Лекционни материали по функционално програмиране за специалност “Компютърни науки”

ПРОГРАМИРАНЕ НА HASKELL

М. Тодорова

Изработването им е подпомогнато от договор № 230 “Компютърно моделиране и софтуерен дизайн (теория, методология, обучение)” на ФНИ към СУ “Св. Кл. Охридски”, 2008.

октомври-декември, 2008 год.

Тема 1

Основни понятия в програмирането на езика Haskell. Дефиниране на величини и функции. Обръщения към функции. Функции и процесите, които те генерират. Основни типове данни в Haskell

1. Основни понятия в програмирането на езика Haskell

Haskell е език за строго функционално програмиране. Създаден е в края на 80-те години на 20-ти век. Именуван е на Haskell B. Curry, който е един от пионерите на ламбда смятането – математическата теория на функциите. Целта е да се интегрират в него хубавите черти на всички, съществуващи до момента функционални езици за програмиране.

Съществуват множество реализации на езика. Ще използваме Hugs 98 (версия 2006) – най-простата реализация на Haskell, която е и най-подходяща за целите на обучението. Последната е реализирана като интерпретатор, т.е. оценява изразите стъпка след стъпка, заради което е по-малко ефективна от реализациите като компилатор, които транслират Haskell програмата директно на машинния език на компютъра.

Допълнителна информация за езика и разнообразно програмно осигуряване за работа с него може да се намери в Интернет на следните адреси:

http://haskell.org/hugs/
http://haskell.org/ghc/
http://www.cs.york.ac.uk/fp/nhc98/

Грубо казано, програмите на езика Haskell са редици от символи, числа, списъци, коментари и др.

Един от основните елементи на всеки език за програмиране е *символът*. Символ е редица от букви, цифри и специални знаци с ограничена или неограничена дължина. В някои езици се различават главни и малки букви, в други - не. В езика Scheme разлика между главна и малка букви няма, но в Haskell има. Символите най-често се използват като идентификатори – за означаване на имена на величини, функции, формални параметри и др. Идентификаторите започват с латинска буква (малка или главна), която се следва от латински букви, цифри, _ и апостроф (mag’s е коректен идентификатор).

Правила за именуване:

- Имената на типовете са идентификатори, започващи с главна буква.
- Имената на функциите и величините са идентификатори, започващи с малка буква.
- В качество на първи символ на идентификатор може да се използват някои специални знаци (например ‘_’ е малка латинска буква).

Числата са редици от цифри, евентуално предшествани от ‘-‘ (за означаване на отрицателните числа). Различават се цели числа и реални числа.

Примери:
123, -35, -1.24e-12, 2345.0123

Във функционалните езици съществува базово понятие – *атом*. Символите и числата са атомите в Haskell.

Важно понятие е *списъкът*. В Haskell за означение на списък се използват скобите [и]. Разделител между елементите е ‘,’.

Примери: [1, 2, 3]; [‘a’, ‘b’, ‘c’, ‘d’]; [] са списъци.

Коментарите се означават по два начина:
а) започват с -- и продължават до края на реда
б) започват с {-, завършват с -} и може да заемат няколко реда.

Програмите на езика Haskell са редици от дефиниции (на величини и функции) и коментари. Наричат се *скриптове*. Съществуват два стила за писане на скриптове:

а) *общоприет (традиционен)*

Програмата се състои от програмен код, между линиите на който са записани коментари. Скриптовете, написани в този стил се съхраняват във файлове с разширение “.hs”.

Пример:

```
-- function square
square :: Int -> Int
square x = x * x
-- function add
add :: Int -> Int -> Int
add x y = x + y
```

б) *алтернативен*

Програмата се състои от коментари (без знаците за коментари) и програмен код, който е разположен в редовете, започващи с “>”. Скриптовете, написани в този стил се съхраняват във файлове с разширение “.lhs”.

Пример:

```
function square

> square :: Int -> Int
> square x = x * x

function add

> add :: Int -> Int -> Int
> add x y = x + y
```

2. Дефиниране на величини и функции

Програмата на езика Haskell се състои от известен брой дефиниции на функции и величини. Всяка дефиниция свързва име (идентификатор) със стойност от определен тип.

Дефиниране на величина

```
name :: type
name = expression
```

Тази дефиниция свързва идентификатора name със стойността на израза expression.

Пример:

```
size :: Int
size = (15-5)*8
```

името size се свързва със стойността 80 на израза от тип Int.

Забелязваме, че името size започва с малка буква, а името на типа Int – с главна буква.

Символът :: се чете "е от тип". Така size :: Int се чете "size е от тип Int".

След дефиницията на идентификатора size, той може да се използва.

Пример:

```
Hugs> size - 25
55
```

Дефиниране на функции

В Haskell съществуват два начина (образеца) за дефиниране на функции:

- къри образец (дефинираните по този образец функции се наричат кърингови);
- некъри образец (дефинираните по този образец функции се наричат некърингови).

Разликата между тях е при дефиниране на функции с повече от един аргумент.

Къри образец

***Най-простият вид на дефиниция на функция е*:**

```
name x1  x2  … xk = e
```

където name е идентификатор, започващ с малка буква и определящ името на дефинираната функция; x1, x2, …, xk са формалните параметри – идентификатори, започващи с малка буква, а е е израз, стойността на който определя резултата от изпълнението на функцията. Дефиницията на функцията се предшества от декларация на типа ѝ. Декларацията има следния формат:

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

където name е името на дефинираната функция, а t1, t2, …, tk и t са имената на типовете на формалните параметри и резултата съответно.

Примери:

1. Ще дефинираме функцията square: x --> x^2.

```
square :: Int -> Int
square x = x * x
```

Първата линия на Haskell дефиницията на функцията square декларира типа ѝ. Стрелката в декларацията означава, че square е функция. Типовете, ограждащи стрелката, определят, че square е функция, която има цял аргумент и връща цяла стойност като резултат.

След като е дефинирана, функцията square може да се използва.

```
Hugs> square 5
25
```

2. Ще дефинираме функциите:

```
add: x, y --> x+y
mult: x, y, z --> x*y*z

add :: Int -> Int -> Int
add x y = x + y
mult :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

След дефинициите, add и mult може да бъдат използвани.

```
Hugs> add 3 5
8
Hugs> mult 1 2 3
6
```

Общ вид на дефиниция на функция

***Синтаксис*:**

```
name x1  x2  … xk
     | g1 = e1
     | g2 = e2
          …
     | otherwise = e
```

където

- name е идентификатор, започващ с малка буква и определящ името на дефинираната функция;
- x1, x2, …, xk са идентификатори, започващи с малки букви. Наричат се формални параметри на функцията,
- g1, g2, … са булеви изрази. Наричат се охрани (или условия),

- e1, e2, … e са изрази, определящи резултата от обръщението към функцията. Дефиницията на функцията също се предшества от декларация на типа ѝ. Декларацията има следния формат:

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

където name е името на дефинираната функция, а t1, t2, …, tk и t са имената на типове на формалните параметри и резултата съответно.

Семантика:

Оценяват се условията (охраните) последователно, като се започва от първото условие. Изпълнява се изразът, чиято охрана има стойност True. Ако всички охрани имат стойност False, изпълнява се изразът, съответстващ на otherwise.

Клаузата otherwise не е задължителна и ако я има, задължително е последна.

Пример: Дефиниране на функция, която намира максималното на две цели числа.

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

Обръщението към функцията max се изпълнява по следния начин: ако първата охрана (x >= y) е True, изчислява се изразът след нея (в случая x). В противен случай (първата охрана е False), тогава се гледа следващата охрана. В случая охраната otherwise определя, че е в сила x < y и се изчислява изразът след нея (в случая y).

Пример: Дефиниране на функция, която намира максималното на три цели числа.

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
     | x >= y && x >= z  = x
     | y >= z            = y    -- x не е максималното
     | otherwise         = z
```

Пример за рекурсивно дефинирана функция:

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

Когато охраните са две, изразяващи условие и отрицанието му, вместо охрани може да се използва условният израз if-then-else.

Условен израз if-then-else

Синтаксис

if <условие> then <израз1> else <израз2>

където

- <условие> е булев израз,
- <израз1> и <израз2> са изрази от един и същ тип.

Семантика:

Връща стойността на <израз1>, ако стойността на <условие> е True и стойността <израз2> – в противен случай.

Пример:

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

Пример:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z = if x >= y && x >= z then x else
                if y >= z then y else z
```

някъри образец

***Най-простият вид на дефиниция на функция е*:**

```
name (x1, x2, …, xk) = e
```

където:

- name е идентификатор, започващ с малка буква и определящ името на дефинираната функция;
- x1, x2, …, xk са идентификатори, започващи с малка буква. Наричат се формални параметри,
- **e** е изразът, стойността на който определя резултата от изпълнението на функцията.

Дефиницията на функцията се предшества от декларация на типа ѝ. Декларацията има следния формат:

```
name :: (t1, t2, …, tk) -> t
```

където name е името на дефинираната функция, а t1, t2, …, tk и t са имена на типове на формалните параметри и резултата съответно.

Примери: Ще дефинираме функциите add и mult като некъринговите:

```
add :: (Int, Int) -> Int
add (x, y) = x + y
mult :: (Int, Int, Int) -> Int
mult (x, y, z) = x * y * z
```

Общ вид на дефиниция на некърингова функция

***Синтаксис*:**

```
name (x1, x2, …, xk)
     | g1 = e1
     | g2 = e2
          …
     | otherwise = e
```

където:

- name е идентификатор, започващ с малка буква и определящ името на дефинираната функция;
- x1, x2, …, xk са формалните параметри,
- g1, g2, … са предикати. Наричат се още охрани (или условия),
- e1, e2, … е са изрази, определящи резултата от обръщението към функцията в отделните случаи.

Дефиницията на функцията се предшества от декларация на типа ѝ. Декларацията има следния формат:

```
name :: (t1, t2, …, tk) -> t
```

където name е името на дефинираната функция, а t1, t2, …, tk и t са имената на типовете на формалните параметри и резултата съответно.

Пример: Дефиниране на некърингова функция, която намира максималното на три цели числа.

```
max3 :: (Int, Int, Int) -> Int
max3 (x, y, z)
     | x >= y && x >= z  = x
     | y >= z            = y
     | otherwise         = z
```

3. Обръщение към функция

Къри образец

При този образец обръщението към функции не се оформя като списък.

Синтаксис:

```
name f1 f2 … fk
```

където f1, f2, …, fk са фактическите параметри (изрази от тип, съвместим с типа на формалните параметри).

Например, ако дефиницията на функция за събиране на две числа има вида:

```
add  :: Int -> Int -> Int
add x y = x + y
```

обръщението към нея за събиране на 5 и 7 има вида:

```
add 5 7
```

Типът на кьоринговите функциите може да се представи във вида: t1 (t2 … (tk t) ...). За функциите, които са кьорингови, е възможно частично прилагане на аргументите им. В резултат се получава функция. Обръщенията:

```
(name f1)
(name f1 f2)
      …
(name f1 f2 … fk-1)
```

са функции.

Пример:
(add 12) е функция с един аргумент

```
Hugs> (add 12) 13
25
```

Пример: Ако

```
mult :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

(mult 2); (mult 2 3); ((mult 2) 3) са функции. Допустими са обръщенията:

```
Hugs> (mult 2) 3 5
30
Hugs> (mult 2 3) 5
30
Hugs> ((mult 2) 3) 5
30
```

т.е. `Int -> Int -> Int -> Int` е еквивалентно на `Int -> (Int -> (Int -> Int))`.

Пример: Функцията inc добавя 1 към аргумента си.

```
inc  :: Int -> Int
inc  = add 1
```

В този случай извикването на функцията inc с един параметър се свежда до обръщение към функцията add, единият параметър на която е 1.

Некъри образец
Синтаксис:

```
name (f1, f2, …, fk)
```

където f1, f2, …, fk са фактическите параметри (изрази от тип, съвместим с типа на формалните параметри).

Пример:

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

е некърингова функция.

```
Hugs> add(2, 6)
8
```

За функциите, които са не са кърингови, не е възможно частично прилагане на аргументите им. Не е допустимо обръщение от вида:

```
Hugs> (add 2) 6
```

4. Функции и процесите, които генерират

Линейна рекурсия и итерация

Задача 1. Да се дефинира функция, която намира n!, където n е дадено естествено число.

Първи начин:

$$n! = \begin{cases} 1, & n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

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

или

```
fact :: Int -> Int
fact n = if n == 1 then 1 else n*fact(n-1)
```

```
fact 4 ->
4*fact 3 ->
4*3*fact 2 ->
4*3*2*fact 1 ->
4*3*2*1 ->
4*3*2 ->
4*6 ->
24
```

Тези дефиниции генерират линейно рекурсивен процес.

Втори начин:

```
fact :: Int -> Int
fact n = iter n 1 1

iter :: Int -> Int -> Int -> Int
iter n i p = if i > n then p else iter n (i+1) (p*i)
```

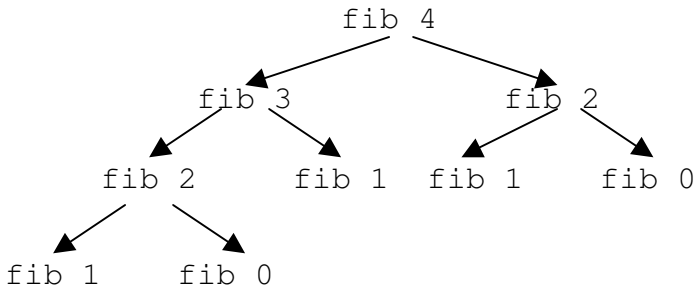
```
fact 4 ->
iter 4 1 1 ->
iter 4 2 1 ->
iter 4 3 2 ->
iter 4 4 6 ->
iter 4 5 24 ->
24
```

Тази дефиниция генерира линейно итеративен процес.

Забележка: Интерпретаторите на Haskell не са опашково рекурсивни (за разлика от тези на Scheme).

Дървовидна рекурсия
Задача 2. Да се дефинира функция, която намира n-тото число на Фибоначи.

```
fib :: Int -> Int
fib n
  | n == 0          = 0
  | n == 1          = 1
  | n > 1           = fib(n-1) + fib(n-2)
```



Дефинираната функция генерира двоично-дървовиден рекурсивен процес.

5. Основни типове данни в Haskell

5.1. Тип Bool

Булевите константи (стойности, литерали) са True и False.
Основни операции за работа с тях са:

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

Таблицата с вярностните стойности на тези функции е известна. Булевите данни могат да са аргументи на функции, както и да са резултати от изпълнението на функции. Като се използват булевите литерали и основните операции може да се дефинират функции.

Пример: Дефиниране на XOR (изключващо или)

Изключващото или е функция, която връща True, когато единият, но не и двата нейни аргументи, има стойност True. Може да се дефинира по следния начин:

```
ExOr :: Bool -> Bool -> Bool
ExOr x y = (x || y) && not (x && y)
```

Пример:

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

или

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

Забелязваме, че последните две дефиниции се състоят от по две равенства и някой от аргументите им са булеви литерали.

Данни от тип Bool могат да се сравняват чрез операторите: >, >=, <, <=, ==, /= за по-голямо, по-голямо или равно, по-малко, по-малко или равно, равно и различно съответно.

5.2. Типове Int и Integer

Целите константи (литерали) се записват като в другите езици за програмиране. За представяне на целите числа от тип Int се използват 32 бита. Множеството от стойностите на типа Int е $[-2^{31}, 2^{31} - 1]$.

За работа с цели числа с неограничена точност се използва типът Integer.

Примери: 1; -234; 0; -123376;
21234123999999999;

За работа с цели числа най-често ще използваме типа Int, тъй като голяма част от стандартните функции са дефинирани за типа Int.

Основни операции за работа с цели числа:

Аритметични оператори

+, -, *	събиране, изваждане и умножение на цели числа;
^	степенуване (2 ^ 35)
div	частно от целочислено деление (div 12 5 или 12 `div` 5)
mod	остатък от целочислено деление (mod 12 5 или 12 `mod` 5)
abs	абсолютна стойност (abs 12; abs (-12))
negate	променя знака на цяло число (negate 12; negate (-12))

gcd — намира най-големия общ делител на две неотрицателни цели числа, поне едно от които е различно от 0.

Забележки:

1. Натуралните числа са неотрицателните цели числа: 0, 1, 2, ...
2. Чрез ограждане на името на всяка двуаргументна функция в обратни апострофи е възможен инфиксен запис на обръщенията към тези функции.
3. Дефиниран е унарният оператор -.
4. Обръщението

```
negate –15
```

предизвиква грешка, тъй като се интерпретира като “negate унарен_минус 15”. Правилният запис е:

```
negate (–15)
```

Оператори за сравнения:

>, >=, = =, /=, <, <= – сравняват цели числа за по-голямо, по-голямо или равно, равно, различно, по-малко и по-малко или равно съответно.

Пример: Проверка дали 3 числа са равни.

```
threeEqual :: Int –> Int –> Int –> Bool
threeEqual x y z = (x == y) && (y == z)
```

Забележка: Приоритет на някои оператори:

^, ** – по-висок
*, /, div, mod
+, -
<, <=, >, >=, ==, /=
&&
|| – по-нисък

5.3. Тип Char

Константите (литералите) от тип Char се означават като знакът се огражда в апострофи.

Примери:

‘a’, ‘b’, …, ‘z’

tab	‘\t’	– знак за табулация
newline	‘\n’	– знак за нов ред
backslash	‘\\’	– знак за обратна наклонена черта
single quote	‘\’’	– знак апостроф
double quote	‘\”’	– знак кавичка

Кодирането ASCII е разширено до стандарта Unicode.

Основни операции за работа със знакови литерали са:

```
ord :: Char –> Int  –– намира кода на символен литерал
```

```
chr :: Int -> Char  -- намира символен литерал със
                    -- зададен код
```
(съдържат се в библиотеката Char.hs). Знакът с код n, където n е цяла константа, може да се намери и чрез: '\n'. Например, '\65' е 'A'.

Дефинирането на функциите ord и chr може да стане по следния начин:
```
ord :: Char  -> Int
ord = fromEnum

chr :: Int -> Char
chr = toEnum
```
където функциите fromEnum и toEnum преобразуват в случая от тип Char в тип Int и обратно. Последните функции принадлежат на сигнатурата на вграден в езика клас.

Данните от тип Char могат да се сравняват чрез операторите: >, >=, <, <=, == и /=.

Примери:
1) проверка дали знак е цифра
```
isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')
```

2) преобразуване на малки в главни букви
```
toUpper :: Char -> Char
toUpper ch = chr (ord ch + ord 'A' - ord 'a')
```

5.4. Реални числа (типове Float, Double, Rational)

Реалните числа (реалните литерали) се записват:
- като десетични дроби

Пример: 0.234102; -25.51; 6543.908; 3.0.
- в научен (експоненциален) формат

Пример: 1.2e-3; 123.76e+2.

Типът Float се използва за работа с реални числа с единична точност (броят на значещите цифри е 7, а диапазонът е от 10^{-38} до 10^{38}).

Типът Double се използва за работа с реални числа с двойна точност (броят на значещите цифри е 15, а диапазонът е от 10^{-308} до 10^{308}).

Типът Rational представя рационални числа с неограничена точност. Ще използваме само типа Float, тъй като повечето от функциите, които ще разглеждаме са дефинирани за него.

Основни оператори, величини и функции за работа с данни от тип Float:

+, -, *	Float -> Float -> Float	събиране, изваждане и умножение
/	Float -> Float -> Float	реално деление

^	Float -> Int -> Float	(x^n за неотрицателно цяло число n)
**	Float -> Float -> Float	x ** y = x^y
==, /=		
>, >=		
<, <=	Float -> Float -> Bool	сравнения
abs	Float -> Float	абсолютна стойност
asin, acos	Float -> Float	обратните функции на sin, cos, tan
atan	Float -> Float	
ceiling	Float -> Int	закръгляват дроб в цяло чрез
floor	Float -> Int	закръгляване нагоре, надолу или
round		по правилата за закръгляване
		(до най-близкото цяло число)
sin, cos,	Float -> Float	sin, cos, tan (градусите се
tan		преобразуват в радиани)
exp	Float -> Float	exp x = e^x
fromInt	Int -> Float	преобразува цяло в реално
log	Float -> Float	ln
logBase	Float -> Float -> Float	логаритмуване при произволна
		основа (основа, число, резултат)
negate	Float -> Float	променя знака
pi	Float	константата pi
signum	Float -> Float	1.0, 0.0 или –1.0
sqrt	Float -> Float	положителния корен квадратен

Езикът е типов. Има механизъм за извод на тип, който е вграден в езика. В редица случаи се налага явно да се укаже типът. За целта в Haskell се използва специалният символ :: (чете се “има тип” или “е от тип”).

Пример:

15 :: Integer

(Константата 15 е от тип Integer).

Haskell поддържа нещо извънредно, т.нар. полиморфни типове, или шаблони типове. Например, [a] означава "Списък от атоми от произволен вид”, като типът на атомите трябва да бъде един и същ в целия списък. т.е. списъците [1, 2, 3] и ['a', 'b', 'c'] ще имат тип [a]. В този случай, в записа [a] символът a има смисъла на типова променлива.

6. Недефинирани стойности на функции

Нека да разгледаме следната дефиниция на функцията fact:

```
fact :: Integer -> Integer
fact n
  |  n == 1          = 1
```

```
  |  n > 1          = fact (n-1) * n
```
и да оценим fact (–5). В резултат се получава съобщение за грешка. Причината е, че fact не е дефинирана за отрицателни цели числа. Възникналият проблем може да се разреши по следните начини:

а) функцията се додефинира за не положителните цели числа

```
fact :: Integer -> Integer
fact n
  |  n == 1         = 1
  |  n > 1          = fact (n-1) * n
  | otherwise       = 0
```

б) чрез функцията error

Синтаксис:

```
error :: String -> a
error "символен низ"
```

Семантика:

Извежда символния низ на екрана и прекратява оценяването.

```
fact :: Integer -> Integer
fact n
  | n == 0     = 1
  | n > 0      = fact(n-1) * n
  | otherwise  = error "not defined"
```

7. Съпоставяне по образец

Формалните параметри на функциите са произволни идентификатори. Например в дефиницията f x y = x * y, x и y са формалните параметри на f. При обръщение към f се използват фактически параметри. Например в обръщението

```
f 19 (f 1 2)
```

19 и f 1 2 са фактическите параметри. Последните са изрази от типа на съответните формални параметри. В повечето езици за програмиране формалните параметри са имена (идентификатори). В езика Haskell формалните параметри могат да са образци (изрази).

Образецът може да бъде:

- литерал;
- идентификатор;
- специален знак за безусловно съпоставяне '_';
- наредено множество (вектор);
- конструктор.

При обръщение към функция се осъществява съпоставяне на формалните параметри (образци) с фактическите параметри.

Пример: Възможно е да се дефинира функция от вида:

```
f :: [Int] -> Int
f [5, x, y] = x*y
```

Тази функция се прилага само към фактически параметър, който е списък с три елемента, като първият трябва да е 5, а другите два – произволни цели числа. Тя не се прилага към по-къси или към по-дълги списъци, както и към спъсъци с първи елемент, различен от 5.

В Haskell е допустимо да се дефинират функции с различни образци в качество на формални параметри. За целта най-често се използва дефиниране на функциите като редица от равенства.

Пример: Функцията sum се дефинира чрез следните равенства:

```
sum :: [Int] -> Int
sum []          = 0
sum [x]         = x
sum [x,y]       = x+y
sum [x,y,z]     = x+y+z
```

Тази функция може да се прилага към списъци с 0, 1, 2 и 3 цели числа. По-късно ще дефинираме sum за списъци с произволна дължина. Елементите [], [x], [x,y], [x,y,z] са *образците* на дефиницията. При обръщение към sum интерпретаторът съпоставя фактическите с формалните параметри и по този начин избира кое равенство да изпълни. Например обръщението sum [2,3,4] съпоставя фактическия параметър [2,3,4] с образеца [x, y, z], свързва x с 2, y с 3, z с 4 и изпълнява 2*3*4.

Пример: Програмата

```
func :: Int -> Int -> Int
func x y
  | x == 1     = y
  |otherwise   = x
```

може да се запише и с използване на равенства по следния начин:

```
func 1 y  = y
func x y  = x
```

Чрез тези равенства са описани двата случая от първата дефиниция. Литералът 1 и променливите x и y са *образците* на дефиницията. Равенствата се прилагат последователно, като до всяко следващо равенство се достига и то евентуално се прилага само ако всички предишни равенства не са дали резултат при съпоставянето на образците с фактическите параметри.

Забелязваме, че във второто равенство от дефиницията на func променливата y не се използва. Тази променлива може да бъде заменена от образеца "_", известен като wildcard (специален символ за безусловно съпоставяне). Тогава програмата има вида:

```
func 1 y  = y
func x _  = x
```

Обръщението func 1 5 се оценява като се съпоставя с лявата страна на първото равенство. В резултат се получава 5. Обръщението func 3 11 не може да се съпостави с лявата страна на първото равенство. Съпоставя се с лявата страна на второто равенство. В резултат x се свързва с 3 и 3 е резултатът от обръщението.

Правила за съпоставяне на образец и фактически параметър:

- Образец от вид литерал се съпоставя успешно с фактически параметър, който е литерал и равен на образеца.
- Образец от вид идентификатор се съпоставя успешно с фактически параметър с произволна стойност.
- Образецът '_' се съпоставя с произволен фактически параметър.
- За да бъде съпоставим векторът-образец (a1, a2, …, an) с фактическия параметър, последният трябва да има вида (b1, b2, …, bn), като всяко bi трябва да бъде съпоставимо със съответното ai.

Както в дефиницията на функция не е възможно формалните параметри да се повтарят, така и в образците не се допуска повторения на имена. Пример на грешна дефиниция на функция, проверяваща дали елемент принадлежи на списък, е следният:

```
member x []     = False
member x (x:xs)= True  -- error x се използва два пъти
member x (y:ys)= member x ys
```

Второто равенство е грешно, тъй като идентификаторът x се среща два пъти.

Забележка: Допуска се повтаряне на символа за безусловно съпоставяне "_".

8. Задачи

Задача 1. Да се дефинира функция, която намира:
a) max(min(x, y), min(p, q))

Решение:

```
max1 :: Int -> Int -> Int
max1 x y = if x >= y then x else y

min1 :: Int -> Int -> Int
min1 x y = if x >= y then y else x

func :: Int -> Int -> Int -> Int -> Int
func x y p q = max1 min1 x y min1 p q
```

б) $s = \sqrt{p(p-a)(p-b)(p-c)}$, където $p = \frac{a+b+c}{2}$, a a, b и c са страни на триъгълник.

Решение:

```
-- perimeter
p :: Float -> Float -> Float -> Float
p a b c = (a+b+c)/2

-- area
area:: Float -> Float -> Float -> Float
area a b c = sqrt(p a b c * (p a b c - a) *
                  (p a b c - b) * (p a b c - c))
```

или

```
-- area
s :: Float -> Float -> Float -> Float
s a b c = sqrt (p1*(p1-a)*(p1-b)*(p1-c))
  where p1 = p a b c            -- локална дефиниция
```

Задача 2. Да се дефинира функция, която реализира бързо степенуване

Решение:

```
-- square
sq :: Int -> Int
sq x = x*x

-- fast exp
fast :: Int -> Int -> Int
fast b n
  | n == 0 = 1
  | odd n  = fast b (n-1) * b
  | even n = sq(fast b (div n 2))
```

Задача 3. Да се дефинира функция, която проверява дали в записа на цяло число се съдържа цифрата 2.

Решение:

```
digit2 :: Int -> Bool
digit2 n
  | n < 0                = digit2 (-n)
  | n == 0               = False
  | (n `mod` 10) == 2    = True
  | otherwise            = digit2 (n `div` 10)
```

Задача 4. Да се дефинира функция gcd1, която намира най-големия общ делител на две положителни цели числа.

Решение:

```
gcd1 :: Int -> Int -> Int
gcd1 n m
   | n == m  = n
   | n > m   = gcd1 (n-m) m
   | n < m   = gcd1 n (m-n)
```

Задача 5. Да се дефинират следните функции:

- isUpper, проверяваща дали латинска буква е главна;
- isLower, проверяваща дали латинска буква е малка;
- isAlpha, проверяваща дали знак е латинска буква.

Решение:

```
isUpper, isLower, isAlpha :: Char -> Bool
isUpper c = c >= 'A' && c <= 'Z'
isLower c = c >= 'a' && c <= 'z'
isAlpha c = isUpper c || isLower c
```

Задачи за самостоятелна работа

Задача 1. Да се дефинира функция, която намира стойността на следната неелементарна функция:

$$f(x) = \begin{cases} x^2 - x + 4, & x > 2 \\ x + 7, & x \in [1, 2] \\ 0, & x < 1 \end{cases}$$

Задача 2. Да се дефинира функция, която връща 0, ако a, b и c не са страни на триъгълник, 3 ако a, b и c са страни на равностранен триъгълник, 2 ако a, b и c са страни на равнобедрен триъгълник и 1 ако a, b и c са страни на произволен триъгълник.

Задача 3. Да се дефинира функция, която установява дали:

а) x принадлежи на интервала [1, 17];

б) x е извън интервала [1, 17];

в) числата a, b и c са различни;

г) поне две от числата a, b и c са равни.

Задача 4. Да се дефинира функция, която генерира:

а) линейно рекурсивен;

б) линейно итеративен

процес за намиране на биномния коефициент C_n^m $(0 \le m \le n)$ по формулата:

$$C_n^m = \frac{n(n - 1)...(n - m + 1)}{m!}.$$

Задача 5. Да се дефинира функция, която генерира:

а) линейно рекурсивен;

б) линейно итеративен

процес за намиране на максималния елемент от серията числа: $4i^2 - 13in + 17n^2$, i = 1, 2, …, n.

Задача 6. Да се дефинира функция, която намира произведението на числата от 1 до n със стъпка 3. Какъв процес генерира тя?

Задача 7. Да се дефинира функция, която намира обърнатото число на дадено естествено число, т.е. числото съставено от същите цифри, но записани в обратен ред. Какъв процес генерира тя?

Задача 8. Да се дефинира функция, която установява дали цифрите на дадено положително цяло число са наредени в намаляващ ред.

Задача 9. Да се дефинира функция, която намира стойността на сумата:

а) $1! + 2! + ... + n!$;

б) $1^n + 2^n + ... + n^n$;

в) $1^1 + 2^2 + ... + n^n$.

Задача 10. Да се дефинира функция, която генерира дървовидно рекурсивен процес за намиране на биномния коефициент $C_n^m (0 \le m \le n)$, като се използват следните зависимости:

$C_n^0 = C_n^n = 1$

$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$, при $0 < m < n$.

Тема 2
Наредени множества (вектори). Списъци. Работа със списъци

1. Наредени множества

Наредените множества или **вектори** са редици от елементи, които могат да са от различни типове. Броят на елементите, както и типовете на елементите трябва да бъдат определени предварително.

В другите езици за програмиране наредените множества се наричат записи или структури.

Типът *вектор с n на брой елемента* се дефинира по следния начин:

$(t_1, t_2, \ldots, t_n)$

където $t_1, t_2, \ldots, t_n$ са имена на типовете на компонентите.

Константите на типа са вектори от вида:

$(v_1, v_2, \ldots, v_n)$

за които е в сила:

$v_1::t_1, v_2::t_2, \ldots, v_n::t_n$

Примери:

`(3.5, "book")` е наредено множество от тип (Float, [Char])

`('x', True, 10)` е наредено множество от тип (Char, Bool, Int)

`([10, 20], sqrt)` е наредено множество от тип ([Int], Float -> Float)

`(12, (25, 35, 45))` е наредено множество от тип (Int, (Int, Int, Int))

Пример: Дефиниране на тип наредено множество Student и константи от тип Student:

```
type Student = (String, Int)
s1 :: Student
s2 :: Student
s1 = ("Ivan Petrov", 44890)
s2 = ("Petar Dimitrov", 33456)
```

В наредените множества е важен редът на елементите. Типът на наредено множество се определя от типа на всеки негов елемент. Така наредените множества (1, (2, 3)) и ((1, 2), 3) са различни и типовете им са съответно (Int, (Int, Int)) и ((Int, Int), Int).

За наредени множества от два елемента често се използва терминът двойка, за наредени множества от три елемента – тройка и т.н. Съществува наредено множество с 0 елементи: величината () има тип ().

Файлът Prelude.hs (прелюдие) съдържа функции за работа с двойки. Това са селекторите fst и snd, дефинирани по следния начин:

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

Освен селекторите, за намиране на компонентите на наредени множества, се използва механизмът на съпоставянето по образец.

Пример: В mult4 x, y, z и t се съпоставят съответно с първия, втория, третия и четвъртия елемент на наредено множество от четири елемента.

```
mult4 :: (Int, Int, Int, Int) -> Int
mult4 (x, y, z, t) = x*y*z*t
```

Задача 1. Да се дефинира функция, която подрежда във възходящ ред елементите на двойка от цели числа.

Решение:

```
sort2 :: (Int, Int) -> (Int, Int)
sort2 (x, y) = if x <= y then (x, y) else (y, x)
```

Задача 2. Да се дефинира функция, която намира минималното и максималното на две цели числа.

Решение:

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

Задача 3. Да се дефинира функция, която намира n-тата двойка на Фибоначи ($n \geq 0$ е дадено цяло число).

Решение:

```
fibStep :: (Int, Int) -> (Int, Int)
fibStep (u, v) = (v, u+v)

fibPair :: Int -> (Int, Int)
fibPair n
     | n == 0 = (0, 1)
     | otherwise = fibStep (fibPair (n-1))
```

В предложените примери компонентите на двойките са от един и същ тип.

2. Списъци

Списъците са редици от произволен брой елементи от един и същ тип. Елементите може да са изрази. За всеки тип a, [a] е типът списък с елементи от тип a.

[] е празният списък.

Ако e1, e2, …, en са елементи от един и същ тип,

```
[e1, e2, …, en]
```

е списък с елементи от този тип.

Типът String е еквивалентен на [Char], т.е.

```
type String = [Char]
```

Примери:

```
[1, 2, 3, 4, 5] :: [Int]
[1+2, 12-4, 8*2] :: [Int]
[True] :: [Bool]
[True, True, False] :: [Bool]
[3 < 1, 'a' >= 'A', 4 == 6] :: [Bool]
['x', 'y', 'z'] :: String
"xyz" :: String
[cos, tan, sin] :: [Float -> Float]
[[], [1, 2], [1, 2, 3]] :: [[Int]]
```

Списъците [1, 2, 1, 2, 2], [2, 1, 1, 2, 2] и [1, 2, 2, 1] са съставени от числата 1 и 2, но са различни.

Във файла Prelude.hs са дефинирани някои стандартни функции за работа със списъци. Такива са:

Функция	**Резултат**
head	първия елемент на непразен списък
tail	списък от елементите на непразен списък, без първия му елемент
length	брой елементи на списък
reverse	списък, записан в обратен ред

Примери:

```
Hugs> head [1,2,3,4]
1
Hugs> tail [1,2,3,4]
[2,3,4]
Hugs> length [[1,2],[3,4]]
2
Hugs> reverse [1,2,3,4]
```

```
[4,3,2,1]
Hugs> reverse "abcd"
"dcba"
```

За конкатенация на два списъка с елементи от един и същ тип се използва дясноасоциативният оператор ++.

Пример:

```
Hugs> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

Задача 4. Да се дефинира функция toEnd, която включва елемент от тип a в края на списък от тип [a] (a е типова променлива).

Решение:

```
toEnd :: a -> [a] -> [a]
toEnd x l = l ++ [x]
```

Най-често *списъците* се записват във вида $[e_1, ..., e_k]$, където k >= 0, а $e_1, ..., e_k$ са изрази от един и същ тип. : и [] се явяват конструктори на списък. Ако k е 0, списъкът е празен. Ако k > 0, в сила е

$$[e_1, \ldots, e_k] = e_1 : (e_2 : (\ldots (e_k : [])))$$

т.е. ":" включва елемент в списък като първи. Конструкторът ":" е дясноасоциативен оператор, т.е. $e_1 : (e_2 : (... (e_k : [])))$ е еквивалентен на $e_1 : e_2 : ... e_k : []$. Тъй като изпълнява функцията на конструктор на списък, понякога ":" се нарича cons.

Пример:

```
Hugs> 1 : []
[1]
Hugs> 2 : [1]
[2,1]
Hugs> 1 : (2 : (3 : []))
[1,2,3]
Hugs> 1 : 2 : 3 :[]      -- : е дясноасоциативен
[1,2,3]
```

Съществуват още начини за записване на списъци от числа, знакове и други типове:

⇒ [n .. m], където n ≤ m, е списъкът [n, n+1, …, m], а ако n > m, списъкът е празен.

Примери:

```
[3 .. 9] = [3, 4, 5, 6, 7, 8, 9]
['d' .. 'n'] = "defghijklmn"
[False .. True] = [False, True]
```

Ако типът на n и m е реален, стъпката отново е 1, а последният елемент на списъка е p, така че |p-m| ≤ 0.5. В случая |p-m| = 0.5, p е по-голямото число, за което е в сила релацията.

Примери:

```
[4.2 .. 8.0] = [4.2, 5.2, 6.2, 7.2, 8.2]
[4.3 .. 8.8] = [4.3, 5.3, 6.3, 7.3, 8.3, 9.3]
[-3.2 .. 1.3] = [-3.2, -2.2, -1.2, -0.2, 0.8, 1.8]
```

⇒ [a, b .. c] е списък, първите два елемента на който са a и b, а останалите се получават със стъпка b-a (ord b – ord a).

Примери:

```
[9, 7 .. 2] = [9, 7, 5, 3]
[0.1, 0.3 .. 1.0] = [0.1, 0.3, 0.5, 0.7, 0.9, 1.1]
['b', 'd' .. 'n'] = "bdfhjln"
```

⇒ [expr | q_1, …, q_n], където expr е израз, а q_i е или генератор от вида p <- listExpr (p е образец, а listExpr е израз от тип списък), или булев израз. В q_i ($2 \le i \le n$) могат да участват променливите, генерирани в q_1, q_2, ..., q_{i-1}.

Генераторът `p <- listExpr` означава, че p получава последователно всички елементи на списъка listExpr.

Примери:

```
1) Hugs> [2*n | n <- [1, 2, 3, 4]]
```

е списък с елементи от вида 2*n, за всяко n принадлежащо на списъка [1, 2, 3, 4], т.е. `[2,4,6,8]`

```
2) Hugs> [2*n-1 | n <- [1, 2, 3, 4]]
```

е списък от числата 2n-1, за всяко n, принадлежащо на списъка [1, 2, 3, 4], т.е.

```
[1,3,5,7]
```

```
3) Hugs> [even n | n <- [2, 4, 6, 8]]
```

Прилага се функцията even над всеки елемент на списъка [2, 4, 6, 8]. Получава се

```
[True,True,True,True]
```

```
4) Hugs> [odd n | n <- [2, 4, 7, 8]]
```

Прилага се функцията odd към всеки елемент на списъка [2, 4, 7, 8]. Получава се

```
[False,False,True,False]
```

```
5) Hugs> [2*n | n <- [2, 4, 5, 7, 8], even n, n > 3]
```

Умножава с 2 елементите на списъка [4, 8], получен от списъка [2, 4, 5, 7, 8], след филтриране с предикатите even и n > 3, т.е.

```
[8,16]
```

```
6) Hugs> [2*n | n <- [2, 4, 5, 7, 8], even n, n > 1]
```

Умножава с 2 елементите на списъка [2, 4, 8], получен от списъка [2, 4, 5, 7, 8], след филтриране с предикатите even и n > 1, т.е.

```
[4,8,16]
```

```
7) Hugs> [2*n | n <- [2, 4, 5, 7, 8], even n, n > 8]
[]
```

Означението на списък чрез генератори и предикати е доста разнообразно. След вертикалната черта може да има генератори на няколко променливи.

Примери:

а) [(x,y)| x <- [1 .. 5], y <- [1 .. 3]] е списъкът от двойки

```
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3),
 (4,1),(4,2),(4,3),(5,1),(5,2),(5,3)]
```

б) [(x,y)| x <- [1,2,3], y <- ['a', 'b', 'c', 'd']] е списъкът от двойки

```
[(1,'a'),(1,'b'),(1,'c'),(1,'d'),(2,'a'),(2,'b'),(2,'c'),
 (2,'d'),(3,'a'),(3,'b'),(3,'c'),(3,'d')]
```

Задача 5. Да се дефинира функция, която от списък от двойки от цели числа намира списък от цели числа, равни на сумите на елементите на двойките. Например, ако l = [(1, 5), (3, 4), (8, 9)], да се намери списъкът: [6, 7, 17].

Решение:

```
sumPairs :: [(Int, Int)] -> [Int]
sumPairs pL = [x+y | (x, y) <- pL]
```

Задача 6. Да се дефинира функция, която от списък от двойки от цели числа намира списък от цели числа, равни на сумите на елементите на онези двойките, които са монотонно растящи. Например, ако l = [(1, 5), (3, 4), (18, 9), (4, 4)], да се намери списъкът: [6, 7, 8].

Решение:

```
sumMonPairs :: [(Int, Int)] -> [Int]
sumMonPairs pL = [x+y | (x, y) <- pL, x <= y]
```

Задача 7. Да се дефинира функция, която намира списък, съдържащ всички цифри на даден символен низ.

Решение:

```
dig :: Char -> Bool
dig ch = ch >= '0' && ch <= '9'

digits :: String -> String
digits str = [ch | ch <- str, dig ch]
```

или

```
digits str = [ch | ch <- str, ch >= '0', ch <= '9']
```

Задача 8. Да се дефинира функция, която проверява дали всички елементи на списък от цели числа са:

а) четни числа;

б) нечетни числа.

Решение:

а) `allEven x = (x == [a | a <- x, even a])`

б) `allOdd x = (x == [a | a <- x, odd a])`

3. Образци на списъци

Всеки списък е или празен (равен на/съпоставим с []), или непразен. Непразният списък има глава и опашка, т.е. има вида x : xs, където x е първият елемент на списъка, а xs е списъкът без първия му елемент. Всеки списък може да бъде конструиран от празния чрез многократно прилагане на оператора ':'.

Образецът – конструктор на списък е или [], или има вида (x : xs), където x и xs са също образци. Правилата за съпоставяне с такъв образец са следните:

- списък е съпоставим с образеца [] точно когато е празен;
- списък е съпоставим с образец от вида (y : ys), когато не е празен, главата му се съпоставя с образеца y и опашката му се съпоставя с образеца ys.

В частност всеки непразен списък е съпоставим с образец от вида (y : ys), където y и ys са идентификатори.

Забележка: Образец, който включва конструктора ':', се загражда в кръгли скоби. Причината е, че прилагането на функция има по-висок приоритет от останалите операции.

4. Работа със списъци

Голяма част от функциите в Haskell са полиморфни (генерични). Например функцията length намира броя на елементите на списък, независимо какъв е типът на елементите на списъка, т.е.

```
length :: [Int] -> Int
length :: [Char] -> Int
length :: [Bool] -> Int
```

и т.н.

В общия случай

```
length :: [a] -> Int
```

където a е променлива, означаваща произволен тип. Типовете `[Char] -> Int`, `[Int] -> Int`, `...` са екземпляри на типа `[a] -> Int`.

Още функции и оператори за работа със списъци

Име	Тип	Семантика

!!
е ляво-асоциативен оператор
	[a] -> Int -> a	Намира указан чрез пореден номер елемент на списък. Броенето започва от 0.
		Пример:
		`Hugs> [1,2,3,4,5]!!3`
		`4`

concat	[[a]] -> [a]	Конкатенира списък от списъци в списък от елементите им.
		Пример:
		`Hugs> concat [[1,2,3], [4,5,6], [9]]`
		`[1,2,3,4,5,6,9]`

last	[a] -> a	Намира последния елемент на непразен списък.
		Пример:
		`Hugs> last [1,2,3,4,5]`
		`5`

init	[a] -> [a]	Намира списъка без последния му елемент. Даденият списък не е празен.
		Пример:
		`Hugs> init [1,2,3,4,5]`
		`[1,2,3,4]`

take	Int -> [a] -> [a]	Взема указан брой елементи от началото на списък и ги включва в нов списък.
		Примери:
		`Hugs> take 4 [1,2,3,4,5]`
		`[1,2,3,4]`
		`Hugs> take 0 [1,2,3,4,5]`
		`[]`

replicate	Int -> a -> [a]	Създава списък от указан брой копия на даден елемент в списък.
		Пример:
		`Hugs> replicate 3 4`
		`[4,4,4]`
		`Hugs> replicate 2 'a'`
		`"aa"`

drop	Int -> [a] -> [a]	Изтриване на указан брой елементи от началото на списък.
		Примери:
		`Hugs> drop 3 [1,2,3,4,5,6,7]`
		`[4,5,6,7]`

		`Hugs> drop 3 [1,2]`
		`[]`
		`Hugs> drop 2 []`
		`[]`

splitAt	Int -> [a] -> ([a], [a])	Разцепва даден списък в указана позиция.
		Пример:
		`Hugs> splitAt 4 [1,2,3,4,5,6,7,8,9]`
		`([1,2,3,4],[5,6,7,8,9])`
		`Hugs> splitAt 2 []`
		`([],[])`

zip	[a] -> [b] -> [(a, b)]	Преобразува два списъка в списък от двойки:
		Примери:
		`Hugs> zip [1,2,3] [4,5,6]`
		`[(1,4),(2,5),(3,6)]`
		`Hugs> zip [1,2] [4,5,6]`
		`[(1,4),(2,5)]`
		`Hugs> zip [] [1,2,3]`
		`[]`

unzip	[(a, b)] -> ([a], [b])	Преобразува списък от двойки в два списъка.
		Пример:
		`Hugs> unzip [(1, 3), (4, 2), (1, 9)]`
		`([1,4,1],[3,2,9])`

and	[Bool] -> Bool	Намира конюнкцията на елементите на списък от булеви константи.
		Пример:
		`Hugs> and [True, False]`
		`False`
		`Hugs> and [True, True]`
		`True`

or	[Bool] -> Bool	Намира дизюнкцията на елементите на списък от булеви константи.
		Пример:
		`Hugs> or [False, True]`
		`True`
		`Hugs> or [False, False, False]`
		`False`

sum	[Int] -> Int
	[Float] -> Float
		Намира сумата от елементите на списък от числа.
		Пример:
		`Hugs> sum [1,2,3]`
		`6`
		`Hugs> sum [1, 3..10]`
		`25`

product	[Int] -> Int
	[Float] ->
	Float
		Намира произведението на елементите на списък от числа.
		Пример:
		`Hugs> product [1,2,3,4]`
		`24`
		`Hugs> product [1, 3..8]`
		`105`

Дефиниции на някои функции за работа със списъци

а) *с използване на образци*

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs

null :: [a] -> Bool
null []   = True
null(_:_) = False
```

б) *с използване на примитивна рекурсия*

В дефиницията, реализираща този вид рекурсия, се описват следните случаи:

- базов случай;
- общ случай, при който се посочва връзката между стойностите на функцията за дадена стойност на аргумента и стойността й за по-проста стойност на аргумента.

Пример: Намиране на сумата на елементите на списък от числа

```
sum :: [Int] -> Int
sum []    = 0
sum (x:xs)= x + sum xs
```

Оценката на обръщението sum [1, 2, 3, 4] има вида:

```
sum [1,2,3,4] ->
1 + sum [2, 3, 4]  ->
1 + (2 + sum [3, 4]) ->
1 + (2 + (3 + sum [4])) ->
1 + (2 + (3 + (4 + sum []))) ->
1 + (2 + (3 + (4 + 0))) ->
10
```

т.е. дефиницията реализира линейно рекурсивен процес.

Пример: Намиране на конкатенацията на списъците, елементи на даден списък

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

Пример: Дефиниция на ++, конкатенираща два списъка

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

Пример: Дефиниция на функция, която проверява дали цяло число принадлежи на списък от цели числа

```
member :: Int -> [Int] -> Bool
member x [] = False
member x (y:ys) = (x == y) || (member x ys)
```

Често се дефинира горната функция по следния погрешен начин:

```
member :: Int -> [Int] -> Bool
member x [] = False
member x (x:ys) = True
member x (y:ys) = member x ys
```

Грешката е в повторното използване на променливата x в образеца на второто равенство на дефиницията. Съществува примитивна функция с аналогично на member действие и с име elem.

Пример: Дефиниция на функция, която увеличава с 1 елементите на даден списък от цели числа

```
plus1 :: [Int] -> [Int]
plus1 []  = []
plus1 (x:xs)= x+1 : plus1 xs
```

Пример: Дефиниция на функция, която филтрира по четност елементите на списък от цели числа.

```
filterEven :: [Int] -> [Int]
filterEven []  = []
filterEven (x:xs)
  | even x     = x : filterEven xs
  | otherwise  = filterEven xs
```

или

```
filterEven l = [n | n <- l, even n]
```

Пример: Дефиниция на функция, която сортира чрез вмъкване във възходящ ред списък от цели числа

```
-- включва елемент в сортиран във възходящ ред
-- списък като запазва сортировката
insert :: Int -> [Int] -> [Int]
insert x []    = [x]
insert x (y:ys)
  | x <= y     = x:(y:ys)
  | otherwise  = y : insert x ys

-- сортировка чрез вмъкване
insertSort :: [Int] -> [Int]
insertSort []       = []
insertSort (x:xs)   = insert x (insertSort xs)
```

в) *с използване на обща рекурсия*

При този вид рекурсия схемата на дефиниране е специфична за всяка задача.

Пример: Дефиниция на функция zip, която преобразува два списъка в списък от двойки от съответните елементи на двата списъка, например

```
zip [2,3,4] [12,13,14] = [(2,12),(3,13),(4,14)]
zip ['x','y'] [2,4,6,8]= [('x',2),('y',4)]
zip [2,4,6,8] [1,3]    = [(2,1), (4,3)]

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip  _      _     = []
```

или

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys)   = (x, y) : zip xs ys
zip (x:xs) []       = []
zip [] xs           = []
```

Пример: Дефиниция на функция, която сортира чрез метода “бързо сортиране” във възходящ ред списък от цели числа

```
qSort :: [Int] -> [Int]
qSort []  = []
qSort (x:xs) = qSort [y | y <- xs, y <= x] ++ [x] ++
               qSort [y | y <- xs, y > x]
```

6. Задачи

Задача 1. Да се конструира:
а) списък от първите 20 естествени числа;
б) списък от първите 20 нечетните естествени числа;
в) списък от първите 20 четни естествени числа;
г) списък от първите 25 степени на двойката;
д) списък от първите 50 триъгълни числа на Ферма;
е) списък от първите 50 пирамидални числа на Ферма.

Решение:
а) [0 .. 19]
б) [1, 3 .. 40] или [1, 3 .. 39]
в) [0, 2 .. 38]
г)

```
powerTwo :: Int -> [Int]
powerTwo 0 = [1]
powerTwo n = (2^n) : powerTwo(n-1)

zad1_g :: [Int]
zad1_g = reverse (powerTwo 24)
```

д) n-тото триъгълно число на Ферма е равно на сумата 1 + 2 + … + n. За реализацията ще използваме функцията map.

```
t_Fermat 1 = 1
t_Fermat n = n  +  t_Fermat(n-1)

zad1_d :: [Int]
zad1_d = map t_Fermat [1..50]
```

е) n-тото пирамидално число на Ферма е сума на n-тото триъгълно число на Ферма и (n-1)-вото пирамидално число на Ферма.

```
p_Fermat 1 = 1
p_Fermat n = t_Fermat n + p_Fermat(n-1)

zad1_e :: [Int]
zad1_e = map p_Fermat [1 .. 50]
```

Задача 2. Да се провери дали два списъка от цели числа са равни (състоят се от едни и същи елементи, записани в един и същ ред).

Решение:
чрез общ формат на дефиниция на функция

```
eq :: [Int] -> [Int] -> Bool
eq x y = if x == [] && y == [] then True else
          if x == [] || y == [] then False else
```

```
            (head x == head y) && eq (tail x) (tail y)
```

чрез равенства (с префиксна форма на запис на функцията)

```
eq :: [Int] -> [Int] -> Bool
eq []  []        = True
eq []  (y:ys)    = False
eq (x:xs) []     = False
eq (x:xs) (y:ys) = (x == y) && (eq xs ys)
```

чрез равенства (инфиксна форма на запис на функцията)

```
eq :: [Int] -> [Int] -> Bool
[] `eq` []          = True
[] `eq` (y:ys)      = False
(x:xs) `eq` []      = False
(x:xs) `eq`  (y:ys) = (x == y) && (xs `eq` ys)
```

Забележка: Ако се дефинира типът на eq по следния начин:

```
eq :: Eq a => [a] -> [a] -> Bool
```

се разширява областта на дефиницията на eq до такива типове данни, които допускат проверка за ==.

Задача 3. Да се дефинира функция se, която сравнява лексикографски за ≤ елементите на два списъка от цели числа. Например [3, 5, 8] ≤ [4, 5]; [3, 5, 11] ≤ [3, 6]; [3, 5] ≤ [3, 5, 7].

Решение:

```
se :: [Int] -> [Int] -> Bool
[] `se` []          = True
[] `se` (y:ys)      = True
(x:xs) `se` []      = False
(x:xs) `se` (y:ys)  = x < y || (x == y && xs `se` ys)
```

Забележка: Ако се дефинира типът на se по следния начин:

```
se :: Ord a => [a] -> [a] -> Bool
```

се разширява областта на дефиницията на se до такива типове данни, които допускат проверка за < и =.

Задача 4. Да се дефинират функции, които сравняват два списъка от цели числа за: различно (ne), по-голямо или равно (ge), строго по-малко (st) и строго по-голямо (gt).

Решение:

```
x `ne` y  =  not (x  `eq` y)
x `ge` y  =  y  `se` x
x `st` y  =  x  `se` y && x `ne` y
```

```
x `gt` y  =  y  `st` x
```

Задача 5. Да се дефинира фунция my_unzip от тип [(a, b)] -> ([a], [b]), която разединява непразен списък от двойки. Например обръщението my_unzip [(1, “aaa”), (2, “bbb”), (3, “ccc”)] да връща ([1, 2, 3], [“aaa”, “bbb”, “ccc”]).

Решение:

```
my_unzip :: [(a, b)] -> ([a], [b])
my_unzip l = (map fst l, map snd l)
```

Задачи за самостоятелна работа

Задача 1. Да се дефинира функция, която подрежда във възходящ ред елементите на тройка от реални числа.

Задача 2. Да се дефинира функция isPalin, която проверява дали символен низ е палиндром.

Задача 3. Да се дефинира функция

```
wc :: String -> (Int, Int, Int)
```

която за даден символен низ намира броя на знаковете, на думите и линиите в него. За край на дума служи знакът ‘ ‘, а за край на линия – знакът ‘\n’.

Задача 4. Да се дефинира функция, която намира сумата от онези елементи на даден списък от естествени числа, които са:

а) четни;

б) степени на двойката;

в) числа на Фибоначи;

г) прости числи.

Задача 5. Да се дефинира функция, която намира стойността на полинома $P(n, x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$.

Задача 6. Нека l1 и l2 са списъци от различни цели числи. Да се дефинира функция, която намира обединението на l1 и l2.

Задача 7. Нека l1 и l2 са списъци от различни цели числи. Да се дефинира функция, която намира сечението на l1 и l2.

Задача 8. Намерете списъка, зададен по следния начин:

```
[3*n-2 | n <- [1, 2, 3, 4, 5]]
[n^2-1 | n <- [1, 2, 3, 4, 5, 6], odd n, n > 1]
[odd n | n <- [gcd 12 18, gcd 8 6, gcd 15 25]]
[3*n-1 | n <- [2, 5 .. 17], even n, n > 3]
[(x, y)| x <- [2 .. 4], y <- [5, 7 .. 8]]
[(x, y)| x <- ['a', 'c' .. 'i'], y <- ['c', 'd']]
```

Задача 9. Да се дефинира функция my_last от тип [Int] -> Int, която намира последния елемент на непразен списък от цели числа.

Задача 10. Да се дефинира функция my_init от тип [Int] -> [Int], която за даден непразен списък от цели числа намира списъка без последния му елемент.

Задача 11. Да се дефинира функция my_take от тип Int -> [Int] -> [Int], която взема указан брой елементи от началото на списък от цели числа и ги включва в нов списък. Например обръщението my_take 4 [1, 2, 3, 4, 5, 6] да връща [1, 2, 3, 4], а my_take 0 [1, 2, 3, 4, 5] да връща [].

Задача 12. Да се дефинира функция my_replicate от тип Int -> Int -> [Int], която създава списък от указан брой копия на дадено цяло число. Например my_replicate 3 5 връща списъка [5, 5, 5].

Задача 13. Да се дефинира функция my_drop от тип Int -> [Int] -> [Int], която изтрива указан брой елементи от началото на списък. Например обръщението my_drop 3 [1, 2, 3, 4, 5, 6] да връща [4, 5, 6], а my_drop 3 [] да връща [].

Задача 14. Да се дефинира функция my_split от тип Int -> [Int] -> ([Int], [Int]), която разцепва даден списък в указана позиция. Например обръщението my_split 3 [1, 2, 3, 4, 5, 6, 7] да връща ([1, 2, 3], [4, 5, 6, 7]).

Задача 15. Да се дефинира функция my_zip3 от тип [a] -> [b] -> [c] -> [(a, b, c)], която съединява елементите на три списъка. Например обръщението my_zip3 [1, 2, 3, 4] [5, 7] [4, 5, 7] да връща [(1, 5, 4), (2, 7, 5)].

Задача 16. Да се дефинират следните функции за работа със списъци:

а) getN, която намира n-тия елемент на списък;

б) listSum, която сумира елементите на два списъка от цели числа. Връща списък от сумите на съответните елементи на списъците, които са параметри. Да се вземе предвид, че дадените списъци могат да са с различна дължина;

в) reverse, която обръща на най-външно ниво списък.

Задача 17. Да се дефинират следните функции за работа със списъци:

а) reverseAll, която има за аргумент списък, елементите на който са списъци и обръща както елементите, така и списъка;

б) position, която връща поредния номер на първото срещане на даден атом в списък;

в) set, която връща списък, съдържащ точно по веднъж всеки атом, който се среща в списък;

г) frequency, която връща списък от двойки от вида (знак, честота). Всяка двойка съдържа знак, принадлежащ на символен низ и честотата на срещането му в низа.

Задача 18. Да се дефинира:

а) списък от факториелите на естествените числа от интервала [1, n];

б) списък от кубовете на естествените числа от интервала [1, n];

в) списък от степените на 5, принадлежащи на интервал

Тема 3

Функции от по-висок ред. Функциите като параметри. Итерация. Функциите като върнати стойности. Композиция на функции. Изследване на свойства на програми на Haskell

1. Функции от по-висок ред. Функциите като параметри

За функционалните езици за програмиране е в сила:

- функциите имат тип;
- функции могат да са резултат от работата на други функции;
- функциите могат да използват в качество на параметри (аргументи) други функции.

Функциите, които използват в качество на параметри други функции, се наричат функции от по-висок ред. По този начин се подчертава тяхната разлика от обикновените функции.

Ще разгледаме някои от най-често използваните функции от по-висок ред.

Прилагане на функция над всеки от елементите на списък (map)

Функцията map е двуаргументна. Първият ѝ аргумент е функция f: a -> b, а вторият – списък [a1, a2, …, ak] с елементи от тип a. В резултат от изпълнението ѝ се получава списък с елементи от тип b имащ вида [f a1, f a2, …, f ak].

Функцията map е примитивна. Може да бъде определена по следния начин:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

или

```
map f xs = [f x | x <- xs]
```

Пример: Дефиниране на функция, която удвоява елементите на списък от цели числа

```
doubleAll :: [Int] -> [Int]
doubleAll xs = map (2*) xs
```

Пример: Дефиниране на функция, която добавя 3 към всеки от елементите на списък от цели числа

```
plus3 :: [Int] -> [Int]
plus3 xs = map (+3) xs
```

Пример: Дефиниране на функция, която събира елементите на подсписъците на даден списък от списъци от цели числа

```
sumAll :: [[Int]] -> [Int]
sumAll xs = map sum xs
```

Още примери:

```
Hugs> map (`div` 2) [2,3,4,5]
[1,1,2,2]
Hugs> map (20 `div`) [2,3,4,5]
[10,6,5,4]
Hugs> map (add 2) [1,2,3,4]
[3,4,5,6]
Hugs> map (2 `add`) [1,2,3,4]
[3,4,5,6]
Hugs> map (0:) [[1,2,3], [], [4,5,6,7]]
[[0,1,2,3],[0],[0,4,5,6,7]]
Hugs> map ("a"++) ["xxxx", "yyyy", "zzzz"]
["axxxx","ayyyy","azzzz"]
Hugs> map last ["aaaa", "bbbb", "cccc"]
"abc"
Hugs> map init ["abca", "bcdb", "cdec"]
["abc","bcd","cde"]
```

където функцията add е дефинирана по следния начин:

```
add :: Int -> Int -> Int
add x y = x + y
```

Филтриране

Функцията filter връща списък от тези елементи на даден списък, които удовлетворяват дадено условие. Може да се дефинира по следния начин:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
      | p x       = x : filter p xs
     | otherwise = filter p xs
```

или

```
filter p xs = [x | x <- xs, p x]
```

Примери:

1)
```
Hugs> filter even [2,3,4,5,6,7,8]
  [2,4,6,8]
```

2)
```
Hugs> filter isSorted [[1,2,1],[2,3,4,8],[],[4]]
  [[2,3,4,8],[],[4]]
```

където

```
isSorted :: [Int] -> Bool
isSorted xs = (xs == insertSort xs)
```

3) Ако

```
evenTuple :: (Int, String) -> Bool
evenTuple (n, s) = even n

Hugs> filter evenTuple [(1,”a”), (2,”b”), (3,”c”),
                        (4,”d”)]
[(2,"b"),(4,"d")]

4) Hugs> filter (>5) [1, 2, 3, 4, 5, 6, 7, 8]
   [6, 7, 8]
5) Hugs> filter (5>) [1, 2, 3, 4, 5, 6, 7, 8]
   [1, 2, 3, 4]
6) Hugs> filter (>5) (filter (<10)
                      [1,3,5,6,7,8,11,13,15,17])
   [6,7,8]
```

Комбиниране на zip и map (zipWith)

От два дадени списъка полиморфната функция zip конструира списък от двойки от съответните елементи на тези списъци. Има следния тип:

```
zip :: [a] -> [b] -> [(a, b)]
```

Ще дефинираме функцията zipWith, която комбинира zip и map. Функцията е полиморфна и с три аргумента: двуаргументна функция от тип a -> b -> c, списък [a1, a2, …, an] от тип [a] и списък [b1, b2, …, bm] от тип [b]. Резултатът от изпълнението на zipWith е списъкът [f a1 b1, f a2 b2, …, f ak bk] от тип [c], където k = min{n, m}. Дефинира се по следния начин:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith f (a:as) (b:bs) =  f a b : zipWith f as bs
zipWith _ _ _           =  []
```

Примери:

```
1) Hugs> zipWith (+) [1,2,3] [4,5,6,7,8,9]
   [5,7,9]
2) Hugs> zipWith (^) [2,3,4] [1,2,3,4,5,6,7]
   [2,9,64]
3) Hugs> zipWith (==) [2, 3, 4] [1,2,3,4,5,6,7]
   [False,False,False]
4) Hugs> zipWith (>) [2, 3, 4] [1,2,3,4,5,6,7]
   [True,True,True]
```

Дясноасоциативно натрупване (акумулиране) на елементи на даден списък (foldr и foldr1)

Чрез функциите foldr и foldr1 се реализира натрупване на елементите на даден списък отдясно наляво. Дефинирани са в Prelude.hs. Преди да реализираме foldr ще разгледаме няколко частни случая, които ще обобщим. Да реализираме функциите sum, product и and:

```
sum []          = 0
sum (x:xs)      = x + sum xs

product []      = 1
product (x:xs) = x * product xs

and []          = True
and (x:xs)      = x && and xs
```

Структурата на тези три дефиниции е еднаква. Различни са стойностите, които функциите връщат когато списъкът е празен (0, 1, True) и операторите, които се използват за натрупване на първия елемент на списъка към резултата от рекурсивното обръщение (+, *, &&). Предавайки тези два израза като формални параметри, се получава една от най-често използваните и полезни функции от по-висок ред за работа със списъци foldr (fold right).

Функция foldr

Има за аргументи: бинарен оператор f от тип a->b->b, начална стойност z от тип b и списък [x1, x2, …, xk] с елементи от тип a. Резултатът от изпълнението й е от тип b. Функцията извършва следните действия:

```
foldr f z [x1, x2, …, xk] =
x1 ‘f‘ ( x2 ‘f‘ ( … ‘f‘ (xk ‘f‘ z) …) =
x1 ‘f‘ (foldr z f [x2, …, xk]) =
f x1 (foldr f z [x2, …, xk])
```

Дефиниция

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

Дефинираните по-горе функции sum, product и and могат да бъдат определени чрез частична параметризация по следния начин:

```
sum       = foldr (+)     0
product   = foldr (*)     1
and       = foldr (&&) True
```

Пример: Дефиниране на функцията concat

```
concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs
```

Функция foldr1

Тази функция е вариант на foldr, в който няма начална стойност. Затова се прилага над непразни списъци. Дефинира се по следния начин:

- foldr1, приложена над дадена бинарна функция f от тип a -> b -> b и списък от един елемент [x], връща в резултат елемента x;
- прилагането на foldr1 над функция f и списък с повече от 1 елемент е еквивалентно на:

```
foldr1 f [x1, x2, …, xk] =
x1 ‘f‘ ( x2 ‘f‘ ( … ‘f‘ xk) …) =
x1 ‘f‘ (foldr1 f [x2, …, xk]) =
f x1 (foldr1 f [x2, …, xk])
```

Дефиниция

```
foldr1 :: (a -> b -> b) -> [b] -> b
foldr1 f [x]   =  x
foldr1 f (x:xs)=  f x (foldr1 f xs)
foldr1 _ []    = error "Prelude.foldr1: празен списък"
```

Примери:

```
foldr1 (+) [3, 5, 7] = 15
foldr1 (*) [3, 5, 7] = 105
foldr1 (*) [1..5] = 120
foldr1 (&&) [True, False] = False
foldr1 min [3,1,2,6,78,0] = 0
foldr1 (++) [”Empty ”, ”string”, ”!”] = "Empty string!"
```

Лявоасоциативно натрупване (акумулиране) на елементи на даден списък (foldl и foldl1)

Чрез функциите от по-висок ред foldl и foldl1 се реализира операцията акумулиране на елементите на даден списък отляво надясно.

Функция foldl

Има за аргументи: бинарен оператор f от тип a->b->a, начална стойност z от тип a и списък [x1, x2, …, xk] с елементи от тип b. Резултатът от изпълнението й е от тип a. Функцията извършва следните действия:

```
foldl f z [x1, x2, …, xk] =
(…((z `f` x1) ‘f‘  x2) … ) ‘f‘ xk
```

Дефиниция

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     =  z
foldl f z (x:xs) =  foldl f (f z x) xs
```

Функциите sum, product и and могат да бъдат определени чрез частична параметризация и по следния начин:

```
sum       = foldl (+)     0
product   = foldl (*)     1
and       = foldl (&&) True
```

Пример:

```
concat :: [[a]] -> [a]
concat xs = foldl (++) [] xs
```

Функция foldl1

Тази функция е вариант на foldl. Няма формален параметър, означаващ начална стойност и затова се прилага над непразни списъци. Дефинира се по следния начин:

- foldl1, приложена над дадена бинарна функция f от тип a -> b -> a и списък от един елемент [x], връща в резултат елемента x;
- прилагането на foldl1 над функция и списък с повече от 1 елемент е еквивалентно на:

```
foldl1 f [x1, x2, …, xk] =
(…((x1 ‘f‘  x2) `f` x3) … )‘f‘ xk
```

Дефиниция

```
foldl1  :: (a -> b -> a) -> [b] -> a
foldl1 f (x:xs)  =  foldl f x xs
foldl1 _ []   =  error "Prelude.foldl1: празен списък"
```

Примери:

```
foldl1 (+)  [3, 5, 7] = 15
foldl1 (*)  [3, 5, 7] = 105
foldl1 (&&) [True, False] = False
foldl1 max  [3, 1, 2, 6, 78, 0] = 78
```

Задача 1. Да се дефинира функция, която по зададен непразен списък от десетични цифри намира десетичното число, съставено от тези цифри. Например, ако l = [1,3,7,5], функцията да намери числото 1375.

Решение: Числото, което съответства на списъка [a1, a2, a3, a4] се получава чрез израза: (((10*0 + a1)*10+a2)*10+a3)*10+a4. Ще дефинираме функцията като използваме foldl и foldl1. Операцията има вида:

op x y = 10*x+y

Тогава изразът, намиращ числото има вида:

(((0 op a1) op a2) op a3) op a4

или

((a1 op a2) op a3) op a4

Функцията, намираща десетичното число има вида:

```
op :: Int -> Int -> Int
op x y = 10*x + y

listToNumber :: [Int] -> Int
listToNumber l = foldl op 0 l
```

или

```
listToNumber :: [Int] -> Int
listToNumber l = foldl1 op l
```

2. Итерация

Итерацията се използва често в математиката. Например взема се начална стойност и към нея се прилага функция до тогава, докато не се получи стойност, която удовлетворява дадено условие.

В Prelude.hs е реализирана функцията от по-висок ред until, която реализира итерация. Има следния синтаксис:

```
until b f x
```

където b е булева функция с тип a -> Bool, f е функция с тип a -> a, а x – начална стойност от тип a.

Семантика: Докато за елемента x условието b не е в сила, се прилага f към x.

Функцията от по-висок ред until може да бъде дефинирана по следния начин:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until b f x
  | b x        = x
  | otherwise  = until b f (f x)
```

Задача 2. Да се намери най-малката степен на 3, която е по-голяма от 100000.

Решение:

```
Hugs> until (> 100000) (3*) 1
```

Задача 3. Да се дефинира функция, която приближено пресмята $\sqrt{a}$, $a \geq 0$ с точност 0.01.

Решение: Ако началното приближение е y = 1, следващото приближение се намира по формулата: $\frac{1}{2}\left(y+\frac{a}{y}\right)$. Процесът на пресмятане на следващо приближение продължава докато абсолютната стойност на разликата на две последователни приближения стане по-малка от 0.01, т.е. $|y^2 - a| < 0.01$.

```
root :: Float -> Float
root a = until goodEnough improve 1.0
  where improve y = 0.5 * (y + a/y)
        goodEnough y = abs (y*y - a) < 0.01
```

3. Ламбда функции

За редица приложения е полезно използването на функции, зададени без имена (ламбда функции). Тъй като клавиатурата не съдържа гръцката буква λ, за означаване на ламбда функции се използва само наклонена черта \.

Синтаксис:
`\x1 x2 … xn -> expression`
Семантика: Анонимно задава функция с формални параметри x1, x2, …, xn.

Примери:
```
plus1 = \x -> x+1
addNum x = \y -> x+y
Hugs> (\x y -> x*x + y*y) 2 3
13
```

Многократно прилагане на числова функция

Нека f е числова функция, а n е положително цяло число. Да се дефинира функция repeated, която намира f^n – n-кратното последователно прилагане на f.

```
repeated :: (a -> a) -> Int -> (a -> a)
repeated f 1 = f
repeated f n = \x -> f (repeated f (n-1) x)
```
или
```
repeated :: (a -> a) -> Int -> (a -> a)
repeated f n
  | n == 1     = f
  |otherwise   = \x -> f (repeated f (n-1) x)
```

Примери:

```
Hugs> (repeated (2+) 4) 3
11
Hugs> (repeated (\x -> x*x) 3) 2
256
```

Диференциране на функция

Нека f е диференцируема функция за всяко реално число, а dx е достатъчно малко реално число. Производната на f може да се разглежда като оператор D: f -> Df, където Df е функция, чиято стойност за произволно реално число x може да се получи по формулата:

$$Df(x)=\frac{f(x+dx)-f(x)}{dx}.$$

Функцията derive, дефинирана по-долу, има за аргументи диференцируема функция f и точност dx и връща в резултат производната ѝ. Може да се дефинира по следния начин:

```
derive :: (Float -> Float) -> Float ->
          (Float -> Float)
derive f dx = \x -> (f (x+dx) - f x)/dx
```

или

```
diff :: (Float -> Float) -> (Float -> Float)
diff f = df
  where df = \x -> (f (x+dx) - f x)/dx
        dx   = 0.01
```

4. Композиция на функции

Ако f :: b -> c и g :: a -> b са функции, то f **.** g (чете се f след g) е функция с тип a -> c и е дефинирана по следния начин:

```
(f . g) x = f (g x)
```

Операторът "**.**" е дясноасоциативен и е дефиниран в Prelude.hl.

Пример:

```
Hugs> filter (not . even) [1,2,3,4,5]
[1,3,5]
Hugs> (filter (>5) . map (1+)) [1,2,3,4,5,6]
[6, 7]
```

Ще дефинираме функция от по-висок ред after, която реализира композиция на две функции.

```
after :: (b -> c) -> (a -> b) -> (a -> c)
after f g = \x -> f (g x)
```

Пример:

```
Hugs> filter (after not odd) [1,2,3,4,5]
[2,4]
```

Пример: Дефиниране на repeated

```
repeated :: (a -> a) -> Int -> (a -> a)
repeated f n
  | n == 1     = f
  |otherwise   = f . repeated f (n-1)
```

5. Изследване на свойства на програми на Haskell

Една от важните черти на функционалното програмиране е възможността строго да се доказват свойства на програмите. Най-често използваната техника е структурната индукция.

Принцип на структурната индукция при работа със списъци

Доказателството, че свойството P(xs) е в сила за всички крайни списъци xs, се извършва на следните две стъпки:

- *базов случай*

Доказва се, че е в сила P([]).

- *индуктивна стъпка*

Доказва се, че е в сила свойството P(x:xs) при предположение, че е в сила индуктивната хипотеза P(xs).

Пример: Нека функциите sum (сумираща елементите на списък от цели числа) и doubleAll (удвояваща елементите на списък от цели числа) са дефинирани по следния начин:

```
sum :: [Int] -> Int
sum []    = 0
sum (x:xs)= x + sum xs

doubleAll :: [Int] -> [Int]
doubleAll []     = []
doubleAll (x:xs) = 2*x : doubleAll xs
```

Ще докажем, че за всеки списък xs е в сила свойството:

```
sum (doubleAll xs) = 2*sum xs.
```

Базов случай: Полагаме xs = [].

Трябва да докажем, че е в сила свойството: sum (doubleAll []) = 2 * sum [].

Лява страна: sum (doubleAll []) = sum [] = 0
Дясна страна: 2*sum [] = 2 * 0 = 0.

Индуктивна стъпка:
Съгласно индуктивната хипотеза е в сила:
`sum (doubleAll xs) = 2*sum xs.`
Ще докажем, че е в сила:
`sum (doubleAll (x:xs)) = 2*sum (x:xs)`

```
Лява страна =
sum (doubleAll (x:xs)) = (съгласно деф. на doubleAll)
sum(2*x : doubleAll xs) =     (съгласно деф. на sum)
2*x + sum (doubleAll xs) =    (съгласно инд. хипотеза)
2*x + 2*sum xs
Дясна страна =
2*sum (x:xs) =                (съгласно деф. на sum)
2*(x + sum xs) =
2*x + 2*sum xs
```

Пример: Нека функциите length и (++) са дефинирани по следния начин:

```
length :: [a] -> Int
length []      = 0
length (x:xs)  = 1 + length xs

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

Ще покажем, че за всеки два списъка xs и ys е в сила
`length (xs ++ ys) = length xs + length ys.`

Базов случай: Полагаме xs = [].
Ще покажем, че
`length ([] ++ ys) = length [] + length ys`

```
лява стр. =
length ([] ++ ys) = length ys

дясна стр. =
length [] + length ys =
0 + length ys
length ys
```

Индуктивна стъпка:

Допускаме, че е в сила

```
length (xs ++ ys) = length xs + length ys
```

Ще докажем, че е в сила

```
length ((x:xs) ++ ys) = length (x:xs) + length ys

лява стр. =
length ((x:xs) ++ ys) =  (съгл. второто равенство на ++)
length (x:(xs ++ ys))=(съгл. второто равенство на length)
1 + length(xs ++ ys) =    (съгл. инд. хипотеза)
1 + length xs + length ys
дясна стр. =
length (x:xs) + length ys =
1 + length xs + length ys
```

6. Верификация на програми на езика Haskell

Да се верифицира функционална програма означава да се докаже, че тя удовлетворява зададена входно-изходна спецификация.

Например, за да се верифицира програмата:

```
fact :: Int -> Int
fact n
  | n == 1     = 1                        --   (1)
  | n > 1      = n * fact(n-1)            --   (2)
```

трябва да се докаже, че fact удовлетворява:

```
fact(1) = 1
fact(n) = 1.2. … (n-1).n, когато n > 1.
```

Подходящ механизъм за доказателството е индукцията.

Базов случай: n е 1. Дали fact(1) = 1?

Последното следва от (1) на дефинцията на fact.

Индуктивна стъпка: n > 1

Допускаме, че е в сила

```
fact(n-1) = 1.2. … .(n-1).
```

Трябва да докажем, че fact(n) = 1.2. … (n-1).n. Съгласно (2) от дефиницията на fact ,

```
fact(n) = n . fact(n-1) = n.(1.2. … .(n-1))
        = 1.2. … .n
```

7. Задачи

Задача 1. Даден е списък от цели числа. Да се дефинира функция, която намира списък от кубовете на четните числа на списъка. За целта да се използват функции от по-висок ред.

Решение:

```
cubeEven :: [Int] -> [Int]
cubeEven l = map (^3) (filter even l)
```

или

```
cubeEven = map (^3) . filter even
```

Задача 2. Даден е списък от цели числа. Да се дефинира функция, която намира сумата от квадратите на нечетните числа на списъка. За целта да се използват функции от по-висок ред.

Решение:

```
sumSqOdd :: [Int] -> Int
sumSqOdd l = foldl (+) 0 (map (^2) (filter odd l))

sumSqOdd = foldl (+) 0 . map (^2) . filter odd
```

Задача 3. Даден е списък, елементите на който са непразни списъци от цели числа. Да се създаде списък, елементите на който са минималните елементи на всеки подсписък на дадения списък.

Решение:

```
-- minList намира минималния елемент
-- на непразен списък от цели числа
minList :: [Int] -> Int
minList [x]    = x
minList (x:xs) = min x minList xs
minList []     = error "Empty List"
```

или

```
minList l = foldr1 min l
```

Тогава

```
minLL :: [[Int]] -> [Int]
minLL l = map minList l
```

Задача 4. Да се намери първото просто число, което е по-голямо от 100000.

Решение:

```
prime :: Int -> Bool
prime 1 = False
prime n = ([k | k <- [2 .. n `div` 2],
```

```
            n `mod` k == 0] == [])

firstPrime :: Int
firstPrime = until prime (2+) 99999
```

Задача 5. Да се дефинира функция deriveN, която намира n-тата частна производна на n-кратно диференцируема реална функция.

Решение:

```
derive :: (Float -> Float) -> Float ->
          (Float -> Float)
derive f dx = \x -> (f (x+dx) - f x)/dx

deriveN :: Int -> (Float -> Float) -> Float ->
                  (Float -> Float)
deriveN n f dx
  | n == 1  = derive f dx
  | n > 1   = derive (deriveN (n-1) f dx) dx
```

Задачи за самостоятелна работа

Задача 1. Да се намери резултатът от оценката на израза:
а) map abs [1, -2, 3, -4, 5, -6]
б) map (2+) [1, 3, 5, 7, 9]
в) map (+3) [1, 3, 5, 7, 9]
г) map (2^) [1, 2, 3, 4, 5]
д) map (^2) [1, 2, 3, 4, 5]
е) map (`div` 2) [1, 2, 3, 4, 5]
ж) map (0:) [[1,2,3], [4,5,6,7], [], [5]]
з) map (++ "\n")["aaaa", "bbbb", "cccc", "dddd"]
Задача 2. Да се намери резултатът от оценката на израза:
а) filter odd [1, -2, 3, -4, 5, 6]
б) filter even [1, 3, 5, 6, 7, 9]
в) filter (>3) [1, 3, 5, 7, 9]
г) filter (4>=) [1, 2, 3, 4, 5, 6, 7]
д) filter (==2) [1, 2, 3, 4, 5, 2]
е) filter (2/=) [1, 2, 3, 4, 5, 2]
Задача 3. Да се намери резултатът от оценката на израза:
а) zipWith (*) [1, -2, 3, -4, 5, 6] [3, 4, 5, 6]
б) zipWith (**) [3, 2, 1] [1, 2, 3, 4]
в) zipWith (<) [3, 2, 1] [1, 2, 3, 4]
г) zipWith (>=) [3, 2, 1] [1, 2, 3, 4]

д) zipWith (^) [3.5, 2.1, 1.5] [1, 2, 3, 4, 5, 6]

е) zipWith logBase [3, 2, 10] [1, 2, 100, 4]

Задача 4. Да се намери резултатът от оценката на израза:

а) foldr (*) 1 [3, 4, 5, 6]

б) foldr (+) 100 [3, 4, 5, 6]

в) foldr (+) 100 []

г) foldr1 (+) []

д) foldl (^) 1 [3, 4]

е) foldl (^) 2 [2, 3]

ж) foldl1 (^) [2, 3]

з) foldl1 (++) [[2, 3], [4, 6]]

и) foldl1 (++) [“aaaa”, “bbbb”, “cccc”]

Задача 5. Като се използват дефинираните функции от по-висок ред, да се дефинира функция, която проверява дали всички елементи на списък са положителни.

Задача 6. Като се използват дефинираните функции от по-висок ред, да се дефинира функция, която:

а) намира минималната стойност на функцията f за входни стойности от 0 до n;

б) проверява дали всички стойности на функцията f за входни стойности от 0 до n са равни;

в) проверява дали всички стойности на функцията f за входни стойности от 0 до n са положителни;

г) проверява дали f 0, f 1, …, f n са в нарастващ ред.

Задача 7. Да се дефинира функция, която намира корен p-ти (p > 1) от x с точност ε, като се използва итерационната формула:

$$y_0 = x$$

$$y_{n+1} = \frac{1}{p}.\left((p-1).y_n + \frac{x}{y_n^{p-1}}\right),\ n = 0, 1, 2, \ldots$$

(ε > 0 е достатъчно малко реално число).

Задача 8. Даден е списък, елементите на който са непразни списъци от цели числа. Да се създаде списък, елементите на който са максималните елементи на всеки подсписък на дадения списък.

Задача 9. Квадратна матрица A(n, n), n ≥ 1 се представя чрез непразен списък от n списъка от по n реални числа. Да се дефинира функция, която повдига на квадрат всеки елемент на A.

Задача 10. Даден е списък, компонентите на който са непразни списъци от реални числа. Да се дефинира функция, която прилага функцията f над всяко число на дадения списък.

Задача 11. Даден е списък, компонентите на който са списъци от реални числа. Да се дефинира функция, която създава списък от сортираните във възходящ ред подсписъци на дадения списък.

Задача 12. Даден е списък, компонентите на който са m непразни списъка от по n естествени числа (n, m > 0). Да се дефинира функция, която намира сумата от онези елементи на списъка, които са прости числа.

Задача 13. Да се намери първото просто число, завършващо на 7, което е по-голямо от 100000.

Задача 14. Функцията reverse е дефинирана по следния начин:

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

Да се верифицира reverse. Да се докаже, че е в сила свойството:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs.
```

Задача 15. Да се верифицират следните функции:

а) length

б) sum

в) double

Задача 16. Да се докаже, че операторът ++ е асоциативен.

Задача 17. Да се докаже, че за всеки краен списък xs е в сила:

а) sum (reverse xs) = sum xs

б) length (reverse xs) = length xs.

Литература

1. H. Abelson, G. Sussman. Structure and Interpretation of Computer Programs (2nd ed.). MIT Press, 1996.
2. M. Felleisen et al. How to Design Programs: An Introduction to Computing and Programming. MIT Press, 2001.
3. P. Hudak, Peterson J., Fasel J. A Gentle Introduction to Haskell 98, 1999 (Internet, 2008).
4. S. Thompson. Haskell: The Craft of Functional Programming (2nd ed.). Addison-Wesley, 1999.