

Лекция 1

**Основни принципи на
функционалното програмиране.
Начални сведения за Haskell и
Hugs**

компютърна система = хардуер + софтуер

Софтуерът (програмното осигуряване)
управлява поведението на хардуера.

софтуер
(програмно
осигуряване) = програми + данни

Данни: информация, която може да бъде съхранена в паметта на компютъра.

Примери за данни:

- ✓ числа
- ✓ букви
- ✓ съобщения по електронната поща
- ✓ карти
- ✓ песни върху CD
- ✓ видео клипове
- ✓ щраквания на мишката
- ✓ програми

Програми: описания на начини за манипулиране на данните.

Езици за програмиране: изкуствени езици, създадени специално за записване на програми.

Изграждане на софтуерни системи

Една голяма система може да съдържа милиони редове програмен код.

Софтуерните системи са измежду най-сложните артефакти, които някога са създавани.

Изграждат се чрез комбиниране на съществуващи компоненти, доколкото това е възможно.

Езици за програмиране

Формални изкуствени езици, предназначени за записване на програми.

Съществуват стотици езици за програмиране, всеки от които има своите силни и слаби страни.

Класификация: съществуват различни класификационни признаци (машинно-ориентирани езици и езици от високо ниво)

Типове езици за програмиране ОТ ВИСОКО НИВО

- Процедурни (императивни)
как?

програма = алгоритъм + структури от данни

Типове езици за програмиране

- Декларативни (дескриптивни)
какво?

програма = списък от дефиниции на функции

или

списък от равенства

или

факти + правила

Основни характеристики на функционалното програмиране

Програмирането във функционален стил се състои от:

- дефиниране на функции, които пресмятат (връщат) стойности. При това тези стойности се определят еднозначно от стойностите на съответните аргументи (фактически параметри)

- прилагане (апликация) на тези функции върху подходящи аргументи, които също могат да бъдат обръщения към функции

Основни предимства на функционалния стил на програмиране

- може да се извършва лесна проверка и поправка на съответните програми поради липсата на странични ефекти
- могат да бъдат доказвани строго (с математически средства) свойства на функционалните програми

Основни недостатъци на функционалния стил на програмиране

- строгата функционалност понякога изисква многократно пресмятане на едни и същи изрази (в Haskell този проблем е преодолян)
- неестествено и често неефективно е използването им при решаване на задачи от процедурен (алгоритмичен) характер

Нашият подход

Haskell – език за строго функционално програмиране:

- език от много високо ниво (грижата за много детайли се поема автоматично)
- с голяма изразителна сила, позволява писане на много кратък и компактен код
- подходящ за работа със сложни данни и за комбиниране на готови компоненти
- дава приоритет на времето на програмиста, а не на компютърното време

Функции

Функцията е програмна част, която връща стойност (резултат).

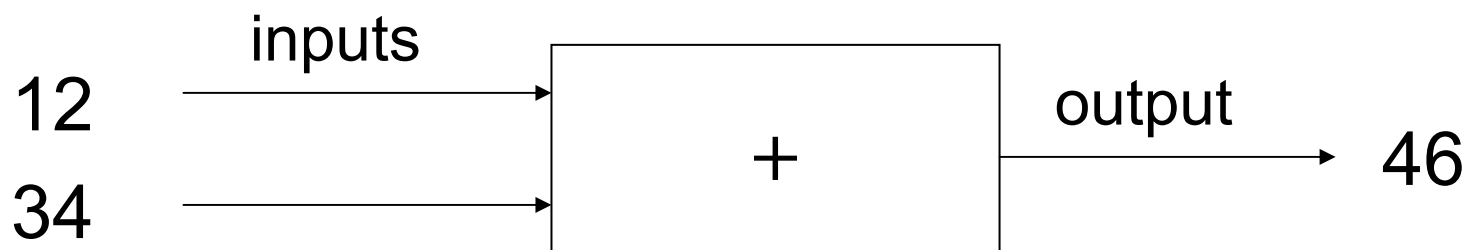
Изобразяване на функциите: чрез диаграми от вида



inputs: входни стойности (аргументи или параметри)

output: изход (върнатата стойност, резултат)

Пример: функцията + за събиране на числа



Процесът на задаване на конкретни стойности на аргументите на функцията и пресмятане на съответната ѝ стойност се нарича **апликация** (прилагане на функцията).

Може да се създават (дефинират) функции с различен брой аргументи (0, 1, 2 или повече).

Аргументите на функцията, както и върнатата от нея стойност, могат да бъдат от различни **типове** (не е задължително да бъдат числа).

Величини. Типове

Величините са основно средство за изразяване на стойностите, с които се работи в една програма.

Всяка величина се характеризира с име, тип и (текуща) стойност.

Типът представлява множество от допустими стойности заедно с определена съвкупност от операции, приложими върху тези стойности.

Общи сведения за езика Haskell

Haskell е език за строго функционално програмиране. Създаден е в края на 80-те години на 20-ти век. Носи името на Haskell B. Curry – един от пионерите на λ -смятането (математическа теория на функциите, дала тласък в развитието на множество езици за функционално програмиране).

Най-популярната среда за програмиране на Haskell е **Hugs** (1998). Тя предоставя много добри средства за обучение и се разпространява безплатно за множество платформи.

Haskell home page:

[http: //www.haskell.org/](http://www.haskell.org/)

Дефиниции

Всяка функционална програма представлява поредица от дефиниции на функции и други величини.

Всяка дефиниция на Haskell свързва (асоциира) дадено **име** (идентификатор) със **стойност** от определен **тип**.

В най-простия случай дефинициите имат вида

`name :: type`

`name = expression`

Една дефиниция от посочения вид свързва името от лявата страна на равенството със стойността на **израза** от дясната страна.

Символът “::” би трябвало да се чете “е от тип”.

Имената на функциите и останалите величини започват с малки букви, докато имената на типовете започват с главни букви.

Пример

```
size :: Int
```

```
size = 12+13
```

Изрази

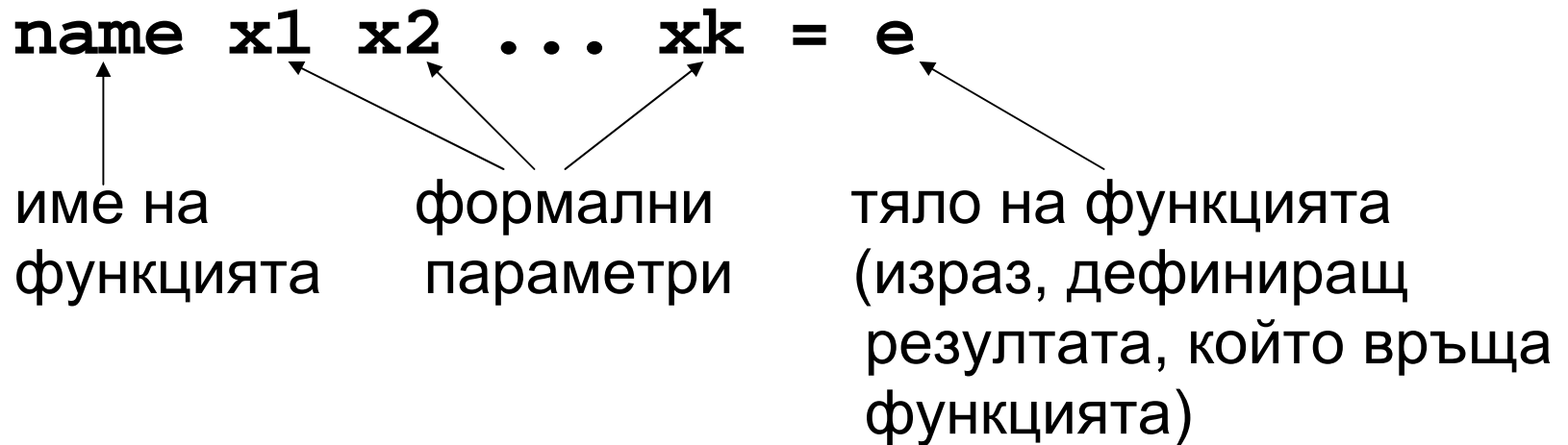
Всеки израз представлява правило за пресмятане (намиране) на стойност.

Изразите се образуват чрез композиция на операции над определени величини.

Операциите се означават с определени знаци или с идентификатори (пореждици от букви и цифри, които започват с буква). Всяка операция се прилага към стойности от определен(и) тип(ове) и формира стойност от определен тип.

Дефиниции на функции

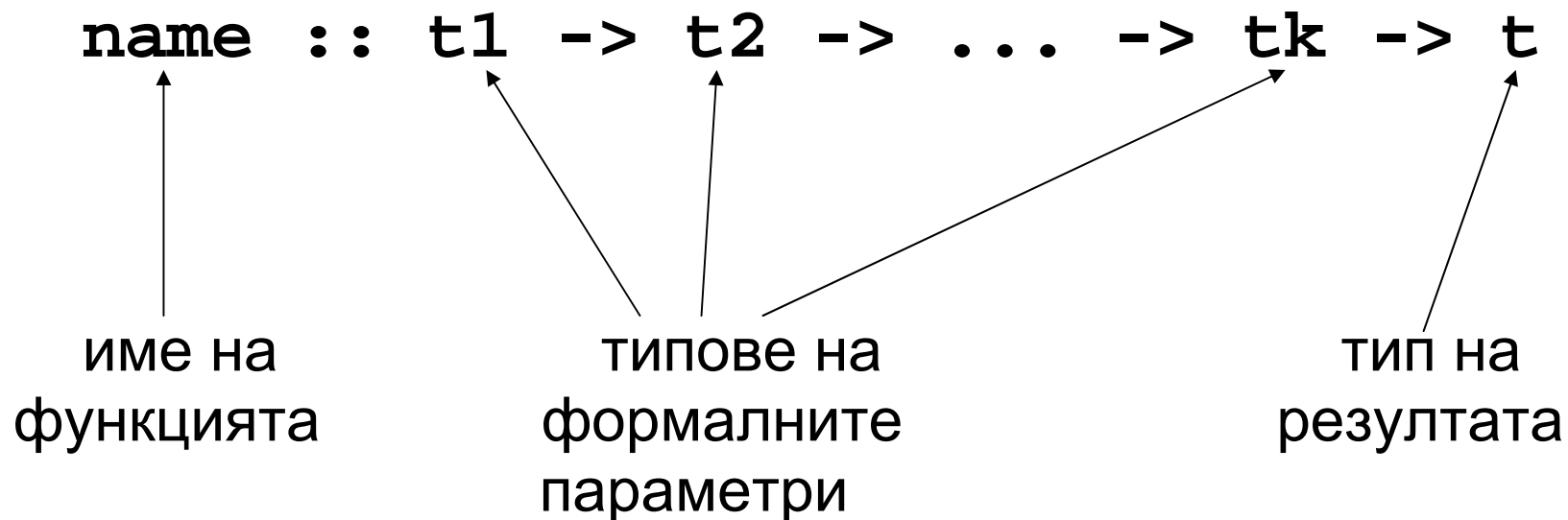
Дефинициите на функции имат следния общ вид:



Формални параметри (аргументи): имена на независимите променливи на функцията (в математическия смисъл на това понятие).

Дефиницията на функция трябва да бъде предшествана от декларация на нейния тип (на типовете на аргументите и типа на резултата, който връща функцията).

Общ вид на декларация на типа на функция:



Примери

```
square :: Int -> Int
```

```
square n = n*n
```

```
> square 3
```

```
9
```

```
> square 5
```

```
25
```



```
average :: Float -> Float -> Float
```

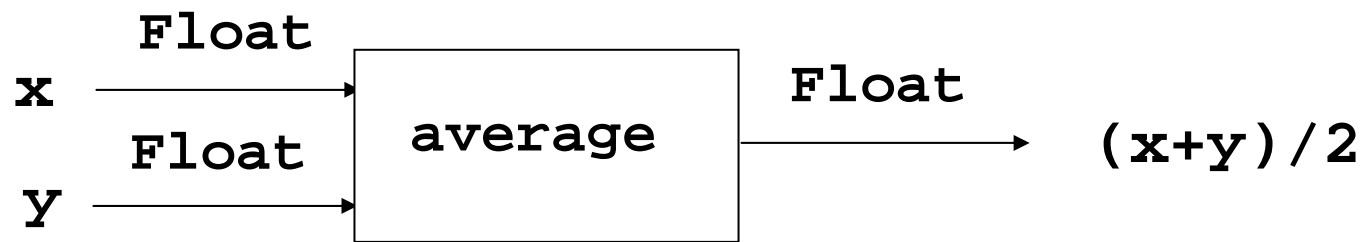
```
average x y = (x+y)/2
```

```
> average 3.4 5.6
```

```
4.5
```

```
> average 3 4
```

```
3.5
```



Общ вид на програмата на Haskell

Програмите на Haskell обикновено се наричат **скриптове (scripts)**. Освен програмния код (поредица от дефиниции на функции) един скрипт може да съдържа и коментари.

Има два различни стила на писане на скриптове, които съответстват на две различни философии на програмиране.

Традиционно всичко в един програмен файл (файл с изходния код на програма на Haskell) се интерпретира като програмен текст (код), освен ако за нещо е отбелязано специално, че представлява коментар. Скриптовете, написани в такъв (traditional) стил, се съхраняват във файлове с разширение “.hs”.

Традиционно коментари се означават по два начина. Символът “--” означава начало на коментар, който продължава от съответната позиция до края на текущия ред. Коментари, които съдържат произволен брой знакове и евентуално заемат повече от един ред, могат да бъдат заключени между символите “{-“ и “-}”.

Алтернативният (literate) подход предполага, че всичко във файла е коментар освен частите от текста, специално означени като програмен код.

В Пример 2 програмният текст е само в редовете, започващи с “>” и отделени от останалия текст с празни редове.

Този вид скриптове се съхраняват във файлове с разширение “.lhs”.

Пример 1. A traditional script

```
{- #####  
    MyFirstScript.hs  
##### -}  
  
-- The value size is an integer (Int), defined to be  
-- the sum of 12 and 13.  
  
size :: Int  
size = 12+13  
  
-- The function to square an integer.  
  
square :: Int -> Int  
square n = n*n
```

```
-- The function to double an integer.
```

```
double :: Int -> Int
```

```
double n = 2*n
```

```
-- An example using double, square and size.
```

```
example :: Int
```

```
example = double (size - square (2+2))
```

Пример 2. A literate script

```
{- #####  
    MyFirstLiterate.lhs  
##### -}
```

The value `size` is an integer (`Int`), defined to be the sum of 12 and 13.

```
> size :: Int  
> size = 12+13
```

The function to square an integer.

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

The function to double an integer.

```
> double :: Int -> Int
> double n = 2*n
```

An example using double, square and size.

```
> example :: Int
> example = double (size - square (2+2))
```

Библиотеки на Haskell

Haskell поддържа множество вградени типове данни: цели и реални числа, булеви стойности, низове, списъци и др., както и предлага вградени функции за работа с данни от тези типове.

Дефинициите на основните вградени функции в езика се съдържат във файл (**the standard prelude**) с името `Prelude.hs`. По подразбиране при стартиране на Hugs (или друга среда за програмиране на Haskell) най-напред се зарежда the standard prelude, след което потребителят може да започне своята работа.

Напоследък, с цел намаляване на обема на the standard prelude, дефинициите на част от вградените функции се преместват от the standard prelude в множество **стандартни библиотеки**, които могат да бъдат включени от потребителя в средата на Haskell при необходимост.

Модули

Възможно е текстът на една програма на Haskell да бъде разделен на множество компоненти, наречени **модули**.

Всеки модул има свое **име** и може да съдържа множество от дефиниции на Haskell. За да се дефинира даден модул, например `Aut`, е необходимо в началото на програмния текст в съответния файл да се включи ред от типа на

```
module Aut where
```

```
.....
```

Един модул може да **импортира** дефиниции от други модули. Например модулет Bee ще може да импортира дефиниции от модула Aut чрез включване на оператор `import` както следва:

```
module Bee where
import Aut
    . . . . .
```

В случая операторът `import` означава, че при дефинирането на функции в Вее могат да се използват всички (**видими**) дефиниции от Aut.

Механизмът на модулите поддържа споменатите по-горе библиотеки.

Механизмът на модулите позволява да се определи кои дефиниции да бъдат достъпни чрез **експортиране** от даден модул за употреба от други модули.

Лекция 10

**Графи. Програми на Haskell за намиране
на пътища в граф**

Основни дефиниции

Граф Γ се нарича наредената двойка $\langle V, R \rangle$, където $V = \{a_1, a_2, \dots, a_n\}$ е множество, R е ненареден списък от двойки елементи на V .

Елементите на V се наричат **върхове** (**възли**) на графа Γ , а елементите на R – **ребра** (**дъги**) на графа Γ .

Ако ребрата на Γ са наредени двойки, графът Γ се нарича *ориентиран*; ако ребрата не са наредени, Γ се нарича *неориентиран* граф.

Геометричното изображение на графите се получава, като:

- изберем толкова различни точки, колкото върха има графът, и съпоставяме на всеки връх по една точка;
- върховете a_i и a_j съединим с:
 - линия, ако графът е неориентиран и $(a_i, a_j) \in R$;
 - стрелка, ако графът е ориентиран и $(a_i, a_j) \in R$.

Основно понятие в теорията на графите е понятието **път**.

Под *път в ориентиран граф* се разбира всяка редица от ребра, имаща вида

$$\langle a_{i_1}, a_{i_2} \rangle, \langle a_{i_2}, a_{i_3} \rangle, \dots, \langle a_{i_{l-1}}, a_{i_l} \rangle, \langle a_{i_l}, a_{i_{l+1}} \rangle.$$

Ще казваме, че този път води от a_{i_1} до $a_{i_{l+1}}$ (има начало a_{i_1} и край $a_{i_{l+1}}$) и има *дължина* l .

Под *път в неориентиран граф* се разбира всяка редица от ребра, имаща вида

$$\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle, \dots, \langle e_1, e_2 \rangle, \langle l_1, l_2 \rangle,$$

където единият елемент на всяко ребро принадлежи на левия съсед, а другият елемент – на десния съсед. От крайните ребра се изисква само по един техен елемент да принадлежи на съседа на реброто. Другите два елемента определят *краищата* на пътя, а броят на ребрата – *неговата дължина*.

Забележка. Често пътищата в графа се представят като поредици от участващите в тях (съставлящите ги) върхове (възли).

Например пътят

$\langle a_{i_1}, a_{i_2} \rangle, \langle a_{i_2}, a_{i_3} \rangle, \dots, \langle a_{i_{l-1}}, a_{i_l} \rangle, \langle a_{i_l}, a_{i_{l+1}} \rangle$ в ориентирувания граф Γ се представя чрез поредицата (списъка) от върхове $a_{i_1}, a_{i_2}, \dots, a_{i_l}, a_{i_{l+1}}$.

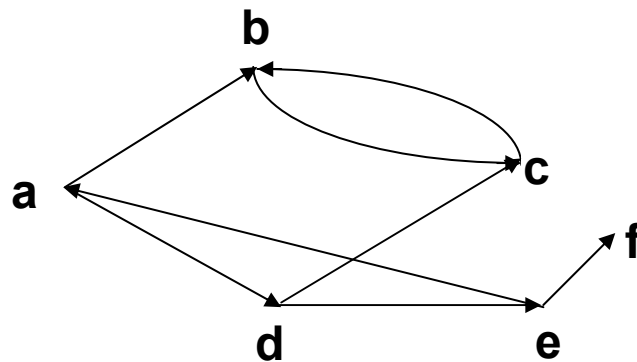
Път със съвпадащи краища (съвпадащи начало и край) се нарича *цикъл*. Път, който съдържа цикъл, се нарича *цикличен*; в противен случай пътят се нарича *ацикличен*.

Графът $\Gamma = \langle V, R \rangle$ се нарича *краен*, ако V и R са крайни.

Представяне на (ориентирани) графи със средствата на езика Haskell

Най-често графът се представя чрез списък, елементите на който съответстват на ребрата (дъгите), които започват от съответните върхове (възли) на този граф.

Примерен граф:



Фиг. 1

Примерни представяния на графа от фиг. 1:

```
[ ("a", ["b", "d"]), ("b", ["c"]), ("c", ["b"]),  
  ("d", ["c", "e"]), ("e", ["a", "f"])]
```

```
[ ("a", ["b", "d"]), ("b", ["c"]), ("c", ["b"]),  
  ("d", ["c", "e"]), ("e", ["a", "f"]), ("f", [])]
```

Второто представяне е по-удобно в случаите, когато е необходимо да се работи със списъка от върховете на графа или поне с техния брой. То е по-целесъобразно и в случаите, когато в графа има “изолирани” върхове (такива върхове, от които не започват и в които не завършват дъги от графа).

**Дефиниции на някои функции
за работа с (ориентирани) графи.
Намиране на път в граф**

```
type Node = String
```

```
type Graph = [(Node,[Node])]
```

```
type Path = [Node]
```

```
graph1 :: Graph
```

```
graph1 = [ ("a",["b","d"]), ("b",["c"]), ("c",["b"]),  
           ("d",["c","e"]), ("e",["a","f"]) ]
```

```
graph2 :: Graph
graph2 = [ ("a", ["b", "d"]), ("b", ["c"]), ("c", ["b"]),
           ("d", ["c", "e"]), ("e", ["a", "f"]), ("f", []) ]

-- graph1 и graph2 са различни представяния на
-- примерния граф от фиг. 1. graph1 включва само
-- елементи, които имат за ключове такива върхове,
-- от които започват ребра в графа; graph2 има
-- за ключове на елементите си всички върхове
-- на графа (вкл. в "изолираните" върхове, ако има
-- такива) .
```

```
assoc :: Eq a => a -> [(a,[b])] -> (a,[b])
-- assoc :: Node -> [(Node,[Node])] -> (Node,[Node])
-- Асоциативно търсене в списък от двойки по даден ключ.
assoc key [] = (key,[])
assoc key (x:xs)
  | fst x==key = x
  | otherwise  = assoc key xs

successors :: Node -> Graph -> [Node]
successors node graph = snd (assoc node graph)
```

```

is_a_node :: Node -> Graph -> Bool
-- Проверява дали node е връх в графа graph.
is_a_node node graph = rt node || lf node
  where rt :: Node -> Bool
        rt x = elem x (map fst graph)
        lf :: Node -> Bool
        lf x = elem x (concat (map snd graph))

is_a_path :: Path -> Graph -> Bool
-- Проверява дали даден списък от "върхове" е път
-- в даден граф.
is_a_path [] _ = True
is_a_path [node] graph = is_a_node node graph
is_a_path (n1:(n2:others)) graph
  | elem n2 (successors n1 graph)
    = is_a_path (n2:others) graph
  | otherwise = False

```



```
correct_path :: Path -> Graph -> Bool
-- Проверява дали даден списък от "върхове" е
-- ацикличен път в даден граф.
correct_path path graph = (is_a_path path graph)
                           && (not (cycled path))

cycled :: Path -> Bool
-- Проверява дали в даден "път" се съдържа цикъл.
cycled [] = False
cycled (n:ns)
  | elem n ns = True
  | otherwise = cycled ns
```

```
gen_next :: Path -> Graph -> [Path]
-- Връща като резултат списък от продълженията
-- на даден път.
gen_next path graph = map (\x -> (x:path))
    (successors (head path) graph)

generate_paths :: [Path] -> Graph -> [Path]
-- Връща като резултат списък от продълженията
-- на пътищата от даден списък.
generate_paths paths graph
    = concat (map (\x -> gen_next x graph) paths)
```

```
connected :: Node -> Node -> Int -> Graph -> Bool
-- Проверява дали съществува път в даден граф graph
-- от върха node1 до върха node2, който има дължина,
-- не по-голяма от n.
connected node1 node2 n graph
  | node1==node2 = True
  | n<=0         = False
  | otherwise    = connect1 [[node1]] node2 n graph
```

```
connect1 :: [Path] -> Node -> Int -> Graph -> Bool
connect1 paths node n graph
  | n<=0          = False
  | null paths    = False
  | otherwise     = if (elem node (map head lnp)) then True
                     else connect1 lnp node (n-1) graph
    where lnp :: [Path]
          lnp = generate_paths paths graph
```

```
assoc1 :: Eq a => a -> [[a]] -> [a]
-- Асоциативно търсене в списък от списъци
-- по даден ключ.
assoc1 _ [] = []
assoc1 key (x:xs)
  | head x==key = x
  | otherwise   = assoc1 key xs
```

```
path :: Node -> Node -> Graph -> Path
-- Намира път в даден граф graph от върха node1
-- до върха node2. Работи с графи, в които явно
-- са посочени "синовете" на всички възли.
-- Основава се на идеята, че ако съществува път
-- между два върха, то съществува и ацикличен път
-- между тези върхове, а дължината на най-дългия
-- ацикличен път в графа не може да надхвърли n-1,
-- където n е броят на върховете в графа.
path node1 node2 graph
  | node1==node2 = [node1]
  | otherwise    = build_path [[node1]] node2
                        (length graph-1) graph
```

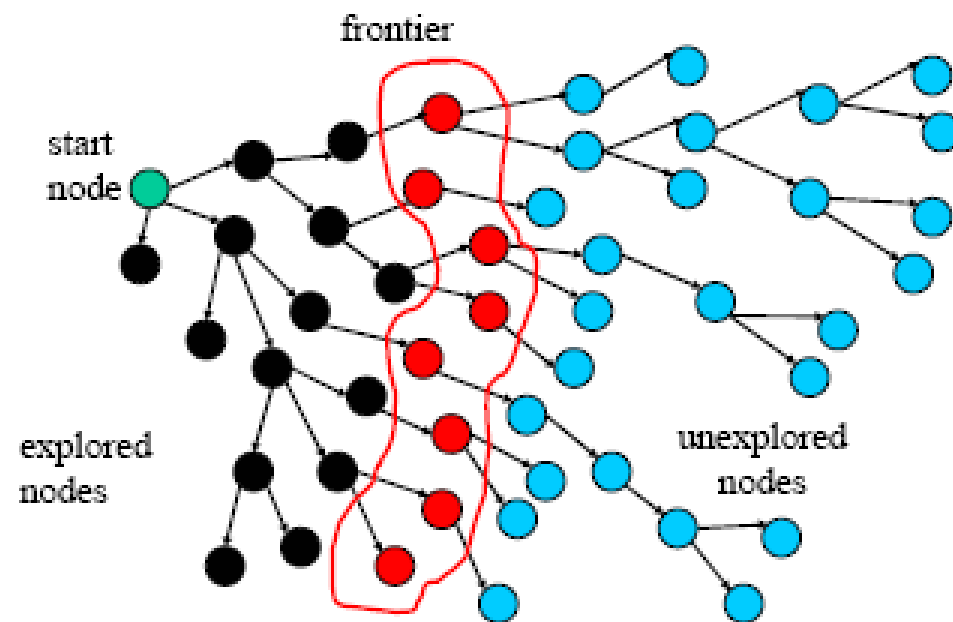
```
build_path :: [Path] -> Node -> Int -> Graph -> Path
build_path paths node n graph
  | n<=0          = []
  | null paths    = []
  | otherwise     = if (null pth)
    then build_path lnp node (n-1) graph
    else reverse pth
    where lnp :: [Path]
          lnp = generate_paths paths graph
          pth :: Path
          pth = assoc1 node lnp
```

Забележка. Дефинираните по-горе функции за намиране на път в граф реализират един “наивен” метод, който е добра илюстрация на използването на функции от по-висок ред при работа със съставни структури от данни (при моделирането на графи чрез списъци).

Основни стратегии на търсене на път в граф

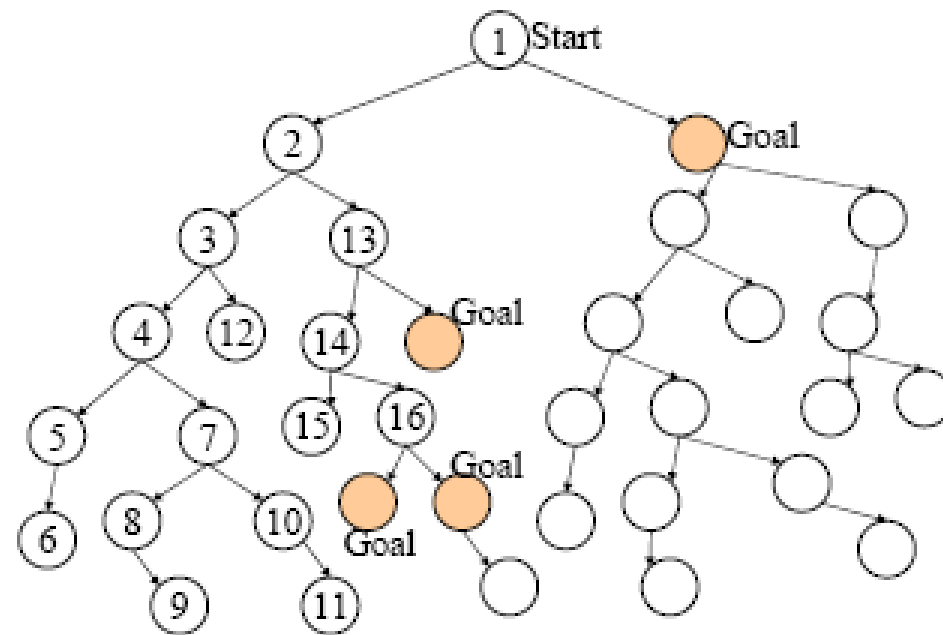
Общ алгоритъм за търсене:

- Даден е граф; известни са началният възел (Start) и целевият възел (Goal);
- Последователно се изследват пътищата от началния възел;
- Поддържа се фронт/граница (frontier) от пътищата, които са били изследвани;
- По време на процеса на търсене фронтът се разширява в посока към неизследваните възли, докато се достигне до целевия възел;
- Начинът, по който фронтът се разширява, както и това, точно кой възел от фронта се избира за разширяване на фронта на следващата стъпка, дефинира *стратегията на търсене (search strategy)*.



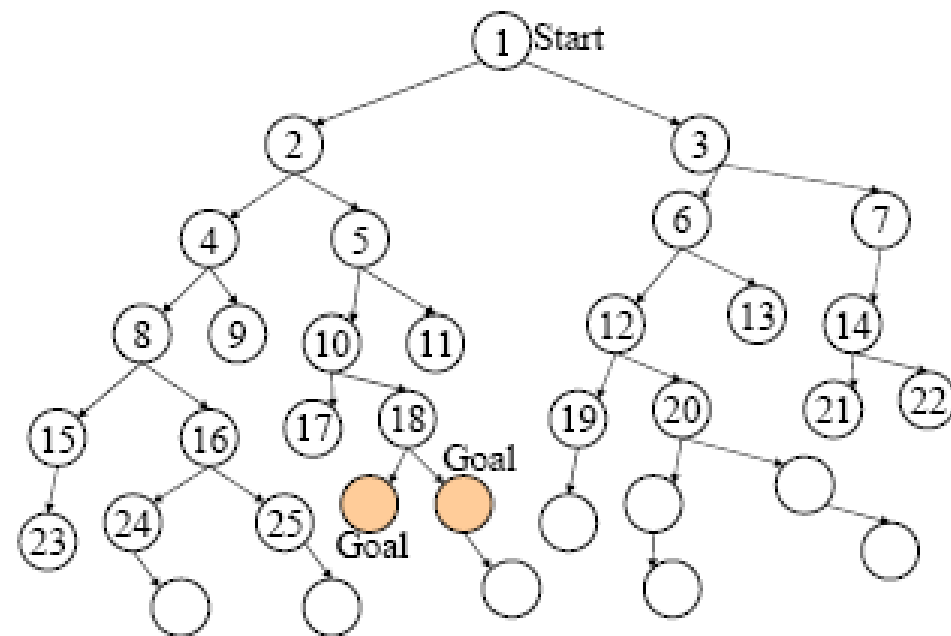
Търсене в дълбочина (depth-first search)

- При търсенето в дълбочина фронтът се обработва като стек.
- Ако фронтът е $[p1, p2, \dots]$, то:
 - избира се $p1$;
 - пътищата $p1'$, $p1''$, \dots , $p1(k)$, които разширяват $p1$, се добавят в началото на стека (преди $p2$), т.е. фронтът придобива вида $[p1', p1'', \dots, p1(k), p2, \dots]$;
 - $p2$ се обработва едва след като се изследват всички пътища, които са продължения на $p1$.



Търсене в широчина (breadth-first search)

- При търсенето в широчина фронтът се обработва като опашка.
- Ако фронтът е $[p_1, p_2, \dots, p_n]$, то:
 - избира се p_1 ;
 - пътищата $p_1', p_1'', \dots, p_1(k)$, които разширяват p_1 , се добавят в края на опашката (след p_n), т.е. фронтът придобива вида $[p_2, \dots, p_n, p_1', p_1'', \dots, p_1(k)]$;
 - p_1' се обработва едва след като се изследват всички пътища p_2, \dots, p_n .
- Намира най-краткия път от началния до целевия възел.



Програмна реализация

```
type Node = String
```

```
type Graph = [(Node, [Node])]
```

```
type Path = [Node]
```

```
graph :: Graph
```

```
-- Примерен граф.
```

```
graph = [ ("a", ["b", "c", "d"]), ("b", ["e", "f"]),  
          ("c", ["g", "i"]), ("d", ["f", "h"]),  
          ("e", ["i"]), ("f", ["j"]), ("h", ["j"]) ]
```

```
assoc :: Eq a => a -> [(a,[b])] -> (a,[b])
-- assoc :: Node -> [(Node,[Node])] -> (Node,[Node])
-- Асоциативно търсене в списък от двойки
-- по даден ключ (същото както по-горе) .
assoc key [] = (key,[])
assoc key (x:xs)
    | fst x==key = x
    | otherwise  = assoc key xs

successors :: Node -> Graph -> [Node]
successors node graph = snd (assoc node graph)
```



```
extend :: Path -> Graph -> [Path]
-- Разширяване на фронта.
extend path graph = concat
  (map (\x -> if (elem x path) then [] else [x:path])
    (successors (head path) graph))
```

```
depth_first_search :: Node -> Node -> Graph -> Path
-- Търсене в дълбочина.
depth_first_search node1 node2 graph = reverse
    (depth_first [[node1]] node2 graph)

depth_first :: [Path] -> Node -> Graph -> Path
depth_first [] _ _ = []
depth_first (path:others) goal graph
    | goal==head path = path
    | otherwise       = depth_first
        ((extend path graph)++others) goal graph
```

```
breadth_first_search :: Node -> Node -> Graph -> Path
-- Търсене в широчина.
breadth_first_search node1 node2 graph = reverse
    (breadth_first [[node1]] node2 graph)

breadth_first :: [Path] -> Node -> Graph -> Path
breadth_first [] _ _ = []
breadth_first (path:others) goal graph
    | goal==head path = path
    | otherwise       = breadth_first
        (others++(extend path graph)) goal graph
```

Примери

`depth_first_search "a" "i" graph` →
`["a", "b", "e", "i"]`

`breadth_first_search "a" "i" graph` →
`["a", "c", "i"]`

Лекция 11

Оценяване на изрази.

Работа с безкрайни списъци в Haskell

“Мързеливо” оценяване (lazy evaluation)

“Мързеливото” оценяване (lazy evaluation) е стратегия на оценяване, която по стандарт стои в основата на работата на всички интерпретатори на Haskell. Същността на тази стратегия е, че интерпретаторът оценява даден аргумент на дадена функция само ако (и доколкото) стойността на този аргумент е необходима за пресмятането на целия резултат. Нещо повече, ако даден аргумент е съставен (например е вектор или списък), то се оценяват само тези негови компоненти, чиито стойности са необходими от гледна точка на получаването на резултата. При това дублиращите се подизрази се оценяват по не повече от един път.

Най-важно от гледна точка на оценяването на изрази в Haskell е *прилагането на функции*. Основната идея тук е еквивалентна на т. нар. **оценяване чрез заместване** в “традиционните” езици за функционално програмиране.

Оценяването на израз, който е обръщение към функцията f с аргументи a_1, a_2, \dots, a_k , се състои в заместване на формалните параметри от дефиницията на f съответно с изразите a_1, a_2, \dots, a_k и оценяване на така получения частен случай на тялото на дефиницията.

Например, ако

$f\ x\ y = x+y$, то

$f\ (9-3)\ (f\ 34\ 3)$

$\longrightarrow (9-3) + (f\ 34\ 3)$

При това изразите $(9-3)$ и $(f\ 34\ 3)$ не се оценяват преди да бъдат предадени като аргументи на f .

Доколкото операцията събиране изисква аргументите ѝ да бъдат оценени, оценяването на горния израз продължава по следния начин:

$f(9-3)(f\ 34\ 3)$

→ ...

→ $6 + (34 + 3)$

→ $6 + 37$

→ 43

В конкретния случай бяха оценени и двата аргумента (фактически параметъра), но това не винаги е така.

Ако например дефинираме

$g\ x\ y = x + 12$, то

$g(9-3)(g\ 34\ 3)$

→ $(9-3) + 12$

→ $6 + 12$

→ 18

Тук x се замества с $(9-3)$, но y не участва в дясната страна на равенството, определящо стойността на g , следователно аргументът $(g\ 34\ 3)$ не се оценява.

Така демонстрирахме едно от преимуществата на “мързеливото” оценяване: ***аргументи, които не са необходими, не се оценяват.***

Последният пример не е особено смислен, тъй като вторият аргумент не се използва в никакъв случай и по същество е излишен в дефиницията на g .

По-интересен е следният пример:

```
switch :: Int -> a -> a -> a
```

```
switch n x y
```

```
  | n > 0          = x
```

```
  | otherwise     = y
```

Ако цялото число n е положително, резултатът съвпада с оценката на x ; в противен случай тя съвпада с оценката на y . С други думи, винаги при оценяване на обръщение към функцията `switch` се оценяват аргументът n (т.е. първият аргумент) и точно един от останалите аргументи.

Нека сега функцията h е дефинирана както следва:

```
 $h :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ 
```

```
 $h\ x\ y = x + x$ 
```

Тогава

```
 $h\ (9-3)\ (h\ 34\ 3)$ 
```

```
 $\longrightarrow (9-3) + (9-3)$ 
```

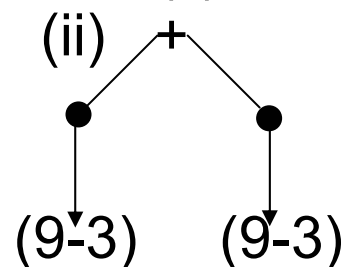
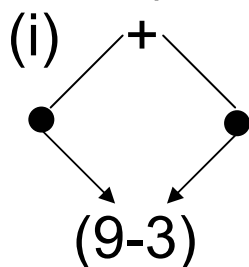
```
 $\longrightarrow 6 + 6$ 
```

```
 $\longrightarrow 12$ 
```

Изглежда, че в последния пример аргументът (9-3) се оценява двукратно, но принципите на “мързеливото” оценяване гарантират, че **дублираните аргументи (т.е. многократните включвания на аргументи) се оценяват по не повече от един път.**

В реализацията на Hugs това се постига, като изразите се представят чрез подходящи графи и пресмятанията се извършват върху тези графи. В такъв граф всеки аргумент се представя чрез единствен възел и към този възел може да сочат много дъги.

Например изразът, до който се свежда оценяването на $h\ (9-3)\ (h\ 34\ 3)$, се представя чрез (i), а не чрез (ii).



Като следващ пример ще проследим оценяването на обръщение към функцията

$$\text{rt} (x, y) = x + 1 :$$
$$\begin{aligned} &\text{rt} (3+2, 4-17) \\ &\longrightarrow (3+2) + 1 \\ &\longrightarrow 6 \end{aligned}$$

Тук се оценява само част от (първият елемент на) двойката, която е аргумент на обръщението към функцията `rt`. С други думи, аргументът се използва, но се оценява само реално необходимата негова част.

Правила за оценяване и “мързеливо” оценяване

Дефиницията на една функция се състои от поредица от условни равенства. Всяко условно равенство може да съдържа множество клаузи и може да включва в специална where клауза произволен брой локални дефиниции. Всяко равенство описва в лявата си страна различни случаи на действието на функцията, която е предмет на дефиницията, т.е. описва резултата, който се връща при прилагане на функцията към различни образци.

Общ вид на дефиниция на функция

$$\begin{array}{lcl}
 f \ p_1 \ p_2 \ \dots \ p_k & & \\
 | \ g_1 & = & e_1 \\
 | \ g_2 & = & e_2 \\
 & \dots & \\
 | \ \text{otherwise} & = & e_r
 \end{array}$$

where

$$v_1 \ a_{1,1} \quad = \ r_1$$

...

$$f \ q_1 \ q_2 \ \dots \ q_k$$

$$= \dots$$

.....

Оценяването на $f a_1 a_2 \dots a_k$ има три основни аспекта.

1. Съпоставане по образец

Аргументите се оценяват с цел да се установи кое от условните равенства е приложимо. При това оценяването на аргументите не се извършва изцяло, а само до степен, която е достатъчна, за да се прецени дали те са съпоставими със съответните образци.

Ако аргументите са съпоставими с образците p_1, p_2, \dots, p_k , то оценяването продължава с използване на първото равенство; в противен случай се прави проверка за съпоставимост на аргументите с образците от второто равенство и тази проверка от своя страна може да предизвика по-нататъшно оценяване на аргументите.

Този процес продължава, докато се намери множество от съпоставими с аргументите образци или докато се изчерпат условните равенства от дефиницията (тогава се получава **Program error**).

Нека например е дадена дефиницията:

```
f :: [Int] -> [Int] -> Int
```

```
f [] ys = 0 (f.1)
```

```
f (x:xs) [] = 0 (f.2)
```

```
f (x:xs) (y:ys) = x+y (f.3)
```

Тогава оценяването на израза `f [1..3] [1..3]` се извършва по следния начин:

f [1..3] [1..3]	(1)
→ f (1:[2..3]) [1..3]	(2)
→ f (1:[2..3]) (1:[2..3])	(3)
→ 1 + 1	(4)
→ 2	

На стъпка (1) няма достатъчно информация, за да може да се прецени дали има съпоставимост с (f.1). Следващата стъпка на оценяване води до (2) и показва, че няма съпоставимост с (f.1).

Първият аргумент от (2) е съпоставим с първия образец от (f.2), следователно трябва да се направи проверка за втория аргумент от (2) и втория образец от (f.2). Следващата стъпка на оценяването, извършена в (3), показва, че вторият аргумент не е съпоставим с втория образец от (f.2). На тази стъпка се вижда също, че е налице съпоставимост на аргументите с (f.3), следователно е налице (4).

2. Условия (guards)

Нека за определеност предположим, че първото условно равенство от дефиницията е съпоставимо с обръщението към f , което трябва да се оцени. Тогава образците p_1, p_2, \dots, p_k в условното равенство се заместват с изразите a_1, a_2, \dots, a_k . След това трябва да се определи коя от клаузите в дясната страна е приложима. За целта условията се оценяват последователно, докато се намери първото условие със стойност `True`; като резултат се връща стойността на съответната на това условие клауза.

Нека например е дадена дефиницията:

```
f :: Int -> Int -> Int -> Int
f m n p
  | m >= n && m >= p = m
  | n >= m && n >= p = n
  | otherwise       = p
```

Тогава

```
f (2+3) (4-1) (3+9)
?? (2+3) >= (4-1) && (2+3) >= (3+9)
?? → 5 >= 3 && 5 >= (3+9)
?? → True && 5 >= (3+9)
?? → 5 >= (3+9)
?? → 5 >= 12
?? → False
```

?? 3>=5 && 3>=12

?? → False && 3>=12

?? → False

?? otherwise True

→ 12

3. Локални дефиниции

Стойностите в клаузите `where` се пресмятат при необходимост (“при поискване”): пресмятането на дадена стойност започва едва когато се окаже, че тази стойност е необходима.

Ако са дадени дефинициите:

```
f :: Int -> Int -> Int
f m n
  | notNil xs = front xs
  | otherwise = n
where
  xs = [m .. n]
```

```
front (x:y:zs) = x+y
front [x]      = x
```

```
notNil []      = False
notNil (_:_)   = True
```

Тогава процесът на оценяване на `f 3 5` изглежда по следния начин:

```
f 3 5
?? notNil xs
?? | where
?? | xs = [3 .. 5]
?? |   → 3:[4 .. 5]
?? → notNil (3:[4 .. 5])
?? → True
```

(1)

```

→ front xs
  |
  | where
  |   xs = 3:[4 .. 5]
  |   → 3:4:[5]                                     (2)
→ 3+4                                                (3)
→ 7

```

За да може да се оцени условието `notNil xs`, започва оценяване на `xs` и след една стъпка (1) показва, че това условие има стойност `True`.

Оценяването на `front xs` изисква повече информация за `xs`, затова оценяването на `xs` се извършва на (продължава с) още една стъпка и така се получава (2). Успешното съпоставяне по образец в дефиницията на `front` в този случай дава (3) и така се получава окончателният резултат.

Ред на оценяването

Това, което характеризира оценяването в Haskell, освен обстоятелството, че аргументите се оценяват по не повече от един път, е ***редът, в който се прилагат функции***, когато има възможност за избор.

- Оценяването се извършва в посока от външните към вътрешните изрази.

В ситуации от типа на

$$\underline{f_1 \ e_1 \ (f_2 \ e_2 \ 10)} \ ,$$

където едно прилагане на функция включва друго, външното обръщение $f_1 \ e_1 \ (f_2 \ e_2 \ 10)$ се избира за оценяване.

- В останалите случаи оценяването се извършва в посока от ляво на дясно.

В израза

$$\underline{f_1 e_1} + \underline{f_2 e_2}$$

трябва да бъдат оценени и двата подчертани израза. При това най-напред се оценява левият израз $f_1 e_1$.

Работа с безкрайни списъци в Haskell

Едно важно следствие от “мързеливото” оценяване в Haskell е обстоятелството, че езикът позволява да се работи с **безкрайни структури**. Пълното оценяване на такава структура по принцип изисква безкрайно време, т.е. не може да завърши, но механизмът на “мързеливото” оценяване позволява да бъдат оценявани само тези части (“порции”) на безкрайните структури, които са реално необходими.

Най-прост пример за безкраен списък: безкраен списък от еднакви елементи, например безкраен списък от единици.

```
ones :: [Int]  
ones = 1 : ones
```

Оценяването на `ones` ще продължи безкрайно дълго и следователно ще трябва да бъде прекъснато от потребителя.

Възможно е обаче съвсем коректно да бъдат оценени обръщения към функции с аргумент `ones`.

Пример

```
addFirstTwo :: [Int] -> Int  
addFirstTwo (x:y:zs) = x+y
```

Тогaвa

```
addFirstTwo ones
```

```
→ addFirstTwo (1:ones)
```

```
→ addFirstTwo (1:1:ones)
```

```
→ 1+1
```

```
→ 2
```

Вградени за Haskell са списъците от вида $[n \dots]$ и $[n, m \dots]$.

Например

$[3 \dots] = [3, 4, 5, 6, \dots]$

$[3, 5 \dots] = [3, 5, 7, 9, \dots]$

Примерни дефиниции на функции – генератори на горните списъци:

```
from :: Int -> [Int]
from n = n : from (n+1)
```

```
fromStep :: Int -> Int -> [Int]
fromStep n m = n : fromStep (n+m) m
```

Тогава

`fromStep 3 2`

`→ 3 : fromStep 5 2`

`→ 3 : 5 : fromStep 7 2`

`→ ...`

Безкрайни списъци могат да се дефинират и чрез определяне на обхвата им (чрез list comprehension). Например списък от всички Питагорови тройки може да бъде генериран чрез избор на стойност на z от $[2 \dots]$, следван от избор на подходящи стойности на x и y , по-малки от тази на z .

```
pythagTriples :: [(Int,Int,Int)]
pythagTriples =
    [ (x,y,z) | z <- [2 .. ], y <- [2 .. z-1],
              x <- [2 .. y-1], x*x + y*y == z*z ]
```

Така pythagTriples = [(3,4,5),(6,8,10),(5,12,13),(9,12,15),
(8,15,17),(12,16,20), ...]

Забележка. Предложената дефиниция на `pythagTriples` е коректна. Не е коректна обаче следната дефиниция:

```
pythagTriples2 :: [(Int,Int,Int)]
pythagTriples2 =
    [ (x,y,z) | x <- [2 .. ],
                y <- [x+1 .. ],
                z <- [y+1 .. ],
                x*x + y*y == z*z ]
```

Последната дефиниция не произвежда резултат, защото редът на избор на стойности на елементите на тройките е неподходящ. Първата избрана стойност за `x` е 2, за `y` е 3 и при тези фиксирани стойности на `x` и `y` следват безброй много неуспешни опити за избор на стойност на `z`.

Като по-сложен пример ще разгледаме реализация на решето на Ератостен, което представлява генератор на безкраен списък от простите числа.

```
primes :: [Int]
primes = sieve [2 .. ]
```

```
sieve :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs,
                               y `mod` x > 0 ]
```

Toraba

primes

```
→ sieve [2 .. ]
→ 2 : sieve [ y | y <- [3 .. ], y `mod` 2 > 0]
→ 2 : sieve (3 : [ y | y <- [4 .. ],
                        y `mod` 2 > 0])
→ 2 : 3 : sieve [ z | z <- [ y | y <- [4 .. ],
                              y `mod` 2 > 0],
                  z `mod` 3 > 0]
→ ...
→ 2 : 3 : sieve [ z | z <- [5,7,9, ... ],
                  z `mod` 3 > 0]
→ ...
→ 2 : 3 : sieve [5,7,11, ... ]
→ ...
```

Можем ли да използваме `primes` за проверка дали дадено число е просто?

Нека `member` е функцията за проверка на принадлежност към списък, дефинирана както следва:

```
member :: Eq a => [a] -> a -> Bool
member []      x = False
member (y:ys) x
  | x==y        = True
  | otherwise   = member ys x
```

Ако оценим `member primes 7`, се получава резултат `True`, но `member primes 6` не дава резултат. Причината отново е в това, че трябва да се проверят безброй много елементи на `primes` преди да се направи заключение, че 6 не е елемент на този списък.

Проблемът може да се реши с отчитане на факта, че списъкът `primes` е нареден. За целта може да се дефинира нова функция, която проверява дали вторият ѝ аргумент се съдържа в наредения списък, който е неин първи аргумент:

```
memberOrd :: Ord a => [a] -> a -> Bool
memberOrd (x:xs) n
  | x < n      = memberOrd xs n
  | x == n     = True
  | otherwise  = False
```

Забележки

1. Записът “Eq a => “ в декларацията на типа на member означава изискването типът a да бъде подклас на класа Eq, в който е дефинирана операцията “еквивалентност” (проверката за равенство ==).
2. Записът “Ord a => “ в декларацията на типа на memberOrd означава изискването типът a да бъде нареден (а да бъде подклас на класа на наредените типове Ord), т.е. за него да са дефинирани операциите за сравнение >, >=, <, <= и операцията за проверка на равенство ==.

Лекция 11

**Оценяване на изрази.
Работа с безкрайни списъци в Haskell**

“Мързеливо” оценяване (lazy evaluation)

“Мързеливото” оценяване (lazy evaluation) е стратегия на оценяване, която по стандарт стои в основата на работата на всички интерпретатори на Haskell. Същността на тази стратегия е, че интерпретаторът оценява даден аргумент на дадена функция само ако (и доколкото) стойността на този аргумент е необходима за пресмятането на целия резултат. Нещо повече, ако даден аргумент е съставен (например е вектор или списък), то се оценяват само тези негови компоненти, чиито стойности са необходими от гледна точка на получаването на резултата. При това дублиращите се подизрази се оценяват по не повече от един път.

Най-важно от гледна точка на оценяването на изрази в Haskell е *прилагането на функции*. Основната идея тук е еквивалентна на т. нар. **оценяване чрез заместване** в “традиционните” езици за функционално програмиране.

Оценяването на израз, който е обръщение към функцията f с аргументи a_1, a_2, \dots, a_k , се състои в заместване на формалните параметри от дефиницията на f съответно с изразите a_1, a_2, \dots, a_k и оценяване на така получения частен случай на тялото на дефиницията.

Например, ако

$f\ x\ y = x+y$, то

$f\ (9-3)\ (f\ 34\ 3)$

$\longrightarrow (9-3) + (f\ 34\ 3)$

При това изразите $(9-3)$ и $(f\ 34\ 3)$ не се оценяват преди да бъдат предадени като аргументи на f .

Доколкото операцията събиране изисква аргументите ѝ да бъдат оценени, оценяването на горния израз продължава по следния начин:

$f(9-3)(f\ 34\ 3)$

→ ...

→ $6 + (34 + 3)$

→ $6 + 37$

→ 43

В конкретния случай бяха оценени и двата аргумента (фактически параметъра), но това не винаги е така.

Ако например дефинираме

$g\ x\ y = x + 12$, то

$g(9-3)(g\ 34\ 3)$

→ $(9-3) + 12$

→ $6 + 12$

→ 18

Тук x се замества с $(9-3)$, но y не участва в дясната страна на равенството, определящо стойността на g , следователно аргументът $(g \ 34 \ 3)$ не се оценява.

Така демонстрирахме едно от преимуществата на “мързеливото” оценяване: ***аргументи, които не са необходими, не се оценяват.***

Последният пример не е особено смислен, тъй като вторият аргумент не се използва в никакъв случай и по същество е излишен в дефиницията на g .

По-интересен е следният пример:

```
switch :: Int -> a -> a -> a
```

```
switch n x y
```

```
  | n > 0          = x
```

```
  | otherwise     = y
```

Ако цялото число n е положително, резултатът съвпада с оценката на x ; в противен случай тя съвпада с оценката на y . С други думи, винаги при оценяване на обръщение към функцията `switch` се оценяват аргументът n (т.е. първият аргумент) и точно един от останалите аргументи.

Нека сега функцията h е дефинирана както следва:

```
 $h :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ 
```

```
 $h\ x\ y = x + x$ 
```

Тогава

```
 $h\ (9-3)\ (h\ 34\ 3)$ 
```

```
 $\longrightarrow (9-3) + (9-3)$ 
```

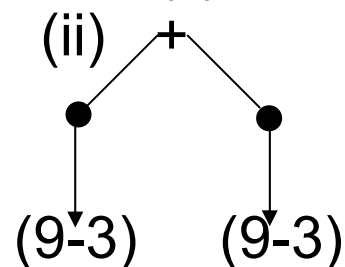
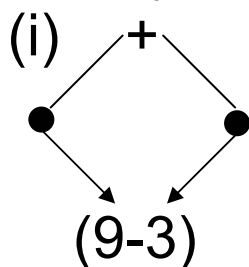
```
 $\longrightarrow 6 + 6$ 
```

```
 $\longrightarrow 12$ 
```

Изглежда, че в последния пример аргументът (9-3) се оценява двукратно, но принципите на “мързеливото” оценяване гарантират, че **дублираните аргументи (т.е. многократните включвания на аргументи) се оценяват по не повече от един път.**

В реализацията на Hugs това се постига, като изразите се представят чрез подходящи графи и пресмятанията се извършват върху тези графи. В такъв граф всеки аргумент се представя чрез единствен възел и към този възел може да сочат много дъги.

Например изразът, до който се свежда оценяването на $h\ (9-3)\ (h\ 34\ 3)$, се представя чрез (i), а не чрез (ii).



Като следващ пример ще проследим оценяването на обръщение към функцията

$$\text{rt} (x, y) = x + 1 :$$
$$\begin{aligned} &\text{rt} (3+2, 4-17) \\ &\longrightarrow (3+2) + 1 \\ &\longrightarrow 6 \end{aligned}$$

Тук се оценява само част от (първият елемент на) двойката, която е аргумент на обръщението към функцията `rt`. С други думи, аргументът се използва, но се оценява само реално необходимата негова част.

Правила за оценяване и “мързеливо” оценяване

Дефиницията на една функция се състои от поредица от условни равенства. Всяко условно равенство може да съдържа множество клаузи и може да включва в специална where клауза произволен брой локални дефиниции. Всяко равенство описва в лявата си страна различни случаи на действието на функцията, която е предмет на дефиницията, т.е. описва резултата, който се връща при прилагане на функцията към различни образци.

Общ вид на дефиниция на функция

$$\begin{array}{lcl}
 f \ p_1 \ p_2 \ \dots \ p_k & & \\
 | \ g_1 & = & e_1 \\
 | \ g_2 & = & e_2 \\
 & \dots & \\
 | \ \text{otherwise} & = & e_r
 \end{array}$$

where

$$v_1 \ a_{1,1} \quad = \ r_1$$

...

$$f \ q_1 \ q_2 \ \dots \ q_k$$

$$= \dots$$

.....

Оценяването на $f a_1 a_2 \dots a_k$ има три основни аспекта.

1. Съпоставане по образец

Аргументите се оценяват с цел да се установи кое от условните равенства е приложимо. При това оценяването на аргументите не се извършва изцяло, а само до степен, която е достатъчна, за да се прецени дали те са съпоставими със съответните образци.

Ако аргументите са съпоставими с образците p_1, p_2, \dots, p_k , то оценяването продължава с използване на първото равенство; в противен случай се прави проверка за съпоставимост на аргументите с образците от второто равенство и тази проверка от своя страна може да предизвика по-нататъшно оценяване на аргументите.

Този процес продължава, докато се намери множество от съпоставими с аргументите образци или докато се изчерпат условните равенства от дефиницията (тогава се получава **Program error**).

Нека например е дадена дефиницията:

```
f :: [Int] -> [Int] -> Int
f []      ys      = 0                      (f.1)
f (x:xs) []      = 0                      (f.2)
f (x:xs) (y:ys) = x+y                    (f.3)
```

Тогава оценяването на израза `f [1..3] [1..3]` се извършва по следния начин:

f [1..3] [1..3]	(1)
→ f (1:[2..3]) [1..3]	(2)
→ f (1:[2..3]) (1:[2..3])	(3)
→ 1 + 1	(4)
→ 2	

На стъпка (1) няма достатъчно информация, за да може да се прецени дали има съпоставимост с (f.1). Следващата стъпка на оценяване води до (2) и показва, че няма съпоставимост с (f.1).

Първият аргумент от (2) е съпоставим с първия образец от (f.2), следователно трябва да се направи проверка за втория аргумент от (2) и втория образец от (f.2). Следващата стъпка на оценяването, извършена в (3), показва, че вторият аргумент не е съпоставим с втория образец от (f.2). На тази стъпка се вижда също, че е налице съпоставимост на аргументите с (f.3), следователно е налице (4).

2. Условия (guards)

Нека за определеност предположим, че първото условно равенство от дефиницията е съпоставимо с обръщението към f , което трябва да се оцени. Тогава образците p_1, p_2, \dots, p_k в условното равенство се заместват с изразите a_1, a_2, \dots, a_k . След това трябва да се определи коя от клаузите в дясната страна е приложима. За целта условията се оценяват последователно, докато се намери първото условие със стойност `True`; като резултат се връща стойността на съответната на това условие клауза.

Нека например е дадена дефиницията:

```
f :: Int -> Int -> Int -> Int
f m n p
  | m>=n && m>=p = m
  | n>=m && n>=p = n
  | otherwise   = p
```

Тогава

```
f (2+3) (4-1) (3+9)
?? (2+3)>=(4-1) && (2+3)>=(3+9)
?? → 5>=3 && 5>=(3+9)
?? → True && 5>=(3+9)
?? → 5>=(3+9)
?? → 5>=12
?? → False
```

```
?? 3>=5 && 3>=12
?? → False && 3>=12
?? → False
?? otherwise    True
→ 12
```

3. Локални дефиниции

Стойностите в клаузите `where` се пресмятат при необходимост (“при поискване”): пресмятането на дадена стойност започва едва когато се окаже, че тази стойност е необходима.

Ако са дадени дефинициите:

```
f :: Int -> Int -> Int
f m n
  | notNil xs = front xs
  | otherwise = n
where
  xs = [m .. n]
```



```
front (x:y:zs) = x+y
front [x]       = x
```

```
notNil []      = False
notNil (_:_)   = True
```

Тогава процесът на оценяване на `f 3 5` изглежда по следния начин:

```
f 3 5
?? notNil xs
?? | where
?? | xs = [3 .. 5]
?? |   → 3:[4 .. 5]
?? → notNil (3:[4 .. 5])
?? → True
```

(1)

```

→ front xs
  |
  | where
  |   xs = 3:[4 .. 5]
  |   → 3:4:[5]                                     (2)
→ 3+4                                                (3)
→ 7

```

За да може да се оцени условието `notNil xs`, започва оценяване на `xs` и след една стъпка (1) показва, че това условие има стойност `True`.

Оценяването на `front xs` изисква повече информация за `xs`, затова оценяването на `xs` се извършва на (продължава с) още една стъпка и така се получава (2). Успешното съпоставяне по образец в дефиницията на `front` в този случай дава (3) и така се получава окончателният резултат.

Ред на оценяването

Това, което характеризира оценяването в Haskell, освен обстоятелството, че аргументите се оценяват по не повече от един път, е ***редът, в който се прилагат функции***, когато има възможност за избор.

- Оценяването се извършва в посока от външните към вътрешните изрази.

В ситуации от типа на

$$\underline{f_1 \ e_1 \ (f_2 \ e_2 \ 10)} \ ,$$

където едно прилагане на функция включва друго, външното обръщение $f_1 \ e_1 \ (f_2 \ e_2 \ 10)$ се избира за оценяване.

- В останалите случаи оценяването се извършва в посока от ляво на дясно.

В израза

$$\underline{f_1 e_1} + \underline{f_2 e_2}$$

трябва да бъдат оценени и двата подчертани израза. При това най-напред се оценява левият израз $f_1 e_1$.

Работа с безкрайни списъци в Haskell

Едно важно следствие от “мързеливото” оценяване в Haskell е обстоятелството, че езикът позволява да се работи с **безкрайни структури**. Пълното оценяване на такава структура по принцип изисква безкрайно време, т.е. не може да завърши, но механизмът на “мързеливото” оценяване позволява да бъдат оценявани само тези части (“порции”) на безкрайните структури, които са реално необходими.

Най-прост пример за безкраен списък: безкраен списък от еднакви елементи, например безкраен списък от единици.

```
ones :: [Int]  
ones = 1 : ones
```

Оценяването на `ones` ще продължи безкрайно дълго и следователно ще трябва да бъде прекъснато от потребителя.

Възможно е обаче съвсем коректно да бъдат оценени обръщения към функции с аргумент `ones`.

Пример

```
addFirstTwo :: [Int] -> Int  
addFirstTwo (x:y:zs) = x+y
```

Тогда

```
addFirstTwo ones  
→ addFirstTwo (1:ones)  
→ addFirstTwo (1:1:ones)  
→ 1+1  
→ 2
```

Вградени за Haskell са списъците от вида $[n \dots]$ и $[n, m \dots]$.

Например

$[3 \dots] = [3, 4, 5, 6, \dots]$

$[3, 5 \dots] = [3, 5, 7, 9, \dots]$

Примерни дефиниции на функции – генератори на горните списъци:

```
from :: Int -> [Int]
from n = n : from (n+1)
```

```
fromStep :: Int -> Int -> [Int]
fromStep n m = n : fromStep (n+m) m
```


Тогава

`fromStep 3 2`

`→ 3 : fromStep 5 2`

`→ 3 : 5 : fromStep 7 2`

`→ ...`

Безкрайни списъци могат да се дефинират и чрез определяне на обхвата им (чрез list comprehension). Например списък от всички Питагорови тройки може да бъде генериран чрез избор на стойност на z от $[2 \dots]$, следван от избор на подходящи стойности на x и y , по-малки от тази на z .

```
pythagTriples :: [(Int,Int,Int)]
pythagTriples =
    [ (x,y,z) | z <- [2 .. ], y <- [2 .. z-1],
              x <- [2 .. y-1], x*x + y*y == z*z ]
```

Така pythagTriples = [(3,4,5),(6,8,10),(5,12,13),(9,12,15),
(8,15,17),(12,16,20), ...]

Забележка. Предложената дефиниция на `pythagTriples` е коректна. Не е коректна обаче следната дефиниция:

```
pythagTriples2 :: [(Int,Int,Int)]
pythagTriples2 =
    [ (x,y,z) | x <- [2 .. ],
                y <- [x+1 .. ],
                z <- [y+1 .. ],
                x*x + y*y == z*z ]
```

Последната дефиниция не произвежда резултат, защото редът на избор на стойности на елементите на тройките е неподходящ. Първата избрана стойност за `x` е 2, за `y` е 3 и при тези фиксирани стойности на `x` и `y` следват безброй много неуспешни опити за избор на стойност на `z`.

Като по-сложен пример ще разгледаме реализация на решето на Ератостен, което представлява генератор на безкраен списък от простите числа.

```
primes :: [Int]
primes = sieve [2 .. ]
```

```
sieve :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs,
                               y `mod` x > 0 ]
```

Toraba

primes

```
→ sieve [2 .. ]
→ 2 : sieve [ y | y <- [3 .. ], y `mod` 2 > 0]
→ 2 : sieve (3 : [ y | y <- [4 .. ],
                        y `mod` 2 > 0])
→ 2 : 3 : sieve [ z | z <- [ y | y <- [4 .. ],
                              y `mod` 2 > 0],
                  z `mod` 3 > 0]
→ ...
→ 2 : 3 : sieve [ z | z <- [5,7,9, ... ],
                  z `mod` 3 > 0]
→ ...
→ 2 : 3 : sieve [5,7,11, ... ]
→ ...
```

Можем ли да използваме `primes` за проверка дали дадено число е просто?

Нека `member` е функцията за проверка на принадлежност към списък, дефинирана както следва:

```
member :: Eq a => [a] -> a -> Bool
member []      x = False
member (y:ys) x
  | x==y        = True
  | otherwise   = member ys x
```

Ако оценим `member primes 7`, се получава резултат `True`, но `member primes 6` не дава резултат. Причината отново е в това, че трябва да се проверят безброй много елементи на `primes` преди да се направи заключение, че 6 не е елемент на този списък.

Проблемът може да се реши с отчитане на факта, че списъкът `primes` е нареден. За целта може да се дефинира нова функция, която проверява дали вторият ѝ аргумент се съдържа в наредения списък, който е неин първи аргумент:

```
memberOrd :: Ord a => [a] -> a -> Bool
memberOrd (x:xs) n
  | x < n      = memberOrd xs n
  | x == n     = True
  | otherwise  = False
```

Забележки

1. Записът “Eq a => “ в декларацията на типа на member означава изискването типът a да бъде екземпляр на класа Eq, в който е дефинирана операцията “еквивалентност” (проверката за равенство ==).
2. Записът “Ord a => “ в декларацията на типа на memberOrd означава изискването типът a да бъде нареден (а да бъде екземпляр на класа на наредените типове Ord), т.е. за него да са дефинирани операциите за сравнение >, >=, <, <= и операцията за проверка на равенство ==.

Лекция 12

**Типове и класове в Haskell.
Дефиниране на нови типове**

Генерични и полиморфни функции. Класове от/на типове

Досега разгледахме два типа функции, които могат да работят с данни от повече от един тип.

Една **полиморфна функция**, например `length` (намиране на дължина на списък, чийто елементи могат да бъдат от произволен тип), има единствена дефиниция, която работи върху всички нейни типове.

Генеричните функции, например == (проверка за равенство), + (събиране на числа от един и същ тип) и show (конвертиране на число, булева стойност и др. в низ), могат да бъдат прилагани към данни от много типове, но за различните типове в действителност се използват различни дефиниции (различни **методи** на генеричната функция).

Генерични функции и `overloading` (пренатоварване; додефиниране)

Аритметичният оператор `+` предизвиква пресмятане на сумата на произволни две числа от един и същ тип. Например, този оператор може да бъде използван за пресмятане на сумата на две цели числа и тогава резултатът ще бъде също цяло число. При събирането на две реални числа (числа с плаваща точка) се получава друго реално число и т.н.

Примери

`> 1 + 2`

`3`

`> 1.1 + 2.2`

`3.3`

С други думи, операторът + може да бъде прилаган към аргументи от всеки числов тип и резултатът от прилагането му ще бъде от същия тип.

Тази идея може да бъде прецизирана с използване на т. нар. *ограничение върху класа (class constraint)* при дефинирането на типа на оператора +:

$$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$

Горното ограничение означава, че за всеки тип a , който е **екземпляр** на класа Num на/от числови(те) типове, функцията $(+)$ е от тип $a \rightarrow a \rightarrow a$.

Всеки тип, който включва едно или повече ограничения върху класа, се нарича ***overloaded*** (*пренатоварен; додефиниран*). Следователно, $\text{Num } a \Rightarrow a \rightarrow a \rightarrow a$ е ***overloaded*** (додефиниран) тип и $(+)$ е ***генерична*** (додефинирана) функция.

В действителност повечето от аритметичните функции, дефинирани в стандартния прелюд на Haskell, са генерични.

Например:

```
(-)      :: Num a => a -> a -> a
(*)      :: Num a => a -> a -> a
negate   :: Num a => a -> a
abs      :: Num a => a -> a
signum   :: Num a => a -> a
```

Нещо повече, числата в Haskell също са додефинирани. Например, **3 :: Num a => a** означава, че числото 3 е от всеки числов тип *a* (за всеки числов тип *a* числото 3 има тип *a*).

Общи сведения за класовете в Haskell

Най-общо, понятието **клас** в езика Haskell се определя като колекция от типове, за които се поддържа множество додефинирани операции, наречени **методи**.

Например, за функцията `elem`, която е “вградена” в Haskell, може да се предположи, че е от тип

```
elem :: a -> [a] -> Bool
```

Това обаче ще бъде вярно само за такива типове `a`, за които е дефинирана операцията (функцията) за проверка за равенство `==`.

Следователно, би било полезно да се разполага със средства, които позволяват да се зададат експлицитно определени ограничения върху даден тип, описан с помощта на типова променлива (type variable).

Множеството (колекцията) от типове, за които са дефинирани съответно множество от функции, се нарича **клас от/на типове** (***type class***) или накратко **клас**.

Например, множеството от типове, за които е дефинирана функцията за проверка на равенство (`==`), се означава като клас `Eq`.

Дефиниране на класа Eq

За да може да се дефинира един клас, е необходимо да се избере (зададе) неговото име и да се опишат ограниченията, които трябва да удовлетворява даден тип a , за да принадлежи на този клас.

Типовете, които принадлежат на даден клас, се наричат **екземпляри** на този клас.

Най-важно (определящо) за класа Eq е наличието на функцията $==$ от тип $a \rightarrow a \rightarrow \text{Bool}$, която проверява дали два елемента на даден клас a , който е екземпляр на Eq, са равни:

```
class Eq a where  
  (==) :: a -> a -> Bool
```

Примерни дефиниции на функции върху екземпляри на Eq

```
allEqual :: Eq a => a -> a -> a -> Bool  
allEqual m n p = (m==n) && (n==p)
```

```
elem :: Eq a => a -> [a] -> Bool  
elem _ [] = False  
elem x (y:ys) = (x == y) || (elem x ys)
```

Signatures („подписи”; функции, характеризиращи даден клас)

Както показвахме по-горе, дефиницията на даден клас включва декларация от вида

```
class Visible a where  
  toString :: a -> String  
  size    :: a -> Int
```

Декларацията включва името на класа (Visible) и т. нар. **signature** („подпис”) на класа, т.е. списък от имената и типовете на функциите, които еднозначно определят (характеризират) класа – това са функциите, които следва задължително да бъдат дефинирани за всички типове, които са екземпляри на този клас.

Следователно, дефиницията на тип има следния общ вид:

```
class Name ty where  
... signature involving the type variable ty ...
```

Дефиниране на екземпляри на клас

Един тип се определя като екземпляр на даден клас, като се дефинират функциите – „подписи” на класа за елементите на този тип.

Например дефиницията

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _      == _     = False
```

определя вградения тип Bool като екземпляр на класа Eq.

Примери за дефиниции на екземпляри на класа Visible:

```
instance Visible Char where
  toString ch  = [ch]
  size _       = 1
```

```
instance Visible Bool where
  toString True  = "True"
  toString False = "False"
  size _         = 1
```

-- Дефиниция на екземпляр на клас с контекст.

```
instance Visible a => Visible [a] where
  toString = concat . map toString
  size      = foldr (+) 1 . map size
```

Горната дефиниция използва *контекста* `Visible a`, за да означава факта, че видими са всички списъци от обекти, които сами по себе си са видими. В десните страни на равенствата, дефиниращи функциите `toString` и `size` за типа `[a]`, са използвани едноименните функции, които действат върху типа `a`.

Дефиниции по подразбиране

Нека отново се върнем към дефиницията на класа Eq. В Haskell този клас е дефиниран по следния начин:

```
class Eq a where
  (==) , (/=) :: a -> a -> Bool
  x /= y      = not (x==y)
  x == y      = not (x/=y)
```

Към операцията за сравнение с цел проверка на равенство се добавя и проверката за неравенство (различие). Освен това са включени и **дефиниции по подразбиране** на /= чрез == и на == чрез /.

Тези дефиниции са валидни по подразбиране за всички екземпляри на класа E_q , но ако за даден тип, който е екземпляр на E_q , някоя от функциите $==$ или $/=$ има конкретна дефиниция, тази дефиниция има приоритет над (припокрива, overrides) дефиницията по подразбиране.

Нещо повече, за всеки екземпляр на класа E_q е необходимо поне едната от двете функции $==$ и $/=$ да има конкретна дефиниция (иначе двойката дефиниции по подразбиране е неизползваема, тъй като ще генерира бездънна рекурсия). Ако е дефинирана конкретно само едната от двете функции (или $==$, или $/=$), то тази дефиниция би била достатъчна и за другата функция, тъй като дефиницията по подразбиране определя връзката между двете функции.

Производни класове

Езикът Haskell позволява да бъдат дефинирани класове, които са подкласове (производни класове, *derived classes*) на други класове.

Най-прост пример в това отношение е класът на наредените типове, *Ord*. За да бъде нареден, един тип трябва да поддържа операциите за сравнение `==`, `/==`, `>`, `>=`, `<`, `<=`. Наличието на първите две операции означава, че наредените типове образуват подмножество на класа *Eq*, т.е. те образуват **подклас** (**производен клас**, ***derived class***) на класа *Eq*.

Това обстоятелство се записва формално по следния начин:

```
class Eq a => Ord a where
  (<) , (<=) , (>) , (>=) :: a -> a -> Bool
  max, min                :: a -> a -> a
  compare                 :: a -> a -> Ordering
```

От друга страна може да се каже, че класът Ord **наследява** операциите на Eq. **Наследяването** (*inheritance*) е една от централните идеи на **обектно ориентираното програмиране**.

Множествени ограничения и множествено наследяване

В една от предишните лекции в курса дефинирахме функция с име `iSort`, която сортираше даден списък от цели числа, като за целта използваше метода за сортиране чрез вмъкване. В действителност тази функция има по-общ тип:

$$\text{iSort} :: \text{Ord } a \Rightarrow [a] \rightarrow [a]$$

Да предположим, че задачата ни е да сортираме даден списък и след това да визуализираме резултата като символен низ. Тогава функцията, която следва да дефинираме, ще бъде от тип

$$\text{vSort} :: (\text{Ord } a, \text{Visible } a) \Rightarrow [a] \rightarrow \text{String}$$

Подобни **множествени ограничения** могат да се появят и в дефиницията на екземпляр, например

```
instance (Eq a, Eq b) => Eq (a,b) where  
  (x,y) == (z,w)    =  x==z && y==w
```

Възможно е също множествени ограничения да бъдат включени в дефиницията на клас, например

```
class (Ord a, Visible a) => OrdVis a
```

В случаите, когато даден клас се дефинира на базата на два или повече класа, се казва, че е налице **множествено наследяване** (*multiple inheritance*).

Кратки сведения за вградените класове в Haskell

Haskell поддържа голям брой базови (вградени) класове, част от които ще представим накратко в настоящата лекция.

Eq – клас на типовете, за които са дефинирани операциите за проверка на равенство и неравенство

Този клас включва типове, чиито стойности могат да бъдат сравнявани за равенство и неравенство (различие), като за целта се използват следните методи:

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

$(/=) :: a \rightarrow a \rightarrow \text{Bool}$

Разгледаните досега типове `Bool`, `Char`, `String`, `Int`, `Integer` и `Float` са **екземпляри** на класа `Eq`. Такива са също и типовете, обхващащи списъци и вектори, чиито елементи са от тип, който е екземпляр на `Eq`.

Ще припомним, че класът Eq е дефиниран както следва:

```
class Eq a where
  (==) , (/=) :: a -> a -> Bool
  x /= y = not (x==y)
  x == y = not (x/=y)
```

Следва да отбележим изрично, че функционалните типове в общия случай не са екземпляри на класа Eq, тъй като няма подходящ механизъм за сравняване на две функции с цел установяване на равенство или неравенство (различие) между тях.

Ord – клас на наредените типове

Този клас включва типове, които са екземпляри на класа *Eq*, между елементите на които съществува (е дефинирана) линейна наредба, следователно техните елементи могат да бъдат сравнявани посредством следните методи:

```
(<)  :: a -> a -> Bool  
(<=) :: a -> a -> Bool  
(<)  :: a -> a -> Bool  
(<=) :: a -> a -> Bool  
min  :: a -> a -> Bool  
max  :: a -> a -> Bool  
compare :: a -> a -> Ordering
```

Типът Ordering включва стойностите LT, EQ и GT, които представят трите възможни резултата от сравняването на два елемента на даден нареден тип.

Тогава функцията compare може да бъде дефинирана например по следния начин:

```
compare x y
| x==y      = EQ
| x<=y      = LT
| otherwise = GT
```

Ползата от такава функция е, че тя помага с помощта на една проверка (с едно обръщение към нея) да се определи точната релация между произволни два елемента на наредения тип, докато използването на оператори, които връщат булев резултат, би изисквало две сравнения.

Дефинициите по подразбиране на операторите за сравнение също използват стойностите от `compare`:

`x <= y = compare x y /= GT`

`x < y = compare x y == LT`

`x >= y = compare x y /= LT`

`x > y = compare x y == GT`

Дефинициите по подразбиране на функциите `max` и `min` изглеждат по следния начин:

```
max x y
  | x >= y      = x
  | otherwise = y
```

```
min x y
  | x <= y      = x
  | otherwise = y
```

Enum – клас на изброимите типове

Дефиницията на този тип изглежда по следния начин:

```
class (Ord a) => Enum a where
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a] -- [n .. ]
  enumFromThen :: a -> a -> [a] -- [n,m .. ]
  enumFromTo :: a -> a -> [a] -- [n .. m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,n' .. m]
```

Тук са включени също подходящи дефиниции по подразбиране на функциите `enumFromTo` и `enumFromThenTo`.

“Подписът” на класа включва също функциите `fromEnum` и `toEnum`, които преобразуват стойности между съответния тип и `Int`.

В случая на типа `Char` тези функции са известни също като `ord` и `chr`:

```
ord :: Char -> Int  
ord = fromEnum
```

```
chr :: Int -> Char  
chr = toEnum
```

Bounded – клас на ограничените типове

Този клас се специфицира посредством декларацията

```
class Bounded a where  
  minBound, maxBound :: a
```

Тук `minBound` и `maxBound` определят най-малката и най-голямата допустима стойност на елементите на съответния тип.

Show – клас на “видимите” типове
(типовете, чиито елементи могат да се преобразуват
в символни низове)

В стандартния прелюд на Haskell е дефиниран класът *Show*, който съдържа всички типове, чиито елементи могат да се преобразуват в символни низове (и в този смисъл могат да бъдат визуализирани, т.е. по принцип са “видими”).

Дефиницията на класа Show изглежда по следния начин:

```
type ShowS = String -> String
```

```
class Show a where  
  showsPrec :: Int -> a -> ShowS  
  show      :: a -> String  
  showList  :: [a] -> ShowS
```

Функцията `showsPrec` е предназначена за гъвкаво и ефективно преобразуване на “дълги” стойности; като начало е достатъчна функцията

```
show :: a -> String ,
```

която реализира конвертирането (преобразуването) в символен низ.

Read – клас на типовете, чиито стойности могат да бъдат четени от низове

Класът *Read* съдържа типове, чиито стойности могат да бъдат четени от символни низове. Като начало за използването на този клас е достатъчно да се познава функцията

```
read :: (Read a) => String -> a
```

Резултатът от изпълнението на тази функция може да не бъде добре дефиниран: необходимо е “входният” низ да включва точно един обект от съответния тип.

Освен това, в много случаи е от съществено значение типът на резултата от изпълнението на `read` да бъде точно специфициран, тъй като този резултат потенциално би могъл да бъде от различни типове.

Например може да се запише

```
(read " 1 " ) :: Int
```

с цел да се посочи явно, че резултатът трябва да бъде от тип `Int`.

Алгебрични типове

Общи сведения за алгебричните типове

Дефиницията на един алгебричен тип започва с ключовата дума `data`, след която се записват името на типа, знак за равенство и **конструкторите** на типа. Името на типа и имената на конструкторите задължително започват с главни букви.

Пример

```
data Day = Monday | Tuesday | Wednesday | Thursday  
         | Friday | Saturday | Sunday
```

Изброени типове

Най-простата разновидност на алгебричен тип се дефинира чрез изброяване на елементите на типа, както беше направено в последния пример.

Следват още примери за изброени типове:

```
data Temp = Cold | Hot
```

```
data Season = Spring | Summer | Autumn | Winter
```

Дефинирането на функции върху такива типове се извършва с помощта на стандартните техники, например с използване на подходящи образци:

```
weather :: Season -> Temp
```

```
weather Summer = Hot
```

```
weather _      = Cold
```

Производни типове

Вместо използването на вектори можем да дефинираме тип с определен брой компоненти като алгебричен тип. Такива типове често се наричат ***производни типове*** (***резултатни типове***; ***product types***).

Пример

```
data People = Person Name Age
```

Тук Name е синоним на String, а Age е синоним на Int:

```
type Name = String
```

```
type Age   = Int
```

Горната дефиниция на People може да бъде интерпретирана както следва:

За да се конструира елемент на типа People, е необходимо да се предвидят (дадат като аргументи) две стойности: едната (нека я наречем st) от тип Name, а другата (нека я наречем n) – от тип Age.

Елементът на People, конструиран по този начин, ще има вида Person st n.

Примери за стойности от тип People:

Person “Aunt Jemima” 77

Person “Ronnie” 14

Алтернативи

Геометричните фигури могат да имат различна форма, например кръгла или правоъгълна. Тези алтернативи могат да бъдат включени в дефиниция на тип от вида

```
data Shape = Circle Float |  
            Rectangle Float Float
```

Дефиниция от вида на посочената означава, че съществуват два алтернативни начина за конструиране на елемент на Shape.

Примерни данни (обекти) от тип Shape:

Circle 3.0

Rectangle 45.9 87.6

Дефиниции на функции върху типа Shape:

```
isRound :: Shape -> Bool
isRound (Circle _)      = True
isRound (Rectangle _ _) = False
```

```
area :: Shape -> Float
area (Circle r)      = pi*r*r
area (Rectangle h w) = h*w
```

Производни екземпляри на класове

Възможно е да се дефинира нов алгебричен тип като например Temp или Shape, който да бъде екземпляр на множество вградени класове.

Примерни дефиниции от посочения вид:

```
data Season = Spring | Summer | Autumn | Winter
             deriving (Eq, Ord, Enum, Show, Read)
```

```
data Shape = Circle Float |
            Rectangle Float Float
            deriving (Eq, Ord, Show, Read)
```

Рекурсивни алгебрични типове

Често характерът на решаваните задачи е такъв, че е естествено някои от алгебричните типове, които потребителят дефинира, да се описват в термините на самите себе си. Такива алгебрични типове се наричат **рекурсивни**.

Например понятието “израз” може да се дефинира или като **литерал** – цяло число, или като комбинация на два израза, в която се използва аритметичен оператор като + или –.

Примерна дефиниция на Haskell:

```
data Expr = Lit Int |  
           Add Expr Expr |  
           Sub Expr Expr
```

Аналогично понятието “двоично дърво” може да се дефинира или като `nil`, или като комбинация от стойност и две поддървета.

Съответната дефиниция на Haskell изглежда по следния начин:

```
data NTree = NilT |  
            Node Int NTree NTree
```

Тази дефиниция е подходяща за моделирането на двоични дървета от цели числа (двоични дървета от тип `Int`).

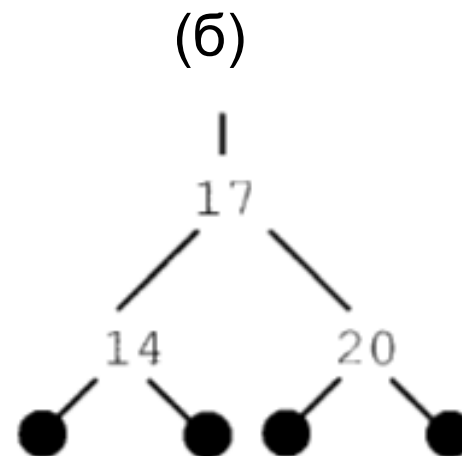
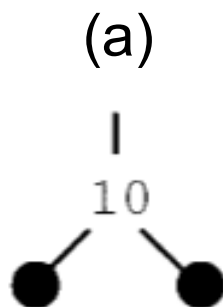
Празното дърво се представя чрез **NilT**, а дърветата от фиг. (a) и (б) се представят чрез

-- (a)

Node 10 NilT NilT

-- (б)

Node 17 (Node 14 NilT NilT) (Node 20 NilT NilT)



Дефиниции на някои функции за работа с двоични дървета от цели числа:

`sumTree, depth :: NTree -> Int`

`sumTree NilT = 0`

`sumTree (Node n t1 t2) = n + sumTree t1 + sumTree t2`

`depth NilT = 0`

`depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)`


```
occurs :: NTree -> Int -> Int
```

```
occurs NilT p = 0
```

```
occurs (Node n t1 t2) p
```

```
  | n==p      = 1 + occurs t1 p + occurs t2 p
```

```
  | otherwise = occurs t1 p + occurs t2 p
```

Лекция 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

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)

Таблицы 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

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 по 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 ($2^{32}-1 = 2147483647$).

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 <code>div 14 3</code> е 4. Може да се запише и инфиксно: <code>14 `div` 3</code> .
mod	Остатък при целочислено деление, например <code>mod 14 3</code> или <code>14 `mod` 3</code> .
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 (“претоварване” или 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**.

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 мантика 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, Float -> Int
floor,
round

конвертиране на реално число
в цяло чрез закръгляне нагоре,
закръгляне надолу или
закръгляне до най-близкото
цяло число

fromInt Int -> Float

конвертиране на цяло число
в реално

Знакове (characters, Char)

Представяват отделни знакове, заградени в единични кавички (апострофи), например 'd' или '3'.

Връзка между знаковете и техните ASCII кодове:

`ord :: Char -> Int`

`chr :: Int -> Char`

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)
```

Проверка дали даден знак е цифра:

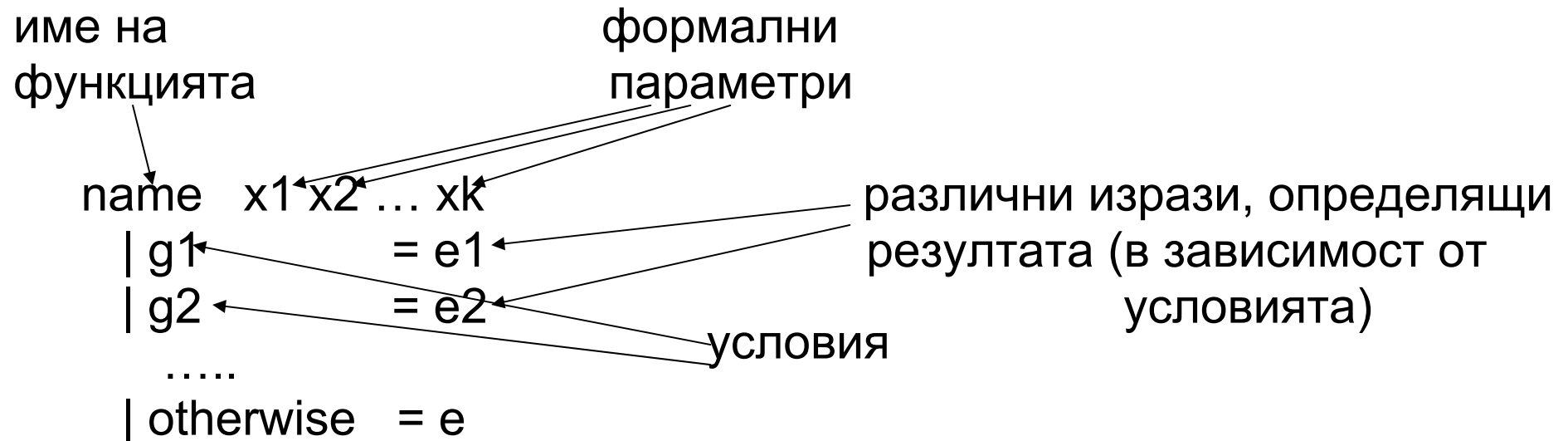
```
isDigit :: Char -> Bool
```

```
isDigit ch = ('0' <= ch) && (ch <= '9')
```


Програмиране на условия (guards)

Условието (“охраняем” 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 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 условията (охраняемите изрази), 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 стандартния прелюд на Haskell

Функцията `max` е вградена (дефинирана в стандартния прелюд 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 условия израз съвпада със стойността на `m`, в противен случай стойността на условия 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

Линейна рекурсия и итерация

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:

$$n! = n \cdot (n-1)!, \quad n \geq 2,$$

$$1! = 1.$$

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

Забележка. Предложената дефиниция не е достатъчно прецизна (функцията няма да работи добре при аргумент, по-малък 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:

$$n! = 1.2.3. \dots .n.$$

Според тази дефиниция $n!$ може да се получи, като най-напред се умножи **1.2**, след това полученят 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.

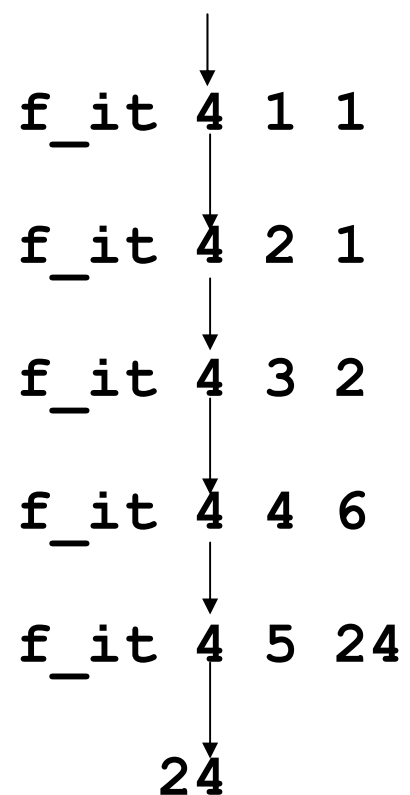
Първоначално **product = counter = 1**, а когато **counter** получи стойност, по-голяма от **n** (т.е. когато counter стане **n+1**), **product** ще има стойност **n!**.

```
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):

factorial 4



Сравнение между 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 аргументи, докато се стигне до обръщение с т. нар. базов (прост, граничен) вариант 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 изчисленията 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.

Ако при даден рекурсивен процес дължината на генерираната верига (и, следователно, обемът на необходимата памет за нейното съхраняване) расте линейно с нарастването на аргумента n (т.е. е линейна функция на n), процесът се нарича **линейно рекурсивен** (говори се още за линеен рекурсивен процес).

Ако при даден итеративен процес броят на стъпките (и, следователно, времето за съответните пресмятания) расте линейно с нарастването на аргумента n (т.е. е линейна функция на n), процесът се нарича **линейно итеративен** (говори се още за линеен итеративен процес).

Забележки

1) За описание както на рекурсивния, така и на итеративния процес използвахме функции, които синтактично са рекурсивни (в дефиницията на такава функция има обръщение към нея самата, т.е. в дефиницията на функцията се посочва нейното име).

2) От отбелязаното току-що следва, че е възможно итерацията да се емулира, т.е. по същество в езиците за програмиране няма нужда от итеративни конструкции. Такива обаче са предвидени в повечето езици за удобство.

3) В програмирането по отношение на синтаксиса на функциите се говори за т. нар. **линейна рекурсия** и **опашкова рекурсия**. При линейната рекурсия дефиницията включва (поражда) само едно рекурсивно обръщение към същата функция с опростени аргументи. Опашковата рекурсия (tail recursion) е линейна рекурсия, при която общата задача се трансформира до нова, по-проста, като при това решението на общата задача съвпада с решението на по-простата, а не се получава от него с помощта на допълнителни операции.

Така общата задача директно се редуцира до по-проста и няма нужда от обратен ход за получаването на решението на общата задача. Затова по принцип използването на опашкова рекурсия създава условия за по-голяма ефективност на съответните функции, отколкото използването на линейна рекурсия от по-общ вид.

Такава ефективност е налице например при опашково рекурсивните функции на езика Scheme (и **не е налице в езика Haskell**), тъй като интерпретаторите на Scheme се реализират така, че при оценяването на обръщения към опашково рекурсивни функции не използват допълнително пространство от системния стек. Затова се казва, че интерпретаторите на езика Scheme имат т. нар. **опашково рекурсивна семантика**.

Дървовидна рекурсия

Примерна задача. Да се състави програма на намиране на n -тото число на Фибоначи.

Както е известно, редицата от числата на Фибоначи има следния вид: 0, 1, 1, 2, 3, 5, 8, 13, 21, n -тото число на Фибоначи се пресмята по формулата:

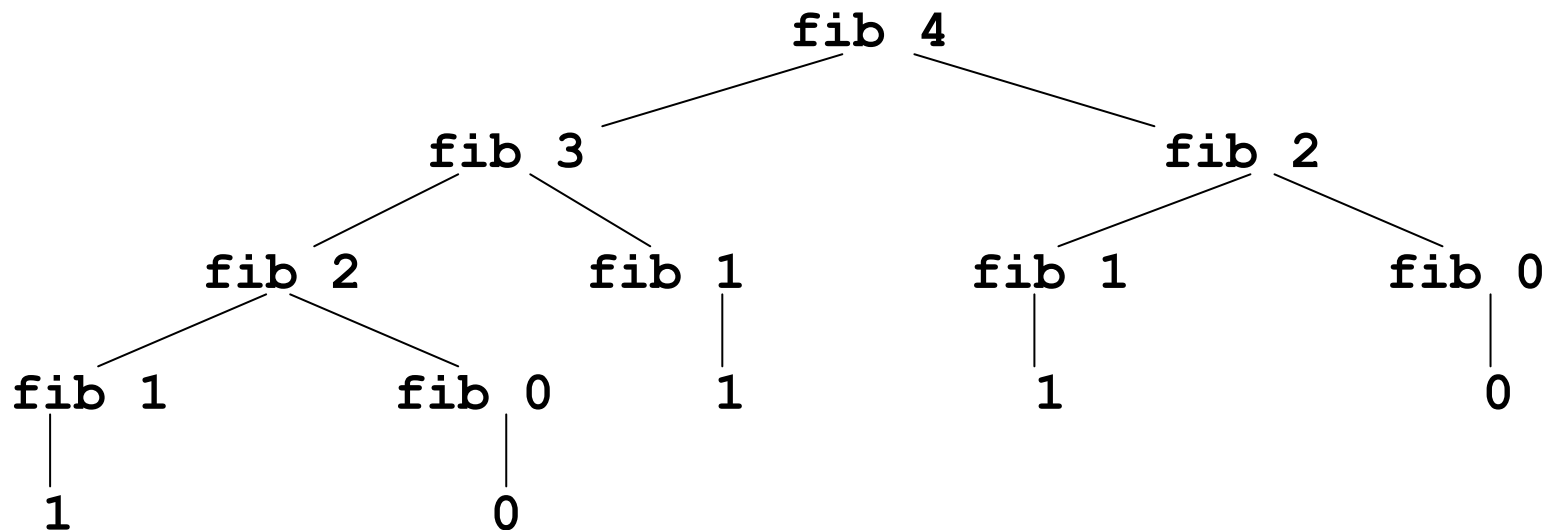
$$\text{Fib}(n) = \begin{cases} 0 & , n=0 \\ 1 & , n=1 \\ \text{Fib}(n-1)+\text{Fib}(n-2) & , n>1 \end{cases}$$

Решение

Първи начин - чрез рекурсивна функция, която реализира рекурсивен процес

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

Процесът на оценяване на обръщението към тази функция може да бъде разглеждан като процес на обхождане **в дълбочина** на дърво от вида:



Затова процесът, породен от тази функция, се нарича дървовидно рекурсивен (говори се още за дървовиден рекурсивен процес).

Анализ:

- $\text{Fib}(n)$ расте експоненциално с нарастването на n . По-точно, $\text{Fib}(n)$ е най-близкото цяло число до $\Phi^n/\sqrt{5}$, където $\Phi = (1+\sqrt{5})/2 \approx 1.6180$ е положителният корен на уравнението $\Phi^2 = \Phi + 1$;
- времето за намиране на $\text{Fib}(n)$ е пропорционално на броя на генерираните рекурсивни обръщания (броя на стъпките, за които се извършва пресмятането), който е равен на броя на възлите в дървото, който от своя страна е от порядъка на 2^{n-1} ;
- обемът на необходимата памет е пропорционален на дълбочината на дървото, която е от порядъка на n .

Втори начин - чрез рекурсивна функция, която реализира итеративен процес

Идея. Използват се две цели числа a и b , инициализирани съответно с 0 и 1, върху които n -кратно се прилага следната едновременна трансформация: $a' := b, b' := a+b$.

```
fibonacci :: Integer -> Integer
fibonacci n = f_it 0 1 n
```

```
f_it :: Integer -> Integer -> Integer -> Integer
f_it a b count
  | count==0      = a
  | otherwise     = f_it b (a+b) (count-1)
```

Анализ. Тази функция поражда линеен итеративен процес, броят на стъпките на който е пропорционален на n .

Следователно, времето, необходимо за пресмятане на $\text{Fib}(n)$ чрез двете предложени функции, е твърде различно (в първия случай то расте експоненциално, а във втория - линейно). При това, разликите са съществени дори при малки стойности на n .

От показаното по-горе обаче не следва, че дървовидната рекурсия е нещо безполезно. Тя е особено подходяща при работа с рекурсивни типове данни. Самият процес на работа на интерпретатора на Haskell (процесът на оценяване на изрази в Haskell) е дървовиден рекурсивен процес. Дори и при числови данни дървовидната рекурсия често е много удобно средство. Например, в задачата за намиране на $\text{Fib}(n)$ решението чрез дървовидна рекурсия е по-неефективно, но е много по-естествено (представлява точен запис чрез средствата на езика Haskell на дефиницията на числата на Фибоначи).

Пример, който демонстрира ползата от дървовидна рекурсия при числови данни: намиране на броя на възможните начини за "разваляне" на дадена парична сума на стотинки (например, на монети от 1, 2, 5, 10, 20 и 50 ст.).

Идея за решение

Избираме и фиксираме една от възможните монети (най-добре е да се въведе наредба между монетите в зависимост от означаваната от тях сума и да се фиксира най-едрата монета). Тогава общият брой на търсените начини е равен на броя на начините за "разваляне" на сумата без използване на избрания вид монети плюс броя на начините за "разваляне" на дадената сума с използване на поне една монета от избрания вид.

Следователно,

$$\begin{array}{lll} \text{брой начини за} & & \text{брой начини за} & & \text{брой начини за} \\ \text{"разваляне" на} & & \text{"разваляне" на} & & \text{"разваляне" на} \\ \text{сумата } a, & = & \text{сумата } a, & + & \text{сумата } a-d, \\ \text{използвайки } n & & \text{използвайки } n-1 & & \text{използвайки } n \\ \text{типа монети} & & \text{типа монети} & & \text{типа монети} \end{array}$$

Тук d е номиналната стойност на фиксирания тип монети.

Базови (гранични) случаи:

- $a=0$: 1 начин;
- $a<0$: 0 начина;
- $n=0$ (и $a>0$): 0 начина.

Лекция 3

**Недефинирани стойности на функции. Оператори.
Въведение в съставните типове данни в Haskell**

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 $f(-2)$ в Hugs, 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 справяне с 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 ***прекръпяване 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

Оператори

Операторите в Haskell са *инфиксни функции*, т.е. такива (двуаргументни) функции, означенията на които се записват между аргументите им, а не преди тях.

По принцип е възможно поредици от прилагания на група оператор да бъдат записани с използване на скоби като например

$((((4+8)+(7+2))*3)+5)$

Този запис обаче не е много удобен, затова по принцип употребата на излишни скоби на практика се избягва.

Това е възможно заради наличието на две важни свойства на оператори – техните **приоритет** (сила на свързването, binding power) и **асоциативност**.

Приоритет

Приоритетът е свойство на операторите, което определя реда на изпълнение на поредица от различни оператори.

Например в алгебрата операциите (операторите) умножение и деление имат по-висок приоритет от събирането и изваждането, а степенуването има по-висок приоритет от умножението и делението.

Това означава, че $2+3*4$ е еквивалентно на $(2+(3*4))$, а 2^3*4 е еквивалентно на $((2^3)*4)$.

В Haskell всеки (вграден) оператор има своя **сила на свързването** (binding power) – цяло неотрицателно число (цяло число между 0 и 9), което определя неговия приоритет.

Например умножението (*) има приоритет (сила на свързването) 7, събирането (+) има приоритет 6 и т.н.

Асоциативност

Асоциативността е свойство на операторите, което определя реда на изпълнение на поредица от еднакви оператори.

Например в алгебрата операциите (операторите) събиране и умножение са **асоциативни**, т.е. редът на изпълнение на поредица от събирания и умножения е без значение:

$$(a + b) + c = a + (b + c),$$

т.е. записът $a + b + c$ може да се интерпретира еднозначно;

$$(a * b) * c = a * (b * c),$$

т.е. записът $a * b * c$ може да се интерпретира еднозначно.

Изваждането и делението обаче не са асоциативни, защото

$$(a - b) - c \neq a - (b - c) \text{ и}$$
$$(a / b) / c \neq a / (b / c)$$

В Haskell повечето оператори, които не са асоциативни, се характеризират или като **ляво асоциативни**, или като **дясно асоциативни**.

Ако един оператор е ляво асоциативен, то всяка поредица от последователни обръщения към него се интерпретира като заградена със скоби от ляво.

Ако един оператор е дясно асоциативен, то всяка поредица от последователни обръщения към него се интерпретира като заградена със скоби от дясно.

Например, изваждането и делението са ляво асоциативни, т.е.
 $a - b - c$ се интерпретира като $(a - b) - c$ и
 $a / b / c$ се интерпретира като $(a / b) / c$.

Обратно, степенуването е дясно асоциативно, т.е.
 $a ^ b ^ c$ се интерпретира като $a ^ (b ^ c)$.

Забележка 1. Най-висок приоритет в Haskell има прилагането на функции, което стандартно се записва в префиксна форма:

$f \ v_1 \ v_2 \ \dots \ v_n$.

Това в частност означава, че записът **$f \ n+1$** се интерпретира като **$(f \ n) + 1$** .

Забележка 2. Знакът “-“ е означение едновременно на инфиксен и префиксен оператор, затова възниква опасност от колизия при използването му в случаи от типа на **$f \ -12$** . Този конкретен израз се интерпретира като означение на разликата (изваждането) $f-12$, а не като прилагане на f към числото -12 .

Дефиниране на оператори

Haskell позволява на потребителя да дефинира нови оператори по същия начин, както се извършва дефинирането на функции.

Имената на операторите могат да включват ASCII символите
! # \$ % & * + . / < > ? \ ^ | : - ~

Името на оператор не може да започва с двоеточие.

Например дефиницията на оператора &&& като функция за намиране на минималното от две цели числа може да изглежда по следния начин:

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

Приоритетът и асоциативността на един оператор, дефиниран от потребителя, могат да бъдат специфицирани явно, например:

infixl 7 &&& означава, че операторът &&& има лява асоциативност и приоритет 7;

infixr 6 ^^ означава, че операторът ^^ има дясна асоциативност и приоритет 6.

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, \dots, t_n) , който включва векторите (v_1, v_2, \dots, v_n) , за които $v_1 :: t_1, v_2 :: t_2, \dots, v_n :: t_n$.

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***:

```
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 по 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))
```

Примери за използване на вектори

Пример 1. Намиране на минималното и максималното от две цели числа.

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

Пример 2. “Бързо” пресмятане на числата на Фибоначи.

```
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))
```

```
fastFib :: Int -> Int
```

```
fastFib = fst . fibPair
```

Забележка. Инфиксният оператор “.” означава **композиция на функции** (последователно прилагане на функции):

$(f.g) x$ е еквивалентно на $f (g x)$.

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.

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

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

- $[n \dots m]$ е списъкът $[n, n+1, \dots, m]$; ако $n > m$, списъкът е празен.

$$[2 \dots 7] = [2, 3, 4, 5, 6, 7]$$

$$[3.1 \dots 7.0] = [3.1, 4.1, 5.1, 6.1]$$

$$['a' \dots 'm'] = \text{"abcdefghijklm"}$$

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

$$[7, 6 \dots 3] = [7, 6, 5, 4, 3]$$

$$[0.0, 0.3 \dots 1.0] = [0.0, 0.3, 0.6, 0.9]$$

$$['a', 'c' \dots 'n'] = \text{"acegikm"}$$

- Както се вижда от примерите, и в двата случая по-горе е възможно големината на стъпката да не позволява достигането точно на m . Тогава последният елемент на списъка съвпада с най-големия/най-малкия елемент на редицата, който е по-малък/по-голям или равен на m .

Определяне на обхвата на списък (*List Comprehension*)

Синтаксис:

- [*expr* | q_1, \dots, q_k] , където *expr* е израз, а q_i може да бъде
- **генератор** от вида $p \leftarrow lExpr$, където p е образец и *lExpr* е израз от списъчен тип
 - **тест**, *bExpr*, който е булев израз

При това в q_i могат да участват променливите, използвани в q_1, q_2, \dots, 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]
```

Генерични функции (полиморфизъм)

Много от вградените функции в Haskell са **полиморфни** или **генерични**, т.е. действат върху аргументи от различни типове.

“Полиморфизъм” буквално означава “наличие на много форми”. **Една функция е полиморфна, когато има много типове.**

Такива са например голяма част от функциите за работа със списъци.

Пример

Функцията `length` връща като резултат дължината (броя на елементите) на даден списък, независимо от типа на неговите елементи.

Следователно може да се запише:

`length :: [Bool] -> Int`

`length :: [Int] -> Int`

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

и т.н.

Обобщеният запис, който капсулира (encapsulates) горните, е
 $\text{length} :: [a] \rightarrow \text{Int}$

Тук ***a*** е ***променлива на тип*** (типова променлива, type variable), т.е. променлива, която означава произволен тип.

Типовете от вида на $[\text{Bool}] \rightarrow \text{Int}$, $[\text{Int}] \rightarrow \text{Int}$, $[[\text{Int}]] \rightarrow \text{Int}$ и т. н. са ***екземпляри*** на типа $[a] \rightarrow \text{Int}$.

Забележка. Променливата ***a*** в записа по-горе може да означава произволен тип, но всички нейни включвания в дадена дефиниция означават един и същ тип.

Някои функции за работа със списъци, реализирани в Prelude.hs:

<code>:</code>	<code>a -> [a] -> [a]</code>	Add a single element to the front of a list. <code>1:[2,3] => [1,2,3]</code>
<code>++</code>	<code>[a] -> [a] -> [a]</code>	Join two lists together. <code>"ab"++"cde" => "abcde"</code>
<code>!!</code>	<code>[a] -> Int -> a</code>	<code>xs!!n</code> returns the <code>n</code> th element of <code>xs</code> , starting at the beginning and counting from 0. <code>[14,7,3]!!1 => 7</code>
<code>concat</code>	<code>[[a]] -> [a]</code>	Concatenate a list of lists into a single list. <code>concat [[2,3],[],[4]] => [2,3,4]</code>
<code>length</code>	<code>[a] -> Int</code>	The length of the list. <code>length "word" => 4</code>

head	[a] -> a	The first element of the list. head "word" => 'w'
last	[a] -> a	The last element of the list. last "word" => 'd'
tail	[a] -> [a]	All but the first element of the list. tail "word" => "ord"
init	[a] -> [a]	All but the last element of the list. init "word" => "wor"
replicate	Int -> a -> [a]	Make a list of n copies of the item. replicate 3 'c' => "ccc"
take	Int -> [a] -> [a]	Take n elements from the front of a list. take 3 "Peccary" => "Pec"

drop	<code>Int -> [a] -> [a]</code>	Drop n elements from the front of a list. <code>drop 3 "Peccary" => "cary"</code>
splitAt	<code>Int->[a]->([a],[a])</code>	Split a list at a given position. <code>splitAt 3 "Peccary" => ("Pec","cary")</code>
reverse	<code>[a] -> [a]</code>	Reverse the order of the elements. <code>reverse [1,2,3] => [3,2,1]</code>
zip	<code>[a]->[b]->[(a,b)]</code>	Take a pair of lists into a list of pairs. <code>zip [1,2] [3,4,5] => [(1,3),(2,4)]</code>

unzip	<code>[(a,b)] -> ([a],[b])</code>	Take a list of pairs into a pair of lists. <code>unzip [(1,5),(2,6)] => ([1,2],[5,6])</code>
and	<code>[Bool] -> Bool</code>	The conjunction of a list of Booleans. <code>and [True,False] => False</code>
or	<code>[Bool] => Bool</code>	The disjunction of a list of Booleans. <code>or [True,False] => True</code>
sum	<code>[Int] -> Int</code> <code>[Float] -> Float</code>	The sum of a numeric list. <code>sum [2,3,4] => 9</code>
product	<code>[Int] -> Int</code> <code>[Float] -> Float</code>	The product of a numeric list. <code>product [0.1,0.4 .. 1] => 0.028</code>

Приложение към Лекция 3:

Библиотечна “база от данни”

Примерна библиотечна “база от данни” (Example: A Library Database)

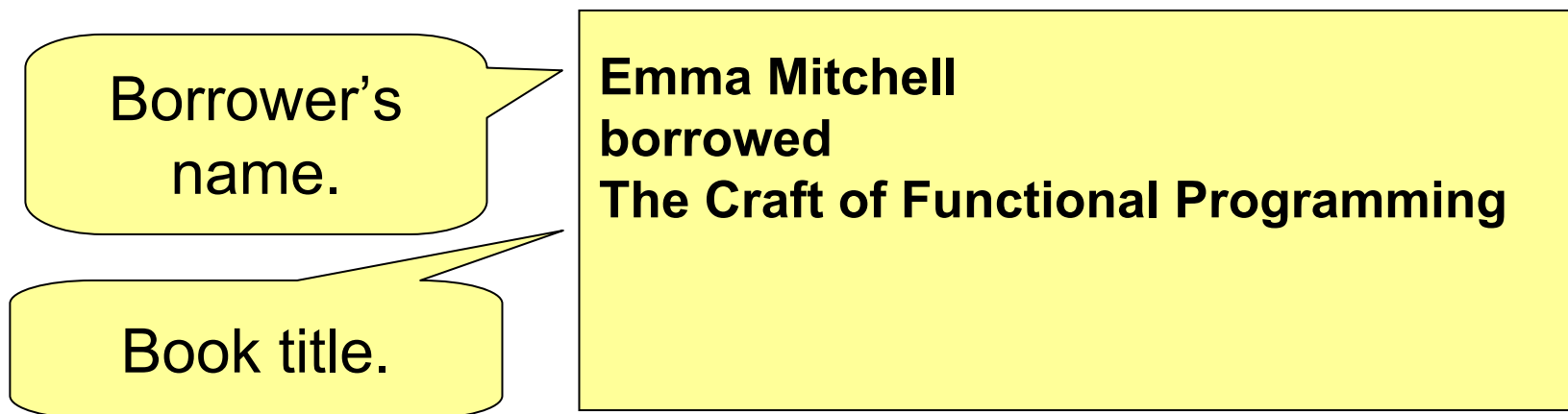
Computerise the librarian's card file:

Emma Mitchell
borrowed
The Craft of Functional Programming

Michael Strang
borrowed
Algorithms and Data Structures

Representing a Loan Card

What is the information on a loan card?



```
type Borrower = String
```

```
type Book = String
```

```
type Card = (Borrower, Book)
```


Representing the Card File

What is the contents of the card file?



a sequence of loan cards!

```
type Database = [Card]
```

```
example :: Database
```

```
example = [ ("Emma Mitchell", "The Craft of Functional Programming"),  
            ("Michael Strang", "Algorithms and Data Structures") ]
```

Querying the Database

What information would we wish to extract?

`books :: Database -> Borrower -> [Book]`

`borrowers :: Database -> Book -> [Borrower]`

`borrowed :: Database -> Book -> Bool`

`numBorrowed :: Database -> Borrower -> Int`

Which Books Have I Borrowed?

books example "Emma Mitchell" =

["The Craft of Functional Programming"]

books example "Mary Sheeran" = []

No books
borrowed.

books db person =

[book | (borrower, book) <- db, borrower == person]

Pattern matching again.

Quiz

Define:

`numBorrowed :: Database -> Borrower -> Int`

Quiz

Define:

`numBorrowed :: Database -> Borrower -> Int`

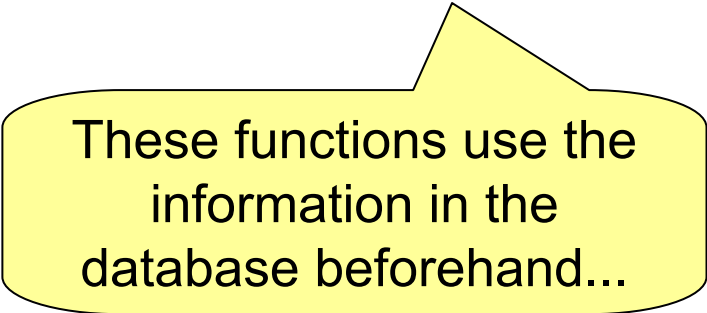
`numBorrowed db person = length (books db person)`

Updating the Database

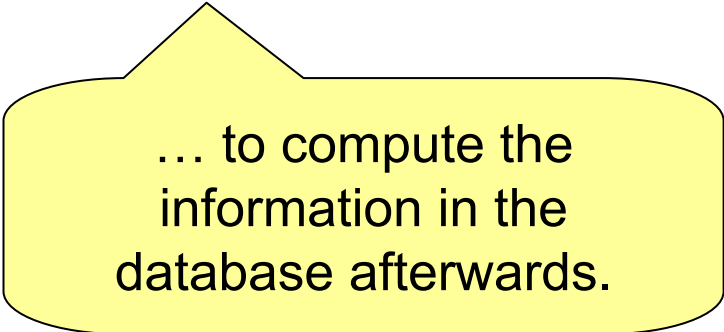
Making and returning loans changes the contents of the database.

`makeLoan :: Database -> Borrower -> Book -> Database`

`returnLoan :: Database -> Book -> Database`



These functions use the information in the database beforehand...



... to compute the information in the database afterwards.

Making a Loan

`makeLoan :: Database -> Borrower -> Book -> Database`

`makeLoan db borrower book = [(borrower, book)] ++ db`



Make a new card.

Returning a Loan

```
returnLoan :: Database -> Book -> Database
```

```
returnLoan db book =
```

```
    [(borrower, book1) | (borrower, book1) <- db,  
                        book1 /= book]
```


Лекция 4

**Локални дефиниции.
Програмиране със списъци в Haskell**

Локални дефиниции в Haskell

Пример 1

Дефиниция на функция, която връща като резултат сумата от квадратите на две числа.

```
sumSquares :: Int -> Int -> Int
sumSquares n m
    = sqN + sqM
    where
        sqN = n*n
        sqM = m*m
```

Пример 2

Най-напред ще дефинираме функцията `addPairwise`, която събира съответните елементи на два списъка от числа, като за целта изчерпва елементите на по-късия списък и игнорира останалите елементи на по-дългия.

Например: `addPairwise [1,7] [8,4,2] = [9,11]`.

```
addPairwise :: [Int] -> [Int] -> [Int]
addPairwise intList1 intList2
  = [m+n | (m,n) <- zip intList1 intList2]
```

Сега ще дефинираме нова функция, `addPairwise'`, която действа подобно на `addPairwise`, но включва в резултата и всички останали елементи на по-дългия списък.

Например: `addPairwise' [1,7] [8,4,2,67] = [9,11,2,67]`.

```
addPairwise' :: [Int] -> [Int] -> [Int]
```

```
addPairwise' intList1 intList2
```

```
  = front ++ rear
```

```
    where
```

```
    minLength = min (length intList1)
                  (length intList2)
```

```
    front      = addPairwise (take minLength
                                intList1)
                                (take minLength
                                intList2)
```

```
    rear       = drop minLength intList1 ++
                  drop minLength intList2
```

Ще обърнем специално внимание, че в тази дефиниция стойността на `minLength` се пресмята само един път, а се използва четири пъти.

Следва още едно решение на последната задача, в което обръщенията към `take` и `drop` са заменени с подходящи обръщения към `splitAt`.

```
addPairwise' :: [Int] -> [Int] -> [Int]
addPairwise' intList1 intList2
  = front ++ rear
  where
    minLength      = min (length intList1)
                      (length intList2)
    front           = addPairwise front1 front2
    rear            = rear1 ++ rear2
    (front1,rear1)  = splitAt minLength intList1
    (front2,rear2)  = splitAt minLength intList2
```

Общ вид на дефиниция на функция с използване на условия (условно равенство) с клауза where:

$$\begin{aligned}
 & f \ p_1 \ p_2 \ \dots \ p_k \\
 & \quad | \ g_1 \qquad \qquad = e_1 \\
 & \quad \dots \\
 & \quad | \ \text{otherwise} = e_r \\
 & \text{where} \\
 & \quad v_1 \ a_1 \ \dots \ a_n = r_1 \\
 & \quad v_2 = r_2 \\
 & \quad \dots \dots
 \end{aligned}$$

Клаузата where тук е присъединена към цялото условно равенство, т.е. към всички негови клаузи.

От горния запис се вижда, че локалните дефиниции могат да включват както дефиниции на променливи, така и дефиниции на функции (такава е например дефиницията на функцията v_1). Възможно е в клаузата `where` да бъдат включени и декларации на типовете на локалните обекти (променливи и функции).

Пример

```
maxsq x y
| sqx > sqy    = sqx
| otherwise    = sqy
where
  sqx = sq x
  sqy = sq y
  sq :: Int -> Int
  sq z = z*z
```


let изрази

Възможно е да се дефинират локални променливи с област на действие, която съвпада с даден израз.

Например изразът

```
let x = 3+2 in x^2 + 2*x - 4
```

има стойност 31.

Ако в един ред са включени повече от една дефиниции, те трябва да бъдат разделени с точка и запетая, например

```
let x = 3+2; y = 5-1 in x^2 + 2*x - y
```

Област на действие на дефинициите

Един скрипт на Haskell включва поредица от дефиниции. **Областта на действие** на дадена дефиниция съвпада с частта от програмата, в която може да се използва тази дефиниция. Всички дефиниции на най-високо ниво в Haskell имат за своя област на действие целия скрипт, в който са включени. С други думи, дефинираните на най-високо ниво имена могат да бъдат използвани във всички дефиниции, включени в скрипта. В частност те могат да бъдат използвани в дефиниции, които се намират преди техните собствени в съответния скрипт.

Пример

```
isOdd, isEven :: Int -> Bool
```

```
isOdd n
  | n <= 0      = False
  | otherwise   = isEven (n-1)
```

```
isEven n
  | n < 0       = False
  | n == 0      = True
  | otherwise   = isOdd (n-1)
```

Локалните дефиниции, включени в дадена клауза `where`, имат за област на действие само условното равенство, част от което е клаузата `where`.

Нека разгледаме още един път дефиницията на функцията `maxsq` от един от предишните примери:

```
maxsq x y
```

```
| sqx > sqy    = sqx  
| otherwise    = sqy  
where  
  sqx = sq x  
  sqy = sq y  
  sq :: Int -> Int  
  sq z = z*z
```

В тази дефиниция областта на действие на дефинициите на sqx , squ и sq и на променливите x и y е означена с големия правоъгълник, а областта на действие на променливата z е означена с малкия правоъгълник.

В случай, че даден скрипт съдържа повече от една дефиниция за дадено име, във всяка точка на скрипта е валидна (видима) “най-локалната” от тези дефиниции.

Пример

`maxsq x y`

	<code>sq x > sq y</code>	<code>= sq x</code>
	<code>otherwise</code>	<code>= sq y</code>
<code>where</code>		
	<code>sq x =</code>	<code>x*x</code>

Тук малкият вътрешен правоъгълник “отрязва” тази част от големия, която съвпада с областта на действие на “вътрешната” променлива с име `x`.

Примерна задача: конструиране и отпечатване на касова бележка.

Да се състави програмна система на Haskell, която моделира процеса на съставяне и отпечатване на касови бележки в резултат на сканирането на бар кодовете на стоките, избрани от купувачите.

Анализ на задачата

Сканиращите устройства на касите на супермаркетите имат за цел разпознаването на бар кодовете на избраните от купувачите стоки и формирането на съответни списъци от бар кодове от типа на

[1234, 4719, 3814, 1112, 1113, 1234] .

Всеки списък от посочения тип трябва да се конвертира в сметка (касова бележка) от вида

Haskell Stores

Dry Sherry, 1lt.....	5.40
Fish Fingers.....	1.21
Orange Jelly.....	0.56
Hula Hoops (Giant).....	1.33
Unknown Item.....	0.00
Dry Sherry, 1lt.....	5.40
Total.....	13.90

Бар кодовете и цените могат да се моделират например с цели числа (ще смятаме, че цените са изразени в пенсове). Имената на стоките ще представяме като символни низове.

Следователно, ще ни бъдат необходими следните типове:

-- Types of names, prices (pence) and bar-codes.

```
type Name      = String
type Price     = Int
type BarCode   = Int
```

Конвертирането ще се осъществи на основата на данните, съдържащи се в “базата от данни” на супермаркета, която свързва бар кодовете, имената и цените на стоките. За представянето на тази “база от данни” ще използваме списък от типа

-- The database linking names prices and bar codes.

```
type Database = [ (BarCode,Name,Price) ]
```

Примерна “база от данни”:

```
codeIndex :: Database
codeIndex = [ (4719, "Fish Fingers" , 121),
               (5643, "Nappies" , 1010),
               (3814, "Orange Jelly", 56),
               (1111, "Hula Hoops", 21),
               (1112, "Hula Hoops (Giant)", 133),
               (1234, "Dry Sherry, 1lt", 540) ]
```

Задачата на нашата програма е да конвертира списъка от бар кодове в списък от двойки от вида `(Name,Price)`, след което да конвертира новополучения списък в подходящ символен низ, който да бъде отпечатан както беше показано по-горе.

Ще бъде полезно да разполагаме със следните дефиниции на типове:

```
-- The lists of bar codes, and of Name,Price pairs.
```

```
type TillType = [BarCode]  
type BillType = [(Name,Price)]
```

Нека приемем за определеност, че дължината на редовете в сметката ще бъде равна на 30:

```
-- The length of a line in the bill.
```

```
lineLength :: Int  
lineLength = 30
```

При тези условия е необходимо да бъдат дефинирани следните функции:

```
makeBill :: TillType -> BillType  
-- Converts a list of bar codes to a list of  
-- name/price pairs.
```

```
formatBill :: BillType -> String
-- Converts s list of name/price pairs to
-- a formatted bill.

produceBill :: TillType -> String
-- Combines the effects of makeBill and formatBill.

produceBill = formatBill . makeBill
```

Приложение към Лекция 4:

**Примерна програмна система
за манипулиране на картинки**

Манипулиране на “картинки” (Pictures: A Case Study)

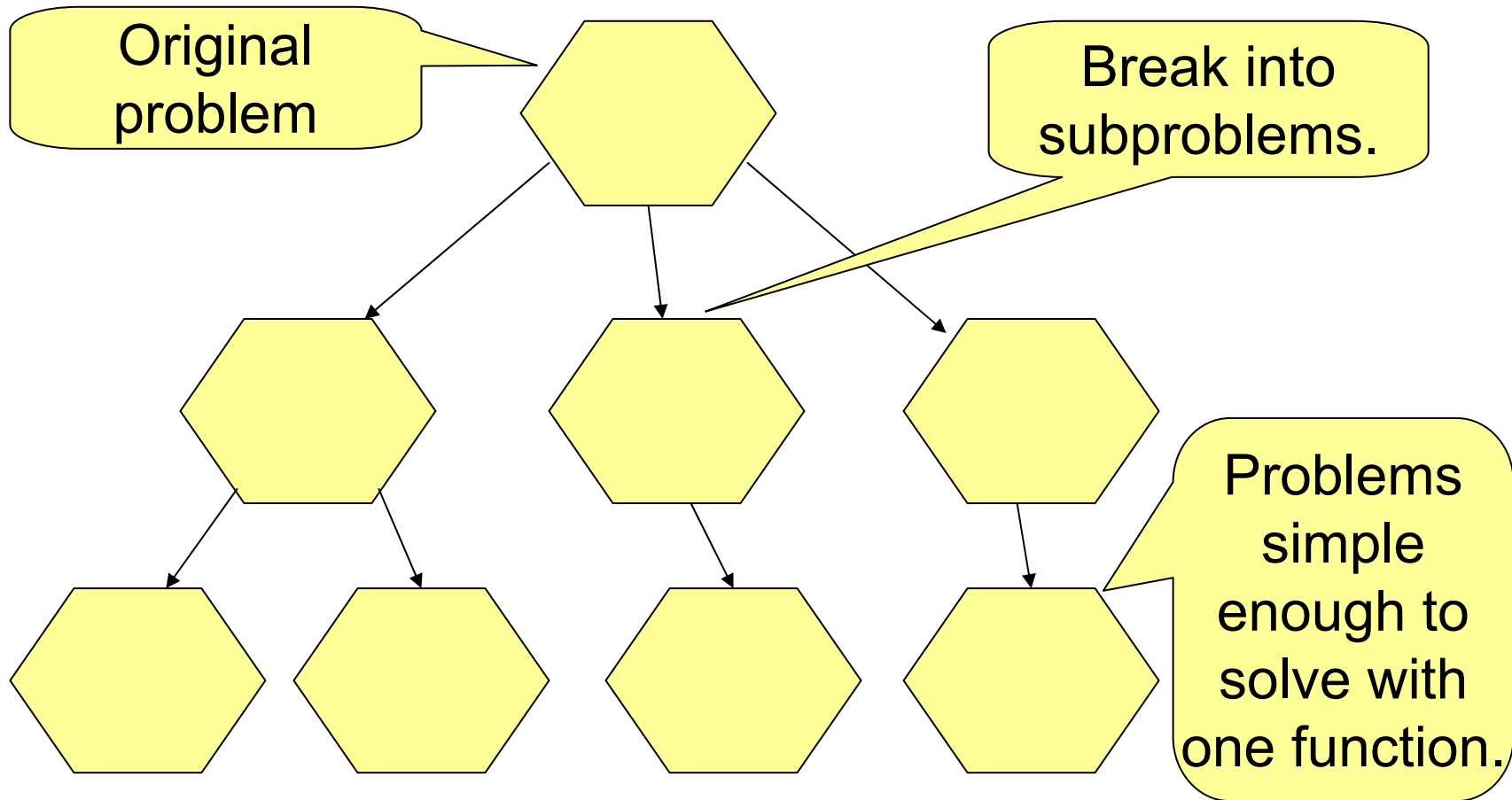
Purpose

We will go through a larger example of designing a part of a program: a library of functions for manipulating pictures.

Why?

- See examples of programming with lists; practice what we have learned.
- Introduce the type Char of characters.
- Discuss the design of larger parts of programs than individual functions.

Top-Down Design

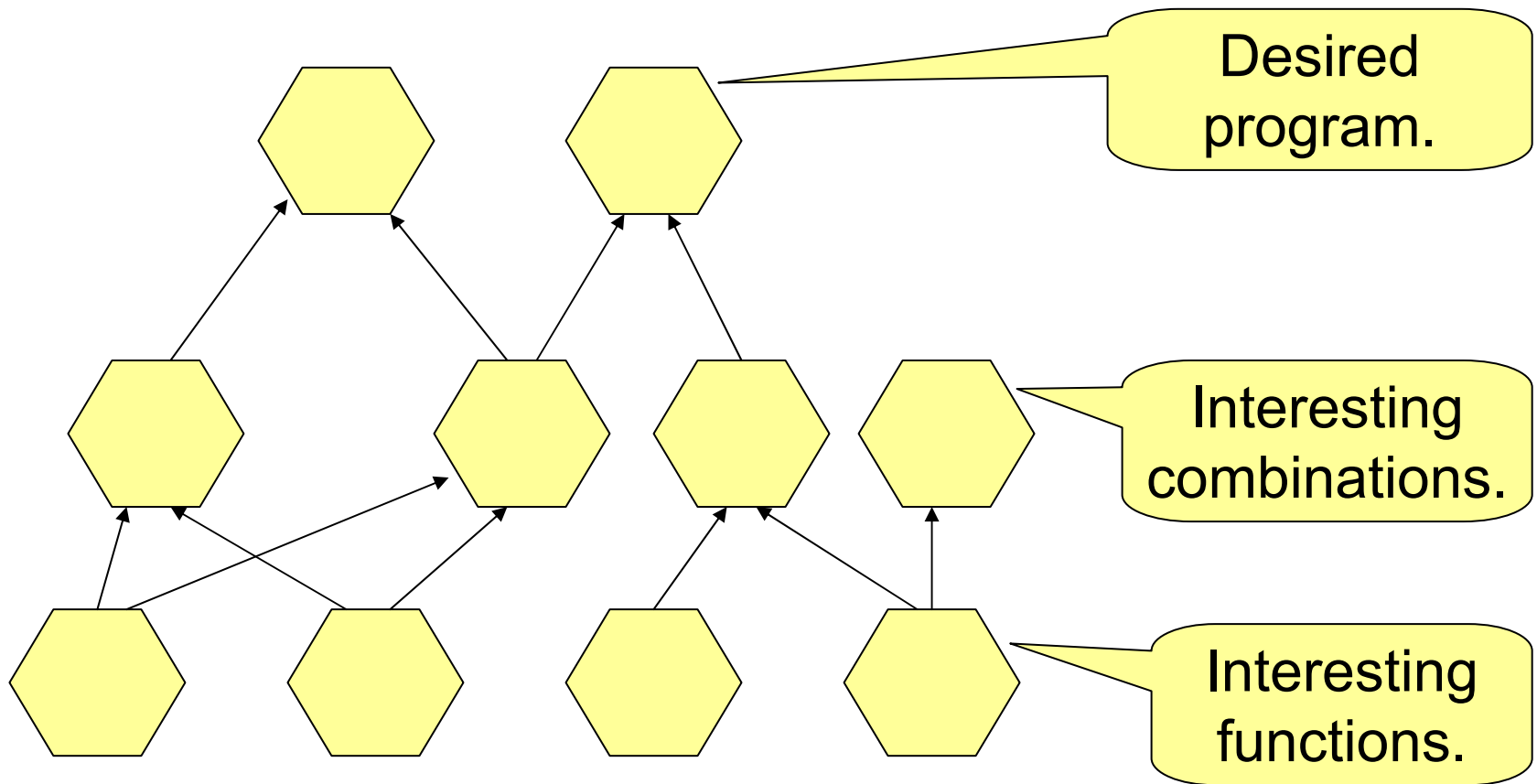


Top-Down Design

What functions might help me solve this problem?

- Break down a difficult problem into manageable pieces - an effective problem solving strategy.
- Work is directed towards solving the problem in hand - no unnecessary programming.
- May not result in many generally useful functions as a by-product.

Bottom-Up Design

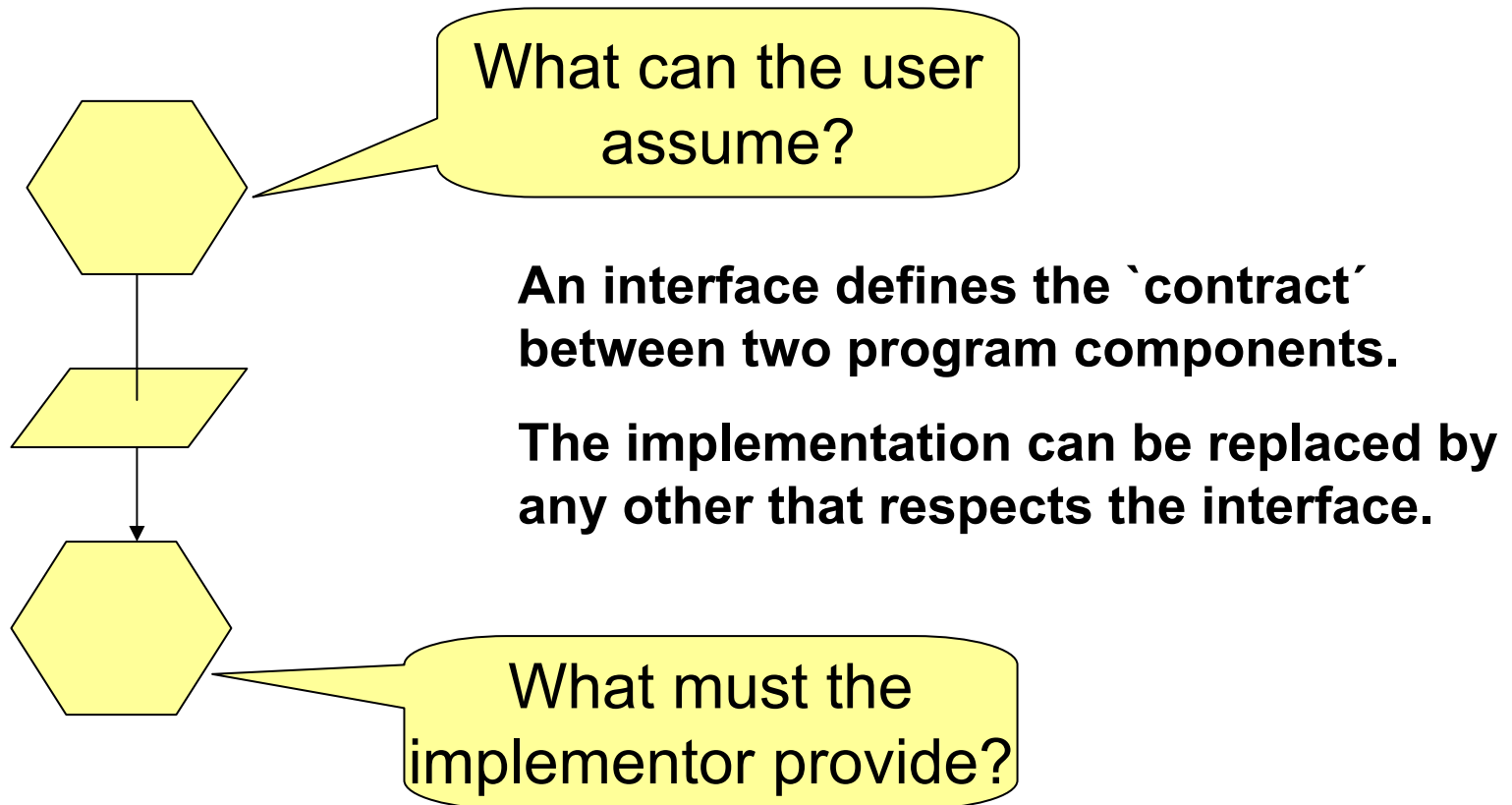


Bottom-Up Design

What can I do with the functions available to me?

- An effective way to build up powerful program components.
- Work is directed towards producing *generally useful* functions, which can then be reused in many applications.
- A solution looking for a problem!

Interfaces



Example: Pictures

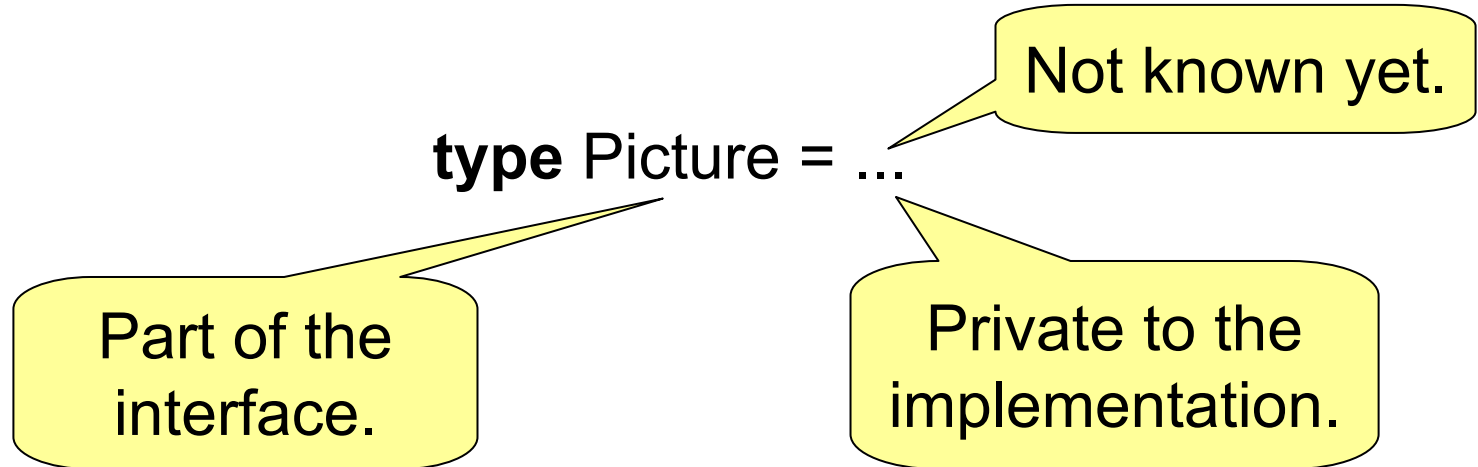
Top-down: We have discovered a need to manipulate rectangular pictures.

Bottom-up: What interesting functions can we provide?

Interface: What should the `contract` between picture users and the picture implementor specify?

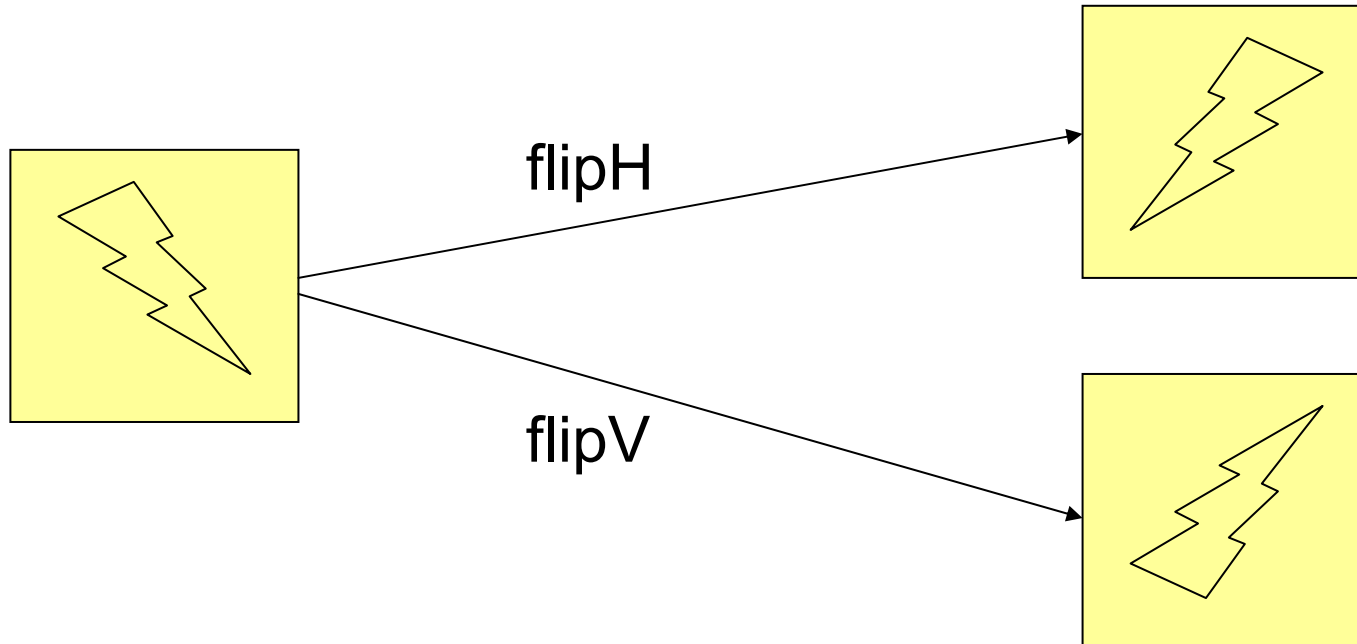
The Type of Pictures

Pictures are a kind of data; we will need a *type* for them.



A type whose name is public, but whose representation is private, is called an *abstract data type*.

Functions on Pictures



`flipH, flipV :: Picture -> Picture`

Properties of Flipping

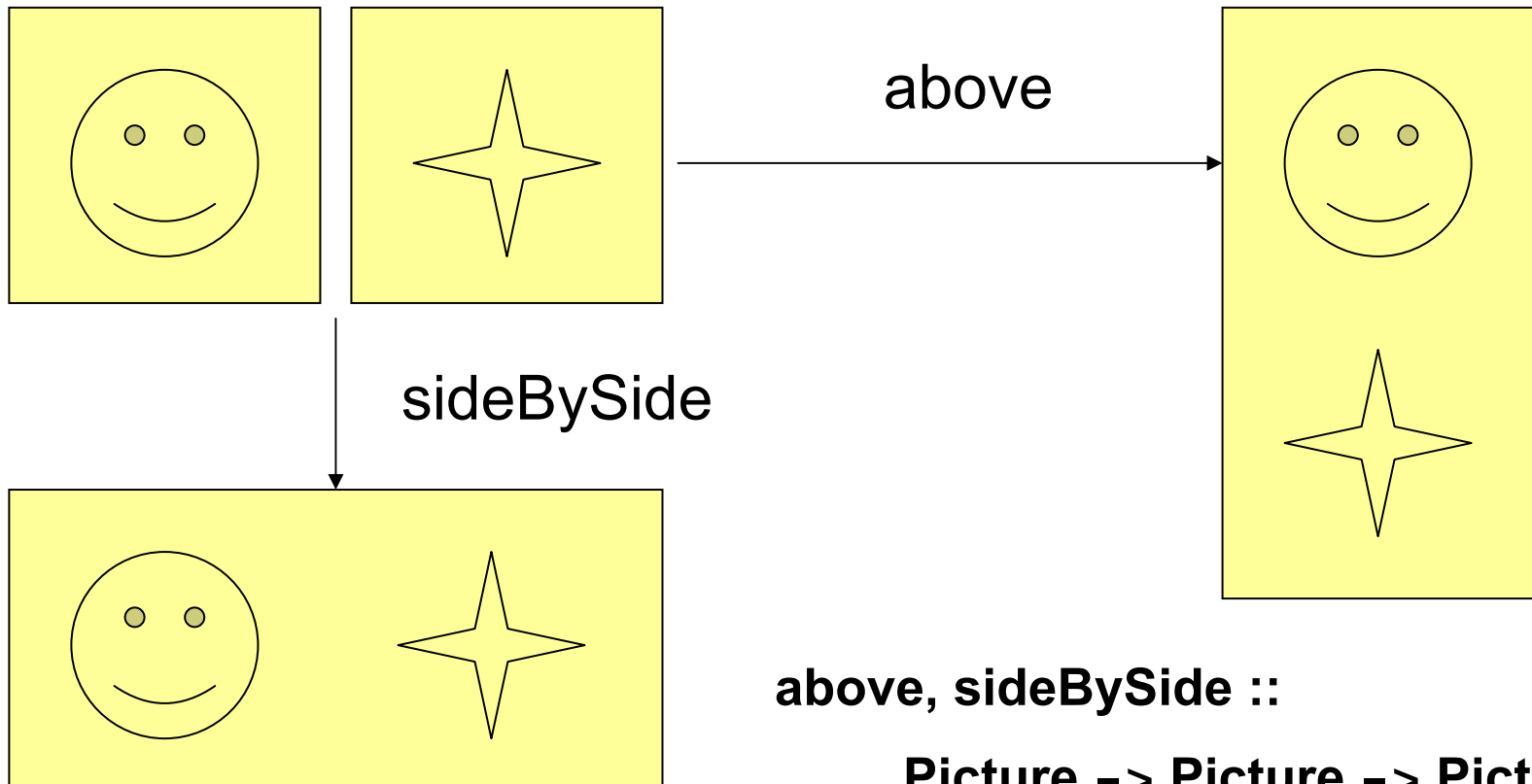
What properties should flipH, flipV satisfy?

- $\text{flipH} (\text{flipH} \text{ pic}) == \text{pic}$
- $\text{flipV} (\text{flipV} \text{ pic}) == \text{pic}$
- $\text{flipH} (\text{flipV} \text{ pic}) == \text{flipV} (\text{flipH} \text{ pic})$

Properties which
the user can assume.

Properties which the
implementor should guarantee.

Combining Pictures

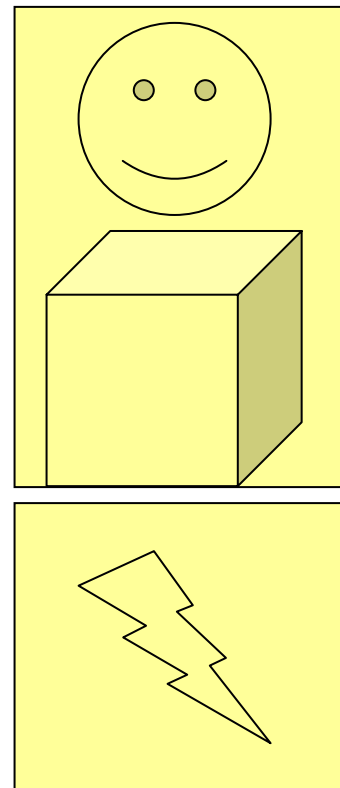
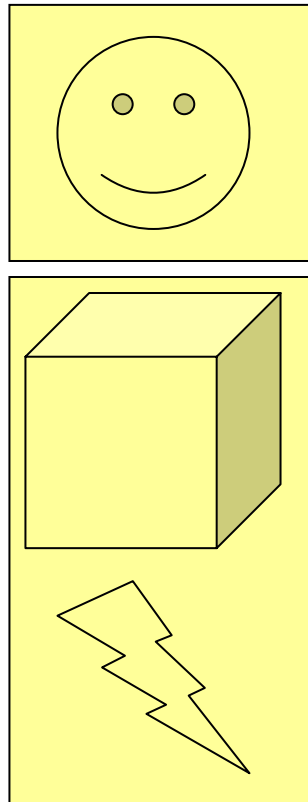


above, sideBySide ::

Picture -> Picture -> Picture

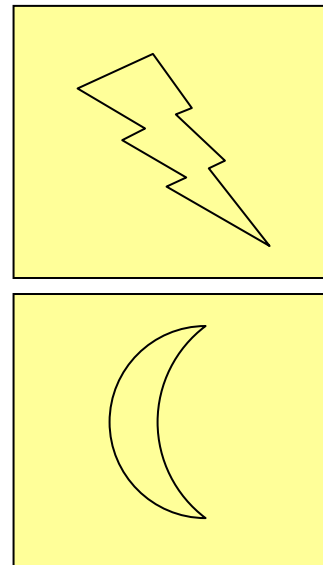
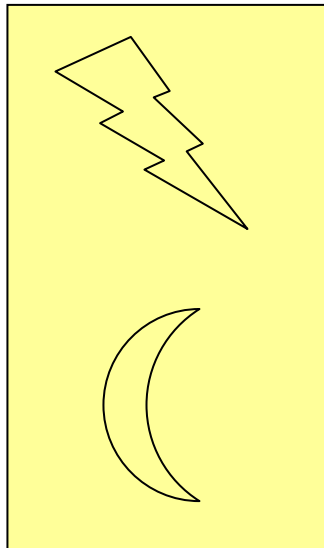
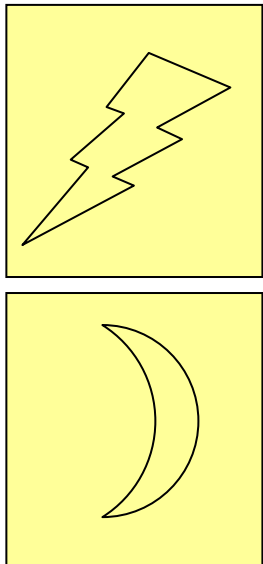
Properties of Combinations

$a \text{ `above` } (b \text{ `above` } c) == (a \text{ `above` } b) \text{ `above` } c$



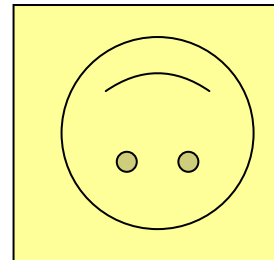
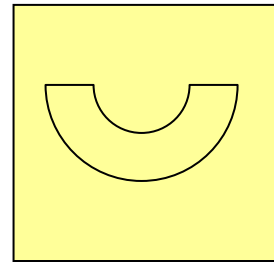
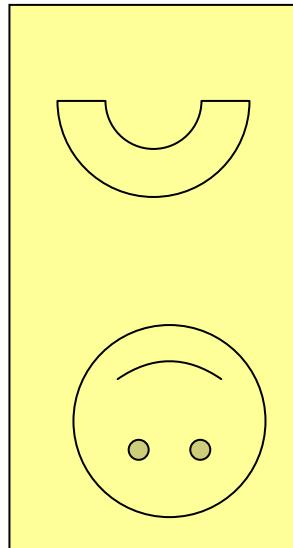
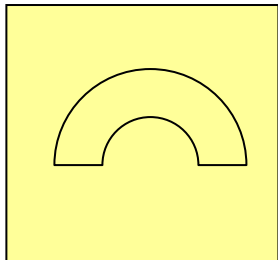
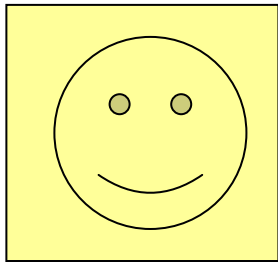
Properties of Combinations

$\text{flipH} (a \text{ `above` } b) == \text{flipH } a \text{ `above` } \text{flipH } b$



Properties of Combinations

$\text{flipV} (a \text{ `above` } b) == \text{flipV } b \text{ `above` } \text{flipV } a$



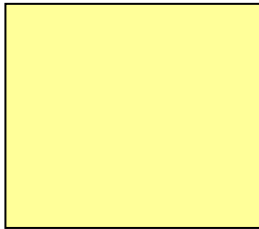
Printing Pictures

What can we do with a picture once we've constructed it?

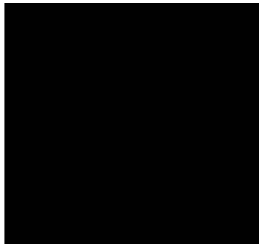
`printPicture :: Picture -> IO ()`

Quiz: Using Pictures

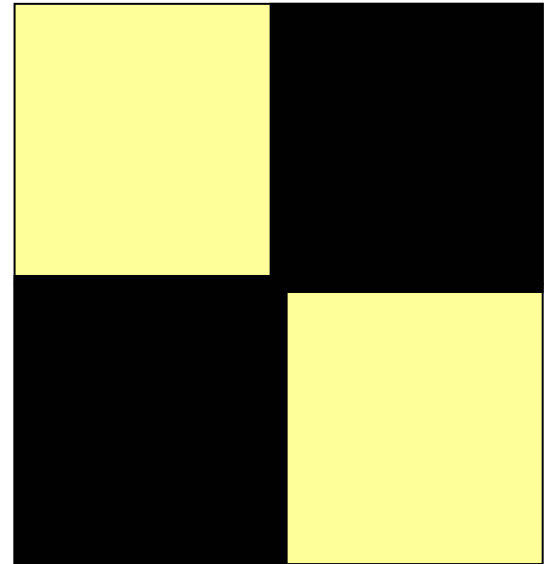
white



black

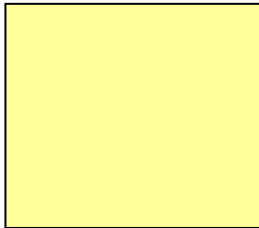


check

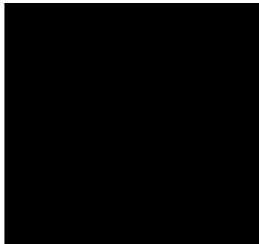


Quiz: Using Pictures

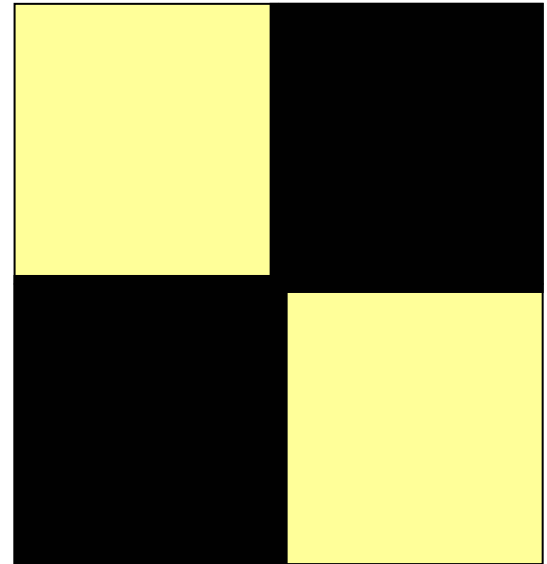
white



black



check



wb = white `sideBySide` black

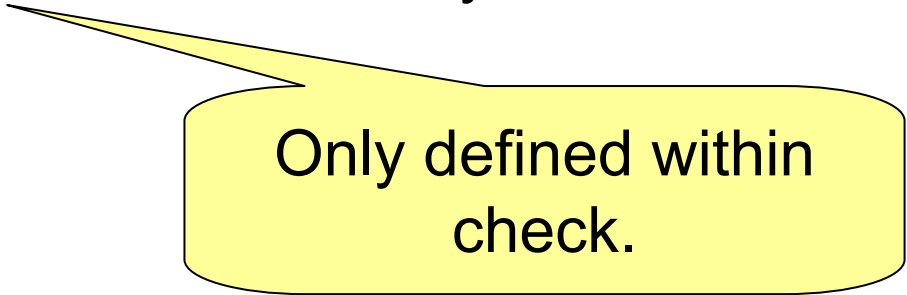
check = wb `above` flipH wb

Local Definitions

Wb is used only to define check. But when we read the program, we can't tell.

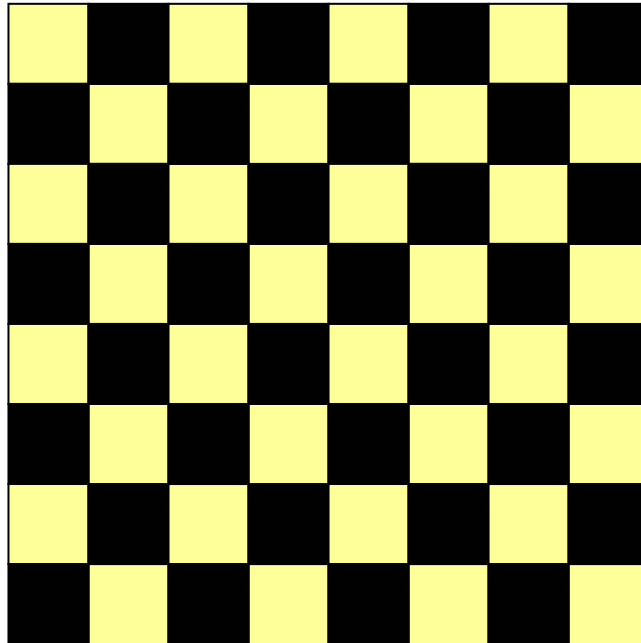
```
check = wb `above` flipH wb
```

```
where wb = white `sideBySide` black
```

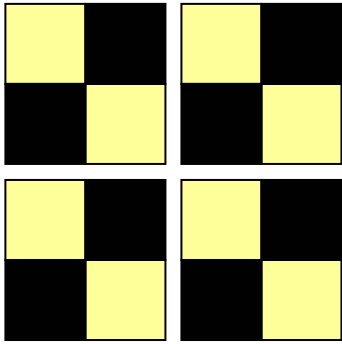


Only defined within
check.

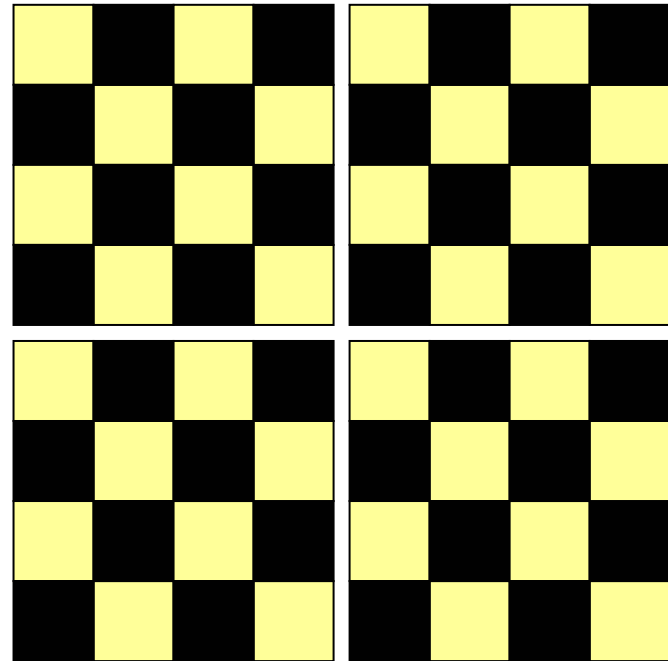
Quiz: Make a Chess Board



Quiz: Make a Chess Board

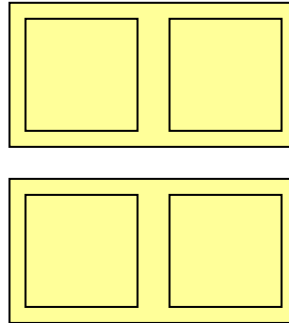


quartet check



quartet (quartet check)

Defining Quartet



```
quartet :: Picture -> Picture
```

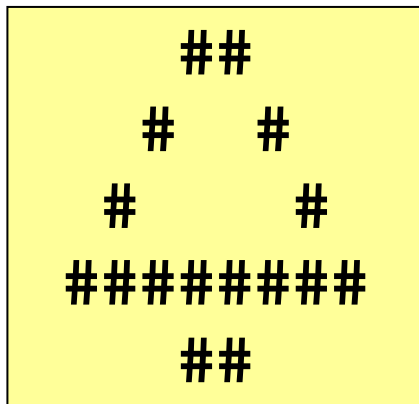
```
quartet pic = two `above` two
```

```
where two = pic `sideBySide` pic
```

A local definition can
use the function arguments.

Representing Pictures

We choose a representation which we can easily print from Hugs - not beautiful, but simple.



```
[ "      ##      ",  
  "     #   #     ",  
  "    #       #   ",  
  " #####",  
  "      ##      " ]
```

```
type Picture = [String]
```

The Truth About Strings

A string is a sequence of *characters*.

type String = [Char]

The type of
characters.

Examples:

'a', 'b', 'c' :: Char

' ' :: Char

A space is also
a character.

Flipping Vertically

flipV

[" ## ",		[" ## ",
" # # ",		" ##### ",
" # # ",	→	" # # ",
" ##### ",		" # # ",
" ## "]"		" ## "]"

flipV :: Picture -> Picture

flipV pic = reverse pic

Quiz: Flipping Horizontally

flipH

[" # " ,	→	[" # " ,
" # " ,		" # " ,
"#####" ,		"#####" ,
" # " ,		" # " ,
" # "]		" # "]

Recall: Picture = [[Char]]

Quiz: Flipping Horizontally

flipH

[" # " ,	→	[" # " ,
" # " ,		" # " ,
"#####" ,		"#####" ,
" # " ,		" # " ,
" # "]		" # "]

flipH :: Picture -> Picture

flipH pic = [reverse line | line <- pic]

Combining Pictures Vertically

```
[ "    #    ",  
  "      #   ",  
  "#####",  
  "      #   ",  
  "    #    "]
```

above

```
[ "  #    ",  
  " #     ",  
  "#####",  
  " #     ",  
  "  #    "]
```

```
[ "    #    ",  
  "      #   ",  
  "#####",  
  "      #   ",  
  "    #    ",  
  "  #     ",  
  " #     ",  
  "  #     ",  
  "  #     "]
```

above p q = p ++ q

Combining Pictures Horizontally

```
[ "   #   ",  
  "    #  ",  
  "#####",  
  "    #  ",  
  "   #   " ]`sideBySide`
```

```
[ "   #   ",  
  "    #  ",  
  "#####",  
  "    #  ",  
  "   #   " ]
```

→

```
      [ "   #   #   ",  
        "    #  #   ",  
        "#####",  
        "    #  #   ",  
        "   #   #   " ]
```

Combining Pictures Horizontally

```
[ "    #    ",  
  "    #    ",  
  "#####",  
  "    #    ",  
  "    #    " ] `sideBySide`  
[ "    #    ",  
  "    #    ",  
  "#####",  
  "    #    ",  
  "    #    " ]
```

sideBySide :: Picture -> Picture -> Picture

p `sideBySide` q = [pline ++ qline | (pline,qline) <- zip p q]

Printing

We will need to be able to *print strings*. Haskell provides a *command* to do so.

`putStr :: String -> IO ()`

A command with no result. Executing the command prints the string.

Main> putStr "Hello!"

Hello!

Main>

Executed the command.

Main> "Hello!"

"Hello!"

Main>

Displayed the value.

Line Breaks

How do we print more than one line?

The special character `'\n'` indicates the end of a line.

```
Main> putStr  
"Hello\nWorld"
```

```
Hello
```

```
World
```

```
Main>
```

```
Main> "Hello\nWorld"
```

```
"Hello\nWorld"
```

```
Main>
```

Printing a Picture

Print `["###",
 "# #",
 "###"]` as `"###\n# #\n###\n"`

A common operation, so there is a standard function

`unlines :: [String] -> String`

`printPicture :: Picture -> IO ()`

`printPicture pic = putStr (unlines pic)`

Extension: Superimposing Pictures

superimpose

[" # " ,	[" # " ,
" # " ,	" # " ,
"#####" ,	"#####" ,
" # " ,	" # " ,
" # "]	" # "]

→

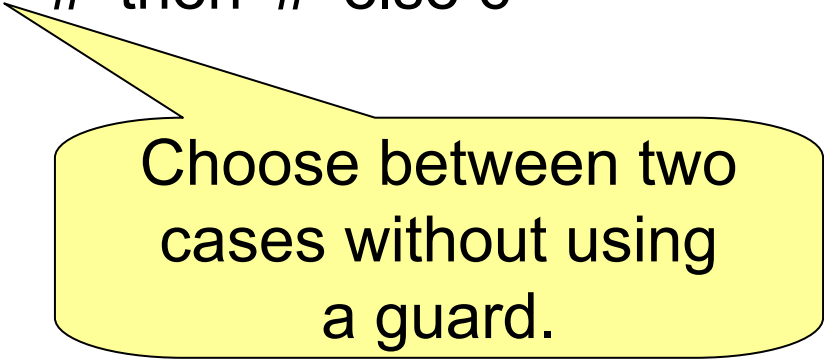
[" ## " ,
" # # " ,
"#####" ,
" # # " ,
" ## "]

Can We Solve a Simpler Problem?

We can superimpose two characters:

`superimposeChar :: Char -> Char -> Char`

`superimposeChar c c' = if c=='#' then '#' else c'`



Choose between two cases without using a guard.

Can We Solve a Simpler Problem?

We can superimpose two lines:

`superimposeLine :: String -> String -> String`

`superimposeLine s s' =`

`[superimposeChar c c' | (c,c') <- zip s s']`

`superimposeLine "# " " #"`

`[superimposeChar '# ' ',`

`→ superimposeChar ' ' ' ', → "# #"`

`superimposeChar ' ' '#']`

Superimposing Pictures

`superimpose :: Picture -> Picture -> Picture`

`superimpose p q =`

`[superimposeLine pline qline | (pline, qline) <- zip p q]`

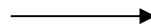
Function Libraries

The picture functions may be useful in many different programs.

Instead of copying them into each one, we place them together in a *library* which other programs may include.

Pictures.hs

```
module Pictures where  
type Picture = ...  
flipV pic = ...
```



Other.hs

```
import Pictures  
... flipV ... flipH ...
```

Defining the Interface

```
module Pictures  
  (Picture,  
   flipV, flipH,  
   above, sideBySide,  
   printPicture) where
```

...

These are the definitions which other programs may use.

Other definitions may appear here, but they are private.

Лекция 5

**Дефиниране на функции
за работа със списъци**

Съпоставяне по образец

Както вече показахме, дефинициите на функции в Haskell имат вида на условни равенства като например

```
mystery :: Int -> Int -> Int
mystery x y
  | x == 0      = y
  | otherwise   = x
```


Същата дефиниция може да се напише и с използване на поредица от две равенства:

```
mystery 0 y = y  
mystery x y = x
```

В тези две равенства последователно са описани отделните възможни случаи за стойността на `mystery`, като за целта са използвани **образци** – в случая литералът `0` и променливите `x` и `y`. Равенствата се прилагат **последователно**, като до всяко следващо равенство се достига и то евентуално се прилага само ако всички предишни не са дали резултат (т.е. ако съпоставянето им е завършило с неуспех).

Във второто равенство от дефиницията на `mystery` променливата `y` не се използва, затова тя може да бъде заменена от образаца, известен като ***wildcard*** (***специален символ за безусловно съпоставяне***):

```
mystery 0 y = y  
mystery x _ = x
```

При описанието на различните възможни случаи за стойността на дадена функция може да се използват и образци, с чиято помощ се именуват елементите на вектори.

Например:

```
joinStrings :: (String,String) -> String  
joinStrings (str1,str2) = str1 ++ str2
```

Образци (обобщение)

Образецът може да бъде:

- **Литерал** като например 24, 'f' или True; даден аргумент се съпоставя успешно с такъв образец, ако е равен на неговата стойност;
- **Променлива** като например x или longVariableName; образец от този вид се съпоставя успешно с аргумент с произволна стойност;
- **Специален символ за безусловно съпоставяне (wildcard) '_'**, който е съпоставим с произволен аргумент;
- **Вектор – образец** (p_1, p_2, \dots, p_n); за да бъде съпоставим с него, аргументът трябва да има вида (v_1, v_2, \dots, v_n), като всяко v_i трябва да бъде съпоставимо със съответното p_i ;

- **Конструктор**, приложим към даденото множество от аргументи.

Забележка. Някои видове образци от тип конструктор ще бъдат разгледани по-нататък.

Списъци и образци на списъци

Всеки списък или е празен (т.е. е равен на/съпоставим с `[]`), или е непразен. Във втория случай списъкът има глава и опашка, т.е. може да бъде записан във вида `x:xs`, където `x` е първият елемент на този списък, а `xs` е списъкът без първия му елемент (опашката на този списък).

Освен това, всеки списък може да бъде конструиран от празния чрез многократно прилагане на ':' за добавяне на елементи и тази идея съответства в максимална степен на вътрешното представяне на списъците в Haskell.

Примери

[] 3:[] = [3] 2:[3] = [2,3] 4:[2,3] = [4,2,3]

При това последният списък може да бъде записан също като 4:2:3:[]

Забележка. Операторът ':' е **дясно асоциативен**, т.е. за всички стойности на x, y и zs е вярно $x:y:zs = x:(y:zs)$

Ще отбележим, че конструкцията `4:2:3:[]` определя *единствения начин*, по който списъкът `[4,2,3]` може да се конструира с използване на `[]` и `'.'`.

Следователно, операторът `'.'` от тип `a -> [a] -> [a]` играе изключително важна роля при работата със списъци – той е ***конструктор на списъци***, тъй като всеки списък може да бъде конструиран по единствен начин с използване на `[]` и `'.'`. По исторически причини (свързани с езика Lisp) този конструктор понякога се нарича **cons**.

Дефиниции на някои функции за работа със списъци
с използване на образци

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

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

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


Сега вече можем да обясним как изглеждат и каква е ролята на образците от тип конструктор (по-точно, образците – конструктори на списъци).

Образецът – конструктор на списъци може да бъде или [], или да има вида (p:ps), където p и ps също са образци.

Правилата за съпоставяне с такъв образец могат да бъдат формулирани по следния начин:

- даден списък е съпоставим с образаца [] точно когато е празен;
- даден списък е съпоставим с образец от вида (p:ps), когато не е празен, главата му се съпоставя успешно с образаца p и опашката му се съпоставя успешно с образаца ps.

В частност всеки непразен списък е съпоставим с образец от вида $(x:xs)$.

Забележка. Образец, който включва конструктор от вида $:$, като правило се загражда в кръгли скоби, тъй като прилагането на функция има по-висок приоритет от останалите операции.

Конструкцията case

Досега показахме как може да се описват различни случаи на съпоставяне на аргументите на дадена функция с определени образци в (чрез) поредица от равенства.

За целта може да се използва и конструкцията case.

Пример. Ще дефинираме функция, която връща като резултат първата цифра от даден низ.

```
firstDigit :: String -> Char
firstDigit str
  = case (digits str) of
      []      -> '\0'
      (x:_)  -> x
```

Забележки

1. ***digits*** е функцията, дефинирана в една от предишните лекции, която намира всички цифри в даден низ:

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

2. Специалният символ от вида '\n' означава знака с ASCII код n.

В горния пример изразът `case` е използван с цел да се разграничат двата основни случая - тези на празен и непразен списък, както и за да се извлекат (получат) части от търсената стойност чрез асоцииране (свързване) на променливи, участващи в образците, със съпоставените им стойности.

В общия случай изразът от тип **case** има следния вид:

```
case e of
  p1 -> e1
  p2 -> e2
  ...
  pk -> ek
```

Тук **e** е израз, който се съпоставя последователно с образците **p₁, p₂, ..., p_k**. Ако **p_i** е първият образец от поредицата, който е съпоставим с **e**, резултатът е **e_i**. При това променливите, които участват в **p_i**, се свързват със съответните им части от **e**.

Примитивна рекурсия върху списъци

Нека предположим, че имаме за задача да намерим сумата на елементите на даден списък от цели числа. Една възможна дефиниция на такава функция изглежда по следния начин:

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

Тази дефиниция е пример за реализация на т. нар. **примитивна рекурсия върху списъци**. В дефиниция, която реализира този тип рекурсия, се описват следните типове случаи:

- **начален (прост, базов) случай**: в горния пример тук се описва стойността на `sum` за `[]`;
- **общ случай**, при който се посочва връзката между стойността на функцията за дадена стойност на аргумента (в случая `sum (x:xs)`) и стойността на функцията за по-проста в определен смисъл стойност на аргумента (`sum xs`).

Общата схема на дефиниция на функция чрез примитивна рекурсия върху списъци е следната:

```
fun []      = ....  
fun (x:xs) = .... x .... xs .... fun xs ....
```

Основната задача, която трябва да се реши, е да се даде отговор на въпроса:

Как стойността на текущо дефинираната функция $f(x:xs)$ се получава от стойността на $f\ xs$?

Оценяване на обръщението към `sum [3,2,7,5]`:

`sum [3,2,7,5]`

→ `3 + sum [2,7,5]`

→ `3 + (2 + sum [7,5])`

→ `3 + (2 + (7 + sum [5]))`

→ `3 + (2 + (7 + (5 + sum [])))`

→ `3 + (2 + (7 + (5 + 0)))`

→ `17`

Други дефиниции на функции чрез примитивна рекурсия върху списъци

Пример 1. Примерна дефиниция (с учебна цел) на функцията `concat`, която е дефинирана в `Prelude.hs`:

$\text{concat } [e_1, e_2, \dots, e_n] = e_1 ++ e_2 ++ \dots ++ e_n$

Дефиниция

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

Пример 2. Примерна учебна дефиниция на функцията ++.

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

Пример 3. Функция, която проверява дали дадено цяло число (число от тип `Int`) съвпада с някой от елементите на даден списък от цели числа.

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

Важна забележка. Изглежда, че горната дефиниция би могла да се запише и по следния начин:

```
elem x []      = False
elem x (x:ys) = True
elem x (y:ys) = elem x ys
```

Тук проверката за равенство се извършва чрез повторно използване на променливата x (както това обикновено се прави при програмирането в логически стил на езика Prolog), но за съжаление **повторната употреба на променливи в образци от типа на предложената по-горе не е разрешена при програмирането на Haskell.**

Пример 4. Ще дефинираме функция, която удвоява елементите на даден списък от цели числа.

Първо решение

```
doubleAll :: [Int] -> [Int]
doubleAll xs = [2*x | x <- xs]
```

Второ решение

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

Пример 5. Ще дефинираме функция, която извлича четните числа измежду елементите на даден списък от цели числа (филтрира по четност елементите на даден списък от цели числа).

Първо решение

```
selectEven :: [Int] -> [Int]
selectEven xs = [x | x <- xs, isEven x]
```

Второ решение

```
selectEven :: [Int] -> [Int]
selectEven [] = []
selectEven (x:xs)
  | isEven x  = x : selectEven xs
  | otherwise =      selectEven xs
```

Пример 6. Сортиране във възходящ ред на даден списък от числа чрез вмъкване.

Дефиниция

```
iSort :: [Int] -> [Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

```
ins :: Int -> [Int] -> [Int]
ins x [] = [x]
ins x (y:ys)
  | x <= y      = x:(y:ys)
  | otherwise   = y : ins x ys
```


Оценяване на примерно обръщение към функцията iSort:

```
iSort [3,9,2]
→ ins 3 (iSort [9,2])
→ ins 3 (ins 9 (iSort [2]))
→ ins 3 (ins 9 (ins 2 (iSort [])))
→ ins 3 (ins 9 (ins 2 []))
→ ins 3 (ins 9 [2])
→ ins 3 (2 : ins 9 [])
→ ins 3 [2,9]
→ 2 : ins 3 [9]
→ 2 : [3,9]
→ [2,3,9]
```

Обща рекурсия върху списъци

Тук ще покажем някои примери на дефиниции на функции за работа със списъци, които не се подчиняват на ограниченията на примитивната рекурсия. При тях схемата на дефиниране е специфична и се подчинява на задачата да се даде отговор на въпроса:

Когато се дефинира $f(x:xs)$, кои стойности на $f ys$ биха помогнали за коректното описание на резултата?

Пример 1. Ще дефинираме функцията `zip`, която трансформира два списъка в списък от двойки от съответните елементи на двата списъка, например

`zip [1,3] [2,4] = [(1,2),(3,4)]`

`zip [1,2] ['a','b','c'] = [(1,'a'),(2,'b')]`

Ще опишем дефиницията на тази функция с рекурсия относно двата аргумента.

Първо решение

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

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
zip _ _ = []
```

Второ решение

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

Оценяване на примерно обръщение към функцията zip:

```
zip [1,5] ['c','d','e']
→ (1,'c') : zip [5] ['d','e']
→ (1,'c') : (5,'d') : zip [] ['e']
→ (1,'c') : (5,'d') : []
→ (1,'c') : [(5,'d')]
→ [(1,'c'),(5,'d')]
```

Пример 2. Функцията `take` ще връща като резултат списък от първите няколко елемента на даден списък, например

```
take 5 "Hot Rats" = "Hot R"  
take 15 "Hot Rats" = "Hot Rats"
```

И тук в дефиницията ще организираме рекурсия по отношение на двата аргумента.

Дефиниция

```
take :: Int -> [a] -> [a]  
take 0 _ = []  
take _ [] = []  
take n (x:xs)  
    | n>0 = x : take (n-1) xs  
take _ _ = error "take: negative argument"
```

Пример 3. Бърза сортировка. Алгоритъмът за бързо сортиране генерира две рекурсивни обръщания към себе си с аргументи – части от списъка, който е зададен като оригинален аргумент.

Дефиниция

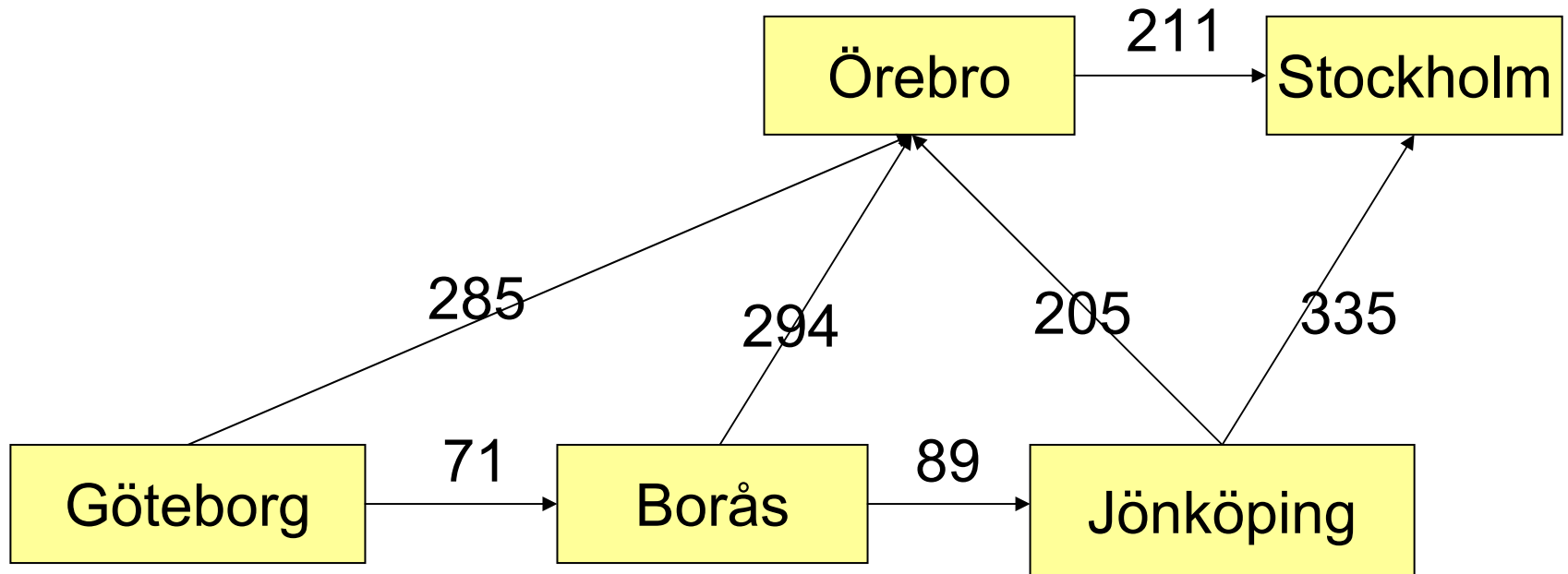
```
qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:xs)
    = qSort [ y | y <- xs, y<=x] ++ [x]
      ++ qSort [ y | y <- xs, y>x]
```

Лекция 6

Характерни приложения на списъците

Намиране на път върху
географска карта

Формулировка на задачата



Кой е най-краткият път от Göteborg до Stockholm?

Очакван резултат

```
> route "Göteborg" "Stockholm"  
  71  Borås  
 160  Jönköping  
 324  Norrköping  
 495  Stockholm
```

Данни от картата

Би трябвало данните от картата на пътищата да се съхраняват във **файл**. Засега вместо файл за целта ще използваме подходящ списък.

Директни пътни връзки:

Göteborg	Borås	71
Borås	Jönköping	89
Jönköping	Norrköping	164
Norrköping	Örebro	105
Norrköping	Uppsala	243
Norrköping	Västerås	153
Norrköping	Stockholm	171

...

План за решаване на задачата

Ще започнем с top-down проектиране:

- Въвеждане на имената на началната и крайната точка на пътуването.
- Въвеждане на данните от пътната карта.
- Намиране на най-късия път.
- Извеждане на резултата.

Главната програма

```
main :: IO ()  
main = do      (from, to) <- readArguments  
               roads <- readRoads  
               putStr (formatRoute (route roads from to))
```

Дефиниции на основните типове:

```
readArguments :: IO (Town, Town)
```

```
readRoads :: IO [Road]
```

```
route :: [Road] -> Town -> Town -> Route
```

```
formatRoute :: Route -> String
```

“Главната програма” се оформя като “команда” с името main.

Top-Down проектиране

Декомпозирахме задачата на подзадачи и написахме главната програма (the top-level program).

Трябва да се погрижим за решенията на отделните подзадачи.

Въвеждане на аргументите

Временно ще оставим настрана тази подзадача.

Ще работим върху останалите подзадачи, като смятаме, че имената на началната и крайната точка на пътуването са известни данни от тип Town:

```
type Town = String
```

Данни от пътната карта

Файлт с данните от картата на пътищата съдържа записи от вида:

Göteborg	Borås	71
Borås	Jönköping	89
Jönköping	Norrköping	164
Norrköping	Örebro	105
...		

В нашата програма ще представяме тези данни в удобен за манипулиране вид:

[("Göteborg", "Borås", 71), ("Borås", "Jönköping", 89), ...]

Типът Road

Една подходяща дефиниция на типа Road, който описва данните от пътната карта, изглежда както следва:

type Road = (Town, Town, Int)

Представяне на данните от пътната карта

Ще предпологаеме, че данните от пътната карта са достъпни под формата на списък с елементи от тип Road.

Намиране на пътищата между началната и крайната точка

Остава да бъдат дефинирани следните функции:

```
route :: [Road] -> Town -> Town -> Route  
formatRoute :: Route -> String
```

Представяне на пътищата

Какво е необходимо да се знае за един път?

- Началната му точка.
- Крайната му точка.
- Дължината му.
- През кои селища минава.

71	Borås
160	Jönköping
324	Norrköping
495	Stockholm

Представяне на път: първи опит

71	Borås
160	Jönköping
324	Norrköping
495	Stockholm

type Route = (Town, Town, Int, [(Town, Int)])

example = ("Göteborg", "Stockholm", 495,
[("Borås", 71), ("Jönköping", 160),
("Norrköping", 324), ("Stockholm", 495)])


Форматиране на път

Ще решим тази подзадача на стъпки:

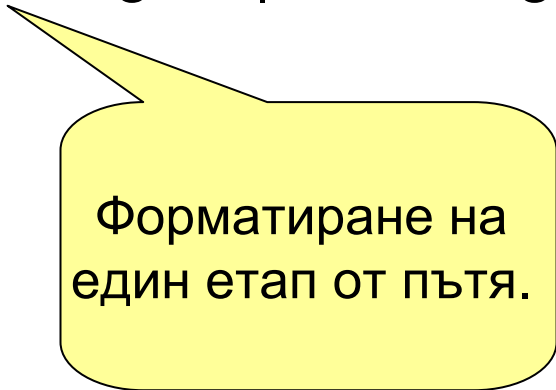
```
formatRoute :: Route -> String
```

```
formatRoute (from, to, dist, stages) =
```

```
  unlines [formatStage s | s <- stages]
```



Обединяване на
редове с line
Breaks.



Форматиране на
един етап от пътя.

Форматиране на етап от пътя

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

Първи опит:

```
formatStage (t, d) = show d ++ " " ++ t
```

Изход:

71	Borås
160	Jönköping
324	Norrköping
495	Stockholm

Допълване с интервали

Втори опит:

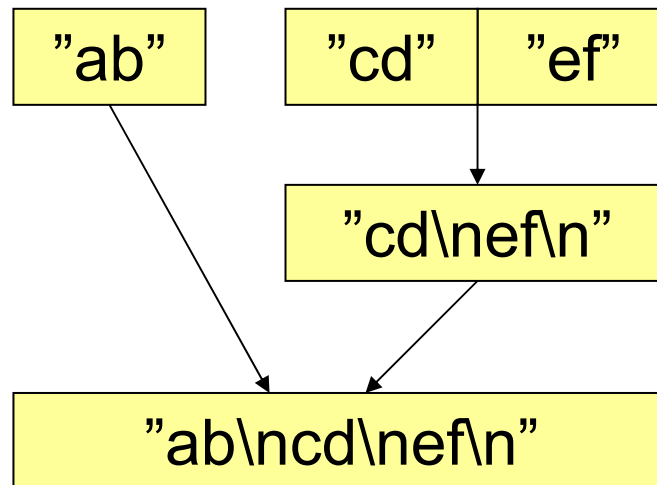
```
formatStage (t, d) = pad 5 (show d) ++ " " ++ t
```

```
pad :: Int -> String -> String
```

```
pad n s = [ ' ' | i <- [length s + 1..n] ] ++ s
```

Толкова интервали, колкото са
необходими за запълване на
празнината между `length s` и `n`.

Обединяване на редове



`unlines :: [String] -> String`

`unlines (x : xs) = x ++ "\n" ++ unlines xs`

`unlines [] = []`

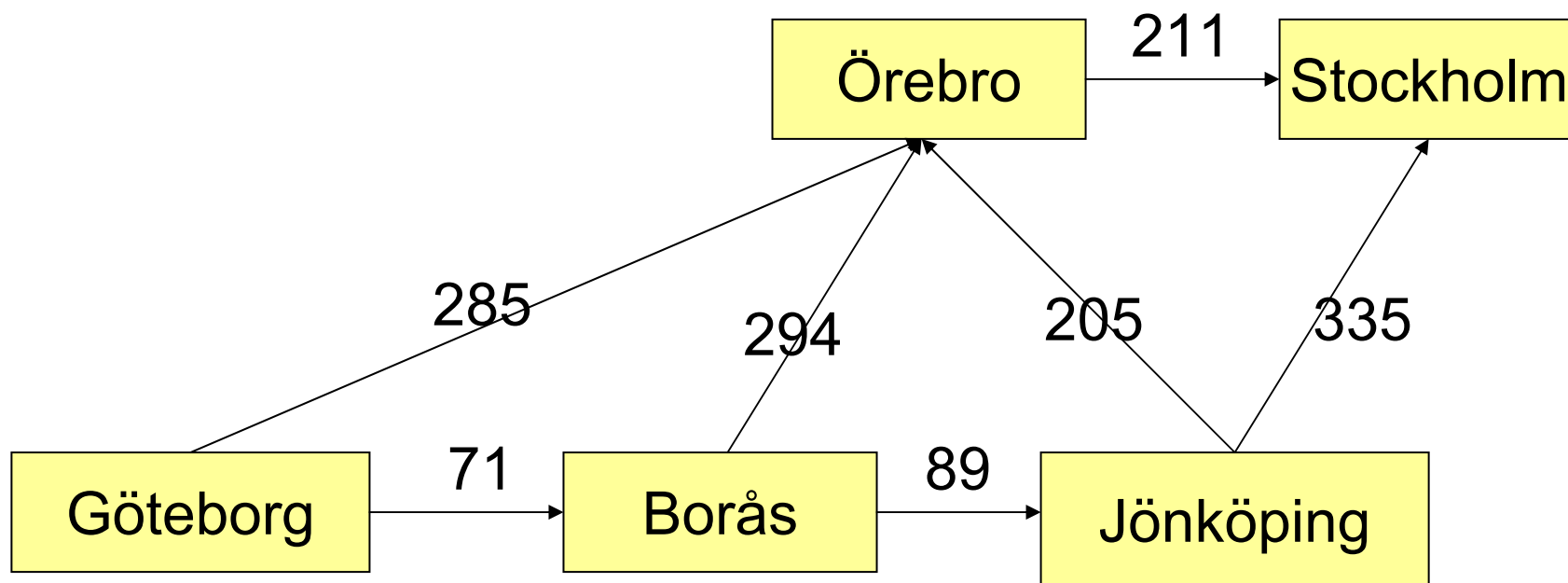
Стандартна
(вградена)
Функция.

Построяване на път

Остава да бъде дефинирана най-сложната функция:

```
route :: [Road] -> Town -> Town -> Route
```

Как може да се достигне от Göteborg до Stockholm?



Може да се намерят най-добрите (най-кратките) пътища от Örebro и Borås, да се прибави разстоянието от Göteborg и да се избере най-добрият получен път.

Намиране на по-добрия от два дадени пътя

Кой път е по-добър?

- По-късият път е по-добър от по-дългия.
- Ако два пътя имат една и съща дължина, по-добър е този, който включва повече етапи (тъй като той предоставя по-детайлно описание на избрания маршрут).

("Goteborg", "Stockholm", 496,
[("Orebro", 285), ("Stockholm", 496)])

е по-добър от

("Goteborg", "Stockholm", 496, [("Stockholm", 496)])

По-добър път

`better :: Route -> Route -> Route`

`better (from,to,dist,stgs) (from',to',dist',stgs')`

`| dist < dist' = (from,to,dist,stgs)`

`| dist > dist' = (from',to',dist',stgs')`

`| length stgs > length stgs' = (from,to,dist,stgs)`

`| length stgs <= length stgs' = (from',to',dist',stgs')`

`best :: [Route] -> Route`

`best [r] = r`

`best (r : rs) = better r (best rs)`

Избира най-добрия
елемент на дадено
множество от пътища.

Обединяване на пътища

Необходимо е да може да се обедини даден едностъпков път (например от Göteborg до Borås) с останалата част от пътя (с по-нататъшния път).

Как могат да се обединят два пътя?

- Първият път трябва да завърши там, откъдето започва вторият.
- Разстоянията се събират.
- Трябва да се коригират етапите на втория път чрез добавяне на дължината на първия път към техните разстояния.

Пример за обединяване на пътища

След обединяването на

("Goteborg", "Boras", 71, [("Boras", 71)])

и

("Boras", "Jonkoping", 89, [("Jonkoping", 89)])

се получава

("Goteborg", "Jonkoping", 160,
[("Boras", 71), ("Jonkoping", 160)])

Първият етап не се променя.

Вторият етап с коригирано разстояние.

joinRoutes

`joinRoutes :: Route -> Route -> Route`

`joinRoutes (from,to,dist,stgs) (from',to',dist',stgs')`

`| to==from' =`

`(from,to',dist+dist', stgs++[(c,d+dist)`

`| (c,d)<-stgs'])`

За всеки етап
от втория
път...

Коригиране на
разстоянието.

Съставяне на едноетапен (едностъпков) път

Необходимо е дадена отсечка от пътната карта да се конвертира в едноетапен път, с цел след това към нея да може да се приложи функцията `joinRoutes`.

type Road = (Town, Town, Int)

roadRoute :: Road -> Route

roadRoute (from,to,dist) = (from,to,dist,[(to,dist)])

Пример:

roadRoute ("Goteborg", "Boras", 71)

—————> ("Goteborg", "Boras", 71, [("Boras", 71)])

Пътищата са двупосочни

С цел да запазим програмата за намиране на пътища проста и разбираема, за всяка съществуваща отсечка от пътната карта (за всеки елемент на списъка, описващ данните от пътната карта) ще добавим и съответната отсечка в обратна посока:

```
bothWays :: [Road] -> [Road]
```

```
bothWays roads =
```

```
roads ++ [(to,from,dist) | (from,to,dist) <- roads]
```

Програма за намиране на пътища

```
route :: [Road] -> Town -> Town -> Route
```

```
route roads from to =
```

```
  best [joinRoutes (roadRoute road)
```

```
        (route roads (endPoint road) to)
```

```
      | road <- bothWays roads, startPoint road == from]
```

```
startPoint, endPoint :: Road -> Town
```

```
startPoint (from, to, dist) = from
```

```
endPoint (from, to, dist) = to
```

Полезни помощни
функции.

Базовият случай

Ако целта (Stockholm) вече е достигната, то пътят до нея е тривиален! Преди рекурсивния случай в дефиницията трябва да се добави:

`route roads from to | from == to = trivialRoute from`

Съставяне на тривиален път:

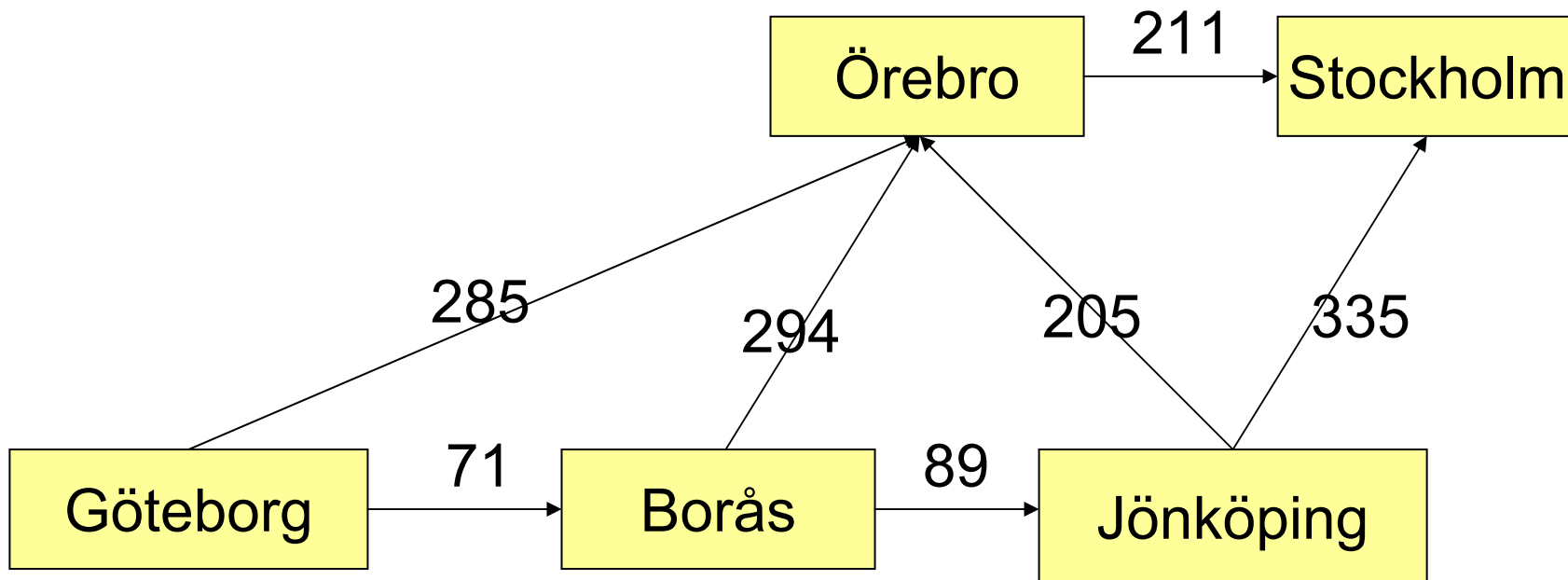
`trivialRoute :: Town -> Route`

`trivialRoute t = (t, t, 0, [])`

Проблем: зацикляне

Оказва се, че при опит за изпълнение на така дефинираната функция се получава зацикляне (Control stack overflow).

Избягване на цикли



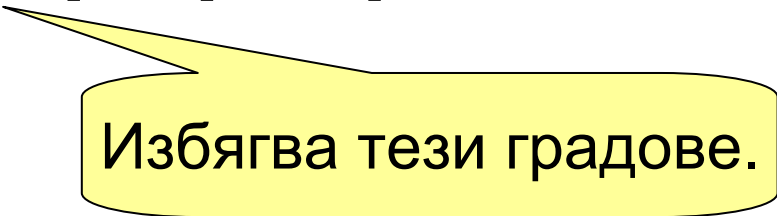
Да се намери най-добрия път от Borås до Stockholm, който не минава през (не включва) Göteborg.

Да се намери най-добрия път от Jönköping до Stockholm, който не минава през Göteborg и Borås.

Намиране на пътища, избягвайки зациклянето

Ще дефинираме

`routeAvoiding :: [Town] -> [Road] -> Town -> Town -> Route`



Избягва тези градове.

Тогава

`route roads from to = routeAvoiding [] roads from to`

routeAvoiding

routeAvoiding avoid roads from to

| from == to = trivialRoute from

| otherwise =

best [joinRoutes (roadRoute road)

(routeAvoiding

(from : avoid)

roads (endPoint road) to)

| road <- bothWays roads, startPoint road == from,

not (endPoint road `elem` avoid)]

Избягва се връщането във from.

Избягва се връщането в град, който е включен в пътя.

Нов проблем: best []

В някои случаи построяването на пътя не може да продължи, без да се наруши изискването за избягване на зациклянията.

В такива случаи търсеният път *не може да бъде намерен!*

Възможно решение: промяна на представянето на пътищата

Необходимо е да могат да се представят невалидните пътища. Нова дефиниция:

type Route = (Bool, ... както досега ...)



True, ако пътят е валиден.

invalidRoute :: Route

invalidRoute = (False, "", "", 0, [])

Коригиране на дефинициите на някои функции

`formatRoute (False, _, _, _, _) = "No valid route"`

`formatRoute (True, ...) = ...както досега...`

`better (True, ...) (True, ...) = ...както досега...`

`better (False, _, _, _, _) r = r`

`better r (False, _, _, _, _) = r`

Винаги се предпочита
валиден път.

`best [] = invalidRoute`

и т.н.

Анализ на решението

Нашата програма работи, но сравнително бавно. Добавянето на нови данни от пътната карта би забавило още повече нейната работа.

Необходими са подобрения!

Лекция 7, част 1

Характерни приложения на списъците

Текстообработка

Една от най-важните задачи при обработката на „чисти“ текстове (plain texts) е подреждането на текста в редове с определена дължина.

Първоначалният текст може да е разпокъсан или недобре подреден. Задачата е той да бъде разделен на редове със зададена дължина и след това евентуално да бъде подравнен чрез добавяне на интервали на подходящи места между думите.

Разделяне на входния текст на думи

Ще предполагаме, че текстът, който трябва да се форматира, е зададен под формата на символен низ.

Първата задача е входният низ да бъде разделен на думи.

Дума е всяка поредица от знакове, която не включва разделители (the whitespace characters space, tab and newline):

-- The "whitespace" characters.

```
whitespace :: String
whitespace = ['\n', '\t', ' ']
```

Най-напред ще дефинираме функцията `getWord`, която връща като резултат първата дума от даден низ, ако този низ започва с дума (а не с разделител), или празен низ – в противния случай.

Примери

`getWord " boo" → ""`

`getWord "cat dog" → "cat"`

В дефиницията на функцията `getWord` ще използваме стандартната (вградената) функция ***elem***, която проверява дали даден обект е елемент на даден списък:

`elem 'a' whitespace → False`

`elem ' ' whitespace → True`

Дефиниция:

```
-- Get a word from the front of a string.
```

```
getWord :: String -> String
```

```
getWord [] = []
```

```
getWord (x:xs)
```

```
    | elem x whitespace = []
```

```
    | otherwise         = x : getWord xs
```


Проследяване на изпълнението на примерно обръщение към функцията `getWord`:

`getword "cat dog"`

—→ `'c' : getword "at dog"`

—→ `'c' : 'a' : getword "t dog"`

—→ `'c' : 'a' : 't' : getword " dog"`

—→ `'c' : 'a' : 't' : []`

—→ `"cat"`

По сходен начин ще дефинираме функция, която отделя („изтрива”) първата дума от даден символен низ (в случай, че този низ започва с дума, а не с разделител):

```
dropWord :: String -> String
dropWord []      = []
dropWord (x:xs)
  | elem x whitespace = (x:xs)
  | otherwise         = dropWord xs
```

Примери

`dropWord "cat dog" → " dog"`

`dropWord " dog" → " dog"`

Отделянето на водещите разделители от даден текст (низ) ще може да се извършва с помощта на функцията `dropSpace`:

```
-- To remove the whitespace character(s) from the  
-- front of a string.
```

```
dropSpace :: String -> String  
dropSpace []      = []  
dropSpace (x:xs)  
  | elem x whitespace = dropSpace xs  
  | otherwise         = (x:xs)
```

Нека предположим, че е даден символен низ ***st***, който не съдържа разделители в началото си.

Тогава разделянето на ***st*** на (списък от съдържащите се в него) думи може да се извърши по следния начин:

- първата дума може да се отдели чрез прилагане на `getWord` към `st`;
- останалите думи могат да бъдат получени чрез отделяне на поредицата от разделители в началото на останалата (след отделянето на първата дума) част на `st`, последвано от прилагане на описваната операция.

Дефиниция:

```
-- A word is a string.
```

```
type Word = String
```

```
-- Splitting a string into words.
```

```
splitWords :: String -> [Word]  
splitWords st = split (dropSpace st)
```

```
split :: String -> [Word]  
split [] = []  
split st  
    = (getWord st) : split (dropSpace (dropWord st))
```

Пример

splitWords " dog cat"

→ split "dog cat"

→ (getWord "dog cat")

 : split (dropSpace (dropWord "dog cat"))

→ "dog" : split (dropSpace " cat")

→ "dog" : split "cat"

→ "dog" : (getWord "cat")

 : split (dropSpace (dropWord "cat"))

→ "dog" : "cat" : split (dropSpace [])

→ "dog" : "cat" : split []

→ "dog" : "cat" : []

→ ["dog" , "cat"]

Разделяне на текста на редове

Сега ще покажем как даден текст (по-точно, даден списък от думи) може да бъде разделен на редове с дължина, не по-голяма от `lineLen`:

```
lineLen :: Int  
lineLen = 80
```

```
-- A line is a list of words.
```

```
type Line = [Word]
```


Функцията `getLine` извлича първия ред от даден списък от думи:

- ако списъкът от наличните думи е празен, то се формира и връща като резултат празен ред;
- ако първата налична дума е `w`, тя се включва в реда, ако има достатъчно място за нея (нейната дължина трябва да бъде не по-голяма от дължината на реда). В такъв случай остатъкът от реда се запълва от (част от) оставащите думи, като се предвиди място за поне един интервал след първата дума;
- ако първата дума не може да се побере в реда, то редът остава празен.

Дефиниция:

```
-- Getting a line from a list of words.
```

```
getLine :: Int -> [Word] -> Line
```

```
getLine len [] = []
```

```
getLine len (w:ws)
```

```
    | length w <= len = w : restOfLine
```

```
    | otherwise      = []
```

```
    where
```

```
        newlen = len - (length w + 1)
```

```
        restOfLine = getLine newlen ws
```

Пример

getLine 20 ["Mary", "Poppins", "looks", "like", ...]

→ "Mary" : getLine 15 ["Poppins", "looks", "like", ...]

→ "Mary" : "Poppins" : getLine 7 ["looks", "like", ...]

→ "Mary" : "Poppins" : "looks" : getLine 1 ["like", ...]

→ "Mary" : "Poppins" : "looks" : []

→ ["Mary", "Poppins", "looks"]

По аналогия с функцията `dropWord` може да се дефинира и функция `dropLine`, която отделя (“изтрива”) първия ред от даден списък от думи.

```
-- Dropping the first line from a list of words.
```

```
dropLine :: Int -> [Word] -> Line
```

```
-- dropLine = .....
```

Тогава функцията `splitLines`, която разделя даден списък от думи на редове с дължина най-много `lineLen`, може да бъде дефинирана както следва:

```
-- Splitting into lines.
```

```
splitLines :: [Word] -> [Line]
splitLines [] = []
splitLines ws
    = getLine lineLen ws
      : splitLines (dropLine lineLen ws)
```

Заклучение

Разделянето на даден текст (символен низ) на редове може да се извърши с помощта на функцията `fill`:

```
-- To fill a text string into lines, we write
```

```
fill :: String -> [Line]
fill = splitLines . splitWords
```

Конвертирането на резултата в подходящ символен низ може да се извърши с помощта на подходяща функция, например

```
joinLines :: [Line] -> String
```

Освен тази функция следва да се дефинира също така и функция, която извършва подравняването на редовете (добавянето на допълнителни интервали между думите с цел подравняване на текста от дясно).

Лекция 7, част 2

Изследване на свойствата на програми на Haskell

Едно от най-големите предимства на програмирането във функционален стил е възможността строго да се доказват свойства на функционалните програми.

Тук ще покажем един прост подход за доказване на това, че определено множество от функции е дефинирано така, че дадено свойство, във формулировката на което участват тези функции, е вярно за всички (крайни) списъци.

Принцип на структурната индукция при работата със списъци

В случай, че трябва да се докаже, че дадено свойство $P(xs)$ е вярно за всички крайни списъци xs , доказателството може да се извърши на две стъпки:

- **Базов случай.** Доказване, че е вярно $P([])$.
- **Индуктивна стъпка.** Доказване, че е вярно $P(x:xs)$ при предположение, че е вярно $P(xs)$.

Твърдението $P(xs)$ във втората стъпка се нарича **индуктивна хипотеза**, тъй като за него се предполага, че е вярно в процеса на доказателството на $P(x:xs)$.

Пример 1. Нека `sum` и `doubleAll` са функции, дефинирани както следва:

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

```
doubleAll :: [Int] -> [Int]
doubleAll []      = []
doubleAll (z:zs)  = 2*z : doubleAll zs
```

Ще покажем, че за всеки списък `xs` е в сила $\text{sum} (\text{doubleAll } xs) = 2 * \text{sum } xs$.

Доказателството ще извършим на две стъпки:

1. Базов случай: трябва да покажем, че $\text{sum} (\text{doubleAll []}) = 2 * \text{sum []}$.
2. Индуктивна стъпка: ще покажем, че е в сила $\text{sum} (\text{doubleAll (x:xs)}) = 2 * \text{sum (x:xs)}$, използвайки за целта индуктивната хипотеза $\text{sum} (\text{doubleAll xs}) = 2 * \text{sum xs}$.

Базов случай

$\text{sum} (\text{doubleAll []}) = \text{sum []} = 0;$
 $2 * \text{sum []} = 2 * 0 = 0.$

Следователно, $\text{sum} (\text{doubleAll []}) = 2 * \text{sum []}$.

Индуктивна стъпка

$$\begin{aligned}\text{sum} (\text{doubleAll} (x:xs)) &= \text{sum} (2*x : \text{doubleAll} xs) \\ &= 2*x + \text{sum} (\text{doubleAll} xs); \\ 2 * \text{sum} (x:xs) &= 2 * (x + \text{sum} xs) = 2*x + 2*\text{sum} xs.\end{aligned}$$

Според индуктивното предположение е в сила
 $\text{sum} (\text{doubleAll} xs) = 2*\text{sum} xs$, следователно

$$\text{sum} (\text{doubleAll} (x:xs)) = 2*\text{sum} (x:xs) .$$

Пример 2. Връзка между length и ++.

Нека length и ++ са функциите за намиране на дължината на списък и конкатенация на списъци, дефинирани както следва:

```
length :: [a] -> Int
length []      = 0
length (z:zs) = 1 + length zs
```

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

Ще покажем, че за всеки два списъка xs и ys е в сила $\text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$.

Доказателство

Базов случай

Ще покажем, че $\text{length } ([] ++ ys) = \text{length } [] + \text{length } ys$.

$\text{length } ([] ++ ys) = \text{length } ys$;

$\text{length } [] + \text{length } ys = 0 + \text{length } ys = \text{length } ys$.

Следователно $\text{length } ([] ++ ys) = \text{length } [] + \text{length } ys$.

Индуктивна стъпка

Ще покажем, че е вярно

$\text{length } ((x:xs) ++ ys) = \text{length } (x:xs) + \text{length } ys$

при условие, че съгласно индуктивното предположение е вярно

$\text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$.

$$\begin{aligned} \text{length } ((x:xs) ++ ys) &= \text{length } (x:(xs ++ ys)) \\ &= 1 + \text{length } (xs ++ ys) = 1 + \text{length } xs + \text{length } ys; \end{aligned}$$
$$\text{length } (x:xs) + \text{length } ys = 1 + \text{length } xs + \text{length } ys.$$

Следовательно

$$\text{length } ((x:xs) ++ ys) = \text{length } (x:xs) + \text{length } ys .$$

Пример 3. Връзка между reverse и ++.

Нека reverse е функцията, която обръща реда на елементите на даден списък, дефинирана по следния начин:

```
reverse :: [a] -> [a]
reverse []      = []
reverse (z:zs) = reverse zs ++ [z]
```

Ще покажем, че

$\text{reverse } (xs ++ ys) = \text{reverse } ys ++ \text{reverse } xs$.

Доказателство

Базов случай

$\text{reverse } ([] ++ ys) = \text{reverse } ys;$

$\text{reverse } ys ++ \text{reverse } [] = \text{reverse } ys ++ [] = \text{reverse } ys.$

Следователно

$\text{reverse } ([] ++ ys) = \text{reverse } ys ++ \text{reverse } [].$

Забележка. Горното доказателство ще е коректно, ако докажем, че добавянето чрез конкатенация ($++$) на празен списък към даден друг списък е операция – идентитет, т.е.

$xs ++ [] = xs$ за всеки списък xs .

Доказателството на последното твърдение може лесно да бъде извършено по индукция (използвайки принципа на структурната индукция).

Индуктивна стъпка

$\text{reverse } ((x:xs) ++ ys) = \text{reverse } (x:(xs ++ ys))$
 $= \text{reverse } (xs ++ ys) ++ [x].$

Според индуктивното предположение е вярно
 $\text{reverse } (xs ++ ys) = \text{reverse } ys ++ \text{reverse } xs.$

Следователно

$\text{reverse } ((x:xs) ++ ys) = (\text{reverse } ys ++ \text{reverse } xs) ++ [x].$

От друга страна

$\text{reverse } ys ++ \text{reverse } (x:xs) = \text{reverse } ys ++ (\text{reverse } xs ++ [x]).$

Тъй като операцията конкатенация на списъци ($++$) е асоциативна, т.е. за всеки три списъка xs , ys и zs е изпълнено $xs ++ (ys ++ zs) = (xs ++ ys) ++ zs$, то следователно
 $\text{reverse } ((x:xs) ++ ys) = \text{reverse } ys ++ \text{reverse } (x:xs).$

Забележки

1. Асоциативността на операцията '++' подлежи на строго доказателство (такова може лесно да се направи отново с използване на принципа на структурната индукция).
2. Принципът на структурната индукция е подходящ за доказване на свойства на функции, дефинирани с помощта на примитивна рекурсия.

Лекция 8

Функции от по-висок ред в Haskell

Шаблони на пресмятания със списъци (Patterns of Computation over Lists)

Голяма част от функциите за работа със списъци, които разгледахме досега, могат да бъдат отнесени към малък брой типове. При това за съставянето на тези типове функции могат да бъдат използвани съответни ***шаблони на пресмятания (patterns of computation)***.

Примери за шаблони на пресмятания (програмни шаблони)

- Прилагане на една и съща операция върху всички елементи на даден списък (*mapping*)

Примери:

- Реализацията на хоризонталното завъртане flipH (в системата за манипулиране на картинки)
- Извличането на вторите елементи на двойките от даден списък (при работата с библиотечната “база от данни”)
- Конвертирането на списъка от бар кодове в списък от вида (Name, Price) (в примера за конструиране на касова бележка)

- Извличане на елементите на даден списък, които удовлетворяват дадено условие (***filtering***)

Примери:

- Извличането на тези двойки, първият елемент на които съвпада с дадено име (при работата с библиотечната “база от данни”)
- Извличането на цифрите/буквите от даден символен низ

- Комбиниране / акумулиране на елементите на даден списък (***folding***)

Примери:

- Конкатенацията на елементите на даден списък от списъци
- Събирането / умножаването на елементите на даден списък от числа

Реализацията на тези и други шаблони на пресмятания (patterns of computation) може да се извърши с помощта на подходящи **функции от по-висок ред**.

Дефиниция

Функция от по-висок ред се нарича всяка функция, която получава поне една функция като параметър (аргумент) или връща функция като резултат.

Наличието на средства за дефиниране и използване на функции от по-висок ред съществено увеличава изразителната сила на съответния език за програмиране.

Функциите като параметри

Тук ще покажем как могат да се дефинират някои често използвани функции от по-висок ред за работа със списъци. Тези функции са дефинирани в `Prelude.hs`, т.е. нашите дефиниции ще бъдат направени само с учебна цел.

Прилагане на дадена функция към всички елементи на даден списък (map)

Декларация на типа:

`map :: (a -> b) -> [a] -> [b]`

input function input list output list

Дефиниция (първи вариант):

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

Дефиниция (втори вариант):

```
map f []      = []  
map f (x:xs) = f x : map f xs
```

Примери

```
doubleAll :: [Int] -> [Int]
-- Удвоява елементите на даден списък.
doubleAll xs = map double xs
  where
    double :: Int -> Int
    double x = 2*x

convertChrs :: [Char] -> [Int]
-- Конвертира знаковете от даден списък
-- в техните ASCII кодове.
convertChrs xs = map ord xs
```

```
type Picture = [String]
flipH :: Picture -> Picture
-- Завърта дадена картинка около ординатната ос
-- ("завърта" хоризонталните координати на точките
-- от картинката).
flipH pic = map reverse pic
```


Филтриране на елементите на даден списък (filter)

Декларация на типа:

`filter :: (a -> Bool) -> [a] -> [a]`

input property input list output list

Дефиниция (първи вариант):

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

Дефиниция (втори вариант):

```
filter p [] = []  
filter p (x:xs)  
  | p x      = x : filter p xs  
  | otherwise =      filter p xs
```

Примери

```
isEven :: Int -> Bool  
isEven n = (n `mod` 2 == 0)
```

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

```
filter isEven [2,3,4,5] → [2,4]  
filter isSorted [[2,3,4,5],[3,2,5],[],[3]]  
→ [[2,3,4,5],[],[3]]
```

Комбиниране на zip и map (zipWith)

Вече разгледахме полиморфичната функция

`zip :: [a] -> [b] -> [(a,b)]`, която комбинира два дадени списъка в списък от двойки от съответните елементи на тези списъци.

Ще дефинираме нова функция, `zipWith`, която комбинира ефекта от действието на `zip` и `map`.

Декларация на типа

Функцията `zipWith` ще има три аргумента: една функция и два списъка. Двата списъка (вторият и третият аргумент на `zipWith`) са от произволни типове (съответно `[a]` и `[b]`). Резултатът също е списък от произволен тип (`[c]`). Първият аргумент на `zipWith` е функция, която се прилага върху аргументи – съответните елементи на двата списъка и връща съответния елемент на резултата, т.е. това е функция от тип `a -> b -> c`.

Следователно

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

Дефиниция

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []
```

Примери

Нека `plus` и `mult` са функции, дефинирани както следва:

```
plus :: Int -> Int -> Int  
plus a b = a+b
```

```
mult :: Int -> Int -> Int  
mult a b = a*b
```

Тогава

```
zipWith plus [1,2,3] [4,5,6] → [5,7,9]  
zipWith mult [1,2,3,4,5] [6,7,8] → [6,14,24]
```

Нека се върнем още един път на примерната система за манипулиране на картинки. В нея дефинирахме функцията

```
sideBySide :: Picture -> Picture -> Picture
sideBySide p q = [pline ++ qline
                  | (pline,qline) <- zip p q]
```

Нова дефиниция на функцията sideBySide:

```
sideBySide pic1 pic2 = zipWith (++) pic1 pic2
```

Забележка. Функциите map, filter и zipWith са дефинирани в Prelude.hs.

Комбиниране/акумулиране на елементите на даден списък (foldr1 и foldr)

Тук ще разгледаме група функции от по-висок ред, които реализират операцията *комбиниране* (*акумулиране*) на елементите на даден списък, използвайки подходяща функция.

Тази операция е достатъчно обща и се реализира от множество стандартни функции, включени в Prelude.hs.

Действие на функцията foldr1

Дефиницията на тази функция включва два случая:

- foldr1, приложена върху дадена функция f и списък от един елемент $[a]$, връща като резултат a ;
- Прилагането на foldr1 върху функция и по-дълъг списък е еквивалентно на

$$\begin{aligned} & \text{foldr1 } f [e_1, e_2, \dots, e_k] \\ &= e_1 \text{ `f` } (e_2 \text{ `f` } (\dots \text{ `f` } e_k) \dots) \\ &= e_1 \text{ `f` } (\text{foldr1 } f [e_2, \dots, e_k]) \\ &= f \ e_1 (\text{foldr1 } f [e_2, \dots, e_k]) \end{aligned}$$

Съответната дефиниция на Haskell изглежда както следва:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Примери

`foldr1 (+) [3,98,1] = 102`

`foldr1 (||) [False,True,False] = True`

`foldr1 min [6] = 6`

`foldr1 (*) [1 .. 6] = 720`

Забележка. Функцията `foldr1` не е дефинирана върху втори аргумент – празен списък.

Разглежданата функция може да бъде модифицирана така, че да получава един допълнителен аргумент, който определя стойността, която следва да се върне при опит за комбиниране по зададеното правило на елементите на празния списък.

Новополучената функция се нарича ***foldr*** (което означава ***fold***, т.е. комбиниране/акумулиране, извършено чрез групиране от дясно – bracketing to the right).

Дефиниция на функцията foldr:

`foldr :: (a -> a -> a) -> a -> [a] -> a`

binary operation
over type a

starting value
of type a

list of values
to be combined

the result
of type a

`foldr f s [] = s`

`foldr f s (x:xs) = f x (foldr f s xs)`

Примери

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

```
and :: [Bool] -> Bool  
and bs = foldr (&&) True bs
```

Забележка. В действителност типът на функцията `foldr` е по-общ:

`foldr :: (a -> b -> b) -> b -> [a] -> b`

Функциите като върнати стойности

Дефиниции на функции на функционално ниво

Дефинирането на някаква функция на функционално ниво предполага действието на тази функция да се опише не в термините на резултата, който връща тя при прилагане към подходящо множество от аргументи, а като директно се посочи връзката ѝ с други функции.

Например, ако вече са дефинирани функциите

$f :: b \rightarrow c$ и

$g :: a \rightarrow b$, то

тяхната композиция $f . g$ (т.е. функцията, за която е изпълнено $(f . g) x = f (g x)$ за всяко x от тип a) може да се дефинира с използване на вградения оператор $'.'$ от тип

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$.

Пример 1. Двукратно прилагане на функция.

$twice :: (a \rightarrow a) \rightarrow (a \rightarrow a)$

$twice f = (f . f)$

Нека например `succ` е функция, която прибавя 1 към дадено цяло число:

```
succ :: Int -> Int  
succ n = n + 1
```

Тогава

```
(twice succ) 12  
→ (succ . succ) 12  
→ succ (succ 12)  
→ 14
```

Пример 2. n-кратно прилагане на функция.

```
iter :: Int -> (a -> a) -> (a -> a)
iter n f
  | n>0      = f . iter (n-1) f
  | otherwise = id
```

Тук id е вградената функция – идентитет.

Дефиниране на функция, която връща функция като резултат

Пример. Функция, която за дадено цяло число n връща като резултат функция на един аргумент, която прибавя n към аргумента си.

```
addNum :: Int -> (Int -> Int)
addNum n = addN
    where
        addN m = n+m
```

Така оценка на обръщението към функцията `addNum` ще бъде функцията с име `addN`. От своя страна функцията `addN` е дефинирана в клаузата `where`.

Използваният подход може да бъде окачествен като индиректен: най-напред посочваме името на функцията – резултат и едва след това дефинираме тази функция.

Ламбда нотация (ламбда изрази)

Вместо да именуваме и да дефинираме някаква функция, която бихме искали да използваме, можем да запишем директно тази функция.

Например в случая на дефиницията на `addNum` резултатът може да бъде дефиниран като

`\m -> n+m`

В Haskell изрази от този вид се наричат **ламбда изрази**, а функциите, дефинирани чрез ламбда изрази, се наричат **анонимни функции**.

Забележка. Символът “\” е избран за означаване на ламбда изразите, защото той наподобява на гръцката буква λ .

Дефиницията на функцията `addNum` с използване на ламбда израз придобива вида

```
addNum n = (\m -> n+m)
```

Частично прилагане на функции

Нека разгледаме като пример функцията за умножение на две числа, дефинирана както следва:

```
multiply :: Int -> Int -> Int  
multiply x y = x*y
```

Ако тази функция бъде приложена към два аргумента, като резултат ще се получи число, например `multiply 2 3` връща резултат 6.

Какво ще се случи, ако multiply се приложи към един аргумент, например числото 2?

Отговорът е, че като резултат ще се получи функция на един аргумент u , която удвоява аргумента си, т.е. връща резултат $2*u$.

Следователно, **всяка функция на два или повече аргумента може да бъде приложена частично към по-малък брой аргументи**. Тази идея дава богати възможности за конструиране на функции като оценки на обръщения към други функции.

Пример. Функцията `doubleAll`, която удвоява всички елементи на даден списък от цели числа, може да бъде дефинирана както следва:

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

Тип на резултата от частично прилагане на функция

Правило на изключването

Ако дадена функция f е от тип

$$t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$$

и тази функция е приложена към аргументи

$$e_1 :: t_1, e_2 :: t_2, \dots, e_k :: t_k \text{ (където } k \leq n),$$

то типът на резултата се определя чрез изключване на типовете t_1, t_2, \dots, t_k :

$$\cancel{t_1} \rightarrow \cancel{t_2} \rightarrow \dots \rightarrow \cancel{t_k} \rightarrow t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t,$$

т.е. резултатът е от тип

$$t_{k+1} \rightarrow t_{k+2} \rightarrow \dots \rightarrow t_n \rightarrow t.$$

Примери

```
multiply 2 :: Int -> Int  
multiply 2 3 :: Int
```

```
doubleAll :: [Int] -> [Int]  
doubleAll [2,3] :: [Int]
```

Забележки

1. Прилагането на функция е **ляво асоциативна операция**, т.е.

$$f\ x\ y = (f\ x)\ y \quad \text{и}$$

$$f\ x\ y \neq f\ (x\ y)$$

2. Операторът ' \rightarrow ' не е асоциативен.

Например записите

$$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \text{и}$$

$$g :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$$

означават функции от различни типове.

Частичното прилагане на функции налага нова гледна точка върху понятието “брой на аргументите на дадена функция”. От тази гледна точка може да се каже, че всички функции в Haskell имат по един аргумент. Ако резултатът от прилагането на функцията върху даден аргумент е функция, тази функция може отново да бъде приложена върху един аргумент и т.н.

Сечения на оператори (operator sections)

Операторите в Haskell могат да бъдат прилагани частично, като за целта се задава това, което е известно, под формата на т. нар. **сечения на оператори** (*operator sections*).

Примери

- (+2) Функцията, която прибавя числото 2 към аргумента си.
- (2+) Функцията, която прибавя числото 2 към аргумента си.
- (>2) Функцията, която проверява дали дадено число е по-голямо от 2.
- (3:) Функцията, която поставя числото 3 в началото на даден списък.

(++"n")

Функцията, която поставя *newline* в края на даден низ.

("n"++)

Функцията, която поставя *newline* в началото на даден низ.

Общото правило гласи, че сечението на оператора **ор** “добавя” аргумента си по начин, който завършва от синтактична гледна точка записа на приложението на оператора (обръщението към оператора).

С други думи,

$(\text{ор } x) y = y \text{ ор } x$

$(x \text{ ор}) y = x \text{ ор } y$

Когато бъде комбинирана с функции от по-висок ред, нотацията на сечението на оператори е едновременно мощна и елегантна. Тя позволява да се дефинират разнообразни функции от по-висок ред.

Например,
`filter (>0) . map (+1)`

е функцията, която прибавя 1 към всеки от елементите на даден списък, след което премахва тези елементи на получения списък, които не са положителни числа.

Лекция 9

Структури от данни в програмирането

Основни дефиниции

Най-общо, под **структура от данни** се разбира организирана информация, която може да бъде описана, създадена и обработена с помощта на компютърна програма (Майер и Бодуен).

Често (макар и не съвсем точно) като синоним на понятието структура от данни се използва терминът **информационна структура**.

За да се определи една структура от данни, е необходимо да се зададат:

- **логическото описание на структурата**, което описва тази структура на базата на декомпозицията ѝ на по-прости структури (компоненти), основните операции над структурата и декомпозицията на основните операции на по-прости операции;
- **физическото представяне на структурата**, което дава методи за представяне на тази структура в паметта на компютъра.

Важна операция над коя да е структура от данни е операцията **достъп до компонентите на структурата**. Тази операция е тясно свързана с физическото представяне на структурата от данни. На базата на тази операция структурите от данни се класифицират като ***прости*** и ***съставни***.

Прости са тези структури от данни, за които операцията **достъп** се осъществява до структурата като цяло. Такива структури са числата, символните (знаковите) и булевите данни. Обикновено на всяка от тези структури в езиците за програмиране съответстват вградени типове данни.

Съставни са тези структури от данни, за които операцията **достъп** се осъществява до компонентите (елементите) на структурата, а не до структурата като цяло. Такива структури са масивът, записът, списъкът (т. нар. свързан списък), опашката, стекът и др.

Съставните структури от данни се делят на **статични** и **динамични**.

Съставна структура от данни, която се състои от фиксиран брой елементи и за която не са допустими операциите включване и изключване на елемент, се нарича **статична**. За статичните структури от данни в паметта на компютъра се отделя фиксирано количество памет.

Такива структури са **масивът** и **записът**.

Съставна структура от данни, която се състои от променлив брой елементи и за която са допустими операциите включване и изключване на елемент, се нарича **динамична**.

Такива структури са **стекът, опашката, дървото, графът** и др.

Не е целесъобразно от практическа гледна точка в език за програмиране с общо предназначение да се поддържат вградени типове данни за всяка от динамичните структури.

Структура от данни масив

Логическо описание на масив

Масивът е **крайна редица** от фиксиран брой елементи от един и същ тип. Към всеки елемент от редицата е възможен пряк достъп, който се осъществява чрез ***индекс***. Операциите включване и изключване на елемент от масив са недопустими, т.е. масивът е **статична структура**.

Физическо представяне

Елементите на масива се записват последователно в оперативната памет. За всеки елемент от редицата се отделя фиксирано количество памет.

Структура от данни запис

Логическо описание

Записът е **крайна редица** от фиксиран брой елементи, които могат да са от различни типове. Възможен е пряк достъп до всеки елемент от редицата, който се осъществява чрез **име**.

Елементите на редица, представляваща запис, се наричат **полета на записа**. Операциите включване и изключване на елемент са недопустими, т.е. структурата е **статична**.

Физическо представяне

Полетата на записа се записват последователно в паметта на компютъра.

Структура от данни множество

Логическо описание

Множеството е **съвкупност** от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими. Възможен е достъп както до отделните компоненти на множеството, така и до структурата като цяло. Достъпът е пряк.

Физическо представяне

Използва се последователно представяне на структурата в паметта. За целта за всеки допустим елемент се посочва дали принадлежи или не принадлежи на множеството.

Структура от данни свързан списък

Логическо описание

Свързаният списък е **крайна редица** от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими в произволно място на редицата. Възможен е достъп до всеки елемент на списъка, като достъпът до първия елемент е пряк, а до останалите елементи – последователен.

Физическо представяне

Тъй като операциите включване и изключване на елемент са възможни на произволно място в списъка, най-естествено е свързаното физическо представяне на списък. Съществуват различни форми на свързано представяне. Най-често се използват:

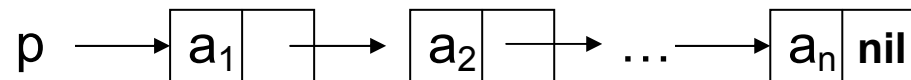
- свързано представяне с една връзка;
- свързано представяне с две връзки.

За свързания списък

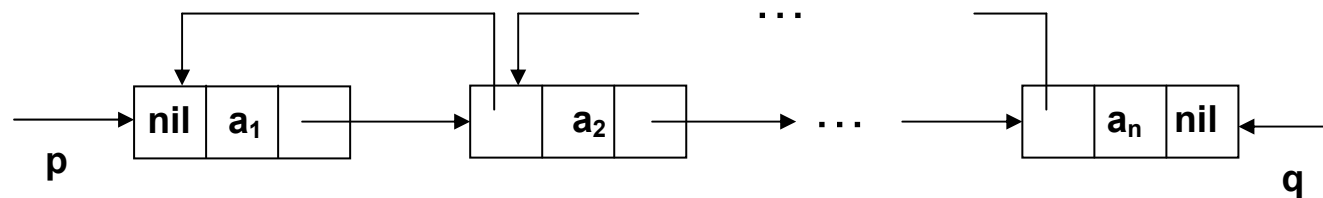
a_1, a_2, \dots, a_n

тези представяния имат следния вид:

Свързано представяне с една връзка



Свързано представяне с две връзки



Структура от данни стек

Логическо описание

Стекът е **крайна редица** от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими само за единия край на редицата, който се нарича ***връх на стека***. Възможен е достъп само до елемента, намиращ се на върха на стека, при това достъпът е ***пряк***.

Примери

Ако редицата a_1, a_2, \dots, a_n е стек с връх a_1 , включването на елемент a от тип, съвпадащ с типа на елементите на стека, води до получаването на нов стек, на върха на който е елементът a , т.е. a, a_1, a_2, \dots, a_n .

Ако редицата a_1, a_2, \dots, a_n е стек с връх a_1 , изключването на елемент (елемента от върха на стека) води до получаването на стека a_2, \dots, a_n .

При тази организация на логическите операции, последният включен в стека елемент се изключва първи. Затова стекът се определя още като структура “последен влязъл – пръв излязъл” (last in – first out, LIFO).

Физическо представяне

Широко се използват два основни начина за физическо представяне на стек: последователно и свързано.

При последователното представяне се запазва блок от паметта, вътре в който стекът да расте и да се съкращава.

Свързаното представяне на стека е аналогично на свързаното представяне на списък с една връзка.

Структура от данни опашка

Логическо описание

Опашката е **крайна редица** от елементи от един и същ тип. Операцията *включване на елемент* е допустима само за единия (например десния) край на редицата, който се нарича **край на опашката**. Операцията *изключване на елемент* е допустима само за другия (левия) край на редицата, който се нарича **начало на опашката**. Възможен е пряк достъп само до елемента, намиращ се в началото на опашката.

При описаната организация на логическите операции, последният включен в опашката елемент се изключва последен, а първият – първи. Затова опашката се определя още като структура от данни “първ влязъл – първ излязъл” (first in – first out, FIFO).

Физическо представяне

Аналогично на стека, при опашката се използват два основни начина за физическо представяне: последователно и свързано.

При последователното представяне се запазва блок от паметта, вътре в който опашката да расте и да се съкращава.

Свързаното представяне на една опашка е аналогично на това на стека, но се добавя още и указател към последния елемент на опашката.

Структура от данни дърво

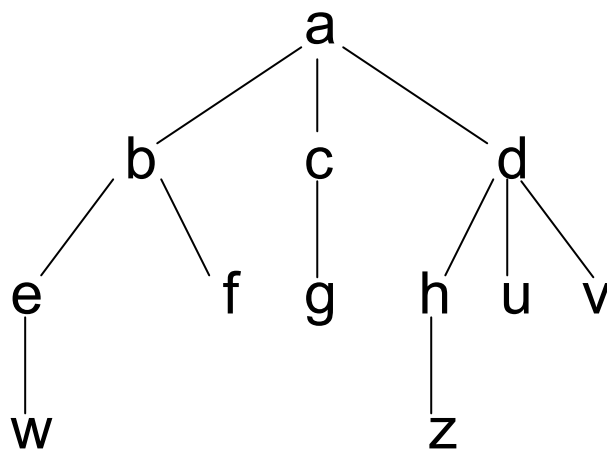
Логическо описание

Нека **Te** е даден тип данни. **Дърво от тип Te** е структура, която е образувана от:

- данна от тип **Te**, наречена **корен на дървото от тип Te**;
- крайно, възможно празно множество с променлив брой елементи – дървета от тип **Te**, наречени **поддървета на дървото от тип Te**.

Пример

Нека **a**, **b**, **c**, ... , **x**, **y**, **z** са елементи от тип **Te**. Следното дърво от тип **Te**



има корен **a** и три поддървета, които са дървета от тип **Te**.

Някои определения

- **Лист** (листо) на дадено дърво – това е корен на поддърво на разглежданото дърво, което няма поддървета.
- **Врџх (вџзел)** – това е корен на поддърво. Върховете, които не съвпадат с корена и листата, се наричат **вътрешни върхове**.
- **Родител (баща)** на даден връх v – това е връхът p , който е такџв, че v е корен на поддърво на p . Коренът на дървото няма родител, а останалите му върхове имат точно по един родител (баща).
- **Предшественици** на даден връх са неговият родител (неговият баща) и предшествениците на неговия баща. Коренът на дървото няма предшественици.

- **Пряк наследник (син)** на даден връх v е всеки връх, за който v е родител (баща). Листата на дървото нямат синове.
- **Наследници** на даден връх са неговите преки наследници (т.е. синовете му) и наследниците на неговите синове. Листата на дървото нямат наследници.
- **Път** в дървото се нарича всяка редица от вида p_1, p_2, \dots, p_k от върхове на дървото, която е такава, че p_i е родител (баща) на p_{i+1} ($i = 1, 2, \dots, k-1$).
- **Височина** на едно дърво се нарича дължината на най-дългия път в дървото, свързващ корена и лист от това дърво.

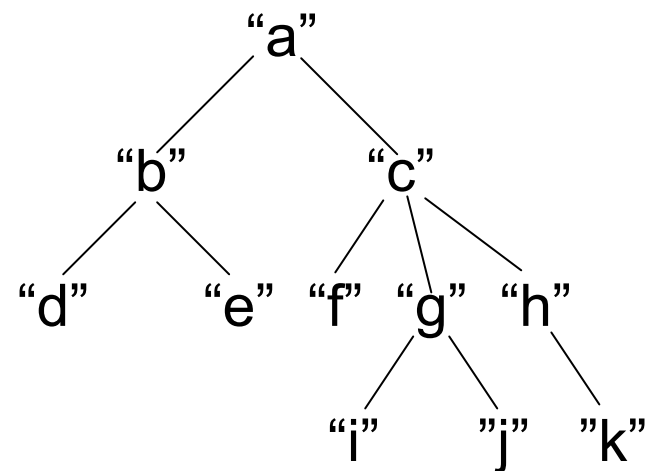
Физическо представяне на дърво

Обикновено се използва свързаното представяне на дърво от тип **Te**, което се осъществява с помощта на свързан списък. При това физическо представяне в един свързан списък се реализират коренът и указателите към поддърветата на дървото от тип **Te**.

Често се използват свързани представяния, основани на описание на дървото чрез задаване на неговите върхове и техните синове (върховете на дървото и техните родители). Тези представяния са най-характерни за езиците за програмиране от типа на Haskell.

Пример

Нека е дадено следното дърво от тип **String**:



Представяне на даденото дърво чрез списък от върховете му и техните синове:

```
[ ("a", ["b", "c"]), ("b", ["d", "e"]), ("c", ["f", "g", "h"]),  
  ("g", ["i", "j"]), ("h", ["k"])]
```

Представяне на даденото дърво чрез списък от върховете му и техните бащи:

```
[ ("b", "a"), ("c", "a"), ("d", "b"), ("e", "b"), ("f", "c"),  
  ("g", "c"), ("h", "c"), ("i", "g"), ("j", "g"), ("k", "h")]
```

Дефиниции на някои функции за работа със списъци

```
type Node = String
```

```
type Treetd = [ (Node, [Node]) ]
```

```
type Treebu = [ (Node, Node) ]
```

```
type Path = [Node]
```

```
tree1 :: Treetd
```

```
tree1 = [ ("a", ["b", "c"]), ("b", ["d", "e"]),  
          ("c", ["f", "g", "h"]), ("g", ["i", "j"]), ("h", ["k"]) ]
```

```
tree2 :: Treebu
```

```
tree2 = [ ("b", "a"), ("c", "a"), ("d", "b"),  
          ("e", "b"), ("f", "c"), ("g", "c"),  
          ("h", "c"), ("i", "g"),  
          ("j", "g"), ("k", "h") ]
```



```
assoc :: Eq a => a -> [(a,[b])] -> (a,[b])
-- assoc :: Node -> [(Node,[Node])] -> (Node,[Node])
-- Осъществява търсене в асоциативен списък
-- по даден ключ.
assoc key [] = (key,[])
assoc key (x:xs)
  | fst x==key = x
  | otherwise  = assoc key xs
```

```

successors :: Node -> Treetd -> [Node]
-- Намира преките наследници на даден връх
-- (възел) node в дървото tree.
successors node tree = snd (assoc node tree)

is_a_node :: Node -> Treetd -> Bool
-- Проверява дали node е връх в дървото tree.
is_a_node node tree = rt node || lf node
  where rt :: Node -> Bool
        rt x = elem x (map fst tree)
        lf :: Node -> Bool
        lf x = elem x (concat (map snd tree))

is_a_leaf :: Node -> Treetd -> Bool
-- Проверява дали node е лист в дървото tree.
is_a_leaf node tree = is_a_node node tree
                      && null (successors node tree)

```

```

parent :: Node -> Treetd -> Node
-- Намира родителя (бащата) на върха node в дървото tree.
parent node [] = ""
parent node tree
  | elem node (successors first_node tree) = first_node
  | otherwise                             = parent node (tail tree)
  where first_node = fst (head tree)

root :: Treetd -> Node
-- Намира корена на дървото tree.
root tree
  | parent first_node tree=="" = first_node
  | otherwise                  = root (tail tree)
  where first_node = fst (head tree)

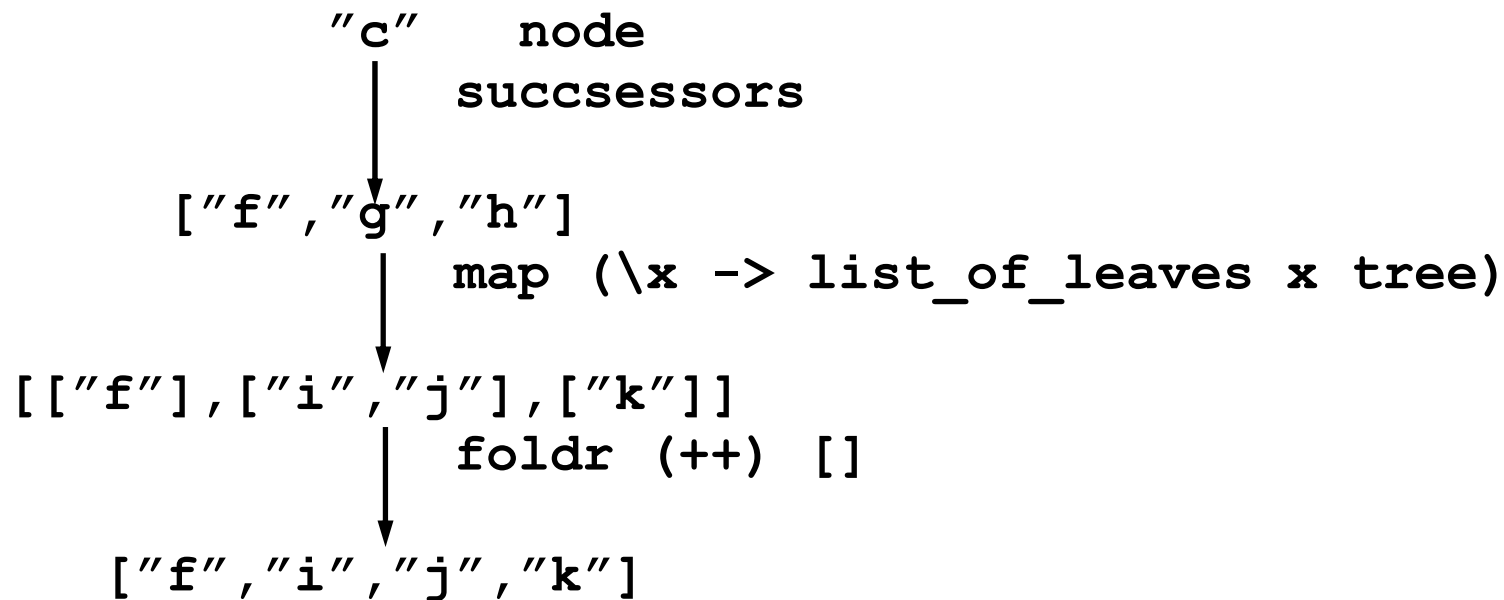
```

```
list_of_preds :: Node -> Treetd -> [Node]
-- Намира предшествениците на върха node в дървото tree.
list_of_preds node tree
  | node==(root tree) = []
  | otherwise          = father:(list_of_preds father tree)
  where father = parent node tree
```

```

list_of_leaves :: Node -> Treed -> [Node]
-- Намира листата на поддървото на дървото tree
-- с корен върха node.
list_of_leaves node tree
  | is_a_leaf node tree = [node]
  | otherwise           = foldr (++) []
    (map (\x -> list_of_leaves x tree)
      (successors node tree))

```



```
list_of_paths :: Node -> Treetd -> [Path]
-- Намира пътищата в дървото tree от даден връх node
-- до листата на поддървото с корен node.
list_of_paths node tree
  | is_a_leaf node tree = [[node]]
  | otherwise           = map (\x -> (node:x))
    (foldr (++) []
      (map (\x -> list_of_paths x tree)
        (successors node tree)))
```

```

    "c"      node
      ↓      successors
    ["f","g","h"]
      ↓      map list_of_paths
    [[["f"]],["g","i"],["g","j"],["h","k"]]
      ↓      foldr (++) []
    ["f"],["g","i"],["g","j"],["h","k"]
      ↓      map (\x -> (node:x))
    ["c","f"],["c","g","i"],["c","g","j"],["c","h","k"]

```

```
height :: Treetd -> Int
-- Намира височината на дървото tree.
height tree = foldr1 max
  (map length (list_of_paths (root tree) tree))
```