**Haskell**

**Дефиниция на величина:**

name :: type

name = expression

**Дефиниция на функция:**

name :: t1 -> t2 -> … ->tk -> t -- където:

name x1 x2 … xk -- name - име на функция

|g1 = e1 -- x1 .. xk - формални параметри; xi е от тип ti; t - тип резултата

|g2 = e2 -- e1 ..ek e - изрази, определящи резултата

……… -- g1 ,g2 .....- булеви израз

|otherwisе е

**Условен израз**

If <условие> then <израз1> else <израз2>

max2 :: Int -> Int ->Int

max2 x y = if x>=y then x else y

**Обръщение към функция :**

name f1 f2 … fk

**Локални дефиниции**

f p1 … pn let p1 =e1

| g1 = e1 p2 = e2

| otherwise = e …

where v1:: t1 -> … ->tk ->t pn = en

v1 a1 … ak = expr in expr

където v1 се използва в е1…е

**Оператори**

+ , -, \*, / - лявоасоциативни 23+12 = (+) 23 12

\*\*, ^ - дясноасоциативни ==,/=,<,<=,>,=> -не са асоциативни

**Библиотеки**

import Data.Char – включва библиотека Data.Char с всички нейни елементи

import Data.Char hiding (name) - не включва дадена дефиниция

import Data.Char (name1, name2) – използваме самo някои дефиниции

**Модул** – може да съдържа множество от дефиниции на величини и функции

Извиква се с името на файла, в който се намира (със същото име)

module <name> (name1, name2,….., nameN) where

import Module1(…)

…

import Modulep(…)

definition1

…

definitionk

**Съпоставяне с образец**

func x y -- еквивалентна на

| x==1 = y -- func 1 y = y ‘\_‘ - символ за безусловно съпоставяне

|otherwise =x -- func x y = x

**Типове данни**

|  |  |
| --- | --- |
| **Тип Bool - True False**  Операции && - and, || - or, not – not  True > False  **Int , Integer –**  Int – ограничено , Integer – не  Операции : +,-,\*  ==, /= ,<,<=,>,>=  ^ - степенуване  div - целочислено делене  mod – остатък  abs - абсолютна стойност  negate - променя знака  gcd - най-голям общ делител | **Float, Double, Rational**  Операции : +,-,\*  ==, /= ,<,<=,>,>=  ^ - степенуване на степен - цяло число  \*\* - сгепенуване с всякакво число  abs - абсолютна стойност  negate - променя знака  sin, cos, tan, asin, acos, atan  ceiling  floor  round  fromInt – Int ->Float  log - ln  logBase – Float ->Float - >Float  pi  signum - намира знака  sqrt – квадратен корен |
| **Списъци** – елементите са от един тип  [e1, , en] или [ ]  head – намира първия елемент на непразен списък  tail – намира списък без първия му елемент  length – намира броя на елементите на списък  reverse – намира обратния списък  [ 1, 2, 3] ++ [ 4, 5] => [1, 2, 3, 4, 5]  *Записване на списък*  1 : 2 : 3 : [ ] => [1, 2, 3]  [1 .. 4] => [1, 2, 3, 4]  [6, 4, .. 1] => [6, 4, 2]  *List comprehension*  [2\*n| n<-[1, 2, 3, 4] , even n] => [4,8] | Функции:  !! [a]->Int->a получава n-ти елемент на списъка  init [a] -> [a] списък без последен елемент  split at Int->[a]->([a],[a]) разделя в указаното място  zip [a]->[b]->[(a,b)]  unzip [(a,b)]->([a],[b])  sum [Int]->Int събира елементите  prod [Int]->Int умножава елементите  take Int->[a] -> [a] взима първите n елемента  drop Int->[a] -> [a] изтрива първите n елемента  replicate Int->a->[a] създава списък от повторения  last [a] -> a намира последен елемент  concat [[a]]->[a] конкатенира списък от списъци |
| **Вектори** - (v1,v2,…,vn) където v1::t1 … vn::tn  Селектори на двойки (x,y):  fst (x,y) = x snd (x,y) = y |  |

**Функции от по-висок ред**

**map :: (a->b) -> [a] ->[b]** с аргументи f: a->b , [a1, a2, …. ,an] и резултат [f a1, f a2, ….., f an]

**filter :: ( a-> Bool ) -> [a] -> [a]** връща списък с елементи , удовлетворяващи условието

**zipWith :: (a->b->c) -> [a] -> [b] -> [c**] с аргументи f: a->b->c , [a1, a2, …. ,an] , [b1, b2, ….. , b3]

и резултат [f a1 b1 , f a2 b2, ….., f ak bk] където k=min{n,m}

**foldr :: (a->b->b) ->b ->[a] ->b** с аргументи f: a->b->b , начална стойност z от тип b , [x1, x2, …. ,xn]

и резултат от тип b , където foldr f z [x1, x2, … , xn] == f x1 (foldr f z [x2, … , xn])

**foldr1 –** върши същото като foldr но няма начална стойност и се прилага само над непразни списъци

**foldl :: (a->b->b) ->b ->[a] ->b** с аргументи f: a->b->b , начална стойност z от тип b , [x1, x2, …. ,xn]

и резултат от тип b , където foldl f z [x1, x2, … , xn] == (….(( z ‘f’ x1) ‘f’ x2) …. ) ‘f’ xk

**foldl1 –** върши същото като foldl но няма начална стойност и се прилага само над непразни списъци

**until b f x** където b::a->Bool , f :: a->a , x начална стойност от тип а

Докато за х не е вярно условието b , към него се прилага f

**Функции като върнати стойности**

**Ламбда изрази : \ x1 x2 …. xn -> expression** анонимно задава функция с параметри x1 x2 …. xn

**Многократно прилагане на числова функция : repeated :: (a->a) -> Int -> (a->a)**

**repeated f 1 = f**

**repeated f n = \x-> f(repeated f (n-1) x)**

**Композиция на функции : Ако f::b->c и g::a->b ,то f.g е от тип a->c и (f.g)x==f(g x) ‘.’ е дясноасоциативен**

**Безкрайни списъци Пр. ones::[int] ones = 1: ones**

**Задавяне : [n ..] [1 ..] == [1,2,3,……..]**

**[n , m ..] [1,3 ..] == [1,3,5,7,9,……]**

**вградена iterate - многократно прилага функцията**

**>iterate (\*3) 1 == [1,3,9,27….]**

**Дефиниция чрез определяне на обхвата**

**pytagTriple :: [(Int,Int,Int)]**

**pytagTriple = [(x,y,z) | z<-[3..] , y<-[2 .. z-1] , x<-[2 .. y-1] , x\*x+y\*y=z\*z]**

**Полиморфизъм**

**Параметричен полиморфизъм**

**length :: [a]->Int а – произволен тип**

**Специален полиморфизъм**

**(+):: Num a => a->a->a a – може да е само от клас Num**

**Класове – съвкупност от типове (екземпляри) и функции (методи)**

**Декларация на клас**

**class Name tv where Name – име на класа – с главна буква**

**Signature tv – променлива задаваща тип**

**Signature – съвкупност от имена и типове на функциите определящи класа**

**Дефиниция екземплярите на класа**

**instance Name Type where**

**Дефиниция на функциите от Signature**

**Основни класове**

**Num : поддържа (+), (\*) , (-) , negate, abs, signum, fromIntegral**

**Enum : поддържа succ , pred , toEnum, fromEnum , enumFrom , enumFromThen , enumFromTo , enumFromThenTo**

**Eq : поддържа (==) , (/=)**

**Производни класове**

**class Eq a => Ord a where class ( Num a , Ord a) => Real a where**

**compare :: a->a->a toRational :: a->Rational**

**(<) :: a->a->Bool**

**(>) :: a->a->Bool**

**(<=) :: a->a->Bool**

**(>=) :: a->a->Bool**

**min :: a->a->a**

**max :: a->a->a**