1 Машины Тьюринга

1. Опишите правила перехода для одноленточной машины Тьюринга, которая вычитает одно число в унарной записи из другого. Числа записаны подряд без пробела. Например, если изначально на ленте находится $SSSSSOSSO$, то после остановки машины на ленте справа от курсора должно быть $SSSO$.

2. Сделайте то же самое, но не расширяя алфавит новыми символами.

3. Пусть дана таблица бинарной логической операции:

a	b	$a \oplus b$
0	0	n_0
0	1	n_1
1	0	n_2
1	1	n_3

   Опишите конфигурацию машины Тьюринга, которая принимает на вход биты n_1, n_2, n_3, n_4, пробел, одно число в двоичной записи, пробел, второе число в двоичной записи. В случае несовпадения длин поданных чисел число меньшей длины дополняется ведущими нулями. Требуется найти результат побитового применения заданной логической операции к поданным числам. Например, вход 0110 0011 1010 означает, что требуется найти xor чисел 3 и 10, то есть результат должен быть 1001.

   Затем сделайте то же самое, только числа n_1, n_2, n_3, n_4 записаны на второй ленте.

Рекомендуется воспользоваться сервисом вроде https://turingmachinesimulator.com/.

2 Примитивная рекурсия

Для этого задания предоставляется код на Haskell, с помощью которого можно оформлять примитивно рекурсивные функции. Задание можно сдавать не только на Haskell, но тогда нужно предоставить *полностью строго* расписанные построения примитивно рекурсивных функций, а именно, должно быть везде указано, где используются проекции, где подстановки, где что-то ещё. Кажется, что проще воспользоваться предоставленным кодом, но это не обязательное требование.

2.1 Механизм работы

2.1.1 Служебное

```
{-# LANGUAGE GADTs,
             DataKinds,
             StandaloneDeriving,
             FlexibleContexts,
             FlexibleInstances
#-}

import Data.Traversable (fmapDefault, foldMapDefault)
import Data.Foldable (toList)
```

2.1.2 Натуральные числа

Совершенно стандартное определение натуральных чисел.

```haskell
data Nat = Z | S Nat

instance Num Nat where
    Z + b = b
    (S a') + b = S (a' + b)
    Z * b = Z
    (S a') * b = b + a' * b
    abs a = a
    signum a = S Z
    fromInteger n | n < 0 = undefined
                  | n == 0 = Z
                  | otherwise = S (fromInteger (pred n))
    negate a = Z

instance Enum Nat where
    fromEnum Z = 0
    fromEnum (S n) = 1 + fromEnum n
    toEnum = fromIntegral
    succ = S
    pred Z = Z
    pred (S n) = n

instance Show Nat where
    show = show . fromEnum
```

2.1.3 Конечные множества

В Haskell существует более общее понятие, чем индуктивные определения, а именно, можно задавать, какие значения может принимать типовой параметр в зависимости от выбранного конструктора. Такие типы называются GADT. Типовой параметр, который варируется в зависимости от выбора конструктора, называется не параметром, а *индексом* данного типа.

Рассмотрим самый простой GADT — индуктивный тип конечных множеств мощности n.

```haskell
data Fin n where
    FinZ :: Fin (S n)
    FinS :: Fin n -> Fin (S n)

deriving instance Show (Fin n)
```

`FinZ` — конструктор первого элемента множества. Один элемент есть в любом непустом множестве, поэтому для любого n выполняется `FinZ : Fin (S n)`. `FinS` — конструктор элемента, следующего за каким-то другим. Если у нас есть элемент n-элементного множества, то мы можем расширить это множество следующим элементом и получить $n+1$-элементное множество.

Например, `FinS (FinS FinZ)` — индекс третьего элемента — может находиться только во множествах мощности 3 и более.

Если какой-то элемент находится в n-элементом множестве, его можно считать и частью $n+1$-элементного множества:

```haskell
extendFin :: Fin n -> Fin (S n)
extendFin FinZ = FinZ
extendFin (FinS s) = FinS (extendFin s)
```

Такие конструкции часто используют для индексации каких-то структур: согласно типам, в `Fin n` находятся только числа от 0 до $n-1$.

2.1.4 Списки известной длины

Точно так же можно применять GADT для построения списков известной во время компиляции длины. Такими списками очень неудобно пользоваться — например, как написать их конкатенацию? — но в некоторых случаях, как сейчас, они могут быть полезны.

Для наших целей определим списки, к которым можно прикреплять не голову, а хвост. Эти списки мы будем использовать для хранения аргументов примитивно рекурсивных функций, а дописывать в хвост там нужно чаще, чем в голову.

```
infixl 5 :>

data List n a where
    Nil  :: List Z a
    (:>) :: List m a -> a -> List (S m) a

instance Traversable (List n) where
    traverse f Nil = pure Nil
    traverse f (xs :> x) = (:>) <$> traverse f xs <*> f x

instance Functor (List n) where
    fmap = fmapDefault

instance Foldable (List n) where
    foldMap = foldMapDefault

instance Show a => Show (List n a) where
    show = show . toList
```

К сожалению, у таких списков есть проблема: простым образом мы можем генерировать только списки константной длины. Например, нельзя написать функцию `replicate :: Int -> a -> List k a`, поскольку, согласно этой сигнатуре, функция должна вернуть список любой длины, какую захочет пользователь, а привязать k к переданному значению `Int` нельзя. Кажется, очевидное решение — написать функцию `replicate :: a -> List k a` и делать как раз список запрошенной длины. Увы, k — это тип, и так как они стираются при компиляции, осуществлять сопоставление с образцом по k нельзя.

При этом выход из этой проблемы есть, пусть и несколько неудобный: можно сделать класс типов, который предоставляет функцию, генерирующую список заданной длины. Тогда если задать экземпляры этого класса для всех натуральных чисел, то можно получить желаемое.

Например, сгенерируем список длины k, в котором в убывающем порядке перечислены все элементы конечного множества мощности $|k|$, то есть список $[k-1, k-2, \cdots, 2, 1, 0]$.

```
class ListFin k where
    listFin :: List k (Fin k)

instance ListFin Z where
    listFin = Nil

instance ListFin k => ListFin (S k) where
    listFin = (fmap FinS listFin) :> FinZ
```

2.1.5 Примитивно рекурсивные функции

Наконец, сами примитивно рекурсивные функции. Они тоже представлены как GADT, и индекс отвечает за арность функции. Например, функция “0” имеет арность 0, поскольку не принимает никаких аргументов, а функция Succ имеет арность 1, поскольку принимает один аргумент.

Использование индексов даёт важное преимущество: каждый экземпляр этой структуры данных полностью корректно сформирован и является некоторой примитивно рекурсивной функцией. Корректность подстановок по арности обеспечивается системой типов. При этом никакие функции, которые на самом деле можно выразить в терминах представленных конструкций, мы не забраковали. Получается, что тут тот редкий случай, когда совершенно все ограничения, накладываемые предметной областью, можно выразить в системе типов.

Отдельно нужно отметить, что оператор проекции хранит в себе смещение относительно *последнего* элемента в списке аргументов. Это решение связано с тем, что все рассматриваемые конструкции наиболее важные аргументы держат ближе к концу списка. Так, аргумент `FinZ` — это последний аргумент, аргумент `FinS FinZ` — предпоследний и так далее.

```
data Prim k where
    Zero :: Prim Z
    Succ :: Prim (S Z)
    Proj :: Fin k -> Prim k
    Sbst :: Prim n -> List n (Prim k) -> Prim k
    Recc :: Prim k -> Prim (S (S k)) -> Prim (S k)

deriving instance Show (Prim k)

eval :: Prim k -> List k Nat -> Nat
eval Zero Nil = Z
eval Succ (Nil :> p) = S p
eval (Proj FinZ) (xs :> x) = x
eval (Proj (FinS s)) (xs :> x) = eval (Proj s) xs
eval (Sbst f gs) xs = eval f ((\g -> eval g xs) <$> gs)
eval (Recc f g) (xs :> y) = case y of
                            Z -> eval f xs
                            S y' -> let r = xs :> y'
                                    in eval g (r :> eval (Recc f g) r)
```

2.2 Примеры определений примитивно рекурсивных функций

Несколько вспомогательных функций, чтобы уменьшить количество кода:

```
unOpComb unOp a = Sbst unOp (Nil :> a)
binOpComb binOp a1 a2 = Sbst binOp (Nil :> a1 :> a2)
```

Несколько функций для упрощения тестирования:

```
run :: Prim k -> List k Integer -> Int
run term = fromEnum . eval term . fmap fromInteger

runUnOp term a = run (term pLast) (Nil :> a)
runBinOp term a b = run (term pSecondToLast pLast) (Nil :> a :> b)
```

2.2.1 Проекции

Самые простые примитивно рекурсивные функции — это проекции. Чтобы не запутаться, введём несколько имён для наиболее часто используемых из них.

Функция π_1^1:

```
pId :: Prim (S Z)
pId = Proj FinZ
```

Последний аргумент:

```
pLast = Proj FinZ
```

Предпоследний:

```
pSecondToLast = Proj (FinS FinZ)
```

Введём также три специальных имени, которые можно использовать при задании g из определения оператора примитивной рекурсии: там нужно обращаться к двум аргументам — к номеру рекурсивного вызова и к результату предыдущего вызова — а также, быть может, к переменной из набора χ дополнительных переменных, не относящихся к самой рекурсии. Здесь дана функция, которая позволяет получить последнюю переменную из χ.

```
pRecCallResult = Proj FinZ
pRecIteration = Proj (FinS FinZ)
pRecLastX = Proj (FinS (FinS FinZ))
```

2.2.2 Числа

Определим инкремент как унарный оператор. Это нам позволить рассматривать его как $S(k)$-арную, а не только 1-арную функцию.

```
tSucc = unOpComb Succ
```

Также введём вспомогательные функции, которые будут по натуральному числу или по `Integer`-у возвращать соответствующую константную примитивно рекурсивную функцию.

```
tN Z = Sbst Zero Nil
tN (S n) = tSucc (tN n)
tI = tN . fromInteger
```

Несколько констант для удобства:

```
tZero = tN Z
tOne = tN (S Z)
```

2.2.3 Арифметические и логические операции

```
infixr 3 .&&
infixr 2 .||
infix  4 .=
infix  4 .<=
infix  4 .>
infixl 6 .+
infixl 6 .-
infixl 7 .*

(.+) = binOpComb $ Recc pId (tSucc pRecCallResult)
(.*) = binOpComb $ Recc tZero (pRecCallResult .+ pRecLastX)
tPred = unOpComb $ Recc Zero pRecIteration
(.-) = binOpComb $ Recc pId (tPred pRecCallResult)
tToBool = tNot . tNot
(.&&) a b = tToBool a .* tToBool b
(.||) a b = tNot (tNot a .&& tNot b)
tNot a = tOne .- a

(.=) = binOpComb $ let (x, y) = (pSecondToLast, pLast)
                   in tOne .- (x .- y) .- (y .- x)
x .<= y = tSucc y .> x
```

```
x .> y = tToBool (x .- y)

ifThenElse p t e = p .* t .+ tNot p .* e
```

Рассмотрим построение одной из них, скажем, `(.*)`:

```
(.*) = binOpComb $ Recc tZero (pRecCallResult .+ pRecLastX)
```

Здесь используется оператор примитивной рекурсии, где $f = 0(\,)$ и

$$g = (x, y, z) \mapsto (+)(\pi_3^3(x, y, z), \pi_3^1(x, y, z))$$

Синтаксически кажется, что в коде не происходит операция подстановки, хотя должна бы, однако на самом деле она просто скрыта внутри `binOpComb`.

2.2.4 Суммы и произведения

Рекурсивное определение для произвольной агрегации результатов, будь то сумма, произведение или минимум, выглядит так:

$$\begin{aligned} A(\chi, 0) &= z(\chi, 0) \\ A(\chi, S(n)) &= z(\chi, S(n)) \oplus_\chi A(\chi, n) \end{aligned}$$

Здесь A — операция, χ — набор переменных (на лекции они обозначались за x_i), а $\oplus_\chi$ — операция, которая зависит от χ. Ясно, что это определение выражается через оператор примитивной рекурсии. Рассмотрим, какие f и g из определения примитивной рекурсии надо взять.

$$\begin{aligned} f:&\ \ \chi \mapsto z(\chi, 0) \\ g:&\ \ (\chi, n, A(\chi, n)) \mapsto \oplus(\chi, z(\chi, S(n)), A(\chi, n)) \end{aligned}$$

Сначала рассмотрим f. Понятно, как её выразить через базовые операции примитивной рекурсии: если χ — набор из k переменных, то

$$f = z(\pi_k^1, \pi_k^2, \cdots, \pi_k^k, 0(\,))$$

Как раз для того, чтобы программно получить список $\{\pi_k^i\}_i$, и нужна определённая ранее функция `listFin`.

Чтобы определить g, нам тоже понадобится `listFin`, однако нужно заметить, что g принимает не только χ, но и дополнительные два аргумента. Если мы хотим избежать их подавать в $\oplus$ и z — а мы хотим! — то нужно сгенерировать не список $\pi_{k+2}^1, \pi_{k+2}^2, \cdots, \pi_{k+2}^{k+2}$, а список $\pi_{k+2}^1, \pi_{k+2}^2, \cdots, \pi_{k+2}^k$. К сожалению, такой функции у нас нет. Однако вспомним, что проекции у нас содержат позицию от *конца*. Получается, если мы сгенерируем список проекций длиной k, то это будут проекции $\pi_{k+2}^3, \pi_{k+2}^4, \cdots, \pi_{k+2}^{k+2}$. Значит, нам достаточно указатель в каждой проекции сдвинуть на одну позицию влево, что делается функцией `FinS`. Применяя её дважды к каждому элементу списка, имеем нужную нам проекцию для получения χ.

```
tAggregate binOp z = Recc f g
  where f = Sbst z ((Proj <$> listFin) :> tZero)
        g = Sbst binOp (chi :> Sbst z (chi :> tSucc pRecIteration) :>
                        pRecCallResult)
          where chi = ((Proj . FinS . FinS) <$> listFin)

tProd f = tAggregate (pSecondToLast .* pLast) f
tSum f = tAggregate (pSecondToLast .+ pLast) f

tFact = unOpComb $ Sbst (tProd (tSucc pId)) (Nil :> tPred pLast)

tFactTestS = [ runUnOp tFact 0 == 1
```

```
  , runUnOp tFact 1 == 1
  , runUnOp tFact 2 == 2
  , runUnOp tFact 3 == 6
  , runUnOp tFact 4 == 24
  , runUnOp tFact 5 == 120
  ]
```

2.2.5 Остаток от деления

Сначала небольшая вспомогательная функция: для подсчёта $x \pmod y$ мы хотим осуществлять рекурсию не по y, а по x, однако рекурсивно разбирается всегда последний аргумент. Таким образом, нужна функция, которая будет переставлять местами последние два аргумента. Если имеется функция t' и мы хотим получить t такую, что она равна t', только последние два аргумента переставлены, это можно выразить так:

$$t(\chi, x, y) = t'(\chi, \pi_k^{-1}(\chi, x, y), \pi_k^{-2}(\chi, x, y))$$

```
tFlip t = Sbst t (((Proj . FinS . FinS) <$> listFin) :> pLast :> pSecondToLast)
```

На самом деле в данном случае можно было обойтись и более простой функцией $t(x, y) = t'(\pi_2^2(x, y), \pi_2^1(x, y))$, однако это ещё один повод увидеть, как работать со списками проекций неизвестной заранее длины.

Само определение модуля можно сделать примитивно рекурсивным:

$$\begin{aligned} 0 \quad & \bmod y \;\; = 0 \\ S(x) \quad & \bmod y \;\; = \textbf{if } S(x \bmod y) = y \textbf{ then } 0 \textbf{ else } S(x \bmod y) \end{aligned}$$

```
tMod = binOpComb $ tFlip $ Recc tZero $
        ifThenElse
          (tSucc pRecCallResult .= pRecLastX)
          tZero
          (tSucc pRecCallResult)

tModTests = [ runBinOp tMod 0 6 == 0
            , runBinOp tMod 1 6 == 1
            , runBinOp tMod 2 6 == 2
            , runBinOp tMod 3 6 == 3
            , runBinOp tMod 4 6 == 4
            , runBinOp tMod 5 6 == 5
            , runBinOp tMod 6 6 == 0
            , runBinOp tMod 7 6 == 1
            , runBinOp tMod 8 6 == 2
            ]
```

2.2.6 Примитивная рекурсивность функции через график и верхнюю границу

Определение полностью повторяет то, что было на лекции:

$$G(g, r) = \chi \mapsto \left((\chi, t) \mapsto \textstyle\sum_{y=0}^{t} r(\chi, y) \cdot \pi_k^{-1}(\chi, y)\right)(\chi, g(\chi))$$

```
tByBoundAndGraph bound r = Sbst (tSum (r .* pLast))
                                ((Proj <$> listFin) :> bound)
```

Например, определим функцию $y = \lfloor \sqrt{x} \rfloor$ как функцию, которая явно не больше, чем x, причём $y^2 \leq x$ и $(y+1)^2 > x$.

```
tFloorSqrt = let tSqr x = x .* x
             in unOpComb $ tByBoundAndGraph Succ
              (tSqr pLast .<= pSecondToLast .&&
               tSqr (tSucc pLast) .> pSecondToLast)

tFloorSqrtTests = [ runUnOp tFloorSqrt 0 == 0
                  , runUnOp tFloorSqrt 1 == 1
                  , runUnOp tFloorSqrt 2 == 1
                  , runUnOp tFloorSqrt 3 == 1
                  , runUnOp tFloorSqrt 4 == 2
                  , runUnOp tFloorSqrt 5 == 2
                  , runUnOp tFloorSqrt 6 == 2
                  , runUnOp tFloorSqrt 7 == 2
                  , runUnOp tFloorSqrt 8 == 2
                  , runUnOp tFloorSqrt 9 == 3
                  ]
```

2.2.7 Кванторы

Определим ограниченный квантор всеобщности $\forall z \leq g(\chi).P(\chi, z)$ так:

```
tForall g p = Sbst (tProd p) ((Proj <$> listFin) :> g)

tForallTests = [ run (tForall (tI 5 .* pLast) (tSucc pLast .> pLast))
                     (Nil :> 5) == 1
               , run (tForall (tI 20) (pLast .<= tI 20)) Nil == 1
               , run (tForall (tI 20) (pLast .<= tI 19)) Nil == 0
               , run (tForall pLast (pLast .<= pSecondToLast)) (Nil :> 5) == 1
               ]
```

Квантор μ реализуем с помощью такой формулы:

$$\mu y < g(\chi).P(\chi, y) = G(g, (\chi, y) \mapsto (\forall z \leq (y-1).((\chi, y, z) \mapsto \neg P(\chi, z))) \wedge (P(\chi, y) \vee y = g(\chi)))$$

```
tMin :: ListFin k => Prim k -> Prim (S k) -> Prim k
tMin bound pred = tByBoundAndGraph bound
  (pLast .= tZero .|| tForall (tPred pLast) (removeSecondToLast (tNot pred)) .&&
   (pred .|| pLast .= Sbst bound ((Proj . FinS) <$> listFin)))
  where removeSecondToLast t = Sbst t (((Proj . FinS . FinS) <$> listFin) :> pLast)
```

Например, найдём минимальный такой x, что $x^2 > 5x + 8$:

```
tMinTests = [ run (tMin (tI 9) (pLast .* pLast .> tI 5 .* pLast .+ tI 8)) Nil == 7
            , run (tMin (tI 9) (tI 2 .> tZero)) Nil == 0
            ]
```

2.3 Задачи

4. Реализуйте функцию, которая находит 2^n.
5. Реализуйте примитивно рекурсивную нумерацию пар.
6. Реализуйте нахождение n'того числа Фибоначчи. Тут может быть полезно предыдущее задание.
7. На лекции была дана функция Аккермана в таком виде:

```haskell
ack :: Int -> Integer -> Integer
ack 0 x = x + 1
ack i x = go (ack (i-1)) (x+2) x
  where go :: (Integer -> Integer) -> Integer -> (Integer -> Integer)
        go f 0 = id
        go f n = f . go f (n - 1)
```

Слегка перепишем её:

```haskell
ack' :: Nat -> Nat -> Nat
ack' Z = S
ack' (S i') = \x -> go (ack' i') (S (S x)) x
  where go f Z = id
        go f (S n') = f . go f n'
```

Она выглядит как примитивно рекурсивная: везде очень простые функции, каждый рекурсивный вызов происходит только с аргументом, меньшим на единицу. При этом, как мы знаем, функцию Аккермана нельзя посчитать с помощью только примитивно рекурсивных операций. В чём тут проблема?