Функциональное программирование. Семинар 2.

Разбор дз

- Помнить про захват переменной при подстановке
- Следить за ассоциативностью
- Сказать про эта-редукцию, что если терм *можно* записать короче, то его *нужно* записать короче
- Стратегии редукции, я никакой не указал, подразумевалась нормальная. Были люди отправившиеся аппликативным путем -- молодцы, вроде тоже получилось
- Замкнутые термы -- ожидалось, что искомые термы для `xor` и `pow` не содержат свободных переменных

Recap

Я забыл рассказать, как проверять числа Чёрча на ноль.

```
> iszero = \n -> n (\x -> fls) tru
```

Обратите внимание на лямбду внутри. Она для любого переданного ей параметра возвращает ложь. Убедимся, что оно работает

```
> iszro 0
> = (\n -> n (\x -> fls) tru) 0
> = 0 (\x -> fls) tru
> = (\s z -> z) (\x -> fls) tru = tru
>
> iszro 1
> = (\n -> n (\x -> fls) tru) 1
> = 1 (\x -> fls) tru
> = (\s z -> s z) (\x -> fls) tru
> = (\x -> fls) tru = fls
```

Можно по индукции доказать, что для любого `n > 0` эта штука возвращает fls. Но мы оставим это любителям строгих доказательств.

Разминка

Верно ли, что эти термы эквивалентны? В каком смысле?

```
> \x -> x
```

```
> \y -> y
> \x y -> x y
> \x y z -> x y z
```

Первые два -- альфа-эквивалентны(напоминание). Второй внезапно эквивалентен первым двум тоже. Эта-преобразование и эта-экспансия работают. Кроме того, если мы выведем типы этих термов(что мы научимся делать на следующем занятии), то заметим, что везде будет тип как у комбинатора **I**

Нормальные формы

Найдите NF и WHNF для термов:

```
> w 1
> w n
```

Где `w = \x -> x x`.

Нормальная форма -- ну тут все ясно, все что можно вычислить, мы вычислили. Слабая заголовочная нормальная форма -- это нечто более стремное. Это либо нормальная форма, либо абстракция, например:

```
> \x -> (\y -> y) x
```

Каррирование

На лекции вам рассказывали, что от функций над кортежами можно переходить к функциям, которые принимают аргументы по одному. Такие функции можно применять частично, фиксируя один аргумент. Мы уже видели, что это работает на примере домашнего задания. Давайте напишем терм который бы брал функцию, заданную на паре и возвращал бы функцию, принимающую аргументы по одному

```
> curry = \f -> (\x y -> f (pair x y))
```

А теперь напишем обратную процедуру, uncurry. Она делает ровно обратное.

```
> uncurry = \f -> (\p -> f (fst p) (snd p))
```

Вычитание единицы.

Решим опциональное задание из домашней работы. Нам понадобятся комбинаторы:

```
> zp = pair 0 0
> sp = \p -> pair (snd p) (succ (snd p))
```

Тогда предшествование для чисел Чёрча определяется следующим образом:

```
> pred = \n -> fst (n sp zp)
```

Давайте поймем, что оно действительно работает:

```
> pred 1
> = fst (1 sp zp)
> = fst (sp zp)
> = fst ((\p -> pair (snd p) (succ (snd p))) (pair 0 0))
> = fst (pair 0 1)
> = 0
```

Опять же, можно по индукции доказать, что для любого `n > 0 pred (\s z -> s^{n} (z)) = \s z -> s^{n - 1} (z).` Есть баян про то, что вычитание придумал Клини, когда ему вырывали зуб мудрости, а нынче наркоз уже не такой забористый. Возникает вопрос, за какое время работает эта конструкция? За O(n), очевидно.

Примрек

Давайте попробуем обобщить, то что у нас было. Введем следующие комбинаторы:

```
> xz = \x -> pair x 0
> fs = \f p -> pair (f (fst p) (snd p)) (succ (snd p))
> rec = \m f x -> fst (m (fs f) (xz x))
```

Комбинатор `rec` и называется комбинатором примитивной рекурсии, знакомьтесь Предшествование тогда выразится как:

```
> pred' = \n -> rec n (\x y -> y) 0
```

Факториал?

```
> fac = \n -> rec n (\x y -> mult x (succ y)) 1
```

Сумма чисел от 0 до n?

```
> sum = \n -> rec n (\x y -> plus x (succ y)) 0
```

Списки

Список -- каноничный пример типа, который определяется рекурсивно. У списков есть конструкторы(термы, позволяющие конструировать списки):

```
> nil = \c n -> n
> cons = \e l c n -> c e (l c n)
```

Пример списков:

```
> [] = nil -- пустой список тоже вполне себе список
> [1, 2, 3] = cons 1 (cons 2 (cons 3 nil)) = \c n -> c 1 (c 2 (c 3 n))
```

Научимся проверять, пуст список или нет(очень похоже на iszero).

```
> isempty = \l -> l (\c n -> fls) tru
```

Например:

```
> isempty nil
> = (\c n -> n) (\c n -> fls) tru
> = tru
>
> isempty (cons 1 nil)
> = (cons 1 nil) (\c n -> fls) tru
> = (\c n -> c 1 n) (\c n -> fls) tru
> = (\c n -> fls) 1 tru
> = fls
```

Можно ли написать более короткую версию этой функции(я вот не умею)?

И научимся брать у него голову:

```
> head = \l -> l K
> head (cons 1 nil) = (\c n -> c 1 n) K = \n -> K 1 n = \n -> 1
```

Заметим, что мы как-то стремно взяли голову списка. Хочется получить элемент, а не функцию, фиг пойми от чего. Но если приглядеться, то таким образом мы можем застражзовать себя от взятия головы у пустого списка. С помощью эта-экспансии этот терм добивается до `headOrDefault`, который кроме списка, принимает некоторое значение по умолчанию, которое и возвращается в качестве головы пустого списка.

```
> headOrDefault = \l d -> l K* d
```

Неподвижные точки

Вам показали на лекции как с помощью комбинатора неподвижной точки ввести рекурсию в лямбда-исчисление и показали пример, как можно посчитать факториал. С помощью него можно например решать уравнения на термы. Например давайте сконструируем терм, который "поедает" все что ему дают. То есть найдем терм `F` такой, что любого `M` выполнялось бы

```
> F M = F
```

Ну заметим, что у нас вообще-то есть комбинаторы, которые умеют возвращать первый или второй аргумент. Тогда можно записать это добро в виде:

```
> F M = K F M
> F = \m -> K F m
> F = K F
> F = Y K
```

Можно теперь делить числа(представим, что сравнивать мы умеем).

```
> div = \m n -> if (le m n) 0 (succ (div (minus m n) n))
> div = Y (\f m n -> (le m n) 0 (succ (f (minus m n) n)))

> mod = \m n -> if (le m n) m (mod (minus m n) n)
> mod = Y (\f m n -> (le m n) m (f (minus m n) n))
```

Как вы думаете, что делает этот терм?

```
> L = \a b c d e f g h i j k l m n o p q s t u v w x y z r
    -> (r (t h i s i s a f i x e d p o i n t c o m b i n a t o r))
> Yk = (L L L L L L L L L L L L L L L L L L L L L L L L L L)
```

В общем, есть прикольная статья, в которой автор показывает, как можно создавать свои комбинаторы неподвижной точки. Пример выше взят именно оттуда

Домашнее задание

1. Приведите пример терма(замкнутого) который бы:
    - Находился в WHNF, но не в HNF
    - Находился бы в HNF, но не в NF
2. Напишите следующие функции над числами Черча:
    - Вычитание `m - n`:

```
minus = \m n -> ?
```

- Проверка на равенство `m == n`:
  ```
  equals = \m n -> ?
  ```
- Сравнения `m < n` `m > n` `m <= n` `m >= n`:
  ```
  lt = \m n -> ?
  gt = \m n -> ?
  le = \m n -> ?
  ge = \m n -> ?
  ```

3. Напишите предикат `isEven`, возвращающий `tru`, если его аргумент четное число и `fls` -- в противном случае. В этом задании нельзя использовать комбинатор неподвижной точки.

   ```
   isEven = \m -> ?
   ```

4. Напишите следующие функции над списками.

   - `length` -- длина списка
     ```
     length = \l -> ?
     ```
   - `sum` -- сумма элементов списка
     ```
     sum = \l -> ?
     ```
   - `map` -- применяет функцию `f` ко всем элементам списка `map succ [1, 2, 3] = [2, 3, 4]`
     ```
     map = \f l -> ?
     ```
   - `reverse` -- разворачивает список. `reverse [1,2,3] = [3,2,1]`
     ```
     reverse = \l -> ?
     ```
   - `tail` -- хвост списка. `tail [1, 2, 3] = [2, 3]`. Тут может помочь примитивная рекурсия
     ```
     tail = \l -> ?
     ```

5. Используя комбинатор неподвижной точки найдите терм F такой что:

   - Для любых `M` и `N` было бы верно `F M = M F`
   - Для любых `M` и `N` было бы верно `F M N = N F (M N F)`

6. Пусть `f` и `g` определены взаимно-рекурсивно:

   ```
   f = F f g
   g = G f g
   ```

   Используя комбинатор неподвижной точки найдите нерекурсивные определения функций `f` и `g`