

Обзор парадигм программирования

Юрий Литвинов
yurii.litvinov@gmail.com

30.11.2018

Математические модели вычислений

- ▶ Что можно посчитать имея вычислительную машину неограниченной мощности?
- ▶ Формальные модели вычислений:
 - ▶ Машина Тьюринга
 - ▶ λ -исчисление Чёрча
 - ▶ Нормальные алгоритмы Маркова
- ▶ Тезис Чёрча: «Любая функция, которая может быть вычислена физическим устройством, может быть вычислена машиной Тьюринга.»

Машина Тьюринга

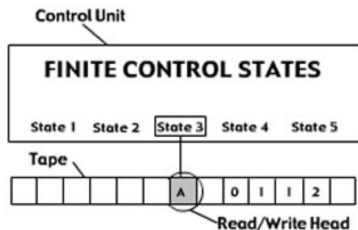
- ▶ Формально,

$$M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$$

$$\delta : (Q/F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

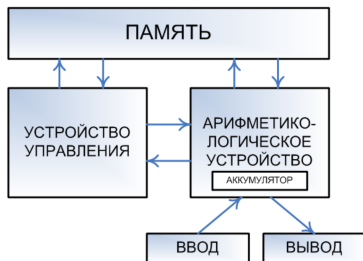
- ▶ Неформально:

- ▶ Бесконечная лента с символами из Σ и b
- ▶ Считывающая головка
- ▶ Внутренняя память Q
- ▶ Таблица переходов δ , которая по текущему состоянию из Q и текущему символу на ленте из Γ говорит машине, что делать:
 - ▶ перейти в состояние
 - ▶ записать символ на ленту
 - ▶ сместиться влево/вправо



Архитектура фон Неймана

- ▶ Принцип последовательного программного управления
- ▶ Принцип однородности памяти
- ▶ Принцип адресуемости памяти
- ▶ Принцип двоичного кодирования
- ▶ Принцип жесткости архитектуры



Структурное программирование

- ▶ Пришло на смену неструктурированному программированию в начале 70-х
 - ▶ FORTRAN — 1957 год, язык высокого уровня, но не структурный
- ▶ Любая программа может быть представлена как комбинация
 - ▶ последовательно исполняемых операторов
 - ▶ ветвлений
 - ▶ итераций
- ▶ Статья Дейкстры «Go To Statement Considered Harmful» (1968г)

Языки-представители

- ▶ Алгол
- ▶ Паскаль
- ▶ С
- ▶ Модула-2
- ▶ Ада

Подробнее: Ада

- ▶ Разработан в начале 80-х по заказу минобороны США
- ▶ Особенности:
 - ▶ Строгая типизация
 - ▶ Минимум автоматических преобразований типов
 - ▶ Встроенная поддержка параллелизма
- ▶ Реализация: GNAT (<https://www.adacore.com/community>)

```
with Ada.Text_IO;  
use Ada.Text_IO;
```

```
procedure Main is  
begin  
  Put_Line ("Hello World");  
end Main;
```

Ада, модульная система

```
package types is
```

```
  type Type_1 is private;
```

```
  type Type_2 is private;
```

```
  type Type_3 is private;
```

```
  procedure P(X: Type_1);
```

```
  ...
```

```
private
```

```
  procedure Q(Y: Type_1);
```

```
  type Type_1 is new Integer range 1 .. 1000;
```

```
  type Type_2 is array (Integer range 1 .. 1000) of Integer;
```

```
  type Type_3 is record
```

```
    A, B: Integer;
```

```
  end record;
```

```
end Types;
```


Ада, многопоточность и рандеву

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Main is
```

```
  task After is
```

```
    entry Go(Text: String);
```

```
  end After;
```

```
  task body After is
```

```
  begin
```

```
    accept Go(Text: String) do
```

```
      Put_Line("After: " & Text);
```

```
    end Go;
```

```
  end After;
```

```
begin
```

```
  Put_Line("Before");
```

```
  After.Go("Main");
```

```
end;
```

Ада, ограничения и контракты

```
type Not_Null is new Integer
```

```
  with Dynamic_Predicate => Not_Null /= 0;
```

```
type Even is new Integer
```

```
  with Dynamic_Predicate => Even mod 2 = 0;
```

```
function Divide (Left, Right : Float) return Float
```

```
  with Pre => Right /= 0.0,
```

```
    Post => Divide'Result * Right < Left + 0.0001
```

```
    and then Divide'Result * Right > Left - 0.0001;
```

Объектно-ориентированное программирование

- ▶ Первый ОО-язык — Симула-67, были и более ранние разработки
- ▶ Популярной методология стала только в середине 90-х
- ▶ Развитие связано с широким распространением графических интерфейсов и компьютерных игр

Основные концепции

- ▶ Программа представляет собой набор объектов
- ▶ Объекты взаимодействуют путём отправки сообщений по строго определённым интерфейсам
- ▶ Объекты имеют своё состояние и поведение
- ▶ Каждый объект является экземпляром некоего класса

Основные концепции (инкапсуляция)

- ▶ Инкапсуляция — сокрытие реализации от пользователя
- ▶ Пользователь может взаимодействовать с объектом только через интерфейс
- ▶ Позволяет менять реализацию объекта, не модифицируя код, который этот объект использует

Основные концепции (наследование)

- ▶ Наследование позволяет описать новый класс на основе существующего, наследуя его свойства и функциональность
- ▶ Наследование — отношение «является» между классами, с классом-наследником можно обращаться так же, как с классом-предком
 - ▶ Принцип подстановки Барбары Лисков

Основные концепции (полиморфизм)

- ▶ Полиморфизм — классы-потомки могут изменять реализацию методов класса-предка, сохраняя их сигнатуру
- ▶ Клиенты могут работать с объектами класса-родителя, но вызываться будут методы класса-потомка (позднее связывание)

Пример кода

```
class Animal
```

```
{
```

```
    public:
```

```
        Animal(const string& name) {
```

```
            this.name = name;
```

```
        }
```

```
        void rename(const string &newName) {
```

```
            name = newName;
```

```
        }
```

```
        virtual string talk() = 0;
```

```
    private:
```

```
        string name;
```

```
};
```


Пример кода (2)

```
class Cat : public Animal
{
    public:
        Cat(const string& name) : Animal(name) {}
        string talk() override { return "Meow!"; }
};
```

```
class Dog : public Animal
{
    public:
        Dog(const string& name) : Animal(name) {}
        string talk() override { return "Arf! Arf!"; }
};
```

Пример кода (3)

```
...  
Cat *cat1 = new Cat("Барсик");  
Animal *cat2 = new Cat("Шаверма");  
Dog *dog = new Dog("Бобик");  
  
std::vector<Animal *> animals{cat1, cat2, dog};  
  
for (Animal *animal : animals) {  
    std::cout << animal->talk();  
}  
...
```

Языки-представители

- ▶ Java
- ▶ C#
- ▶ C++
- ▶ Object Pascal / Delphi Language
- ▶ Smalltalk

Функциональное программирование

- ▶ Вычисления рассматриваются как вычисления значения функций в математическом понимании (без побочных эффектов)
- ▶ Основано на λ -исчислении

λ-исчисление

- ▶ λ-исчисление — формальный способ описать математические функции
 - ▶ $\lambda x. 2 * x + 1$ — функция $x \rightarrow 2 * x + 1$
- ▶ Функции могут принимать функции в качестве параметров и возвращать функции в качестве результата
- ▶ Функция от n переменных может быть представлена, как функция от одной переменной, возвращающая функцию от $n - 1$ переменной (карринг)
- ▶ Формальная система, не требующая математических оснований
 - ▶ На самом деле, математика может быть построена на λ-исчислении

Языки-представители

- ▶ Лисп (List Processing)
- ▶ ML (OCaml)
 - ▶ F#
- ▶ Haskell
- ▶ Erlang

Особенности

- ▶ Программы не имеют состояния и не имеют побочных эффектов
 - ▶ Нет переменных
 - ▶ Нет оператора присваивания
- ▶ Порядок вычислений не важен
- ▶ Циклы выражаются через рекурсию
- ▶ Ленивые вычисления
- ▶ Формальные преобразования программ по математическим законам

Пример на языке Haskell

Факториал:

```
fact :: Integer -> Integer
```

```
fact 0 = 1
```

```
fact n | n > 0 = n * fact (n - 1)
```

QSort:

```
sort [] = []
```

```
sort (pivot:rest) = sort [y | y <- rest, y < pivot]
```

```
    ++ [pivot]
```

```
    ++ sort [y | y <- rest, y >= pivot]
```


F#, мерджсорт

```
let rec merge l r =  
    match (l, r) with  
    | ([], r) -> r  
    | (l, []) -> l  
    | (x::xs, y::ys) -> if (x < y) then x::(merge xs r) else y::(merge l ys)
```

```
let rec mergesort l =  
    match l with  
    | [] -> []  
    | x::[] -> l  
    | _ ->  
        let (left, right) = List.splitAt (List.length l / 2) l  
        let ls = mergesort left  
        let rs = mergesort right  
        merge ls rs
```

F#, бесконечная последовательность простых чисел

```
let isPrime number =  
    seq {2 .. sqrt(double number)}  
    |> Seq.exists (fun x -> number % x = 0)  
    |> not
```

```
let primeNumbers =  
    Seq.initInfinite (fun i -> i + 2)  
    |> Seq.filter isPrime
```

Логическое программирование

- ▶ Программа представляет собой набор фактов и правил, система сама строит решение с использованием правил логики
 - ▶ Использует логику предикатов как математическую формализацию
- ▶ Создавалось в 60-х для решения задач искусственного интеллекта и экспертных систем
 - ▶ Автоматическое доказательство теорем
- ▶ Могут использоваться разные стратегии доказательства
 - ▶ В общем случае, программа — это набор фактов и правил + стратегия вывода, которая управляет тем, как новые факты получаются из существующих
 - ▶ В формальной логике стратегия вывода обычно не важна, для компьютеров это критично
- ▶ Дедуктивные базы данных — хранят факты и правила вывода

Пролог

- ▶ Появился в 1972 г. как научная разработка
- ▶ Реализации:
 - ▶ SWI-Prolog (<http://www.swi-prolog.org/>)
 - ▶ Amzi Prolog (<http://www.amzi.com/>)
 - ▶ Turbo Prolog
- ▶ Использует метод резолюций – последовательно перебирая правила и факты, пытается подобрать такой набор переменных, которые бы им удовлетворяли
 - ▶ Пример:
 - ▶ `cat(tom)`
 - ▶ `?- cat(tom).`
Yes
 - ▶ `?- cat(X).`
X = tom

Пример программы

```
sibling(X, Y) :- parent_child(Z, X), parent_child(Z, Y).  
parent_child(X, Y) :- father_child(X, Y).  
parent_child(X, Y) :- mother_child(X, Y).  
mother_child(trude, sally).  
father_child(tom, sally).  
father_child(tom, erica).  
father_child(mike, tom).
```

```
?- sibling(sally, erica).
```

Yes

```
?- father_child(Father, Child).
```

Императивное программирование

```
?- write('Hello world!'), nl.
```

```
Hello world!
```

```
true.
```

```
program_optimized(Prog0, Prog) :-  
    optimization_pass_1(Prog0, Prog1),  
    optimization_pass_2(Prog1, Prog2),  
    optimization_pass_3(Prog2, Prog).
```

QSort

```
quicksort(Xs, Ys) :- quicksort_1(Xs, Ys, []).
```

```
quicksort_1([], Ys, Ys).
```

```
quicksort_1([X|Xs], Ys, Zs) :-  
    partition(Xs, X, Ms, Ns),  
    quicksort_1(Ns, Ws, Zs),  
    quicksort_1(Ms, Ys, [X|Ws]).
```

```
partition([K|L], X, M, [K|N]) :-
```

```
    X < K, !,
```

```
    partition(L, X, M, N).
```

```
partition([K|L], X, [K|M], N) :-
```

```
    partition(L, X, M, N).
```

```
partition([], _, [], []).
```

Рекурсивное программирование, РЕФАЛ

- ▶ РЕкурсивных Функций АЛгоритмический
 - ▶ В. Турчин, 1966г.
- ▶ Ориентирован на символьные вычисления
 - ▶ ИИ, перевод, манипуляции с формальными системами (лямбда-исчисление, например)
- ▶ Использует нормальные алгорифмы Маркова в качестве математической формализации
- ▶ Программа записывается в виде набора функций
 - ▶ Функция — упорядоченный набор предложений
 - ▶ Предложение состоит из шаблона и того, на что надо заменить шаблон
 - ▶ Выражения в угловых скобках (активные выражения)
 - ▶ Переменные
- ▶ Вычисление продолжается, пока в «поле зрения» Рефал-машины не окажется выражение без угловых скобок

Рефал, пример

Hello, world:

```
$ENTRY Go { = <Hello>;}  
Hello {  
  = <Prout 'Hello world'>;  
}
```

Палиндром:

```
Palindrom {  
  s.1 e.2 s.1 = <Palindrom e.2> ;  
  s.1 = True ;  
  = True;  
  e.1 = False ;  
}
```

Стековое программирование

- ▶ Язык Форт (Forth)
 - ▶ Разработан в 60-х Чарльзом Муром «для себя»
 - ▶ Был широко распространён для программирования встроенных систем и задач, естественным образом выражающихся в терминах стеков
 - ▶ Синтаксический анализ
 - ▶ Анализ естественных языков

Форт, подробнее

- ▶ Основной элемент программы: слово
- ▶ Форт-система состоит из словаря (набора слов) и стеков — арифметического и командного (с их помощью производятся вычисления)
- ▶ Используется обратная польская нотация

Примеры

▶ `25 10 * 50 + .`

Вывод: 300 ok

▶ `: FLOOR5 (n -- n') DUP 6 < IF DROP 5 ELSE 1 - THEN ;`

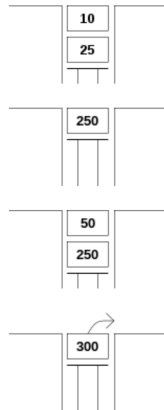
▶ то же самое на C:

```
int floor5(int v) { return v < 6 ? 5 : v - 1; }
```

▶ более красиво на Форте:

`: FLOOR5 (n -- n') 1- 5 MAX ;`

▶ `: HELLO (--) CR ." Hello, world!";`



Форт, пример

\ Напечатать знак числа

```
:.SIGN ( n -- )  
  ?DUP 0= IF  
    ." НОЛЬ"  
ELSE  
  0> IF  
    ." ПОЛОЖИТЕЛЬНОЕ ЧИСЛО"  
  ELSE  
    ." ОТРИЦАТЕЛЬНОЕ ЧИСЛО"  
THEN  
THEN  
;
```

Реализации

- ▶ SwiftForth
 - ▶ <https://www.forth.com/swiffforth/>
- ▶ Gforth
 - ▶ <http://www.gnu.org/software/gforth/>
- ▶ Десятки других реализаций
 - ▶ <http://www.forth.org/commercial.html>
- ▶ Книжка
 - ▶ Броуди Л. «Начальный курс программирования на Форте»

