тенсивно обрабатывают большие объемы данных и для повышения эффективности, читай быстродействия, могут использовать более сложные структуры и для хранения данных на внешних носителях, используя, например, индексные указатели, уточняющие положение нужного фрагмента данных.

# 5 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.*

## 5.1 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.

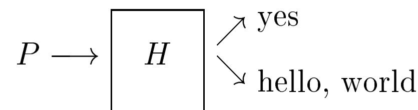
Рассмотрим, например, такую, весьма простую задачу: является ли текст `hello, world` первым, что печатает C-программа (можно Pascal-программа, или даже Фортран-программа?). Разумеется, проще всего было бы запустить ее и посмотреть на результат. Однако может оказаться, что ждать придется долго, очень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

ждемся 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).

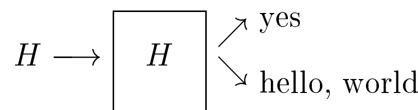
Было бы замечательно, если бы мы написали программу, которая проверяет любую программу $P$ со входом $I$ и определяет, печатается ли при ее выполнении `hello, world`. Покажем, что такой программы не может быть.

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.

## 5.2 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;
$\Sigma$ — конечное множество входных символов;
$\Gamma$ — множество ленточных символов, $\Sigma \subset \Gamma$;
$\delta$ — 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.

# 6 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 $T[s + 1 \dots s + m] = P[1 \dots m]$. 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

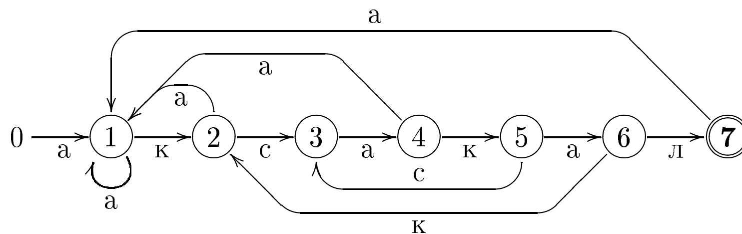
- $Q$ — конечное множество состояний;
- $q_0 \in Q$ — начальное состояние;

- $F \subset Q$ — конечное множество заключительных, или *допускающих* состояний;
- $\Sigma$ — конечное множество (алфавит) входных символов;
- $\delta$ — 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 $\delta$ 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); $\Sigma$ — 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, $\sigma$ определена на всем $\Sigma^*$ ($Sigma^* = \Sigma \cup \{\}$. 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 $\delta$ 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.

## 6.1 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 Паре́то.*

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. Тогда 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 $2^\circ$ 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, $\phi = 50°$. Возможные значения критериев расположены выше и правее кривой и на кривой.

## 6.2 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. Далее каждую ячейку делим на 8 равных кубов в трехмерном случае или на 4 квадрата в двумерном. Если оказывается, что в какой-то из полученных ячеек частиц нет, такая ячейка в дерево не включается, в противном случае процесс рекурсивно продолжается дальше, пока в ячейке не окажется ровно одна частица. В результате получается дерево.

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

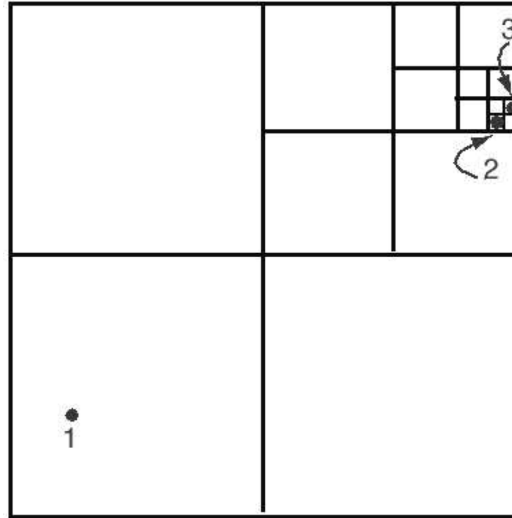



Рис. 1: Построение дерева Барнса-Хата для системы из 3 частиц (двумерный случай).

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 $p$ (достаточно 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

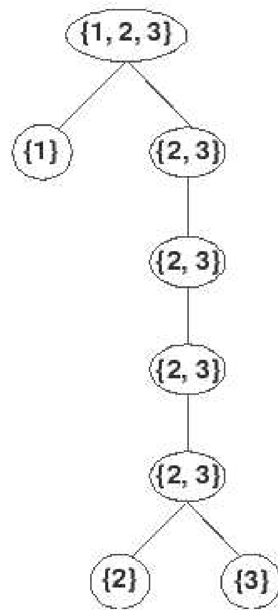



Рис. 2: Дерева Барнса-Хата
для приведенной системы.

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 $O(N)$).

## 6.3 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).

# 7 Языки программирования и программное обеспечение

## 7.1 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):

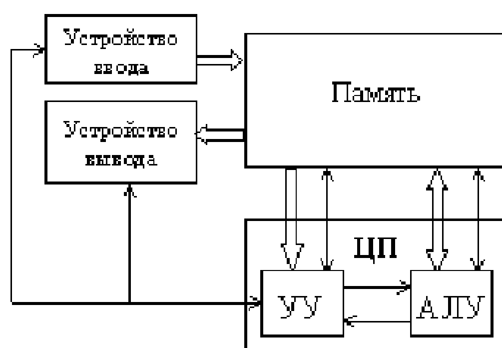



Рис. 3: Схема машины фон Неймана

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 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.

## 7.2 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.

## 7.3 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 $\alpha$). Ну возможно, потом еще обработать получившийся список и получить, если со-

ответствующее 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.*

## 7.4 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.

## 7.5 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.

## 7.6 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.

## 7.7 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 име-

ет 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.

### 7.7.1 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 памя-

тью. 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.

### 7.7.2 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.

### 7.7.3 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 $\alpha$ — 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.

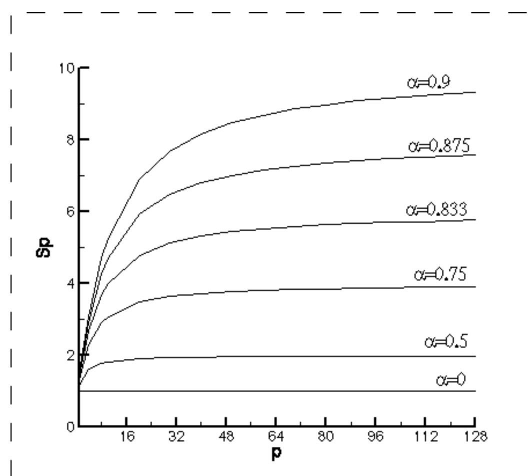


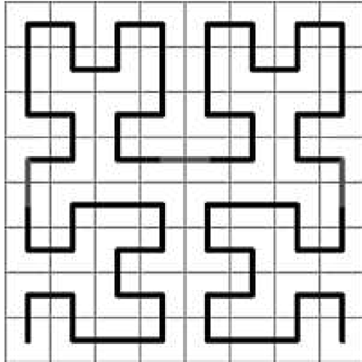

Рис. 5: Закон Амдала

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.

## 7.8 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.

## 7.9 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