Функциональное программирование

[1. Синтаксис языка ЛИСП. Списки. Атомы. Виды атомов. Комментарии](#h.pn039ba7i129)

[2. Вычисление выражений. Нормальный и аппликативный порядок. Особая форма quote](#h.3dyziipqb0gv)

[3. Особая форма define. Способы использования define. Блочная структура программы](#h.pwxym5ayq1gq)

[4. Условные выражения и предикаты. Тест стратегии вычисления](#h.zgyzyeko9uor)

[5. Понятие вычислительного процесса. Итерация, линейная рекурсия, древовидная рекурсия](#h.jkimesn8g4ko)

[6. Оценивание вычислительного процесса. Порядки роста](#h.d99sd1btqyqr)

[7. Понятие вычислительной модели. Требования к модели](#h.uetk18y3n1ov)

[8. Детерминированная машина Тьюринга. Вычислимость по Тьюрингу. Тезис Тьюринга](#h.8vcl1o2h9mrb)

[9. Расширения машины Тьюринга](#h.7m8rrn5phqg2)

[10. Примитивно рекурсивные функции Чёрча](#h.8hsycmcrk25c)

[11. Примитивно-рекурсивные операторы](#h.vdonjuv84xs0)

[12. Функция Аккермана и её смысл](#h.5l08t096eiq3)

[13. Частично-рекурсивные функции. Тезис Чёрча](#h.b2ti7ykn1kg6)

[14. Нумерация алгоритмов и вычислимых функций](#h.kuib2sdzz58b)

[15. Теорема параметризации](#h.90hvb94ae1zl)

[16. Понятие универсального алгоритма. Реализация на трёхленточной МТ](#h.z1zlk51eakov)

[17. Проблема несчётности множества общерекурсивных функций](#h.ymlay8ob1ou5)

[18. Проблема определения общерекурсивности](#h.iqyndjt4jiih)

[19. Проблема самоприменимости](#h.kyvxj8uyh2vl)

[20. Проблема остановки](#h.yc3artid6tmb)

[21. Необходимость частично-рекурсивной функции](#h.2aylr08e1bs5)

[22. Теорема Райса (классификация алгоритмов)](#h.1goqm02gg7d4)

[23. Проблема соответствий Поста (поиск конкатенации)](#h.ayud5440kovt)

[24. Труднорешаемые проблемы](#h.vlyq3ac9j440)

[25. Особая форма лямбда (lambda)](#h.esdbydqgncgl)

[26. Особая форма let. Сравнение let и define](#h.aah1mel4tpsv)

[27. Реализация особой формы let](#h.6q6fcfo4tun4)

[28. Числа Чёрча](#h.pucgvis6zad6)

[29. Пары значений. Реализация пар значений](#h.gcco9x412by0)

[30. Вертикальные барьеры абстракции. Конструкторы. Селекторы.](#h.ei9ixehgh4ky)

[31. Списки. Основные функции для работы со списками. Реализация списков.](#h.eefg9b9tseer)

[32. Реализация вспомогательных функций для работы со списками (обращение по индексу, мэпинг)](#h.kagv3r5oue9f)

[33. Горизонтальные барьеры абстракции. Диспетчеризация по типу](#h.sn1g19ba41fn)

[34. Программирование, управляемое данными](#h.84iwykxouedp)

[35. Система с обобщёнными операциями](#h.ryb8hcq0994w)

[36. Приведение типов. Реализация](#h.ff8k2rpeqmdx)

[37,38. Моделирование объектов.Особые формы set! и begin. Моделирование простых объектов](#h.t1euh1rhmogx)

[39. Моделирование сложных объектов](#h.ry1borvx9j5b)

[40. Оператор присваивания. Достоинства и недостатки его использования](#h.q7847otnuy1w)

[41. Модель вычисления с окружениями](#h.cyb574vwh7h2)

[42. Моделирование потоков](#h.gyqzll78ts7o)

# 1. Синтаксис языка ЛИСП. Списки. Атомы. Виды атомов. Комментарии

Языки программирования: виды решаемых задач. Алгоритмически неразрешимые задачи.

Два наиболее простых формализма задания правил протекающих процессов:

* + Машина Тьюринга и их модификации
  + λ-исчисление. Всё построено на функциях. Автор – Чёрч.
    - LISP
    - Common LISP
    - Haskell
    - Erlang

**Виды и свойства процессов**

**LISP** (LISting Processing) (Lots of Idiotic Silly Parentheses) — функциональный язык. Основной конструкцией языка является *список*:

**Синтаксис в форме БНФ:**

Список ::= Элемент | Список элементов

Элемент ::= Атом | Список

Атом ::= Число | Символ | #t | #f | ‘ ‘ | “ “

**Синтаксис в форме дерева:**

* Список
  + Элемент
    - Атом
      * Логические признаки: #t (true), #f (false)
      * Число
      * Символ (нетипизированный)
      * Строки (заключаются в апострофы или в кавычки.)
    - Список

Типов мало, данных много. Типы отдельно не определяются. Например, (Ivanov Ivan Ivanych) — тип не нужен, потому что в нём нет необходимости. Комментарии — точка с запятой.

# 2. Вычисление выражений. Нормальный и аппликативный порядок. Особая форма quote

**LISP** – интерпретируемый язык.

Интерпретация данных (a, b, c, d, e):

* a – функция (особая форма), которую необходимо выполнить
* b, c, d, e, … – аргументы этой функции

**Особая форма** — это инструкция (например, «закрыть файл», «окончание работы», и т.п.)

Примеры:

(+ 1 2) → 3

(+ (\* 2 3) 4) → 10

Функции для работы с типами:

* (number? 1) Проверка, является ли числом (в данном случае true).
* (list? ...) Проверка, является ли списком.
* (string? ...) Проверка, является ли строкой.
* (eqv? a b) Эквивалентны ли списки?

Стратегии вычисления:

* **Аппликативный порядок** *(снизу вверх)*: сначала вычисляются все аргументы, затем они передаются на вход функции.
* **Нормальный порядок** *(сверху вниз)*: функция сначала определяет необходимые ей аргументы, а потом они вычисляются по мере необходимости.

**Пример**. ((i > j) && (i > 0)):

* В *аппликативном* порядке сначала будут вычислены значения условия, а затем применена операция И.
* В *нормальном* порядке функция сама вычисляет значения своих аргументов. Некоторые вычисления могут пропускаться (например, второе условие может не вычисляться, если первое == false).

**Оператор quote** — выполняет функцию экранирования. Запрещает интерпретацию (вычисление) списка. Синтаксис:

* (quote (…))
* `(…)

# 3. Особая форма define. Способы использования define. Блочная структура программы

**Синтаксис**: (define, что, как)

**Примеры:**

* (define a 1)
* (define a (sin (+ b 1))) ; a=sin(b+1)
* (define (f x) (+ x 1))

Повторно присваивать значение одной переменной нельзя.

**Забавные примеры**

(define a b) – Значение a равняется значению b

(define a (b)) – Значение a равняется значению функции без аргументов b

(define (a) b) – функция a, не принимающая аргументов и возвращающая b

(define (a) (b)) – функция a возвращает результат работы функции b

(define a `b) – символ a равняется символу b

(define a `(b)) – a равняется списку из одного элемента (b)

**Блочная структура программы**

(define ЧТО

(define ...)

КАК ) ← здесь можно использовать то, что определено во вложенном define.

(define (f x)

(define (g y) (+ y 1))

(g (\* x 10)))

- абстракции высших порядков

(define (a n) ... )

(define (sum n)

(if (= n 0) 0

(+ (a n) (sum (- n 1))))

Чтобы написать функцию, способную подсчитывать сумму другого ряда. Где-то **тут** передается функция как параметр:

(define (sum **a** n)

(if (= n 0) 0

(+ (a n) (sum a (- n 1))))

Использование:

(sum a 10)

Типа

(define (b x) (+ x 20))

(sum sin 100)

# 4. Условные выражения и предикаты. Тест стратегии вычисления

**Оператор cond**

Синтаксис:

(cond (усл рез) … (усл рез) (else рез))

Пример (получение числа корней квадратного уравнения):

(define (D a b c) ( - ( \* b b) (\* 4 a c)))

(define (RC a b c)

(cond ((> (D a b c) 0 ) 2 )

((= (D a b c) 0 ) 1 )

(else 0))

)

**Оператор if**

Синтаксис

(if условие результат1 результат2)

Пример (нахождение факториала):

(define (f n) (if (= n 1) 1 (\* n (f (- n 1)))))

Пример (тест Бэрбриджа):

(define (p) (p))

(define (test x y) (if (= x 0) 0 y))

(test 0 (p))

А вот так не проще? проще.

(define (test) (if #t 0 (test)))

(test)

Ещё вариант теста:

(define (p) (p))

(define (test x) 0)

(test (p))

Данный пример является проверкой стратегии вычисления интерпретатора — является ли она нормальной или аппликативной. При аппликативной стратегии будет наблюдаться зацикливание (точнее, уход в рекурсию), при нормальной — вернётся результат 0.

# 5. Понятие вычислительного процесса. Итерация, линейная рекурсия, древовидная рекурсия

Каждый шаг работы интерпретатора — вычисление функций. Всё, что нужно вычислить, записывается в стек. Вычисления идут до тех пор, пока стек не станет пуст.

Простейшим примером рекурсии является **линейная рекурсия**, когда функция содержит единственный условный вызов самой себя. В таком случае рекурсия становится эквивалентной обычному циклу. Действительно, любой циклический алгоритм можно преобразовать в линейно-рекурсивный и наоборот. Наряду с линейной рекурсией, когда определение объекта включает в себя единственный аналогичный объект, существует еще и **древовидная** рекурсия, когда таких включаемых объектов несколько. Для рекурсивных функций это выглядит как два и более отдельных вызова функцией самой себя, либо как рекурсивный вызов функции в цикле.

# 6. Оценивание вычислительного процесса. Порядки роста

Оценка программы:

* Количество операций
* Максимальная глубина используемого стека.

Эти характеристики сильно зависят от исходных данных. Поэтому для оценки используется **порядок роста** — показатель, определяющий, как увеличивается количество операций и объём необходимой памяти по мере увеличения размеров данных, например:

* Логарифмический порядок
* Линейный порядок
* Полиномиальный (обозначается Р) порядок — такие задачи могут быть распараллелены.
* Экспоненциальный (обозначается NP) порядок

Существует также известная [проблема P=NP](http://www.google.com/url?q=http%3A%2F%2Fru.wikipedia.org%2Fwiki%2F%D0%A0%D0%B0%D0%B2%D0%B5%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%BE%D0%B2_P_%D0%B8_NP&sa=D&sntz=1&usg=AFQjCNHZHdZyV692egKQyQ1uzmNuOa6FSw).

**Пример.** Итеративное нахождение факториала. Расход памяти не зависит от глубины рекурсии.

(define (f n) (fiter 1 1 n))

(define (fiter p c m)

(if (> c m) p

(fiter (\* c p) (+ c 1) m))

)

Интерпретатор идентифицирует рекурсивную операцию как последнюю и не сохраняет точку возврата.

# 7. Понятие вычислительной модели. Требования к модели

**Вычислительная модель** – набор правил (соглашений, стандартов, формализмов) выполнения алгоритма.

Свойства математической модели:

* Дискретность шагов
* Детерминированность промежуточных и конечных результатов
* Элементарность шагов
* Результативность (в результате шага что-то меняется)
* Массовость (алгоритм работает с разными входными данными)

# 8. Детерминированная машина Тьюринга. Вычислимость по Тьюрингу. Тезис Тьюринга

**Детерминированная машина Тьюринга** (ДМТ) — имеет конечное число состояний и бесконечную ленту.

**Программа** для ДМТ — набор правил вида: q1a1 → q2a2D, где

q1 – текущее состояние

a1 – значение под головкой

q2 – новое состояние

a2 – новое значение под головкой

D – перемещение после выполнения операции

Остановка происходит, когда нет ни одного выполнимого правила.

Функция **вычислима по Тьюрингу**, если существует ДМТ, правильно считающая её значение.

Вход X **допустИм**, если ДМТ, получая этот вход, рано или поздно останавливается.

|  |  |
| --- | --- |
| **Сильный тезис Чёрча — Тьюринга (тезис Чёрча — Тьюринга — Дойча)**: любой конечный физический процесс, не использующий аппарат, *связанный с непрерывностью и бесконечностью*, может быть вычислен физическим устройством. | Конечный процесс → Устройство |
| **Физический тезис Чёрча — Тьюринга**: любая функция, которая может быть вычислена физическим устройством, может быть вычислена машиной Тьюринга. | Устройство → МТ |
| **Тезис Тьюринга**: всякий алгоритм может быть реализован с помощью машины Тьюринга. Вычислительные возможности ДМТ и CPU эквивалентны: на CPU можно написать эмулятор ДМТ, и наоборот, на ДМТ можно записать все состояния CPU. Любую математическую функцию можно запрограммировать. | Алгоритм → МТ |

# 9. Расширения машины Тьюринга

1. **Однонаправленная (односторонняя) МТ** – с лентой, ограниченной только с одной стороны.
2. **Многоленточная МТ** – на каждой ленте своя головка, в правилах присутствуют условия и действия для всех лент сразу. Эквивалентна ДМТ. q1a1b1c1 → q2a2b2c2DaDbDс
3. **Недетерминированная МТ**. Если существует пара «ленточный символ — состояние», для которой существует 2 и более команд, такая машина Тьюринга называется *недетерминированной*. Эквивалентна ДМТ. [выполняет N шагов, когда ДМТ потребуется c×eN]
4. **Рандомизированная МТ** – двухленточная МТ, одна из лент которой заполнена случайными числами (например, нулями и единицами). Некоторые алгоритмы работают медленнее, но некоторые — быстрее, например быстрая сортировка (со случайным выбором пропорции разбиения она стабильнее)

# 10. Примитивно рекурсивные функции Чёрча

(λ-исчисление) Все алгоритмы переписываются в виде только функций. Определение понятия **примитивно рекурсивной функции** является индуктивным. Оно состоит из указания класса базовых примитивно рекурсивных функций и двух операторов (*суперпозиции* и *примитивной рекурсии*), позволяющих строить новые примитивно рекурсивные функции на основе уже имеющихся. (Примитивно-рекурсивные функции – функции, которые можно построить на основе уже заданных. )

Базис примитивно-рекурсивной функции:

* Число 0
* Инкремент f(x)=x+1
* Функция проекции f(x1, …, xn) = xk

Операторы:

* Суперпозиция: <f, g1, …, gm> = f(g1, …, g m)



* Оператор примитивной рекурсиии <f, g, h> =
  + 
  + 

В данном определении переменную  можно понимать как счётчик итераций,  — как исходную функцию в начале итерационного процесса, выдающего некую последовательность функций  переменных, начинающуюся с , и  — как оператор, принимающий на вход  переменных , номер шага итерации , функцию  на данном шаге итерации, и возвращающий функцию на следующем шаге итерации.

Можно определить следующие функции так:

* Функция сложения:

sum(x, 0) = g(x) = x

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

* Функция умножения:

mul(x, y+1) = mul(x, y) + x

mul(x,0)=0

* Степень:

pow(x, 0) = 1

pow(x, y+1) = mul(pow(x,y), x)

* Декремент:

Dec(0) = 0

Dec(x + 1) = x

* Вычитание:

sub(x, 0) = x

sub(x, y+1) = dec(sub(x, y))

* Модуль разности:

m(x, y) = sum(sub(x,y), sub(y,x))

* Знак числа (на самом деле – ноль/не ноль):

sg(0) = 0

sg(y+1) = 1

* Минимум:

min(x,y) = sub(x, sub(x,y))

max(x,y) = sub(x,y)+y

# 11. Примитивно-рекурсивные операторы

См. билет 10 (суперпозиция, примитивная рекурсия)

# 12. Функция Аккермана и её смысл

Pn(a, x) — действие *n*-ной степени над а,х

P0(a, x) = a + x

P1(a, x) = a\*x

P2(a, x) = ax

Рассмотрим B(n, x) = Pn(2, x)

B(0, x) = 2 + x

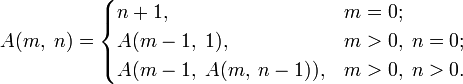
B(1, x) = 2∙x

B(n+1, 0) = sg(n)

B(n+1, x+1) = B(n, B(n+1, x))

Функция Аккермана: A(x) = B(x, x). Растёт быстрее любой рекурсивной функции. Нельзя вычислить через три базисные функции. Невычислима с помощью примитивно-рекурсивных операций => эта модель не полная.

Теперь из Википедии (на экзамене лучше говорить по лекциям): **Функция Аккермана** - простой пример всюду определённой вычислимой функции, которая не является примитивно-рекурсивной. Она принимает два неотрицательных целых числа в качестве параметров и возвращает натуральное число, обозначается . Эта функция растёт очень быстро, например, число  настолько велико, что количество цифр в порядке этого числа многократно превосходит количество атомов в наблюдаемой части Вселенной.



Функция Аккермана в силу своего определения имеет очень глубокий уровень рекурсии, что можно использовать для проверки способности компилятора оптимизировать рекурсию. Например, компилятор, который анализируя вычисление *A*(3, 30) способен сохранять промежуточные значения вроде *A*(3, *n*) и *A*(2, *n*), может ускорить вычисление *A*(3, 30) в сотни и тысячи раз. Также вычисление *A*(2, *n*) напрямую вместо рекурсивного раскрытия в *A*(1, *A*(1, *A*(1,...*A*(1, 0)...))) прилично ускорит вычисление. Вычисление *A*(1, *n*) занимает линейное время *n*. Вычисленние *A*(2, *n*) требует квадратичное время, т.к. оно раскрывается в O(*n*) вложенных вызовов *A*(1, *i*) для различных *i*. Вычисление *A*(3, *n*) требует времени пропорционально 4*n*+1.

*A*(4, 2) невозможно посчитать с помощью простого рекурсивного применения за разумное время. Вместо этого для оптимизации рекурсивных вызовов используются сокращенные формулы.

# 13. Частично-рекурсивные функции. Тезис Чёрча

**Ограниченный оператор наименьшего числа** даёт такую функцию, которая на входе принимает

P(x1, …, xn) и Z а на выходе даёт

* такое Y, при котором P(x1, …, xn, Y) = 0, или
* число Z, если такого Y не существует.

**Неограниченный оператор наименьшего числа**

В отличие от ограниченного оператора, в нём не задаётся Z. (который пробегает от 0 до z, т.е. бесконечный цикл). Вместо z задается условие.

**Частично-рекурсивные функции** – функции, построенные с помощью трёх базисных функций, оператора суперпозиции, оператора примитивной рекурсии, *а также ограниченного и неограниченного оператора наименьшего числа (оператора минимизации аргумента)*.

* **Оператор минимизации аргумента**. Пусть  — функция от *n* натуральных переменных. Тогда результатом применения оператора минимума аргумента к функции  называется функция  от  переменной, задаваемая следующим определением:

, при условии 

«То есть функция  возвращает минимальное значение последнего аргумента функции , при котором её значение равно 0. В терминах императивного программирования — примитивно рекурсивные функции соответствуют программным блокам, в которых используется только арифметические операции, а также условный оператор и оператор арифметического цикла (оператор цикла, в котором число итераций известно на момент начала цикла). Если же программист начинает использовать оператор цикла while, в котором число итераций заранее неизвестно и, в принципе, может быть бесконечным, то он переходит в класс частично рекурсивных функций.» ВП

Частично рекурсивные функции для некоторых значений аргумента могут быть не определены, так как оператор минимизации аргумента не всегда корректно определён, поскольку функция  может быть не равной нулю ни при каких значениях аргументов. С точки зрения императивного программирования, результатом частично рекурсивной функции может быть не только число, но и исключение или уход в бесконечный цикл, соответствующие неопределённому значению.

**Тезис Чёрча**: всякая вычислимая функция (т.е. вычисляемая некоторым алгоритмом) является частично-рекурсивной.

Частично-рекурсивные функции вычислительно эквивалентны машине Тьюринга.

# 14. Нумерация алгоритмов и вычислимых функций

«Множество всех алгоритмов счётно.» ©

«Любую программу можно представить как последовательность 16-ричных чисел, склеить их и получить число и рассматривать как входные данные другой программы» © С машиной Тьюринга аналогично: любому символу можно поставить в соответствие код, а количество состояний — конечно.

Одну функцию могут реализовывать несколько алгоритмов (например a+b и b+a). Множество всех вычислимых функций меньше множества всех алгоритмов ⇒ можно пронумеровать.

# 15. Теорема параметризации

Пусть *f(x,y)* вычисляется по алгоритму *Pf*. Для фиксированного *x=a* получаем функцию *g(y) = f(a, y)*, которая вычисляется по алгоритму *Pg* c индексом *k(x)*. Существует функция *k(x)*, которая по *х* даёт номер алгоритма Pg — т.е. на основе *х* автоматически строится алгоритм.

# 16. Понятие универсального алгоритма. Реализация на трёхленточной МТ

Универсальный алгоритм U(n, x): применяет к *x* алгоритм *Pn* и возвращает результат. Универсальный алгоритм существует! Строится трёхленточная МТ:

1. В начале входные данные, в конце — результат.
2. Текущее состояние МТ.
3. Правила МТ.

Если U существует, то мы можем с помощью одной вычислительной модели эмулировать любую другую МТ.

**Проблемы, связанные с остановкой**

Общерекурсивные программы не зацикливаются при любых аргументах. Зацикливающиеся программы не определены. Проблема — в невозможности перечисления общерекурсивных функций.

# 17. Проблема несчётности множества общерекурсивных функций

**Общерекурсивная функция** — частично рекурсивная функция, определённая для всех значений аргументов.

**Множество всех общерекурсивных функций несчётно.**

**<Доказательство>**

{f1, …, fn} – множество всех общерекурсивных функций. Построим функцию g(n) = fn(n)+1, g(n) отличается от любой общерекурсивной функции, однако сама является общерекурсивной, т.к. не зацикливается. Следовательно, множество всех общерекурсивных функций несчётно(их нельзя пронумеровать).

Пп. g(x)=fn(x) | ∀x ⇒ x=n

fn(n)+1 ≠ fn(n)

**</Доказательство>**

# 18. Проблема определения общерекурсивности

**Не существует алгоритма, который бы проверял, является ли поданная на вход функция общерекурсивной.**

**<Доказательство>**

Пусть g(m) =

* 1, если Pm общерекурсивна
* 0, если Pm необщерекурсивна

Построим функцию f(x), которая принимает себя на вход и прибавляет единицу.

f(x) =

* Px(x) + 1, если g(x) = 1 (Px общерекурсивна)
* 0, если g(x) = 0

При нашем предположении f(x) никогда не зацикливается. f(x) = / переобозначим/

* U(x, x) + 1, если g(x) = 1
* 0, если g(x) = 0

f(x) ≢ Pn(x) ← общ. рек. // f отличается от общерекурсивной

f(n) = U(n, n) + 1 = Pn(n) + 1 // f - общерекурсивна.

С одной стороны, f(x) общерекурсивная, с другой — отличается от общерекурсивной. Следовательно, предположение относительно g неверно, и функция g не существует.

**</Доказательство>**

# 19. Проблема самоприменимости

**Невозможно создать программу, которая определяет, зацикливается ли программа Pn, если ей на вход подать саму себя.**

Допустим, есть программа А, которая:

* Принимает на входе программу P
* Выполняет P(P)
* Если P(P) напечатала “Hello World”, выводит“yes”, иначе “Hello World”

Проблема:

* Вызываем А(А). Результат неизвестен. Такую программу реализовать нельзя.

**<Доказательство>**

Путь существует функция f(n) =

* 1, если Pn самоприменима
* 0, если Pn не самоприменима

Отправим ей на вход G(n) :=

* 0, если f(n) = 0
* цикл., если f(n) = 1

Если g самоприменима, то f=1 и g должна циклиться.

Предположение неверно, f(n) не существует.

**</Доказательство>**

# 20. Проблема остановки

**Невозможно создать программу, которая по данному алгоритму *Р* и аргументу *x* определяет, зациклится ли *P(x)*.**

**<Доказательство>**

Пп. Существует g(x,y) =

* 1, если Px(y) не зацикливается
* 0, если Px(y) зацикливается

f(x)=g(x,x) По предыдущему, предположение неверно.

**</Доказательство>**

# 21. Необходимость частично-рекурсивной функции

Существует частично-рекурсивная (т.е. зацикливающаяся при некоторых *х*) функция *f(x)*, такая, что любая общерекурсивная функция *g(x)* тождественно не равна *f(x)*.

*g(x)* никогда не зацикливается. Но для *x*, при которых *f(x)* не зацикливается, *g(x)* возвращает *f(x)*.

**<Доказательство>**

Предположим, что g(x)≡f(x). Если f(x) не зацикливается, то

f(g)=U(g,g)+1=g(g)+1.

**</Доказательство>**

**Итоги**

1. Общерекурсивные функции перечислить нельзя.
2. Анализировать программы на зацикливание нельзя (на этапе компиляции), даже на конкретных данных.
3. Отказаться от зацикливающихся (частично-рекурсивных) программ нельзя – они иногда требуются.

# 22. Теорема Райса (классификация алгоритмов)

**Нетривиальное свойство алгоритма** – принадлежность его к некоторому классу алгоритмов.

**Всякое нетривиальное свойство алгоритма неразрешимо алгоритмически.**

А теперь из Википедии: «Для любого нетривиального свойства вычислимых функций определение, вычисляет ли произвольный алгоритм функцию с таким свойством, является алгоритмически неразрешимой задачей. Здесь свойство называется нетривиальным, если существуют и вычислимые функции, обладающие этим свойством, и вычислимые функции, не обладающие им.»

# 23. Проблема соответствий Поста (поиск конкатенации)

Пусть даны два набора строк:

* a1, a2, …, an
* b1, b2, …, bn

Из строк первого набора конкатенацией элементов с индексами i1, i2, … iK строится строка A, из строк второго набора с теми же индексами строится строка B. Требуется определить, существует ли такой набор индексов, при котором А=B.

**Такую программу написать нельзя**. (Потому что если такого набора индексов не существует, программа не сможет этого понять и будет пытаться найти последовательность всё длиннее и длиннее - *прим. ред.*)

Всё до этого было связано с анализом программ, а это - анализ данных. Но написать такую программу мы тоже не можем. Если требуется доказать, что некоторая проблема неразрешима, то надо её свести к существующей неразрешимой проблеме.

**Механизмы абстракций**

**Механизмы абстракций** — отделение логики использования от логики реализации.

Использование абстракций должно быть проще реализации. *Нет функциям с 50-ю параметрами!*

Абстракции 2-х типов:

* **Абстракции функций**. Например, блочная структура, особые формы lambda, let.
* **Абстракция данных** (как устроено внутри не волнует). Например, пара значений.

# 24. Труднорешаемые проблемы

Труднорешаемые проблемы - это проблемы, которые имеют экспоненциальный рост (NP сложные). К ним относятся разнообразные переборы. Для того, чтобы доказать, что задача является труднорешаемой, её нужно свести к уже известной труднорешаемой проблеме.

# 25. Особая форма лямбда (lambda)

(lambda (аргументы) выражение) — результатом является функция, которая принимает аргументы и вычисляется по указанному выражению.

(define (f x) (+ x 10)) ≡ (define f (lambda (x) (+ x 10)))

передаем функцию

(sum (lambda (x) (+ x 20)) 30)

Лямбда позволяет создавать функции, которые в качестве результата возвращают другие функции.

(define (multiplier n) (lambda (x) (\* x n)))

Вызывем. Умножает один аргумент на 10

(sum (multiplier 20) 10)

(define (multiplier n)

(define (mul x) (\* x n))

mul)

((multiplier 2) 3)

# 26. Особая форма let. Сравнение let и define

Позволяет заводить внутренние локальные переменные

(let (

(переменная\_1 выр\_1)

(переменная\_2 выр\_2)

(переменная\_3 выр\_3)

)

выр

)

В выр можно использовать все объявленные переменные.

Например,

(define (f x)

(let ((y (\* x x)))

( / y (+ y 1))

)

)

# 27. Реализация особой формы let

Перепишем оператор let в виде лямбда-выражения:

( **(lambda ( переменная\_1 переменная\_2 ...) выр)** выр\_1 выр\_2 ...)

# 28. Числа Чёрча

**Числа Чёрча** — это реализация натуральных чисел. Требуется 0 и инкремент.

(define zero

(lambda (f)

(lambda (x) x))) ; Это возвращает f(x)

(define (inc n)

(lambda (f)

(lambda (x) (f ((n f) x))) ; Это возвращает f(n)

))

0: zero = (lambda (f) (lambda (x) x))

1: (inc zero) = (lambda (f) (lambda (x) (f x)))

2: (inc (inc zero)) = (lambda (f) (lambda (x) (f (f x))))

(define one (inc zero))

(define two (inc one))

(one f) → возвращает → g(x)=f(x)

(two f) → g(x)=f(f(x))

1: (lambda (f)(lambda (x)(f x)))

2: (lambda (f)(lambda (x)(f (f x))))

3: (lambda (f)(lambda (x)(f (f (f x)))))

4: (lambda (f)(lambda (x)(f (f (f (f x))))))

5: (lambda (f)(lambda (x)(f (f (f (f (f x)))))))

6: (lambda (f)(lambda (x)(f (f (f (f (f (f x))))))))

7: (lambda (f)(lambda (x)(f (f (f (f (f (f (f x)))))))))

8: (lambda (f)(lambda (x)(f (f (f (f (f (f (f (f x))))))))))

9: (lambda (f)(lambda (x)(f (f (f (f (f (f (f (f (f x)))))))))))

10: (lambda (f)(lambda (x)(f (f (f (f (f (f (f (f (f (f x))))))))))))

11: (lambda (f)(lambda (x)(f (f (f (f (f (f (f (f (f (f (f x)))))))))))))

12: (lambda (f)(lambda (x)(f (f (f (f (f (f (f (f (f (f (f (f x))))))))))))))

13: не влезает на строку. А до этого момента выучить обязательно!!!111111111111

# 29. Пары значений. Реализация пар значений

**Абстракция данных**

Пара значений: (a b) = p

Операции:

* создание пары: (cons a b)
* первый элемент: (car p)
* второй элемент: (cdr p)
* является ли парой: (pair? p)

Выполняются два соотношения (*К.О. уведомляет*):

* (car (cons a b)) → a
* (cdr (cons a b)) → b

**Реализация пары на Lisp**

(define (cons x y) - создает пару

(define (dispatch m) - если 0, то возвращает *х*, иначе *у*

(if (= m 0) x y))

dispatch)

(define p (cons x y))

(define (car p) (p 0)) - в данном случае р - это как раз возвращенная функция от cons

(define (cdr p) (p 1))

Можно написать вот так:

(define (cons a b) - cons возвращает функцию, которая будет принимать один параметр m

(lambda (m)

(if (= m 0) a b)))

# 30. Вертикальные барьеры абстракции. Конструкторы. Селекторы.

**Барьер абстракций** — это набор функций, который позволяет работать с конкретным типом данных.

Для пары это : cons, car, cdr.

**Функции-селекторы**: car, cdr — позволяют получить элемент из структуры (в данном случае из пары)

**Функция-конструктор**: cons

Любой тип данных — это набор операций, это наш барьер абстракций.

*Нам не важно, что стоит за функциями, они есть и все, дальше нам разбираться не надо.*

# 31. Списки. Основные функции для работы со списками. Реализация списков.

(a (b (c nil)))

(cons a (cons b (cons c nil))))

Так писать неудобно, поэтому:

(list a b c …)

(list a b c) – здесь a,b,c вычисляются

`(a b c) – здесь a,b,c не вычисляются (для тех, кто забыл — это особая форма quote)

Функции такие же, как для пары: car, cdr, list?.

# 32. Реализация вспомогательных функций для работы со списками (обращение по индексу, длина, мэпинг)

Получает в списке *items* *n*-й элемент:

(define (list-ref items n)

(if (= n 0)

(car items)

(list-ref (cdr items)(- n 1))

)

)

Получить длину списка:

(define (length items)(if (**null?** items) 0 (+ 1 (length (cdr items)))))

Применение функции ко всем элементам (*мэпинг* — отображение):

(define (map proc items)

(if (null? items)

nil

(cons (proc (car items))

(map proc (cdr items))

)

)

)

# 33. Горизонтальные барьеры абстракции. Диспетчеризация по типу

**Горизонтальные барьеры** отделяют «высокоуровневые» операции от «низкоуровневых» представлений.

Концептуальный уровень <-> Логический уровень <-> Физический уровень.

*Главное, что нет вызовов физического из концептуального уровня.*

**Вертикальный барьер** даёт нам возможность отдельно разрабатывать и добавлять альтернативные представления. У одной и той же логики может быть несколько реализаций. Функционально они одинаковы, но по-разному работают. Необходимо поддерживать несколько реализаций одного интерфейса. Подходы:

* Диспетчеризация по типу
* Программирование, управляемое данными

**Диспетчеризация по типу**

Рассматриваем пример с точкой на плоскости. Будем использовать две системы координат: декартову и полярную. У функций для декартовых координат будет префикс d\_, а у полярных — p\_. Создаём соответствующие конструкторы, которые внутри автоматически пересчитывают в нужные координаты:

* d\_cons(x,y)
* d\_cons(r,ϴ)

И дополнительные методы:

* d\_get\_x
* d\_get\_y
* d\_get\_r
* d\_get\_ϴ

Такой же набор функций для точки в полярной системе координат, но реализация другая.

Любые данные (точки) представляются в виде:

(метка (b c))

Метка показывает, в каком виде хранится пара, декартовая или полярная система.

(define (get\_x point)

(if (eq? (car point) `dec)

(d\_get\_x (cdr point))

(p\_get\_x (cdr point))

)

)

# 34. Программирование, управляемое данными

|  |  |  |
| --- | --- | --- |
|  | Операции ↓ |  |
| Реализации → | d\_x | d\_y |
|  | p\_x | p\_y |

Занести в таблицу значений: (put операция код\_реализации функция)

Возвращает нужную функцию:(get op type)

(define (install decart)

(put `get\_x `dec d\_gex\_x)

(put `get\_y `dec d\_gex\_y)

)

(define (install decart)

(define …)

(define …)

)

(define (get\_x point)

(

(get `get\_x (car point))

(cdr point)

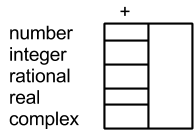
)

)

*В этом и в предыдущем методе функция-селектор делает столбик таблицы, а install делает строчку. Это как управление сообщениями. Каждая функция соответствует типу.*

# 35. Система с обобщёнными операциями

Обобщённые — это значит, что может принимать аргументы разных типов.



*В код зашиваем еще и информацию, какие аргументы. Тип первого аргумента соответствует первому столбцу.*

|  |  |  |  |
| --- | --- | --- | --- |
| n+i | i+i | n+r | ... |
|  |  |  |  |

# 36. Приведение типов. Реализация

Добавляются столбцы, которые реализуют приведение. Сколько типов, столько и столбцов.

|  |  |  |
| --- | --- | --- |
| + | coercion-int | ... |
|  | n → i | ... |
|  | i → i | ... |
|  | r → i | ... |

|  |  |
| --- | --- |
| a+b: если типы одинаковые, то вычисляем. Иначе приводим. Когда два аргумента, можем жёстко прописать, что именно к чему приводить.  Иерархия типов. Если не можем выполнить приведение, переходим к надтипу.  Для каждой операции определяется набор допустимых значений типов аргументов и правил получения типа результат. Используется иерархия типов. |  |

# 37,38. Моделирование объектов.Особые формы set! и begin. Моделирование простых объектов

**Моделирование внешних объектов.**

*Нужна операция для изменения состояния объекта - это какая-то простая\составная информация.*

*Какое-то время выполняется процесс. Время жизни объекта никак не связано со временем процесса.*

Моделирование объектов всегда требует операцию присваивания.

Особая форма set: (set! <что> <как>)

(begin <op1>.. <opn>) набор операций, которые надо выполнить; возвращает результат их выполнения.

(define balance 100)

(define (withdraw amount)

(if (<= amount balance)

(begin

(set! balance (- balance amount))

balance

)

“No money, no honey”)

)

Надо спрятать баланс

(define (new-withdraw)

(let ((balance 100))

(lambda (amount)

(if <= amount balance)

(begin

(set! balance (- balance amount))

balance

)

“No money, no honey”)))

))

(define (make-withdraw balance)

(lambda (amount)

(if (>= balance amount)

(begin (set! balance (- balance amount))

balance)

"Недостаточно денег на счете")))

При помощи make-withdraw можно следующим образом создать два объекта W1 и W2:

(define W1 (make-withdraw 100)) - счет1

(define W2 (make-withdraw 100)) - счет 2

(W1 50)

50

(W2 70)

30

Реализуем счет с тремя операциями: get/check - сколько денег

withdraw

add

(define (make balance)

(define (get) balance)

(define (withdraw amount) … )

(define (add amount) … )

(define (dispatch m)

(cond

((= m ‘get) get)

((= m ‘withdraw) withdraw)

((= m ‘add) add)

(else “error”)

)

)

ditpatch

)

Call:

(define a (make 100))

((a `add) 50)

((a `withdraw) 20)

(a `get)

# 39. Моделирование сложных объектов

Ха, а модель не поменялась. Каждый объект — абстракция (набор функций: конструктор, селектор, и вводится мутатор). Отличие от простых объектов в том, что в сложных объектах более одного поля, и соответственно больше селекторов и методов.

(make `get)

(make `withdraw)

(define p (cons a b))

(set! p ...)

(set! p (cons (car p) new\_b))

Механизм изменяемые пары:

(define (cons x y)

(define (set\_x value)...)

(define (set\_y value) (set! y value))

(define (get\_x ) x)

(define (dispatch m)

(cond

((= m ‘get\_x) get\_x)

((= m ‘set\_x) set\_x)

…

(else “error”)

)

)

dispatch

)

(define (car p) ((p ‘get\_x)))

(define (set\_car! p value)

((p ‘set\_x) value)

)

# 40. Оператор присваивания. Достоинства и недостатки его использования

**Императивные языки** — те, где есть оператор присваивания.

* Достоинства:
  + простота и привычность
  + генератор случайных чисел (внутренняя переменная, которую пересчитываем)

(define rand

(define x (rand\_init))

(lambda ()

(set! x (rand\_update))

)

)

* Недостатки:
  + нужна формальная модель вычисления
  + для каждой переменной надо знать значение до и после set. *Не подходит модель аппликативная и нормальная. Нужна другая.*

(a (b value) (c value)) не понятно, какое будет value.

* + усложняется сравнение данных

(define a (make 100))

(define b (make 100))

(= a b) Сравнивать объекты или содержимое -?

* + важен порядок действий. Можно вычислять аргументы параллельно, поэтому нужна синхронизация потоков.

# 41. Модель вычисления с окружениями

[ тут понятнее <http://newstar.rinet.ru/~goga/sicp/sicp.pdf> стр 227 ]

Главный принцип - необходимо хранить значение переменной, а не вычислять каждый раз заново.

Окружение — набор кадров

Кадр — таблица связываний

[небесной красоты картинка]

Есть окружение более высокого уровня.

При запросе переменной она ищется сначала в последнем кадре, потом в родительском. Новый кадр создается в конце.

Процедура = пара:

* Ссылка на окружение, где была создана эта процедура
* Тело процедуры

(define x 10) ; создается новый кадр

(define (f x) 0)

(define (lambda (x) 0))

[еще картинка]

Когда (f 10) создается новое окружение, у которого в качестве родительского окружения указывается окружение, в котором создана функция, … .

Окружение живет, пока на него существуют ссылки (*привет, garbage collector*)

(define (h x) (lambda))

# 42. Моделирование потоков

Параллелизм:

(parallel-execute p1 p2 p3 …)

Сериализация доступа:

(make-serializer) - создание сериализатора, который может создать функцию f(x), гарантирующую, что в один момент времени будет выполняться только одна из них. Иными словами, на входе функция, на выходе lock-функция.

Пример:

(define s (make-serializer)) — создание сериализатора

(define s-put (s put)) — создание lock-функции s-put на основе функции put c помощью cериализатора s.

Семафоры:

(make-mutex) - аналогично.

(define m (make-mutex))

(m ‘acquire) - залочить

(m ‘release) - освободить

Реализация make-serializer:

(define (make-serializer)

(let (m (make-mutex)))

(lambda (p)

(define (serialized-p .args)

(begin (m ‘acquire)

let ((val p .args)))

(begin (m ‘release)

val))))

serialized-p)))

.args - для тех случаев, когда количество параметров четко не определенно.

Поток - упорядоченное множество данных небольшого объема (заявок). Заявки передаются из внешней среды в приложение, а приложение возвращает результат. (От меня) По сути, поток в лиспе - это список, у которого определен только текущий элемент. Остальные элементы пока неизвестны. Они находятся с помощью функции.

Операции работы с потоками (с префиксом stream-):

* Проверка на пустоту: (stream-null? S)
* Получение первого элемента: (stream-car S)
* Получение продолжения: (stream-cdr S)
* Создание потока: (cons-stream x y)
* Пустой поток (не операция): the-empty-stream

Поток:

( элемент ; функция, возвращающая продолжение )

↓

( элемент ; функция, возвращающая продолжение )

↓

и т.д.

Функции:

* Получение n-го элемента потока

(define (stream-ref s n)

(if (= n 0) (stream-car s)

(stream-ref (stream-cdr s) (- n 1))))

* Маппинг

(define (stream-map s p)

(if (stream-null? s)

the-empty-stream

(cons-stream

(p (stream-car s)

(stream-map (stream-cdr s) p)

)

)

)

Работа с потоками (реализация):

Для начала, нужны две функции:

(define (delay x) (lambda () x)) - создает отложенное вычисление функции

(define (force x) (x)) - вычисляет функцию

(define (cons-stream x y))

(cons x (delay y)))

(define (stream-car s) (car s))

(define (stream-cdr s) (force (cdr s)))

Примеры потоков:

* цифорки :3

(define (integers n)

(cons-stream n

(integers (+ n 1))))

* случайные цифорки :3

(define random

(cons-stream random\_init

(stream-map rand-update random)))

Недостатки:

1. Не понятно, как представить задачу в виде потока.
2. Параллелизм: неизвестно, какому потоку отдать приоритет.

Мне печеньки за последний билет ♥ ты супер :\*