

Потоки. «Завязывание узлов»

Часто обработку данных в программе можно представить в виде взаимодействия «потоков объектов», где элементами потоков могут быть сообщения, сигналы, даже числа.



поток – («бесконечная») последовательность объектов



узел – элемент программы, осуществляющий обработку потоков

Обработчик ожидает, когда ему поступят элементы из всех входных потоков, осуществляет их обработку, и выдает элементы в выходной поток (или потоки).

Такая модель очень хорошо согласуется с принципами функционального программирования. Обработка начинается только тогда, когда готовы аргументы (ленивые вычисления), причем порядок вычислений не важен («узлы» могут работать параллельно).

Если есть схема взаимодействия потоков, то программа, реализующая ее – это просто набор описаний узлов.

Завязывание узлов. Пример.

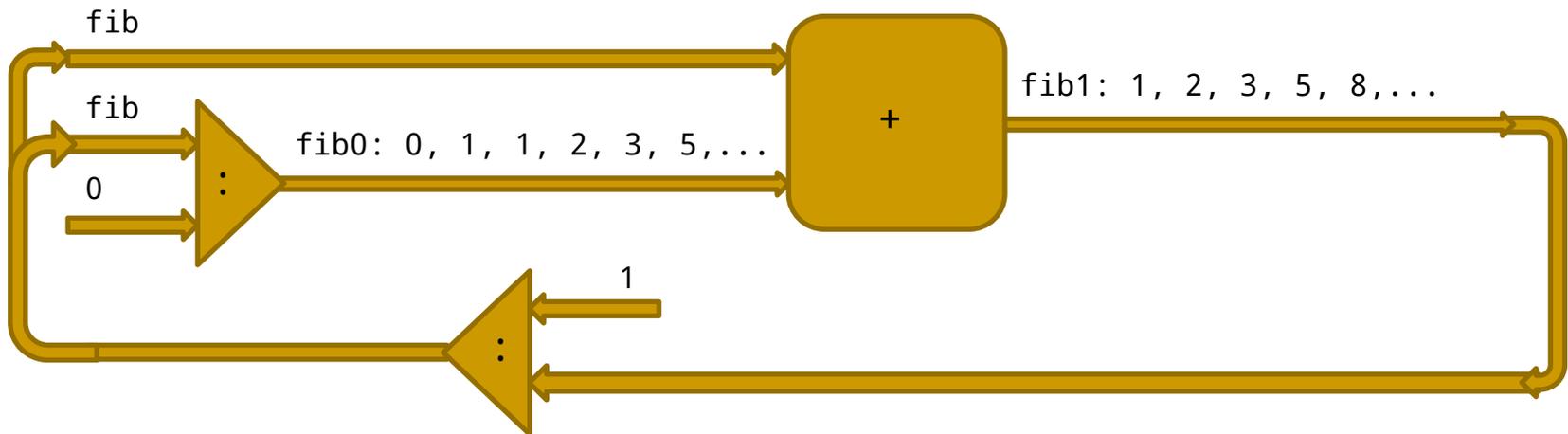
Построим программу для вычисления последовательности чисел Фибоначчи.

Допустим, что последовательность чисел Фибоначчи уже построена.

fib: 1, 1, 2, 3, 5, 8, ...

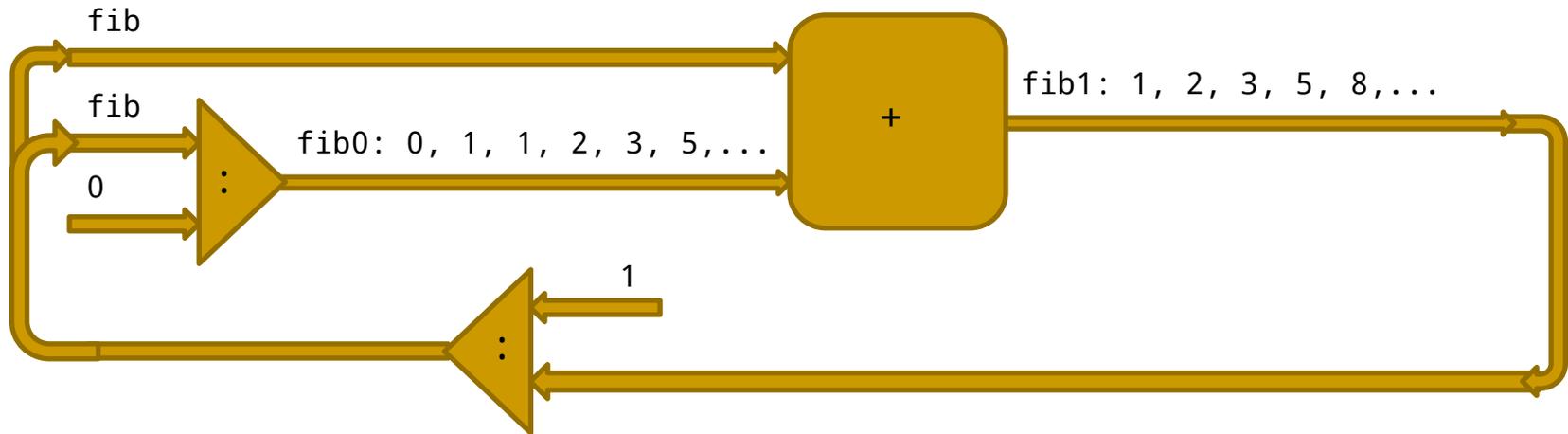


Добавим в начало потока ноль и сложим почленно с исходной последовательностью чисел Фибоначчи.



Если теперь в начало потока fib1 добавить 1, то снова получится последовательность чисел Фибоначчи.

Получение последовательности чисел Фибоначчи завязыванием узлов



Все узлы «завязаны». Готовая программа получается как совокупность описаний всех узлов.

```
fib0 = 0:fib
fib1 = zipWith (+) fib fib0
fib  = 1:fib1
```

Напомним, что

```
zipWith f (e1:t1) (e2:t2) = (f e1 e2) : (zipWith t1 t2)
```

Функциональное представление множеств: описание грамматики языка

Рассматриваем языки, описываемые регулярными выражениями, например:

```
vowel      = 'a' | 'o'
consonant  = 'c' | 'l' | 'k' | 'b'
letter     = vowel | consonant
open       = consonant vowel | consonant vowel vowel
closed     = open consonant
syllable   = open | closed
```

Цель: создавать языки с помощью операций

```
type Language = String -> Bool
vowel        = (symbol 'a') `alt` (symbol 'o')
consonant    = (symbol 'c') `alt` (symbol 'l') `alt`
              (symbol 'k') `alt` (symbol 'b')
letter       = vowel `alt` consonant
open         = (consonant `cat` vowel) `alt`
              ((consonant `cat` vowel) `cat` vowel)
closed       = open `cat` consonant
syllable     = open `alt` closed
```

Проверка принадлежности слова языку:

```
syllable "cool"
```

Программирование регулярных выражений

```
type Language = String -> Bool
symbol      :: Char -> Language
alt         :: Language -> Language -> Language
cat         :: Language -> Language -> Language

symbol c word = [c] == word
```

```
(lang1 `alt` lang2) word = (lang1 word) || (lang2 word)
```

$(lang1 \text{ `cat` } lang2)$ – язык, содержащий слова, которые можно разбить на два подслова так, что первое принадлежит $lang1$, а второе – $lang2$.

Пустое слово можно разбить только на два пустых подслова, отсюда уравнение:

```
(lang1 `cat` lang2) [] = (lang1 []) && (lang2 [])
```

Два варианта разбивки непустого слова: $w = \lambda w$ и $w = a_1 a \beta$. Отсюда уравнение:

```
(lang1 `cat` lang2) word@(x:s) = (lang1 [] && lang2 word) ||
    (lang11 `cat` lang2) s where lang11 w = lang1 (x:w)
```

Расширение техники работы с регулярными выражениями

```
(<+>)    :: Language -> String -> Language
(lang <+> word) w = (w == word) || (lang w)

(<->)    :: Language -> String -> Language
(lang <-> word) w = (w /= word) && (lang w)

iter      :: Language -> Language          -- iter exp ~ exp*
iter lang [] = True
iter lang w  = (lang1 w) || ((lang1 `cat` (iter lang1)) w)
               where lang1 = lang <-> []

poss      :: Language -> Language          -- poss exp ~ [exp]
poss lang = lang <+> []

digit = symbol '0' `alt` symbol '1' `alt` symbol '2' `alt`
        symbol '3' `alt` symbol '4' `alt` symbol '5' `alt`
        symbol '6' `alt` symbol '7' `alt` symbol '8' `alt` symbol '9'
unsigned = digit `cat` (iter digit)
integral = poss ((symbol '+') `alt` (symbol '-')) `cat` unsigned
number   = integral `cat` poss (symbol '.' `cat` unsigned)
           `cat` poss (symbol 'e' `cat` integral)
```

Еще один пример функциональной программы

```
type FGraph = (Int, (Int->Int->Bool))
type LGraph = [(Int, [Int])]
```

```
myFGraph :: FGraph
myFGraph = (6, \x y -> case (x, y) of
    (1,4) -> True
    (1,5) -> True
    (2,6) -> True
    (3,1) -> True
    (3,5) -> True
    (5,4) -> True
    (_,_) -> False
)
```

```
myLGraph :: LGraph
myLGraph = [(1, [4,5]), (2, [6]), (3, [1,5]), (4, []), (5, [4]), (6, [])]
```

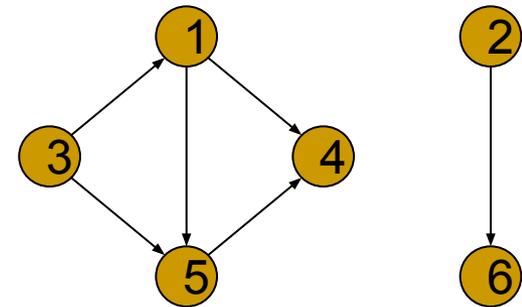
-- функции преобразования графа из одного представления в другое

```
convertF2L :: FGraph -> LGraph
```

```
convertL2F :: LGraph -> FGraph
```

```
convertF2L (n, func) = [(i, [j | j <- [1..n], func i j]) | i <- [1..n]]
```

А как перейти обратно из спискового представления в функциональное?



Поиск по ассоциативному списку

Пусть задан ассоциативный список пар:

```
ls = [(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D')]
```

```
lookup :: Eq a => a -> [(a,b)] -> b    -- lookup 2 ls =>
'B'
```

```
lookup key ((y, value):t) | y == key = value
                          | otherwise = lookup key t
```

Что будет результатом работы, если ключа в списке пар нет?

Введем новый тип данных:

```
data Maybe a = Nothing | Just a
```

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b    -- lookup 2 ls => Just
'B'
```

```
-- lookup 5 ls => Nothing
```

```
lookup _ [] = Nothing
```

```
lookup key ((y, value):t) | y == key = Just value
                          | otherwise = lookup key t
```

```
isJust, isNothing :: Maybe a -> Bool
```

```
fromJust :: Maybe a -> a    -- выдает ошибку, если применяется к Nothing
```

```
fromMaybe :: a -> Maybe a -> a -- задано значение "по умолчанию"
```

Работа с функциональным представлением графа

```
-- функция преобразования графа из спискового представления в функциональное
convertL2F :: LGraph -> FGraph
convertL2F s = (length s, \x y -> (let (Just z) = lookup x s in elem y z))
```

Запрограммируем обход графа «в ширину»

```
type Graph = (Int, (Int->Int->Bool))
type Set    = Int -> Bool
```

Работа с множествами в функциональном представлении:

```
empty :: Set
empty e = False

(+++), (-- -) :: Set -> Set -> Set
(s1 +++ s2) e = s1 e || s2 e
(s1 -- - s2) e = s1 e && not (s2 e)

list :: Int -> Set -> [Int]
list n s = filter s [1..n]

isEmpty :: Int -> Set -> Bool
isEmpty n s = null (list n s)
```

Множество вершин графа, инцидентных заданной:

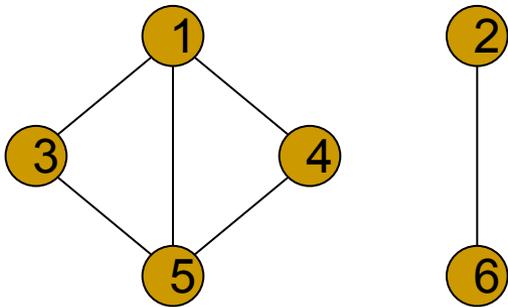
```
neib :: Int -> Graph -> Set
neib i (n, f) = f i

neighbors :: Int -> Graph -> [Int]
neighbors i (n, f) = list n (f i)
```

Пример обработки графа

Получить список вершин, достижимых из заданной:

```
traverse :: Int -> Graph -> Set
traverse i (n, f) = trav empty (==i) where
  trav passed front =
    if isEmpty n front
    then empty
    else front +++
        trav newPassed
            ((foldr (+++) empty (map f (list n front))) ==-
newPassed)
  where newPassed = passed +++ front
```



```
gr :: Graph
gr = (6, f) where
  f 1 3 = True
  f 1 4 = True
  f 1 5 = True
  f 2 6 = True
  f 3 5 = True
  f 4 5 = True
  f a b | a > b = f b a
  f a b = False
```

```
list 6 (traverse 1 gr) => [1, 3, 4, 5]
```

2.4. Классы в *Haskell*.

Класс определяет набор операций (функций). Тип данных принадлежит некоторому классу, если для него определены все функции, объявленные в классе.

```
class Eq t where
    (==), (/=) :: t -> t -> Bool
```

Мы объявляем, что некоторый тип данных принадлежит этому классу, с помощью определения «экземпляра» класса.

```
instance Eq Bool where
    True  == True    = True
    False == False   = True
    _     == _       = False
    x     /= y       = not (x == y)

instance (Eq t)=> Eq [t] where
    []     == []      = True
    (x1:s1) == (x2:s2) = (x1 == x2) && (s1 == s2)
    _     == _       = False
```

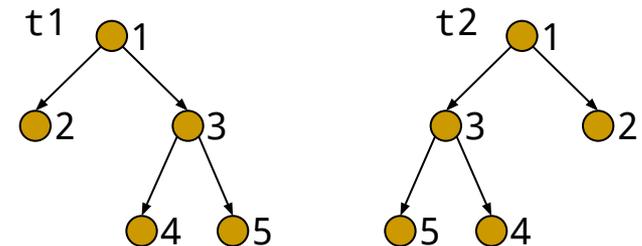
Пример: определение операций сравнения над деревьями

```
data Tree a = Empty |  
            Node (Tree a) a (Tree a)
```

```
instance (Eq t) => Eq (Tree t) where  
    Empty          == Empty          = True  
    (Node t11 n1 tr1) == (Node t12 n2 tr2) =  
        (n1 == n2) && (t11 == t12) && (tr1 == tr2)  
    _ == _ = False
```

```
instance (Eq t) => Eq (Tree t) where  
    Empty          == Empty          = True  
    (Node t11 n1 tr1) == (Node t12 n2 tr2) =  
        (n1 == n2) && (((t11 == t12) && (tr1 == tr2)) ||  
        ((t11 == tr2) && (tr1 == t12)))  
    _ == _ = False
```

```
t1 = Node (Node Empty 2 Empty) 1  
      (Node (Node Empty 4 Empty) 3  
            (Node Empty 5 Empty))  
t2 = Node (Node (Node Empty 5 Empty) 3  
          (Node Empty 4 Empty)) 1  
      (Node Empty 2 Empty)
```



t1 == t2 ?

Вывод значений различных типов в виде строки

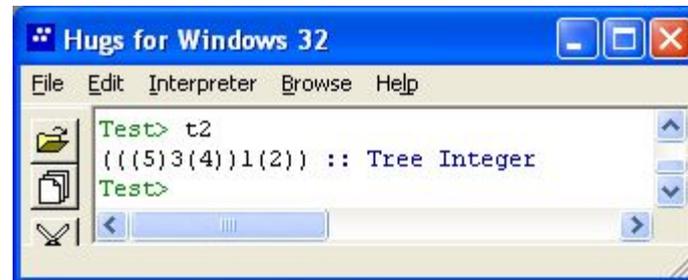
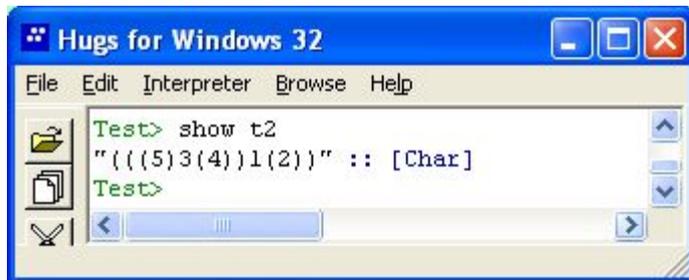
Немного упрощенная версия класса для вывода значений в виде строки:

```
class Show t where
  show      :: t -> String
  showsPrec :: Int -> t -> String -> String
  show v    = showsPrec 0 v []
  showsPrec _ v s = show v ++ s

instance (Show t) => Show (Tree t) where
  showsPrec _ Empty      = id
  showsPrec _ (Node t1 n tr) =
    ('(' :) . (shows t1) . (shows n) . (shows tr) . (')' :)
```

Здесь используется также стандартная функция `shows`, которая определена следующим образом:

```
shows :: (Show a) => a -> String -> String
shows = showsPrec 0
```



Расширение классов

```
class (Eq t) => Ord t where
  (<), (<=), (>), (>=) :: t -> t -> Bool
  a < b = not (a >= b)
  a <= b = not (a > b)
  a > b = not (a <= b)
  a >= b = not (a < b)
```

```
instance (Ord t) => Ord (Tree t) where
  Empty <= _ = True
  (Node t11 n1 tr1) <= (Node t12 n2 tr2) =
    (t11 <= t12) && (n1 <= n2) && (tr1 <= tr2)
  _ <= _ = False
  t1 < t2 = (t1 <= t2) && (t1 /= t2)
```

Это «плохое» определение операций сравнения.

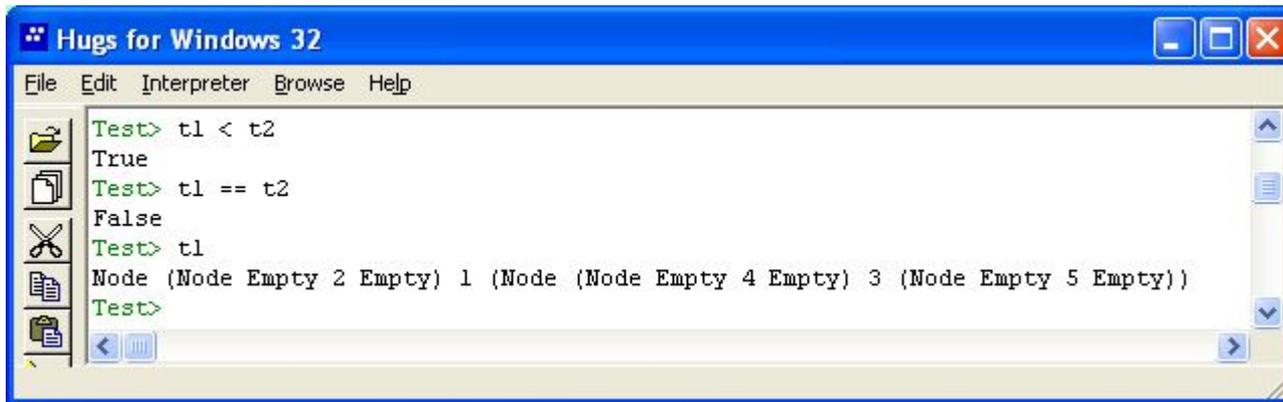
Так определенные операции не обладают обычными свойствами для операций сравнения.

Задание: определить операции сравнения деревьев так, чтобы для них выполнялись обычные свойства линейного порядка.

Стандартная реализация операций для встроенных классов

```
data Tree a = Empty |  
            Node (Tree a) a (Tree a)  
              deriving (Eq, Ord, Show)
```

- объекты равны, если равны все составляющие их части
- сравнение происходит в лексикографическом порядке составляющих объекты конструкторов: `Empty < (Node t1 n t2)` для любых `t1`, `n` и `t2`
- преобразование в строку: выводятся имена конструкторов и их аргументы



```
Hugs for Windows 32  
File Edit Interpreter Browse Help  
Test> t1 < t2  
True  
Test> t1 == t2  
False  
Test> t1  
Node (Node Empty 2 Empty) 1 (Node (Node Empty 4 Empty) 3 (Node Empty 5 Empty))  
Test>
```

