В.А. Нестеренко

Языки и методы программирования

Курс лекций
для студентов механико-математического факультета
специальности “Прикладная математика”

Содержание.

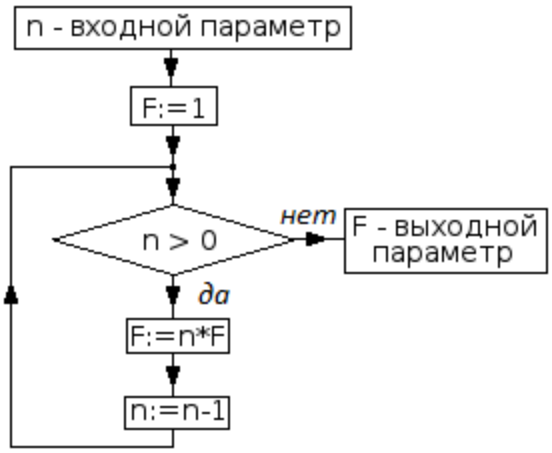
0. **Введение**
1. **Язык программирования как способ записи алгоритма**
2. **LISP - язык функционального программирования.**
3. **Функции высших порядков.**
4. **λ - исчисление.**
5. **Функциональное представление данных.**
6. **Логическое программирование.**
7. **Объектно-ориентированное программирование.**
8. **Квантовые вычисления.**
9. **Литература.**

1. **Язык программирования как способ записи алгоритма**

Представим общую схему использования компьютеров при решении реальных задач:

Вначале строим модель, описывающую объекты и процессы, имеющие отношения к решаемой задаче. На этом этапе происходит замена реальных объектов на математические (вектора, множества, функции, ...), а их взаимодействие представляется в посредством системы уравнений. На следующем этапе составляется компьютерная программа (на том или ином языке программирования) для решения системы уравнений математической модели. При этом, математические объекты представлены в виде программных объектов соответствующего типа данных, а метод решения полученной системы уравнений в виде алгоритма, реализованного в программе. Затем следует выполнение программы на компьютере, анализ полученных результатов и, при необходимости, уточнение или выбор модели и повторение перечисленных выше шагов. В представленной схеме ясно видна роль языка программирования: это способ записи алгоритма решения исходной задачи с целью использования компьютера. Ещё раз: **язык программирования это всего лишь способ записи алгоритма**.

Процедурное программирование. Язык программирования это не единственный способ записи алгоритма. Помимо языков программирования существуют и другие способы представления алгоритмов. Один из наиболее известных и широко используемых при анализе алгоритмов — блок-схема.




```
<ввод n>
F=1;
while (n > 0) {
  F = n*F;
  n = n-1;
}
<вывод F>
```

В качестве примера рассмотрим задачу вычисления факториала: $F = n! = 1*2*...*n$. Здесь приведены два способа записи алгоритма вычисления факториала: в виде блок-схемы и в виде фрагмента программы на языке программирования С. В обеих случаях алгоритм представлен в виде последовательности шагов, выполняя которые в заданном порядке мы получаем решение исходной задачи. Такой подход к программированию (представление алгоритма в виде последовательности шагов) называется процедурное программирование, соответствующие языки программирования называются

процедурными языками. К этим языкам относятся Pascal, C, Basic, Fortran, …

Функциональное программирование. Введём функцию $f(n)=n!$ и, воспользовавшись свойствами факториала, определим её следующим образом:

$$f(n)=\begin{cases}1, если n=0\\ n\cdot f(n-1)\end{cases}$$

Приведённое определение задаёт способ нахождения факториала, не сводящийся к выполнению заданной последовательности шагов, как это было в предыдущем примере. В данном случае алгоритм вычисления факториала определён в виде правила отображения множества значений аргумента n в множество значений функции $f(n)$. Этот подход к программированию называется функциональное программирование, соответствующие языки программирования Lisp, Haskell, ML — функциональные языки.

Логическое программирование. Кроме процедурного и функционального программирования существует ещё один способ записи алгоритма, основанный на определении того, что мы считаем правильным решением задачи. В случае факториала это будет выглядеть следующим образом: правильным значением $0!$ будет 1, правильным значением $n!$ для других n будет $n*f(n-1)$ при правильном значении $f(n-1)$. Подобный подход называется логическим программированием. На языке программирования Prolog подобное представление алгоритма будет выглядеть следующим образом:

```
Factorial(1, 0) :- .
Factorial(F, N) :- Factorial(F1, N1), N is N1+1, F is N*F1.
```

Процедурное, функциональное и логическое программирование — три разных подхода к программированию, основанные на разных способах представления алгоритма. В соответствии с этим, языки программирования, как способ записи алгоритма, можно разделить на три группы: процедурные, функциональные и логические языки программирования. В следующих лекциях более подробно познакомимся с процедурным, функциональным и логическим подходами к программированию.

2. LISP - язык функционального программирования.

LISP - язык программирования, разработанный Джоном Маккарти в начале 60-х годов. Первоначально, язык программирования LISP создавался как язык обработки списков (List Processor). Однако, впоследствии оказалось, что язык LISP является адекватным средством программной реализации функциональных алгоритмов, алгоритмов представленных в виде определения функции. Синтаксис языка LISP позволяет практически один-в-один переводить функциональную нотацию в программу:

```
              | x  если  x > y
max(x,y) =    |
              | y

(defun max (x y) (if (> x y) x y))
```

Язык программирования LISP содержит большой набор предопределенных функций и предоставляет средства определения новых. Для компоновки законченной программы из набора вспомогательных функций используется суперпозиция (вложенность) функций с возможностью рекурсивного вызова. В LISP'е имеются средства позволяющие управлять процессом вычисления функций (отмена и принудительные вычисления, отложенные вычисления и т.п.). Выбранный способ представления функции (в виде списка) позволяет реализовывать не только обычные функции но и функционалы: функции которые в качестве своего аргумента и (или) возвращаемого значения используют другие функции.

Согласно определению А.П.Ершова: "Функциональное программирование - это способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значение функции, а единственным правилом композиции - оператор суперпозиции функции. Никаких ячеек памяти, ни операторов присваивания, ни циклов, ни, тем более, блок-схем, ни передач управления." - язык LISP можно считать функциональным языком программирования.

В настоящее время существует множество диалектов языка LISP: FreeLisp, XLisp, muLisp, CommonLisp и т. д.. Некоторых из них содержат элементы нетипичные для функционального подхода к программированию (блок, цикл и т.п.). Появление подобных конструкций скорее дань традиции, чем насущная необходимость. В предлагаемой лекции мы рассмотрим лишь те средства языка, которые являются необходимыми для реализации функциональных алгоритмов.

Атомы и списки. Язык программирования LISP оперирует с двумя основными структурами данных - атомами и списками. Атомы и списки не являются типами данных в том смысле, как это понимается в процедурных языках программирования (C, PASCAL, ...). В языке LISP атомы и списки - структуры данных предоставляющие способ представления одних объектов через совокупность других (обычно более простых), и ничего более. В языке нет ограничений на множество значений принимаемых атомами или списками.

Атом - элементарный объект, не имеющий структуры, он не может быть разбит на составные части. В программе атомы могут обозначать константы и функции, атомы могут обозначать самих себя - те последовательности символов, которыми атомы изображаются. Последнее обстоятельство позволяет использовать язык LISP в качестве базового языка при организации символьных и аналитических вычислений, так как дает возможность работать с объектами, которым не присвоено никакого конкретного значения. Не следует отождествлять атомы с переменными в процедурных языках программирования. Во-первых, переменная всегда должна принадлежать какому-либо определенному типу (вещественная переменная, символьная переменная, ...). Во-вторых, переменная всегда должна иметь определенное значение заданного типа. В-третьих, переменная всегда отождествляется с некоторой ячейкой памяти определенного размера и фиксированной структурой данных хранящихся в ней. В отличии от переменной атом представляет собой способ маркировки любого объекта, допустимого в языке LISP.

Атомы в языке LISP могут быть свободными или связанными. Свободный атом обозначает сам себя - последовательность символов, образующих его имя. Здесь и далее условимся обозначать стрелкой "-->" значение любого допустимого в языке LISP выражения. Тогда

```
127 --> 127
TRUE --> TRUE
```

Связанный атом в качестве своего значения может иметь любое допустимое LISP-выражение: атом или список. Установить некоторое значение атома (связать его) или изменить значение, установленное ранее, можно с помощью операции присвоения, для обозначения которой используется функция SETQ. Например:

```
(setq A 276)
A --> 276

(setq BETA Z124)
BETA --> Z124

(setq D DELTA)
```

```
D --> DELTA

(setq X A)
X --> 276
```

В первом примере атом A приобретет числовое значение 276. В четвертом примере атом X получит значение 276, так как вместо атома A в момент подстановки будет использовано его значение.

В некоторых версиях языка LISP атомы изначально являются не связанными с определенным значением, даже с самим собой. В этом случае, если мы хотим в качестве значения атома установить его самого, можно воспользоваться присваиванием

```
(setq A (quote A)) или (setq A 'A)
```

В дальнейшем мы всегда будем считать, что свободный атом в качестве своего значения имеет сам себя или связан сам с собой.

Особую роль в языке играют атомы T и NIL. В логических выражениях они имеют значение "истина" и "ложь", соответственно. Кроме того, атом NIL эквивалентен пустому списку. Числовые атомы, имена которых состоят из последовательности цифр, в качестве своего значения имеют соответствующее число и это значение не может быть изменено.

Списки являются основной структурой данных языка LISP. При помощи списков можно представить практически составной объект: множество, тензор, граф, ... Функции также могут быть представлены в виде списка. Поэтому любая программа на языке LISP является какой-либо функцией, которая может иметь своим значением список представляющий другую функцию. Иными словами, исполнение программы на LISP'е может приводить к порождению новой программы.

Списком называется последовательность элементов, заключенных в скобки, элементом списка может быть либо атом, либо список. Или, используя формальное определение:

```
<список> ::= <пустой список> | (<элемент> <хвост списка>)
<элемент> ::= <атом> | <список>
<хвост списка> ::= <список>
```

В качестве разделителей элементов списка обычно используется пробел или запятая. Примеры списков:

```
(A B C D E)
(1 2 3 4 5 6)
(1 (2 3) (4 (5 6)))
(TEST)
```

```
()
```

Второй и третий примеры - разные списки, т.к. они состоят из разных элементов. Первый из этих списков построен только из атомов, второй - из атома и двух списков. Четвертый пример - список из одного элемента — атома, атом TEST и список (TEST) - разные объекты. Последний пример - пустой список, он не содержит каких-либо элементов. Пустой список эквивалентен атому NIL.

Кроме представления различных объектов, списки используются для вызова функций. В этом случае, интерпретатор языка LISP воспринимает первый элемент списка как имя функции, а остальные элементы списка - как ее аргументы. Например:

```
(+ 2 2) --> 4
(* (+ 3 2) (- 3 2)) --> 5
```

здесь использованы функции арифметических операций сложения, умножения и вычитания.

Функции для работы со списками. Поскольку списки являются основными структурами данных в языке LISP, постольку основными функциями языка следует считать функции для работы со списками. Первая из них функция list - предназначена для построения списка. Функция list имеет прозвольное число аргументов и в качестве своего значения возвращает список, составленный из значений своих аргументов:

```
(list 1 2 3) --> (1 2 3)
(setq A 1)
(setq B 2)
(setq C 3)
(list A B C) --> (1 2 3)
(list (list 1 2) (list 3 4)) --> ((1 2) (3 4))
```

В последнем случае список состоит из двух элементов, которые в свою очередь являются списками: (1 2) и (3 4).

Далее, для выделения элементов списка предназначены функции car и cdr. В качестве своего аргумента эти функции должны получать список, а возвращаемыми значениями будут первый элемент списка и остаток списка соответственно (см. формальное определение списка приведенное ранее). Например:

```
(car (list 1 2 3)) --> 1
(cdr (list 1 2 3)) --> (2 3)
(setq L (list (list 1 2) (list 3 4)))
```

```
L --> ((1 2) (3 4))
(car L) --> (1 2)   - первый элемент списка L
(cdr L) --> ((3 4)) - остаток после удаления первого элемента
```

Для выделения произвольного элемента списка используется суперпозиция функций car и cdr:

```
(setq L (list 1 2 3 4))
L --> (1 2 3 4)
(car L) --> 1
(car (cdr L)) --> 2
(car (cdr (cdr L))) --> 3
```

Поскольку различные комбинации car и cdr используются довольно часто, то в языке LISP имеется возможность порождать специальные имена функций, эквивалентных комбинациям car и cdr. Имена таких функций начинаются с символа "c", затем произвольная комбинация символов "a" и "d" и завершающий символ "r". Каждый символ "a" в такой комбинации имени функции соответствует вызову функции car, а символ "d" - cdr. То есть,

(cddar X) эквивалентно (cdr (cdr (car X)))

Для выделения второго элемента списка можно использовать функцию cadr, третьего - caddr, четвертого - cadddr и т.д..

Функция cons служит для добавления в список нового элемента. Она имеет два аргумента, второй аргумент обычно является списком, а первый - добавляется в этот список в качестве головного элемента и результат возвращается в качестве значения функции. Другими словами, функция cons собирает вместе то, что функции car и cdr разбили на части. Если L - произвольный список, то

```
(cons (car L) (cdr L)) --> L
```

Еще примеры:

```
(cons 1 (list a b c)) --> (1 a b c)
(cons (list 1) (list a b c)) --> ((1) a b c)
```

Для объединения двух списков предназначена функция append, она имеет два аргумента - оба должны быть списками, в результате своего действия она соединяет эти два списка в один:

```
(append (list 1 2 3) (list a b c)) --> (1 2 3 a b c)
```

Следует различать действия функций cons и append, так например:

```
(append (list a b) (list c d)) --> (a b c d)
```

но

```
(cons (list a b) (list c d)) --> ((a b) c d)
```

Это два разных списка, первый из них - список из четырех элементов - (a b c d), второй - список из трех элементов - ((a b) c d), причем первый элемент сам является списком - (a b).

Кроме приведенных функций: list, car, cdr, cons, append; в языке LISP имеются другие функций для обработки списков: функция reverse - инвертирующая список

```
(reverse (list 1 2 3)) --> (3 2 1)
```

функция length - определяющая количество элементов в списке

```
(length (list a b c)) --> 3
```

и так далее. Однако, большинство подобных функций может быть определено через основные - list, car, cdr, cons и append.

Другие функции. Функция setq применяется для связывания атома, она имеет два аргумента: первый - имя атома, второй - произвольное выражение. В результате выполнения функции setq указанный атом принимает значение равное вычисленному значению второго аргумента:

```
(setq A 123)
A --> 123
(setq B (list A 7))
B --> (123 7)
```

В качестве своего значения функция setq возвращает значения второго аргумента. Следовательно, возможно "ступенчатое" использование функции:

```
(setq X (cons C (setq Y (list D E))))
Y --> (D E)
X --> (C D E)
```

Действие функции setq не совсем такое, какое ожидается от "нормальной" функции - вычисление значения функции в соответствии со значениями ее аргументов. Для setq основным результатом является не возвращаемое значения, а побочный эффект - связывание атома. Другой пример побочного эффекта: функция print своим значением имеет значение своего единственного аргумента, ее побочный эффект - полученное значение направляется на стандартное устройство вывода. С подобными функциями всегда следует

обращаться осторожно, т.к. побочные эффекты могут привести к непредсказуемым результатам и нарушают строго функциональный подход, они могут свести на нет основное достоинство функционального подхода к программированию - модульность программ.

В языке LISP реализованы основные арифметические операции. Внешнее их представление близко к математической записи и не требует подробных комментариев. Символы +, -, *, / обозначают операции сложения, вычитания, умножения и деления соответственно.

(+ 2 3) --> 5

(* (+ (- 5 1) 7) 2) --> 18

Функции, вырабатывающие логические значения, называются предикатами. Атом T используется как стандартное обозначение логического значения "истина", атом NIL - "ложь". Одним из примеров предиката является часто используемая функция atom. Её аргументом может быть любое выражение. Функция возвращает значение T, если значением аргумента является атом, и NIL в противном случае. Например:

```
(atom 734) --> T
(atom (list 1 2 3)) --> NIL
```

Функция null проверяет: является список пустым или нет:

```
(null (list 1 2 3)) --> NIL
(null (cdr (list 1))) --> T
```

Функции not, and и or выполняют логические операции отрицания, умножения и сложения:

```
(not T) --> NIL
(and T NIL) --> NIL
(or T NIL) --> T
```

Для оптимизации процесса вычислений функции and и or иногда реализуются как "ленивые" функции. Другими словами: если первый аргумент функции and имеет значение NIL, то второй аргумент не вычисляется; аналогично для функции or: если первый аргумент имеет значение T - не вычисляется второй аргумент. Это обстоятельство иногда может приводить к нежелательным последствиям. Например:

```
(and T (setq X 1))
(and NIL (setq Y 2))
```

В первом случае атом X получит значение 1; во втором случае атом Y не получит никакого значения, т.к. второй аргумент функции and может не вычис-

ляться. (Приведенный пример может служить иллюстрацией нежелательных последствий побочных эффектов вычисления функции).

В языке LISP для числовых атомов определены встроенные функции сравнения: "<" - меньше, ">" - больше, "=" - равно:

```
(> 7 5) --> T
(= 6 3) --> NIL
(setq THREE 3)
(< THREE 3) --> NIL
```

Для выбора одного из двух вариантов нахождения значения функции предназначена условная функция if. Эта функция имеет три аргумента, если значение первого аргумента "истина", то функцией возвращаеся значение второго аргумента, в противном случае — третьего:

(if T a1 a2) --> a1

(if NIL a1 a2) --> a2

Для функции fakt

```
           | 1 если N=0
fakt(N) = |
           | N*fakt(N-1)
```

можно записать следующее правило вычисления:

(if (= N 0) 1 (* N (fakt (- N 1))))

Если необходимо выбирать более чем из двух вариантов — можно воспользоваться условным выражением cond. Функция cond имеет произвольное число аргументов, каждый из которых должен быть списком из двух элементов. При вычислении функции cond LISP просматривает поочередно каждую пару начиная с первой, если значение первого элемента очередной пары "истина", то возвращается значение второго элемента пары и обработка функции cond прекращается, в противном случае происходит переход к следующей паре. Если ни одна пара не имеет значение "истина" для первого элемента, то значение функции cond неопределено. Во избежание этого в качестве первого элемента последней пары следует использовать атом T.

Например:

```
(cond ((= N 0) 1)
   (T (* N (fakt (- N 1)))))
```

Функция cond может быть представлена как суперпозиция нескольких функций if:

```
(cond (<t1> <e1>)
   (<t2> <e2>)
   ...
   (T <eN>) )
```

эквивалентно

```
(if <t1> <e1> (if <t2> <e2> (... <eN>) ... ))
```

Порядок вычислений в LISP'е. Любое допустимое LISP-выражение определённым образом обрабатывается интерпретатором языка LISP - "вычисляется". Вычисления проводится по-разному, в зависимости от того, какой объект обрабатывается - атом или список.

Вычисление атома заключается в использовании значения, установленного в процессе связывания атома. Значением атома может быть другой атом, список или функция. Вычисление атома производится независимо от того, одиночный это атом, является ли он элементом списка или аргументом функции (ранее указывалось, что два последних случая фактически совпадают). Наглядным примером служит последовательность присваиваний

```
(setq A B)
A --> B

(setq B C)
B --> C

(setq D B)
D --> C
```

в результате первого присваивания атом A получит значение B (мы считаем, что в этот момент атом B не связан с другим значением, его значением является он сам) и, аналогично, при втором присваивании атом B получает значение C. В третьем случае, при исполнении функции setq будет вычислено значение атома B (в данном случае C) и атом D будет связан с соответствующим значением. При последующих вычислениях атомов A и D будут использованы их значения B и C соответственно:

```
(setq E A)
E --> B

(setq F D)
F --> C
```

В результате проведенных операций атом A имеет своим значением B, а атом B, в свою очередь, C. При вычислении значения атома интерпретатор языка LISP использует лишь один уровень связи значений. В операции (setq X A) используется связь A-->B и игнорируется следующий уровень B-->C. Для принудительного вычисления различных уровней связи значений предназначена функция eval. При вызове функции eval (как и любой другой функции языка LISP) вычисляется значение ее аргумента, а значением функции eval будет значение от полученного результата. Иными словами, значением функции eval будет значение значения ее аргумента. Например:

```
(setq X Y)
X --> Y

(setq Y Z)
(eval X) --> Z

(setq Z T)
(eval (eval X)) --> T
```

Предотвратить или отменить вычисление выражения можно при помощи функции quote. (quote X) всегда имеет своим значением X, независимо от того, является атом X связанным или свободным. Если действительны присваивания для X, Y и Z в последних примерах, то функции quote будет действовать следующим образом:

```
X --> Y

(quote X) --> X

(eval X) --> Z

(eval (qoute X)) --> Y

(eval (eval (quote X) --> Z
```

Функцию отмены вычислений можно использовать для того, чтобы связанный атом сделать свободным:

```
(setq X (quote X))
X --> X
```

Ранее в качестве основных объектов языка LISP мы выделили атомы и списки. Функции мы не указали, что может показаться довольно странным для функционального языка программирования, если не учитывать одно обстоятельство: в языке LISP функции представлены в виде списков. Первый элемент списка - имя функции, следующие элементы - аргументы функции. Вычисление любого списка сводится к попытке интерпретации его первого

элемента как имени функции, а остальных элементов как аргументов этой функции. Попытка задать список непосредственно во входной строке языка окончится неудачей, так как интерпретатор воспримет список как функцию и попытается её вычислить. Ранее, для задания списка мы использовали функцию list, можно также использовать функцию quote, отменяющую интерпретацию списка. Однако результаты применения функций list и quote могут быть разными, если элементы из которых строится список являются связанными:

```
(setq A 1)

(setq B 2)

(setq C 3)

(list A B C) --> (1 2 3)

(quote A B C) --> (A B C)
```

Любая функция языка программирования LISP может использовать значение другой функции в качестве своего аргумента, т.е. допускается вложенность или суперпозиция функций. В этом случае интерпретатор языка придерживается "естественного" порядка вычислений: перед нахождением значения функции любого уровня вложенности вычисляются значения ее аргументов.

```
(cons (car (list 1 2)) (quote 2 3)) --> (1 2 3)
```

В приведенном примере перед нахождение значения функции cons вычисляются значения ее аргументов - функции car и quote; исполнение функции car, в свою очередь, приводит к предварительному вычислению функции list и так далее. Функции quote и eval позволяют управлять порядком вычислений: отменять вычисление выражения или проводить повторные вычисления соответственно.

Функция setq "неправильная" функция, так как при ее исполнении не вычисляется значение ее первого аргумента:

```
(setq X Y)
X --> Y

(setq X 1)
X --> 1
```

в последнем случае значение 1 принимает атом X, хотя и является связанным, если бы вычислялось его значение, то значение 1 принял бы атом Y.

"Правильная" функция set - имеет тот же побочный эффект что и setq, но вычисляет значения обоих своих аргументов:

```
(setq A B)
A --> B

(set A 1)
A --> B
B --> 1
```

Дело в том, что функция setq более короткая форма записи функции set с отменой вычисления первого аргумента.

```
(setq A B) <==> (set (quote A) B)
```

Посколько функция quote используется довольно часто, для нее часто используется более лаконичная форма записи:

```
(quote X) <==> 'X
```

Определение собственных функций. Как бы ни было велико количество предопределенных функций (в LISP'е их около сотни), рано или поздно возникает необходимость в ведении новых. Более того, программная реализация функционального алгоритма сводится к определению некоторой функции для которой входные данные играют роль аргументов, а возвращаемое значение - ответ решаемой задачи. Для определения новых функций в языке LISP предназначена функция defun, она имеет три аргумента: первый - имя определяемой функции, второй - список формальных параметров, третий - выражение определяющее значение функции. Своим значением defun возвращает имя определяемой функции:

```
(defun fakt (N)
   (if (= N 0) 1 (* N (fakt (- N 1))))) --> fakt
```

Возвращаемое значение функции defun не является главным в ее использовании. Основным является побочный эффект функции defun: она заносит во внутренний словарь имя новой функции и связывает его с правилом вычисления возвращаемого значения.

В качестве примера определим функцию atoms, подсчитывающую число атомов в своём аргументе. Функция будет работать следующим образом: Если аргументом является список U, то число атомов равно сумме числа атомов в (car U) и (cdr U). Так как число атомов в каждом из слагаемых меньше, чем в исходном списке, при каждом новом обращении мы будем обрабатывать всё более и более короткие списки и, наконец, придём к случаю,

когда аргумент либо пустой список, либо атом. Значением функции atoms при этом должно быть 0 или 1, соответственно.

```
(defun atoms (U)
   (cond
      ((null U) 0)
      ((atom U) 1)
      (T (+ (atoms (car U)) (atoms (cdr U))))
   )
)

(atoms '(A B C)) --> 3

(atoms '(17 217 (13 4) 22)) --> 5

(atoms '()) --> 0
```

Определим функцию append:

```
(defun append (L1 L2)
   (if (null L1) L2 (cons (car L1) (append (cdr L1) L2)))
)
```

Встроенная функция reverse инвертирует список. Напишем собственную функцию reverse, которая будет инвертировать список и все его подсписки любого уровня вложенности:

```
(defun reverse (L)
   (if (atom L) L
      (append (reverse (cdr L)) (list (reverse (car L))))
   )
)

(reverse1 '(A B C D E)) --> (E D C B A)

(reverse1 '(A B ((C D) E))) --> ((E (D C)) B A)
```

Приведенные примеры определения функций atoms, reverse и append демонстрируют типичный метод реализации функций обработки списков: при помощи car и cdr список разбивается на части и каждая из них обрабатывается по отдельности и полученные результаты объединяются в конечный ответ.

Рассмотрим еще один интересный метод часто используемый в функциональном программировании. Вернемся к определению функции reverse: отдельные элементы списка, являющегося конечным результатом, "накапливаются" на стеке при рекурсивном вызове функции. Когда мы добираемся до самого глубокого уровня вложенности функций, начинается возврат из по-

следовательности вызовов и элементы списка снимаются со стека и собираются в конечный результат (конкретные детали могут зависеть от реализации языка, но суть не изменится). Введем явно объект накапливающий отдельные элементы результирующего списка: определим функцию rev с двумя аргументами - списками, второй из них будет последовательно получать элементы первого и присоединять к себе в обратном порядке:

```
(defun rev (L1 L2)
   (if (null L1) L2 (rev (cdr L1) (cons (car L1) L2)))
)
```

Теперь можно определить функцию reverse1 аналогичную reverse:

```
(defun reverse1 (L) (rev L nil))
```

Такой метод носит название "метод накапливающего параметра".

Рассмотрим еще один пример - сортировки списка, составленного из числовых элементов. Вначале определим функцию insert, которая имеет два аргумента: числовой атом X и упорядоченный список L, а своим значением возвращает упорядоченный список построенный из исходных списка и числа:

```
(defun insert (X L)
   (cond
      ((null L) (list X))
      ((<= X (car L)) (cons X L))
      (T (cons (car L) (insert X (cdr L))))
   )
)
```

Теперь определим функцию sort, которая берет элементы неупорядоченного списка L1 и добавляет их в упорядоченный список L2:

```
(defun sort (L1 L2)
   (if (null L1) L2
      (sort (cdr L1) (insert (car L1) L2))
   )
)
```

и, наконец, определим функцию сортировки sort-ins; действие которой сводится к вызову функции sort с пустым списком в качестве второго аргумента:

```
(defun sort-ins (L) (sort L NIL))
```

Пример: символьное дифференцирование. Операция символьного дифференцирования может быть легко определена на языке программирования LISP т.к. нахождение производной сводится к последовательному применению конечного числа правил, например: производная от константы равна нулю, производная от суммы равна сумме производных и т.д..

Введем функцию двух аргументов DIF(f,x), первый аргумент - дифференцируемое выражение, второй - переменная, по которой берется производная. Дадим определение этой функции в математической нотации:

```
                | 0              если f = const
                | 1              если f = x
                | DIF(f1,x)+DIF(f2,x)       если f = f1+f2
                | DIF(f1,x)-DIF(f2,x)       если f = f1-f2
  DIF(f,x) =    | DIF(f1,x)*f2+f1*DIF(f2,x)     если f = f1*f2
                | DIF(f1,x)/f2-f1*DIF(f2,x)/f2/f2  если f = f1/f2
                | . . .
                | DIF(f,x) во всех остальных случаях
```

При желании определение функции DIF можно дополнить правилами дифференцирования элементарных функций sin, cos, exp и т.д., например:

```
  DIF(exp(f),x) = exp(f)*DIF(f,x)
```

Реализуем функцию DIF на языке LISP:

```
(defun DIF (F X)
 (cond
  ((eq F X) 1)
  ((atom F) 0)
  ((eq (car F) '+)(list '+ (DIF (cadr F) X) (DIF (caddr F) X)))
  ((eq (car F) '-)(list '- (DIF (cadr F) X) (DIF (caddr F) X)))
  ((eq (car F) '*)(list '+ (list '* (DIF (cadr F) X) (caddr F))
                           (list '* (cadr F) (DIF (caddr F) X))
                  )
   )
  ((eq (car F) '/)(list '- (list '/ (DIF (cadr F) X) (caddr F))
                           (list '/ (list '* (cadr F)
                                          (DIF (caddr F) X)
                                    )
                                    (list '* (caddr F)
                                          (caddr F)
                                    )
                           )
                  )
  )
  (T (list 'DIF F X))
 )
)
```

3. Функции высших порядков.

Функции, которые мы рассматривали до сих пор, получали в качестве своих аргументов некоторые выражения, интерпретируемые как данные, и в соответствии со значениями своих аргументов возвращали результат. Однако, некоторые функции могут иметь своим аргументом другую функцию, то есть, множеством значений аргумента будет не множество данных, а множество функций. В математике такие функции называются функционалы, а в функциональном программировании - функции высших порядков.

Одним из основных типов функционалов являются функции применяющие функциональный аргумент к его параметрам. Такие функционалы называются применяющими (аппликативными) функционалами. Применяющие функционалы позволяют интерпретировать данные как программу и выполнять ее для некоторых входных данных. Если обычная функции позволяет исполнять фиксированный алгоритм (реализованный посредством данной функции) для различных наборов входных данных, то применяющий функционал позволяет использовать различные алгоритмы в рамках одной программы.

В LISP'е имеются две стандартные функции - funcall и apply, которые используются как применяющие функционалы. funcall имеет нефиксированное число аргументов, первый из которых интерпретируется как некоторая функция, а остальные - ее аргументы; в качестве своего значения funcall возвращает результат вычисления функции от значений ее аргументов:

```
(funcall f a1 a2 ... aN) <==> f(a1,a2,...,aN)
```

Например, определим список OP:

```
(setq OP '(+ - * /))
OP --> (+ - * /)
```

тогда:

```
(funcall (car OP) 6 2) --> 8
(funcall (cadr OP) 6 2) --> 4
(funcall (caddr OP) 6 2) --> 12
(funcall (cadddr OP) 6 2) --> 3
```

Если при вызове funcall требуется непосредственно указать имя F какой-либо функции, то следует использовать выражение (function F) в качестве первого аргумента, функция function в качестве своего значения возвращает функциональный объект ассоциированный с F. Существует и часто используется короткая форма записи

(function F) <==> #'F

Например:

```
(funcall (function max) 7 4) --> 7
```

или

```
(funcall #'max 7 4) --> 7
```

Применяющий функционал apply во многом подобен funcall. Функция apply имеет два аргумента, первый из которых представляет функцию, применяемую к элементам списка составляющего второй аргумент apply.

```
(apply F (a1 a2 ... aN)) <==> F(a1,a2,...,aN)
```

Например:

```
(apply #'+ '(7 4)) --> 11
(apply #'eval '((+ 2 3))) --> 5
```

Существуют, как минимум, две причины для использования функций высших порядков в языке LISP. Первая - реализовать понятие функционала и вторая - предоставить возможность программирования стандартных способов реализации вычислительного процесса. Довольно часто, при определении различных функций используется одинаковая структура организации последовательности вычислений, разница проявляется лишь в конкретных деталях исполнения какого-либо шага. Использование функций высшего порядка позволяют записать такую последовательность в виде определения универсальной функции, а конкретные детали передавать ей через аргументы. Проведем аналогию со структурным программированием: представьте, что в языке PASCAL нет операторов цикла и каждый раз для проведения повторяющейся последовательности шагов вы должны использовать комбинацию условного оператора и оператора перехода, а затем вы получаете средство, позволяющее один раз определить подобную комбинацию IF и GOTO и в дальнейшем использовать ее как самостоятельный оператор передачи управления.

Рассмотрим определения нескольких функций. Функция SumAll находит сумму элементов списка составленного из числовых атомов:

```
(defun SumAll (L)
  (if (null L) 0 (+ (car L) (SumAll (cdr L))))
)

(SumAll '(1 2 3)) --> 6
```

Функция append объединяет два списка в один:

```
(defun append (L1 L2)
  (if (null L1) L2 (cons (car L1) (append (cdr L1) L2)))
)

(append '(a b c) '(1 2 3)) --> (a b c 1 2 3)
```

Функция CountAll подсчитывает количество элементов в списке:

```
(defun CountAll (L)
  (if (null L) 0 (+ 1 (CountAll (cdr L))))
)

(CountAll '((1 2 3) a (b c))) --> 3
```

Все перечисленные функции (SumAll, append, CountAll) имеют одинаковую структуру:

```
(defun F (L [<другие параметры>])
  (if (null L) X0 (OP2 (car L) (F (cdr L))))
)
```

где X0 - значение, которое принимает функция F в случае пустого списка L; OP2 - двуместная операция, определяющая конкретное действие над элементами списка. Таким образом, приведенные определения функций различаются лишь значениями X0 и OP2 и имеют одинаковый, универсальной способ обработки списка: список разбивается на две части (car и cdr) и некоторая двуместная операция применяется к голове списка и результату действия определяемой функции на хвост списка. Используя применяющий функционал мы можем определить функцию, реализующую такую структуру вычислений:

```
(defun reduce (OP2 L X0)
  (if (null L) X0 (funcall OP2 (car L)
                               (reduce OP2 (cdr L) S0)
  )               )
)
```

reduce - стандартное название для подобной функции в функциональных языках программирования. В разных диалектах языка LISP ее синтаксис может немного различаться, но смысл остается неизменным. В нашем определении функция reduce имеет три аргумента: OP2 - имя функции двух аргументов, L - некий список (e1 e2 ... eN) и начальное значение X0; значением функции reduce будет результат многократного применения OP2.

```
(reduce (function OP2) '(e1 e2 ... eN) X0)
  <==> OP2(e1,OP2(e2,...OP2(eN,X0)...))
```

Теперь, для вычисления суммы элементов числового списка вместо определения функции SumAll просто вызовем

```
(reduce #'+ '(1 2 3 4 5) 0) --> 15
```

или для конкатенации двух списков

```
(reduce #'cons '(1 2 3) '(a b c)) --> (1 2 3 a b c)
```

Если функция max возвращает больший из двух своих аргументов, то найти максимальный элемент произвольного числового списка можно следующим образом: пусть

```
L --> (7 4 15 21 4 6))
```

тогда

```
(reduce #'max (cdr L) (car L)) --> 21
```

Для подсчета числа элементов в списке определим вспомогательную функцию

```
(defun Add1 (x y) (+ 1 y))
```

и теперь, вместо CountAll можно использовать

```
(reduce #'Add1 '((1 2 3) a (b c)) 0) --> 3
```

Использование функций высших порядков способствует улучшению модульности программ, делает их более короткими и более понятными, облегчает процесс написания и отладки.

Lambda — выражение. Некоторым недостатком в использовании функций высших порядков является необходимость давать имена функциям, используемым в качестве аргументов. Например, для подсчета числа элементов в списке нам пришлось ввести вспомогательную функцию Add1 и маловероятно, что эта функция будет использована в программе более одного раза. При определении функции мы задаем некоторое правило соответствия значения функции со значениями ее аргументов и связываем имя функции с этим правилом. Lambda - выражение позволяет задавать правило вычисления не связанное с каким-либо именем. В языке программирования LISP lambda - выражение имеет вид

```
(lambda <список формальных параметров> <тело функции>)
```

Так например, для нахождения суммы квадратов любых двух чисел можно задать следующее правило вычисления:

```
(lambda (x y) (+ (* x x) (* y y)))
```

для функции Add1 соответствующее правило имеет вид

```
(lambda (x y) (+ 1 y))
```

Lambda - выражение может использовано везде, где допустимо использование имени функции. Например:

```
((lambda (x y) (+ (* x x) (* y y))) 2 3) --> 13
```

вместо (Fun 2 3) или

```
(reduce #'(lambda (x y) (+ 1 y)) '(1 (2 3) 7 8) 0) --> 4
```

вместо

```
(reduce #'Add1 '(1 (2 3) 7 8) 0) --> 4
```

Обычно lambda - выражение используется в качестве фактического параметра функции высшего порядка; в том случае, если определение функции и ее вычисление желательно делать в одном месте программы или при формировании функции как результата исполнения другой функции.

4. λ - исчисление

λ -исчисление это абстрактная модель вычислений, основанная на представлении алгоритма в виде математических функций, отображающих входные данные задачи в выходные. Было сформулировано Алонзо Чёрчем (Alonzo Church) в 1930 году как теория исчисления анонимных (безымянных) функций. λ -исчисление предоставляет формальный метод представления функций и множество правил вывода для синтаксического преобразования функций. λ -исчисление является теоретическим базисом для функционального программирования.

Синтаксис. λ -нотация используется для представления функциональных выражений. Допустимое λ -выражение:

1. идентификатор (константа или переменная) $x, y, z, \ldots$
2. если x - переменная и M - λ -выражение, то $(\lambda x.M)$ тоже λ -выражение, называемое абстракцией.
3. если M и N - λ -выражения, то (MN) будет λ -выражением, называемым применением.

Абстракция $(\lambda x.M)$ предоставляет функцию, где x - аргумент и M - тело функции. Если M это функция и N её аргументы, то пара (MN) представляет применение M к N .

Для упрощения выражений обычно опускают скобки в тех случаях, когда это не приводит к неоднозначночти. При этом, применения ассоциативны слева: MNP обозначает $((MN)P)$, а абстракции ассоциативны справа: $\lambda x.\lambda y.M$ обозначает $(\lambda x.(\lambda y.M))$ (или ещё короче - $\lambda xy.M$).

Простые примеры λ -выражений:

- x
- $\lambda x.x$
- $\lambda x.\lambda y.x$
- $\lambda x.\lambda y.y$
- $\lambda x.\lambda y.xy$
- $\lambda x.xx$
- $\lambda x.y$
- $\lambda x.yx$
- $\lambda xyz.xz(yz)$

Редукция (Вычисление λ -выражений). Правила вывода задают порядок нахождения конечного значения λ -выражения из его первоначального вида.

1. Простейшим типом λ -выражения является константа. Константа представляет сама себя и её невозможно преобразовать в более простое выражение.
2. δ -редукция. Функция, представленная константой, преобразуется (редуцируется), при наличии достаточного числа аргументов, путём использования встроенных правил. Например, выражение $+12$ означает применение константы (встроенной функции $+$) к аргументам 1 и 2. В общем случае, аргументы функции могут не быть в нужной форме и редукция применяется в несколько этапов, вначале к аргументам функции: $*(+21)(-21) \rightarrow *31 \rightarrow 3$.
3. β -редукция. Следующее правило редукции описывает применение λ -абстракций: $(\lambda x.*xx)2$. Этот случай аналогичен обычному вызову функции с заменой формального параметра его фактическим значением. В λ -исчислении такая замена сводится к текстовой замене вхождений связанной переменной x в теле применяемой λ -абстракции на выражение аргумента: $(\lambda x.*xx)2 \rightarrow *22$ с последующим применением правила редукции для константных функций.

Приведённые правила редукции следует применять столько раз, сколько это возможно, с учётом ассоциативных правил: применения ассоциативны слева, абстракции ассоциативны справа.

Например:

$((\lambda x.\lambda y.+xy)7)8$

$\rightarrow(\lambda y.+7y)8$

$\rightarrow+78$

$\rightarrow 15$

Другой пример (Y-комбинатор):

$Y=\lambda h.(\lambda x.h(xx))(\lambda x.h(xx))$

Рассмотрим применение этого λ -выражения:

Yf

$\rightarrow(\lambda h.(\lambda x.h(xx))(\lambda x.h(xx)))f$

$\rightarrow(\lambda x.f(xx))(\lambda x.f(xx))$

$\rightarrow f(\lambda x.f(xx))(\lambda x.f(xx))$

$\rightarrow f(Yf)$

Выражение Yf называется фиксированной точкой функции f, так как не изменяется при применении функции: $Yf = f(Yf)$. Y-комбинатор мы будем использовать при рассмотрении реализации рекурсии в λ -исчислении.

λ -выражение в языке LISP. В LISP'е λ - выражение записывается в следующем виде:

```
(lambda (x1 x2 ... xn) fn)
```

например:

```
(lambda (x y) (+ (* x x) (* y y))
(lambda (x y) (cons x (cons y nil)))
```

λ - выражение является определением вычисления функции в «чистом» виде без фактических параметров. Для того, чтобы применить такую функцию к некоторым аргументам, нужно в вызове функции поместить λ - выражение на место имени функции:

```
((lambda (x y) (+ (* x x) (* y y)) 1 2)
-> 5

((lambda (x y) (cons x (cons y nil))) 'a 'b)
-> (a b)
```

В соответствии с основными понятиями λ -исчисления λ -выражения можно объединять между собой. Вложенные λ -вызовы можно использовать как в теле λ -выражения, так и в качестве фактических параметров:

```
((lambda (y)
     ((lambda (x) (list y x)) 'a)
 ) 'b)
-> (b a)
```

Рекурсия и λ -функция. Так как в λ -исчислении функции не имеют имени, то организация рекурсивных вычислений имеет некоторые проблемы. Выход состоит в том, чтобы представить рекурсивную функцию использующую саму себя в качестве аргумента.

Рассмотрим рекурсивную функцию sum:

```
(defun sum (n)
   (if (eq n 0) 0 (+ n (sum (- n 1))))
)
```

Перепишем определение sum в виде:

```
(defun sum (f n)
 (if (eq n 0) 0
 (+ n (funcall f f (- n 1)
)  )
```

Мы избавились от использования имени sum в теле функции. Использовать новое определение можно следующим образом:

```
(sum 'sum 4) -> 10
```

Теперь используем λ -функцию и избавимся от имени функции вообще:

```
(  (lambda (f n)
      (if (eq n 0) 0
         (+ n (funcall f f (- n 1)))
      )
   )                ; абстракция
   '(lambda (f n)
      (if (eq n 0) 0
         (+ n (funcall f f (- n 1)))
      )
   )                ; 1-й аргумент применения
   5                ; 2-й аргумент применения
)
-> 15
```

или без дублирования определения функции:

```
(  (lambda (n)
   ((lambda (h) (funcall h h n)
   )                ; абстракция
   '(lambda (f n)
    (if (eq n 0) 0
       (+ n (funcall f f (- n 1)))
    )
   )                ; 1-й аргумент применения
   5                ; 2-й аргумент применения
)
-> 15
```

Общее утверждение: Если E некоторая λ -абстракции, то λ -выражение $Y(\lambda f.E)$ представляет рекурсивное функцию с правилами вычислений задаваемыми E .

Действительно, пусть

```
E = λf.λn.(if (eq n 0) 0 (+ 1 (f (- n 1))))
```

теперь, если **sum = YE** то

```
sum n
-> YE n
-> E (YE) n
-> λf.λn.(if (eq n 0) 0 (+ 1 (f (- n 1))))
   (YE) n
-> (if (eq n 0) 0 (+ 1 (YE (- n 1))))
-> (if (eq n 0) 0 (+ 1 (sum (- n 1))))
```

Используя явное выражение для Y-комбинатора

```
Y = λh.(λx.h(x x))(λx.h(x x))
```

и λ-абстракцию E в результате можем получить последовательное, рекурсивное применение E.

5. Функциональное представление данных

При решении конкретной задачи всегда приходится выбирать некоторое математическое представление объектов участвующих в исследуемом процессе. В языке LISP для этих целей обычно используются списки и обработка различных данных сводится к программам обработки списков. Однако, в функциональном программировании для представления объектов можно использовать функции, соответствующие основным свойствам объектов и допустимым операциям над ними.

Функциональное представление множеств. Если множество это совокупность объектов без конкретизации деталей вхождения этих объектов (количество вхождений, порядок следования, …), то основной операцией для множества будет операция проверки вхождения элемента в множество. Ранее мы определили функцию $in(S\ x)$, которая возвращает значение «истина» или «ложь» в зависимости от того, содержится элемент x в множестве S или нет. Все другие операции над множествами мы определяли через функцию $in(S\ x)$.

Сопоставим множеству S функцию $S(x)$ и определим, что функция $S(x)$ возвращает значение «истина» если объект x принадлежит множеству и «ложь» в противном случае. И в дальнейшем вместо функции $in(S\ x)$ будем использовать функцию $S(x)$. Таким образом мы приходим к функциональному представлению множества: в программах вместо множества как структуры данных S мы будем использовать функцию $S(x)$.

При решении конкретных задач нет необходимости в перечислении всех элементов множества, достаточным знать: принадлежит тот или иной объект множеству или нет. В этом случае множество может быть адекватно представлено в виде функции одного аргумента. Эта функция должна принимать значение "истина", если ее аргумент принадлежит множеству и значение "ложь" в противном случае. Понятно, что разным множествам будут соответствовать разные функции и при изменении множества (добавление или удаление элементов) соответствующая функция должна меняться.

Начнём с определения пустого множества:

```
S0 = λx.nil
```

Применение функции $S0$ к любому аргументу даст значение nil:

```
(S0 a) -> nil
```

Теперь определим функцию *AddElem*, которая позволяет добавить новый элемент *e* в множество *S*:

```
AddElem = λS.λe.(if (S e) S (λx.(if (eq x e) T (S x))))
```

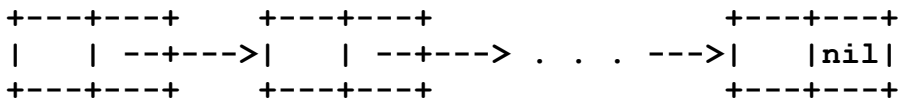
При помощи функции *AddElem* определим множество *S1*, состоящее из элементов *a*, *b* и *c*:

```
S1 = (AddElem (AddElem (AddElem S0 a) b) c)
```

Операции объединения *Union* и *Intersect* пересечения множеств можно определить следующим образом:

```
Union = λS1.λS2.λx.(or (S1 x) (S2 x))
Intersect = λS1.λS2.λx.(and (S1 x) (S2 x))
```

Функциональное представление списков. В LISP'e (где списки являются основной формой представления данных) и других языках программирования для реализации списков обычно используется динамическая структура, отдельные элементы которой связаны между собой посредством ссылок. Такое представление используется настолько часто, что почти стало синонимом списка. Каждый раз, когда встречается слово "список", возникает картинка более или менее похожая на



В связи с этим напомним, что список, по определению, может быть пустым или состоять из "головы" и "хвоста", "хвост" тоже является списком. Любой объект, соответствующий приведенному определению, будет адекватным представлением списка.

Введем функцию одного аргумента и будем считать, что при значении аргумента *CAR* значением функции будет голова списка, а при значении *CDR* - хвост списка. Такая функция может быть использована в качестве представления списка. Любому списку будет соответствовать своя функция и при изменении списка соответствующая функция будет меняться. Вначале введем функцию соответствующую пустому списку:

```
Empty = λS.nil

(Empty CAR) -> nil
(Empty CDR) -> nil
```

Затем, зададим функциональный аналог функции-конструктора *Cons* , которая создаёт список из его "головы" и "хвоста":

```
ConsList = λh.λt.(lambda (x) (if (equal x 'CAR) h t)))
```

Функция *Cons* позволяет конструировать различные списки, например :

```
Ln = (Cons 1 (Cons 2 (Cons 3 EMPTY)))
La = (Cons A (Cons B (Cons C EMPTY)))
```

Вызывая функции *Ln* и *La* с различными комбинациями аргументов *CAR* и *CDR* мы можем получить доступ к различным элементам списка:

```
(Ln CAR) -> 1
((Ln CDR) CAR) -> 2
(((Ln CDR) CDR) CAR) -> 3
```

Можно определить и другие функции для работы со списками, подобно функции *Cons* это будут функции высших порядков, так как в качестве аргумента они должны получать список представленный в виде функции и (или) своим значением могут иметь соответствующую функцию. Функция *Append* осуществляет конкатенацию двух списков :

```
Append = λl1.λl2.
  (if (Null l1) l2 (Cons (l1 CAR) (Append (l1 CDR) l2)) )
```

Здесь функция *Null* проверяет, является список пустым или нет :

```
Null = λl.(and (not (l CAR)) (not (l CDR)))
```

Также можно определить функцию инверсии списка

```
Reverse = λl1.λl2.
  (if (Null l1) l2 (Reverse (l1 CDR) (Cons (l1 CAR) l2)) )
```

и многие другие функции.

Функциональное представление булевых констант. В виде функций можно представить не только структуры данных подобные множествам и спискам, но и скалярные объекты подобные былевым константам, натуральным числам и т.п. .

Начнём с рассмотрения встроенной функции $if(P\ Q\ R)$ которая возвращает Q или R в зависимости от значения предиката P . В этом случае предикат условной функции можно рассматривать как селектор, который

выбирает одно из выражений Q или R. Выразим это явно, объединив в одной функции функцию if и предикат P. Так как предикат P может возвращать два значения, то в результате получаем две функции:

$if(TRUE\ P\ Q) \rightarrow True(P\ Q)$ и $if(FALSE\ P\ Q) \rightarrow False(P\ Q)$

В соответствии с определением if функции $True$ и $False$ возвращать следующие значения:

$True(P\ Q) \rightarrow P$ и $False(P\ Q) \rightarrow Q$

Таким образом мы приходим к функциональному представлению булевых констант $TRUE$ и $FALSE$.

Теперь необходимо определить функции предикаты так, чтобы вместо константы $TRUE$ или $FALSE$ они возвращали соответствующую функцию и тогда вместо функции if можно использовать непосредственно функцию предикат. Например:

```
(if (and A B) Q R) -> ((And A B) Q R)
```

В соответствии с указанными свойствами функций $True(P\ Q) \rightarrow P$ и $False(P\ Q) \rightarrow Q$ эти функции можно определить следующим образом:

```
True = λx.λy.x
False = λx.λy.y
```

Определения функций предикатов вытекает из их свойств:

```
Not = λx.(x False True)
Or = λx.λy.(x True y)
And = λx.λy.(x y False)
```

Убедиться в справедливости этих определений можно непосредственной проверкой результатов действия функций Not, Or и And для различных значений их аргументов. Так например:

```
Not True -> (λx.(x False True)) True
   -> (λx.(x False True)) True
   -> (True False True)
   -> ((λx.λy.x) False True)
   -> False

And True True -> (λx.λy.(x y False)) True True
   -> (True True False)
   -> ((λx.λy.x) True False)
   -> True

And False True -> (λx.λy.(x y False)) False True
   -> (False True False)
   -> ((λx.λy.y) True False)
```

```
-> False
```

и так далее.

Примеры применения:

```
(TRUE 1 2) -> 1
(FALSE 1 2) -> 2
((AND TRUE TRUE) 1 2) -> 1
```

Функциональное представление натуральных чисел. Если отвлечься от представления входных и выходных данных в программах, то с точки зрения реализации алгоритма решения задачи основная цель использования чисел это указать, сколько раз должна быть выполнена некоторая операция.

```
C0 = λs.λx.x
C1 = λs.λx.(s x)
C2 = λs.λx.(s(s x))
C3 = λs.λx.(s(s(s x)))
. . .
Cn = λs.λx.(s(. . .(s x) . . .))
```

Проверка равенства нулю:

```
IsZero = λn.n (True False) True

IsZero C0
-> (λn.n (True False) True) C0
-> C0 (True False) True
-> (λs.λx.x) (True False) True
-> True

IsZero C1
-> (λn.n (True False) True) C1
-> C1 (True False) True
-> (λs.λx.(s x)) (True False) True
-> (True False) True
-> (λx.λy.x False) True
-> True False
-> λx.λy.x False
-> False
```

Функция инкремент

```
Inc = λa.λb.λc.b(a b c)

Inc Cn
-> (λa.λb.λc.b(a b c)) Cn
-> λb.λc.b(Cn b c)
-> λb.λc.b((λs.λx.(s(. . .(s x) . . .))) b c)
-> λb.λc.b(b(. . .(b c) . . .)
```

```
   -> Cn+1
```

Функция вычисления суммы

```
   Add = λa.λb.λc.λd.(a c)(b c d)

   Add Cn Cm
   -> (λa.λb.λc.λd.(a c)(b c d)) Cn Cm
   -> λc.λd.(Cn c)(Cm c d)
   -> λc.λd.(λs.λx.(s(. . .(s x) . . .)) c)(Cm c d)
   -> λc.λd.(λx.(c(. . .(c x) . . .)))(Cm c d)
                  n-раз
   -> λc.λd.(c(. . .(c (Cm c d)) . . .))
              n-раз
   -> λc.λd.(c(. . .(c (λs.λx.(s(. . .(s x) . . .)) c
d)) . . .))
              n-раз              m-раз
   -> λc.λd.(c(. . .(c (c(. . .(c d) . . .)) ) . . .))
              n-раз      m-раз
   -> λc.λd.(c(. . .(c d) . . .))
            n+m-раз
   -> Cn+m
```

Обратите внимание на то, что при выводе результата действия функций *Not* , *Add* , *Inc* , *Add* , ... мы использовали формальные правила редукции λ-выражений и получали разумные результаты. Это дополнительно свидетельствует в пользу использования λ-исчисления в качестве модели вычислений.

6. Логическое программирование

В основе логического программирования лежит представление алгоритма в виде логического выражения. При этом алгоритм решения задачи описывает не способ решения задачи, а наше представление о том, что мы понимаем под правильным решением задачи. Пусть $Factorial(f, n)$ представляет решение задачи о нахождении факториала целого положительного числа n. В этом случае предикат $Factorial(f, n)$ принимает значение «истина» если f является факториалом n и «ложь» в противном случае. Например:

```
Factorial(120, 5) -> true
Factorial(17, 4) -> false
```

На языке программирования PROLOG определение факториала выглядит следующим образом:

```
Factorial(1, 0) :- .
Factorial(F, N) :- Factorial(F1, N1), N is N1+1, F is N*F1.
```

В данном случае мы определяем, что предикат $Factorial(f, n)$ должен иметь значение «истина» при значениях аргументов $f=1$ и $n=0$. При других значениях f и n функция $Factorial(f, n)$ примет значение «истина» лишь в том случае, если $Factorial(f1, n1)$ будет «истина» и одновременно будут справедливы соотношения $n=n1+1$ и $f=n*f1$. Таким образом наше понимание того, что будет правильным значением факториала мы определили через комбинацию некоторых утверждений - других предикатов.

Ещё один пример:

```
Sum(0, 0) :- .
Sum(S, N) :- Sum(S1, N1), N is N1+1, S is N*S1.
```

В дальнейшем мы покажем, что приведённое определение факториала и другие программы на языке программирования PROLOG могут быть представлены в виде выражений математической логики.

Факты, Правила и Вопросы. Написание и использование программ на языке PROLOG состоит из следующих этапов:

1. Объявление некоторых *фактов* об объектах и отношениях между ними.

   ```
   Factorial(1, 0) :- .
   Sum(0, 0) :- .
   ```

2. Определение некоторых *правил* об объектах и отношениях между ними. Правила определяются через факты и другие правила.

```
Factorial(F, N) :- Factorial(F1, N1), N is N1+1, F is N*F1.
Sum(S, N) :- Sum(S1, N1), N is N1+1, S is N*S1.
```

3. Формулировки *вопросов* об объектах и отношениях между ними.

```
?- Factorial(120, 5).
?- Sum(17, 5).
```

Факты — утверждения которые всегда истины.

Правила — утверждения, истинность которых зависит от истинности других утверждений.

Вопросы — формирование запросов системе PROLOG относительно истинности утверждений.

Работу в системе PROLOG можно понимать следующим образом: факты и правила формируют некоторую базу, а вопрос приводит к поиску в этой базе информации относительно истинности утверждений. Так запрос об истинности утверждения

```
?- Factorial(120, 5).
true
```

не противоречит введённым ранее фактам и правилам и приводит к соответствующему ответу системы. Напротив, запрос

```
?- Factorial(10, 5).
false
```

не соответствует имеющимся сведениям о функции факториал.

Если в вопросе вместо конкретного значения аргумента мы укажем имя переменной, то система попытается подобрать такое значение переменной, чтобы утверждение в запросе было истинным:

```
?- Factorial(X, 5).
X = 120
```

Так как в утверждениях и правилах нет явной разницы между входными и выходными данными задачи, то мы можем также сформулировать вопрос — какое число имеет значение факториала равное 120:

```
?- Factorial(120, X).
X = 5
```

и получить соответствующий результат, или не получить:

```
?- Factorial(100, X).
false
```

Таким образом, мы имеем возможность находить решение как прямой (по входным данным получать выходные) так и обратной задачи (по известным выходным данным находить соответствующие значения входных данных).

Более того, в ответ на вопрос

```
?- Factorial(X, Y).
X = 1, Y = 0
X = 1, Y = 1
X = 2, Y = 2
X = 6, Y = 3
X = 24, Y = 4
. . .
```

система попытается найти все возможные пары значений переменных X и Y обращающих в «истину» предикат $Factorial(X, Y)$.

Пример: Ханойские башни. Стержни:

- исходный — левый
- итоговый — центральный
- вспомогательный (запасной) — правый

Условие: Требуется переместить все диски с исходного стержня на итоговый, вспомогательный стержень можно использовать для временного хранения дисков. За один ход можно перемещать один верхний диск, нельзя ставить диск на диск меньшего размера.

Решение:

- Задача решена в том случае, если на левом стержне нет дисков.
- Переместить N-1 дисков с исходного стержня на вспомогательный, используя итоговый как запасной.
- Переместить один диск с исходного стержня на итоговый.
- Переместить N-1 дисков со вспомогательного стержня на итоговый, используя исходный как запасной.

```
hanoi(N) :- carry(N, left, middle, right).
carry(0, _, _, _) :- !.
carry(N, A, B, C) :- M is N-1,
                     carry(M, A, C, B),
                     output(A, B),
                     carry(M, C, B, A).
output(X, Y) :- write("переместили с ", X, " на ", Y).
```

Списки. В системе PROLOG списки синтаксически представлены перечислением элементов заключённых в угловые скобки: [**a,b,c**] . Как обычно список состоит из «головы» и «хвоста». Для представления «головы» и «хвоста» в PROLOG'е используется форма [**X|Y**] , здесь **X** обозначает «голову» списка, а **Y** - «хвост».

Введём функцию *belong*(*X*,*Y*) , которая определяет принадлежность элемента *X* списку *Y*:

```
belong(X, [X|_]) :- .
belong(X, [_| Y]) :- belong(X, Y).

?- belong(b, [a, b, c, d, e]).
true
```

Проверка того, что какой-то объект является списком:

```
list([]) :- .
list([_ | _]) :- .
```

Сортировка списка:

```
sort([], []) :- .
sort([X|L], M) :- sort(L, N), insert(X, N, M).

insert(X, [A|L], [A|M]) :- A < X, insert(X, L, M) !.
insert(X, L, [X|L]) :- .
```

PROLOG и математическая логика.

Так как программа на языке программирования PROLOG сводится к формированию правил относительно истинности некоторых утверждений, то теоретическим базисом логического программирования является математическая логика. Так например программа нахождения факториала

```
Factorial(1, 0) :- .
Factorial(F, N) :- Factorial(F1, N1), N is N1+1, F is N*F1.
```

может быть представлена в виде следующего логического выражения:

```
P(F,N) =
Fact(1,0) &
((Fact(F1,N1) & (N is N1+1) & (F is N*F1)) -> Fact(F,N))
```

Можно провести и обратные преобразования, для случая нахождения максимума двух чисел получаем:

```
P(X,Y) =
```

```
max(X,X,X) & (X>Y -> max(X,Y,X)) & (X<Y -> max(X,Y,Y))
```

или соответствующая программа:

```
max(X,X,X) :- .
max(X,Y,X) :- X > Y.
max(X,Y,Y) :- X < Y.
```

7. Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) представляет собой технологию программирования, основанную на идее классификации и абстрагирования объектов. В основе ООП программирования лежит понятие «класс объектов», абстрагирующее от особенностей отдельного объекта и представляющее характерные черты и свойства некоторой группы объектов.

С точки зрения программной реализации в языках программирования появляется новая конструкция - «класс», являющаяся обобщением типа данных. В отличии от типа данных «класс» описывает не только множество возможных значений объекта, но и набор допустимых операций для представителей данного класса.

Сравнение ООП и обычного подхода. Предположим, что в рамках большого проекта перед нами стоит задача программной реализации стека и наша реализация будет использоваться в других частях проекта. Мы решаем, что для хранения данных будем использовать массив, а для организации доступа к элементам стека введём две стандартные функции: *Push* - запись элемента в стек и *Pop* - извлечение элемента из стека.

Введём массив:

```
int S[100];
```

и указатель вершины стека:

```
int Point;
```

Отметим, что *S*[100] и *Point* должны быть глобальными переменными, так как к ним необходим доступ из других частей проекта.

Инициализация стека заключается в установке начального значения указателя:

```
Point = -1;
```

и это значение указателя соответствует пустому стеку (если необходима проверка).

Для реализации операции записи элемента в стек определим функцию *Push* :

```
void Push (int x) {
   S[++Point] = x;
}
```

и функцию *Pop* для извлечения данных из стека:

```
int Pop () {
   return S[Point--];
}
```

Подобная реализация имеет ряд очевидных недостатков:

- Использование глобальных переменных. В этом случае при разработке большого проекта трудно избежать конфликта имён.
- Если требуется использовать ещё один стек, то необходимо вводить новые глобальные переменные *S1*[100] и *Point1* и определять новые функции для работы с элементами стека (массив *S*[100] и указатель *Point* можно передавать функциям *Push* и *Pop* в качестве аргументов, но это может привести к излишнему усложнению при реализации проекта). В любом случае требуется участие автора реализации стека в разработке других частей проекта, что плохо согласуется с идеологией структурного программирования.
- Возможен непосредственный доступ к переменным *S*[100] и *Point* и произвольное изменение значений этих переменных приводит к разрушению стека и, в общем случае, к снижению надёжности проекта.

Теперь посмотрим, как выглядит реализация стека при использовании методов и средств объектно-ориентированного программирования.

```
class STACK {
   int S[100];
   int Point;
public:
   STACK ()     { Point = -1; }
   void Push (int x){ S[++Point] = x; }
   int Pop (void) { return S[Point--]; }
};
```

Здесь мы использовали новую конструкцию *class*, которая является основным средством реализации ООП в программах.

Пример программы:

```
void main ()
{
   STACK stack1, stack2;

   stack1.Push (17);
   stack1.Push (5);
   stack2.Push (stack1.Pop ());

   printf ("%d\n", stack1.Pop());
   printf ("%d\n", stack2.Pop());
```

```
}
```

Инкапсуляция. Объединение структур данных с функциями (методами), предназначенными для манипулирования этими данными. Инкапсуляция реализуется путём введения нового механизма типизации данных — класса.

Пример представления списка:

```
class ELEM {
   friend class LIST;
   int Inf;
   ELEM* Next;
   ELEM () { Inf = 0; Next = NULL; }
   ELEM (int I, ELEM* N) { Inf = I; Next = N; }
};

class LIST {
   ELEM* Head;
public:
   LIST () { Head = NULL; }
   int Car (void);
   LIST Cdr (void);
   LIST Cons (int);
   void Clear (void);
   void print ();
};

void LIST::Clear ()
{  ELEM *p;
   while (p=Head) {
      Head = p->Next;
      delete p;
   }
}

int LIST::Car ()
{  return Head->Inf;}

LIST LIST::Cdr ()
{  return (LIST&)Head->Next;  }

LIST LIST::Cons (int x)
{  Head = new ELEM (x, Head);
   return (LIST&)Head;
}

void LIST::print ()
{  ELEM *p = Head;
   printf ("List:");
   while (p) {
      printf (" %d", p->Inf);
      p = p->Next;
```

```
    }
    printf ("\n");
  }
```

Пример использования:

```
  void main ()
  {
    LIST L;

    L.Cons (1); L.Cons (2); L.Cons (3);
    L.print ();


    printf ("%d\n", L.Car ());
    L.Cdr ().print();

    L.Clear ();
    L.print ();
  }
```

Ещё один пример, комплексные числа:

```
  class COMPLEX {
    float re, im;
  public:
    COMPLEX ()   { re = 0.0; im = 0.0; }
    COMPLEX (float r, float i) { re = r; im = i; }
    void print ();
    friend COMPLEX operator + (COMPLEX, COMPLEX);
    friend COMPLEX operator * (COMPLEX, COMPLEX);
  };

  void COMPLEX::print() {
    printf ("(%5.2f, %5.2f)", re, im);
  }

  COMPLEX operator + (COMPLEX a, COMPLEX b) {
    return COMPLEX (a.re+b.re, a.im+b.im);
  }

  COMPLEX operator * (COMPLEX a, COMPLEX b) {
    return COMPLEX (a.re*b.re - a.im*b.im,
a.re*b.im+a.im*b.re);
  }

  void main ()
  {
    COMPLEX z1, z2(1, 2);

    z1 = z1 + z2;
    z2 = z1 * z2;
```

```
   z1.print ();
   z2.print ();
   (z1 + z2).print ();
}
```

Инкапсуляция приводит к появлению нового понятия: абстрактный тип данных. Абстрактный тип данных это тип скрывающий детали реализации, доступ к данным этого типа осуществляется только через интерфейс. Свойства абстрактного типа данных определяются только его интерфейсом (членами класса раздела *public*), а не его внутренней структурой или деталями реализации. В качестве примера рассмотрим реализацию стека на базе списка:

```
/* Здесь должна быть реализация списка */
class STACK {
   LIST S;
public:
   void Push (int x);
   int Pop (void);
};

void STACK::Push(int x)
{ S.Cons (x); }

int STACK::Pop ()
{  int x = S.Car ();
   S = S.Cdr ();
   return x;
}

void main ()
{
   STACK stack;

   stack.Push (1);
   stack.Push (2);
   stack.Push (3);

   printf ("%d\n", stack.Pop ());
   printf ("%d\n", stack.Pop ());
   printf ("%d\n", stack.Pop ());
}
```

В приведённом примере методы класса *STACK* функции *Push* и *Pop* образуют интерфейс класса. Работа с содержимым стека возможна только через эти методы, а конкретная реализация стека (массив или список) никак не влияет на его использование.

Права доступа к элементам класса. Другой важной стороной инкапсуляции (помимо объединения данных и методов класса) является управление доступом к элементам класса. В соответствии с установленными правами доступа элементы класса могут принадлежать к одной из следующих категорий:

- *public* — общедоступные. Доступ к таким элементам возможен в любом месте программы в соответствии с общими правилами области видимости и времени существования объектов.
- *private* — частные, скрытые. Доступ к таким элементам защищён и возможен только для элементов данного класса или дружественным к классу.
- *protected* — используется для регулирования прав доступа при наследовании классов (об этом позже).

Доступность элементов класса задаётся при помощи меток *public* и *private*. Эти метки могут присутствовать в определении класса произвольное число раз и в произвольном порядке. По умолчанию, если не задано иное, элементы класса определены как *private*.

```
class TEST {
   . . .
   <закрытая секция>
   . . .
public:
   . . .
   <общедоступная секция>
   . . .
private:
   . . .
   <закрытая секция>
   . . .
public:
   . . .
   <общедоступная секция>
   . . .
};
```

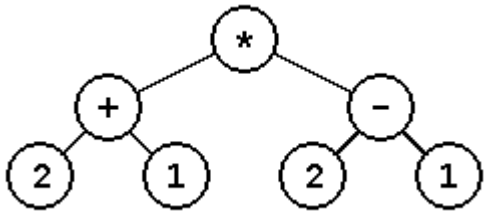
Смотри примеры приведённые ранее

Объекты, объявленные дружественными (*friend*) в определении класса, не являются элементами класса, но имеют разрешение на доступ к приватным элементам класса.

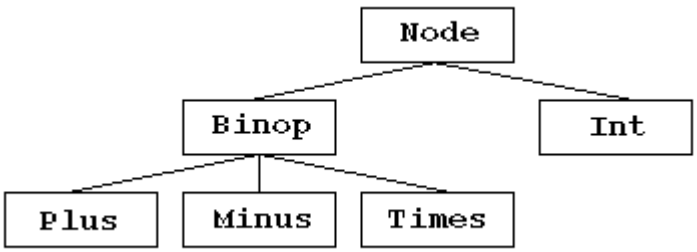
Наследование. Довольно часто некоторые элементы разные классы похожи между собой: имеют одинаковый смысл и реализацию. В подобном случае бывает полезно выделить такие элементы в отдельный (базовый) класс и

использовать его в качестве основы при построении других (производных) классов путём специализации или расширения базового класса. Подобный образ действия называется ***наследование*** и наряду с инкапсуляцией является основой ООП.

В качестве примера использования механизма наследования рассмотрим задачу реализации синтаксического дерева арифметических выражений для программы калькулятора. Для конкретного выражения $(2+1)*(2-1)$ это дерево имеет вид:



Узлы этого дерева представляют различные объекты: числа, бинарные операции сложения, вычитания и умножения. У всех этих объектов есть одна общая черта: они являются узлами дерева. В рассматриваемом случае все узлы дерева можно разбить на две группы: числа и бинарные операции. Бинарные операции помимо общих свойств (два операнда) имеют отличительные черты характерные для операций сложения, вычитания и умножения. В соответствии с этим иерархическая структура классов для задачи будет иметь вид:



Общие свойства всех элементов синтаксического дерева собраны в классе *Node*:

```
class Node {
public:
   enum {PLUS, MINUS, TIMES, INT};
   int code;
   Node (int c) : code (c) {}
   int eval ();
};
```

Теперь этот класс используем в качестве базового при построении производных классов *Int* и *Binop*:

```
class Int : public Node {
public:
   int val;
   Int (int v) : Node (INT) { val = v; }
};

class Binop : public Node {
public:
   Node *left, *right;
   Binop(int c,Node *l,Node *r) : Node(c) {left=l;right=r;}
};
```

На базе класса бинарных операций создаём классы *Plus*, *Minus* и *Times*:

```
class Plus : public Binop {
public:
   Plus (Node *l, Node *r) : Binop (PLUS, l, r) {}
};

class Minus : public Binop {
public:
   Minus (Node *l, Node *r) : Binop (MINUS, l, r) {}
};

class Times : public Binop {
public:
   Times (Node *l, Node *r) : Binop (TIMES, l, r) {}
};
```

Определяем метод *eval()* для вычисления значения арифметического выражения *соответствующего дереву:*

```
int Node::eval ()
{
   switch (code) {
      case INT:
         return ((Int*)this)->val;
      case PLUS:
         return ((Binop*)this)->left->eval() + ((Binop*)this)-
>right->eval();
      case MINUS:
         return ((Binop*)this)->left->eval() - ((Binop*)this)-
>right->eval();
      case TIMES:
         return ((Binop*)this)->left->eval() * ((Binop*)this)-
>right->eval();
   }
   return 0;
```

```
}
```

Пример создания и вычисления значения выражения:

```
void main () {
// вычисляем (2+1)*(2-1)
   Node *expr = new Times(
      new Plus(new Int(2), new Int(1)),
      new Minus(new Int(2), new Int(1))
   );

   printf ("%d", expr->eval ());
}
```

Наследование можно понимать так, что производный класс наследует элементы базового класса и добавляет к ним свою, специфическую информацию. Так, например, имеем базовый класс:

```
class B {
. . .
public:
   void b ();
. . .
};
```

и производный класс:

```
class D : public B {
. . .
};
```

Представитель производного класса *D*

```
D d;
```

может использовать метод базового класса как свой собственный:

```
d.b ();
```

Использование модификатора прав доступа *public* при создании производного класса оставляет права доступа к элементам базового класса без изменения. Можно ограничить доступ к элементам базового класса, в этом случае следует использовать модификатор *private*. Наследование прав доступа к элементам базового класса формируется по следующим правилам:

Доступ в базовом классе	Модификатор прав доступа	Наследование прав доступа

private	private	не доступны
protected		private
public		private
private	public	не доступны
protected		protected
public		public

Если модификатор прав доступа не указан, то по умолчанию используется *private*, который ограничивает *protected* и *public* до *private*. В любом случае доступ к элементам базовых классом не может быть облегчён, а только ужесточается.

Наследование является важной, основополагающей чертой объектно-ориентированного программирования. В основе ООП лежит процесс разработки и построения иерархии классов.

8. Квантовые вычисления

Предпосылки возникновения интереса к квантовым вычислениям:

- Уменьшение элементной базы до таких размеров, для которых начинают сказываться законы квантовой физики.
- Квантовые свойства систем (суперпозиция состояний, запутанность состояний, …) позволяют реализовывать алгоритмы принципиально отличные от классических.

Аппаратная реализация вычислительных систем возможна на элементной базе, функционирующей на основе двух, принципиально различных, физических принципах: классическая физика и квантовая физика.

Основой вычислительной системы является элемент, предназначенный для хранения минимальной информации - бит.

В классической физике бит задаётся системой с двумя устойчивыми (или квазиустойчивыми) состояниями: триггер, конденсатор, магнитный домен, … . Одно из этих состояний условно обозначается «0», другое - «1». В математической модели это элемент множества $\{0,1\}$.

В квантовой физике бит это квантовая система с двумя возможными состояниями: спин электрона в магнитном поле, энергетические уровни атома, … . В математической модели состояние системы описывается вектором в 2-мерном пространстве: $|\psi> = \alpha \cdot |e_1> + \beta \cdot |e_2>$. Здесь $|e_1>, |e_2>$ - базисные вектора, α, β - амплитуды состояния, $|\alpha|^2 + |\beta|^2 = 1$. Потенциально, вектору $|\psi>$ соответствует бесконечное множество состояний задаваемых значениями амплитуд α, β . При попытке определить состояние системы посредством измерения, происходит взаимодействие квантовой системы с макроскопическим прибором и система переходит в одно из базисных состояний $|e_1>$ или $|e_2>$ с вероятностями $|\alpha|^2$ и $|\beta|^2$ соответственно. Такая квантовая система, представляющая бит, называется квантовый бит или кубит (qbit).

Поведение квантового компьютера, как и любой квантовомеханической системы, в стационарном случае описывается уравнением Шрёдингера для стационарной системы $|\psi(t)> = e^{iHt}|\psi(0)>$ (оператор Гамильтона H не зависит от времени).

Здесь:

$|\psi(0)>$ - начальное состояние системы кубитов, входные данные программы.

$|\psi(t)>$ - конечные состояние системы кубитов, результат вычислений.

H - оператор Гамильтона, эрмитов оператор ($H^{+}=H$, $H^{+}=(H^{*})^{T}$) не зависящий от времени.

Таким образом, в общем случае, процесс квантовых вычислений можно описать уравнением

$$|\psi_2> = U|\psi_1>$$

где U - унитарный оператор ($U^{+}\cdot U = I$), задающий вычисления в соответствии с заданным алгоритмом. Р. Фейнман [2] показал каким образом, по заданному оператору U , можно построить гамильтониан системы и, тем самым, физически реализовать процесс квантовых вычислений.

В классической физике процесс измерения не меняет состояния системы. Наблюдатель не влияет на исследуемый объект, его вмешательство в поведение системы может быть сделано сколь угодно малым. В квантовом случае ситуация существенно иная: принципиально невозможно исключить влияние наблюдателя на поведение исследуемой системы.

В квантовой механике наблюдаемой величине соответствует эрмитов оператор: $A^{+}=A$. В качестве базиса пространства состояний удобно использовать собственные вектора $|\xi_k>$ оператора A :

$$A|\xi_k> = \lambda_k \xi_k$$

здесь λ_k - собственные значения оператора A .

При измерении системы с вектором состояния

$$|\psi> = \sum_k a_k |\xi_k>$$

будет наблюдаться значение λ_k с вероятностью $|a_k|^2$.

Можно считать, что в отличии от классического случая система получает фиксированное значение физической величины лишь после процесса измерения. До измерения состояние кубита не определено и является суперпозицией (смесью) разных состояний:

$$\alpha\cdot|e_1> + \beta\cdot|e_2> \qquad \alpha, \beta \in \mathbb{C} \quad |\alpha_i|^2 + |\beta|^2 = 1$$

Это одно из обстоятельств, определяющее принципиальное отличие классической и квантовой вычислительных систем.

Квантовый параллелизм. Алгоритм Дойча. Тот факт, что квантовая система одновременно может находиться в суперпозиции нескольких состоя-

ний, позволяет использовать новый тип параллельных вычислений - квантовый параллелизм.

Рассмотрим в качестве примера квантовых вычислений алгоритм Дойча.

Возьмём булеву функцию одной переменной $f(x):\{0,1\}\rightarrow\{0,1\}$. Существует всего четыре таких функции:

1. $f(x)=0$
2. $f(x)=1$
3. $f(x)=x$ или $f(0)=0,\ f(1)=1$
4. $f(x)=\neg x$ или $f(0)=1,\ f(1)=0$

Первые два случая соответствуют функции константе. Случаи 3 и 4 - не константа. Чтобы определить, к какому типу принадлежит функция, при использовании классической системы потребуется выполнить два запроса на вычисление её значения. При использовании квантовой схемы потребуется только один запрос.

Вначале возьмём систему из одного квантового кубита в базисном состоянии соответствующем логическому "0". Получить такое состояние довольно просто: берём кубит в произвольном состоянии, которое является суперпозицией базисных $|q>=\alpha\cdot|0>+\beta\cdot|1>$ и производим измерение в выбранном базисе. Если результат измерения "0", то переходим к следующему шагу, если нет - берём другой кубит и повторяем процесс измерения.

Далее, для определённости будем полагать:

$$|0>=\begin{pmatrix}1\\0\end{pmatrix}\quad |1>=\begin{pmatrix}0\\1\end{pmatrix}$$

$$H=\frac{1}{\sqrt{2}}\cdot\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

К полученному кубиту применим преобразование Адамара H (не путать с гамильтонианом!):

$$H|0>=\frac{1}{\sqrt{2}}\cdot\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\cdot\begin{pmatrix}1\\0\end{pmatrix}=\frac{1}{\sqrt{2}}\cdot\begin{pmatrix}1\\1\end{pmatrix}=\frac{1}{\sqrt{2}}\cdot(|0>+|1>)$$

и затем ещё одно преобразование - фазовый запрос O_f :

$$O_f=\begin{pmatrix}(-1)^{f(0)} & 0\\ 0 & (-1)^{f(1)}\end{pmatrix}$$

$$O_f H|0> = \frac{1}{\sqrt{2}}\cdot\begin{pmatrix}(-1)^{f(0)} & 0\\ 0 & (-1)^{f(1)}\end{pmatrix}\cdot\begin{pmatrix}1\\1\end{pmatrix} = \frac{1}{\sqrt{2}}\cdot\begin{pmatrix}(-1)^{f(0)}\\(-1)^{f(1)}\end{pmatrix} = \frac{1}{\sqrt{2}}\cdot((-1)^{f(0)}|0> + (-1)^{f(1)}|1>)$$

Теперь ещё раз преобразование Адамара:

$$H O_f H|0> = \frac{1}{2}\cdot\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}\cdot\begin{pmatrix}(-1)^{f(0)}\\(-1)^{f(1)}\end{pmatrix} = \frac{(-1)^{f(0)}+(-1)^{f(1)}}{2}\cdot|0> + \frac{(-1)^{f(0)}+(-1)^{f(1)}}{2}\cdot|1>$$

Таким образом, в результате последовательного применения преобразования Адамара, фазового запроса и ещё одного преобразования Адамара к кубиту в состоянии "0" мы получаем суперпозицию состояний состояние $|0>$ и $|1>$ с амплитудами

$$\alpha = \frac{(-1)^{f(0)}+(-1)^{f(1)}}{2}$$

$$\beta = \frac{(-1)^{f(0)}-(-1)^{f(1)}}{2}$$

Обратим внимание на тот факт, что использование одного запроса к системе позволило вычислить функцию для двух значений аргумента. Вначале мы приготовили суперпозицию состояний, затем выполнили некоторые вычисления в соответствии с алгоритмом решаемой задачи. Так как состояние квантовой системы не определено, то вычисление функции производится для всех, потенциально возможных значений аргументов. В этом и состоит квантовый параллелизм.

Теперь выделим из результата $H O_f H|0> = \alpha|0> + \beta|1>$ интересующую нас информацию. Для функция $f(x)$ типа константа $f(0)=f(1)$ получаем амплитуды $\alpha=\pm 1$, $\beta=0$ и измерение конечного состояния с вероятностью $P=|\alpha|^2=1$ определит, что система находится в состоянии $|0>$. Для функции не являющейся константой $f(0)\neq f(1)$, амплитуды равны $\alpha=0$, $\beta=\pm 1$ и с вероятностью $P=|\beta|^2=1$ получаем состояние $|1>$.

В результате: если в процессе измерения полученного состояния $H O_f H|0>$ квантовой системы мы получим результат $|0>$, то это значит, что функция $f(x)$ является константой или не константой в противном случае.

Система n кубитов. Алгоритм Дойча-Джоза. Алгоритм Дойча-Джоза является обобщением алгоритма Дойча на случай функции n переменных: $f(x_{0,}\ldots,x_{n-1}):\{0,1\}^n \to \{0,1\}$. Как и прежде будем считать, что $f(x_{0,}\ldots,x_{n-1})$ является либо функцией константа, и её значение равно "0" или "1" для всех 2^n вариантов значений аргументов. Или сбалансированной функцией , если в половине всех случаев она принимает значение "0" и значение "1" - в другой половине случаев. Другие варианты распределения значений функции среди 2^n комбинаций 0 и 1 аргументов не рассматриваются. Алгоритм Дойча-Джоза позволяет посредством одного запроса к квантовой системе определить, к какому типу (константа или сбалансированная) принадлежит искомая функция.

Для представления булевых переменных $x_0.x_1,\ldots,x_{n-1}$ в вычислительной системе будем использовать регистр (систему) n кубитов $|q_0,q_1,\ldots,q_{n-1}>$. Регистру из n кубитов соответствует вектор в 2^n -мерном пространстве. Введём базис в этом пространстве и сопоставим вектора этого базиса значениям кубитов регистра следующим образом:

$$e_0=|0,\ldots,0,0,0>=\begin{pmatrix}1\\0\\0\\\ldots\\0\end{pmatrix}\quad e_1=|0,\ldots,0,0,1>=\begin{pmatrix}0\\1\\0\\\ldots\\0\end{pmatrix}\quad e_2=|0,\ldots,0,1,0>=\begin{pmatrix}0\\0\\1\\\ldots\\0\end{pmatrix}$$

$$\ldots\quad e_{n-1}=|1,\ldots,1,1,1>=\begin{pmatrix}0\\0\\0\\\ldots\\1\end{pmatrix}$$

Т. е., если последовательность нулей и единиц в регистре кубитов (базисных) $|q_0,q_1,\ldots,q_{n-1}>$ рассматривать как некое число в двоичном представлении, то это число является номером единственной ненулевой (единичной) компоненты базисного вектора в 2^n -мерном пространстве.

Для произвольных n кубитов $|q_i>=\alpha_i|0>+\beta_i|1>,\ 0\leqslant i\leqslant n-1$ вектор

$$|u^n>=\begin{pmatrix}u_0^n\\\ldots\\u_{p-1}^n\end{pmatrix}\quad \text{где } p=2^n$$

соответствующий n кубитам, можно получить воспользовавшись рекуррентными соотношениями:

$$|u^1\rangle = |q_0\rangle = \begin{pmatrix}\alpha_0\\ \beta_0\end{pmatrix} \quad \dots \quad |u^n\rangle = |q_0 q_1 \dots q_{n-1}\rangle = \begin{pmatrix} u_0^{n-1}\alpha_{n-1}\\ u_0^{n-1}\beta_{n-1}\\ u_1^{n-1}\alpha_{n-1}\\ u_1^{n-1}\beta_{n-1}\\ \cdot\ \cdot\ \cdot\\ u_{p/2-1}^{n-1}\alpha_{n-1}\\ u_{p/2-1}^{n-1}\beta_{n-1}\end{pmatrix} \quad \text{где} p = 2^n$$

Теперь приготовим систему из n кубитов в состоянии "0":

$|\psi_0\rangle = |0,0,\dots,0\rangle = |0\rangle \otimes |0\rangle \otimes \dots \otimes |0\rangle$ здесь $\otimes$ - тензорное произведение.

Преобразование Адамара:

$$|\psi_1\rangle = H^n |\psi_0\rangle = H \otimes \dots \otimes H\, |\psi_0\rangle = H|0\rangle \otimes \dots \otimes H|0\rangle$$

Фазовый запрос:

$$|\psi_2\rangle = O_f |\psi_1\rangle \qquad O_f = \begin{pmatrix} (-1)^{f(u_0)} & & & \\ & (-1)^{f(u_1)} & & 0 \\ & & \dots & \\ & 0 & & (-1)^{f(u_{p-1})} \end{pmatrix} \quad p = 2^n$$

К полученному результату ещё раз преобразование Адамара:

$$|\psi_3\rangle = H^n |\psi_2\rangle$$

Проводя рассуждения, аналогичные предыдущим для алгоритма Дойча, мы придём к выводу, что в конечном состоянии системы

$$\alpha|0,0,\dots,0\rangle + \beta|всёпрочее\rangle = H^n O_f H^n |0,0,\dots,0\rangle$$

амплитуда α принимает значение ± 1 если функция $f(x_{0,}\dots,x_{n-1})$ является константой, и $\alpha = 0$ для сбалансированной функции.

Алгоритм Гровера. Алгоритм Гровера - это квантовый алгоритм нахождения решения уравнения $f(x_{0,}\ldots,x_{n-1})=1$, где $f(x_{0,}\ldots,x_{n-1}):\{0,1\}^n \to \{0,1\}$ булева функция n переменных.

Как и в предыдущем случае начнём с системы нулевых n кубитов: $|\psi_0>=|0,0,\ldots,0>$. Затем применим оператор Адамара и приготовим смесь состояний: $|\psi_1>=H^n|\psi_0>$. Теперь несколько раз применим унитарный оператор G :

$$|\psi_2>=G\ldots G|\psi_1>$$

где: $G=RO_f$, O_f - фазовый запрос, R - оператор диффузии:

$$R=\begin{pmatrix} 1-1/2^{n-1} & 1/2^{n-1} & \ldots & 1/2^{n-1} \\ 1/2^{n-1} & 1-1/2^{n-1} & \ldots & 1/2^{n-1} \\ \ldots & \ldots & \ldots & \ldots \\ 1/2^{n-1} & 1/2^{n-1} & \ldots & 1-1/2^{n-1} \end{pmatrix}$$

Для получения наилучшего результата оператор G следует применять $\lceil \pi/4\cdot\sqrt{2^n} \rceil$ раз ([ ] обозначает целую часть).

И ещё раз оператор Адамара: $|\psi_3>=H^n|\psi_2>$.

В итоге получаем вектор состояния:

$$|\psi_3>=H^n \underbrace{G\ldots G}_{k} H^n|0,0,\ldots,0> \qquad k=\frac{\pi}{4}\sqrt{2^n}$$

Напомним, что вектор состояния $|\psi_3>$ является суперпозицией 2^n возможных состояний системы n кубитов. Состояние, соответствующее решению уравнения $f(x_{0,}\ldots,x_{n-1})=1$, будет иметь максимальную амплитуду и может проявиться в процессе измерения с наибольшей вероятностью. Вероятность получить неправильный результат в алгоритме Гровера оценивается как $O(1/2^n)$.

Эмуляция квантовых вычислений. Квантовый бит (qbit) $|q>$ - двухкомпонентный вектор состояния квантовой системы. В общем случае $|q>=\alpha\cdot|0>+\beta\cdot|1>$. Где α,β - комплексные числа, $\alpha^2+\beta^2=1$;

|0> и |1> - базисные вектора, соответствующие двум устойчивым состояниям системы, условно обозначаемым «0» и «1».

Сопоставим квантовому биту класс:

```
class QBIT {
public:
   COMPLEX a;
   COMPLEX b;
   QBIT (COMPLEX A, COMPLEX B);
   void print ()  { printf("("); a.print_re(); b.print_re();
printf(")"); }};
```

Конструктор класса:

```
QBIT::QBIT (COMPLEX A, COMPLEX B) {
   float norm = sqrt (A.mod2() + B.mod2());
   if ( norm == 0.0 ) {
      a.re = M_SQRT1_2;   a.im = 0.0;
      b.re = M_SQRT1_2;   b.im = 0.0;
   }
   else {
      a = 1.0/norm *A;
      b = 1.0/norm *B;
   }
}
```

Вектора базисных состояний:

```
QBIT e0(COMPLEX(1,0),COMPLEX(0,0));     // базисный вектор |0>
QBIT e1(COMPLEX(0,0),COMPLEX(1,0));     // базисный вектор |1>
```

Система (регистр) N кубитов:

```
template <int N> class QBITN {
   COMPLEX s[1<<N];
public:
   QBITN (QBIT V[N]);          // регистр из N кубитов |V> = a*|
0> b*|1>
   void print ();

   void gH ();                    // преобразование Адамара
   void Of ();                    // Фазовый поворот
   void R ();                     // Диффузия
};
```

Конструктор класса QBITN, строит систему N кубитов из массива V[N] кубитов:

```
template <int N>
QBITN<N>::QBITN (QBIT V[N]) {
```

```
   int P = 1<<N;
   int i, j, k, n;

   for (i=0; i<P; i++) {
      s[i] = COMPLEX (1.0, 0.0);
   }

   for (k=2, n=0; k<=P; k*=2, n++) {
      for (i=0; i<P/k; i++) {
         for (j=0; j<k/2; j++) {
            s[i + 2*j*P/k]     = s[i + 2*j*P/k] * V[n].a;
            s[i+P/k + 2*j*P/k] = s[i+P/k + 2*j*P/k] * V[n].b;
         }
      }
   }
}
```

Метод print():

```
template <int N>
void QBITN<N>::print () {
   int p = 1<<N;
   int i;

   printf("(");
   for (i=0; i<p; i++) {
      printf(" ");
      s[i].print_re();
   }
   printf(")\n");
};
```

Эта функция используется для вывода амплитуд соответствующего вектора состояния. Напомним, просматривать значения амплитуд возможно лишь при эмуляции квантовых вычислений на классическом компьютере. При выполнении реальных квантовых вычислений результат можно получить лишь в виде вероятности появления того или иного базисного состояния.

Некоторые унитарные пробразования системы кубитов.

Преобразование Адамара:

```
template <int N>
void QBITN<N>::gH () {
   int P = 1<<N;
   float U[P][P];

   int p, q, r, x;
   int i;
```

```
   COMPLEX s1[P];

   for (p=0; p<P; p++) {
      for (q=0; q<P; q++) {
         r = p&q;
         x = 0;
         for (i=0; i<N; i++) {
            x = x + (r&1);
            r >>= 1;
         }
         U[p][q] = (x&1 ? -1 : 1) * pow (M_SQRT1_2, N);
      }
   }

   for (p=0; p<P; p++) {
      s1[p] = s[p];
   }

   for (p=0; p<P; p++) {
      s[p] = COMPLEX (0,0);
      for (q=0; q<P; q++) {
         s[p] = s[p] + U[p][q] * s1[q];
      }
   }
}
```

Преобразование диффузии:

```
template <int N>
void QBITN<N>::R () {
   int P = 1<<N;
   COMPLEX z[P];
   int i, j;

   for (i=0; i<P; i++) {
      z[i] = s[i];
      s[i] = -1 * z[i];
   }

   for (i=0; i<P; i++) {
      for (j=0; j<P; j++) {
         s[i] = s[i] + 1.0/(1<<(N-1)) * z[j];
      }
   }
}
```

Фазовый запрос:

```
template <int N>
void QBITN<N>::Of () {
   int P = 1<<N;
   int i;
```

```
   for (i=0; i<P/2; i++)
      s[i] = -1*s[i];
}
```

Здесь, в зависимости от вида используемой булевой функции, необходимо явно указать изменение знака соответствующих компонент вектора состояния системы N кубитов. В данном случае используется функция у которой половина значений равна "1", а другая половина — "0".

Используя введённые классы QBIT и QBITN выполним вычисления соответствующие алгоритмам Дойча-Джоза и Гровера. Для определённости будем использовать систему 4-х кубитов (соответствующий вектор состояния будет иметь 16 компонент).

Алгоритм Дойча-Джоза:

В качестве исходного состояния системы возьмём кубиты в состоянии $|0>$:

```
QBIT v[N] = {e0, e0, e0, e0};
QBITN <N> q (v);
```

Применяя преобразование Адамара приготовим смесь состояний, затем фазавый запрос и ещё раз преобразование Адамара:

```
q.gH();
q.Of();
q.gH();
```

В случае используемой константной функции $f(x_{0,}\ldots,x_{n-1})$ со всеми значениями равными "0" или "1" на выходе получаем вектора состояния

(1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00)

или

(-1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00)

При использовании «сбалансированной» функции вектор состояния на выходе алгоритма будет иметь вид

(0.00 x.xx … x.xx)

Конкретные величины x.xx зависят от того, как значения функции «связаны» со значениями её аргументов.

Алгоритм Гровера:

Для определённости возьмём функцию $f(x_{0,}\ldots,x_{n-1})$ с единственным ненулевым значением при $x_0=0,x_1=0,x_2=0,x_3=1$. Этому случаю соответствует вектор состояния:

$$|q_{0,}q_{1,}q_{2,}q_3>=\begin{pmatrix}0\\1\\\ldots\\0\end{pmatrix}$$

Что приводит к следующему оператору фазового запроса:

```
template <int N>
void QBITN<N>::Of () {
    s[1] = -1*s[1];
}
```

Как и в предыдущем случае приготовим «смешанное» начальное состояние:

```
QBIT v[N] = {e0, e0, e0, e0};
QBITN <N> q (v);
q.gH();
```

Затем применим нескольо раз преобразования фазового запроса и диффузии:

```
for (i=0; i<iterations; i++) {
   q.Of();
   q.R();
}
```

Для системы 4-х кубитов число наилучшее итераций будет равно .

$$\text{iterations}=\left[\frac{\pi\sqrt{16}}{4}\right]=3$$

Посмотрим, как меняются амплитуды состояния системы после каждой итерации.

начальное состояние:
(0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25)

1-я итерация:
(0.19 0.69 0.19 0.19 0.19 0.19 0.19 0.19 0.19 0.19 0.19 0.19 0.19 0.19 0.19 0.19)

2-я итерация:
(0.08 0.95 0.08 0.08 0.08 0.08 0.08 0.08 0.08 0.08 0.08 0.08 0.08 0.08 0.08 0.08)

3-я итерация (лучший результат):
(0.05 0.98 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05)

4-я итерация:
(-0.16 0.76 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16 -0.16)

Заключение. В настоящий момент квантовые вычисления, вычисления основанные на использовании квантовых свойств элементной базы, находятся в начальной стадии развития. Дальнейший прогресс в этой области будет зависить от решения технических задачи, связанных с созданием элементной базы квантовых компьютеров. Кроме этого, требуется разработка регулярного метода, позволяющего строить унитарное преобразование, соответствующее заданному алгоритму вычислений.

9. Литература

1. Лавров С., Программирование. Математические основы, средства, теория. С-Петербург, «БХВ-Петербург», 2001
2. Керниган Б., Риччи Д. Язык программирования С. М. 1992
3. "Введение в программирование на языке ЛИСП." У.Маурер, Москва, Мир, 1976.
4. "Функциональное программирование. Применение и реализация." П.Хендерсон, Москва, Мир, 1983.
5. "Мир Лиспа." Э.Хювенен и Й.Сеппянен, т.1, Москва, Мир, 1990.
6. А. Филд, П. Харрисон «Функциональное программирование» Москва, Мир, 1993 (глава 6)
7. Ч. Уэзерелл «Этюды для программистов» Москва, Мир, 1982 (глава 28 — Построение интерпретатора языка Трак)
8. У. Клоксин, К. Меллиш «Программирование на языке Пролог» Москва, Мир, 1987.
9. Страуструп Б. Язык программирования С++. М. 1991
10. Ландау Л.Д., Лифшиц Е.М. Квантовая механика (нерелятивистская теория). — Издание 6-е, исправленное. — М.: Физматлит, 2004. («Теоретическая физика», том III).
11. Р. Фейнман «Квантовомеханические ЭВМ», УФН, 1986г., том 149, вып 4, стр. 671-688
12. М. Вялый «Квантовые алгоритмы: возможности и ограничения», http://www.lektorium.tv/course/?id=22805