт IEEE 754 (IEEE — Institute of Electrical and Electronics Engineers), который с тех пор реализуется всеми производителями компьютеров (единственное исключение — компьютеры CRAY).

В стандарте определены три типа форматов вещественных чисел.

- Вещественные числа одинарной точности (4 байта)
- Вещественные числа двойной точности (8 байтов)
- Вещественные числа расширенной точности (10 байтов)

De facto стандартным стал и формат вещественных чисел четверной точности (16 байтов), которые иногда используются в научных расчетах.

Каждый формат определяет также представление для некоторых специальных вещественных чисел: $\pm\infty$ (бесконечность), NaN (Not a Number, т.е. не число). Операции с этими значениями определяются в стандарте следующим образом:

операция	результат
$x/\pm\infty$	0
$\pm\infty\cdot\pm\infty$	$\pm\infty$
$\pm x\ /\ 0\ (x\neq 0)$	$\pm\infty$
$\infty+\infty$	∞
$\pm 0/\pm 0$	NaN
$\infty-\infty$	NaN
$\pm\infty/\pm\infty$	NaN
$\pm\infty\cdot 0$	NaN

Кроме этого, любая операция с NaN дает NaN.

Вещественные числа представляются в такой форме

$$2^{k+1-N}n$$

где k, n — целые ($1 - 2^K < k < 2^K$ и $-2^N < n < 2^N$), а K и N — некоторые фиксированные числа. Например, при $K = 7$, $N = 24$ число $2.625 = 2\frac{5}{8} = \frac{21}{8}$, может быть представлено с $k, n = 20{,}21$; $19{,}42$; $18{,}84$ и т.д. Из этого примера хорошо видно, насколько мала доля вещественных чисел, которые могут быть точно представлены в компьютере.

Неоднозначность представления разрешается так называемой нормализацией, которая выражает вещественное число в виде

$$\pm 2^{e-b}(1+f)$$

где e — целое число, $b = 2^K - 1$ — так называемое *смещение*, f — неотрицательная дробь ($0 < f < 1$). В нашем примере $b = 127$, $e = 128 = 01000000_2$ и $f = \frac{5}{16} = 0.0101_2$.

Стандарт позволяет работать и с ненормализованными числами

$$\pm 2^{1-b}(0+f),$$

если $k = 2 - 2^K$ и $0 < |n| < 2^{N-1}$, что несколько увеличивает возможности представления чисел, близких к нулю.

Итак, согласно IEEE 754 вещественные числа кодируются следующим образом:

$$s\,\underbrace{kkkkkkkk}_{\text{e}}\,\underbrace{nnnnnnnnnnnnnnnnnnnnnnn}_{f}$$

Здесь s — знак числа, $kkkkkkkk$ — $K+1$ битов *порядка* e, $nnnnnnnnnn\,nnnnnnnnnnnnn$ — $N-1$ битов *мантиссы* f. Для чисел одинарной точности $K+1 = 8, N = 24$, для двойной точности $K+1 = 11, N = 53$ и для четверной точности $K+1 = 15, N = 113$.

В формате чисел одинарной точности приняты следующие правила:

e	f	значение
0	0	± 0
0	$\neq 0$	$\pm 2^{-126} f$
от 1 до 254	-	$\pm 2^{e-127}(1+f)$
255	0	$\pm\infty$
255	$\neq 0$	NaN

При $e = 0$ $f \neq 0$ мы имеем дело с ненормализованными числами. Их точность (число значащих цифр мантиссы) меньше, чем нормализованных (e =1–254). Заметим также, что -0 и $+0$ различаются.

При работе с целыми числами была проблема переполнения. Для вещественных чисел эта проблема сохраняется, но отодвигается существенно дальше благодаря возможности явно указывать порядок числа. Если величина представимых целых чисел не может превышать примерно $2^{31} \approx 2 \cdot 10^9$, то величина представимых вещественных чисел может достигать $2^{128} \approx 3 \cdot 10^{38}$ при одинарной точности и $2^{1024} \approx 2 \cdot 10^{308}$ при двойной.

С другой стороны, ограниченное число знаков мантиссы часто приводит к новой проблеме — так называемой *потере точности* представления вещественных чисел при вычислениях. Например, при вычитании близких чисел (таких, скажем, как $10^6 + \frac{2}{3} = 10000000.6(6)$ и $10^6 + \frac{1}{3} = 1000000.3(3)$) результат будет содержать много "придуманных" компьютером цифр (в указанном примере при одинарной точности результат будет примерно следующим 0.33xxxxxx, т.е. истинный результат,равный $\frac{1}{3}$, будет воспроизведен только с точность двух знаков).

1.2.5 Строки и указатели

При представлении таких данных как символ или число мы использовали жестко ограниченное сверху количество байтов. Однако для хранения текстовых строк, очевидно, каждый раз требуется разный размер памяти. В таких случаях применяется *строковый тип данных*, интерпретируемых как последовательности символов.

На аппаратном уровне поддерживаются операции копирования и сравнение на равенство данных этого типа. При реализации программным путем других операций (определение длины строки, выделение подстроки, включение и удаление подстрок и т.п.) следует учитывать ряд особенностей кодирования символов. В добавление к проблеме кодировки кириллицы, упомянутой нами ранее, трудности может вызвать также то, что алфавитный порядок букв может не совпадать с порядком их кодов, а также существование особых букв (букв с ударением или с диакритическими знаками), причем в некоторых скандинавских языках "алфавитный" порядок может зависеть от сочетания букв.

Следует отметит, что в большинстве случаев реальная длина строки оказывается меньше размера памяти, который выделен для ее хранения. Поэтому в конце строки записывается специальный символ "конца строки". В шестнадцатеричном виде он записывается как 0D 0A в системе Windows или как 0D в системе Unix/Linux.

1.2.6 Указатели

Обмен информацией между памятью и арифметическим устройством компьютера происходит не по 1 биту (это очень медленно), а группами битов — словами (и даже группами слов). Каждое слово имеет свой номер — адрес. Таким образом, с каждой адресуемой областью памяти связано два двоичных числа — ее *адрес* и хранимое в ней *значение*. Многие языки программирования позволяют работать с адресами, как с данными, и данные, значения которых интерпретируются как адреса, называются *указателями*.

Операции с указателями либо реализованы аппаратно, либо определены в том или ином языке программирования:

- разыменование — унарная операция, состоящая в получении значения по адресу;
- сложение/вычитание;
- проверка равенства и неравенства;
- получение адреса — операция, обратная разыменванию.

Поясним суть этих операций, используя простой пример. Пусть у нас имеется переменная символьного типа S, размещенная по адресу 534. Кроме этого, в памяти машине, начиная с адреса 1056, хранится строка: “Студент имярек любит информатику."Адрес (начала) строки присвоен переменной типа указатель P (т.е. $P = 1056$).

Рассмотрим следующую операцию разыменования (для примера используем синтаксис Си, где символ этой операции “*”):

```
S = *P
```

В результате выполнения этой операции переменная S будет содержать символ “С”.

Если же будет выполнен оператор

```
S = *(P+3)
```

то S станет равной символу “д”, размещенному в данной строке на три позиции правее символа “С”. Отметим, что результат не зависит, используется ли однобайтовая или двухбайтовая (Unicode) кодировка символов. Компьютер сам “выясняет”, в каких единицах (в байтах или двухбайтовых словах) следует производить сложение адресов.

Рассмотрим теперь операцию получения адреса считая, что две предыдущие операции были выполнены (также используем синтаксис Си, где операция получения адреса обозначается символом “&”),

```
P = &S
```

В результате содержимое указателя P изменится с 1056 на 534 (в переменной P будет теперь находиться адрес ячейки S).

Указатели играют важную роль при представлении динамических структур данных, т.е. структурированных данных, которые могут изменяться во время работы программы.

1.3 Структуры данных

1.3.1 Введение

В предыдущем разделе мы рассмотрели представление данных, названных простейшими, то есть данные, операции с которыми поддерживались на аппаратном уровне[1]. Теперь познакомимся с более сложной организацией данных.

1.3.2 Данные и их обработка. Структура хранения. Операции создания и уничтожения. Доступ к данным

При решении физических, экономических и других задач обрабатываемая информация представляет собой абстракцию интересующей нас части реального мира, и данные задачи — это не просто безликие независимые байты, целые и вещественные числа. Всегда есть определенные *структурные отношения* между элементами данных, и, чтобы эффективно решать даже относительно несложные задачи, важно ясно представить себе как структурные связи, существующие между данными, так и способы представления этих структур в компьютере, и методы работы с различными структурами.

Простейший пример структурированных данных — комплексные числа. Как известно, они состоят из двух взаимосвязанных частей: вещественной части, представимой естественно вещественным числом, и комплексной части, являющейся также вещественным числом.

Другой пример — характеристики объектов какого-то типа. Ясно, что разные параметры одного объекта образуют единую группу. Например, при решении небесно-механических задач для каждой планеты Солнечной системы потребуются задать:

1. Название планеты — строка.

[1]Следует заметить, что существуют компьютеры, в которых поддерживается работа и с более сложными данными.

2. Масса планеты — вещественное число.

3. Элементы орбиты — группа вещественных чисел.

Данные, относящиеся к определенной задаче, состоят из элементарных единиц (или атомов), как правило содержащих простейшие элементы, рассмотренные в предыдущем разделе, или набор таких элементов. Возможные способы, посредством которых элементы данных логически связываются друг с другом, характеризуют различные структуры данных.

Замечание. Различные языки программирования поддерживают далеко не все рассматриваемые здесь структуры. Но ведь структуры данных отражают логическую взаимосвязь данных задачи и поэтому, даже если используемый язык и не поддерживает напрямую нужные структуры, мы, так или иначе, моделируем существующие взаимосвязи с помощью доступных средств. Желательно, конечно, использовать наиболее адекватные средства, но важнее мыслить адекватно.

В реальной задаче структура данных является некоторой, возможно сложной, комбинацией рассмотренных ниже базовых структур.

1.3.3 Простейшие структуры данных

По аналогии с данными, *простейшими структурами* будем называть структуры, операции с которыми в той или иной степени поддерживаются распространными языками программирования, (некоторые из них могут поддерживаться и на аппаратном уровне). Такие структуры обычно используются не только как самостоятельные объекты, но и служат для представления (моделирования) более сложных структур.

Простейшие структуры с элементами разных типов

В терминах Паскаля это *запись*, в Си — *структура* (в узком смысле). Иногда эту структуру называют *объект* или *узел*. Обычно все элементы (или *поля*) такой структуры (записи) связаны с одним объектом решаемой задачи, и каждое поле является какой-то осмысленной единицей информации о нем. Количество полей произвольно. Элементы такой структуры имеют разный тип: целые или вещественные числа, байты, строки и т.д., включая другие структуры (записи).

Важно, что количество и состав полей записи (структуры) не меняются в процессе выполнения программы. Подобная статичность структуры облегчает многие операции, как то: копирование структуры (известен ее

размер), доступ к полям структуры (известно положение каждого поля относительно всей структуры) и т.д.

Рассмотрим возможную структуру данных для объектов из каталога скоплений галактик Абеля (всего около 4000 скоплений; электронная база данных, основанная на этом каталоге, была разработана в Астрономическом институте и доступна по адресу — *http://*). Очевидно, что записи здесь обязательно должны включать следующие поля:

1. N: номер объекта в каталоге (строка: символ A и порядковый номер);

2. $Coords$: координаты скопления (структура):

   - RA: прямое восхождение (вещественное число двойной точности);
   - DE: склонение (вещественное число двойной точности);

3. Red: красное смещение (вещественное число одинарной точности);

4. $Numb$: "богатство" — количество галактик в некотором радиусе (целое число);

5. $S1$: тип галактики по классификации 1 (строка);

6. $S2$: тип галактики по классификации 2 (строка).

Ясно, что подобная структура будет содержать основную информацию об одном объекте — скоплении галактик и состоять из элементов разных типов: строк (N, $S1$, $S2$), целого ($Numb$) и вещественного (Red) чисел и структуры, состоящей из двух вещественных чисел (RA, DE). Заметим, что информация о скоплении галактики не ограничивается только приведенными выше характеристиками. С каждой такой записью могут еще быть связаны записи с информацией об объектах, обнаруженных в этом скоплении: галактиках, звездах, неизвестных объектах (может быть брак на фотографии), и для полного описания этих данных потребуется более сложная структура данных.

Как правило языки программирования позволяют определять такие записи и получать доступ к их отдельным полям. Если, например, Gal — объект описанной выше структуры, то $Gal.Red$ — красное смещение этого объекта, а $Gal.Coords.DE$ — его склонение. Синтаксис, определяющий доступ к отдельным полям, может несколько меняться в зависимости от используемого средства (языка программирования), но это не более, чем

детали, суть же одна, мы перечисляем какое-то подмножество характеристик, относящихся к тому или иному объекту.

Для обсуждаемых структур данных с элементами разных типов в целом обычно определены только операции сравнения и копирования. Операции, которые разрешены для элементов структуры, определяются типом данных, к которому они отнесены, иными словами, если элемент структуры — строка, то с ним возможны все операции, разрешенные для данных строкового типа. Кроме этого, важной операцией является и выбор элемента структуры по его имени.

Массивы, многомерные массивы

С чем-то похожим на *массив* — упорядоченный набор однотипных элементов, число которых фиксировано, — мы уже имели дело при рассмотрении строк (последовательностей символов).

Самый простой массив — одномерный (его иногда называют *вектором.* Элементы массива располагаются в памяти компьютера последовательно, и поскольку они одного типа (или являются простейшими структурами одного вида), то адрес i-го элемента (A_i) может быть получен по формуле

$$A_i = A_{i_1} + sizeof(element) * (i - i_1),$$

где A_{i_1} — адрес первого элемента массива, имеющего в общем случае индекс i_1, $sizeof(element)$ — количество байтов, которое необходимо для размещения одного элемента массива, $sizeof(element) * (i - i_1)$ — "смещение" элемента с порядковым номером i относительно первого элемента.

Рассмотрим, как при помощи одномерного массива можно описать каталог (множество) галактик Абеля и получить i-й элемент. Элементами такого массива будут структуры данных с элементами разных типов. Воспользуемся структурой для скопления, описанной в предыдущем пункте. Порядковый номер скопления (A+порядковый номер) можно исключить из рассмотрения, т.к. он будет соответствовать номеру элемента в массиве. При описании массива (каталога) необходимо задать имя ($Catalog$) и размерность (N) массива. Тогда $Catalog(i)$ — структура, содержащая информацию об i-м скоплении, а $Catalog(i).Coords.DE$ даст склонение i-го скопления. Заметим, что синтаксис в разных языках может различаться.

Операции, которые разрешены для элементов массива, определяются типом данных, к которому они относятся (как и для полей записей). Массив — структура по сути статическая, поэтому, как и в предыдущем случае, операция доступа к элементам массива проста и быстра.

Часто встречаются и массивы большей размерности. Простой пример — любая таблица. В математических расчетах многомерные массивы используются для представления тензонов и матриц, особенно часто появляющихся при решении разнообразных уравнений численными методами.

Вообще говоря, многомерные массивы мало чем отличаются от одномерных. Причем многомерный массив можно легко представить в виде эквивалентного одномерного массива.

Положение элемента многомерного массива задается набором индексов (двумя в случае матриц). Адрес элемента, например, с индексами (i, j) можно получить, зная адрес A_{i_1,j_1} первого элемента, имеющего индексы (i_1, j_1),

$$A_{i,j} = A_{i_1,j_1} + (N_1 * sizeof(element)) * (i - i_1) + sizeof(element) * (j - j_1),$$

где N_1 — размерность массива по первому индексу.

Как правило, языки программирования предоставляют хорошую поддержку для массивов — сами выделяют требуемую память, обеспечивают доступ к запрашиваемому элементу массива. Необходимо только иметь в виду, что располагать в памяти многомерные массивы можно по-разному, изменяя сначала первый индекс, а затем следующие (т.е. по столбцам как в Фортране), либо наоборот (т.е. по строкам как в Си и Паскале).

Хэши и их реализация, хэш-функции, коллизии

Для рассмотрения следующей структуры немного расширим понятие массива. Допустим, мы имеем дело с каталогом малых планет. У этих планет есть имена, представимые строками (например, планета, названная в честь заведующего кафедрой небесной механики “Холшевников”), и хотелось бы иметь возможность по имени планеты получать всю информацию о ней. Существующие имена планет уникальны, и следовательно их представление в компьютере в виде последовательностей “0” и “1” также уникальны. Трактуя последовательность битов в имени планеты как двоичное целое число, можно рассматривать это число как “индекс” элемента массива и использовать данный элемент для хранения информации о планете. Тогда, например, запись вида $Catalog$(“Агекян”) позволила бы получить очень быстрый доступ к информации о малой планете “Агекян” (названной в честь профессора СПбГУ Т.А.Агекяна). Такие индексы называются *ключами*, а подобное обобщение массива — *хэшем*.

Безусловно, использование таких “ассоциативных массивов” (или, как их называет Кнут “памятью с содержательной адресацией”), весьма удобно, но как эффективно реализовать работу с такой структурой? Мы можем использовать одномерный массив, в котором, однако, будет заполнена полезной информацией, только малая (ну, очень малая) часть, для которой индексы совпадают с ключом. Даже на компьютере с очень большой оперативной памятью мы вряд ли сможем разместить небольшой хэш, если будем использовать для его представления одномерный массив. Например для массива с ключами, являющимися именами малых планет, (не самое длинное!) имя “Холшевников” потребует 11 байтов для кодирования, и следовательно размерность массива должна будет превосходить $(2^8)^{11} \sim 2 \cdot 10^{26}$, а сам занимать более 10^{18} Гигабайтов!

Очевидно, что память следует использовать более эффективно. Предположим, что мы нашли функцию $i = f(k)$, отражающую множество всех значений ключей k в отрезок натурального ряда $[1, N]$, такой что $N \ll M$, где M — количество всех возможных ключей. Тогда по произвольному ключу k мы найдем малое целое значение i, которое может быть использовано в качестве индекса, и, следовательно, все данные можно таким образом разместить в массиве небольшой размерности N и далее работать с его элементами. Такой метод называется *хэшированием*, а функция f — *хэш-функцией*.

Примером хэш-функции может служить функция взятия остатка от деления $f(k) = k \pmod p$, где k — ключ, трактуемый как целое число, p — простое число. Этот пример, кстати, хорошо поясняет, почему хэширование получило название от английского глагола hash — перемалывать. Хэш-функция должна бы быть взаимооднозначной, но вряд ли можно придумать функцию, тем более простую, которая бы (для любых ключей) обладала таким свойством. На практике будут встречаться случаи, когда разным значениям ключей будут соответствовать одинаковые значения функции. Такая ситуация называется *коллизией*, и существуют стандартные способы ее разрешения (например, элемент хэша может быть сделан головой цепного списка записей, содержащих такие данные).

(Равномерное распределение и перемешивание бит?)

(Пример(ы) из Романовского???)

Интересно, что эта же задача возникает и в криптографии. В настоящее время для обеспечения безопасности используются коды, состоящие из 64 бит ($\approx 2 \cdot 10^{18}$), их надо отобразить в индексы, определяющие пользователя (на Земле $\approx 5 \cdot 10^9$ человек). Снова хэш-функция!. В этом случае, однако, к самой хэш-функции требования более строгие:

- функция должна быть определена для всех мыслимых аргументов;

- значения функции должны лежать в определенном интервале;
- функция должна относительно легко вычисляться;
- она должна быть “необратимой”, то есть за “конечное” время по значению функции h нельзя найти такой аргумент x, что $f(x) = h$;
- коллизий не должно быть, то есть за “конечное” время невозможно найти два таких аргумента x и y, что $f(x) = f(y)$.

1.3.4 Списки как абстрактные структуры

Подведем некоторые итоги. Мы рассмотрели три простейшие структуры: структуры в узком смысле или записи, массивы и хэши. С каждой из рассмотренных структур связывались некоторые операции. Если отвлечься от операции создания того или иного объекта, то такими операциями являются: выбор некоторого (определенного) поля объекта для структуры; выбор элемента массива с некоторым индексом; или выбор элемента хэша по значению ключа. И соответственно изменения таких элементов. Именно в этих, “статических” структурах эти операции просто и эффективно реализуются, а, значит, быстро работают.

Вообще рассмотрение класса операций, которые будут выполняться с данными **совместно** со структурой наших данных — хороший тон (и это еще мягко сказано). Структуры данных задачи в равной степени определяется и присущими этим данным свойствами, и теми функциями, которые предполагается выполнять с нашими данными. Предположим, что в наш “звездный каталог” мы должны вставить какую-то новую “звезду”, после звезды с порядковым номером i. Для этого нам необходимо переопределить все элементы массива с индексами $> i$: $Star(i+1) = Star(i)$. Конечно, чтобы не потерять информацию, придется начинать этот сдвиг с последней звезды, это очевидно, и, если такую операцию нам нужно проделать один раз, то мы можем смириться с тем, что из-за включения в каталог новой звезды мы должны переместить в среднем $N/2$ звезд. Но если в нашей задаче требуется выполнить такую операцию много раз, а N велико, то время, потраченное только на эту операцию включения, будет недопустимо большим.

В приведенномм примере простая структура — массив — не позволяет нам просто и эффективно сохранять важные структурные отношения (порядок), присущие нашим данным. В этом случае существенным для задачи является не только количество и тип элементов структуры, но и существующие между этими элементами связи, причем эти связи могут возникать, изменяться и уничтожаться в процессе решения задачи.

Подобную *структуру данных* можно определить как множество элементов и связей между ними, делая таким образом определенный акцент на связи.

Данное абстрактное определение структуры достаточно общее, т.е. включает широкий класс структур. Это, конечно, плюс, но есть и два минуса. Первый — то, что моделировать эти структуры в компьютере нам придется самим, используя простейшие данные и структуры, рассмотренные выше. При этом представление структур будет определяться теми операциями, которые предполагается над ними производить[2]. Заметим здесь же, что все операции придется реализовывать программно, и это второй минус.

Линейные списки; стек, очередь, дек

Что означает список, понятно из опыта обыденного использования этого слова. Можно положить, что *список* — это один элемент и подсписок. Не будем пояснять данное определение, его смысл станет понятен из приводимых далее примеров. Сначала рассмотрим списки, в которых определено отношение упорядоченности. Дадим формальное определение таких линейных структур.

Линейным списком будем называть множество, состоящее из $n \geq 0$ элементов или *узлов*, структурные свойства которого ограничиваются лишь линейным относительным положением узлов: существует первый узел X_1, и каждому узлу X_k $(1 < k \leq n)$ всегда предшествует узел X_{k-1}.

Часто выполняются следующие операции над линейными списками:

- получение доступа к k-му узлу (получить или изменить содержимое этого узла);
- включение нового узла перед узлом k;
- исключение k-го узла;
- объединение двух линейных списков в один;
- разбиение линейного списка на несколько подсписков;
- копирование линейного списка;
- определение количества узлов в списке;
- сортировка узлов списка по некоторым полям узлов;

[2]Очевидно, что следует выбирать такое представление, при котором наиболее часто выполняемые операция будет производиться наиболее эффективно.

- поиск в списке узла с заданным значением.

В редких случаях требуются все перечисленные операции, и в зависимости от требуемых операций мы можем выбрать тот или иной способ представления линейных списков. Собственно, нам уже встречалась удачная реализация линейных списков, в которых определяющей операцией была первая из перечисленных выше, это — одномерный массив.

Предположим, однако, что определяющими (часто выполняемыми) операциями, являются операции включения/исключения узла. В этом случае использование одномерного массива едва ли может нас удовлетворить.

В некоторых случаях, однако, предыдущее утверждение можно оспорить. Все зависит от значения k в операциях включения/исключения. Линейные списки, в которых эти операции производятся при граничных значениях k, имеют специальные названия: *стек*, в котором все операции включения/исключения (а часто и операции доступа) делаются в одном и том же конце списка, *очередь*, где все включения делаются в одном конце списка, а все исключения — в другом, *дек*, где все включения и исключения делаются на обоих концах списка.

Несмотря на такую исключительность стек (другие структуры тоже, но в меньшей степени) — очень широко используемая структура. Одна даже имеет несколько названий: стек, список push-down, список LIFO и др.

Стеки явно или неявно используются всегда, когда одна задача сводится к другой, а та к третьей и т.д. Мы накапливаем в стеке эти задачи и удаляем их по мере решения. Собственно так мы думаем, и так же организован процесс входов в подпрограммы и выходов из них.

Для моделирования рассмотренных структур, особенно стека, можно использовать одномерный массив довольно эффективно.

В общем же случае такое “последовательное” представление требует более эффективной и более гибкой замены.

Представление линейных списков, связные списки

Добавим в нашу структуру $Catalog$ еще одно поле $Next$, являющееся указателем, и поместим в этот указатель адрес (может быть относительный) следующего элемента типа $Catalog$. Увеличение структуры на одно поле в подавляющем большинстве случаев несущественно, но зато как облегчаются операции включения и исключения элементов! Теперь операция включения будет состоять в присваивании полю $Next$ нового элемента адреса $(i+1)$-го элемента и в присваивании полю $Next$ элемента

i адреса нового элемента. Операция удаления элемента из связного списка еще проще — указатель $(i-1)$-го элемента следует сделать равным адресу $(i+1)$-го элемента.

Остается, правда, неясным вопрос, куда денутся исключенные из списков элементы и откуда берутся вновь включаемые. Большинство языков высокого уровня предоставляют механизмы работы с памятью. Отметим, что просто удалить исключаемый элемент нельзя, поскольку не всегда известно, не ссылается ли на данный узел какой-нибудь элемент. Для удаления элементов существует специальный механизм "сборки мусора", который запускается, когда доступная задаче память исчерпывается.

Увеличение структуры $Star$ на одно поле в подавляющем большинстве случаев несущественно, но зато как облегчаются операции включения/исключения новых элементов! (Схемы, nil?)

Можно сформулировать общий критерий использования связного и последовательного представлений. Если при работе с линейным списком связи между элементами не меняются, то использование массива предпочтительнее, поскольку он занимает меньше памяти и с ним удобнее работать. В случаях, когда основными операциями являются включения/исключения элементов, связное представление предпочтительнее.

Заметим, что если это необходимо (например, если используется язык, в котором нет средств работы с указателями) легко смоделировать работу со связными списками при помощи массивов.

Деревья, бинарные деревья, сбалансированные (AVL) деревья

Все до сих пор рассмотренные структуры являются линейными структурами, в которых определено отношения порядка. В вычислительных алгоритмах встречаются и нелинейные структуры. К важнейшим из таких структур относятся *деревья*.

Формально можно определить дерево, как конечное множество T, состоящее из одного или более *узлов*, таких что имеется один узел, который называется *корнем* дерева, а все остальные узлы содержатся в $m \geq 0$ попарно не пересекающихся множествах $T_1, \ldots, T_m$, каждое из которых является деревом. Деревья $T_1, \ldots, T_m$ называются *поддеревьями* корня.

Обратите внимание, что это определение является *рекурсивным*: мы определили дерево в терминах самих же деревьев: деревья с n узлами $(n > 1)$ определяются через деревья с количеством узлов $< n$ и, следовательно, в конце концов мы придем к деревьям, состоящим из одного узла, и на этом рекурсия закончится.

Существуют и нерекурсивные определения деревьев, но они представляются менее подходящими, поскольку рекурсивность является естественной свойством таких структур.

Дадим еще несколько определений. Число поддеревьев данного узла является *степенью* узла. Узел с нулевой степенью называется *концевым узлом* или *листом*. Говорят, что корень имеет *уровень* 1, а остальные узлы имеют уровень на единицу больше уровня содержащего их поддерева. Если имеет значение порядок поддеревьев $T_1, \ldots, T_m$, то такое дерево является *упорядоченным*.

Часто используется и терминология *генеалогического дерева*. Говорят, что корень является *отцом* корней своих поддеревьев. Каждый корень, кроме корня всего дерева, является *ребенком* своего отца. Если не ограничиваться смежными уровнями, то можно употреблять термины *предок* и *потомок*.

(Несколько способов графического представления дерева: в виде дерева корнем вверх, в виде вложенных множеств, вложенных скобок, в виде уступчатого списка, обозначения Дьюи).

Одним из важных типов деревьев является *бинарное дерево* — дерево, в котором каждый узел имеет либо два поддерева, называемых *левым* и *правым*, либо ни одного. Для представления бинарного дерева необходимо к информационной части, например, к структуре $Star$ добавить еще две ссылки: на левое поддерево $Left$ и на правое поддерево $Right$.

При работе с деревьями часто выполняется операция *обхода* дерева, при которой каждый узел проходиться ровно один раз. Полное прохождение дерева даст нам, кстати, линейную расстановку узлов.

Для прохождения бинарного дерева можно использовать один из трех способов обхода дерева: *прямой*, *обратный* или *концевой*. Определяются эти способы рекурсивно, если бинарное дерево не пусто, то обход состоит в том, чтобы:

прямой порядок	обратный порядок	концевой порядок
попасть в корень	пройти левое поддерево	пройти левое поддерево
пройти левое поддерево	попасть в корень	пройти правое поддерево
пройти правое поддерево	пройти правое поддерево	попасть в корень

Всякое дерево можно естественным образом представить в виде бинарного дерева, и, обратно, всякое бинарное дерево можно представить в виде дерева. (Леса?)

(Примеры?, представление арифметических выражений, скобочная запись, обязательно)

Деревья как структуры широко используются в самых разнообразных задачах. Например, в задачах поиска (см., например, выше раздел

о теореме Шеннона). Очевидно, что операция поиска очень эффективна в данных, организованных в бинарные деревья. Если использовать линейные структуры, например массив, то время поиска пропорционально количеству элементов массива $O(N)$. Если же для организации данных использовать бинарное дерево, то, как это подмечено еще в древней Греции (дихотомия), время пропорционально $O(\log_2(N))$. При больших N разница огромна.

Правда, это время зависит не только и не столько от N, сколько от глубины данного дерева. В самом плохом случае это время будет пропорционально N, в самом хорошем — $O(\log_2(N))$. Поэтому при поиске следует иметь дело с так называемыми *сбалансированными* деревьями, в которых количество узлов в левом и правом поддеревьях каждого узла различаются не более, чем на единицу. В таких деревьях операции вставки в дерево нового узла или удаления узла из дерева, как правило, приводят к тому, что дерево перестает быть сбалансированным, и требуются дополнительные усилия, чтобы сохранить свойство сбалансированности. Эта задача немного упрощается, если сбалансированными считать деревья, в которых высота левого и правого поддерева каждого узла, отличаются не более, чем на единицу. Деревья, обладающие этим свойством, называются *AVL-деревьями.* При операции вставки или удаления узла свойство сбалансированности меняться не должно. В этом случае при операции поиска среднее количество действий будет пропорционально $\mathrm{O}(\log_2(N))$.

Естественно использовать деревья и в задачах с данными, обладающими иерархической структурой. Рассмотрим, например, задачу эволюции больших звездных систем под действием гравитационных сил — задачу N тел, где N очень велико ($\approx 10^8$).

В простейшем случае, когда действуют только гравитационные силы, для каждого тела нужно вычислять силы притяжения Ньютона от всех остальных тел, всего $O(N^2)$ таких вычислений. Если $N \approx 10^8$, то задача уменьшения этих вычислений становится жизненно важной. Были предложены методы (Greengard и Barnes-Hut) вычисления взаимодействий N тел, включающих на каждом шаге интегрирования (временном шаге) два действия:

1. построение дерева с листьями – частицами (корень дерева — вся система);

2. обход этого дерева с тем, чтобы с нужной точностью вычислить силы, действующие на частицу.

Алгоритм решения этой задачи будет рассмотрен в разделе “Работа с деревьями в Treecode”, здесь же лишь скажем, что использование деревьев в данной задаче позволяет снизить трудоемкость вычислений вычислений с $O(N^2)$ до $O(N \ln_2 N)$.

Так же, как и для линейных списков, для представления деревьев применяются два способа, один использует массивы, а другой — указатели. Здесь лучше называть их не последовательный и связный как ранее, а статический и динамический соответственно.

Одним из возможных способов представления двоичного дерева при помощи массива является следующий. В первый элемент массива поместим узел дерева, в следующие два элемента — два узла второго уровня, в последующие четыре элемента — четыре узла третьего уровня и т.д.

При динамическом представлении узел должен содержать помимо информационных полей поля, являющиеся указателями на корни соответствующих поддеревьев. Подобное представление деревьев более естественно и просто и поэтому более часто используется.

Графы и их представление

Графы, как и деревья, является нелинейными структурами. Они используются для моделирования таких сложных объектов как сети (транспортная сеть, информационная сеть, сеть интернет и т.д.). Сетевой объект состоит из множества элементов одного типа, связанных между собой. Например, в случае (сети) метрополитена элементами могут служить станции метро, а в качестве связей выступать туннели, их соединяющие. Элементы, в данном случае станции, могут характеризоваться: названием, информацией о наличии пересадки на другую линию и т.п., а туннели — длиной и т.д.

Строгие определения таковы. *Граф* — это множество точек, называемых *вершинами*, и линий, соединяющих пары вершин и называемых *ребрами*. Каждая пара вершин может быть соединена не более чем одним ребром. Две вершины, соединенные ребром, называются *смежными*. Если $V_1, V_2, ...V_k$ — подмножество вершин графа таких, что V_i и V_{i+1} — смежные вершины, то последовательность ребер $l_{12}, l_{23}, ...l_{k-1,k}$ называется *путем* длины k от вершины V_1 к вершине V_k. Путь называется *простым*, если различны вершины $V_1, V_2, ...V_{k-1}$ и различны вершины $V_2, V_3, ...V_k$. Граф называется *связным*, если любую пару вершин связывает какой-то путь. *Циклом* называется простой путь, имеющий длину больше двух и идущий от вершины к ней самой.

Если для каждого ребра определено направление, то получаем *ориентированный* граф. В отличие от обычного, в ориентированном графе две

вершины могут быть соединены множеством дуг (ребер), а также начало дуги может совпадать с концом, т.е. вершина может быть соединена сама с собой.

Наиболее часто используемые операции над графами:

1. Определение принадлежности заданной дуги или ребра графу.

2. Удаление и добавление дуги или ребра.

3. Перебор всех дуг или ребер, выходящих из заданной вершины.

и т.д.

При программировании задач, предполагающих работу с сетевыми объектами, необходимо решить вопрос о представлении графа. Выбор представления определяется методом (алгоритмом) решения задачи, а также соображениями экономии памяти при обработке больших графов.

Существует два стандартных способа представления графа:

1. Используя набор списков смежных вершин.

2. При помощи матрицы смежности.

При представлении графа в виде списка смежных вершин применяется одномерный массив (A) размерности N, где N — количество вершин. Для i-й вершины графа соответствующий элемент массива $A(i)$ содержит указатель на связный список, элементы которого вершины, смежные с i-й вершиной.

Второе представление используется, когда количество ребер M сравнимо с $N*N$. Здесь создается матрица (двумерный массив) B размерности $N*N$, элементы которой задаются следующим образом: $B(i,j)=1$, если i-я вершина соединена ребром (дугой для ориентированного графа) с j-й вершиной, и $B(i,j)=0$ в противном случае.

1.3.5 Структурируемые потоки данных

Тексты

Если строки — это последовательность символов, или байтов, то когда мы говорим о *тексте*, мы подразумеваем нечто большее, чем просто последовательность символов или даже слов. Нам важнее, что текст имеет свою, присущую только этому тексту, логическую структуру. И эта структура, вообще говоря, помогает интерпретировать информацию, содержащуюся именно в данном тексте.

Логическая структура документа — иерархическая и хорошо представляется древовидной структурой. Рассмотрим, например, первую лекцию. Ее можно представить в виде линейного списка, или в виде дерева:

```
1. 1-ая Лекция
   2. Введение
      3. Цели и задачи курса
      4. Притча Дейкстры
   5. Информация
      6. Введение. Сообщение и информация
      7. Связь сообщения и информации, их интерпретация и обработка
      8. Дискретные сообщения. Знаки. Алфавит
      9. Наборы двоичных знаков. Слова. Коды. Символы
     10. Шенноновские сообщения. Количество информации.
         Теорема кодирования Шеннона
```

Логические единицы, присущие тексту, зависят от его вида. В прозе — это главы, абзацы, предложения. Поэтический текст можно разделить на стихотворения, строфы, строки. Драматургический текст содержит действия, сцены, реплики, ремарки. Документация может включать разделы, определения, правила, команды, иллюстрации и т.д. (вплоть до текстов программ). Конечно, между перечисленными типами документов нет четкой границы, а приведенными выше единицами не исчерпывается все многообразие существующих текстовых структур.

Ясно, что сравнительно большой текст, как правило имеет иерархическую (нелинейную) структуру. Однако внешние носители заставляют нас хранить тексты по сути в виде линейных списков строк — текстовых строк. Если переход от иерархической структуры к линейной выполняется просто (например, обход дерева, переводящий его в линейный список), то обратный переход едва ли возможен, если мы не дополним текст информацией о его логической структуре. Такая вносимая в текст информация называется его *разметкой*.

Важнейшие требования, которым должны удовлетворять стандарты разметки, очевидны:

- Во-первых, важно определить логическую структуру текста, как этот текст будет в дальнейшем обрабатываться — зависит от стоящей задачи и будет выполняться соответствующими обработчиками, логическая структура текста, разумеется, не должна зависеть от той обработки, которой этот текст будет подвергаться.

- Во-вторых, мы должны иметь возможность работать с любыми документами, в каждом конкретном случае определять структуру

текста, характерную именно для этого конкретного документа, то есть задавать тип документа.

- В-третьих, текст не должен зависеть от какой-то конкретной системы представления.

Именно эти свойства характерны для стандартного языка обобщенной разметки SGML (Standard Generalized Markup Language). Собственно, SGML — это метаязык, с его помощью можно описать нужный язык разметки, а затем использовать этот язык для представления и обработки текста. Широко известный пример — HTML (Hypertext Markup Language), тривиальный язык разметки, определенный с помощью SGML.

В SGML для текстовых структурных единиц используется термин *элемент*. Для элементов SGML определены только отношения к другим элементам, никакими другими свойствами они не обладают. Эти отношения определяются в специальном наборе описательных утверждений с простым синтаксисом — DTD (Document Type Definition, т.е. *определении типа документа*). Подобное определение может быть задано как для конкретного документа, так и для документов какого-то широкого класса.

В виде иллюстрации приведем SGML-разметку (DTD построим позже) поэтической антологии:

```
<anthology>
<poem><title>***</title>
<author>Саша Белоснежка<img src="abs.jpg"></author>

<stanza>
<line>Кто стучится на Руси</line>
<line>Длинным файлом по PC</line>
<line>Это он, это он,</line>
<line>Электронный почтальон</line>
</stanza>
</poem>

<poem><title>Впечатление</title>
<author>ВИК</author>

<stanza>
<line>Чуть заметен бледный щит Стожар,</line>
<line>Ярко блещет пояс Ориона.</line>
<line>И морозной искрою зажжённый</line>
```

```
<line>Буйствует сияния пожар!..</line>
</stanza>
</poem>
</anthology>
```

Здесь заложена следующая структура: корень — весь текст (anthology); объекты второго уровня — стихотворения (poem), включающие в себя название (title), автора (author), и собственно строфы стихотворений (stanza). Последний, четвертый уровень иерархии — строки (line). С помощью элемента img можно поместить в текст фотографию автора.

Набор описательных утверждений (DTD) может выглядеть, например, так:

```
<!ELEMENT anthology   - - (poem+)>
<!ELEMENT poem        - - (title?, author, stanza+)>
<!ELEMENT title       - O (#PCDATA) >
<!ELEMENT author      - O (#PCDATA, img?) >
<!ELEMENT stanza      - O (line+)   >
<!ELEMENT line        O O (#PCDATA) >
<!ELEMENT img         - O EMPTY >
<!ATTLIST img src CDATA #REQUIRED>
```

Каждое правило для элементов состоит из трех частей: название (или группа названий, например anthology), два следующих поля — информация о том, может ли быть опущена начальная и конечная метки, последнее поле — модель содержимого — указывает, что́ могут содержать экземпляры элемента, зарезервированное слово `#PCDATA` (Parsed Character Data), например, означает, что описываемый элемент может включать любые разрешенные символьные данные.

Знак “+” в модели содержимого показывает, что элемент может быть повторен более одного раза (если бы стоял знак “*”, то элемент мог бы отсутствовать вовсе или мог быть повторен сколько угодно раз), а знак “?” — элемент может встречаться не более одного раза. `EMPTY` — зарезервированное слово, означающее, что у элемента нет содержимого.

Элементы могут иметь *атрибуты*, которые также описываются в DTD в утверждении ATTLIST. В нашем примере в список аттрибутов элемента img включен аттрибут src, значение которого, `CDATA`, может включать любые разрешенные символы, этот аттрибут всегда должен задаваться (`#REQUIRED`).

Кроме того, произвольную часть документа можно поименовать, определив *объект*. Например,

```
<!ENTITY auml   "&#x00E4;">
```

Это определение можно использовать так: `&auml`.

Нетрудно написать DTD, определяющий всем известный язык HTML. Напомним, что SGML описывает лишь структуру текста, а интерпретация, визуализация и другая обработка является отдельной независимой задачей. Для решения подобных задач разработан язык DSSSL (Document Style Semantics and Specification Language), где можно определить, например, правила визуализации.

Астрономические таблицы

Разумеется с помощью этой (SGML) техники можно описывать не только тексты. Астрономический пример этого — VOTable — гибкий формат для хранения и обмена табличных данных, в частности астрономических данных. Точнее VOTable основан не на SGML, а на XML, более широко используемым сейчас в Интернете языком разметки, но эти средства достаточно близки, чтобы в данном контексте их различием можно было пренебречь.

Свойства XML, однако, позволяют сделать данные не только переносимыми, но и распределенными в сети Интернет.

Вот пример VOTable-документа:

```
<?xml version="1.0"?>
<!DOCTYPE VOTABLE SYSTEM "http://us-vo.org/xml/VOTable.dtd">
<VOTABLE version="1.0">
  <DEFINITION>
  <COOSYS ID="myJ2000" equinox="2000." epoch="2000." system="eq_FK5"/>
  </DEFINITION>
  <RESOURCE>
    <PARAM name="Observer" datatype="char" arraysize="*" value="William Herschel">
      <DESCRIPTION>This parameter is designed to store the observer's name
      </DESCRIPTION>
    </PARAM>
    <TABLE name="Stars">
      <DESCRIPTION>Some bright stars</DESCRIPTION>
      <FIELD name="Star-Name" ucd="ID_MAIN" datatype="char" arraysize="10"/>
      <FIELD name="RA" ucd="POS_EQ_RA"  ref="myJ2000" unit="deg"
             datatype="float" precision="F3" width="7"/>
      <FIELD name="Dec" ucd="POS_EQ_DEC"  ref="myJ2000" unit="deg"
             datatype="float" precision="F3" width="7"/>
      <FIELD name="Counts" ucd="NUMBER" datatype="int" arraysize="2x3x*"/>
      <DATA>
        <TABLEDATA>
        <TR>
```

```
          <TD>Procyon</TD><TD>114.827</TD><TD> 5.227</TD>
          <TD>4 5 3 2 1 2 3 4 6</TD>
        </TR>
        <TR>
          <TD>Vega</TD><TD>279.234</TD>
          <TD>38.782</TD><TD>8 7 8 6 8 6</TD>
       </TR>
       </TABLEDATA>
      </DATA>
    </TABLE>
  </RESOURCE>
</VOTABLE>
```

Данные в VOTable представляются в одном из трех форматов: TABLEDATA, FITS и BINARY. Что такое TABLEDATA ясно из примера. FITS — формат, широко используемый и разработанный астрономами для передачи данных еще в семидесятых годах. FITS-файл можно или прямо включить в <FITS>...</FITS>, или предварительно восстановить метаданные. В формате BINARY содержатся просто двоичные данные (поток битов).

Свойства XML, однако, позволяют сделать данные не только переносимыми, но и распределенными в сети Интернет.

А вот фрагмент DTD для VOTable:

```
<!-- DOCUMENT TYPE DEFINITION for VOTable =
                            Virtual Observatory Tabular Format -->

<!-- VOTABLE is the root element -->
<!ELEMENT VOTABLE (DESCRIPTION?, DEFINITION?, INFO*, RESOURCE*)>
<!ATTLIST VOTABLE
  . . . . . .
>
<!-- RESOURCE can contain other RESOURCE -->
<!ELEMENT RESOURCE (DESCRIPTION?, INFO*, COOSYS*, PARAM*, LINK*,
          TABLE*, RESOURCE*)>
  . . . . . .
<!ELEMENT DESCRIPTION (#PCDATA)>
```

Точное описание VOTABLE выходит за рамки данного курса. Отметим, что использование в качестве метаязыка XML позволяет обрабатывать данные с помощью уже разработанного для XML программного обеспечения, отделить описание данных от самих данных (хранить их отдельно, на разных сайтах), сделать данные, распределенные в Интернете, доступными для всего астрономического сообщества.

Это не единственное применение SGML/XML в астрономии. Можно упомянуть, например, AML, AIML и т.д.

1.3.6 Файлы и их форматы

Любая информация, любые данные, требующиеся для решения той или иной задачи, или получающиеся в результате или в процессе решения, как правило, сохраняются на внешних носителях (винчестерах, CDROM'ах и т.д.) в виде *файлов*.

Файл можно рассматривать как последовательность байт, вообще говоря, не имеющую каких-либо структурных отношений — просто последовательность (поток) байт,— игнорируя иерархию, определенную в приложении (с точки зрения приложения обычно файла — это группа взаимосвязанных *записей*, в свою очередь запись — группа взаимосвязанных *полей*, а поле — группа байтов).

Структурные отношения, присущие данным, т.е. их логическая структура, в файле обычно отсутствует; они определяются и используются лишь в программах, которые работают с данным файлом.

С файлом связан ряд атрибутов. В большинстве случаев файл имеет *имя* — строку символов, которое идентифицирует файл, определяет его местонахождение. В число атрибутов входит также информация о правах доступа, собственнике файла, времени создания и т.д.

Операции, производимые над файлами, включают: создание, копирование, удаление, получение информации о файле. Кроме этого, с содержимым файла выполняются следующие действия: открытие (связывание имени файла с содержимым и выполнение некоторых вспомогательных действий), чтение, запись, позиционирование, закрытие. Операции над содержимым файла:

В заключение сделаем два замечания:

1. Оставляя определение структуры данных файла и его интерпретацию за программой, мы не можем расчитывать на то, что та или иная программа правильно обработает любой файл. Последовательность байтов файла обрабатывается конкретной программой и поэтому подчиняется правилам, определенным в этой программе. Однако для часто используемых (графических, мультимедийных, астрономических и т.д.) программ эти правила известны (а часто представляются некоторым стандартом), а файлы, им удовлетворяющие, как говорят, имеют определенный *формат*.

Как правило, указание на формат файла содержится в его расширении, например, `.fits`, `.gif`, `.pdf`, `.ps`, `.mpeg`, `.txt`, `.rtf` и т.д. Разумеется, соблюдение такого порядка удобно, но вовсе не обязательно: удовле-

творяет ли тем или иным правилам файл или нет, проверяет вызванная программа. В `Unix` есть специальная команда (программа) `file`, которая определяет формат файла по его содержимому, то есть определяет какому из известных правил удовлетворяет заданный файл.

2. Мы коснулись только "последовательных" файлов. Программы, работающие, например, со сложными базами данных, обычно обрабатывают большие объемы информации и для повышения эффективности (быстродействия), могут использоваться более сложные способы хранения данных на внешних носителях (например, индексные указатели, уточняющие положение нужного фрагмента данных и т.п.).

2 Алгоритмы

Алгоритм — набор конечного числа правил, задающих последовательность выполнения операций для решения задачи. Алгоритм имеет пять важных свойств:

1. конечность: алгоритм всегда должен заканчиваться после конечного числа шагов;

2. определенность: каждый шаг должен быть точно определен;

3. наличие входных данных: алгоритм имеет некоторое число входных данных;

4. наличие выходных данных: алгоритм имеет одну или несколько выходных величин, имеющих определенные отношения к входным данным;

5. эффективность: алгоритм считается эффективным, если все необходимые операции достаточно просты, чтобы их можно было выполнить точно и за конечное время.

Если мы откажемся от требования конечности, то такой набор правил можно назвать *вычислительным методом*. Что касается определенности, то правила желательно сформулировать на языке, исключающем неточности и известном читателю. Естественный язык (русский, английский или эсперанто) не совсем подходит для этого, поскольку он допускает неоднозначную интерпретацию. Для описания алгоритмов используются формально определенные *языки программирования*, в которых каждое утверждение имеет абсолютно точно определенный смысл. Запись вычислительного метода на таком языке называется *программой*.

На практике нужны не просто алгоритмы, а хорошие (в каком-то определенном смысле) алгоритмы. Лепота алгоритма может, например, характеризоваться временем, необходимым для его выполнения, простотой, требуемыми ресурсами и т.д. Часто нужно из нескольких алгоритмов нужно выбрать лучший. Исследование рабочих характеристик алгоритмов называется *анализом алгоритмов* — важная и непростая задача. Можно выделить анализ алгоритмов в среднем, когда исследуется поведение алгоритма в средних условиях, анализ алгоритма в самых неблагоприятных или, наоборот, самых благоприятных условиях. Например, число шагов в алгоритме Эвклида нахождения наибольшего общего делителя двух целых чисел m и n в среднем равно $T_n = (12 \ln 2/\pi^2) \ln n$ (получение этой оценки — очень трудная математическая задача).

Для *построения алгоритмов* существует множество стандартных приемов (например, пошаговые алгоритмы, алгоритмы перебора, рекурсивные алгоритмы). Некоторые из этих приемов будут показаны в дальнейшем, при рассмотрение конкретных алгоритмов.

Можно ли эффективно решить, существует ли алгоритм решения той или иной задачи? Любую ли задачу может решить компьютер? Этими вопросами занимается наука, вернее раздел науки, которая называется *теорией алгоритмов*.

2.1 Введение в теорию алгоритмов

Что не все задачи можно решить с помощью компьютера, можно интуитивно догадаться, осознав, что количество всех программ (на всех возможных языках) — не более, чем счетно, тогда как множество задач — не счетно. Даже множество всех функций, отображающих натуральные числа в натуральные не является счетным. Доказывается это элементарно:

Предположим, что множество всех таких функций счетно, следовательно мы можем их перенумеровать: f_i. Определим функцию g следующим образом:

$$g(i) = f_i(i) + 1.$$

Функция g безусловно входит в рассматриваемое множество, и, очевидно, отличается по определению от каждой функции f_i, что противоречит предположению о том, что множество наших функций счетно.

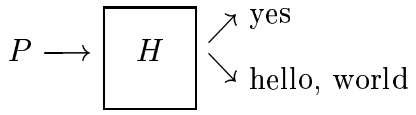
Рассмотрим, например, такую, весьма простую задачу: является ли текст `hello, world` первым, что печатает C-программа (можно Pascal-программа, или даже Фортран-программа?). Разумеется, проще всего было бы запустить ее и посмотреть на результат. Однако может оказать-

ся, что ждать придется долго, очень долго... Ну конечно, если программа состоит из одного оператора печати `hello, world`, то ответ будет утвердительным, да и проверить не сложно. Но если программа должна прежде что-то сделать, например, найти целое положительное решение уравнения $x^n + y^n = z^n$, где n — входные данные программы, и только в случае успешного решения вывести `hello, world`, то вряд ли мы дождемся приветствия — при $n > 2$ это уравнение не имеет решения (а доказывалась теорема Ферма более 300, точнее 358, лет!). То есть программа в пару десятков строчек, которую каждый из вас может легко написать, решения не найдет. Ну в данном-то случае 300-летними стараниями математиков все ясно, а другие задачи? (Ограничения, связанные с величиной представимых целых чисел мы не рассматриваем, их легко обойти или, по крайней мере, очень сильно ослабить, определив структуры для целых чисел произвольной длины, например использовав для представления таких чисел связанные списки).

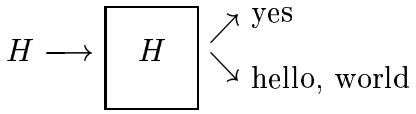
Было бы замечательно, если бы мы написали программу, которая проверяет любую программу P со входом I и определяет, печатается ли при ее выполнении `hello, world`. Покажем, что такой программы не может быть.

Предположим, что нам удалось написать программу H, которая получая на входе некоторую программу P, выводит `yes`, если P печатает `hello, world`, или, в противном случае, H выводит `hello, world`, если P не печатает `hello, world`. Небольшие ограничения (вход I, например, включается в P) не меняют сути.

Итак



Подадим на вход нашей программы H ее саму



Что же получается? Если H (в рамке) выдает `yes`, то это означает, что программа на входе H печатает `hello, world`, а если H (в рамке) выдает `hello, world`, то это означает, что программа на входе H печатает НЕ `hello, world`. В обоих случаях мы получили, не то что ожидали. Противоречие и доказывает, что такая программа H не может существовать. То есть, не может существовать программы H, которая бы могла определить, печатает ли данная программа P со входом I текст `hello, world`.

Задачи, которые нельзя решить с помощью компьютера, называются *неразрешимыми*. Легко убедиться, что неразрешима, например, и такая задача: по данной программе и ее входу определить, останавливается ли она когда-нибудь, т.е. не зацикливается ли она при данном входе. Ее легко *свести* к предыдущей, добавив в конец программ P, оператор, выводящий текст `hello, world`. Поскольку задача о `hello, world` неразрешима, то неразрешима и задача остановки.

Таким образом, не существует такой программы, которая для произвольной программы P и произвольных входных данных I может установить, закончится ли вычисление P с данными I.

Вот еще несколько неразрешимых задач:

- Для произвольного утверждения в достаточно богатой математической теории установить, является ли оно теоремой.
- Установить эквивалентность двух произвольных процедур.

2.1.1 Машина Тьюринга. Полиномиальные задачи

Установить разрешимость задачи — значит понять можно ли решить данную задачу или нет. Важно также выделить задачи, решение которых не требует чрезмерно большого времени.

Теория алгоритмов, вообще говоря, не должна быть связана с программами на каком-либо конкретном языке. Нужна модель компьютера, с одной стороны, достаточно простая, а с другой — модель позволяет делать заключения, справедливые для любого алгоритма.

Такой моделью является *машина Тьюринга*. Машина Тьюринга состоит из *конечного управления*, которое может находиться в любом из конечного множества состояний. Есть *лента*, разбитая на *клетки*, в которых могут храниться символы из некоторого конечного их множества. На ленте записан *вход*, представляющий цепочку символов конечной длины из *входного алфавита*, остальные символы ленты (до бесконечности слева и справа) содержат *пустой символ*, или *пробел*. Машину Тьюринга можно описать семеркой $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Здесь
Q — конечное множество состояний управления;
Σ — конечное множество входных символов;
Γ — множество ленточных символов, $\Sigma \subset \Gamma$;
δ — функция переходов $(p, Y, D) = \delta(q, X), q \in Q, X \in \Gamma$, здесь $p \in Q$ — следующее состояние, $Y \in \Gamma$ — новый записываемый на ленту символ, D — направление сдвига головки (L или R);
$q_0 \in Q$ — начальное состояние;

$B \in \Gamma, B\neg \in \Sigma$ — пустой символ;
$F \subset Q$ — множество заключительных, или *допускающих* состояний.

(Пример?)

Работа машины Тьюринга состоит из переходов, подразумевающих следующие действия:

1. считать текущий символ и изменяем состояние управления;
2. записать ленточный символ в текущую позицию;
3. сдвинуть головку влево или вправо.

Итак входная цепочка помещается на ленту, текущая клетка ленты — крайняя слева. Если машина Тьюринга в конце концов достигает допускающего состояния, то говорят, что входная цепочка *допускается*, в противном случае — нет.

Несмотря на простоту конструкции можно показать, что машина Тьюринга в некотором смысле не отличается от обычного компьютера. Компьютер может имитировать машину Тьюринга (можете написать соответствующую программу на любимом языке), и, наоборот, машина Тьюринга может имитировать компьютер. Более того, число переходов машины Тьюринга при разрешении некоторой цепочки символов можно выразить в виде некоторого полинома от числа шагов, совершаемых компьютером, то есть если задачу можно решить за полиномиальное время на обычном компьютере, то ее можно решить за полиномиальное время на машине Тьюринга, и наоборот.

Таким образом, машина Тьюринга представляет собой абстрактную вычислительную машину, мощность которой (в смысле вычислимости) совпадает с мощностью реальных компьютеров.

Итак, мы встречали уже "неразрешимые" задачи (например, задача останова). Если мы имеем дело с такой задачей, мы должны довольствоваться знанием того, что решить ее с помощью компьютера нельзя. Как правило, мы имеем дело с полиномиальными задачами, время работы которых ограничено некоторым полиномом от размера начальных данных, время работы на входе длины n составляет $O(n^k)$. Очевидно, используемые на практике алгоритмы должны быть полиномиальными.

Кроме того, бывают задачи, для которых существует решающий их алгоритм, но время его работы не есть $O(n^k)$ ни для какого фиксированного k. Такие алгоритмы имеют лишь теоретический интерес.

Для ряда задач не найдены полиномиальный алгоритмы и не доказано, что таких алгоритмов не существует. К таким алгоритмам относятся

так называемые *NP-полные* алгоритмы. Время их работы, по-видимому, не полиномиально, хотя проверить результат можно быстро.

Таких задач известно много. Например, задача о гамильтоновом цикле (простой цикл, проходящий через все вершины графа), задача коммивояжера, раскраска графа и т.д.

Вряд ли целесообразно искать алгоритм, решающий точно NP-полную задачу, лучше построить приближенный алгоритм.

Чтобы представить, чем отличается полиномиальные задачи от экспоненциальных, давайте посмотрим следующую таблицу. (Гэри, Джонсон. Вычислительные машины и труднорешаемые задачи)

Пусть N — некий условный максимальный размер задачи, решаемой на современной машине за 1 час. Сравнительная динамика увеличения этого размера при росте быстродействия для задач, решаемых полиномиальными и экспоненциальными алгоритмами, показана в таблице. Для упрощения обозначений мы используем одну и ту же букву N для всех типов алгоритмов.

Временная сложность алгоритма	На современных ЭВМ	На ЭВМ, в 100 раз более быстрых	На ЭВМ, в 1000 раз более быстрых
n	N	$100N$	$1000N$
n^2	N	$10N$	$31.6N$
n^3	N	$4,64N$	$10N$
n^5	N	$2,5N$	$3,98N$
2^n	N	$N+6,64$	$N+9,97$
3^n	N	$N+4,19$	$N+6,29$

Таким образом, если мы имеем квадратичный алгоритм, то при увеличении производительности ЭВМ в 1000 раз, мы получим выигрыш во времени только в 30 раз, а если наш алгоритм — экспоненциальный производительность увеличится весьма незначительно.

Грубо говоря, если на современной машине мы успеваем обработать по экспоненциальному алгоритму 100-битовое слово, то для того, чтобы обработать 110 бит, нам придется увеличить быстродействие в 1000 раз.

2.2 Построение алгоритмов. Анализ алгоритмов

Известно множество стандартных приемов, которые используются при построении алгоритмов. Вот некоторые из них:

- пошаговые алгоритм (или перебор);

- “разделяй и властвуй”;
- рекурсивные алгоритмы;
- перебор с возвратом;
- построение конечного автомата;
- динамическое программирование;
- жадные алгоритмы.

Одну и ту же задачу часто можно решить, используя различные алгоритмы. Полезно оценить, сколько же вычислительных ресурсов (время, память, внешняя память) требует тот или иной метод, тот или иной алгоритм. Нередко, одни алгоритмы требуют экспоненциального времени решения, тогда как другие, может быть дающие лишь приближенное решение, но зато требующие только полиномиального времени, причем с весьма малым показателем степени. (При анализе алгоритмов будем считать, что мы используем обычную однопроцессорную машину и не распараллеливаем вычисления.)

И построение алгоритмов, и их анализ будем проводить при решении конкретных задач.

Начнем с простой задачи поиска образца в строке. Допустим мы имеем некоторый текст T длины n символов и образец — тот текст который мы хотим найти в T: P длины $m < n$. Требуется найти все вхождения образца P в текст T. Будем говорить, что образец P входит в текст T со сдвигом s, если $0 \leq s \leq n - m$ и $T[s+1 \ldots s+m] = P[1 \ldots m]$. Говорят, что s — *допустимый сдвиг*. Таким образом, наша задача состоит в нахождении всех допустимых сдвигов.

Простейший, очевидный и прямолинейный путь — просмотр теста, чтобы найти первую букву образца, а затем проверка совпадения следующих букв. Такая программа проста и каждый может ее без труда написать. Очевидно трудоемкость алгоритма (в худшем случае) $O(m \cdot (n - m + 1)) \approx O(m \cdot n)$.

Можно ожидать, что есть и более эффективные способы, поскольку в простейшем случае информация о тексте T, получаемая при проверке сдвига s, никак не используется при проверке последующих сдвигов, а такая информация в поиске последующих сдвигов оказаться полезной.

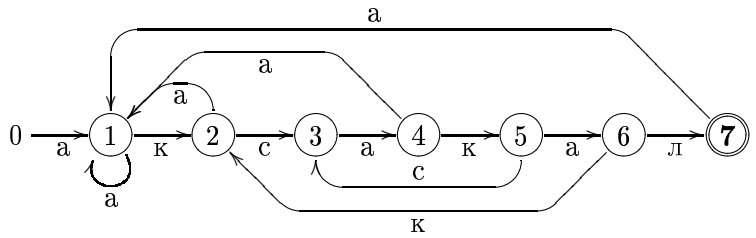
В алгоритмах поиска строк часто оказывается полезным использование конечных автоматов. Собственно уже рассмотренная нами раньше машина Тьюринга является конечным автоматом. Определим *конечный автомат (КА)* как пятерку $M = (Q, q_0, F, \Sigma, \delta)$, где

- Q — конечное множество состояний;
- $q_0 \in Q$ — начальное состояние;
- $F \subset Q$ — конечное множество заключительных, или *допускающих* состояний;
- Σ — конечное множество (алфавит) входных символов;
- δ — функция переходов $Q \times \Sigma \longrightarrow Q$.

Первоначально конечный автомат находится в состоянии q_0. По очереди читая символы из входной строки, например символ a, автомат переходит в состояние $\delta(q, a)$. Если $q \in F$, говорят, что он *допускает* прочитанную строку, если $q \notin F$, прочитанная часть строки *отвергается*.

Для каждого образца P можно построить конечный автомат, ищущий этот образец в тексте. Пусть, например, $P =$ `аксакал`.

Как построить конечный автомат для заданного образца, с этим мы разберемся чуть позже, а пока предположим, что конечный автомат уже построен. Вот он



Здесь состояния Q нашего конечного автомата обозначены кружками; $q_0 = 0$ — начальное состояние; $F = \{m\}$ — множество допускающих состояний содержит только один элемент; функция δ задана дугами $\delta(q, x)$ задает то состояние, в которое перейдет автомат после считывания символа x, находясь в состоянии q (отстутствие стрелки, отвечающей какому-либо символу, означает переход в состояние 0); Σ — можно считать, например, что множество входных символов содержит все однобайтные символы.

Показанный на рисунке автомат можно считать построенным, если мы построим функцию $\delta(q, x)$. Построим сначала некоторую вспомогательную функцию (так называемую *суффикс функцию*) $\sigma : \Sigma^* \to \{0, 1, \ldots, m\}$. Эта функция должна ставить в соответствие строке x длину максимального суффикса x, являющегося префиксом P:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

Поскольку $P_0 = \epsilon$ является суффиксом любой строки, σ определена на всем Σ^* ($Sigma^* = \Sigma \cup \{\}$. Если длина P равна m, то $\sigma(x) = m$ тогда и только тогда, когда P — суффикс x. Если $x \sqsupset y$, то $\sigma(x) \leq \sigma(y)$.

Таким образом, конечный автомат, который соответствует образцу $P[1..m]$, можно определить следующим образом:

- множество состояний: $Q = \{0, 1, \ldots, m\}$,
- начальное состояние $q_0 = 0$,
- множество допускающих состояний $F = \{m\}$,
- функция переходов δ определена следующим образом:

$$\delta(q, a) = \sigma(P_q a).$$

  Здесь P_q — строка $P[1..q]$.

Легко написать программу, которая вычисляет значение функции $\delta(q, a)$, для этого потребуется при самой грубой реализации время порядка $O(|\Sigma| m^3)$. Существует, однако, метод, с помощью которого функция переходов строится за время $O(|\Sigma| m)$. Таким образом, суммарная сложность поиска подстроки в тексте $O(|\Sigma| m + n)$. Впрочем, если текст большой, а образцы короткие, то первым слагаемым в этой оценке можно пренебречь. Объем памяти (для хранения полученной таблицы переходов — $O((|\Sigma| + 1)m)$)???

Мы рассмотрели предыдущий метод так подробно прежде всего потому, чтобы поближе познакомиться с конечными автоматами. Разумеется вышеизложенный алгоритм совсем не единственный, выполняющий поиск образца в тексте за приемлемое время. Алгоритм *Кнута–Морриса-Пратта* очень похож на предыдущий, но вместо функции перехода вычисляется префикс-функция (за время $O(m)$), которая несет информацию о том, где в образце P повторно встречаются различные префиксы этой строки(образца). Использование этой информации позволяет избежать проверки заведомо недопустимых сдвигов.

В алгоритме *Рабина-Карпа* строки P и T рассматриваются как большие числа в системе счисления с основанием $d = 255$:

$$P = P[1]d^{m-1} + P[2]d^{m-2} + \ldots + P[m]$$

$$T_s = T[s+1]d^{m-1} + T[s+2]d^{m-2} + \ldots + T[s+m]$$

Ищутся те сдвиги s, где $T_s = P$.

P по схеме Горнера вычисляется за время $O(M)$, за такое же время вычисляется T_0, остальные T_i вычисляются за время $O(1)$:

$$T_{s+1} = (T_s - T[s+1]d^{m-1})d + T[s+1+m]$$

Конечно, "числа" T_s и P будут настолько велики, что операции с ними придется реализовывать "вручную", и таким образом сложность операций умножения, сложения и т.д. будет намного больше, чем $O(1)$. К счастью, эту трудность можно преодолеть: будем производить все вышеприведенные вычисления по модулю некоторого фиксированного числа q (своего рода хэш-функция!). В этом то случае все числа не будут превосходить q и все T_j вместе с P будут действительно вычислены за время $O(n+m)$. Обычно в качестве q выбирают такое простое число, чтобы dq помещалось в машинное слово. Увы из равенства чисел T_s и P не следует равенства соответствующих строк, мы должны проверить совпадают ли строки на самом деле. Если не совпадают, то произошло так называемое холостое срабатывание. В худшем случае алгоритм требует времени $O((n-m+1)\cdot m)$, то есть такого же как и самый элементарный алгоритм (плюс затраты на арифметические действия). В среднем же ожидаемое время работы алгоритма $O(n) + O(m(v + n/q))$, где v — количество вхождений образца в текст.

По-видимому, наиболее эффективным алгоритмом поиска образцов является алгоритм *Бойера-Мура*. Этот алгоритм отличается от рассмотренных выше. Во-первых, он производит сравнение $P[1..m]$ и $T[s+1..s+m]$ справа налево. Во-вторых, он вводит две так называемые эвристики (эвристика стоп-символа и эвристика безопасного суффикса). Эти эвристики позволяют вовсе не рассматривать некоторые значения сдвига s. Если при проверке сдвига s обнаруживается что строка не совпадает с образцом, то каждая из эвристик указывает значение, на которое можно увеличить s, не опасаясь пропустить допустимый сдвиг. Из двух сдвигов ($j - \lambda[T[s+j]]$ для эвристики стоп-символа и $\gamma[j]$ для эвристики безопасного суффикса) выбирается наибольший.

Стоп-символ — это первый справа символ в тексте, отличный от соответствующего символа образца. Например, если стоп-символ выявляется при первом же сравнении ($P[m] \neq T[s+m]$) и не встречается нигде в образце, то сдвиг s можно сразу увеличить на m. Если стоп-символ встретился на j-й позиции, а в образе он появляется на k месте ($k = 0$, если он не появляется в образце вообще), то можно увеличить сдвиг s на $j - k$, если $k < j$.

Эвристика безопасного суффикса предлагает в случае несовпадения $P[j]$ и $T[s+j]$ $(j < m)$ увеличить сдвиг на

$$\gamma[j] = m - max\{k : 0 \leq k < mP[j+1..m] \sim P_k\}$$

где $\sim$ означает, что одна из подстрок является суффиксом другой.

В худшем случае время алгоритма Бойера-Мура $O((n-m+1) \cdot m + |\Sigma|)$. На практике, однако, этот алгоритм часто оказывается наиболее эффективным.

2.2.1 Построение оптимального по Парето множества в задачах управления системой КА

Рассмотрим задачу об управлении системой из N ИСЗ. Если в начальный момент элементы орбит всех наших спутников удовлетворяют некоторым условиям, позволяющим системе функционировать нормально (например для системы навигационных спутников с каждой точки поверхности Земли одновременно должны быть видимы не менее трех спутников), то под действием различных возмущений эти условия в конце концов перестают выполняться. Требуется с помощью каких-либо управляющих воздействий, например, включая в определенные моменты двигатели, восстанавливать нужную конфигурацию системы.

Естественно проводить оптимизацию по критериям, обеспечивающим экономичность управления. Для наших управляющих воздействий (включение двигателей в определенные моменты) можно ограничиться двумя минимизируемыми критериями. Один из этих критериев — количество затрачиваемого топлива, а другой — общее число коррекций. Последний критерий связан с надежностью системы.

Если бы мы оптимизировали лишь по одному критерию, задача решалась бы просто: либо численно, либо аналитически мы нашли бы значения параметров управления (моменты, в которые мы должны включать двигатели, дополнительные импульсы, которые мы придаем спутникам и т.д.), при которых требовалось бы, например, минимальные затраты топлива.

Если же таких критериев два или больше, то вообще говоря, мы не можем найти значения параметров, которые одновременно минимизировали бы и тот, и другой критерий. Дело тут не в астрономии или небесной механике, а в сути оптимизации. Действительно, если мы найдем параметры, минимизирующие решение по какому-то одному критерию, например, по топливу, то маловероятно, чтобы это решение было оптимально и по другому (независимому) критерию, например, по числу

коррекций. Исключением могут быть только зависимые критерии, скажем, если бы одним критерием служило количество топлива в тоннах, а другим — количество топлива в пудах.

В общем же случае мы можем получить только кривую (в случае двух критериев), состоящую из точек, которые нельзя улучшить по обоим критериям сразу. Такие решения называются *оптимальными по Парéто.*

Вернемся к нашим спутникам. Рассмотрим систему навигационных спутников типа Navstar (12-часовые ИСЗ на трех круговых орбитах с одинаковым наклонением и равноотстоящими восходящими узлами; на одной орбите находятся $N = 8$ спутников). Требование к системе — обеспечить видимость из любой точки поверхности Земли в любой момент времени четырех ИСЗ — выполняется при равномерном распределении спутников по орбитам.

Допустим мы умеем (подробности в других курсах) вычислять, когда и какие импульсы осуществлять, для системы из N спутников, в зависимости от их эволюции, то есть умеем вычислять и положения спутников, и импульсы для поддержания их конфигурации. Задача состоит в том, чтобы сделать это оптимальным по Парето (то есть по двум критериям сразу) образом. Ясно, что чем реже мы будем делать коррекции, тем сильнее должны быть импульсы. Вряд ли можно использовать очень малые импульсы очень редко.

Из чего выбирать? Будем "шевелить" положения каждого спутников, перебирая все допустимые конфигурации. Для первого спутника возьмем некоторый сектор $0 < \phi \approx 2\pi/N$ (чуть больше) и будем перебирать с некоторым шагом угол, определяющий положения спутника. Положение второго спутника будем перебирать в секторе от $h < h+\phi$ от первого спутника и т.д. Для каждой допустимой конфигурации вычисляются требуемые для поддержания рабочего состояния импульсы и вычисляются два критерия: F_1 — суммарный импульс и F_2 — общее число коррекций (в единицу времени).

Итак, мы знаем как перебирать, мы умеем поддерживать систему в рабочем состоянии и вычислять значения критериев. Нужно построить множество оптимальных по Парето точек.

В процессе перебора формируется массив оптимальных точек. Чтобы поиск оптимальных точек осуществить наиболее эффективным способом, этот массив следует упорядочить по одному из критериев, например, по F_1. Для элементов этого массива требуется определить лишь три операции — поиск места элемента в массиве, включение нового элемента в массив, исключение элемента из массива. Все эти операции особенно эффективно реализуются, если используется списковая структура. То-

гда для каждой новой точки $z = \{x_i, y_{-j}\}$ находится место в списке, такое, что $F_1(z_1) \leq F_1(z) \leq F_1(z_2)$, где z_1 и z_2 — два последовательных элемента списка. Если $F_2(z_1) \geq F_2(z) > F_2(z_2)$, то точка z должна быть включена в список, а все последующие точки $z_k = z_2, z_3, \ldots$, для которых $F_2(z_k) > F_2(z)$, должны быть исключены. Отметим, что машинное время в основном тратится на перебор. Для шага в 2° общее число рассмотренных точек $N \approx 1.6 \cdot 10^{11}$.

На рисунке изображено множество оптимальных по Парето значений критериев для системы из восьми ИСЗ, $\phi = 50^\circ$. Возможные значения критериев расположены выше и правее кривой и на кривой.

2.2.2 Работа с деревьями в Treecode

Решение задачи N тел, даже численное, весьма трудоемкая задача. Особенно, если требуется высокая точность (эволюция солнечной системы на временах порядка ее жизни), или нас интересует системы с *очень* большими N (до 10^8 и больше). Ясно, что если мы *честно* будем вычислять силы взаимодействия для всех тел, то ресурсоемкость такого алгоритма будет $O(N^2)$. Конечно это не экспоненциальная задача, но при рассматриваемых (больших) N даже небольшое снижение ресурсоемкости алгоритма, например, до $N \ln N$, может позволить нам прослеживать эволюцию системы, состоящей из существенно бóльшего числа тел, возможно за счет некоторой потери точности, с чем можно, однако смириться, если такая потеря не окажет заметного влияния на качественное изучение эволюции нашей системы.

Именно для решения таких задач и был разработан иерархический метод. Суть его состоит в следующем. Каждый временной шаг состоит из двух фаз:

1. строится дерево нашей системы: корень дерева — вся система, листья — частицы;

2. построенное дерево обходится и при этом с нужной точностью вычисляются силы для каждой частицы.

Начинается дерево с большой ячейки (куба в трехмерном случае или квадрата — в двумерном, для иллюстрации мы будем использовать именно двумерный), содержащей всю систему N тел. Это — корень дерева. Далее каждую ячейку делим на 8 равных кубов в трехмерном случае или на 4 квадрата в двумерном. Получим таким образом узлы следующего уровня. (В этой задаче деревья — не бинарные!) Если оказывается, что в какой-то из полученных ячеек частиц нет, такая ячейка в дерево

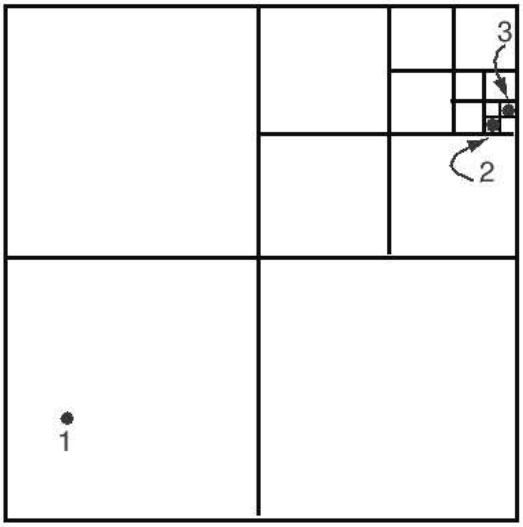


Рис. 1: Построение дерева Барнса-Хата для системы из 3 частиц (двумерный случай).

не включается, в противном случае процесс рекурсивно продолжается дальше, пока в ячейке не окажется ровно одна частица. В результате получается дерево Барнса-Хата. В пространственном случае каждый узел будет иметь в общем случае восемь поддеревьев, так что можно назвать наши деревья “восьмеричными” (или “октарными”).

Ясно, что при вычислении силы, действующей на частицу 1, можно, если тела 2 и 3 достаточно далеки от тела 1, заменить тела 2 и 3 фиктивным телом ячейки с массой, равной сумме масс тел 2 и 3 и находящейся в центре их масс. Если требуется бóльшая точность, то можно заменить их диполем, добавив к предыдущему еще некоторые дополнительные члены. Если же тела близки (как 2 и 3 в нашем случае), придется вычислить силу честно. Конечно, необходимо определить критерии, когда мы должны честно вычислять силу, а когда мы можем целую ветвь дерева заменить всего лишь одной фиктивной массой. Рассмотрение таких критериев выходит за рамки данного курса, но если они установлены, вычисление силы, скажем для тела p, выполняется за один обход дерева, начиная с корня.

В каждом узле q, в который мы попадаем при обходе, мы можем встретиться с тремя возможностями: Случай первый: q — тело. В этом случае сила взаимодействия p и q должна непосредственно учитываться, точнее ссылка на q добавляется в список взаимодействий тела p (чтобы впоследствии вычислить силу). Если q — не тело, то необходимо выделить еще два случая. Случай второй: если q достаточно далека от p (до-

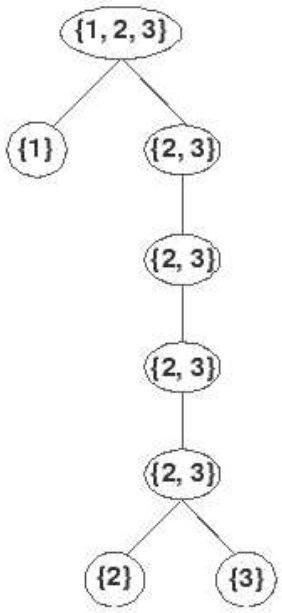


Рис. 2: Дерева Барнса-Хата
для приведенной системы.

статочно отделена от p), то в список взаимодействий добавляется сама ячейка q. Случай третий: ячейка q слишком близка к p, чтобы вычислить силу в этом случае необходимо проверить всех потомков q.

Подчеркнем, что только в первом случае мы "честно" вычисляем силу; во втором случае мы заменяем ВСЕ поддерево одной фиктивной массой, или мультиполем, что тоже не требует больших вычислений; в последнем случае нам еще только предстоит столкнуться с одним из первых двух случаев, и чаще всего это будет именно второй случай, когда мы исключаем из рассмотрения все поддерево.

Конечно, для реального решения задачи придется рассмотреть еще много деталей. В частности, и дерево можно строить по-другому. Например, идя снизу и объединяя попарно ближайшие тела (вообще-то считается, что это хуже). Можно рассматривать различные критерии отделенности ячеек, применять различные интеграторы (как правило, применяется симплектический leap-frog интегратор), но основное уже сделано: использование иерархической структуры позволяет нам сильно облегчить задачу и снизить ее ресурсоемкость с $O(N^2)$ до $O(N \ln N)$ (в некоторых случаях утверждается, что даже до $O(N)$).

2.2.3 Операция с символьными строками. Обратная польская запись

Давайте построим дерево для вычисления арифметического выражения:

```
A*B+C
```

(Рис.)

В таких языках, как C, Pascal, Fortran, любое арифметическое выражение преобразуется в последовательность инструкций процессора, вычисляющих это выражение. То есть, на самом деле это выражение приводится к более прозрачному виду.

Его (это дерево) можно записать, например, так (см. представления деревьев).

```
(+ (* A B) C)
```

Такая запись, когда сначала записываются операции, затем идут операнды, называется префиксной записью. Символы "+" и "*" здесь можно трактовать как имена функций. Можно заменить их более человеческими именами:

```
(add (multiply A B) C)
```

Выражения в таком виде можно практически без обработки вычислять на компьютере. Действительно, вызываем функцию "add", и если какой-то из операндов тоже записан в виде вызова функции (у нас это "multiply"), то вызываем сначала функцию для вычисления операндов, и т.д. Снова рекурсивное определение и рекурсивный процесс!!! Собственно в некоторых языках программирования именно так и задаются вычисления (и не только числовые), программисты просто используют префиксную, а не привычную инфиксную запись, зато транслятору (или интерпретатору) вычислить такое выражение, не преобразуя его к какому-либо иному виду, не составляет труда.

Обрабатывать арифметические выражения станет еще проще, если знак операции (или имя функции) поместить не перед операндами, а после них:

```
A B * C +
```

Ряд языков (и ряд калькуляторов!) использует именно такое представление арифметических выражений. Такое (постфиксное представление) называют *обратной польской записью* (реже префиксное представление называют *прямой польской записью*)

Реализация вычисления записанного в обратной польской записи выражения удивительно проста. Если встречается операнд, то его значение помещается на стек (вспомните, что это такое). Если операция (имя функции), то со стека снимается столько операндов, сколько требует данная операция, и выполняется соответствующая функция.

Реализация такого механизма столь проста, что языки его использующие, были чрезвычайно популярны во времена, когда оперативная память составляла несколько десятков килобайт. Один из таких языков широко используется и сейчас (Postscript).

3 Языки программирования и программное обеспечение

3.1 Модель фон Неймана

В 1946(?5) году (???1954???) Джон фон Нейман, (с соавторами???) описал архитектуру некоторого абстрактного вычислителя, который сейчас принято называть *машиной фон Неймана.* Эта машина является *абстрактной моделью* ЭВМ, однако, эта абстракция отличается от абстрактных исполнителей алгоритмов (например, машины Тьюринга). Если машину Тьюринга принципиально нельзя реализовать из-за входящей в ее архитектуру бесконечной ленты, то машина фон Неймана не поддается реализации, так как многие детали в архитектуре этой машины намеренно *не конкретизированы* (чтобы не сковывать творческого подхода к делу у инженеров-разработчиков новых ЭВМ).

В машине фон Неймана зафиксированы особенности архитектуры, которые, по мнению авторов, должны быть присущи всем компьютерам. Все современные компьютеры по своей архитектуре отличаются от машины фон Неймана, но эти отличия, так сказать, второго порядка. Их удобно рассматривать именно как отличия от базовой архитектуры. Принципы, сформулированные фон Нейманом, до сих пор еще лежат в основе архитектуры большинства ЭВМ.

Вот схема машины фон Неймана (толстые стрелки — потоки команд и данных):

Вот эти общие принципы:

1. Принцип линейности и однородности памяти: структурно основная память состоит из перенумерованных ячеек; процессору в любой момент времени доступна любая ячейка. (Т.о. можно давать имена областям памяти, чтобы впоследствии можно было обращаться к

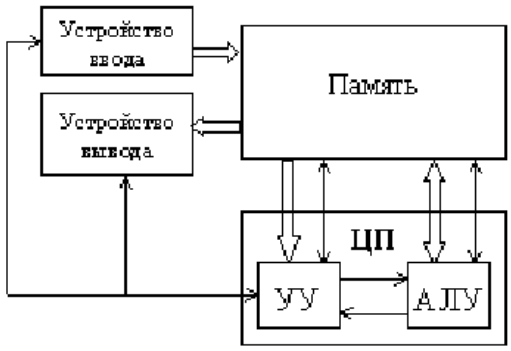


Рис. 3: Схема машины фон Неймана

этим областям.) Время доступа к ячейки одинаково для всех ячеек памяти (время записи и время чтения могут различаться);

2. Принцип неразличимости команд и данных: программы и данные хранятся в одной и той же памяти. Программа в процессе выполнения также подвергнуться изменению. Команды одной программы могут быть получены, как результат исполнения другой.

3. Принцип автоматической работы. Программа состоит из набора команд, которые выполняются процессором (УУ) *автоматически* в определенной последовательности (если командой не предписывается эту последовательность изменить, программа — набор записанных в памяти машинных команд, описывающих шаги некоторого алгоритма, т.е. программа — это запись алгоритма на языке машины. Язык машины — набор всех возможных команд).

4. Принцип последовательного выполнения команд: Устройство Управления выполняет некоторую команду от начала до конца, а затем выбирает для выполнения следующую команду.

Конечно, современные ЭВМ в какой-то степени нарушают все эти принципы. Существуют машины, которые различаются команды и данные (в ячейках памяти присутствует специальный *тег*). В известной мере нарушается принцип однородности и линейности памяти (адрес памяти представляется двумя числами «сегмент-смещение», или память вообще может не иметь адресов, т.н. ассоциативная память). Принцип последовательного выполнения команд нарушается в многопроцессорных (векторных) архитектурах. Нарушаются и некоторые принципы, которые фон Нейман считал самоочевидными и поэтому не формулировал их явно. Например, в архитектуре фон Неймана предполагается, что во

время выполнения программы не меняются ни число узлов компьютера, ни взаимосвязи между этими узлами, таким образом, если Вы вставляете в дисковод дискету, то тем самым Вы нарушаете этот принцип. Тем не менее, все эти отличия не нарушают, а лишь расширяют общее представление об архитектуре.

Явно *не фон-неймановские* компьютеры, существуют в теории, например, квантовые компьютеры, но большинство, как уже было сказано, — компьютеры с архитектурой фон Неймана, либо отличающиеся от них лишь некоторым числом деталей.

3.2 Императивные языки программирования

Итак, поскольку бóльшая часть компьютеров следует архитектуре фон Неймана, то естественно, что бóльшая часть языков программирования языков программирования разрабатывалась на основе этой архитектуры. Такие языки называются *императивными. Директивные или процедурные* — синоним *императивных.*

Несколько слов о том, зачем вообще нужно знакомиться с различными концепциями языков программирования. Причин для этого много

- Язык, который используется для решения той или иной задачи, естественно налагает ограничения на используемые структуры управления, структуры данных, абстракции. Выразительная сила языка влияет на глубину наших мыслей. Осознав разнообразие свойств языков программирования, мы получим больше возможностей для выражения своих идей, даже если те или иные средства отсутствуют в используемом языке, мы можем их смоделировать с помощью доступных средств.

- Чем больше языков мы знаем, тем более обоснованный выбор наиболее подходящего языка мы делаем, решая стоящую перед нами задачу.

- Чем больше языков мы знаем, тем легче нам понять концепции новых языков программирования (аналогия с естественными языками).

- Разобравшись почему тот или иной язык разработан так, а не иначе, мы лучше будем представлять, как наиболее рационально использовать этот язык.

- Критический разбор языков программирования помогает при конструировании сложных систем.

- Познакомившись с концепциями языков программирования можно (относительно) объективно оценить текущее состояние используемого программного обеспечения.

Конечно, в задачу данного курса не входит сравнительное изучение языков программирования, но познакомиться с существующими парадигмами программирования. (Парадигма — способ мышления, не связанный с конкретным языком, свод норм мышления, модель постановки проблем и их решения.) В своей алгоритмической части современные языки поддерживают несколько парадигм программирования, самыми распространенными являются императивные языки.

Главными элементами императивных языков программирования, являются переменные, которые моделируют ячейки памяти; операторы присваивания основаны на пересылке данных; итеративная форма повторений является наиболее эффективным методом в архитектуре фон Неймана. Операнды выражений передаются из памяти в процессор, а результат вычисления возвращается в ячейку памяти (левая часть оператора присваивания). Команды хранятся в соседних ячейках памяти, что облегчает использование итерации, и, наоборот, использование рекурсии, даже в случаях, когда это использование более естественно, зачастую не столь эффективно.

Первым таким языком был созданный Конрадом Цузе в 1945 году язык Plankalkül. Наиболее широко известен Фортран. Примеры императивных языков программирования привести не трудно. Эти языки известны каждому. Кроме Фортрана, это Паскаль, C, Алгол, Basic и т.д.

Директивная программа предписывает, как достичь заданную цель, описывая по шагам все (промежуточные) действия. Заметим, что даже определение алгоритма, приведенное нами в предыдущей части, тяготеет к этой классической (императивной) парадигме.

Рассмотрим пример. Предположим, вам надо пройти в городе из пункта А в пункт Б. Директивная программа — это список команд примерно такого рода:

от пункта А по ул. Садовой на север до Сенной площади, оттуда по Московскому проспекту два квартала, потом повернуть налево и идти до Серпуховской улицы, по этой улице направо и по левой стороне до дома 12, который и есть пункт Б.

В директивной программе действия задаются явными командами, подготовленными ее составителем. Исполнитель же просто им следует. Хотя команды в различных языках директивного программирования и выглядят по-разному, все они сводятся либо к присваиванию какой-

нибудь переменной некоторого значения, либо к выбору следующей команды, которая должна будет выполняться. Присваиванию может предшествовать выполнение ряда арифметических и иных операций, вычисляющих требуемое значение, а команды выбора реализуются в виде условных операторов и операторов повторения (циклов).

Для классических директивных языков характерно, что последовательность выполняемых команд совершенно однозначно определяется ее входными данными. Как говорят, поведение исполнителя императивной программы полностью детерминировано.

Собственно наш мир локально императивен. Если взять достаточно узкую задачу, то ее можно вполне легко описать методами последовательного программирования. Практика показывает, что более сложные императивные программы (компиляторы, например) пишутся и отлаживаются долго (годами). Переиспользование кода и создание предметно-ориентированных библиотек упрощает программирование, но ошибки в реализации сложных алгоритмов проявляются очень часто.

Императивное программирование наиболее пригодно для реализации небольших подзадач, где очень важна скорость исполнения на современных компьютерах. Кроме этого, работа с внешними устройствами, как правило, описывается в терминах последовательного исполнения операций ("открыть кран, набрать воды"), что делает такие задачи идеальными кандидатами на императивную реализацию.

Императивные языки хорошо известны и мы не будем углубляться в их анализ.

3.3 Декларативные языки

Функциональное программирование, как и другие модели “неимперативного” программирования, обычно применяется для решения задач, которые трудно сформулировать в терминах последовательных операций. Практически все задачи, связанные с искусственным интеллектом, попадают в эту категорию. Среди них следует отметить задачи распознавания образов, общение с пользователем на естественном языке, реализацию экспертных систем, автоматизированное доказательство теорем, символьные вычисления. Эти задачи далеки от традиционного прикладного программирования.

Пример о движении из пункта A в пункт B из предыдущего раздела в декларативном изложении будет выглядеть так:

план города, в котором указаны оба пункта, плюс правила уличного движения. Руководствуясь этими правилами и планом города, курьер сам найдет

путь от пункта А к пункту Б.

Декларативные программы не предписывают выполнять определенную последовательность действий, в них лишь дается разрешение совершать их. Исполнитель должен сам найти способ достижения поставленной перед ним составителем программы цели, причем зачастую это можно сделать различными способами — детерминированность в данном случае отсутствует.

Согласитесь, такая возможность впечатляет.

Функциональная программа состоит из совокупности определений функций, которые в свою очередь представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. При этом функции часто либо прямо, либо опосредованно вызывают сами себя (рекурсия).

Каждая функция возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается; этот процесс повторяется до тех пор, пока начавшая процесс вычислений функция не вернет конечный результат пользователю.

"Чистое" функциональное программирование не содержит оператора присваивания, в нем вычисление любой функции не приводит ни к каким побочным эффектам, отличным от собственно вычисления ее значения. Разветвление вычислений основано на механизме обработке аргументов условного предложения, а циклические вычисления реализуются с помощью рекурсии.

Отсутствие оператора присваивания делает переменные, используемые в функциональных языках программирования, очень похожими на переменные в математике — получив однажды свои значения, они больше никогда их не меняют. Отсутствие побочных эффектов в процессе вычисления функций приводит к тому, что порядок выполнения отдельных фрагментов программы не существенен — итоговое значение в любом случае будет одинаковым.

Функциональное программирование весьма красиво и иногда в качестве первого языка программирования, изучаемого студентами, выбирается Haskell или Lisp. Для успешного овладения данным стилем программирования, впрочем, необходимо весьма глубокое понимание многих разделов математики.

В чистом LISP'е существует только два типа структур данных: атомы и списки. Атомы — либо символы, либо числовые константы. Списки определяются круглыми скобками:

```
(A B C D)
```

или

```
(A (B C)  D (E (F G)))
```

Списки, как правило, представляются в виде односвязных линейных списков, каждый узел которых содержит два указателя и представляет собой элемент списка. Первый указатель указывает на атом или первый узел подсписка, второй указатель указывает на следующий элемент списка.

Список `(A B C D)` можно интерпретировать, как данные, тогда это список из четырех элементов, а можно как программу. В последнем случае функция `A` применяется к трем параметрам `B`, `C` и `D`.

Собственно мы описали весь синтаксис языка LISP.

Поразительно, что такой простой язык на протяжении четверти века доминировал в области искусственного интеллекта, да и сейчас еще остается самым распространенным в этой сфере.

Вот пример LISP-программы. Здесь определяется функция, сравнивающая два списка и возвращающая `TRUE`, если списки идентичны, и `NIL`, в противном случае

```
(DEFUN equal_lists (l1 l2)
 (COND
   ((ATOM l1) (EQ l1 l2))
   ((ATOM l2) NIL)
   ((equal_lists (CAR l1) (CAR l2))
     (equal_lists (CDR l1) (CDR l2)))
   (T NIL)
 )
)
```

На LISP'е реализованы многие системы аналитических вычислений: Reduce, Macsyma, многие программы автоматического проектирования, например AutoCAD, и много других программ, включая редактор всех времен и народов Emacs.

Функциональные языки программирования: Lisp, Scheme, ML, Haskell.

Чтобы программировать на LISP'е недостаточно познакомиться с его правилами и функциями. И синтаксис, и семантика LISP'а, как мы видели, просты. Однако парадигма функциональных языков настолько отлична от парадигмы привычных нам языков императивных, что главная трудность заключается в том, чтобы "думать" по-лисповски, "повернуть свои мозги", приучить себя мыслить по-другому. Нужно ли такое

“преодоление себя”? Да, многие (трудные) задачи решаются проще всего именно с использованием функциональных языков. Например системы аналитических вычислений. Пусть в выражение (символьное) $1 - X^2$ мы хотим вместо X подставить, например, $\cos(\alpha)$. Это обычное действие при работе с символьными выражениями. Если наше первоначальное выражение представлено списком (MINUS 1 (POWER X 2)), то все, что нам нужно сделать, это вместо атома X подставить список (COS α). Ну возможно, потом еще обработать получившийся список и получить, если соответствующее правило определено, $\sin(\alpha)$. В рамках парадигмы LISP'а такие действия описываются чрезвычайно просто.

Интересно, что на протяжении многих лет (вечность по компьютерным меркам) во многих университетах США LISP был первым языком, с которого студенты начинали изучение программирования.

Замечание о том, что при переходе от императивных языков к функциональным необходимо переучиваться “думать”, в полной мере относится и к языкам, относящимся к другим парадигмам, например, к языкам *логического программирования.*

3.4 Языки логического программирования

Языки логического программирования, самым известным из которых является Пролог, конечно можно было бы отнести к декларативным языкам, в том смысле, что в программах указывается лишь описание желаемого результата, а не детальная процедура его получения, но отличие от функциональных языков столь велико, что их можно выделить в отдельный класс.

И синтаксис, и семантика логических языков отличается отличается от синтаксиса и семантики и императивных, и функциональных языков.

Логическое программирование — это использование формальной логической записи для описание программы. В качестве такой записи используется исчисление предикатов. Исчисление предикатов обеспечивает основную форму для передачи информации, а метод доказательства, названные резолюцией и разработанный Робинсоном в 1965 году, предоставляет способы логического вывода.

Программы языка Пролог состоят из набора утверждений. Утверждения могут быть двух видов: факты и правила.

```
раздел(астрометрия, астрономия).
раздел(астрофизика, астрономия).
раздел(небесная механика, астрономия).
предмет(движение тел, небесная механика).
```

```
предмет(положение звезд, астрометрия).
```

Эти факты утверждают, что выполнено отношение `раздел` между объектами `астрофизика` и `астрономия` и между объектами `небесная механика` и `астрономия` и т.д.

А вот правила:

```
астрономическая_наука(X) :- раздел(X, астрономия).
астрономия_изучает(X) :- предмет(X, Y), раздел(Y, астрономия).
```

Первое правило утверждает, что если `X` — раздел астрономии, то `X` является астрономической наукой, а второе — что если `X` предмет науки `Y` **и** `Y` — раздел `астрономии`, то астрономия изучает `X`.

Имеющиеся факты и правила можно использовать для того, чтобы задать вопрос:

```
?- астрономическая_наука(астрология).
```

и получить ответ

```
нет
```

На вопрос

```
?- астрономическая_наука(X).
```

получим ответ

```
X = астрометрия;
X = астрофизика;
X = небесная механика;
```

А на вопрос:

```
?- астрономия_изучает(движение тел).
```

естественно получим ответ

```
да
```

В Прологе присутствуют также основные операции работы со списками.

Несмотря на ряд проблем, возникающих при использовании Пролога, логическое программирование способно работать во многих приложениях. Например, логическое программирование естественным образом соответствует нуждам реализации систем управления реляционными базами данных.

Реально используется язык Пролог для создания экспертных систем, где база данных (или база знаний) может быть неполной.

Подходит Пролог для обработки текстов на естественных языках.

И вообще концепция логического программирования интересна и красива сама по себе, а развитие компьютерной техники и разработка новой техники логического вывода может привести к развитию языков логического программирования, которые позволят эффективно решать задачи, указывая только *что*, а не *как* следует сделать.

3.5 Программирование в ограничениях

Программирование в ограничениях (constraint programming) — достаточно новое направление в декларативном программировании. Появилось оно во многом в результате развития систем символьных вычислений, искусственного интеллекта и исследования операций.

Программирование в ограничения — подход, в котором в программе определяется тип данных решения, предметная область решения и ограничения на значение искомого решения. Решение находится системой.

Программирование в ограничениях — это программирование в терминах "постановок задач".

Постановка задачи — это конечный набор переменных $V = \{v_1, \ldots, v_n\}$, соответствующих им конечных (перечислимых) множеств значений $D = \{D_1, \ldots, D_n\}$, и набор ограничений $= \{C_1, \ldots, C_m\}$. Ограничения представлены как утверждения, в которые входят в качестве "параметров" переменные из некоторого подмножества $V_j, j = 1..m$ набора V. Решение такой задачи — набор значений переменных, удовлетворяющий всем ограничениям C_j.

Синтаксически такую постановку задачи можно записать как "правило" для "типизированного" Пролога:

```
problem(V1:D1, ..., Vn:Dn) :-
    C1, ... Cm.
```

Семантически, однако, программирование в ограничениях отличается от традиционного логического программирования в первую очередь тем, что исполнение программы рассматривается не как доказательство утверждения, а нахождение значений переменных. При этом порядок "удовлетворения" отдельных ограничений не имеет значения, и система программирования в ограничениях, как правило, стремится оптимизировать порядок "доказательства" утверждений с целью минимизации отката в случае неуспеха. С этой целью набор ограничений может быть соответствующим образом преобразован — по правилам, аналогичным

правилам Пролога. Любую задачу можно рассматривать как ограничение: “значения переменных должны быть решением этой задачи”. Часто о программировании в ограничениях говорят исключительно как о “дополнительной” ветви логического программирования.

В качестве примера, мы можем задать системе программирования в ограничениях следующий запрос:

```
? (X : integer) X>1, member(X, [1,2,3]).
```

Типичная Пролог-система на таком запросе выдаст ошибку: является неинициализированной переменной, и его нельзя сравнивать с числом 1. Система, поддерживающая программирование в ограничениях, воспримет эти “утверждения” как ограничения (а не как цели, которые требуется доказать), и выдаст нам требуемые решения: $= 2$ и $= 3$.

Системы символьных вычислений нередко позволяют использовать “допущения” — по сути, те же ограничения. И на следующий (простой) запрос:

```
assume X>0.
when X+1<10 ?
```

выдавать ответ:

```
X in (0..9).
```

Как правило, такие системы могут доказывать достаточно нетривиальные математические утверждения, выводя “минимальными необходимыми ограничениями”, и проверяя эти ограничения на совместность.

В задачах исследования операций и реализации искусственного интеллекта часто используется некоторое “пространство решений”, сужением которого достигается необходимый результат. Такие “сужения” естественным образом представляются как ограничения.

3.6 Объектно-ориентированное программирование

Объектно-ориентированное программирование появилось на свет божий из недр событийно-управляемого программирования.

В событийно-управляемом программировании отдельные процессы максимально автономны, единственное средство общения между ними — посылка сообщений (порождение новых событий). События могут быть как общими для всей системы, так и индивидуальными для одного или нескольких процессов. В таких терминах достаточно удобно описывать,

например, элементы графического интерфейса пользователя, или моделирование каких-либо реальных процессов (например, управление уличным движением) — так как понятие события является для таких задач естественным.

Поддержка объектно-ориентированного программирования в настоящее время включена во все популярные языки, как императивные (Python, C++, Java), так и декларативные (CLOS, Prolog++).

Несмотря на такую поддержку парадигма объектно-ориентированного программирования отличается от всех уже рассмотренных и и, чтобы эффективно использовать его необходимо "думать объектно-ориентировано".

Основу ООП обеспечивают три свойства

- абстрактные типы данных;
- наследование;
- полиморфизм.

Абстрактные типы данных

Абстракция — представление о некотором объекте, содержащее только существенные в данном контексте свойства. Абстракция позволяет объединить экземпляры объектов в группы, внутри которых можно не рассматривать общие свойства, абстрагироваться от них, а изучать только свойства, отличные для отдельных элементов группы. Таким образом, абстракция, позволяя программисту сосредоточиться лишь на существенных свойствах объекта, является средством против сложности программирования, она позволяет сделать большие и сложные программы более управляемыми.

Собственно один вид абстракции вам хорошо знаком. Это абстракция процесса. Действительно, вызов подпрограммы, скажем подпрограммы сортировки, не зависит от алгоритма сортировки, используемого в данной подпрограмме, является абстракцией реального процесса сортировки. Все, что нам нужно знать при вызове этой подпрограммы, — это имя упорядочиваемого объекта, тип его элементов и тот факт, что этот самый вызов выполнит требуемую сортировку.

Собственно эта абстракция, использование подпрограмм, хорошо знакома. Но давайте вспомним (из первой части), что данные неразрывно связаны с теми операциями, которые для них определяются, то есть абстракция данных связана с абстракцией процессов, реализующих эти операции.

Теперь представим, что мы пишем большую программу. Если размер нашей программы превышает несколько тысяч строк, мы сталкиваемся

с необходимостью разделить ее на группы логически связанных подпрограмм и данных (такие группы часто называют *модулями*). Это позволяет нам, во-первых, лучше организовать и поддерживать логическую структуру программы и управлять ею и, во-вторых, избежать повторной компиляции неизменных частей программы. Способ объединения в единое целое подпрограмм и данных известен как *инкапсуляция*. Инкапсуляция является основой абстрактной системы.

Абстрактный тип данных — это инкапсуляция, которая содержит только представления (структуры) данных одного конкретного типа и подпрограммы, которые выполняют операции с данными этого типа. С помощью управления доступом несущественные детали описания типа можно скрыть от внешних модулей, использующих данный тип, внешний модуль, как правило, даже не знает, как реально представляется используемый тип. Экземпляр абстрактного типа данных называется *объектом*.

Абстрактный тип данных должен удовлетворять следующим условиям:

- представление (определение типа) и операции над объектами данного типа содержатся в одной синтаксической единице (такой единицей может быть модуль или другая конструкция, определяемая тем или иным языком), создавать переменные этого типа можно и в других модулях;

- представление объектов данного типа скрыто от программных модулей, использующих этот тип, над такими объектами можно производить лишь те операции, которые прямо предусмотрены в определении типа.

Наследование

Допустим, мы разработали какой-то сложный тип данных, описали множество операций над этим типом данных и хотим использовать уже созданные программы в другой, близкой задаче. Проблема заключается в том, что в новой задаче и свойства и операции уже разработанного типа данных не вполне подходят для решения нашей новой задачи. Уже имеющиеся типы необходимо как-то, пусть немного, но модифицировать. Модификации, хоть и небольшие, не так то просто выполнить, для этого надо разобраться в уже существующем коде. Кроме того, как правило модификации влекут за собой изменения в программах, использующих этот тип данных.

И еще. По крайней мере до сих пор, все наши типы данных являлись независимыми, находились на одном уровне иерархии. В то же время

наша задача может содержать связанные между собой объекты, находящиеся в некоторой субординации друг с другом.

Эти проблемы позволяет решить второе важное свойство ООП — *наследование*.

Абстрактные типы данных обычно называются *классами*. Классы представляются в виде иерархической древовидной структуры, в которой классы с более общими чертами располагаются в корне дерева, а специализированные классы и в конечном итоге индивидумы располагаются в ветвях. На рисунке показана одна из возможных иерархий классов, включающая класс "Солнечная система" на самом верхнем уровне иерархии, объекты класса "Планета" — на втором уровне, и объекты класса "Спутники" — на нижнем.

Рис. 4: Наследование

Класс, который определяется через наследование от другого класса, называется *производным* классом или *подклассом*. Класс, от которого производится новый класс, называется *родительским* классом. Подпрограммы, определяющие операции над объектами класса, называются *методами*. Вызовы методов иногда называют *сообщениями* (вспомним событийно-управляемое программирование). Весь набор методов объекта называют *протоколом сообщений* или *интерфейсом*. Таким образом вычисления в объектно-ориентированном программе определяются сообщениями, передаваемого от одного объекта другому.

В простейшем случае класс наследует все сущности (переменные и методы) родительского класса, этим наследованием, однако, можно управлять. В дополнение к наследуемым сущностям производный класс может добавлять новые и модифицировать методы. Новый метод *замещает* наследуемую версию метода.

Полиморфизм

Полиморфизм это концепция, позволяющая иметь различные реализации для одного и того же метода. Их выбор осуществляется в зависимости от типа объекта, переданному методу при вызове.

Это свойство позволяет подменять один объект другим, способным

обрабатывать те же сообщения. Таким образом обеспечивается более легкое расширение программных средств при их разработке и поддержке.

Объектно-ориентированное программирование активно используется как средство проектирования сложных систем и моделирования, например, в языке UML.

3.7 Параллельное программирование

Внимательный читатель (слушатель?) может заметить некоторую неувязку в изложении различных парадигм программирования. Действительно, парадигма императивного программирования основана на архитектуре фон Неймана. Даже определение алгоритма несет оттенок именно этой архитектуры. Вспомним: *Алгоритм — набор конечного числа правил, задающих последовательность выполнения операций для решения задачи.* То есть, согласно этому определению, алгоритм указывает *как* решать задачу. Языки логического программирования ставят во главу угла не *как* следует решать задачу, а *что* надо получить. А для этого необходимо сформулировать *что* является объектом решения, то есть построить модель исследуемых объектов. Тем не менее, пока архитектура фон Неймана оставалась по существу единственной распространенной архитектурой, императивное, последовательное программирование наиболее полно отвечало используемым компьютерам.

Развитие компьютерной техники, однако, привело к появлению компьютеров (CDC, середина шестидесятых), в которых имелась возможность обрабатывать данные параллельно, выполняя одну и ту же операцию одновременно над целым массивом (вектором) значений — появилась "векторная" архитектура. Появились машины с несколькими процессорами, несколько компьютеров (в том числе и многопроцессорных), объединенных высокоскоростной коммуникационной сетью, составляли одну "параллельную" машину. Именно компьютеры с несколькими векторными процессорами и высокоскоростной общей памятью первоначально назывались *суперкомпьютерами.* Сейчас под суперкомпьютером подразумевают вычислительную систему, входящую в список TOP500 самых высокопроизводительных компьютеров мира (самый медленный — 245/384 Гигафлопс, здесь первое число — максимальная достигнутая производительность, второе — теоретическая; а самый быстрый Earth-Simulator/5120 фирмы NEC — 35860/40960 Гигафлопс, работает в Японии; из TOP500 два компьютера работают в России, один, собственной разработки, в Объединенном суперкомпьютерном центре — 734.60/1024 Гигафлопс — на 95 месте, второй — HP SuperDome 750, 345/576 Гигафлопс, 357 место, в СберБанке; данные на 16.11.2003, 22 версия TOP500).

Использование таких параллельных компьютеров, даже если они не входят в список TOP500, требуют своего подхода, своей парадигмы параллельных вычислений. Если обратиться к уже рассмотренным парадигмам, то можно сделать вывод, что императивное программирование, по крайней мере в своем чистом виде, никак не подходит к организации параллельной работы. А вот функциональное программирование имеет черты для такой работы весьма ценные. Вспомним, что в LISP'е нет операторов присваивания, нет побочных эффектов, а переменные, раз получившие значение, уже не меняют его. Все это облегчает параллельные вычисления. В Прологе тоже не важен порядок разрешения правил, и поиск подходящих правил и фактов легко организовать параллельно.

Однако это лишь общие соображения. Для того, чтобы познакомиться с парадигмой параллельных вычислений поближе, нужно и более детально изучить архитектуру параллельных компьютеров, и рассмотреть различные подходы к организации параллельных вычислений.

3.7.1 Архитектуры параллельных компьютеров

Но сначала рассмотрим различные архитектуры компьютеров. Наиболее известна классификация компьютерных архитектур Майкла Флинна (1972). В ее основу положено описание работы компьютера с потоками команд и данных. Выделяется четыре класса архитектур:

1. SISD — один поток команд и один поток данных. Это, как легко догадаться, обычные компьютеры, выполняющие в каждый момент времени только одну операцию над одним элементом данных, то есть уже известная архитектура фон Неймана.

2. SIMD — один поток команд и несколько потоков данных. Все процессоры находятся под управлением главного процессора, называемого контроллером, и нескольких процессоров обработки данных или процессорных элементов. Если в команде встречаются данные, контроллер рассылает команду на все процессорные элементы, которая и выполняется одновременно на всех или на нескольких процессорных элементах. Этот подход пригоден только для ограниченного круга задач, характеризующихся высокой степенью регулярности (упорядоченности). Большинство параллельных алгоритмов не могут эффективно выполняться на SIMD-компьютерах. Примером компьютеров данной архитектуры являются обычно машины с векторными процессорами. Вот реальные машины этого класса: MasPar MP, CM-2, DAP и другие.

3. MISD — несколько потоков команд и один поток данных. Машин этого не существует. С большими оговорками можно отнести сюда компьютеры с конвейерными процессорами. Впрочем, известна конструкция с *систолическим массивом* процессоров, где в каждом цикле работы каждый процессорный элемент получает данные от своих соседей, выполняет одну команду и передает результат соседям.

4. MIMD — несколько потоков команд и несколько потоков данных. Наиболее богатая и давно известная категория. Довольно давно появились компьютеры с несколькими независимыми процессорами, но поначалу на разных процессорах выполнялись разные и независимые программы. MIMD-компьютеры являются лидерами на рынке высокопроизводительных систем. Индивидуальные процессоры в этом случае работают под управлением своих собственных программ, чем достигается большая гибкость заданий, выполняемых процессорами в каждый данный момент. Машины MIMD типа могут эмулировать машины SIMD типа полностью или частично. В этот класс попадают симметричные параллельные вычислительные системы, рабочие станции с несколькими процессорами, кластеры рабочих станций и т.д. Примерами подобных машин являются IBM SP, Intel Paragon, Thinking Machines CM5, Cray T3D, Cray T3E, Meiko CS-2, HP SPP-1600, Parsytec CC и другие. Собственно кластер рабочих станций настолько дешевое решение, что каждый у есть, скажем дома, два компьютера может легко создать параллельную машину.

Не менее важна классификация параллельных машин по организации памяти. Память может быть доступна глобально всем процессорам или локально — у каждого процессора своя память. В первом случае говорят об *общей (разделяемой) памяти* (shared memory). Во втором случае мы имеем дело с *распределенной (локальной) памятью* (distributed memory), каждый процессор может адресоваться только к собственной памяти, а обмены между процессорами происходят путем передачи сообщений, содержащих ту или иную информацию. Для вычислений естественно предпочтительнее машины с разделяемой памятью, но они заметно дороже и, кроме того, их труднее изменять (например, добавив несколько процессоров), поэтому бóльшее распространение получили машины с распределенной памятью.

И, наконец, еще одной важной характеристикой параллельной машины является способ общения отдельных процессоров друг с другом. Задачей соединения процессоров является обеспечение наименьшего числа

пересылок (или времени пересылок), необходимых для переноса данных между любыми двумя процессорами системы. Машины с разделяемой памятью обычно используют динамическую связь между различными узлами (процессорами) и памятью. Связь реализуется с помощью переключателей, которые устанавливают связь между процессорами и памятью. Машины с распределенной памятью обычно используют статическую сеть. Топология связей определяет пространственные связи процессоров и является важным параметром в реализации параллельного алгоритма. Процессоры не связанный друг с другом непосредственно передают свои сообщения через промежуточный процессор. Кластеры персональных компьютеров или рабочих станций объединяются относительно низкоскоростной сетью.

3.7.2 Распараллеливание вычислений

Теперь от архитектур параллельных систем вернемся к параллельному программированию. Важно понять, что тот или иной подход к программированию параллельных вычислений, зависит в первую очередь от архитектуры используемой для решения задачи параллельной машины.

Существуют два основных подхода к распараллеливанию вычислений. Это параллелизм данных и параллелизм задач. В англоязычной литературе соответствующие термины — data parallel и message passing.

В основе обоих подходов лежит распределение вычислительной работы по доступным пользователю процессорам параллельного компьютера. При этом приходится решать разные проблемы. Прежде всего это достаточно *равномерная загрузка процессоров*, так как если основная вычислительная работа будет ложиться на один из процессоров, то вся работа будет мало отличаться от обычных последовательных вычислений. Другая проблема — *скорость обмена информацией между процессорами.* Если вычисления выполняются на высокопроизводительных процессорах, загрузка которых достаточно равномерная, но скорость обмена данными низкая, основная часть времени будет тратиться впустую на ожидание информации, необходимой для дальнейшей работы данного процессора.

Рассматриваемые подходы различаются методами решения этих двух основных проблем. Разберем более подробно параллелизм данных и параллелизм задач.

Параллелизм данных предполагает, что одна операция выполняется сразу над всеми элементами массива данных. Различные фрагменты такого массива обрабатываются на векторном процессоре или на разных процессорах параллельной машины. Распределением данных между процессорами занимается программа. Распараллеливание выполняется уже

на этапе компиляции, а роль программиста сводится лишь к управлению оптимизацией такого распараллеливания. В этом случае, как правило память разделяемая и используются специальные компиляторы, High Perfomance Fortran или C*, например.

Однако векторные машины и машины с разделяемой памятью не так доступны, как, скажем, кластеры персональных компьютеров. В случае кластеров персональных компьютеров или MIMD-машин с распределенной памятью. Программирование, основанное на параллелизме задач предполагает, что задача разбивается на несколько самостоятельных задач, и каждый процессор решает свою задачу. Чем больше число задач, которые допускают одновременное решение, тем бóльшую эффективность мы получаем, при этом все эти программы должны обмениваться результатами своей работы с помощью вызова неких стандартных процедур. Программист при этом ответственен за распределение данных между процессорами и подзадачами и обмен данных. Это более трудоемкий процесс, чем в предыдущем случае, как в отладке, так и в обеспечении равномерной загрузки процессоров и минимизации обменов между процессами, кроме того возможны тупиковые ситуации, когда посланные данные не доходят до процесса, которому они посланы. Вместе с тем эта более гибкая система может использоваться на параллельных компьютерах самой дешевой архитектуры — кластерах персональных компьютеров. Специализированные библиотеки для организации таких параллельных процессов (MPI, Message Passing Interface) или PVM (Parallel Virtual Machines) доступны (в том числе и в исходных кодах) и работают в различных операционных средах (Linux, MS Windows и т.д.).

Кроме доступности и дешевизны, программирование, основанное на параллелизме задач, еще и достаточно близкó к привычной архитектуре (фон Неймана), поскольку задача разбивается на ряд *последовательных* подзадач.

3.7.3 Закон Амдала

Попробуем оценить выигрыш (ускорение), который мы можем получить, используя для решения задачи n процессоров. Определим выигрыш параллельного алгритма S_p:

$$S_p = \frac{T_1}{T_p},$$

здесь T_1 — время работы параллельного алгоритма на одном процессоре, T_p — время работы параллельного алгоритма на p процессорах $1 \leq S_p \leq p$.

Закон Амдала. Пусть α — доля вычислений, которые выполняются в параллельном режиме, $1-\alpha$ — доля вычислений в последовательном режиме (эти режимы не совмещаются). Тогда последовательные вычисления занимают время $(1-\alpha)T_1$, а параллельные — $\alpha T_1/p$. Оба режима занимают время $T_p = T_1(1-\alpha+\alpha/p)$. Формула для выигрыша (ускорения) имеет вид:

$$S_p = \frac{p}{p-\alpha(p-1)}$$

и

$$\lim_{p\to\infty} S_p = \frac{1}{1-\alpha}$$

Зависимость ускорения S_p от числа процессоров и от доли параллельных вычислений представлена на рисунке.

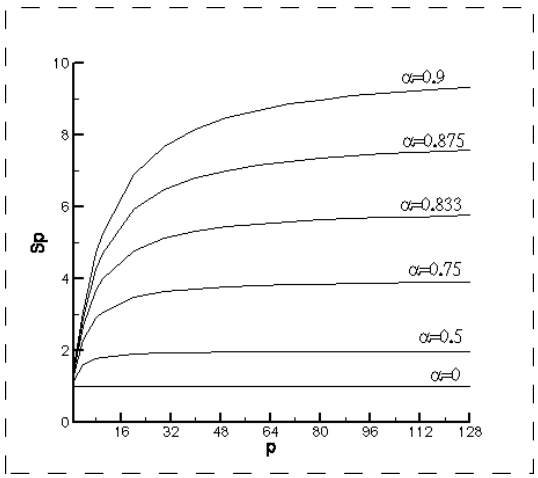


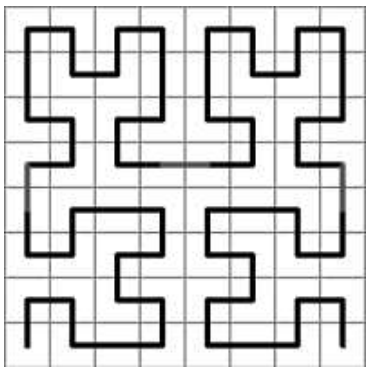
Рис. 5: Закон Амдала

Из рисунка видно, что для программ с небольшой степенью параллелизма использование большого числа процессоров не дает сколько-нибудь заметного выигрыша в быстродействии. Начиная с некоторого места, увеличение числа процессоров дает только небольшой выигрыш в производительности. Отметим, что на практике приходится принимать во внимание время обмена данными и это еще ухудшает ситуацию, может даже наблюдаться снижение производительности при увеличении числа процессоров.

Разработка программ для параллельных машин — непростое (по крайней мере непривычное) дело. Если архитектура используемой Вами машины предполагает параллелизм данных, большую часть работы можно возложить на транслятор. “Векторные” машины в этом случае

хорошо справятся с операциями, требующими одних и тех же действий над массивами данных, например, сложение, перемножение, ... матриц. Задачи, решающиеся методами конечных элементов, в которых для вычисления значений некоторых параметров требуется только знание этих параметров в соседних точках, хорошо укладываются в эту схему. Однако, во-первых, не все задачи позволяют эффективно использовать векторные машины, и, во-вторых, как мы знаем, по причине своей дешевизны наиболее распространенными являются параллельные машины, состоящие из нескольких самостоятельных компьютеров (кластеры PC, например). В этом случае мы должны использовать другой подход к распараллеливанию задачи — параллелизм процессов. Разбиение задачи на ряд параллельно выполняющихся процессов целиком ложится на разработчика программы, здесь нет общих решений и эту часть нельзя возложить на компилятор.

Вспомним задачу N тел и рассмотренный нами алгоритм Treecode. Допустим $N = 10^6$. Если бы мы могли использовать N машин для решения нашей задачи, мы могли бы каждой поручить вычисление на каждом шаге сил, действующих на каждое из N тел. Вряд ли, однако, нам это удастся. Во-первых, 10^6 тел это далеко не предел, а, во-вторых, алгоритм TreeCode хорош тем, что позволяет нам не вычислять силу для каждой пары, заменяя с небольшой потерей точности множество далеких тел одним. Как же разбить задачу на несколько независимых процессов, на относительно независимые "параллельные" части? Как решить, что силы, действующие на данное тело, нужно вычислять в одном процессе, а силы, действующие на другое — в другом. При этом процессы по возможности должны обрабатывать близкие тела (минимизируя таким образом обмен сообщениями между процессами), а алгоритм такого разбиения должен быть и концептуально, и технически прозрачен и обеспечивать сбалансированную работу p процессоров. Идея состоит в том, чтобы разбить все тела на p групп с весами, зависящими от требуемого времени вычислений. Сделать это в пространственном случае невозможно (вспомните Парето, где было два параметра), поэтому все сводится к разбиению на p равных в смысле затрат на вычисления групп одномерного(!) пространства, в котором тела упорядочены, и таким образом разбиение элементарно. Как строить требуемую кривую, хорошо известно. Такие кривые (которым принадлежат, например, все точки квадрата, или куба в пространственном случае) известны уже более ста лет и носят название кривых Пеано-Гильберта.



На рисунке приведена кривая Пеано-Гильберта, которая иллюстрирует метод разбиения метод распараллеливания алгоритма TreeCode в двумерном случае на 16 процессоров. Трехмерный случай принципиально ничем не отличается. Организовав разбиение на p процессов, мы, конечно, еще должны обеспечить взаимодействие различных процессов с помощью посылаемых процессами друг другу сообщений, с информацией о состоянии процесса и полученными данными.

3.8 Специализированные языки. Языки управления базами данных

Мы рассмотрели различные парадигмы программирования. Можно классифицировать известные языки и по другим признакам. (В одном из докладов военно-морскому флоту было приведено более 2570 различных свойств языков программирования.) Например, по принадлежности к тому или иному семейству языков:

- *C-подобные*,
- *Pascal-подобные*,
- *Prolog-подобные*,
- семейства универсальных языков,
- семейство уникальных языков (Forth, Postscript) и т.д.

По степени абстракции от аппаратуры:

- *языки низкого уровня* (ассемблер);
- *языки высокого уровня* (сложные структуры, доступ к памяти осуществляется только через операции);
- *языки сверхвысокого уровня* (команды выполняются на полностью абстрактной машине, доступ к памяти скрыт).

По ориентации на предметные области, специализированные языки:

- *языки форматирования текстов* (TeX, LaTex),

- *языки разметки* (SGML, XML),
- *языки скриптов* (Perl, Tcl/Tk, bash, csh),
- *языки описания аппаратуры* (VHDL — Very high speed integrated circuit Hardware Description Language),
- *языки создания графики* (Postscript),
- *языки описания виртуальной реальности* (VRML),
- *языки конфигурирования* (autoconf),
- *промежуточные языки* (расширения, PSP).

К языкам, ориентированным на предметные области, можно отнести и языки работы с базами данных. Несколько слов о БД. Террабайты информации наблюдательных данных, прежде всего с космических миссий, нельзя эффективно использовать без баз данных.

БД — совокупность, связанных между собой данных. Можно выделить такие блоки:

- *поля* — поле содержит один элемент данных;
- *запись* — несколько связанных полей представляют собой запись;
- *таблицы* — наиболее общий элемент базы данных — объединение идентичных по смыслу записей.

Если рассматривать реляционные БД, то каждую таблицу можно трактовать как некоторое *отношение*, запись (строка таблицы) называют еще *кортежем*, а поле — *аттрибутом отношения* $k \in K$, K — множество всех атрибутов отношения — называют *схемой отношений*. Если определить *домен* как множество возможных значений атрибутов D_k, то отношение $D(K)$ — это подмножество прямого произведения доменов $X_{k \in K} D_k$.

Такое, может быть чересчур формальное, определение отношения позволяет свести все требуемые для работы с базой данных операции к операциям реляционной алгебры.

Как правило БД содержит несколько (много) связанных таблиц, то есть для описания какой-либо совокупности данных нужно разработать совокупность схем отношений, которые используются для представления информации. Эта совоукупность называется *схемой БД*. Схема — это скелет базы, а саму базу данных составляет набор описанных в схеме таблиц — значение отношений. База данных Солнечной системы могла бы

содержать два отношения: таблицу, описывающую планеты Солнечной системы

- N: номер планеты $k \in [0, 1, \ldots, 9]$;
- $Name$: название планеты;
- M: масса;
- $Elements$: элементы орбиты;
- NoS: число спутников;
- N_Sat: номер спутника (ссылка на другое отношение);
- ....

Пример кортежа:

```
3, Земля, 332958, {1, 0.167, ...}, 1, 1, ...
```

Здесь масса приведена в обратных массах Солнца ($M_{♁} = 1/332958 M_{\odot}$).

Второе отношение — таблица спутников планет

- N: номер спутника;
- $Name$: название спутника;
- M: масса;
- $Elements$: элементы орбиты;
- N_planet: родительская планета;
- ....

Пример кортежа этого отношения:

```
1, Луна, 81.3, {1, 384402, 0.055, ...}, 3, ...
```

Между отношениями могут существовать функциональные зависимости. Зависимость “один к одному” связывает единичный кортеж одного отношения с кортежем другого. Зависимость “один ко многим” связывает единичный кортеж (планеты) одного отношения с несколькими кортежами другого (спутники).

Для поддержки целостности данных в каждом кортеже обязательно должен быть *ключ*, поле, значение которого уникально для каждой записи. В нашем случае это номер планеты для первого отношения и номер спутника — для второго.

Представление данных в БД не зависит от их физической организации, а только от их структуры и отношений. В *реляционных* базах данных это обеспечивается за счет использования математической теории отношений. Собственно для получения информации из базы необходимо выполнить преобразования таблиц, причем достаточно использовать небольшое число операций (реляционной алгебры):

- *объединение* (нескольких наборов кортежей в один в теоретико-множественном смысле);
- *пересечение* (нескольких наборов кортежей в один),
- *вычитание* (эти три операции — теоретико-множественные и операнды должны иметь одну и ту же схему отношения);
- *произведение* (из двух таблиц составляется таблица, в которой каждый кортеж одной сцепляется с каждым кортежем другой);
- *выборка* (получение таблицы, в которой присутствуют только кортежи, удовлетворяющие заданному условию);
- *проекция* (удаление из таблицы некоторых атрибутов);
- *расширение* (добавление в таблицу некоторых атрибутов);
- *соединение* (в произведении оставляются только кортежи, в которых одинаковы значения некоторых атрибутов, например, список планет со спутниками);
- *деление.*

Эти операции применяются для формирования новых таблиц, которые представляют интересующий нас результат. Существует специальный язык запросов SQL (Structured Query Language) служит для работы с реляционными базами данных, то есть позволяет создавать требуемую таблицу из имеющихся в БД, используя эти операции.

В теории БД определен ряд свойств, так называемые *нормальные формы*, для отношений, имеющих зависимости. Эти свойства, если они соблюдаются, позволяют избежать многих проблем, возникающих при создании базы данных, например, проблему избыточности данных.

БД должна обеспечивать целостность данных. В процессе обновления данных целостность может нарушаться. Во избежании таких событий вводится понятие *транзакции* — последовательность операций, которые должны быть или все выполнены, или все не выполнены. Например, если мы хотим изменить нашу базу данных, чтобы хранить массы планет не в обратных массах Солнца, а в граммах, мы не должны встретиться с ситуацией, когда часть записей уже заменена и массы заданы в граммах, а часть еще имеет массы, заданные в обратных к массе Солнца значениях. Средства управления транзакциями, разумеется, тоже входят в SQL.

3.9 Программное обеспечение

Несколько слов о программном обеспечении. Прежде всего это, конечно, операционная система.

В настоящее время, если не принимать во внимание различные модификации, можно ограничиться знанием только двух систем — это Windows и UNIX. Да, разумеется, можно вспомнить еще несколько, из которых я бы выделил Plan9, но две приведенные системы — вне конкуренции. У каждой есть свои достоинства и недостатки и нет смысла противопоставлять их, можно просто сравнить:

	Windows	Linux
архитектура	PC	PC, Alpha, Sparc, HP, ...
приложения	клиентские	серверные
интерфейс	графический	командная строка/графический
пользователи	все чайники	профессионалы разработчики
создатели	много	Кен Томпсон, Денис Ритчи

Они решают разные задачи и у них разные цели. Windows прежде всего коммерческая система. (Байка Джона?) С Linux вы уже познакомились, и знаете, почему выбрана именно эта система. Повторю еще раз.

Самое важное, что UNIX — это открытая система. Вот определение открытой системы (комитет IEEE POSIX, вспомним стандарт IEEE-754):

Открытая система — это система, реализующая открытые спецификации на интерфейсы, службы и форматы данных, достаточные для того, чтобы обеспечить:

- возможность переноса (мобильность) прикладных систем, разработанных должным образом, с минимальными изменениями на широкий диапазон систем;
- совместную работу (интероперабельность) с другими прикладными системами на локальных и удаленных платформах;

- взаимодействие с пользователями в стиле, облегчающем последним переход от системы к системе (мобильность пользователей).

"Открытые спецификации" в данном определении понимается как "общедоступная спецификация, которая поддерживается открытым, гласным согласительным процессом, направленным на постоянную адаптацию новой технологии, и соответствует стандартам".

Согласно этому определению открытая спецификация не зависит от конкретной технологии (от конкретных технических или программных средств или продуктов отдельных производителей). Спецификация одинаково доступна любой заинтересованной стороне.

А теперь вспомним, что любой научный результат должен быть

а) доступен всему научному сообществу;

б) воспроизводим.

Очевидно, что, если все программное обеспечение, в нашем случае астрономическое, будет реализовано в рамках концепции открытых систем, то оно будет и доступно, и переносимо (воспроизводимо).

Unix, работающий практически на всех существующих архитектурах, является открытой системой. Именно поэтому он так распространен в научных исследованиях.

Собственно и сама парадигма Интернета зародилась в рамках научного учреждения — в Европейском центре ядерных исследований (CERN) — сотрудники, участвующие в программе CERN'а, но живущие в разных концах света, нуждались в инструменте, позволяющем интуитивно понятным способом обмениваться данными и информацией по сети. И протокол HTTP, и схема адресации URL были разработаны Тимом Бернерс-Ли в CERN'е и естественно (для научного сообщества) к результатам этой работы был предоставлен доступ всему Интернет сообществу. В том же CERN'е Андерс Берглунд, отвечающий за организацию текстовой обработки, ввел в употреблению SGML, принятый в 1986 году в качестве международного стандарта. Не удивительно, что Тим Бернерс-Ли разработал HTML под большим влиянием и буквы и духа SGML. (Формально DTD для HTML было разработано лишь через несколько лет.)

Конечно, если вы пишете небольшую программу (порядка 1000 операторов), то достаточно использовать языковые средства, имеющие общепринятые стандарты. Например, Фортран или C. Правда, нужно еще убедиться насколько используемый компилятор придерживается принятых (открытых) стандартов. В UNIX'е практически всегда можно быть

уверенным в том, что стандарты доступны и соблюдаются. Это становится особенно важным, если разрабатывается большой программный комплекс, над которым одновременно работает несколько человек, да и находятся они далеко друг от друга. В астрономии проблема стоит еще более остро. Объем данных наблюдений столь велик (и продолжает расти), что без стандартов, без неукоснительно следования этим стандартам, было бы просто невозможно работать с имеющимися данными.

Астрономам всегда требовалось проводить большой объем вычислений (или обработки данных). Будь то вычисление эфемерид, построение высокоточных теорий движения планет и спутников, расчет эволюции звезд или обработка террабайт информации, поступающей с многочисленных космических проектов. Именно поэтому они всегда были самыми квалифицированными и пользователями, и разработчиками программного обеспечения. Начиная с Чарльза Беббиджа (середина позапрошлого века), создавшего первую, хотя и не электронную, Аналитическую машину, которая имела многие черты современного компьютера (например, выполняемая программа). Впервые Беббидж доложил о своей разностной машине членам Королевского Астрономического общества — для создания астрономических таблиц требовалось множество вычислений. Кстати, первым программистом (не из притчи) можно считать Аду Лавлайс, которая заинтересовавшись проектом Аналитической машины Беббиджа, разработала первые программы.

В 50-х годах, когда появились первые языки программирования (Фортран, Алгол), был разработан метод описания их синтаксиса, известный как форма Бэкуса-Наура. Питер Наур известен и как астрометрист.

```
<цифра> ::= 0|1|2|3|4|5|6|7|8|9
<целое число> ::= <цифра>|<цифра><целое число>
<число со знаком> ::= <число>|+<число>|-<число>
```

Потребности управления астрономической аппаратурой привели к созданию языка Forth, который позволял писать программы, которые занимали не больше памяти, чем такие же программы, написанные непосредственно на машинном языке. Создан этот язык был в Астрономическом учреждении Чарльзом Муром и стал в 80-е годы чрезвычайно популярным. И сейчас широко используется язык описания страниц PostScript, имеющий много общего с Фортом.

Уже довольно давно (70-е годы) астрономы осознали необходимость стандартов для переносимости данных, разработав специальный формат (FITS). Еще до эры Интернета астрономы использовали удаленные наблюдения, а в эпоху Интернета самые большие, самые используемые научные Базы Данных — астрономические. Сюда включаются и данные

многочисленных миссий, и электронные публикации, и различные астрономические приложения.

Вернемся к программному обеспечению.

Все, перечисленное выше, эффективно “работает” в “открытых системах”, следующих общепринятым стандартам.

Кроме всего прочего, парадигма открытых систем позволяет решать задачи, используя “ортогональные” средства, то есть средства, не зависящие друг от друга. Не важно, пользуетесь ли вы для вычислений языком Фортран или Питон (что хуже в смысле эффективности), вы можете легко визуализировать ваши результаты с помощью любой доступной графической системы. Альтернативное решение, например, визуализация с помощью вызовов фортрановских подпрограмм из некоторой библиотеки потребовало бы изменения вашей программы и в случае, если у вас изменился алгоритм вычислений, и в случае, если вам потребовалось изменить, скажем, цвет некоторой кривой.

К программному обеспечению относятся трансляторы (C, Fortran), интерпретаторы (Perl, Python), командные языки (bash, csh), инструментальные средства (утилиты, сжатие, архивация и т.д.), прикладные программы (обработку текстов; проведение вычислений; организация информации; управление вводом-выводом). Обычно различные функции настолько тесно переплетаются друг с другом, что трудно сказать, где кончается одна и начинается другая. Хотя большинство функций в той или иной степени используется в любой программе, одна из них всегда преобладает.

Среди прикладных программ, по преобладанию некоторых функций, выделяют:

- текстовые редакторы (Emacs, VI),
- средства разработки (в том числе отладчики),
- графические редакторы (GIMP),
- средства работы с документами ((La)TeX, OpenOffice),
- электронные таблицы (OpenOffice Calc),
- системы управления базами данных (MySQL, Postgress),
- музыкальные редакторы (MusiXTeX),
- мультимедийные средства,
- интегрированные пакеты прикладных программ.

Конец

Содержание