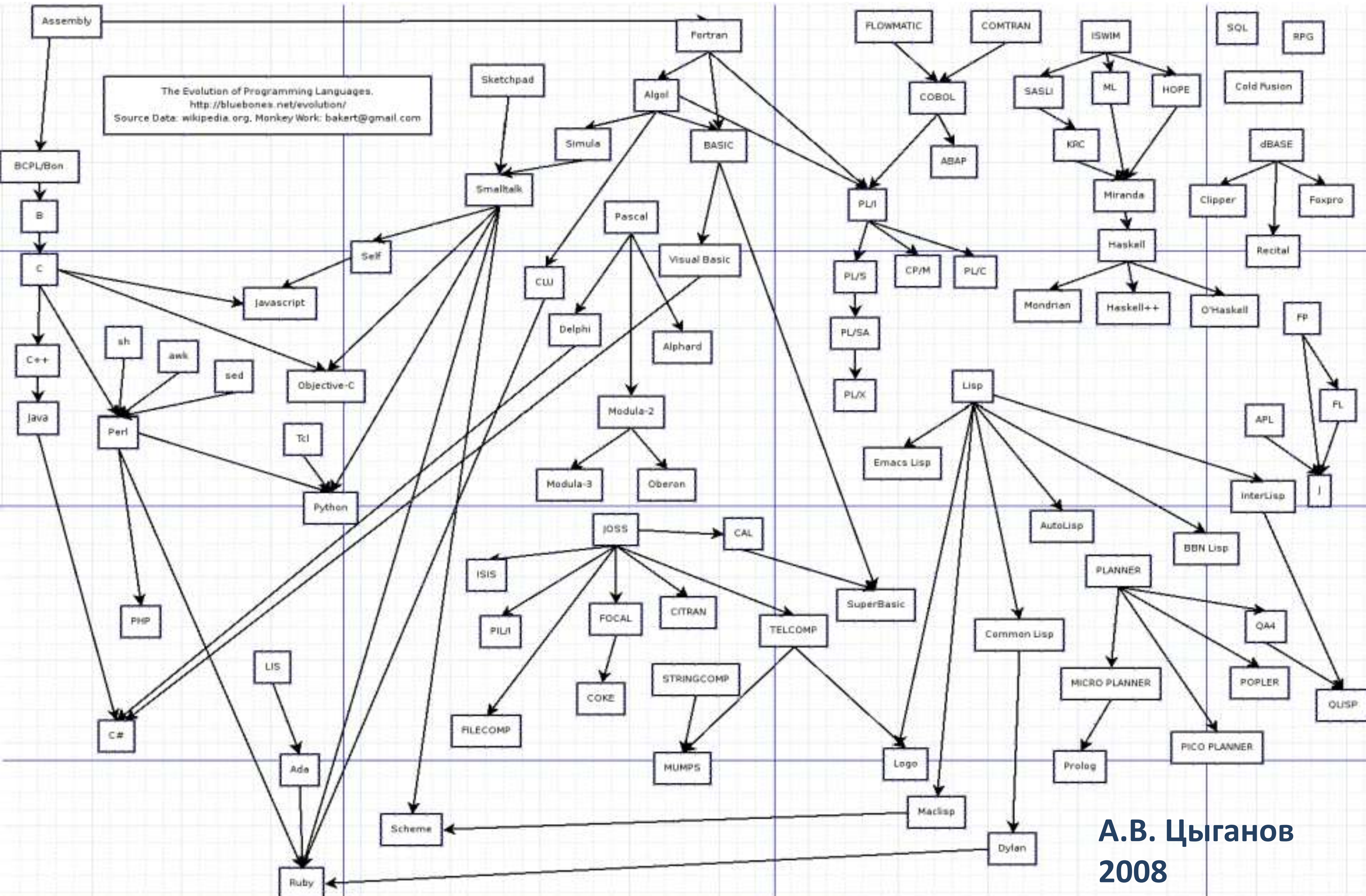
Введение в теорию алгоритмов (2)



А.В. Цыганов
2008

Что объединяет все эти языки?

Алгоритмический язык — формальный язык, используемый для записи, реализации и изучения **алгоритмов**.

Большинство языков программирования являются алгоритмическими языками, т.е. ***формализованными языками*** с чётко описанным синтаксисом и точно заданной ***семантикой*** его грамматических категорий.

В логико-лингвистическом и гносеологическом аспекте алгоритмические языки представляют собой одну из моделей **императива** (повелительного наклонения), и потому выступают, с одной стороны, как средство фиксации операционного знания, а с другой – как инструмент коммуникации.

Дальнейшим развитием идеи алгоритмического языка явились языки программирования более общего, не обязательно алгоритмического характера, которые в конечном счёте тоже нацелены на получение машинных программ, но во многих случаях их тексты допускают **определённую свободу в выполнении** и, как правило, дают лишь материал для **синтеза алгоритмов**, а не сами эти **алгоритмы**.

- Вычисление факториала:

```
fact(1)=1.
fact(N)=N*fact(N-1).
```

Логическое программирование (Mercury)

```
fact(1,1).
fact(N,F) :- N1 is N-1,
             fact(N1,F1),
             F is F1*N.
```

Логическое программирование (Prolog)

Императивное программирование (Pascal)

```
function fact(x:integer):integer;
var i, r : integer;
begin
  r:=1;
  for i:=1 to x do r:=r*i;
  fact:=r
end;
```

```
let rec fact = function
    1 -> 1
  | x -> x*fact(x-1);;
```

Функциональное программирование (F#)

В чем же отличия?

Почему надо «знать» машину Тьюринга?

Потому, что это - начала математики, в нем вводится понятие алгоритм

Например, складывая в столбик, мы фактически реализуем машину Тьюринга:

$$\begin{array}{r} 587 \\ +\ 204 \\ \hline 791 \end{array}$$

7 + 4 – мы реально не производим вычисления, не прибавляем как школьники первого класса к ***7*** палочкам еще ***4*** палочки. Нет, мы знаем что ***7+4 = 1*** и ***1*** «**в памяти**». Т.е. ее надо перенести влево как это и делается в машине Тьюринга. Точно также мы складывая ***8*** и ***0*** получаем 8, но так как мы находимся в особом состоянии, которое называется «**перенос единицы**» мы знаем, что в нижней строке надо записать 9.

Итак - вместо реальных вычислений у нас в голове производится формальная замена одних символов другими – т.е. выполняется программа или алгоритм.

К сожалению, у многих программистов теоретическое определение алгоритма заменяется на своё, более близкое: алгоритм – эта программа, которую пишу **Я** (это приводит к неприятностям, например на экзамене)

Со школьной скамьи мы знаем, что **7 x 8 = 56**.

Фактически здесь имеет место употребления **нормальных алгоритмов Маркова**.

Цепочка символов **7x8** заменяется на цепочку **56**. (**транслятор**)

Никаких действий не происходит.

Машина Тьюринга и алгоритмы Маркова играют в понимании математики, в умении квалифицированно решать задачу такую же фундаментальную роль, как и постулаты Евклида.

Естественно, что рабочий или прораб на стройке может не знать аксиом Евклида, они ему не нужны.

Но **архитектор** всегда помнит о них (не использует в работе, но знает, чувствует их), т.к. должен учитывать геометрию (по крайней мере, начертательную) при проектировании и строительстве.

Точно также и абитуриенты гуманитарных вузов могут не помнить постулаты Евклида, но я уверен, что, не зная и не понимая их сущности, невозможно поступить ни на мехмат МГУ, ни на физтех, ни в любой другой приличный университет или институт

Определение алгоритма

Вычислимость.

Множество A $(A \subseteq U)$ ***вычислимо***, если существует некоторая процедура $П$, порождающая все элементы $a \in A$ и только их.

Пример

Задано множество действительных чисел – D и уравнение

$$y = ax + b,$$

определённое на четверках чисел $D^4 = U$.

Обозначим множество четверок, которое удовлетворяет уравнению, через A.

Множество A вычислимо, т.к. существует процедура $П(a, b, x) \rightarrow y$, которая по любой тройке **< a,b,x>** вычисляет **y**.

Разрешимость.

Множество A ***разрешимо*** на M ($A \subseteq M$), если существует процедура (алгоритм), которая для любого $m \in M$ определяет, принадлежит ли m к A ($m \in A$) или нет ($m \notin A$).

Проблема *разрешимости* сводится к *вычислимости* соответствующего предиката

$$P(m) = \begin{cases} 1, \text{ истина}, & m \in A \\ 0, \text{ ложь}, & m \notin A. \end{cases}$$

Пример.

Пусть A есть множество решений уравнения

$$y = ax + b.$$

Существует тривиальный алгоритм Π проверки для каждой четвёрки **<a,b,x,y>** в виде подстановки этих чисел в уравнение и выполнения указанных в уравнении действий, который ***вычисляет*** предикат и таким образом решает проблему ***разрешимости*** для заданного уравнения.

Пример. **Теорема Ферма.**

Для уравнения

$$x^n + y^n = z^n,$$

где $x, y, z, n \in N$, можно ли предложить алгоритм Π, порождающий все решения уравнения при заданном n?

Разрешимость множества решений уравнения решается просто, т.к. имеется тривиальный алгоритм вычисляющий предикат. Например,

$$\Pi(x = 1; y = 1; z = 2; n = 2) = (1^2 + 1^2 = 2)$$ правда

$$\Pi(x = 2; y = 2; z = 5; n = 2) = (2^2 + 2^2 \neq 5)$$ ложь

Итак, множество решений уравнения

$$x^n + y^n = z^n$$

разрешимо, но ***не вычислимо*** (пока не известен алгоритм или он не существует).

Пример.

Имеет ли уравнение $x^3 + y^3 + z^3 = 29$ целое решение?

Легко проверить (*вычислить предикат*), что есть решение (***3,1,1***).

Уравнение $x^3 + y^3 + z^3 = 30$ также имеет решение, найденное в 1990 году

(***-283059965, -2218888517, 2220422932***).

Для уравнения $x^3 + y^3 + z^3 = 33$ решений не известно.

Эти задачи связаны с 10 проблемой Гильберта и в 1970 году Ю. Матиясевич доказал, что ***не существует алгоритма*** нахождения решений таких задач!

Итак, взаимосвязь между вычислимостью и разрешимостью:

1) если ***А вычислимо*** в ***М***, то ***А разрешимо*** в ***М***;

2) если ***А разрешимо*** в ***М***, то из этого не следует его ***вычислимость***.

Проблемы разрешимости и вычислимости множеств требуют более точного определения понятия **алгоритма.**

Кажется, что **никакой нормальный программист** такие объяснения все равно читать не будет, поскольку и так ясно, что такое алгоритм. Но все же учится необходимо, чтобы не принять за алгоритм то, что им не является с одной стороны, а с другой эта теория необходима для эффективного использования языков **РЕФАЛ, Snowball, Lisp, Prolog, Python…………**

Понятие ***алгоритма*** в обиход математики ввел немецкий математик Шрёдер (1912–14), назвав «достаточно простую» механическую процедуру по имени арабского математика Ал–Хорезми (IX в.) алгоритмом.

Точного понятия алгоритма нет и никогда не будет сделано по причине очень «туманного» содержательного объекта, который необходимо формально определить.

Алгоритм - некоторый точно определенный инструмент, при помощи которого можно конструировать любые интуитивно понятные порождающие (**вычисляющие**) и распознающие (**разрешающие**) процедуры.

Алгори́тм — точный **набор инструкций**, описывающих **порядок действий исполнителя** для достижения результата решения задачи за конечное время.

Алгоритм — это понятные и точные предписания исполнителю совершить конечное число шагов, направленных на решение поставленной задачи.

Алгоритм — это последовательность действий, либо приводящая к решению задачи, либо поясняющая почему это решение получить нельзя.

«**Алгоритм** — это конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечность, определённость, ввод, вывод, эффективность». (**Д. Кнут**)

«**Алгоритм** — это всякая система вычислений, выполняемых по строго определённым правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи». (**А. Колмогоров**)

«**Алгоритм** — это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату». (**А. Марков**)

«**Алгоритм** — точное предписание о выполнении в определённом порядке некоторой системы операций, ведущих к решению всех задач данного типа». (**Философский словарь / Под ред. Розенталя**)

Алгоритм, как точно или явно определённый инструмент, для конструирования реальных процедур должен обладать следующими свойствами.

1. Конструктивность.

Любые *А* и *М* (интуитивно понятные и содержательные множества) должны быть конструктивно описаны в виде **структур или типов данных**.

Любые интуитивно понятные функции (предикаты) должны быть **конструктивно** (формально) описаны в терминах определения инструмента.

2. Детерминированность.

Вообще говоря следует из 1), т.к. **интуитивно** алгоритм предполагает реализацию функций и только их. (входным данным отвечает только один результат)

3. Результативность.

Определение и распознавание начального (правильные входные данные) и конечного события (правильные выходные данные), связанного с правильным или неправильным результатом работы алгоритма.

Дискретность — алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.

Завершаемость (конечность) — при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов. С другой стороны, вероятностный алгоритм может и никогда не выдать результат, но вероятность этого равна 0.

Понятность — алгоритм для исполнителя должен включать только те команды, которые ему (исполнителю) доступны, которые входят в его систему команд.

Массовость — алгоритм должен быть применим к разным наборам исходных данных.

Типы алгоритмов. История создания.

За 30–40 годы XX столетия интенсивного поиска универсального уточнения алгоритма было предложено примерно **20** формальных конструкций алгоритмов, которые условно можно разбить на три типа.

1. **Алгоритмические машины (АМ).**
2. **Функции вычислимые алгоритмом**
3. **Исчисления.**

a) Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ. — М.: «Вильямс», 2006. с. 1296.

b) Дональд Кнут Искусство программирования, том 1. Основные алгоритмы — 3-е изд. — М.: «Вильямс», 2006. — С. 720. — ISBN 0-201-89683-4

c) Колмогоров А. Н. Теория информации и теория алгоритмов. — М.: Наука, 1987. — 304 с.

d) Марков А. А., Нагорный Н. М. Теория алгорифмов, изд. 2. — М.: ФАЗИС, 1996.

Алгоритмические машины

Все они имеют один процессор, выполняющий небольшой набор очень примитивных действий, простую структуру данных (структуру памяти) в виде бесконечной ленты, простую логику (правила) управления процессором.

В этом случае ***алгоритм*** представляется набором правил команд процессора, последовательность выполнения которых управляет состоянием памяти - т.е. **программой.**

Основные АМ, которые повлияли на создание реальных вычислительных машин, реальных языков программирования и концепции организации вычислительных процессов:

I. Машина **Тьюринга** - 1937 г.

II. Машина **Поста** - 1937 г.

III. Нормальный алгоритм (алгорифм) **Маркова** (НАМ) - 1953 г.

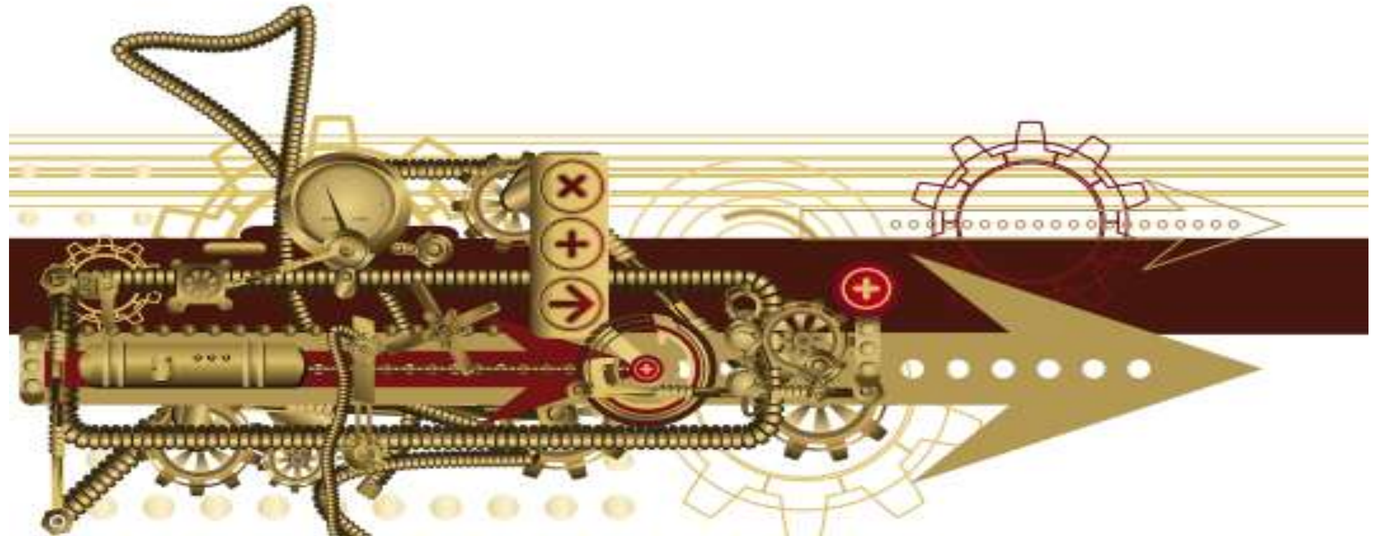
Принципы, положенные в основу функционирования алгоритмических машин.

1) **Однопроцессорность** и поэтому последовательная работа процессора только над одним потоком команд.

2) **Управление** процессора происходит под действием внешних либо внутренних событий.

3) **Алгоритм** понимается как последовательность переходов из состояния в состояние под действием команд или последовательностей команд.

4) **Управляющие** внутренние события воспринимаются как изменения в памяти процессора.



2. Функции вычислимые алгоритмом.

В этом случае сам ***алгоритм*** не определяется формально, а существует как бы в виде «всем понятной механической процедуры».

Основное внимание уделяется механизму конструирования функций из базового набора элементарных функций, определенных на некотором фиксированном множестве.

Любая функция, вычислимая на интуитивном (содержательном) уровне, должна быть сконструирована из **базовых функций.**

Пример:

Рекурсивные функции на множестве натуральных чисел были предложены Клини в 1938 г. Конструктивные механизмы рекурсивных функций очень просты, их применение в процессе построения «функции от функции» позволяет явно выстраивать структуру функции в отличие от АМ, где функция определяется процедурно, через последовательность действий.

Операция, называемая операцией **рекурсии**, или **примитивной рекурсии**, применяется к ***k***-местной функции ***f*** и ***(k+2)***-местной функции ***g***.
(“местность”=“арность”)

Ее результатом будет ***(k+1)-***местная функция ***h***, определяемая так:

$h(x_1,\ldots,x_k,0)=f(x_1,\ldots,x_k)$

$h(x_1,\ldots,x_k,y+1)=g(x_1,\ldots,x_k, y, h(x_1,\ldots,x_k,y))$

В последовательности $h(x_1,\ldots,x_k,0)$, $h(x_1,\ldots,x_k,1)$,... каждое значение определяется через предыдущее, поэтому если какое-то из значений не определено, то не определены и все последующие.

Для единообразия будем считать, что нуль-местные функции (функции без аргументов) суть константы; это позволяет рекурсивно определять функции одной переменной.

Базисные функции:

константа ***0*** - функция обнуления,

операции прибавления единицы $s : x \longrightarrow x+1$

семейства функций проекции: это семейство для каждого ***k*** содержит ***k*** штук ***k***-местных функций $\pi_k^i(x_1,\dots,x_k)=x_i$.

Сложение. Функция **<x,y> ⟶ sum(x,y) = x+y** получается с помощью рекурсии:

sum(x,0)=x;

sum(x,y+1)=sum(x,y)+1.

Надо, конечно, представить правую часть второго равенства как результат подстановки. Формально говоря, ***h(x,y,z)*** в определении рекурсии надо положить равным ***s(z)***, где ***s*** функция прибавления единицы.

Умножение. Функция **< x,y> ⟶prod(x,y) = xy** получается с помощью рекурсии (**с использованием сложения!!!!!!!**):

prod(x,0)=0;

prod(x,y+1)=prod(x,y)+x.

Рекурсивная функция:

f(0)=0;

f(n) = n + f(n-1), n > 0

может быть переведена в замкнутую форму

$$f = \frac{n(n+1)}{2}$$

Замкнутая форма может быть найдена не для всех рекурсивных функций (соотношений). Для некоторых из них найдены лишь приближенные замкнутые формы.

Примеры: треугольник Паскаля (Пингалы, Хайама, Яна Хуэя) ,

числа Фиббоначи

Первые 15 строк треугольника Паскаля ($n = 0, 1, \ldots, 14$)

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Рекурсия + мемоизация = динамическое программирование

При формальном определении РФ впервые были найдены способы построения (конструирования) всех возможных функций, вычислимых алгоритмами.

Слова «всех возможных функций» должны пониматься так: если кто-то придумал некоторую (очень сложную) функцию, вычислимую «механическим способом», то такая функция может быть записана в виде формальной схемы по правилам РФ.

Понятно, что все конструкторские механизмы (принципы и схемы) РФ так или иначе должны быть реализованы в языках программирования.

Существует **гипотеза Чёрча**, что класс РФ совпадает с классом всех функций, допускающих алгоритмическое вычисление.

3. Исчисления.

Представляют собой **формальные системы**, в которых определены понятия правильно построенных формул (**ППФ**) и конечного набора правил {**P**} преобразования **ППФ**.

Алгоритм понимается как цепочка применения последовательности правил из {P} к начальной ППФ для получения вывода конечной ППФ.

В **аксиоматических формальных системах** конечные ППФ всегда выводятся из ППФ, которые называются аксиомами.

Содержательно исчисление задаёт конструктивный способ порождения слов некоторого языка.

Например, исчисление конечных разностей Ньютона даёт возможность порождать (исчислять) для некоторого правильного выражения формул с дифференциалами все эквивалентные ему выражения, заданные правилами тождественных преобразований. Исчисление высказываний позволяет конструктивно порождать, все возможные логические законы в виде тождественно истинных высказываний. Исчисление предикатов позволяет строить формальные теории в виде системы формул, которые истинны в выбранной интерпретации

Примеры исчислений в IT

- **Исчисление функций**, вычисляемых на множестве натуральных чисел предложено Эрбраном и Гёделем в 1938 г.

- **λ-исчисление** А.Чёрча также может быть отнесено к этому типу алгоритмов, предложено в 1937 г.

- **Формальные грамматики**, порождающие языки, предложены Хомским в 1953 – 1956 г.

- **Распознающие автоматы (конечные и магазинные автоматы), конструктивная логика и т.д.**

- **Любые языки программирования, которые имеются на сегодняшний день**, при условии бесконечной памяти.

Структура алгоритма
(составляющие алгоритма)

- **Процессорная структура**. (исполнитель алгоритма). Во всех теоретических конструкциях алгоритмов и большинстве алгоритмических языках это единственный процессор.

- **Структуры данных**. Структура данных это способ организация записи, хранения и извлечение данных.

Данные – это элементы множеств, которые нужно порождать или распознавать. Содержательные данные могут быть определены весьма экзотическим образом и требуют иногда большого искусства в отображении их на организацию данных, которые разрешены в тех или иных конструкциях алгоритмов и языков программирования.

- **Информационная структура алгоритма (ИСА)**. Структура функций есть описание конструирования функции из базовых функций . Заметим, что функция может быть определена двумя способами:

а) структурой (структурным описанием)

б) процедурно (последовательностью базовых операций).

Языки программирования содержат оба способа задания функций.

- **Логическая структура алгоритма (ЛСА) или программы (ЛСП)**. ЛСА суть описание организации вычислительного процесса, который управляется состоянием памяти.

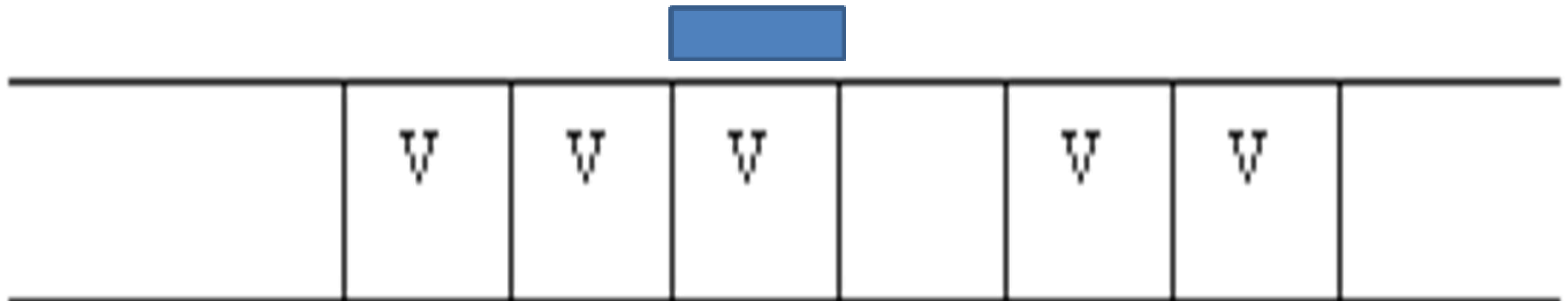
Машина Поста

Абстрактная машина Поста состоит из

бесконечной ленты, разделенную на **одинаковые** клетки, каждая из которых может быть либо пустой, либо заполненной меткой «V»,

головки, которая может

1. перемещаться вдоль ленты на одну клетку вправо или влево,
2. наносить в клетку ленты метку, если этой метки там ранее не было,
3. стирать метку, если она была,
4. проверять наличие в клетке метки.

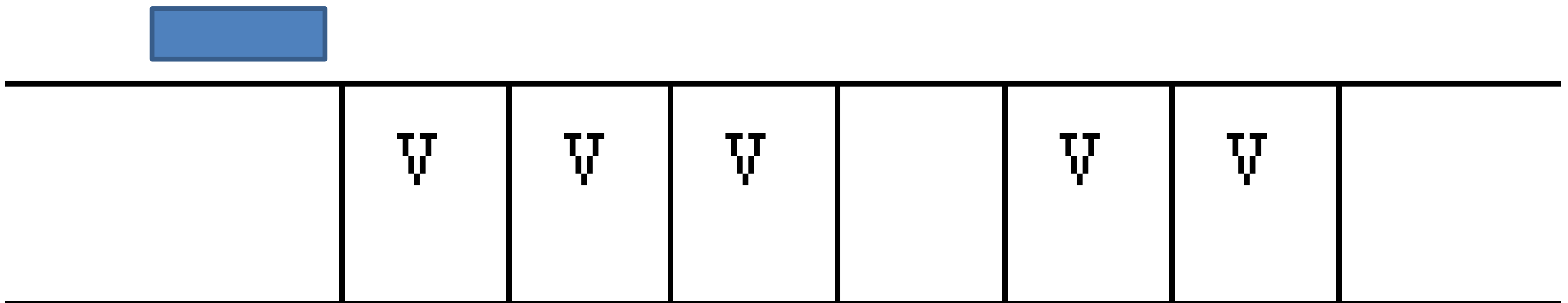


Информация о заполненных метками клетках ленты характеризует состояние ленты, которое может меняться в процессе работы машины.

В каждый момент времени головка («-») находится над одной из клеток ленты и, как говорят, обозревает ее.

Информация о местоположения головки вместе с состоянием ленты характеризует состояние машины Поста

Для работы машины нужно задать программу и начальное состояние (т. е. состояние ленты и позицию каретки). Мы будем понимать под ***начальным состояние головки*** ее положение против пустой клетки левее самой левой метки на ленте.



Команда машины Поста имеет следующую структуру:

n K m,

n - порядковый номер команды,

K - действие, выполняемое головкой, (**их всего шесть**)

m - номер следующей команды, подлежащей выполнению.

Ситуации, в которых головка должна наносить метку там, где она уже имеется, или, наоборот, стирать метку там, где ее нет, являются ***аварийными (недопустимыми)***.

Программой для машины Поста (алгоритмом) будем называть непустой список команд, такой что на **n**-м месте команда с номером **n**; номер каждой команды совпадает с номером какой-либо команды списка.

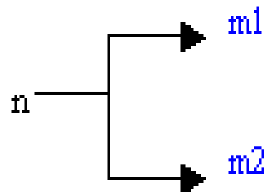
Движение головки на одну клетку вправо n→m.

Движение головки на одну клетку влево n←m.

Нанесение метки на клетку, над которой находится головка n M m.

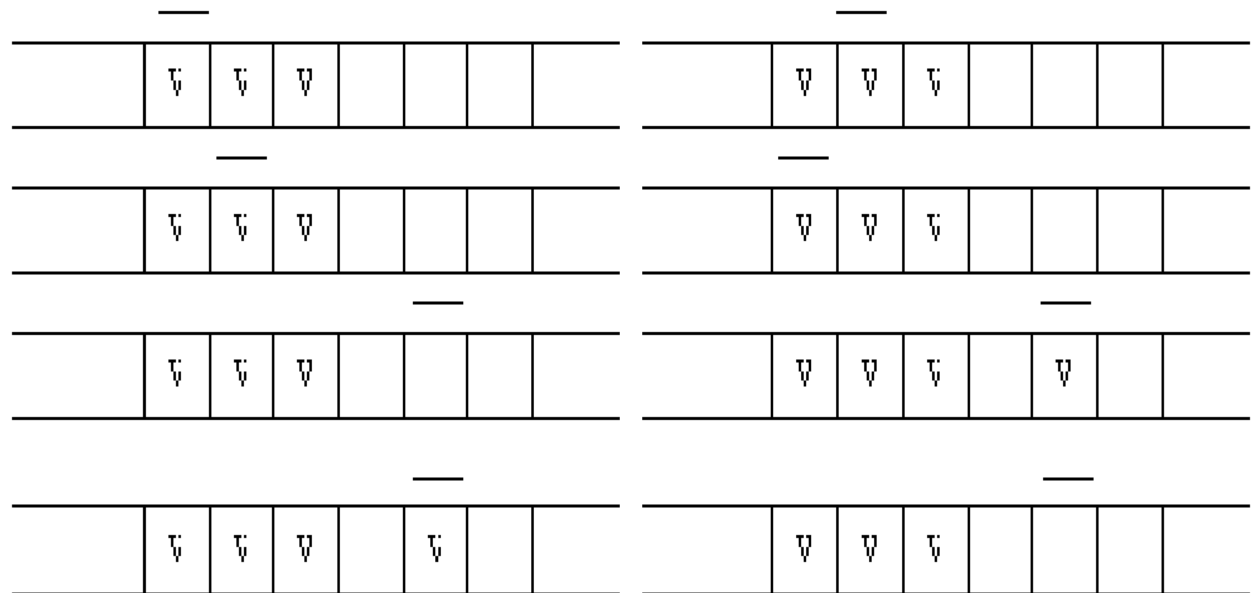
Стирание метки из клетки, над которой находится головка n C m.

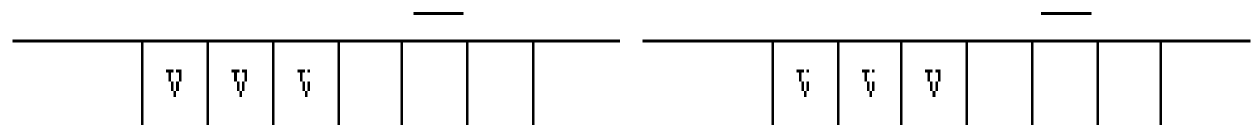
Проверка наличия метки в клетке, над которой находится головка. Если метка отсутствует, то управление передается команде m1, а иначе m2.



Остановка машины

n Стоп m





С точки зрения свойств ***алгоритмов***, изучаемых с помощью машины Поста, наибольший интерес представляют причины останова машины при выполнении программы.

- ***останов по команде "стоп»***

Такой останов называется ***результативным*** и указывает на корректность алгоритма (программы);

- ***останов при выполнении недопустимой команды***.

В этом случае останов называется ***безрезультативным***;

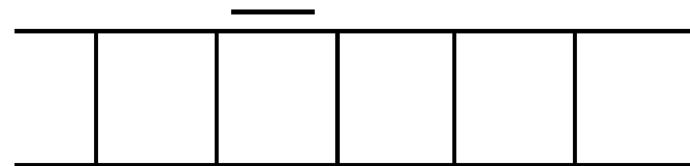
- ***машина не останавливается никогда***.

В этом и в предыдущем случае мы имеем дело с ***некорректным алгоритмом (программой)***.

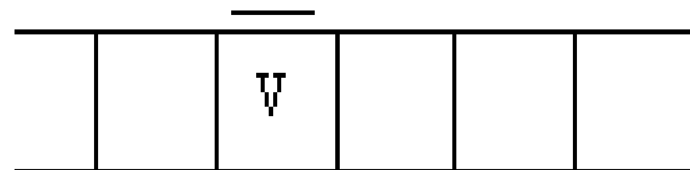
Пример программы (алгоритма)

Начальное состояние

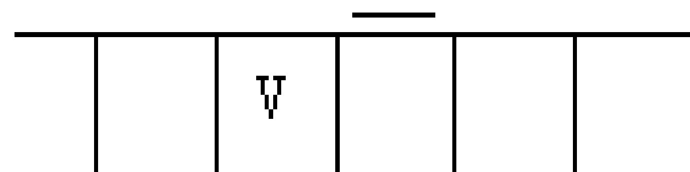
1 М 2



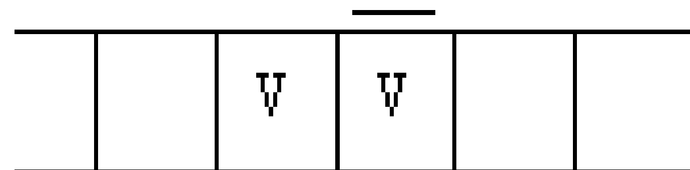
2 → 3



3 М 4



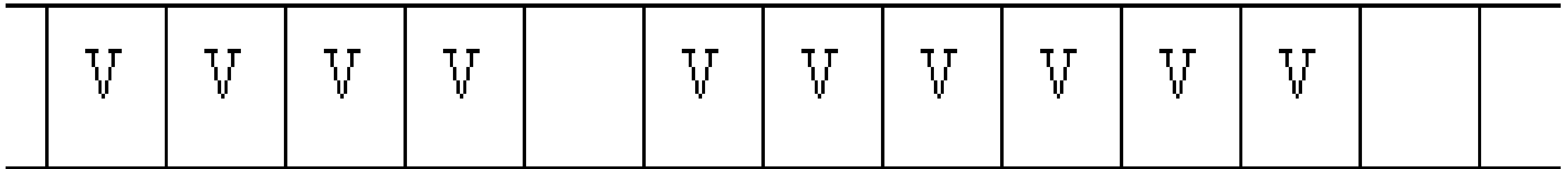
4 Стоп 4



Число **k** представляется на ленте машины Поста идущими подряд **k+1** метками (одна метка означает число «0»).

Между двумя числами делается интервал как минимум из одной пустой секции на ленте.

Например, запись чисел 3 и 5 на ленте машины Поста будет выглядеть так:

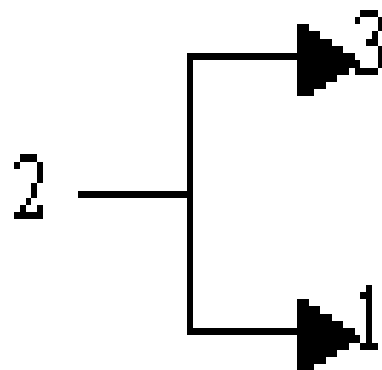


Используемая в машине Поста система записи чисел является ***непозиционной***

Составим программу для прибавления к произвольному числу единицы. Предположим, что на ленте записано только одно число и головка находится над одной из клеток, в которой находится метка, принадлежащая этому числу.

Для решения задачи можно переместить головку влево (или вправо) до первой пустой клетки, а затем нанести метку.

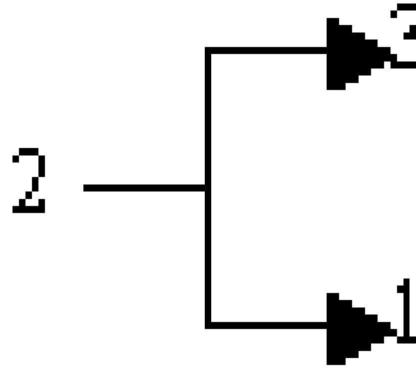
Программа, добавляющая к числу метку справа



3 М 4

4 Стоп 4

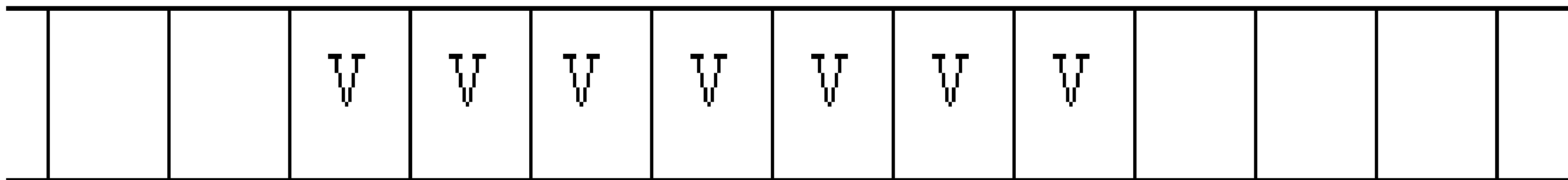
Программа, добавляющая к числу метку слева



3 M 4

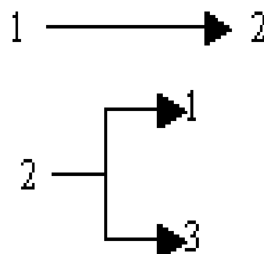
4 Стоп 4

Предположим, что головка расположена на расстоянии нескольких клеток слева от числа, к которому нужно прибавить единицу



В этом случае программа усложняется.

Появится "блок поиска числа" - две команды, приводящие головку в состояние, рассмотренное в предыдущем примере



Машина Тьюринга (МТ).

состоит из

- ✓ **счетной ленты**

  разделенной на ячейки иногда **ограниченной слева**, но не справа

- ✓ **читающей и пишущей головки**

  которая может читать буквы рабочего алфавита $\mathbf{A = \{a_0, a_1, \ldots, a_t\}}$, стирать их и печатать.

- ✓ **лентопротяжного механизма**

- ✓ **операционного исполнительного устройства**

  которое может находиться в одном из дискретных состояний $\{s_0, s_1, \ldots, s_k\}$, принадлежащих алфавиту **S** внутренних состояний.

Порядок работы МТ описывается ***таблицей машины Тьюринга***.

Эта таблица является матрицей с четырьмя столбцами и **(k + 1)(t + 1)** строками. Каждая строка имеет вид

$$s_i\ a_j\ v_{ij}\ s_{ij}\ , \qquad 0 <= i < k, \quad 0 <= j <= t,$$

Если МТ находится в исходном состоянии s_i, то головка прочитывает символ a_j в рабочей ячейке, выполняет команду v_{ij} и переходит в состояние s_{ij}

Что такое v_{ij}

- **r** - переместить ленту вправо,
- **l** - переместить ленту влево,
- **stop** - остановить машину;
- - действие МТ, состоящее либо в занесении в ячейку ленты символа алфавита a_k , стирая при этом существующий в ячейке символ, либо в изменении состояния s_{ij}

Машина Тьюринга называется **детерминированной**, если каждой комбинации состояния и ленточного символа в таблице соответствует не более одного правила.

$$s_i \ a_j \ v_{ij} \ s_{ij}$$

Если существует пара «ленточный символ — состояние», для которой существует 2 и более команд, такая машина Тьюринга называется **недетерминированной,** (при этом машина либо “удачлива”, либо делает копии, либо может быть вероятностной).

Каждое правило перехода предписывает машине, в зависимости от текущего состояния и наблюдаемого в текущей клетке символа, записать в эту клетку новый символ, перейти в новое состояние и переместиться на одну клетку влево или вправо.

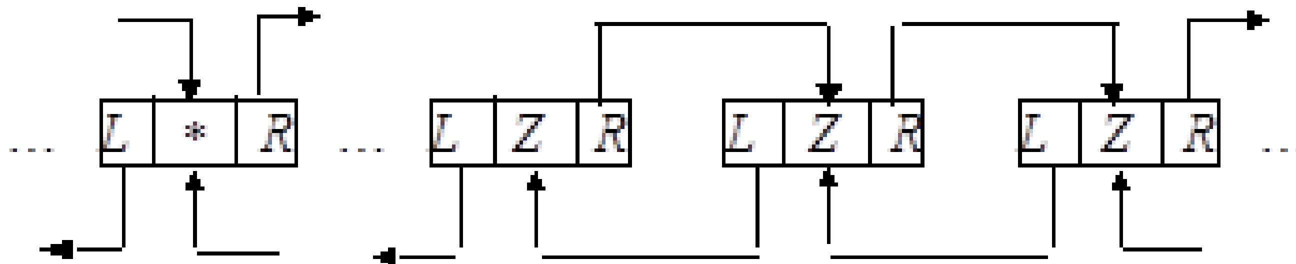
Некоторые состояния машины Тьюринга могут быть помечены как терминальные, и переход в любое из них означает конец работы, остановку алгоритма.

Математическое описание МТ

1) $MT = \langle A, S, P, D \rangle$ (запись обозначает: МТ определяется формально четверкой конечных множеств), где

- $A = \{a_1, a_2, ..., a_n\}$ – алфавит рабочих символов
- $S = \{s_0, s_1, ..., s_k\}$ – алфавит состояний
- $D = \{Л, П, Н, stop\}$ – алфавит действий
- $P = \{p_1, p_2, ..., p_i, ..., p_l\}$ – набор правил вида – $(s_i, a_i \to a_j, s_j, d_j)$, где $S_i, S_j \in S; a_i, a_j \in A; d_j \in D \;\; i, j = 1, 2, ..., l$ (иногда называют программой МТ).

Лента МТ представляет своеобразную **структуру данных**, которая может моделироваться двунаправленным линейным списком, вообще говоря, растущим в обе стороны.



∗ – спейсер, ограничивающий запись слова - строка Тьюринга (строки Ляпунова и Маркова)

Сложение двух чисел x и y в унарной системе, как пример порождающей МТ.

$A = \{1, +, (,), =, \varphi\}$ – алфавит рабочих символов (располагаются на ленте).

$S = \{S_0, S_=, S_+, S_), S_k\}$ – алфавит состояний, переход в то или иное состояние, определяет логику работы алгоритма.

Начальное состояние ленты

(11+111)

где ***x=11*** и ***y=111***.

УГ сдвинута (установлена) на левый спейсер «(».

Результат порождения ***x+y=11111*** .

$\longrightarrow$ - означает заменить на

Правила	Условие		Команда П
$P_1 : S_0(\to S_= r$	если $S_0($	то	$\Pi_1 \ (\to S_= r$
$P_2 : S_= + \to 1 S_+ r$	если $S_= +$	то	$\Pi_2 \ (+ \to 1 S_+ r$
$P_3 : S_= 1 \to 1 S_= r$	если $S_= 1$	то	$\Pi_3 \ (1 \to 1 S_= r$
$P_4 : S_+) \to \varphi S_) l$	если $S_+)$	то	$\Pi_4 \ ) \to \varphi S_) l$
$P_5 : S_+ 1 \to 1 S_+ r$	если $S_+ 1$	то	$\Pi_5 \ 1 \to 1 S_+ r$
$P_6 : S_) 1 \to) S_k \ stop$	если $S_) 1$	то	$\Pi_6 \ 1 \to) S_k \ stop$

Проверьте работу программы !!!! (что делает четвертое правило?)

Логическая схема алгоритма (ЛСА) задаётся либо матрицей переходов, либо графом:

Граф переходов МТ есть направленный бинарный граф **G=<S,V>,**

- ***S*** – вершины, которые соответствуют множеству состояний МТ;
- ***V*** – множество дуг, которые соответствуют переходам (правилам) из состояния S_i в состояние S_j

каждая дуга помечается состоянием памяти a_i (символом a_i, содержащимся в ячейке ленты, на которую указывает головка) и командой $П_j$, которую должна выполнить УГ в данном такте *j*.

$$(S_i, a_i \rightarrow S_j, \Pi_j \text{ или при именовании тактов } S_t, a_t \rightarrow S_{t+1}, \Pi_{t+1}).$$

Граф переходов G_S замечателен тем, что он содержит все возможные «траектории» работы МТ, которые могут быть сколь угодно длинными (но конечными) и (это основное) определяет **ЛСА** работы МТ, начиная с начального состояния S_0, до момента останова.

Граф G_S в этом случае задаёт так называемую **машину состояний**.

ЛСА, также определяющая все возможные траектории (последовательности состояний и соответствующих команд) может быть определена и другой графовой конструкцией – **блок–схемой алгоритма**

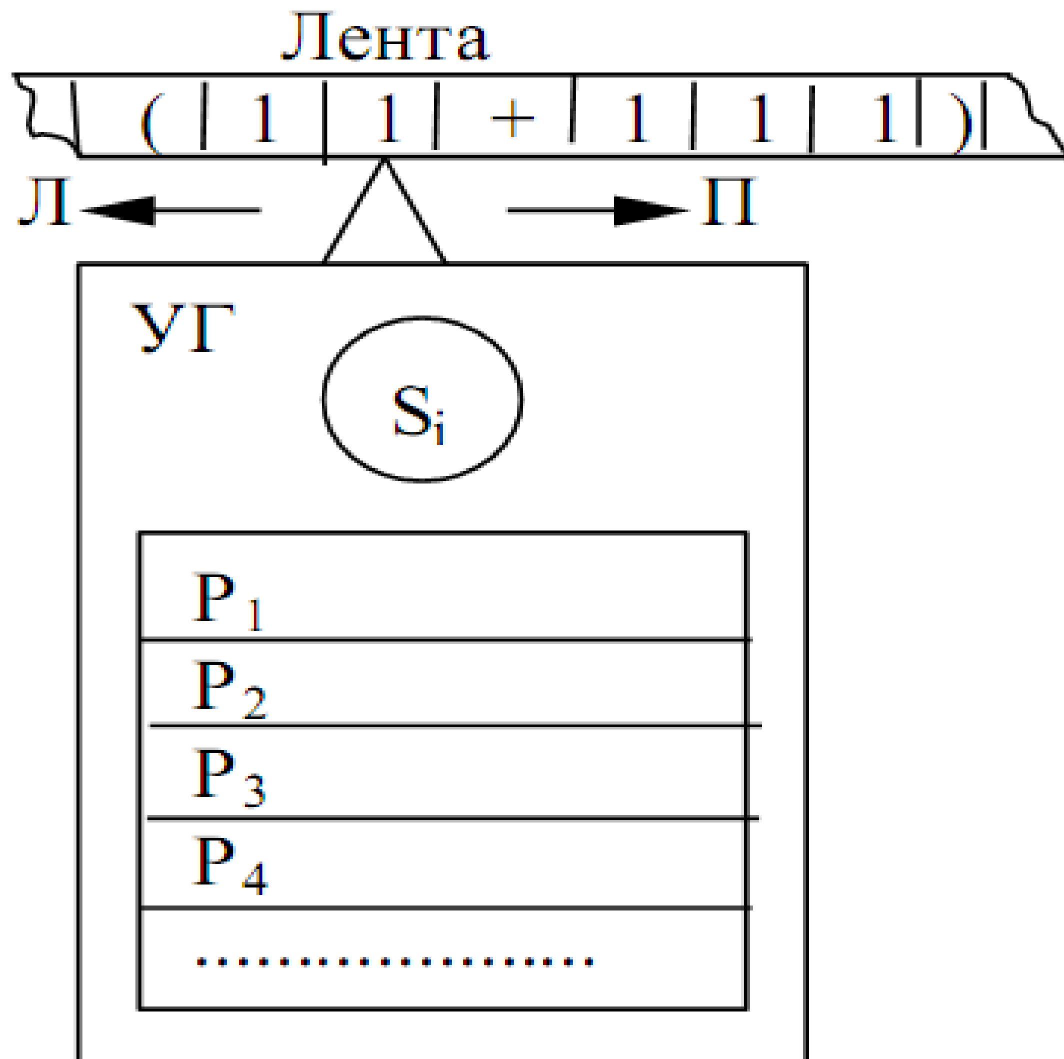
Блок–схема алгоритма (БСА) представляет направленный бинарный граф **G=<R,V>** где ***R*** – вершины двух типов

П – соответствуют действиям, командам или программам

r – соответствуют процедурам, распознающим состояния памяти.

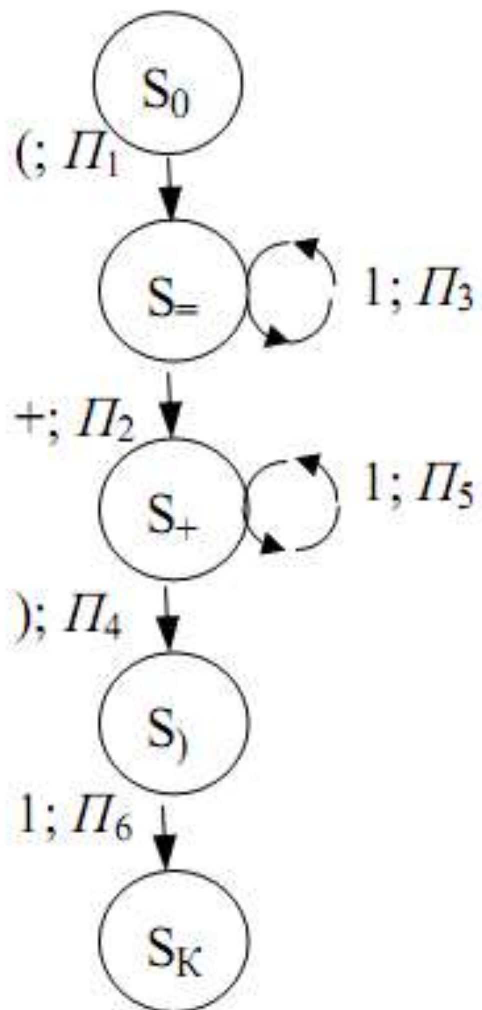
Каждая дуга $v \in V$ помечается либо состоянием памяти (условием перехода), либо символом «**≡**» безусловного перехода.

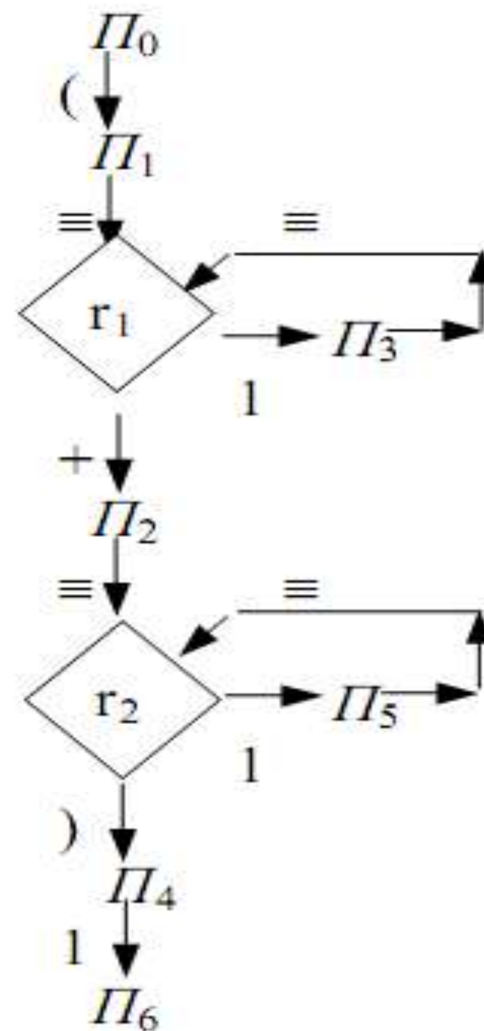
Пример: сложение двух чисел



Машина состояний

Блок-схема

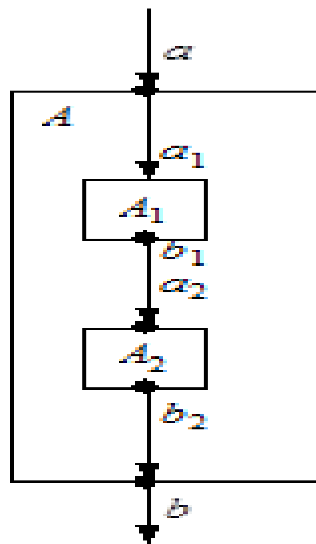




Структурное (структурированное) программирование - запрет GO TO.

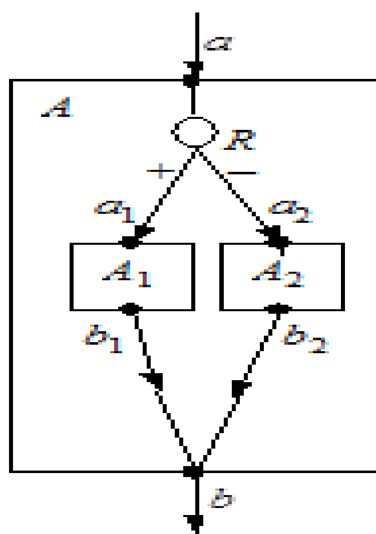
В начале 70–х годов Дейкстра предложил принцип последовательного уточнения логической структуры алгоритмов. Внутреннее содержание каждого программного блока в БСА уточняется одним из четырёх способов, показанных на рисунке.

а) последовательность



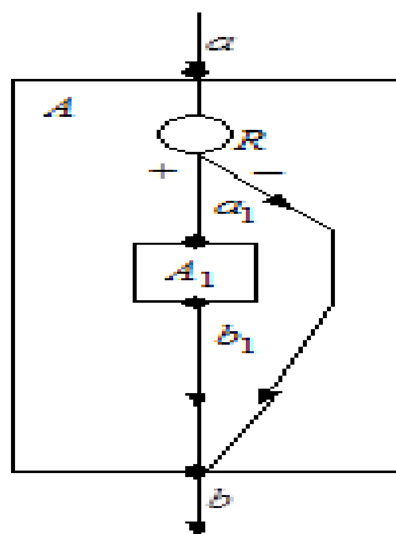
begin
begin
A_1
end
begin
A_2
end
end

б) альтернатива



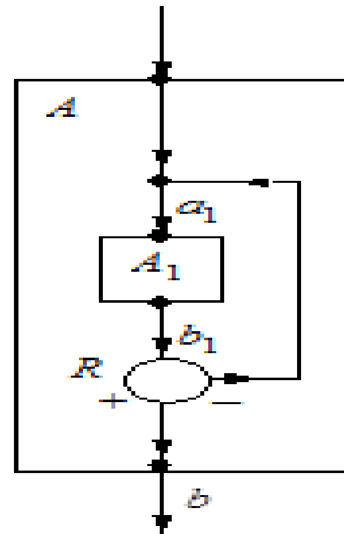
begin
begin
if R then A_1
else A_2;
end
end end

в) цикл с предусловием



begin
begin
while R do A_1
end
end

г) цикл с постусловием



begin
begin
repeat A_1
until R
end
end

В одной довольно известной компании программистам стали давать премии за **количество строчек кода**, и они в результате стали это число раздувать посредством всяких ухищрений.

Использовали машинный анализ исходного кода, а не просто количество "физических" строк в файлах. Иными словами, это скорее был подсчёт отдельных инструкций (**statements**) в языке, хотя назывался он "количество строк кода".

Программисты быстро обнаружили, что если добавить "else" перед точкой с запятой, то их продуктивность удваивается.

```
if (v > 0) then x = 3 else ;
if (v > 0) then y = sqrt(v) else ;
if (v > 0) then print(y) else ;
```

там где обычно программист написал бы

```
if (v > 0) then begin x = 3; y = sqrt(v); print(y) end ;
```

Дело в том, что "обычный" способ считался за 3 строчки кода (или 4, если добавить пустой else в конце), а "новый" за 6.

Нормальные алгоритмы (алгорифмы) Маркова

Для формализации понятия алгоритма российский математик ***А.А.Марков*** предложил использовать ***ассоциативные исчисления***.

- Пусть имеется алфавит (конечный набор символов).
- Составляющие его символы будем называть ***буквами***.
- Любая конечная последовательность букв алфавита (линейный их ряд) называется ***словом*** в этом алфавите.
- Рассмотрим два слова **N** и **M**, если **N** является частью **M**, то говорят, что **N** входит в **M**.
- **Подстановки** позволяют менять слова

Строка Маркова представляет собой слово , но в отличие от ленты МТ строка обладает свойством **сжатия** при замене символов алфавита на «пустой» символ $\varnothing$ и **расширения** при вставлении символов.

Это свойство СМ позволяет значительно упростить распознающие алгоритмы, но также значительно усложнить их моделирование на процедурных языках.

Зададим в некотором алфавите конечную систему **подстановок**

N - M, S - T,..., где **N, M, S, T**,... - слова в этом алфавите.

Любую подстановку **N - M** можно применить к некоторому слову **K** следующим способом: если в **K** имеется одно или несколько вхождений слова **N**, то любое из них может быть заменено словом **M**, и, наоборот, если имеется вхождение **M**, то его можно заменить словом **N**.

Пример

В алфавите **A = {a, b, c}** имеются слова

N = ab, **M = bcb**, **K = abcbcbab**.

Заменив в слове **K** слово **N** на **M**, получим

bcb cbcbab или **abcbcb bcb**.

Подстановка **ab - bcb** недопустима по отношению к слову **bacb**,
так как ни **ab**, ни **bcb** не входит в это слово.

К полученным с помощью допустимых подстановок словам можно снова применить допустимые подстановки и т.д.

Совокупность всех слов в данном алфавите вместе с системой допустимых подстановок называют ассоциативным исчислением.

Чтобы задать ассоциативное исчисление, достаточно задать алфавит и систему подстановок.

Каждому исчислению соответствует некоторое множество пар слов **<X,Y>** что **X** можно преобразовать в **Y** по правилам этого исчисления.

Теорема.

Для всякого исчисления множество пар слов перечислимо.

Существует исчисление, для которого это множество неразрешимо.

Основные определения:

Слова P_1 и P_2 в некотором ассоциативном исчислении называются ***смежными***, если одно из них может быть преобразовано в другое однократным применением допустимой подстановки.

Последовательность, слов **P, P_1, ...,M** называется ***дедуктивной цепочкой***, ведущей от слова **P** к слову **M**, если каждое из двух рядом стоящих слов этой цепочки - смежное.

Слова **P** и **M** называют ***эквивалентными***, если существует цепочка от **P** к **M** и обратно.

Пример

Алфавит: **{a, b, c, d, e}**

Подстановки:

ac - ca;

abac - abace;

ad - da;

eca - ae;

be - cb;

eda - be;

bd - db;

edb - be.

Слова **abede** и **acbde** – смежные (подстановка **be** - **cb**).

Слова **abcde** - **cadbe** эквивалентны.

Может быть рассмотрен специальный вид ассоциативного исчисления, в котором подстановки являются ориентированными: **N->M** (стрелка означает, что подстановку разрешается производить лишь слева направо).

Для каждого ассоциативного исчисления существует задача: для любых двух слов определить, являются ли они эквивалентными или нет.

Любой процесс вывода формул, математические выкладки и преобразования также являются дедуктивными цепочками в некотором ассоциативном исчислении.

Построение ассоциативных исчислений является универсальным методом детерминированной переработки информации и позволяет формализовать понятие алгоритма.

Ассоциативное исчисление называется **двусторонним**, если оно вместе с каждым правилом **X-Y** содержит и симметричное правило **Y- X**.

***Алгоритмом** в алфавите* **А** называется понятное точное предписание, определяющее процесс над словами из **А** и допускающее **любое слово** в качестве исходного.

Алгоритм в алфавите **А** задается в виде системы допустимых подстановок, дополненной точным предписанием о том, в каком порядке нужно применять допустимые подстановки и когда наступает остановка.

Алфавит А: **{a, b, c}** . **Система подстановок В**: **cb - ee; cca - ab; ab - bca.**

Предписание о применении подстановок: в произвольном слове **Р** надо сделать возможные подстановки, заменив левую часть подстановок на правую; повторить процесс с вновь полученным словом.

babaac -> **bbcaaac** -> **остановка**

bcacabc -> **bcacbcac** -> **bcacccac** -> **bcacabc** ->

бесконечный процесс (остановки нет), так как мы получили исходное слово.

Предложенный А.А.Марковым способ уточнения понятия алгоритма основан на понятии ***нормального алгоритма***, который определяется следующим образом.

Пусть задан алфавит **А** и система подстановок **В**.

Для произвольного слова **Р** подстановки из **В** подбираются в том же порядке, в каком они следуют в **В**.

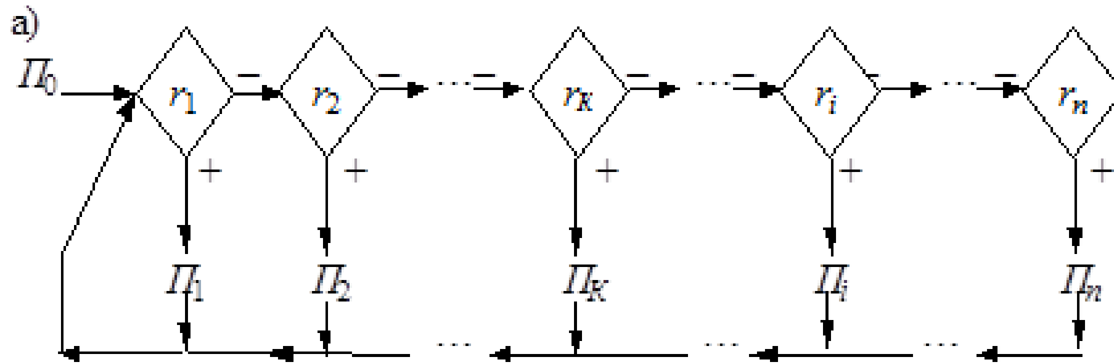
Если подходящей подстановки нет, то процесс останавливается.

В противном случае берется первая из подходящих подстановок и производится замена ее **правой частью** первого вхождения **ее левой части** в **Р**.

Затем все действия повторяются для получившегося слова **Р1**.

Если применяется последняя подстановка из системы подстановок **В**, процесс останавливается.

Стандартная логическая структура любого НАМ



Простота построения распознающих алгоритмов на структурах данных типа “строка Маркова” сделало НАМ весьма популярным как для теории, так и для практики программирования.

Программа на языке алгоритмов Маркова - представляет из себя набор правил.

Каждое правило представляет собой замену

$$S_1 \rightarrow S_2$$

где S_1 и S_2 некие строки. Правило представляет подстановки, **последовательно** применяемые ко входной строке и приводящие в итоге ее к требуемой выходной строке.

Работает алгоритм этот следующим образом:

1. Берётся исходная строка и мы начинаем перебирать правила с самого первого, анализируя, может ли оно быть применено (существует ли в строке S подстрока S_1).
   Если не может -> анализируется следующее по порядку правило.

2. Если не одно правило не подошло, алгоритм завершается, текущее состояние строки S является результатом работы алгоритма.

3. Если же правило применимо - совершается замена самого левого вхождения подстроки S_1 на строку S_2 . Причём, (что очень важно!) далее правила начинают перебираться опять с начала – т.е. пункт 1.

4. Ещё есть так называемые *терминальные* правила, обозначающиеся точкой в конце: При срабатывании этого правила алгоритм завершается и текущее состояние строки S считается результатом работы.

Пример

Приведем пример нормального алгоритма, описывающего сложение натуральных чисел (представленных наборами единиц).

Алфавит: **A={+,1}**

Система подстановок **B**: (**упорядоченная!!!**)

1. **1+ -> + 1**
2. **+ 1 -> 1**
3. **1 -> 1**

Слово **P: 11+11+111**.

P = 1 1 + 11 + 111

P_1 = 1 + 1 11 + 111

P_2 = + 1 111 + 1 11

P_3 = + 111 + 1 111

P_9 = 1111111

Алгоритм ***нормализуем***, если можно построить эквивалентный ему нормальный алгоритм.

Принцип нормализации: **все алгоритмы нормализуемы.**

I. ***Суперпозиция алгоритмов***.
II. ***Объединение алгоритмов***.
III. ***Разветвление алгоритмов***.
IV. ***Итерация алгоритмов***.

Любой нормальный алгоритм **эквивалентен** некоторой машине Тьюринга, и наоборот — любая машина Тьюринга эквивалентна некоторому нормальному алгоритму.

Нормальные алгоритмы оказались удобным средством для построения многих разделов конструктивной математики.

Кроме того, заложенные в определении нормального алгоритма идеи используются в ряде ориентированных на обработку символьной информации языков программирования — например, в языке **Рефал**.

Умножение десятичного числа на 2.

0[0]>[0]0
0[1]>[0]1

1[0]>[0]2
1[1]>[0]3

2[0]>[0]4
2[1]>[0]5

3[0]>[0]6
3[1]>[0]7

4[0]>[0]8
4[1]>[0]9

5[0]>[1]0
5[1]>[1]1

6[0]>[1]2
6[1]>[1]3

7[0]>[1]4
7[1]>[1]5

8[0]>[1]6
8[1]>[1]7

9[0]>[1]8
9[1]>[1]9

[0]>.
[1]>1.

Императивный стиль

1. **Программа** - последовательность операторов (команд), выполняемых компьютером.
2. Использующий императивный стиль, **программист должен объяснить компьютеру**, как нужно решать задачу.
3. Императивная программа очень похожа на **приказы, выражаемые повелительным наклонением в естественных языках,** то есть это последовательность команд / логических переходов, которые должен выполнить компьютер.
4. **Основные конструкции:**
   - ✓ Манипулирование ячейками памяти
   - ✓ Оператор присваивания
   - ✓ Явные операторы передачи управления
   - ✓ Циклы, условный оператор
5. Решая задачу, императивный **программист вначале создает модель** в некоторой формальной системе, а затем переписывает решение на императивный язык.
6. Для человека рассуждать в терминах компьютера довольно неестественно. Во-вторых, последний этап этой деятельности (**переписывание решения на язык программирования),** что по сути не имеет отношения к решению исходной задачи.
7. Часто императивные программисты даже разделяют работу в соответствии с двумя описанными выше этапами. Одни люди, постановщики задач, придумывают решение задачи, а другие, кодировщики, переводят это решение на машинный язык.

Декларативный стиль

1. **программа** - совокупность утверждений, описывающих фрагмент предметной области или сложившуюся ситуацию;

   описывается результат (его свойства), а не методы его достижения.

2. программируя в декларативном стиле, программист должен описать, что нужно решать.

3. **В основе декларативных языков лежит формализованная человеческая логика**. Человек лишь описывает решаемую задачу, а поиском решения занимается императивная система программирования.

В итоге получаем значительно большую скорость разработки приложений, значительно меньший размер исходного кода, легкость записи знаний на декларативных языках, более понятные, по сравнению с императивными языками, программы

Парадигма программирования - исходная концептуальная схема постановки проблем и их решения; вместе с языком, ее формализующим, парадигма формирует стиль программирования

Процедурная
Функциональная
Логическая
Объектно-ориентированная

Языки программирования

Императивные (алгоритмические языки)

- Процедурные языки (**Fortran**, **Pascal**, **C**, …)
- Объектно-ориентированные языки (**C++**, SmallTalk …)

Декларативные (неалгоритмические) языки

- Языки **логического** программирования (**Prolog**, …)
- Языки **функционального** программирования (**Lisp**, …)

парадигма

процедурное программирование

1. **Программа состоит из структур данных** (объектов обработки) и алгоритма (метода обработки). Процесс вычисления это последовательность изменения состояний этих структур, которые программист описывает **явно**!!!!
2. **Базовая концепция - переменная**, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма .
3. Для управления процессом выполнения используются следующие конструкции, последовательность, ветвление, цикл и вызов подпрограммы.
4. Эта парадигма является самой старой. Она развивалась по мере появления новых концепций в языках программирования: трансляция (Ассемблер, Fortran, Cobol), типизация (Pascal), модули (Modula), специализация на конкретной области применения (RPG, Clipper) и универсальность (PL/I, C, Ada).

парадигма

функциональное программирование

1. Процесс вычисления трактуется как вычисление значений **функций** в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).
2. Поэтому единственной управляющей конструкцией является **вызов функции**.
   В языке, реализующем функциональную парадигму, существует некоторое множество базовых функций, и все другие функции строятся из базовых функций с помощью композиции.
3. Теоретической основой является **лямбда-исчисление и теория рекурсивных функций.**
4. В настоящее время существуют сотни функциональных языков программирования, ориентированных на разные классы задач и виды технических средств: Common Lisp, Haskell, F# …

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на **аргументы**, но и на **состояние внешних по отношению к функции переменных**, а также иметь побочные эффекты и менять состояние внешних переменных.

Таким образом, в **императивном программирова**нии при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить **разные данные на выходе** из-за влияния на функцию состояния переменных.

А в **функциональном языке** при вызове функции с одними и теми же аргументами мы всегда получим **одинаковый результат**: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом.

парадигма

логическое программирование

1. Логическое программирование основано на теории и аппарате **математической логики** с использованием математических принципов резолюций.
2. Вместо описания алгоритма решения задачи описывается **мир задачи**, какие имеются объекты, их свойства и отношения между ними.
3. За основу описания берутся отношения между объектами.
4. Логическая программа представляет собой набор отношений, которые называются фактами, и правил, на основании которых могут быть получены новые отношения. Она не задает никакого процесса вычислений. Это своего рода база данных (БД) о предметной области задачи. Ее применение инициализируется запросом. **Поиск ответа на запрос заключается в попытке логического вывода запроса на основании фактов и правил, имеющихся в БД**. Поиск решения выполняется специальной программой - интерпретатором.
5. Основной (самый популярный) язык – Prolog, с множеством диалектов.
6. Другие (менее популярные) языки: Datalog, Mercury, Oz.

парадигма

объектно-ориентированное программирование

Основные концепции - понятия **объектов** и **классов** (либо, в менее известном варианте языков с прототипированием, — прототипов).

Класс — это **тип, описывающий устройство объектов**. Понятие «класс» подразумевает некоторое поведение и способ представления. Понятие «объект» подразумевает нечто, что обладает определённым поведением и способом представления. При этом элементы такой структуры (члены класса) могут сами быть не только данными, но и методами (то есть процедурами или функциями). Такое объединение называется **инкапсуляцией**.

Объект — сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции (и связывания) исходного кода на выполнение). Каждый объект наследует свойства своего класса и может иметь свои собственные свойства.

Прототип — это объект-образец, по образу и подобию которого создаются другие объекты.

Программа в ООП - это совокупность объектов, обменивающихся между собой сообщениями. Мир задачи описывается как совокупность объектов, обладающих некоторыми свойствами и вступающих во взаимодействие. За основу описания берутся не отношения, а сами объекты.

Наличие **инкапсуляции** достаточно для объектности языка программирования, но ещё не означает его объектной ориентированности — для этого требуется наличие **наследования**.

Но даже наличие инкапсуляции и наследования не делает язык программирования в полной мере объектным с точки зрения ООП - необходим **полиморфизм.**

Абстракция – это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, абстракция – это набор всех таких характеристик.

Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым или родительским. Новый класс – потомком, наследником или производным классом.

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Источники

1. А. Н. Колмогоров, Теория информации и теория алгоритмов. — М.: Наука, 1987. — 304 с.
2. Марков А. А., Нагорный Н. М. Теория алгорифмов, изд. 2. — М.: ФАЗИС, 1996
3. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ. — М.: «Вильямс», 2006. с. 1296.
4. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы — 3-е изд. — М.: «Вильямс», 2006. — С. 720. — ISBN 0-201-89683-4
5. С. А. Абрамов Лекции о сложности алгоритмов, МЦНМО, 2009 г., 256 стр.
6. А. К. Гуц, Математическая логика и теория алгоритмов, Либроком, 2009 г., 120 стр.
7. Лекции В.М. Абрамова "Алгоритмы и алгоритмические языки", МФТИ.
8. Лекции Л.Н. Столярова "Информатика и применение компьютеров в научных исследованиях", МФТИ.
9. Н.К. Верещагин, А.Х. Шень, Основы теории вычислимых функций, курс INTUIT.
10. Н.К. Верещагин, А.Х. Шень, Языки и исчисления, курс INTUIT.
11. В.С. Рублев, Языки логического программирования, курс INTUIT