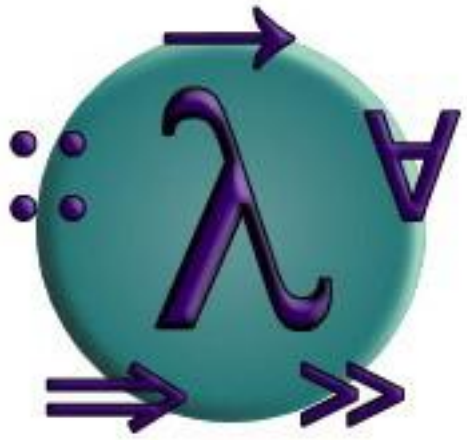
# PROGRAMMING IN HASKELL



Определение функций

# Условные выражения

Как и в большинстве языков программирования, функции могут быть определены с помощью условных выражений.

```
abs  :: Int → Int
abs n = if n ≥ 0 then n else -n
```

Abs берет целое  n and возвращает  n в абсолютном значении

Условные выражения могут быть вложенными:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
           if n == 0 then 0 else 1
```

Note:

- В Haskell, условные выражения всегда должны иметь ветвь else, что позволяет избежать возможных проблем неоднозначности с вложенными условиями.

# Выражения охраны

В качестве альтернативы условий, функции могут быть определены с помощью  уравнений охраны.

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

Тот же пример, но с использованием охраны.

Охраняемые уравнения могут быть использованы в случае нескольких условий:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

Note:

- otherwise определяет значение для всех остальных случаев

# Pattern Matching (Образцы)

Многие функции определяются с помощью сопоставления аргументов с образцами

```
not      :: Bool → Bool
not False = True
not True  = False
```

not отображает False в True, и True в False.

Функции могут быть определены различными способами  с использованием образцов. Например:

```
(&&)          :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

Может быть определена более  компактно

```
True && True = True
_    && _    = False
```

Но данное определение является более эффективным, т.к. так как позволяет избежать вычисление  второго аргумента, если первый аргумент является ложным:

```
True  && b = b
False && _ = False
```

Note:

- Символ подчеркивания _ является образцом, соответствующими любому значению аргумента.

- Образцы сопоставляются по порядку. Например, следующее определение всегда будет возвращать False:

```
_    && _    = False
True && True = True
```

- Образцы не должны повторять переменные. Например, следующее определение даст ошибку

```
b && b = b
_ && _ = False
```

# Работа со списками

Каждый непустой список строится путем многократного использования оператора (:) “cons” , который добавляет элемент в начало списка.

[1,2,3,4]

Означает 1:(2:(3:(4:[]))).

Списковые функции используют образец (шаблон) x:xs .

```
head       :: [a] → a
head (x:_)  = x

tail       :: [a] → [a]
tail (_:xs) = xs
```

head и tail возвращают из любого непустого списка первый элемент и оставшуюся часть списка

Note:

- x:xs  соответствует непустому списку:

```
> head []
ERROR
```

- x:xs должен быть заключен в скобки, т.к. применение  функции имеет более высокий приоритет, чем cons (:). Например, такое определение будет ошибочным:

```
head x:_ = x
```

# Лямбда-выражения

Функции могут быть построены без указания имени функции с использованием  лямбда-выражения.

$$\lambda x \rightarrow x + x$$

Безымянная (анонимная) функция, которая принимает число x и возвращает результат x + x.

Note:

- Символ λ является греческой буквой лямбда, на клавиатуре набирается как обратный слэш \.
- В математике для обозначения безымянных функций используется символ □ , в нашем случае x □ x + x.
- В Haskell, использование λ символа для обозначения безымянных функций идет от лямбда-исчисления, на теории функций которых базируется Haskell

# Для чего можно использовать?

Лямбда-выражения могут быть использованы как формальное средство определения каррированых функций.

Например:

```
add x y = x + y
```

означает

```
add = λx → (λy → x + y)
```

Лямбда-выражения могут быть также использованы при определении функций, которые возвращают функции в качестве результата.

Например:

```
const    :: a → b → a
const x _ = x
```

Более естественно определяется

```
const  :: a → (b → a)
const x = λ_ → x
```

Лябда выражения могут  использоваться, чтобы избежать именования функции, которые используются только один раз.
Например:

```
odds n = map f [0..n-1]
        where
          f x = x*2 + 1
```

Может быть упрощена

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

# Sections

An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.

Например:

```
> 1+2
3

> (+) 1 2
3
```

# Виды программ

Программы на Haskell бывают двух видов: это *приложения* (executable) и *библиотеки* (library). Приложения представляют собой исполняемые файлы, которые решают некоторую задачу, к примеру – это может быть компилятор языка, сортировщик данных в директориях, календарь, или цитатник на каждый день, любая полезная утилита. Библиотеки тоже решают задачи, но решают их внутри самого языка. Они содержат отдельные значения, функции, которые можно подключать к другой программе Haskell, и которыми можно пользоваться.
Программа состоит из *модулей* (module). И здесь работает правило: один модуль – один файл. Имя модуля совпадает с именем файла. Имя модуля начинается с большой буквы, тогда как файлы имеют расширение .hs. Например **FirstModule**.hs.

# Описание модуля

---

-- шапка

**module Имя(определение1, определение2,..., определениеN) where**

**import Модуль1(...)**

**import Модуль2(...)**

**...**

---

-- определения

**определение**1

**определение**2

**...**

Каждый модуль содержит набор определений. Относительно модуля определения делятся на *экспортируемые* и *внутренние*. Экспортируемые определения могут быть использованы за пределами модуля, а внутренние – только внутри модуля, и обычно они служат для выражения экспортируемых определений. Модуль состоит из двух частей – шапки и определений.

# Декларативная и композиционная запись

- В Haskell существует несколько встроенных выражений, которые облегчают построение функций и делают код более наглядным. Их можно разделить на два вида: выражения, которые поддерживают *декларативный стиль* (declarative style) определения функций, и выражения которые поддерживают *композиционный стиль* (expression style).
- Что это за стили? В декларативном стиле определения функций больше похожи на математическую нотацию, словно это предложения языка. В композиционном стиле мы строим из маленьких выражений более сложные, применяем к этим выражениям другие выражения и строим ещё большие.
- В Haskell есть полноценная поддержка и того и другого стиля. Выбор стиля скорее дело вкуса, существуют приверженцы и того и другого стиля, поэтому разработчики Haskell не хотели никого ограничивать.
- where-**выражения – декларативный стиль**
- let-**выражения –композиционный стиль**
- **Более подробно ru-Haskell-book-1.pdf стр. 59**

```
square a b c = sqrt(p * pa * pb* pc)
      where p = (a + b + c) / 2
                pa = p -a
                pb= p -b
                pc = p –c

square a b c = let p = (a + b + c) / 2
                   in sqrt ((let pa = p -a in p * pa) *
                                (let pb= p -b
                                 pc = p -c
                              in pb* pc))
```

# функции

- Скоро в армию!
- Функция определяет годность к армии , в зависимости от индекса массы тела.
- ИМТ =вес/ рост в квадрате

Параметр - индекс массы тела bmi

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = “must be getting fat"
    | bmi <= 25.0 = ” it's all right"
    | bmi <= 30.0 = ” need to lose weight!!"
    | otherwise   = ”urgently needs to lose weight !!!"
```

# функции

- Скоро в армию!
- Функция определяет годность к армии , в зависимости от индекса массы тела.
- ИМТ =вес/ рост в квадрате

2 параметра – вес, рост **weight height**

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    |  weight / height ^ 2  <= 18.5 = “must be getting fat!"
    |  weight / height ^ 2  <= 25.0 = ” it's all right "
    |  weight / height ^ 2  <= 30.0 = ” need to lose weight!!"
    | otherwise   = ” urgently needs to lose weight !!! "
```

# функции

- Скоро в армию!
- Функция определяет годность к армии , в зависимости от индекса массы тела.
- ИМТ =вес/ рост в квадрате

2 параметра, сам индекс считается в функции **where bmi = weight / height ^ 2**

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    |  bmi  <= 18.5 = “must be getting fat!"
    |  bmi  <= 25.0 = ” it's all right "
    |  bmi  <= 30.0 = ” need to lose weight!! "
    | otherwise   = ” urgently needs to lose weight !!! "
   where bmi = weight / height ^ 2
```

# функции

- Скоро в армию!
- Функция определяет годность к армии , в зависимости от индекса массы тела.
- ИМТ – вес / рост в квадрате

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    |  bmi  <=  skinny = “must be getting fat!"
    |  bmi  <=  normal = ”it's all right "
    |  bmi  <=  fat  = ” need to lose weight!! "
    | otherwise   = ” urgently needs to lose weight !!! "
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

# функции

- Скоро в армию!
- Функция определяет годность к армии , в зависимости от индекса массы тела.
  - ИМТ – вес / рост в квадрате

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    |  bmi  <=  skinny = “must be getting fat!"
    |  bmi  <=  normal = ” it's all right "
    |  bmi  <=  fat  = ” need to lose weight!! "
    | otherwise   = ” urgently needs to lose weight !!! "
    where bmi = weight / height ^ 2
          (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

# Скоро в армию!

- Функция определяет годность к армии , в зависимости от индекса массы тела.
- ИМТ =вес/рост в квадрате

```
bmiTell :: (RealFloat a) =>a->a->String
bmiTell weighth eight
    |bmi<=skinny=“must be getting fat!"
    |bmi<=normal=”it's all right"
    |bmi<=fat =”need to lose weight!!"
    |otherwise=”urgently needs to lose weight !!!"
    where bmi=weight/height^2
        (skinny, normal, fat)=(18.5,25.0,30.0)
```

# let <bindings> in <expression>

```
cylinder::(RealFloat a )=> a -> a -> a
cylinder r h=
    let sideArea=2*pi*r*h
        topArea=pi*r^2
    in  sideArea + 2 * topArea
```

```
ghci>[let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
ghci>4 *(let a = 9 in a+1) + 2
42
```

# Case expressions

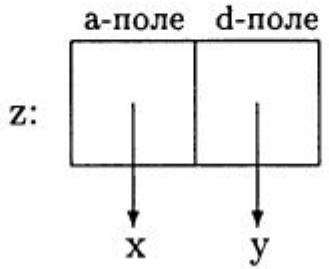
```
head‘ :: [a]->a
head‘ [] =error "No head for empty lists!"
head‘ (x:_) =x

head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x

Case expression of pattern -> result
                   pattern -> result
                   pattern -> result
```

# Примеры

```
describeList::[a]->String
describeList xs = "The list is” ++ case xs of [] ->"empty."
                                              [x]->"a singleton list."
                                               xs->"a longer list."
describeList :: [a] -> String
describeList xs ="The list is"++ what xs
    where what [] ="empty."
          what [x] ="a singleton list."
          what xs ="a longer list."
```

# ДОПОЛНИТЕЛЬНО: Программная реализация

Каждый объект (значение) занимает в памяти компьютера какое-то определенное место. Однако в парах хранятся не сами значения объектов, а указатели на них, поэтому атомы — это указатели (адреса) на ячейки, в которых содержатся объекты. В этом случае пара `z = (x:y)` графически может быть представлена так, как показано на следующем рисунке.

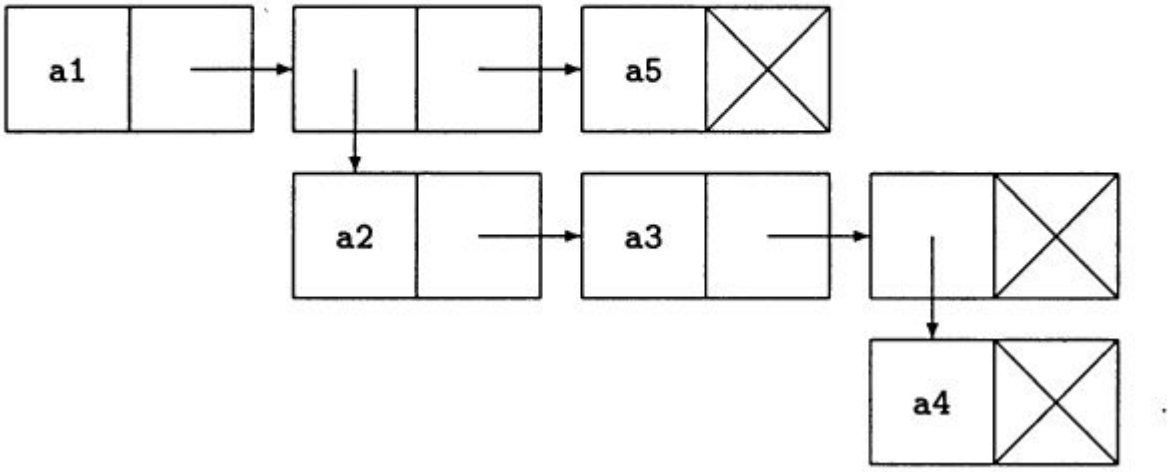



Именно адрес пары ячеек памяти, где содержатся указатели на **x** и **y**, и есть объект **z**. Как видно на рис. 2.1, каждая пара представлена двумя адресами — указателем на голову и указателем на хвост. Традиционно первый указатель называется «*a*-поле», а второй указатель — «*d*-поле»[2].

cAr cDr ( из LISP)

«contents of address register» «contents of decrement register»

Таким образом, списочная структура, которая рассмотрена несколькими абзацами ранее (`[a1, [a2, a3, [a4]], a5]`), может быть представлена так, как показано на следующем рисунке:



# Общий вид определения функции

```
<function_name> <patterns> "=" <expression>
```

# КЛОЗ

Клоз (от англ. *clause*) представляет собой запись одного варианта вычисления некоторой функции, зависящий от вида образцов, записанных около имени функции (см. общий вид определения функции — терм `patterns`). По определению, клозы функций выглядят так (запись в математической нотации):

$$\textbf{def}\ fp_1, \ldots, p_n = expr,$$

где:

1) `def` и = — константы абстрактной математической нотации;

2) $f$ — имя определяемой функции;

3) $p_i, i = \overline{1,n}$ — последовательность образцов;

4) $expr$ — выражение, значение которого возвращается функцией $f$.

```
function p1 p2 ... pn = expression
```

# образцы

Образцом (калька с англ. — *pattern*) называется выражение, построенное с помощью операций конструирования данных, которое используется в определениях функций для сопоставления с данными. В синтаксисе языка Haskell переменные обозначаются строчными буквами (по крайней мере, должны начинаться со строчной буквы), константы используются непосредственно.

Примеры образцов:

1) `5` — просто числовая константа;

2) `x` — просто переменная;

3) `x:(y:z)` — пара;

4) `[x, y]` — список.

Пустой список `[]` и непустой список `(x:xs)`.

# Образцы и клозы на примере last

```
last [x]    = x
last (x:xs) = last xs
```

В этом определении представлены два клоза, в каждом из которых используется по одному образцу (в силу того, что функция `last` принимает на вход единственный аргумент — список). В первом клозе используется образец `[x]`, представляющий собой список из одного аргумента. Во втором клозе имеется образец `(x:xs)`, представляющий собой список из более чем одного элемента.

```
last []  = error "Функция last не˜может обработать пустой список."
```

```
last []     = error "Функция last не˜может обработать пустой список."
last [x]    = x
last (_:xs) = last xs
```

образцы вида `(n + k)`

```
-- Факториал
fac 0       = 1
fac (n + 1) = (n + 1) * fac n
```

# Использование λ исчислений

```
add = \x y -> x + y
```

$\lambda xy.(x + y),$

```
inc = \x -> x + 1
```

$\lambda x.(x + 1).$

То есть видно, что $\lambda$-абстракции кодируются на языке Haskell просто. Символ ($\lambda$) заменяется на символ (\), а символ (.) (точка) заменяется на стрелку (->). Остальное дается без изменения (естественно, принимая во внимание синтаксис языка Haskell).

```
add = \x -> \y -> x + y
```

$\lambda x.\lambda y.(x + y).$

# Инфиксный способ записи функций

Инфиксная запись — это запись символа операции или имени функции между своими аргументами в том случае, если операция или функция принимает на вход два аргумента. Инфиксная запись противопоставляется префиксной и постфиксной. Префиксная запись, то есть запись имени функции перед своими аргументами, — обычная запись для функциональных языков программирования, в том числе и для языка Haskell. Постфиксная запись (иногда называемая обратной польской записью) используется при построении трансляторов.

```
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x : (xs ++ ys)
```