Д.з.

findMajor

Найти число больше суммы всех остальных

- Идея: можно сначала сосчитать сумму s всех чисел
  - Тогда условие: x > s – x

- С помощью maximum

```
findMajor xs = let s = sum xs
                   x = maximum xs
               in if x > s - x then Just x else Nothing
```

  - Не работает для пустого списка ☹

- С помощью filter

```
findMajor xs = let s = sum xs
                   xs1 = filter (\x -> x > s - x) xs
               in if xs1 == []  then Nothing  else Just (head xs1)
```

findMajor - продожение

- Написать специальный find

```
find cond [] = Nothing
find cond (x:xs) =
   if cond x
   then Just x
   else find cond xs
```

  - (На самом деле именно это и делает стандартный find в Data.List, т.е. его и писать не надо)

```
findMajor xs = let
    s = sum xs
  in find (\x -> x > s - x) xs
```

allDiffLists

```
allDiffLists n k = allDiffLists' n k []
```

- allDiffLists' n k s (s – те элементы, которые мы уже включили)

```
allDiffLists' n 0 _ = [[]]
allDiffLists' n k s = [x:xs | x<-[1..n], not (elem x s),
                        allDiffLists' n k (x:s)]
```

- То же с приемом "представление множества с помощью функции"

```
allDiffLists n k = allDiffLists' n k (\t -> False)
```

- allDiffLists' n k cond (cond – условие, которое мы проверяем)

```
allDiffLists' n 0 _ = [[]]
allDiffLists' n k cond = [x:xs | x<-[1..n], not (cond x),
                        xs <- allDiffLists' n (k-1)
                            (\t -> cond t || t == x)]
```

findInLists

- Без failure continuation как-то так:

  искать в первом подсписке
  if *нашли*
  then *вернуть то, что нашли*
  else *искать в хвосте*

- С использованием failure continuation

```
findInLists cond (xs:xss) err =
    find cond xs
         (                                  -- Если не нашли, то…
          findInLists cond xss err
        )
findInLists cond [] err = err
```

  - Обошлись без if!


Continuations (продолжения). Continuation-passing style (CPS)

Continuation-passing style

- Continuation: параметр–функция. Задает, *что делать после окончания работы функции*
  - failure continuation – вызываем, если функция завершилась не успешно
  - Continuation-passing style (CPS) - вызываем всегда!

Continuation-passing style – простой пример

- Обычная функция:

```
sqr x = x*x
```

- CPS

```
sqr_cps x f =
    f (x*x)
```

Что делать с результатом, когда мы его получим

- Примеры вызова:
  - Сосчитать sin(5*5)

    sqr_cps 5 sin
  - Сосчитать 5*5 + 1

    sqr_cps 5 (+1)
  - Сосчитать 5*5

    sqr_cps 5 id

Зачем??

•см. следующие слайды…

CPS и рекурсия.
Пример: факториал

- Обычная программа

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

- CPS стиль

```
fact_cps 0 f = f 1

fact_cps n f =
     fact_cps (n-1) f1
      where
         f1 res = f (res *n)
```

- Или то же:

```
fact_cps n f = fact_cps (n-1)
      (\res -> f(res*n))
```

- Или, короче:

```
fact_cps n f = fact_cps (n-1)
      (f.(*n))
```

- Вызов:

```
fact_cps 3 (^2)
      □ 36
fact_cps 5 id □ 120
```

После вычисления (n-1)! нам еще надо:
•домножить на n
•ну и потом вызвать f

Как это работает?

- Обычный fact

fact 4 □
  fact 3 □
    fact 2 □
      fact 1 □
        fact 0 □ 1
      * 1
    * 2
  * 3
* 4

- fact_cps

fact_cps 4 id □
fact_cps 3 id.(*4) □
fact_cps 2 id.(*4).(*3) □
fact_cps 1 id.(*4).(*3).(*2) □
fact_cps 0 id.(*4).(*3).(*2).(*1) □
(id.(*4).(*3).(*2).(*1)) 1 □
24

Постепенно сооружаем функцию, примерно как логическую функцию в задачах про allDiff

Чего так можно добиться?

- Оказывается, к такому виду можно привести любую программу

- fact_cps n f = fact_cps (n-1) …
  - Что можно сказать fact_cps?
    - Хвостовая рекурсия

- Для хвостовой рекурсии не нужен стек
  - Значит CPS программам *вообще* не нужен стек!
    - Чудес не бывает! Куда делся стек?
    - Стек – это и есть параметр f. Там храниться *то, что нам еще надо сделать*

Некоторые применения

- Можно реализовывать сложную передачу управления
  - Peter Landin: как программу с goto перевести в функциональную программу?
- Вариант при разработке компилятора: автоматически переводить в CPS стиль
  - Тогда не нужен будет стек
  - Например, Scheme
- Асинхронные вычисления
  - Выполнить действие A, а потом, когда оно завершиться, выполнить с результатом действие B
  - Например, .NET Task Parallel Library
    http://msdn.microsoft.com/en-us/library/ee372288(v=vs.110).aspx

Еще про >>=.

>>= для других типов

Три поиска

- Найти в списке:
  - первое число, меньшее 5
  - первое число, большее 10
  - первое число, не равное 7

  Вернуть сумму этих чисел, или [], если что-то не нашли

```
find cond [] = []
find cond (x:xs) =
   if cond x
   then [x]
   else find cond xs
```

- Можно использовать list comprehension!

```
f xs = [x+y+z | x<- find (<5) xs,
                y<-find (>10) xs,
                z<- find (/=7) xs]
```

- Используя do

```
f xs = do
   x <- find (<5) xs
   y <- find (>10) xs
   z <- find (/=7) xs
   return (x+y+z)
```

«Выполнять до первой неудачи»

```
f xs = do
   x <- find (<5) xs
   y <- find (>10) xs
   z <- find (/=7) xs
   return (x+y+z)
```

- Если пустой список обозначает неудачу поиска то *do – «выполнять до первой неудачи»*

- Или можно то же через >>=

```
f xs = find (<5) xs >>= \x ->
          find (>10) xs >>= \y ->
             find (/=7) xs >>= \z ->
                return (x+y+z)
```

do для Maybe

- >> и return (и, следовательно, do) определены и для Maybe
  - Смысл такой же – «выполнять до первой неудачи»

  - Например: Найти число, которые больше всех остальных вместе в xs, число которое больше всех остальных в yx и вернуть их произведение. Если не получится, вернуть Nothing.

```
do
  x <- findMajor xs
  y <- findMajor ys
  return  (x*y)
```

Что такое монады, формально

Монада – это тип, для которого определены операции

- >>=
- return

Строго говоря операции еще должны удовлетворять некоторым правилам (Monadic laws)

```
return a >>= f   ≡   f a
m >>= return   ≡   m
(m >>= f) >>= g   ≡   m >>= (\x -> f x >>= g)
```

Уже знаем два примера монад:

- List
- Maybe

В каких случаях используют монады?

```
f xs = do
   x <- find (<5) xs
   y <- find (>10) xs
   z <- find (/=7) xs
   return (x+y+z)
```

Вычислить find (<5) xs, **потом**
вычислить find (>10) xs, **потом**
вычислить find (/=7) xs

И, кроме этого выполнять
дополнительные действия
(проверять, удачно ли завершились вызовы)

Т.е. мы описываем некоторые действия

- Которые должны выполняться одно за другим
- И при этом должно автоматически происходить что-то дополнительное

- *Примерно* как если бы в обычном языке мы могли бы переопределить точку с запятой

Функция print

print *выражение*

```
print 56
56
```

- Ничего не возвращает, только печатает
- Смысл очень понятен
- Но непонятно как это сочетается отсуствием побочных эффектов и т.д.
  - Об этом в следующий раз

- Последовательность print записывается с помощью do
- Например:

```
do  print 56
    print 3.14159
    print "abc"
```

```
56
3.14159
"abc"
```

Пример: вывод + рекурсия

```
pr 0 = print 0
pr n = do
     print n
    pr (n-1)

pr 5
5
4
3
2
1
0
```

Задача про >>> и ее продолжение

>>>

- Что-то вроде композиции, но специально для функций вида
  список -> (значение, список)
- Пример вызова:
  f = find (>3) >>> find (>5)

```
f >>> g = \xs ->
      let
         (_, xs1) = f xs
      in g xs1
```

Недостатки >>>

- Нужно ли еще что-то, чтобы гибко комбинировать функции такого вида?
  - Пример 1:

    Найти число большее 3, потом число, больше 5 и вернуть их сумму
  - Пример 2:

    Найти число t, большее 3, а потом найти число, большее t

- С помощью >>> не написать…
  - Д.з.: обобщить >>>, чтобы устранить этот недостаток
    - Подсказка: я бы предложил ввести функцию >>>=

Символьные вычисления

eval

```
data Expr = Num Integer | X | Add Expr Expr | Mult Expr Expr
```

- eval *выражение значение_для_X*

```
eval (Num i)  _ = i
eval X n = n
eval (Add x y) n = eval  x n + eval y n
eval (Mult x y) n = eval  x n * eval y n
```

diff

```
data Expr = Num Integer | X | Add Expr Expr | Mult Expr Expr
          deriving Show
```

- В Expr обязательно deriving Show

```
diff (Num _) = Num 0
diff X = Num 1
diff (Add x y) = Add (diff x) (diff y)
diff (Mult x y) = Add (Mult (diff x) y) (Mult (diff y) x)
```

Если хотим использовать несколько переменных?

X □

Var "a", Var "sum" и т.д.

Add (Mult (Num 10) (Var "a")) (Var "b")

изображает 10*a + b

- Какой должен быть параметр у eval?
  - Список пар (строка, значение)

- Например:

eval (Add (Mult (Num 10) (Var "a")) (Var "b"))

[("a", 5), ("b", 7)]

Про некоторые доп. задачи

fromStr

```
toStr Empty = 'e`
toStr (Node v l r) = 'n':toStr l ++ toStr r
```

- создать строку ☐ дописать строку!

  fromStr’ t s – дописать представление t к строке s

```
toStr’ Empty s = 'e':s
toStr’ (Node v l r) s = let
      s1 = toStr’ r s
      s2 = tpStr’ l s1
       in 'n':s2
```

- Задача на занятии на листке: записать эти правила не используя прямо строки s, s1, s2 – т.е. с помощью операций над функциями