ФЛП

11 сентября 2014 г.

12:34

Что может являться образцом:

* числа;
* константы;
* названия параметров;
* списки с элементами-образцами;
* оператор «:» с образцами слева и справа;
* оператор «+» слева от которого образец, а справа натуральное число (но по умолчанию отключено, нужно {-# language NPlusKPatterns #-}).

isEmpty :: [a] -> Book

isEmpty [] = True

isEmpty \_ = False

head :: [a] -> a

head (x:\_) = x

tail :: [a] -> [a]

tail (\_:xs) = xs

swap :: [Int] -> [Int]

swap [] = []

swap [x] = [x]

swap (x:y:xs) -> y:x:xs

**Принципы работы рекурсивных подпрограмм**

Самый простой пример:

f x = f x

* параметр рекурсивного вызова должен быть проще, чем параметр основного вызова;
* имеется нерекурсивное определение для основного базового случая.

fact :: Integer -> Integer

fact 0 = 1

fact n = n \* fact (n - 1)

Второе правило можно заменить на:

fact s@(n + 1) = s \* fact n

Лучше использовать хвостовую рекурсию, чтобы стек не заполнялся:

fact :: Int -> Int

fact n = fact' n 1

fact' :: Int -> Int -> Int

fact' 0 acc = acc

fact' n acc = fact' (n - 1) (acc \* n)

**Рекурсивные функции для работы над списками**

length :: [a] -> Integer

length [] = 0

length (\_:xs) = length xs + 1

reverse :: [a] -> [a]

reverse [] = []

reverse (x:xs) = reverse xs ++ [x]

(++) :: [a] -> [a] -> [a]

[] ++ ys = ys

(x:xs) ++ ys = x:(xs ++ ys)

qsort [] = []

qsort (x:xs) = qsort less ++ [x] ++ qsort greater

where

less = filter (<x) xs

greater = filter (>=) xs

Пирамидальная сортировка на защиту второй лабы

isort [] = []

isort (x:xs) = insert x (isort xs)

insert :: a -> [a] -> [a]

insert x [] = [x]

insert x (y:ys) = if ( ...

zip :: [a] -> [b] -> [(a, b)]

zip [] \_ = []

zip \_ [] = []

zip (x:xs) (y:ys) = ...

odd :: Integer -> Bool

odd 1 = False

odd (n + 1) = even n

even :: Integer -> Bool

even 1 = True

even (n + 1) = odd n

**Абстракция списка**

[ функция | x <- список, предикаты ]

[(x, y) | x <- [1..3], y <- [4..5]]

**Понятие типа данных. Множественное значение типов**

Тип -- это коллекция связанных значений.

Тип (Bool -> Bool) представляет собой все функции, которые отображают тип Bool в тип Bool.

Если f -- это функция, которая переводит аргументы типа a в результат типа b, и e -- это выражение типа a, то f(e) имеет тип b.

**Полиморфизм и перегрузка типов функций**

Length [1, 2, 3, 4]

Length ["foo", "bar"]

Length :: [a] -> Int

Возможность для Length принимать на вход любой тип выполняется при помощи переменных типа. Переменные типа должны начинаться с маленькой буквы, обычно для них применяют названия a, b, c, d.

Тип, который содержит одну или более переменных типа называется полиморфным, как и выражения с такими типами.

**Перегруженные типы**

Возможность применения (+) к различным числовым типам делается путем добавления ограничения классов.

(+) :: Num a => a -> a -> a

**Основные классы типов**

Класс -- это набор типов, которые поддерживают определённые перегруженные операции, которые называются методами.

Eq -- класс, содержащий сравнительные типы. Это те типы, которые сравниваются на равно и неравно. И в нем определены два метода:

(==) :: a -> a -> Bool

(/=) :: a -> a -> Bool

Ord -- класс, который содержит упорядоченные типы.

(<) :: a -> a -> Bool

(>)

(<=)

(>=)

min

max

Show -- все стандартные типы.

show :: a -> String

Read -- все стандартные типы.

read :: String -> a

Num -- все числовые типы.

(+)

(-)

(\*)

abs

negate

signum

Integral -- целочисленные типы

div

mod

Fractional -- все вещественные

(/)

**Определение синонимов типов**

type String = [Char]

type Pos = (Int, Int)

type Board = [Pos]

~~type Tree = (Int, (Tree, Tree))~~

**Определение новых типов**

data Bool = True | False

Все, что справа -- конструктор типа

data Move = Left | Right | Bottom | Top

move :: Move -> Pos -> Pos

moves :: [Move] -> Pos -> Pos

moves [] p = p

moves (m; ms) p = moves ms (move m p)

data Shape = Circle Float | Rect Float Float

square :: Float -> Shape

square n = Rect n n

area :: (Shape) -> Float

area (Rect n n) = n \* n

area (Circle r) = pi \* r^2

----------------------

data Bool = False | True deriving (Eq, Ord, Show, Read)

class Eq a where

(==) :: a -> a -> Book

x /= y :: !(x == y)

instance Eq a Bool where

False == False = True

True == True = True

\_ == \_ = False

**Определение параметризироаанных полиморфных типов**

type Parser a = String [(a, String)]

type Assoc k v = [(k, v)]

data Maybe a = Nothing | Just a

find :: [a] -> a -> Maybe a

find [] \_ = Nothing

find (x:xs) y = if y == x then Just x else find xs y

**Рекурсивные типы**

data List a = Nil | Cons a (List a)

data Nat = Zero | Next Nat

len:: List a -> Int

len Nil = 0

len (Cons \_ xs) = 1 + len xs

len (Cons 2 (Cons 3 Nil))

data Tree = Node Tree Int Tree | Leaf Int

t = Node (Node (Leaf 2) 3 (Leaf 3)) 4 (Leaf 5)

occurs :: Int -> Tree -> Bool

occurs a (Node l x r) = x == a || (occurs a l) || (occurs a r)

occurs a (Leaf x) = x == a

data Tree a = Leaf a | Node (Tree a) (Tree a)

data Tree a = Node a [Tree a]

**Функции высшего порядка**

Функции первого порядка -- преобразует входные значения в выходные.

Концепция функций высшего порядка заключается в том, что функции должны иметь тот же статус, что и любой другой объект.

В функциональных языках функции ведут себя так же как и любой другой тип:

1. Функции имеют тип.
2. Функции могут быть результатом работы других функций.
3. Функции могут использовать в качестве параметров другие функции. Такте функции называются функциями высшего порядка.

quads :: Num a => [a] -> [a]

quads [] = []

quads (x:xs) = x \* x : quads xs

evens :: Integral a => [a] -> [a]

evens [] = []

evens (x:xs) = if even x then x : evens xs else evens xs

quads' :: (a -> a) -> [a] -> [a]

quads' \_ [] = []

quads' f (x:xs) = (f x) : quads' f xs

map :: (a -> b) -> [a] -> [b]

map f xs = [f x | x <- xs]

map twice [1..5]

map sqrt [1..10]

filter :: (a -> Bool) -> [a] -> [a]

filter p xs = [x | x <- xs, p x]

filter even [1..10]

**Карринг, частичная параметризация и операторные секции**

bigger :: (Ord a) => (a, a) -> a

bigger (x, y) = if x > y then x else y

biggerc :: Ord a => a -> a -> a

biggerc x y = if x > y then x else y

let a = (curry bigger) 2 3

curry :: ((a, b) -> c) -> (a -> b -> c)

curry f x y -> f (x, y)

plusc :: Num a => a -> a -> a

plusc x y = x + y

succ :: a-> a

succ = plusc 1

**Операторные секции**

filter (>2) [1..5]

map (2^) [1..5]

**Композиция функций**

f . g x аналогично f(g(x))

(.) :: (b -> c) -> (a -> b) -> (a -> c)

f . g = \x -> f(g x)

filter (not . even) [1..5]

**Функции группы свёртки (fold \*)**

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f e [] = e

foldr f e (x:xs) = x `f` foldr f e xs

sum = foldr (+) 0

product = foldr (\*) 1

and = foldr (&&) True

decimal [x1, x2, ...] = 10 \* ( 10 \* ( 10 \* 0 + x0) + x1) + x2)

😈, как n 😈 x = 10 \* n + x

..(0 😈 x0)😈 x1)😈x2)

foldl (😈) e [x1, x2, ..., xn] = (...((e😈 x1)😈...)😈 xn)

foldl :: (b -> a -> b) -> b -> [a] -> b

foldl f e [] = e

foldl f e (x:xs) = foldl f (e `f` x) xs

foldl (+) e [a, b, c]

== foldl (+) (e (+) a) [b, c]

== foldl (+) ((e (+) a) (+) b) [c]

== foldl (+) ((e (+) a) (+) b) (+) c []

== ((e (+) a) (+) b) (+) c

op :: Integer -> Integer -> Integer

op a b = 10 \* a + b

decimal :: [Integer] -> Integer

decimal = foldl op 0

**Средства ввода-вывода**

**Принципиальные трудности реализации в функциональных программах ввода-вывода**

val :: Int

val = 42

f :: Int -> Int

f n = val + n

**Изменение внешнего мира**

printAString :: RealWorld -> String -> RealWorld

readAString :: RealWorld -> (RealWorld, String)

rw = RealWorld

main rw =

let rw' = printAString rw 'Enter passwd'

(rw'', phrase) = readAString

in

printAString rw' ('Your phrase' + phrase)

printAString2 :: String -> (RealWorld -> a) -> RealWorld -> a

readAString2 :: (String -> RealWorld -> a) -> RealWorld -> a

main rw = printAString2 'Please enter passwd' (readAString2 (\phrase -> printAString2 ('Your phrase' + phrase))) rw

**Отличие функций и действий**

putStrLn :: String -> IO ()

getLine :: IO String

getChar :: IO Char

**Внутренняя организация команды do. Команды >>= и >>**

Do-нотация -- это гибкий механизм, который поддерживает две вещи:

1. Последовательно выполнение команд, в которых применяется ввод и вывод.
2. Используется для получение результатов из команд ввода и вывода.

putStrLn :: String -> IO ()

putStrLn s = do putStr s

putStr '\n'

readAndPut :: IO ()

readAndPut = do s <- getLine

putStr 'read ' + s

(>>=) :: IO a -> ( a -> IO b) -> IO b

addOne :: IO ()

addOne = do

line <- getLine

putStrLn (show (1 + read line :: Int))

addOne = getLine >>= (\line -> putStrLn (show (1 + read line :: Int)))

putStrLn "foo" >> putStrLn "bar" :: IO ()

(>>) :: IO a -> IO b -> IO b

m >> k = m >>= \\_ -> k

while :: IO Bool -> IO () -> IO ()

while test action = do

result <- test

if result then do

action

while test action

else return IO ()

readAll = while (do res <- isEOF

return (not res))

(do line <- getLine

putStrLn line)

readFile :: FilePath -> IO String

writeFile :: FilePath -> String -> IO ()

appendFile :: FilePath -> String -> IO ()

type FilePath = String

catch :: IO a -> (IO Error -> IO a) -> IO a

catch...

random :: (RandomGen g; Random a) => g -> (a, g)

random (mkStdGen 100) :: (Int, StdGen)

do

let (randNum, newCond) = rando...

**Понятие монады. Использование управляющих конструкций для структурирования вычисления.**

Монада -- это способ структурировать вычисления с точки зрения значений и последовательности вычислений, использующих эти значения.

Монады позволяют программисту строить вычисления, используя последовательные модули, которые сами могут быть последовательностями вычислений. Монада определяет каким образом вычисления комбинируются для создания новых вычислений, тем самым освобождая программиста от необходимости кодировать комбинацию вручную, каждый раз, когда она потребуется.

data Maybe a = Nothing | Just a

*Свойства монад*

1. Модульность. Отделение комбинирования вычислений от самих вычислений.
2. Гибкость. Можно написать стратегию вычислений.
3. Изолированность. Можно создавать подобия императивных вычислений.

m

return :: (a -> m a)

>>= :: (m a -> (a -> m b) -> m b)

Пример:

type Sheep = ...

father :: Sheep -> Maybe Sheep

mother :: Sheep -> Maybe Sheep

maternalGrandfather :: Sheep -> Maybe Sheep

maternalGrandfather s = case (mother s) of

Nothing ...

comb :: Maybe a -> (a -> Maybe b) -> Maybe b

comb Nothing \_ = Nothing

comb (Just x) f = f x

Just x 'comb' father

ggf :: Sheep -> Maybe Sheep

ggf s = (Just s) 'comb' father 'comb' father 'comb' father

*Законы монад*

1. (return x) >>= f == f x
2. m >>= return == m
3. (m >>= f) >>= g == m >>= (\x -> f x >>= g)

*Использование монады IO Ref*

data Point = XY {

x :: IORef,

y :: IORef

}

newPoint (x, y) = do

xRef <- newIORef x

yRef <- newIORef y

return (x y xRef yRef)

movePoint (dx, dy) (XY x y) = do

modifyIORef x (+ dx)

modifyIORef y (+ dy)

printPoint (XY x y) = do

x' <- readIORef x

y' <- readIORef y

print (x', y')

main = do

p <- newPoint (1,2)

printPoint p

movePoint (3,2) p

printPoint p

**Примеры использования языка Хаскел. Распознавание и трансляция выражения.**

Парсер -- это функция, которая принимает строку, а отдает некоторое дерево, которое явно показывает структуру данной строки.

Определение типа Парсер:

type Parser = String -> [(a, String)]

return :: a -> Parser a

return v = \inp -> [(v, inp)]

failure :: Parser a

failure = \inp -> []

item :: Parser a

item = \inp -> case inp of

[] -> []

(x:xs) -> [(x, xs)]

parse :: Parser a -> String -> [(a, String)]

parse p inp = p inp

parse (return "2") "abc" === [("2", "abc")]

parse item "ABC" === [("A", "BC")]

parse failure "ABC" === []

(>>=) :: Parser a -> (a -> Parser b) -> Parser b

p >>= f = \inp -> case parse p inp of

[] -> []

[(v, out)] -> parse (f v) out

test = do

t1 <- item

t2 <- item

return (t1, t2)

(+++) :: Parser a -> Parser a -> Parser a

p +++ f = \inp -> case parse p inp of

[] -> parse f inp

[(v, out)] -> [(v, out)]

parse (item +++ return "b") ""

digit :: Parser Char

digit = do

x <- item

if isDigit x then return x

else failure

sat :: (Char -> Bool) Parse Char

sat p = do

x <- item

if P x then return x

else failure

digit = sap isDigit

many :: Parser a -> Parser [a]

many p = many1 p ++ return []

many1 :: Parser a -> Parser [a]

many1 p = do

v <- p

vs <- many p

return (v:vs)

nat :: Parser Int

nat = many digit >>= return . read

alphanum :: Parser Char

alphanum = sat isAlphanum

lower :: Parse Char

lower = sat isLower

ident :: Parser String

ident = do

x <- lower

xs <- many alphanum

return (x:xs)

char :: Char -> Parser Char

char c = sat ( == c)

space :: Parser ()

space = many (sat isSpace) >> return ()

string :: String -> Parser String

string [] = return []

string (x:xs) = do

char x

string xs

return (x:xs)

token :: Parser a -> Parser a

token p = space >> p >>= (\inp -> space >> return inp)

natural = token Nat

symbol :: String -> Parser String

symbol xs = token (string xs)

Грамматика:

expr ::= term (+ expr)

term ::= factor (\* expr)

factor ::= (expr) | nat

nat ::= 0|1|2|...

Haskell:

expr :: Parser Int

expr = do

t <- term

do

symbol " + "

e <- expr

return (t + e)

+++ return t

term :: Parser Int

term = do

t <- factor

do

symbol " \* "

t <- term

return (f \* t)

+++ return f

factor :: Parser Int

factor = do

symbol "("

e <- expr

symbol ")"

return e

+++ natural

data Op = Add | Sub | Mul | Div

valid :: Op -> Int -> Int -> Bool

valid Add \_ \_ = True

valid Mul \_ \_ = True

valid Sub x y = x > y

valid Div x y = x 'mod' y == 0

apply :: Op -> Int -> Int -> Int

apply Add x y = x + y

apply Mul x y = x \* y

apply Sub x y = x - y

apply Div x y = x 'div' y

data Expr = Val Int | App Op Expr Expr

eval :: Expr -> [Int]

eval Val n = [n | n > 0]

eval (App o l r) = [apply o x y | x <- eval l,

y <- eval r,

valid o x y]

# подмножества

subs :: [a] -> [[a]]

subs [] = [[]]

subs (x:xs) = ys ++ map (x:) ys

where

ys = subs xs

# размещение элемента во все возможные позиции

interleave :: a -> [a] -> [[a]]

interleave x [] = [[x]]

interleave x (y:ys) = (x:y:ys) : map (y:) (interleave x ys)

# перестановки

perms :: [a] -> [[a]]

perms [] = [[]]

perms (x:xs) = concat (map (interleave x) (perms xs))

# все возможные размещения

choices:: [a] -> [[a]]

choices [] = [[]]

choices xs = concat (map perms (subs xs))

# разбиения

split :: [a] -> [([a], [a])]

split [] = []

split [\_] = []

split (x:xs) = ([x], xs) : [(x : ls, rs) | (ls, rs) <- split xs]

exprs :: [Int] -> [Expr]

exprs [] = []

exprs [n] = [Val n]

exprs ns = [e | (ls, rs) <- split ns,

l <- exprs ls,

r <- exprs rs,

e <- combine l r]

combine l r = [App o l r | o <- ops]

ops = [Add, Mul, Sub, Duv]

solutions :: [Int] -> Int -> [Expr]

solutions ns n = [e | ns' <- choices ns,

e <- exprs ns,

eval e == n]

**Унификация** -- поиск решения в Прологе (сопоставление с образцом)

**Механизм поиска решений в Прологе**

call -- начало поиска предложения, которое унифицируется с целью;

exit -- показывает, что цель была удовлетворена (решена), устанавливает маркеры перехода в базе данных, связывает переменные;

redo -- повторяет цель, отменяет связывание переменных, продолжает поиск с позиции маркера.

fail -- показывает, что больше подходящих предложений нет.

? location (x, kitchen)

call : location (x, kitchen)

exit : location (apple, kitchen)

x = apple;

redo : location (x, kitchen)

exit : location (wm, kitchen)

x = wm

yes

А если x = wm;, то:

redo : location (x, kitchen)

fail : location (x, kitchen)

no

sleeps (x)

? sleeps (jane)

yes

Предположим нам необходимо сформулировать сложную логику. Предположим, что мы знаем следующее:

1. Джону нравится любой, кому нравится вино

likes (vasya, wine)

likes (john, x) :- likes (x, wine)

?-likes (john, miron) no

?-likes (john, vasya) yes

1. .. и те, которые закусывают:

likes (john, x) :- likes (x, wine), likes (x, food)

1. Все женщины, которые нравятся Джону:

likes (john, x) :- woman (x), likes (x, wine)

1. Человек может вытащить себя из болота:

может\_вытащить\_из\_болота (x)

мвиб (x) :- мюнхгаузен (x)

мвиб (x) :- индиана\_джонс (x)

мвиб (x) :- в\_синей\_будке (x)

мвиб (x) :- кто\_может\_лгать (x)

кто\_может\_лгать (x)

Посчитать плотность населения

population (russia, 146).

area (russia, 17).

population (usa, 320).

area (usa, 9.5).

density (x, y) :- population (x, p), area (x, s), y is p/s.

**Работы с базами данных языка Пролог**

asserta(x) -- вставить в начало;

assertz(x) -- вставить в корень;

retract(x) -- удаляет.

goto(Place) :- can\_go(Place), move(Place), look(Place).

can\_go(Place) :- here(X), connect(X, Place)

move(Place) :- retract(here(X)), asserta(here(Place))

**Предикаты специального вида**

! -- отсечение (cut) -- «больше обратно не возвращайся»

data(one)

data(two)

data(three)

cut\_test\_a(X) :-

data(X), !.

cut\_test\_a('last clause').

?- cut\_test\_a(X), write(X), nl, fail.

**Особенности использования арифметических операций**

X is 2+2

4

X is (8+2)/2

5

X is (8-2)/2

3

X > Y

X < Y

X >= Y

X =< Y

?- X is 2+3, X < 5

no

?- X is 3+3, X > 5

yes

**Представление предиката в виде операторов**

function(arg1, ..., argn)

display

?- display (2+2)

+(2, 2)

?- display (3 \* 4 + 6)

+ (\*(3, 4), 6)

infix: 3+4

prefix: -7

postfix: 8factorial

is\_in(apple, kitchen)

op(35, xfx, is\_in)

?- apple is\_in kitchen

yes

?- apple is\_in X

X = kitchen

op(33, fx, room)

room kitchen = room (kitchen)

**Рекурсия, влияние порядка правил на эффективность работы программы**

predok(Predok, Potomok) :-

isParent(Predok, Potomok).

predok(Predok, Potomok) :-

isParent(Predol, Human),

predok(Human, Potomok).

предок2 (Предок, Потомок) :-

родитель(Предок, Потомок).

предок2 (Предок, Потомок) :-

предок2(Человек, Потомок),

родитель(Предок, Человек).

fact(1, 1).

fact(X, F) :-

X > 1,

XN is X - 1,

fact(XN, F1),

F is X \* F1.

**Списки на Прологе**

[]

[1, 2, 3]

[mon, tue, fri]

[g(mon), b(tue), b(w), vb(th), vg(fri)]

[H|T]

[H1, H2, H3 | [T]]

length([], 0).

length([\_|T], X) :-

length(T, Y),

X is 1 + Y.

Принадлежность элемента списку

member(X, [X | \_]).

member(X, [\_ | T]) :-

member(X, T).

Конкатенация двух списков

[1, 2, 3], [4, 5]

conc([], Y, Y).

conc([X|XT], Y, [X|L]) :-

conc(XT, Y, L).

Конкатенация, вычитания, деление списка на два подсписка всеми способами, проверка, состоит ли список из двух, разбиение списка на две части по одному элементу

reverse([], []).

reverse([H | T], S) :-

reverse(T, S1), conc(S1, [H], S).

palindrom(X) :- reverse(X, X).

delete\_all(\_, [], [])

delete\_all(X, [X|T], T1) :-

delete\_all(X, T, T1).

delete\_all(X, [Y | T], [Y | T1]) :-

X <> Y,

delete\_all(X, T, T1).

**Представление и обработка деревьев в языке пролог**

Дерево -- ациклический граф

tree(2, tree(3, empty, empty), empty)

?- tree(2, x, empty)

Принадлежит ли значение дереву.

member(X, tree(X, \_, \_)) :- !.

member(X, tree(\_, L, \_)) :- member(X, L), !.

member(X, tree(\_, \_, R)) :- member(X, R), !.

Функция замены элемента в дереве:

replace(X, Y, empty, empty).

replace(X, Y, tree(X, \_, \_), tree(Y, \_, \_))!.

replace(X, Y, tree(K, L, R), tree(K, L, G)) :- !, replace(X, Y, L, L1), replace(X, Y, R, R1).

Количество элементов в дереве:

count(empty, 0).

count(tree(\_, L, R), Result) :-

count(L, S1), count(R, S2),

Result is S1 + S2 + 1.

countLeafs(empty, 0) :- !.

countLeafs(tree(\_, empty, empty), 1) :- !.

countLeafs(tree(\_, L, R), X) :-

countLeafs(L, S1), countLeafs(R, S2),

X is S1 + S2.

Сумма

tree\_sum(empty, 0) :- !

tree\_sum(tree(X, L, R), Z) :-

tree\_sum(L, S1), tree\_sum(R, S2),

Z is S1 + S2 + X.

Высота дерева

tree\_height(empty, 0) :- !.

tree\_height(tree(X, L, R), H) :-

tree\_height(L, HL), tree\_height(R, HR), max(HL, HR, M),

H is M + 1.

Включение элемента в упорядоченное бинарное дерево:

insert(X, empty, tree(X, empty, empty)).

insert(X, tree(X, L, R), tree(X, L, R)) :- !.

insert(X, tree(Y, L, R), tree(Y, L1, R1)) :-

X > Y, insert(X, R, R1).

insert(X, tree(Y, L, R), tree(Y, L1, R)) :-

insert(X, L, L1).