Федеральное агентство по образованию РФ

ГОУ ВПО Пермский государственный технический университет

кафедра ИТАС

КУЗНЕЦОВ Д.Б.

КОНСПЕКТ ЛЕКЦИЙ

ПО ДИСЦИПЛИНЕ

Функциональное и логическое программирование

Для студентов направления

«Информатика и вычислительная техника» (230100)

Специальности АСОИУ, ЭВТ, ПОВТ

дневной, заочной и дистанционной форм обучения

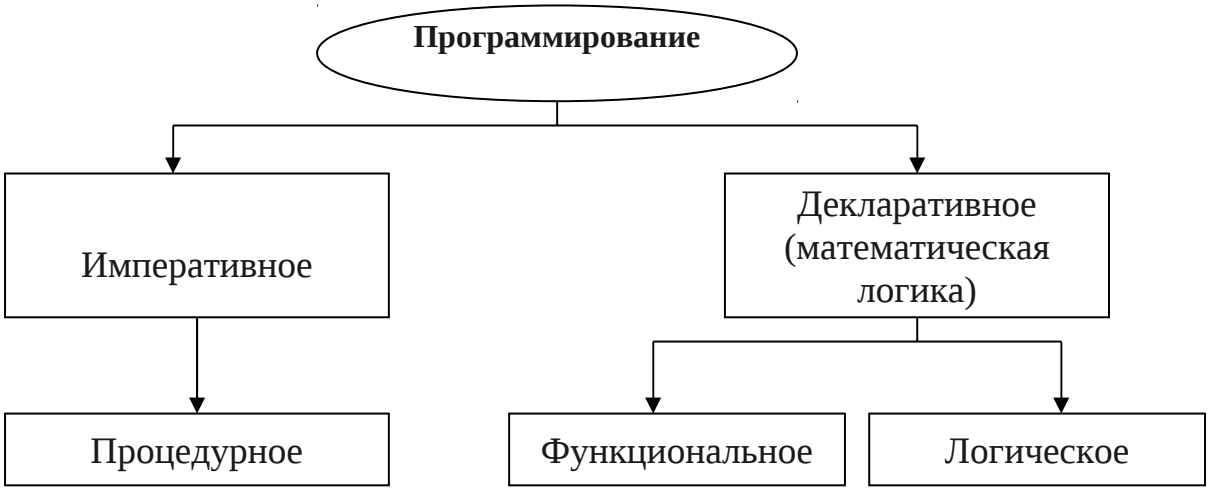
2008

Составители: Д.Б. Кузнецов

Конспект лекций по дисциплине «Функциональное и логическое программирование»: для студентов направления «Информатика и вычислительная техника» (230100), специальностей АСОИУ, ЭВТ, ПОВТ дневной, заочной и дистанционной форм обучения / ПГТУ. – Пермь: 2008.

утверждено на заседании кафедры ИТАС 6 февраля 2008г. протокол №20/ ПГТУ. – Пермь: 2008.

Виды программирования



История науки

Математическая логика, возникшая почти 100 лет назад в связи с внутренними потребностями математики, нашла применение в теоретическом (и практическом) программировании и, судя по всему, взаимодействие этих двух наук в недалеком будущем сможет принести новые плоды.

Причины, по которым программисты обратились к математической логике, а логики заинтересовались программированием, имеют довольно глубокие корни. Следует, наверное, вспомнить, что математическая логика занимается построением формальных языков, предназначенных для представления таких фундаментальных понятий, как функция, отношение, аксиома, доказательство, и изучением основанных на этих языках логических и логико-математических исчислений. Именно в недрах математической логики были найдены математически точные понятия алгоритма и вычислимой функции, развита семантика формальных языков и теорий, построены системы логического вывода – и все это, заметим, было сделано в 30-40-х годах, т.е. еще в “докомпьютерную эру”.

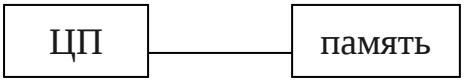
Программирование также имеет дело с формальными языками – языками программирования, являющимися средствами общения человека и компьютера. В своем стремлении сделать эти средства удобными и естественными для человека вычислительная наука не могла пройти мимо тех языков описания вычислимых функций и отношений, которые были созданы в рамках математической логики с учетом многовекового математического опыта. В результате появились принципиально новые языки функционального и логического программирования: Лисп, Рефал, ML, Миранда, Пролог и другие – особенно активно разрабатываемые в последнее десятилетие. С другой стороны, языки высокого уровня нуждаются в детально проработанной семантике, которая выходит на первый план при написании трансляторов. И здесь идеи и аппарат математической логики, уже занимавшейся проблемами семантики, оказались весьма кстати. Поначалу для формализации смысла программ использовалось известное логикам еще с 30-х годов λ-исчисление А.Чёрча, а затем абстрактным базисом денотационной

семантики стала мощная и элегантная теория областей, развитая Д.Скоттом. Важное значение приобретает теория логического вывода, причем не только в задачах искусственного интеллекта, но и как средство рассуждения о программах и доказательства их свойств (скажем, правильности относительно спецификации) или, наоборот, как метод построения правильных программ (например, путем их извлечения из конструктивных доказательств).

В заключение, приведем цитату Д.Гриса *«Мне лучше всего было бы пройти хороший курс логики десять лет назад»*.

Введение

Традиционные процедурные языки предназначаются для описания процессов вычислений на Фоннеймоновской машине.



Каждое действие программы – это изменение оперативной памяти.

Мы описываем не формулы для решения задачи, мы описываем процесс, т.е. как задача должна решаться. Подобные программы сложны для понимания и доказательств.

Функциональное программирование было задумано как альтернатива процедурному программированию (в конце 50-х годов).

Функциональная программа представляет собой некоторое выражение (в математическом смысле); выполнение программы означает вычисление значения этого выражения. Другими словами, функциональные программы соответствуют математическим объектам (позволяют производить строгие суждения – правильно/неправильно).

Результат работы императивной (процедурной) программы полностью и однозначно определен ее входом, можно сказать, что финальное состояние (или любое промежуточное) представляют собой некоторую функцию (в математическом смысле) от начального состояния, т.е. $\sigma' = f(\sigma)$, где σ' - новое значение состояния после завершения программы, включающее в себя то, что можно рассматривать как “результат” работы программы. Во время исполнения каждая команда изменяет свое состояние; следовательно, состояние проходит через некоторую конечную последовательность значений: $\sigma = \sigma_0 \to \sigma_1 \to \sigma_2 \to \sigma_3 \to ... \to \sigma_n = \sigma'$.

В функциональном программировании используется именно такая точка зрения: программа представляет собой выражение, соответствующее функции f. Функциональные языки программирования поддерживают построение таких выражений, предоставляя широкий выбор соответствующих языковых конструкций.

Рассмотрим кратко языки функционального программирования:

- ✓ Lisp (Лисп) – базируется на λ- исчислении (определенная формализация понятия функции): для обработки списков (используют в графических и текстовых редакторах), для задач искусственного интеллекта;
- ✓ Рефал – базируется на нормальных алгорифмах Маркова;

- ✓ Миранда – комбинаторная логика ⇒ Haskell (Haskell-98).

Процесс написания программ сродни математическому мышлению. Функциональный подход имеет ряд преимуществ перед императивным. Как было сказано выше, функциональные программы более непосредственно соответствуют математическим объектам, и следовательно, позволяют проводить строгие рассуждения. Установить значение императивной программы, т.е. той функции, вычисление которой она реализует, может оказаться довольно трудно. Напротив, значение функциональной программы может быть выведено практически непосредственно.

Рассмотрим на примере: Нахождение факториала на языке функционального и процедурного программирования:

$$n!: n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot 1;$$

$$fact(n): \begin{cases} 1, n: 0; \\ n\cdot fact(n-1), n>0. \end{cases}$$

1) язык Си (процедурный):

```
int fact (int n)
{
int x = 1;
while (n > 0)
{
x = x * n;
n = n - 1;
}
return (x);
}
```

2) язык Haskell (функциональный):

```
fact n =
       if n = = 0 then 1
       else n * fact(n - 1)
```

Практически сразу видно, что программа на языке Haskell соответствует частичной функции нахождения факториала. Однако, для программы на языке Си это соответствие не очевидно.

Сравним *функциональный* и *императивный* подходы, заметив следующие свойства:

Программирование	
Процедурное	Функциональное
1) переменные	
Именованные ячейки памяти, изменяемые константы.	Настоящие переменные (в частности не используется оператор присваивания).
2) циклы	
Циклы есть.	Не может быть циклов (нет счетчика); используются рекурсивные функции вместо циклов.
3) программа – это …	
Последовательное изменение памяти.	Вычисление функции. Последовательное выполнение команд бессмысленно, поскольку одна команда не может повлиять на выполнение следующей (т.к. негде сохранять промежуточные значения).
4) функции	
При вызове функции возможны побочные эффекты (два вызова одной о той же функции с одними и теми же аргументами могут дать разные результаты; изменение значений глобальных переменных).	Функции используются широко, могут даже использоваться в качестве аргументов для других функций и возвращать в качестве результата; побочных эффектов нет.

Из этого следует, что вычисление любого выражения не может иметь никаких побочных эффектов, и значит, порядок выполнения его подвыражений не оказывает влияния на результат. Таким образом, функциональные программы легко поддаются распараллеливанию, поскольку отдельные компоненты выражений могут вычисляться одновременно.

λ- исчисление

Подобно тому, как теория машин Тьюринга является основой императивных языков программирования, λ- исчисление служит базисом и математическим “фундаментом”, на котором основаны все функциональные языки программирования.

λ- исчисление было изобретено в начале 30-х годов логиком А.Чёрчем, который надеялся использовать его для формализма для обоснования математики. Вскоре были обнаружены проблемы, делающие невозможным его использование в

этом качестве (сейчас есть основания полагать, что это не совсем верно) и λ-исчисление осталось как один из способов формализации понятия алгоритма.

В математике, когда необходимо говорить о какой-либо функции, принято давать этой функции некоторое имя и впоследствии использовать его, как, например, в следующем утверждении:

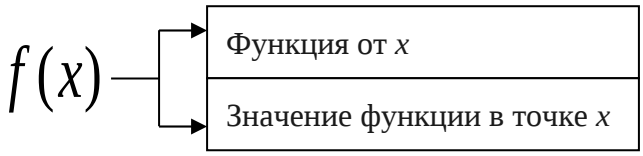
Пусть $f : R \to R$ определяется следующим выражением:

$$f(x) := \begin{cases} 0, x = 0; \\ x^2 \cdot \sin\frac{1}{x^2}, x \neq 0. \end{cases}$$

Тогда $f'(x)$ не интегрируема на интервале $[0,1]$.

Многие языки программирования также допускают определение функций только с присваиванием им некоторых имён. Например, в языке Си функция всегда должна иметь имя. Это кажется естественным, однако поскольку в функциональном программировании функции используются повсеместно, такой подход может привести к серьёзным затруднениям. Представьте себе, что мы должны оперировать с арифметическими выражениями в подобном стиле:

Пусть $x = 2$ и $y = 4$. Тогда $xx = y$.



λ- нотация позволяет определять функции с той же лёгкостью, что и другие математические объекты.

λ- выражением будем называть конструкцию вида: $\lambda x.f(x)$

Использование λ- нотации позволяет чётко разделить случаи, когда под выражением типа $f(x)$ мы понимаем саму функцию f и ее значение в точке x. Кроме того, λ-нотация позволяет формализовать практически все виды математической нотации. Если начать с констант и переменных и строить выражения только с помощью λ- выражений и применений функций к аргументам, то можно представить очень сложные математические выражения.

Пример:

$\lambda x.x^2$ - представляет собой функцию, возводящую свой аргумент в квадрат.

$\lambda x.2x + a$ - от a не зависит.

Язык λ- исчисления:

λ - абстракция

$x_1, x_2, ..., x_n$ - символы переменных

$c_1, c_2, ...$ - символы констант

$*$ - аппликация $M * N$ (применение аргумента к функции); $M(N)$ - математическая запись аппликации. Символ $*$ иногда опускают, т.е. MN.

λ-терм:

1) переменная или константа;
2) $\lambda x.M$ **,** где x – переменная, M - λ-терм;
3) $M * N$ - λ-терм, если M и N - λ-термы

Правила вывода:

- Если M=N, то N=M (симметричность);
- Если M=N, N=P, то M=P (транзитивность);
- Если M=N, то MP=NP;
- Если M=N, то $\lambda x.M = \lambda x.N$;

Конверсия

λ- исчисление основано на операциях конверсии, которые позволяют переходить от одного терма к другому, эквивалентному ему. По сложившейся традиции эти конверсии обозначают греческими буквами $\alpha\beta$. Они определяются следующим образом:

α-конверсия: $\lambda x.M = (\lambda y.M)^x_y$ - x заменяем на y (можем в функции менять имя переменной).

Пример:

$\lambda x.2x + a = (\lambda y.2y + a)$

β-конверсия: $(\lambda x.M)N = M^x_N$ - все вхождения x меняем на N.

$(\lambda x.M)N = P$ - β-редекс

$M_N = Q$

Преобразование P в Q – это β-редукция: $P \underset{\beta}{\triangleright} Q$

Редукция – “уменьшение” (в большинстве случаев λ-терм укорачивается).

Примеры:

1) $(\lambda x.x + 8)(9) \underset{\beta}{\triangleright} 9 + 8;$
2) $(\lambda x.x + 8)(a) \underset{\beta}{\triangleright} a + 8;$
3) $(\lambda z.x + y + z + t)(a) \underset{\beta}{\triangleright} x + y + a + t;$

β-нормальная форма – это λ-терм без редексов (когда выполнены все β-редукции).

4) $(\lambda x.((\lambda y.xy)u))(\lambda v.v) \underset{\beta}{\triangleright}$
   $(\lambda x.(xu))(\lambda v.v) \underset{\beta}{\triangleright}$
   $(\lambda v.v)u \underset{\beta}{\triangleright} u;$
5) $(\lambda x.((\lambda y.xy)u))(\lambda v.v) \underset{\beta}{\triangleright}$

$(\lambda y.(\lambda v.v)y)u) \underset{\beta}{\triangleright}$

$((\lambda y.y)u) \underset{\beta}{\triangleright} u;$

Теорема Чёрча-Россера для β-редукции:

β-нормальная форма единственна.

Если $P \underset{\beta}{\triangleright} M$, $P \underset{\beta}{\triangleright} N$, то $\exists T$: $M \underset{\beta}{\triangleright} T$, $N \underset{\beta}{\triangleright} T$.

6) $(\lambda x.xx)(\lambda x.xx) \underset{\beta}{\triangleright} (\lambda x.xx)(\lambda x.xx);$

Но, не для каждого λ-терма существует β-нормальная форма.

Поправка: Теорема верна, если для λ-терма существует β-нормальная форма.

Операция каррирования позволяет записать функции нескольких аргументов в обычной λ-нотации. Идея заключается в том, чтобы использовать выражения вида $\lambda xy.x+y$. Такое выражение можно рассматривать как функцию $R \to (R \to R)$, т.е. если его применить к одному аргументу, результатом будет функция, которая затем принимает другой аргумент. Таким образом: $(\lambda xy.x+y)12 = (\lambda y.1+y)2 = 1+2$.

$\lambda x.(\lambda y.E) \equiv \lambda xy.E$ - важен порядок параметров.

$*(x+1) = 8$ - 8 записывается в следующую ячейку за x ($x[1] = 8$ - более понятный вариант).

Упражнение: умножение на 3 через операцию сложения

$a * 3$

$(a+(a+a))$

$(\lambda x.a+x)(a+a)$

$(\lambda x.a+x)((\lambda x.a+x)a);$

$(\lambda z.(\lambda yt.y(yt))(\lambda x.z+x)(z)) \underset{\beta}{\triangleright}$

$(\lambda z.(\lambda x.z+x)(\lambda x.z+x)(z)) \underset{\beta}{\triangleright}$

$(\lambda z.z+((\lambda x.z+x)(z))) \underset{\beta}{\triangleright}$

$(\lambda z.z+(z+z)).$

Свободные и связанные переменные

Переменные в λ-выражениях могут быть *свободными* и *связанными*. В выражении вида x^2+x переменная x является свободной; его значение зависит от значения переменной x и в общем случае ее нельзя переименовать. Однако в таких выражениях как $\sum_{i=1}^{n} i$ или $\int_0^x \sin y dy$ переменные i и y являются связанными; если вместо i везде использовать обозначение j, то значение выражения изменится.

Следует понимать, что в каком-либо подвыражении переменная может быть свободной (как в выражении под интегралом), однако во всём выражении она связана какой-либо *операцией связывания переменной*, такой как операция

суммирования. Та часть выражения, которая находится “внутри” операции связывания, называется *областью видимости* переменной.

$x \to M$

$\lambda x.M$

Любое вхождение переменной x в терм $\lambda x.M$ называется связанным (по средствам λx).

Пример:

$\lambda x.x + y$, x – связанная переменная; y – свободная переменная.

Комбинаторы

Терм без свободных вхождений переменных, называется *замкнутым* или *комбинатором*. Замкнутый терм имеет фиксированный смысл независимо от значения любых переменных.

В теории комбинаторов установлено, что с помощью нескольких базовых комбинаторов и переменных можно выразить любой терм без применения операции λ-абстракции.

1	$I = \lambda x.x$	тождественный комбинатор	$IX \underset{\beta}{\triangleright} X$
2	$B = \lambda xyz.x(yz)$	композиция	$BFGX \underset{\beta}{\triangleright} F(GX)$
3	$C = \lambda xyz.xzy$	коммутатор	$CFXY \underset{\beta}{\triangleright} FYX$
4	$K = \lambda xy.x$	образование константы	$KXY \underset{\beta}{\triangleright} X$
5	$W = \lambda xy.xyy$	дублирование	$WFX \underset{\beta}{\triangleright} FXX$
6	$S = \lambda xyz.xz(yz)$	подстановка и композиция	$SFGX \underset{\beta}{\triangleright} FXGX$
7	$y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$	комбинатор неподвижной точки (для организации рекурсии)	$Yf \underset{\beta}{\triangleright} f(yf)$

Пример:

$(\lambda x.x)(8) \underset{\beta}{\triangleright} 8$

$I8 \underset{\beta}{\triangleright} 8$

} тождественный комбинатор

$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(z) \underset{\beta}{\triangleright}$

$(\lambda x.z(xx))(\lambda x.z(xx)) \underset{\beta}{\triangleright}$

$z(\lambda x.z(xx)(\lambda x.z(xx)))$

Нумерация Чёрча

n-кратная композиция $x(x(x...(xy)))$

$x^n y$

Z_n или $\overline{n}$

(ноль раз произведена операция над y)

$\overline{0}\ xy \underset{\beta}{\triangleright} y$

$Z_n = \overline{n} = \lambda xy.x^n y$

$\overline{n}\ xy \underset{\beta}{\triangleright} x^n y$ (x, взятое n-раз от y)

Комбинатор прибавления единицы

$\overline{\sigma} = \lambda uxy.x(uxy)$

Добавление единицы к числу Чёрча:

$\overline{\sigma}\ \overline{n} \underset{\beta}{\triangleright} \overline{n+1}$

$(\lambda uxy.x(uxy))(\lambda xy.x^n y)$

Пример:

$(\lambda uxy.x(uxy))(\lambda xy.x^n y) \underset{\beta}{\triangleright} ... \lambda xy.x^{n+1} y$

$\underset{\beta}{\triangleright} (\lambda xy.x((\lambda xy.x^n y)xy)) \underset{\beta}{\triangleright} \lambda xy.x(x^n y)$

$\underset{\beta}{\triangleright} \lambda xy.x^{n+1} y$

Комбинатор упорядоченной пары
(проверка на ноль)

$D = \lambda xyz.z(Ky)x$

$DXY\ \overline{0} \underset{\beta}{\triangleright} X$

$DXY(\overline{n+1}) \underset{\beta}{\triangleright} Y$

Комбинатор примитивной рекурсии

$R = \lambda xyu.u(Qy)(D\ \overline{0}\ x)1$

$Q = \lambda yv.D(\overline{\sigma}(v\ \overline{0}))(y(v\ \overline{0})(v1))$

$RGX\ \overline{0} \underset{\beta}{\triangleright} X$

$RGX(\overline{n+1}) \underset{\beta}{\triangleright} G\ \overline{n}(RGX\ \overline{n})$

Способы организации рекурсии

1. Самовызов (функция вызывает сама себя)

$$длина(x)=\begin{cases} x=nil, 0 \\ x\neq nil, длина(x-1)+1 \end{cases}$$

2. Рекурсия с накапливающимся параметром

$$длина1(x,l)=\begin{cases} x=nil, l \\ x\neq nil, длина1(cdr(x), l+1) \end{cases}$$

где x – список, l – его длина.

Не вызываем ни какую функцию, меняем только параметр. И в результате выдаём этот параметр. Доходим до пустого x и возвращаем l.

Недостаток: не задано l.

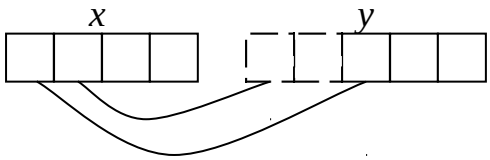
3. Вспомогательные функции

$$длина(x)=длина1(x,0)$$

Программа обращения списка:

Функция обращения: *obr(x)=if x=nil then x*
else obr1(x,nil)

obr1(x,y)=if x=nil then y
else obr1(cdr(x), cons(car(x),y))



На языке LISP:
(obr1 (cdr x) (print (cons (car x) y)))

Функции высших порядков

- функции, у которых в качестве аргумента выступает функция.

Пусть есть список: (5 8 4)

Необходимо увеличить каждый элемент списка на единицу, т.е. функция должна вернуть: (6 9 5)

$$S(x)=\begin{cases} x=nil, x \\ x\neq nil, cons(car(x)+1, S(cdr(x))) \end{cases}$$, где x – список

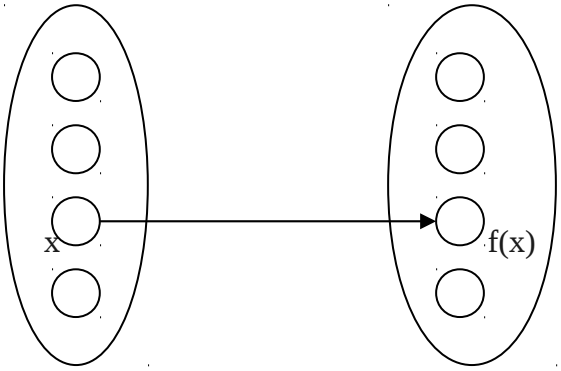
Отладка:

S(5 8 4)	cons(6, S(8 4))
S(8 4)	cons(9, S(4))
S(4)	cons(5, S())
S()	()

На языке LISP:
(defun S(x) (if x

(cons (+ (car x) 1)
(S (cdr x))
) nil))

В результате каждый элемент списка увеличится на единицу.



С помощью функции отображается другой список.

Вызов функции: *((lambda (x) (+ x 3)) 7)*

λ-нотация: $(\lambda x.x+3)7 \underset{\beta}{\triangleright} 7+3=10$

Обобщим функцию:

(defun otobr(x f)
(if x (cons (f (car x)) (otobr (cdr x) f)) nil)

Например:

(otobr '(7 3) (lambda (x) (+ x 3)))

В результате получим: *10 6*

Чтобы каждый раз не писать *lambda (x) (+ x 3))*, присвоим значение:

(setq ft (*lambda (x) (* x 2)))*

(otobr '(4 8) ft)

В результате получим: *8 16*

Комбинаторная логика

Комбинаторная логика позволяет избежать использования связанных переменных, усложняющих λ-исчисление.

Комбинатор - λ-терм без свободных переменных.

$I = \lambda x.x$

$Ix = x|I5 = 5$

$(\lambda x.x)5 = 5$

1) в комбинаторной логике часто используют два комбинатора *S* и *K*, называемых *базисными комбинаторами*.

K - $\lambda xy.x$ - $KXY = X$

S - - $SFGX = FX(GX)$

2) аппликация: $M * N$ или MN
3) левоассоциативность: $KLM = ((KL)M)$

Языки, которые основываются на комбинаторной логике:

ML (Milner, 1984 год)
Миранда (Turner, 1985 год)

Из этих языков был “собран” язык Haskell (98). Для работы с языком Haskell используется:

интерпретатор HUGS	компилятор ghc интерпретатор ghci

```
>2+2
4
>let pi=3.14
3.14
```

Примечание: let пишется в интерпретаторе ghci, в компиляторе нет let.

```
>let area r=pi*r^2
>area 7
153.93
```

```
>let umn a b =a*b
```
(использование функции умножение с двумя аргументами)

Списки:
```
>let spis=[7,2,3]
>1:9:spis
[1,9,7,2,3]
```
(добавляем в начало списка)

Примечание: буквы пишутся в одиночных кавычках.

Пример:
Создадим файл *fac.hs*.

В файле напишем функцию вычисления факториала:
fac::Integer -> Integer
fac 0 = 1
*fac n | n>0 = n*fac(n-1)*

Другой вариант:
*fac n = if n>0 then n*fac(n-1)*
else 1
Получаем:
>:l fac.hs
>fac 5
120
>fac (6)
720

Зададим в качестве аргументов функцию:
>let twice f x = f(f(x))
или:
>let twice f x = f f x
>let pl1 x = x+1
>twice pl1 7
9

Выразим тождественный комбинатор через S и K:
$Ix = x$
$I = SKK$
$SFGX = FX(GX)$
$KXY = X$

>let k x y = x
>let s x y z = x z (y z)
или:
>let s x y z = x(z, y(z))
>let I x = s k k x
>i 9
9

Алгорифмы Маркова.
Язык Рефал.

Пример1:

Запишем правила

$ba \to ab$

$\to \bullet$

На входе строчка: *abab*

Сначала используем первое правило ($ba \to ab$) до тех пор, пока оно подходит, только потом переходим ко второму правилу.

Получаем: *aabb*

Пример2:
Вычислить сумму палочек:
| | | + | | + | | |
Правила:

$|+ \to +|$

$+ \to$

$\to \bullet$

Язык Рефал

$$a \longrightarrow b$$

$$\longrightarrow \bullet$$

Функция: *a_на_b*

$e_1 a e_2 \to e_1 b < a\_на\_b \ e_2 >$

e_1, e_2 - Произвольные выражения (набор символов)
Выход из рекурсии: $e_1 \to e_1$

Пример1:
Вместо слова *приииветтттт* написать слово *привет*:

выч_один $e_1 S_2 S_2 e_3 \to e_1 < выч\_один \ S_2 e_3 >$

$S_2 S_2$ - одинаковые символы
$e_1 \to e_1$

Пример2:
Определить слово полиндром или нет:
Например, слово *потоп*

$S_1 e_2 S_1 \rightarrow\!\prec$ *полиндром* $e_2 >$

$\rightarrow$ *Да*

$S_1 \rightarrow$ *Да*

$e_1 \rightarrow$ *Нет*

Введение в логическое программирование

В логическом программировании используются логические функции. Обычная и логическая функции не отличаются аргументами.

$f(x)=(x+5)$ функция в функциональном программировании

$f(x)=(x>5)$ функция в логическом программировании (возвращает значение *истина* или *ложь*).

Функция преобразует одну область в другую.

В логике функции называются *предикатами*.

Предикаты

Под *предикатом* будем понимать логическую функцию, аргументы которой могут принимать значения из некоторой предметной области, а сама функция принимать значения истина или ложь.

Предикаты бывают одноместные $P(x)$, двухместные $P(x,y)$ и т.д.

Ноль-местный предикат – это *высказывание* (его значение не зависит от аргумента).

Пример:

$x>5$ - предикат

$4>5$ - высказывание (можно утверждать относительно него *false*)

У предикатов бывают *кванторы*:

Квантор всеобщности: $\forall xP(x)$

Квантор существования: $\exists xP(x)$

Предикат + Квантор = Предикат

$\forall x(x>5)$ false (ложный предикат);

$\exists x(x>5)$ true

Метод резолюций

- аксиоматическая теория первого порядка, которая использует доказательство от противного.

Теорию изобрел Дж.Робинсон в 1965 году. В теории используется язык предикатов.

Метод основывается на:

1. формализации задачи, получаем некоторые записи в форме предикатов.
2. предваренной нормальной форме (ПНФ); переводим предикаты в ПНФ.
3. сколемизации (определенные правила по отбрасыванию кванторов).
4. получении дизъюнктов (между дизъюнктами значки &, а в самих дизъюнктах - \/).
5. добавлении в систему отрицания выводимой формулы.
6. выполнении резолюций:
   6.1. унификация (приведение предикатов к единой форме);
   6.2. принцип силлогизма.

Рассмотрим подробнее каждый пункт метода.

1. Формализация задачи: из задания, данного нам, мы должны формализовать его, мы должны описать, что нам нужно, а не как решить.

Пример: (данный пример будет использоваться в дальнейшем) (1)
Даны предложения
1) Кто ходит в гости по утрам, тот поступает мудро.
2) У кого есть воздушный шарик, тот ходит в гости по утрам.
3) Воздушный шарик есть у пятачка.

Вопрос: Кто поступает мудро?

1) Разбиваем на аксиомы. В данном примере предложения построены так, что каждое предложение соответствует аксиоме.

А1) Кто ходит в гости по утрам, тот поступает мудро.
А2) У кого есть воздушный шарик, тот ходит в гости по утрам.
А3) Воздушный шарик есть у пятачка.

2) Выбор предикатов:
Замечание: Предиката от предиката $P(R(x))$ *не бывает.*
Г(х) – х ходит в гости по утрам
М(х) – х поступает мудро
Ш(х) – х владеет воздушным шариком

Составим из предикатов аксиомы:
А1) $\forall x Г(x) \to M(x)$
А2) $\forall x Ш(x) \to Г(x)$
А3) $Ш(пятачок)$
Вопрос) $\exists x M(x)$

2. ПНФ. Все кванторы должны находиться слева, знаки отрицания должны стоять непосредственно у самих предикатов. Знаки между предикатами должны быть только & или \/.

Правила: $x \to y = \overline{x} \vee y$
$\neg \forall x P(x) = \exists x \neg P(x)$

Примеры:
1) $\forall x P(x) \& \exists x R(x) = \forall x P(x) \& \exists y R(y) = \forall x \exists y (P(x) \& R(y))$
2) $\forall x P(x) \& \forall x R(x)$: $\begin{cases} a) \forall x (P(x) \& R(x)) \\ b) \forall x \forall y (P(x) \& R(y)) \end{cases}$

3)

$$\neg\forall x(\forall y P(x,y) \vee \neg\exists z R(z) \to Q(x) \,\&\, \forall y M(x,y)) =$$
$$= \neg\forall x(\neg(\forall y P(x,y) \vee \neg\exists z R(z)) \vee Q(x) \,\&\, \forall y M(x,y)) =$$
$$= \exists x((\forall y P(x,y) \vee \neg\exists z R(z)) \,\&\, (\neg Q(x) \vee \neg\forall y M(x,y))) =$$
$$= \exists x((\forall y P(x,y) \vee \forall z \neg R(z)) \,\&\, (\neg Q(x) \vee \exists t \neg M(x,t))) =$$
$$= \exists x \forall y \forall z \exists t((P(x,y) \vee \neg R(z)) \,\&\, (\neg Q(x) \vee \neg M(x,t)))$$

Вернёмся к примеру (1):

А1) $\forall x \neg Г(x) \vee M(x)$

А2) $\forall x \neg Ш(x) \vee Г(x)$

А3) $Ш(пятачок)$

Вопрос) $\exists x M(x)$

3. Сколемизация: отбросить все кванторы (кванторы всеобщности просто отбрасываются, а переменные, которые были связаны с квантором существования, заменяются либо на константы Сколема, либо на функции Сколема: на константу, если левее не было квантора всеобщности, иначе на функцию от всех переменных, стоящих слева в кванторе всеобщности).

Примеры:

$\exists x \forall y \exists z P(x,y,z) = P(a^c, y, f^c(y))$;

$\forall x \forall y \exists z P(x,y,z) = P(x,y,f^c(x,y))$

4. Получение дизъюнктов:

$(P(x) \vee R(x)) \,\&\, (G(x) \vee K(x))$

$(P(x) \vee R(x))$ - первый дизъюнкт

$(G(x) \vee K(x))$ - второй дизъюнкт

$P(x) \vee R(x) \,\&\, G(x) = (P(x) \vee R(x)) \,\&\, (P(x) \vee G(x))$ (по дистрибутивному закону)

В примере (1) получаем:

Д1) $\neg Г(x) \vee M(x)$

Д2) $\neg Ш(x) \vee Г(x)$

Д3) $Ш(пятачок)$

$\neg\exists x M(x) = \forall x \neg M(x)$

$\neg M(x)$

Д4) $\neg M(x) \vee Отв(x)$

Из двух дизъюнктов получаем третий и т.д. В результате получаем пустой дизъюнкт.

Принцип силлогизма:

$A \to B, B \to C \vdash A \to C$

$\frac{\neg A \vee B}{\neg B \vee C}$ получаем $\neg A \vee C$

Вопрос:
Д3) $Ш(пятачок)$
Д5) $\neg Ш(пятачок)$
Д3-Д5: пустой дизъюнкт

$\frac{\neg A \vee B \vee \neg C}{\neg B \vee D \vee E}$ получаем $\neg A \vee \neg C \vee D \vee E$

$\frac{\neg A(x) \vee B(x)}{\neg B(y) \vee C(y)}$ разные аргументы у предикатов, следовательно используем унификацию.

Переменная заменяется либо на переменную, либо на константу, либо на функцию.

y меняем на *x*: получим $\neg A(x) \vee C(x)$.

Для примера (1):
Д1-Д2: Д5 $\neg Ш(x) \vee M(x)$
Д3-Д5: Д6 $M(пятачок)$
меняем *x* на пятачок
Д4-Д6: *пустой дизъюнкт* $\vee Отв(пятачок)$

Хорновская логическая программа

Хорновская логическая программа состоит из *фактов* и *формул*.

1) факты имеют вид:
<предикат>(<константы>)

Ш(пятачок)

2) формулы должны быть вида:
$A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_n \rightarrow A_0$
$\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n \vee A_0$
(только один предикат без отрицания)

Перепишем:
$A_0 : \neg A_1, A_2, A_3, \ldots, A_n$
$A_0 \;\; if \;\; A_1, A_2, A_3, \ldots, A_n$

Выполняем логические функции и получаем логическую функцию A_0.

Язык Prolog

<предикат>(<термы>). *это факт*
<предикат>(<термы>):−<предикаты>. *это правила или формулы*

Интерпретатор: *swi-prolog*
Компилятор: *gprolog*

1. факты и правила оформляются в виде отдельного файла (prog.pl)
2. включается программа в командной строчке $swipl
3. подключаем файл:
?-consult(prog).
4. теперь можно задавать вопросы:
?-<предикат>(<термы>).

Программа на Прологе (предикаты пишутся строчными буквами, переменные прописными):

Файл *shar.pl* :
m(X) := g(X).
g(X) := s(X).
s(‘Pyatak’).

В командной строчке:
$swipl
?-consult(shar).
?-m(X).
x=Pyatak
Yes
?-g(‘Pyatak’).
Yes
?-s(‘Vinny’). спросили есть ли у Винни-Пуха шарик
No
?-

Предикаты с двумя аргументами

catty(‘Murka’).
mouse(‘Mikky’).
fish(‘Mintay’).

```
eat(X,Y):-catty(X),mouse(Y).
eat(X,Y):-catty(X),fish(Y).

?-eat(X,'Mintay').                    (Кто есть минтай?)
X=Murka

Трассировка:
?-trace.
?-m(X).                               (для примера с Пятачком)
call m(_G283)
call g(_G283)
call s(_G283)
exit s(Pyatak)
exit g(Pyatak)
exit m(Pyatak)
```