Тема 2

Основни примитивни типове данни в Haskell. Проектиране на програми на Haskell

Примитивни типове данни в Haskell

Булеви стойности (Bool)

Булевите стойности (константи) True („истина“) и False („лъжа“) представят резултатите от различни видове проверки.

Например:

- сравнение на две числа за равенство
- проверка дали едно число е по-малко или равно на друго
- сравнение на два символни низа за съвпадение

Булевият тип в Haskell се нарича Bool. Булеви константи са True и False, а вградените Булеви оператори, поддържани от езика, са:

&&	and	(конюнкция)
||	or	(дизюнкция)
not	not	(отрицание)

Таблици на истинностните стойности на Булевите оператори

t	not t
T	F
F	T

t1	t2	t1 && t2	t1 || t2
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Примерна дефиниция на функция, която реализира операцията XOR (изключващо „или“):

```
exOr :: Bool -> Bool -> Bool

exOr x y = (x || y) && not (x && y)
```

Литерали и дефиниции

Булевите константи True и False, също както и числата, са известни под името **литерали** (изрази, които не се нуждаят от истинско оценяване, защото стойността на всеки от тях съвпада с името му).

Литералите True и False (както и всички други литерали) могат да бъдат използвани като аргументи при дефинирането на функции.

Примери:

```
myNot :: Bool -> Bool
myNot True   = False
myNot False  = True


exOr :: Bool -> Bool -> Bool
exOr True  x   = not x
exOr False x   = x
```

Дефинициите, които използват константите True и False в лявата страна на равенства (функционални уравнения), обикновено се четат по-лесно от дефинициите, които в лявата си страна съдържат само променливи.

Разгледаните по-горе примери са илюстрация на механизмите на **съпоставяне с образец** (pattern matching) в Haskell, които ще бъдат разгледани по-нататък в курса.

Цели числа (Int и Integer)

Целите числа (числата с фиксирана точка) в Haskell са от тип Int или Integer.

За представяне на целите числа от тип Int се използва фиксирано пространство (32 бита), което означава, че типът Int съдържа краен брой елементи. Константата **maxBound** има за стойност максималното число от тип Int (2^{31}-1 = 2147483647).

За работа с цели числа с неограничена точност може да бъде използван типът Integer.

Haskell поддържа следните вградени оператори за работа с цели числа:

+	Сума на две цели числа.
*	Произведение на две цели числа.
^	Повдигане на степен; 2^3 е 8.
-	Разлика на две цели числа (при инфиксна употреба: a-b) или унарен минус (при префиксна употреба: -a).
div	Частно при целочислено деление, например div 14 3 е 4. Може да се запише и инфиксно: 14 `div` 3.
mod	Остатък при целочислено деление, например mod 14 3 или 14 `mod` 3.
abs	Абсолютната стойност на дадено цяло число (числото без неговия знак).
negate	Функция, която променя знака на дадено цяло число.

Забележка. Чрез заграждане на името на всяка двуаргументна функция в обратни апострофи (backquotes) е възможно записът на обръщението към тази функция да стане инфиксен.

Вградени оператори за сравнения:

>	greater than
>=	greater than or equal to
==	equal to (може да се използва и при аргументи от други типове)
/=	not equal to
<=	less than or equal to
<	less than

Пример. Дефиниция на функция, която проверява дали три цели числа (три числа от тип Int) са еднакви

```
threeEqual :: Int -> Int -> Int -> Bool
threeEqual m n p = (m==n) && (n==p)
```

Overloading
(додефиниране)

Както целите числа, така и Булевите стойности могат да бъдат сравнявани за равенство (съвпадение, еквивалентност) с помощта на оператора ==.

Нещо повече, операторът == може да се използва за сравнение (проверка за равенство) на стойности от всеки тип t, за който равенството е добре дефинирано.

Това означава, че операторът (==) е от тип
Int -> Int -> Bool
Bool -> Bool -> Bool
и по-общо
t -> t -> Bool ,
където типът t поддържа проверката за равенство.

Използването на едно и също име за означаване на различни операции се нарича **overloading (додефиниране на оператора)**.

Реални числа (числа с плаваща точка: Float, Double и Rational)

За представянето на числата с плаваща точка от тип Float в Haskell се използва фиксирано пространство, което рефлектира върху точността на работата с този тип числа.

Допустим запис:

- като десетични дроби, например

0.31426
-23.12
567.345
4513.0

- scientific notation (запис с мантиса и порядък), например

231.61e7 $231.61 \times 10^{7} = 2316100000$
231.61e-2 $231.61 \times 10^{-2} = 2.3161$
-3.412e03 $-3.412 \times 10^{3} = -3412$

Типът Double се използва за работа с числа с плаваща точка с по-голяма (двойна) точност, а типът Rational се използва за представяне на реални (по-точно, рационални) числа с пълна (неограничена) точност.

Някои вградени аритметични оператори:

+ - *	Float –> Float –> Float
/	Float –> Float –> Float
^	Float –> Int –> Float (x^n за неотрицателно цяло n)
**	Float –> Float –> Float (x^y)
== /= <	Float –> Float –> Bool
> <= >=	Float –> Float –> Bool
signum	Float –> Float (връща резултат 1.0, 0.0 или –1.0)
sqrt	Float –> Float

Някои специфични функции за работа с реални числа
(за осъществяване на преход между цели и реални числа):

ceiling, floor, round	Float -> Int	конвертиране на реално число в цяло чрез закръгляне нагоре, закръгляне надолу или закръгляне до най-близкото цяло число
fromIntegral	Int -> Float	конвертиране на цяло число в реално

Знакове (characters, Char)

Представляват отделни знакове, заградени в единични кавички (апострофи), например ‘d’ или ‘3’.

Връзка между знаковете и техните ASCII кодове:

```
ord :: Char -> Int
chr :: Int -> Char
```

Пример 1. Конвертирането на малки букви към главни изисква към съответния код да бъде прибавено определено отместване:

```
offset :: Int
offset = ord ‘A’ – ord ‘a’

toUpper :: Char -> Char
toUpper ch = chr (ord ch + offset)
```

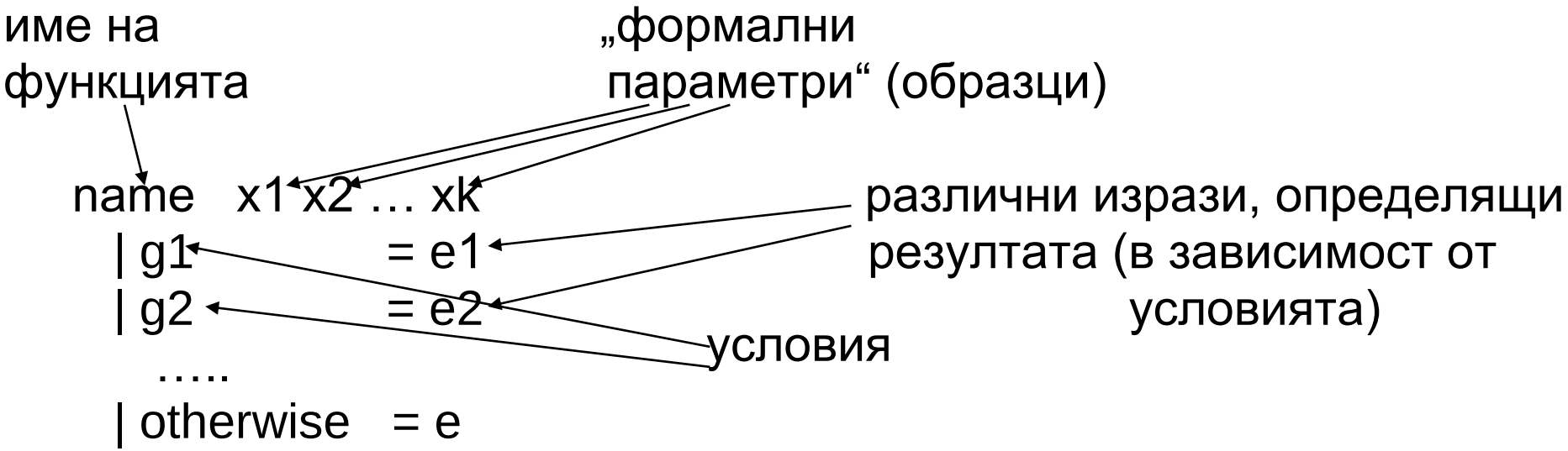
Пример 2. Проверка дали даден знак е цифра:

```
isDigit :: Char -> Bool
isDigit ch = (‘0’ <= ch) && (ch <= ‘9’)
```

Програмиране на условия (guards)

Условието („охраняващ“ израз, guard) е Булев израз. Условия се използват, когато трябва да се опишат различни случаи в дефиницията на функция.

Общ вид на дефиниция на функция с условия:



Забележка. Клаузата otherwise не е задължителна.

Примери

```
max :: Int -> Int -> Int
max x y
    | x >= y     = x
    | otherwise  = y

maxThree :: Int -> Int -> Int -> Int
maxThree x y z
    | x >= y && x >= z   = x
    | y >= z             = y
    | otherwise          = z
```

Когато трябва да се приложи дадена функция към дадено множество от изрази (стойности на съответните аргументи), е необходимо да се установи кой от поредните случаи в дефиницията на функцията е приложим.

За да се отговори на този въпрос, трябва последователно да се оценят охраняващите изрази (условията), докато се достигне до първия срещнат, чиято оценка е True. Съответният израз от дясната страна на равенството определя резултата.

Примери

maxThree 4 3 2
?? 4>=3 && 4>=2
?? ⟶ True && True
?? ⟶ True
⟶ 4

maxThree 6 (4+3) 5
?? 6>=(4+3) && 6>=5
?? ⟶ 6>=7 && 6>=5
?? ⟶ False && True
?? ⟶ False
?? 7>=5
?? ⟶ True
⟶ 7

Предефиниране на функции от стандартната прелюдия на Haskell

Функцията max е вградена (дефинирана в стандартната прелюдия на Haskell, Prelude.hs). Това означава, че ако дефиниция от вида

```
max :: Int -> Int -> Int
```

се появи в някой скрипт, например maxDef.hs, то тази дефиниция ще бъде в конфликт със съществуващата дефиниция от Prelude.hs.

За да може да се предефинира дадено множество от функции от Prelude.hs, е необходимо съответните дефиниции да бъдат скрити чрез включване в началото на maxDef.hs на ред от вида

```
import Prelude hiding (max, min)
```

Това позволява да бъдат предефинирани функциите max и min.

Условни изрази

Аналогично на повечето езици за програмиране и в езика Haskell е възможно да се описват условни изрази в термините на конструкцията if … then … else … .

Общ вид на условен израз в Haskell:

if *condition* then *m* else *n*

Тук *condition* е Булев израз, а *m* и *n* са (еднотипни) изрази. Ако стойността на *condition* е True, то стойността на условния израз съвпада със стойността на *m*, в противен случай стойността на условния израз съвпада със стойността на *n*.

Пример

```
max :: Int -> Int -> Int
max x y
    = if x >= y then x else y
```

Проектиране и съставяне на програми на Haskell

Проектиране: етапът, предшестващ съставянето на детайлния програмен код.

- Анализ на задачата. Определяне на изискванията към проекта и достъпните ресурси за неговата реализация
- Определяне на типа/типовете на входните данни и типа на резултата
- Подбор на подходящо множество от тестови примери
- Декомпозиране на задачата на подзадачи и определяне на методите за решаване на отделните подзадачи

Рекурсия

Техника на програмиране, при която дефиницията на дадена функция или друг обект включва цитиране на самия обект (т.е. дефинирането на един обект може да се основава на позоваване на самия обект).

Компоненти на рекурсивните дефиниции:

- Прост/базов случай („дъно“ на рекурсията)
- Общ случай

Примерна задача. Да се състави програма за пресмятане на **n!** (n е дадено естествено число).

Решение

Първи начин
Използва се дефиницията на **n!**, според която:
n! = n.(n-1)!, n≥2,
1! = 1.

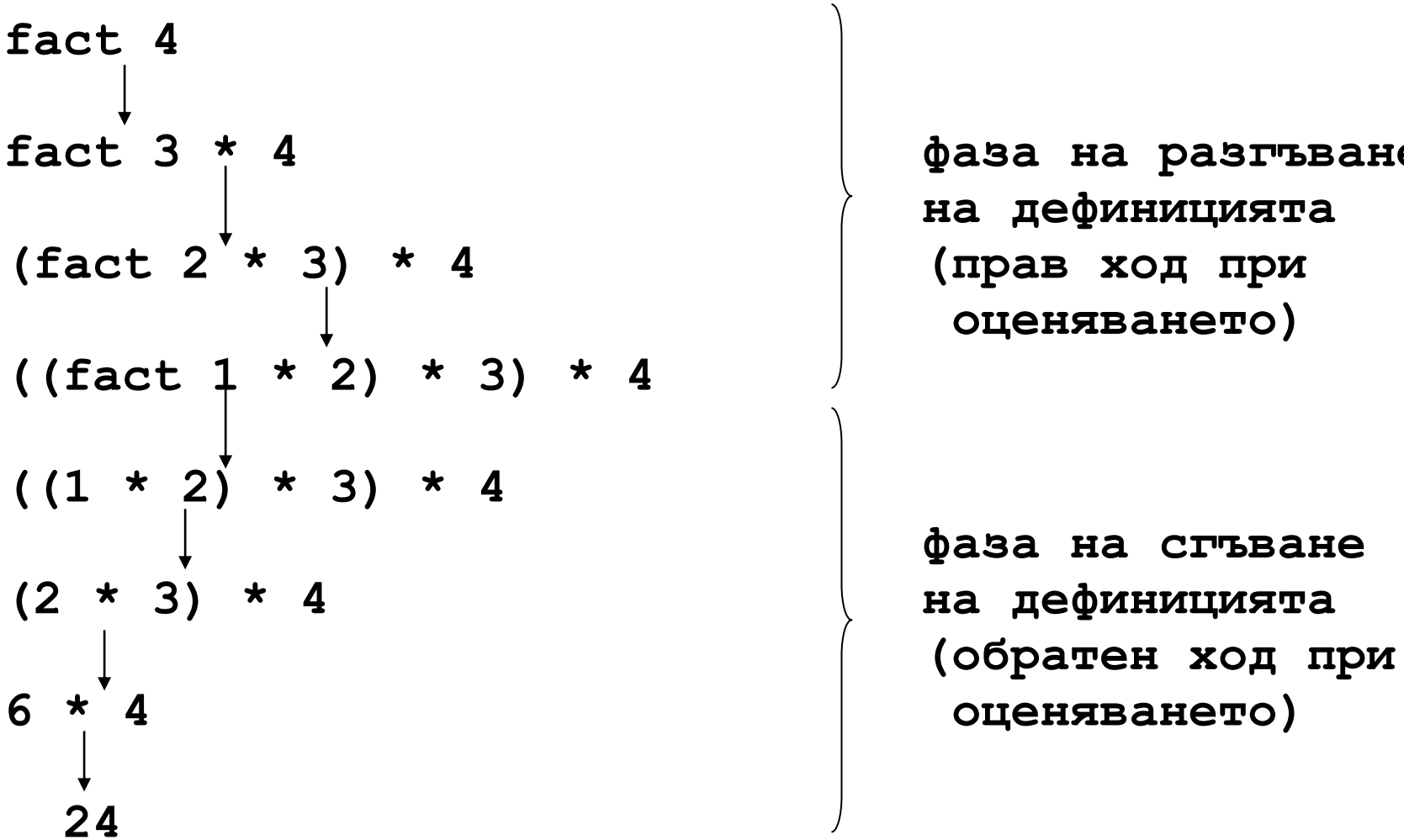
```
fact :: Integer -> Integer
fact n
  | n==1       = 1
  | otherwise = fact (n-1) * n
```

Забележка. Предложената дефиниция не е достатъчно прецизна (функцията няма да работи коректно при стойност на аргумента, по-малка от 1).

Рекурсия и оценяване на изрази

Оценяване на даден израз (обръщение към функция): пресмятане на стойността на този израз (на това обръщение към функция).

Ще проследим процеса на изпълнение (хода на оценяването) на обръщения към дефинираната по-горе рекурсивна функция за пресмятане на n!.


fact 4
fact 3 * 4
(fact 2 * 3) * 4
((fact 1 * 2) * 3) * 4
((1 * 2) * 3) * 4
(2 * 3) * 4
6 * 4
24
фаза на разгъване на дефиницията (прав ход при оценяването)
фаза на сгъване на дефиницията (обратен ход при оценяването)

Втори начин

Използва се дефиницията на **n!**, според която:

n! = 1.2.3. ... .n.

Според тази дефиниция **n!** може да се получи, като най-напред се умножи **1.2**, след това полученият резултат се умножи по **3**, след това - по **4** и т.н., докато се достигне **n** (умножава се накрая по **n** и когато поредният множител стане по-голям от **n**, пресмятането се прекратява). Необходимо е да се съхраняват частичният резултат (**product**) и поредният множител (**counter**). Правилата за промяна на стойностите на **product** и **counter** са:

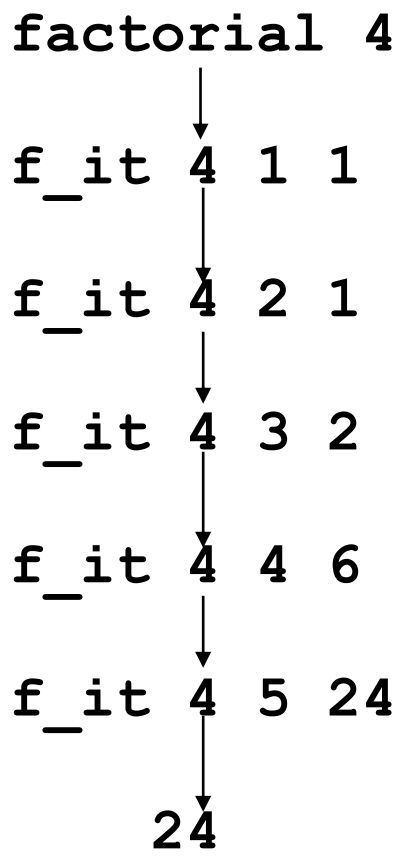
product := product*counter, counter := counter+1.

Първоначално **product = counter = 1**, а когато **counter** получи стойност, по-голяма от **n** (т.е. когато counter стане **n+1**), **product** ще има стойност **n!**.

```
factorial :: Integer -> Integer
factorial n = f_it n 1 1

f_it :: Integer -> Integer -> Integer -> Integer
f_it n count product
  | count>n      = product
  | otherwise    = f_it n (count+1) (product*count)
```

Проследяване на развитието на процеса (хода на изпълнението/оценяването на обръщение към така дефинираната функция):



Сравнение на двете решения

Общи черти:

- реализират пресмятането на една и съща математическа функция;
- броят на стъпките е пропорционален на **n**, т.е. и двата процеса са линейни;
- извършва се една и съща поредица от умножения (1.2, 2.3, ...) и съответно се получават еднакви междинни резултати.

Различия:

- при първия процес има *фаза на разгъване* (увеличаване на броя на участващите операции - в случая умножения) и след това - *фаза на сгъване* (намаляване на броя на операциите). При втория процес броят на участващите (означените) операции е постоянен;

- при изпълнението на първия процес (при използването на първата дефиниция) интерпретаторът трябва да запазва формираната верига от умножения, за да може по-късно да ги изпълни. При изпълнението на втория процес (при използването на втората дефиниция) текущите стойности на променливите **product** и **counter** дават пълна информация за текущото състояние.

Дефиниция. Процесите от първия тип се наричат ***рекурсивни***. При тях се поражда верига от обръщения към дефинираната функция с все по-прости в определен конкретен смисъл стойности на аргументите, докато се стигне до обръщение с т. нар. базов (прост, граничен) вариант на стойности на аргументите, след което започва последователно пресмятане на генерираните вече обръщения. Процесите от втория тип се наричат ***итеративни***. При тях във всеки момент състоянието на пресмятанията се описва (като при това може при необходимост да бъде прекъснато и после - възстановено) от няколко променливи (state variables, *променливи на състоянието*) и правило, с чиято помощ се извършва преходът от дадено състояние към следващото.

Недефинирани стойности на функции

Нека разгледаме следната дефиниция на функция за пресмятане на стойността на n!:

```
fac :: Int -> Int
fac n
  | n==0   = 1
  | n>0    = fac (n-1) * n
```

Ако се опитаме да оценим израза fac (-2), ще получим следното съобщение за грешка:

Program error: {fac (-2)}

Това съобщение се дължи на факта, че нашата функция (формално) не е дефинирана за отрицателни числа.

Една възможност за разрешаване на проблема е тази дефиниция да се разшири така, че функцията да е определена и за отрицателни стойности на аргумента, например

```
fac :: Int -> Int
fac n
  | n==0       = 1
  | n>0        = fac (n-1) * n
  | otherwise = 0
```

Друга възможност е свързана с използване на функцията ***error***, която предизвиква извеждане на определен (даден) текст (***символен низ***) на потребителския екран, последвано от ***прекратяване на оценяването***.

Например:

```
fac :: Int -> Int
fac n
  | n==0       = 1
  | n>0        = fac (n-1) * n
  | otherwise = error "factorial defined only on
                       non-negative integers"
```

Забележка. Функцията ***error*** е от тип

error :: String –> a

Вектори, списъци и символни низове в Haskell (въведение)

Вектори (n-торки, tuples)

Векторът (tuple) представлява наредена n-торка от елементи, при това броят n на тези елементи и техните типове трябва да бъдат определени предварително. Допуска се елементите на векторите да бъдат от различни типове.

Възможно е да се дефинира тип „вектор“ от вида (t_1,t_2, … ,t_n), който включва векторите (v_1,v_2, … ,v_n), за които v_1::t_1, v_2::t_2, … , v_n::t_n.

Примери

```
type ShopItem = (String,Int)
i1 :: ShopItem
i2 :: ShopItem
i1 = ("Salt: 1 kg",139)
i2 = ("Sugar: 0.5 kg",28)
```

Haskell поддържа множество **селектори** за стойностите от тип вектор. Такива са например функциите ***fst*** и ***snd***, които работят с двуелементни вектори:

```
fst (x,y) = x
snd (x,y) = y
```

При дефиниране на функции за работа с вектори често освен (вместо) селекторите се използва апаратът на съпоставянето с образец (pattern matching).

Пример 1

```
addPair :: (Int,Int) -> Int
addPair p = fst p + snd p
```

Пример 2

```
addPair :: (Int,Int) -> Int
addPair (x,y) = x+y
```

Образците могат да съдържат литерали и вложени образци, например

```
addPair (0,y) = y
addPair (x,y) = x+y

shift :: ((Int,Int),Int) -> (Int,(Int,Int))
shift ((x,y),z) = (x,(y,z))
```

Пример за използване на вектори: Намиране на минималното и максималното от две цели числа.

```
minAndMax :: Int -> Int -> (Int,Int)
minAndMax x y
   | x>=y          = (y,x)
   | otherwise     = (x,y)
```

Списъци

Списъкът в Haskell е редица от (променлив брой) елементи от определен тип. За всеки тип t в езика е дефиниран също и типът [t], който включва списъците с елементи от t.

Запис на списъците в Haskell:

[ ]: празен списък (списък без елементи). Принадлежи на всеки списъчен тип.

$[e_1,e_2, \ldots , e_n]$: списък с елементи $e_1, e_2, \ldots , e_n$.

Други форми на запис на списъци от числа, знакове (characters) и елементи на изброими типове:

- [n .. m] е списъкът [n, n+1, … , m]; ако n>m, списъкът е празен.

[2 .. 7] = [2, 3, 4, 5, 6, 7]
[3.1 .. 7.0] = [3.1, 4.1, 5.1, 6.1, 7.1]
[‘a’ .. ‘m’] = “abcdefghijklm”

- [n, p .. m] е списъкът, чийто първи два елемента са n и p, последният му елемент е m и стъпката на нарастване на елементите му е p-n.

[7,6 .. 3] = [7, 6, 5, 4, 3]
[0.0, 0.3 .. 1.0] = [0.0, 0.3, 0.6, 0.9]
[‘a’,’c’ .. ‘n’]= “acegikm”

Както се вижда от примерите, и в двата случая по-горе е възможно големината на стъпката да не позволява достигането точно на m.

Дефиниране на списък чрез определяне на неговия обхват (List Comprehension)

Синтаксис:

`[expr | q1, … , qk]` , където **`expr`** е израз, а **`qi`** може да бъде

- ***генератор*** от вида **`p <- lExp`**, където **`p`** е образец и **`lExp`** е израз от списъчен тип
- ***тест (филтър)***, **`bExp`**, който е булев израз

При това в q_i могат да участват променливите, използвани в q_1, q_2, … , q_{i-1}.

Пример 1

Да предположим, че стойността на ex е [2,4,7]. Тогава записът **`[2*n | n <- ex]`** означава списъка [4,8,14].

Пример 2

`[isEven n | n <- ex]` ⟶ **`[True,True,False]`**

Пример 3

`[2*n | n <- ex, isEven n, n>3]` ⟶ **`[8]`**

Пример 4

```
addPairs :: [(Int,Int)] -> [Int]
addPairs pairList = [m+n | (m,n) <- pairList]

addPairs [(2,3),(2,1),(7,8)] ⟶ [5,3,15]
```

Пример 5

```
addOrdPairs :: [(Int,Int)] -> [Int]
addOrdPairs pairLst = [m+n | (m,n) <- pairLst, m<n]
```

```
addOrdPairs [(2,3),(2,1),(7,8)] ⟶ [5,15]
```

Пример 6

Следващата функция намира всички цифри в даден низ:

```
digits :: String -> String
digits st = [ch | ch <- st, isDigit ch]
```

Тук isDigit е функция, дефинирана в Prelude.hs (isDigit :: Char -> Bool), която връща стойност True за тези знакове, които са цифри (‘0’, ‘1’, … , ‘9’).

Пример 7

Едно определение на обхвата на списък може да бъде част от дефиницията на функция, например

```
allEven xs = (xs == [x | x <- xs, isEven x])
allOdd xs  = ([] == [x | x <- xs, isEven x])
```

Символни низове (типът String)

Символните низове са списъци от знакове (characters), т.е. типът String е специализация на списъците:

type String = [Char]