Съвременно функционално програмиране

Бойко Банчев

boykobb@gmail.com | 0889-392057

Факултет по математика и информатика – СУ (гост-преподавател)
Институт по математика и информатика – БАН (до 2024 г.)

февруари 2025

Същност и роля на ФП

Програмиране чрез функции:
всяко действие е или определение, или пресмятане на израз.

Функции-стойности. Анонимни функции.

Образуване на функции от функции: композиране,
(частично) прилагане, извличане на контекст.

Едноаргументни функции. Разлагане на функция по аргументи.

Съпоставяне на аргумент с образец.

История, корени и развитие на идеите. Ролята на λ- и
комбинàторното смятане и на езиците Lisp, CPL, ISWIM, PAL,
Pop-2, Scheme, FP/FL, ML, Hope, Miranda, Рефал и др.

Функционалност и императивност в програмирането

Критика на императивния (команден) стил на програмиране.

```
// Пример: намиране на средно аритметично.
            avg = 0;
            for (i = 0; i < n; ++i)
              avg += a[i];
            avg /= n;
```

- разкъсване на действието на дребни части;
- работа със структурни данни на части вместо цялостно;
- преплитане на действия и данни;
- елементи не по съществото на задачата;
- уязвимост към грешки;
- ниска съчетаемост на съставките на програмата.

Понятията *променлива*, *контекст*, *свързване*, *последствие*.

ФП и формалното третиране на програми.

Разновидности на ФП

- ФП, основано на λ-смятане – две действия:
  - абстрахиране – извличане, образуване на функция по зададен израз и параметри в него;
  - прилагане на функция.
- Ограничено ФП: като горното, но функциите не са стойности.
- Апликативно или безаргументно ФП: основано на комбинàторно смятане – само прилагане, няма абстрахиране (функциите нямат явни аргументи).
- Конкатенативно ФП: композиране на безаргументни функции (няма явно прилагане).

Примери: функция за намиране на средно аритметично

безименни функции с явен аргумент

```
(lambda [ns] (/ (fold-left + 0 ns) (length ns)))
```
Scheme

```
\ns -> sum ns / genericLength ns
```
Haskell

```
[](vector<N> ns) -> N
  {return accumulate(ns.begin(),ns.end(),0)
          / ns.size();}
```
C++

изрази със стойност функция: безаргументен стил

```
÷∘[/+,length]
```
FP

```
/[sum,tally]
```
Nial

```
liftM2 (/) sum genericLength
```
Haskell

изрази със стойност функция: конкатенативен стил

```
[sum] [size] cleave /
```
Joy

```
[ sum ] [ length ] bi /
```
Factor

```
dup 0 exch {add} forall exch length div
```
PostScript

Разпространение на ФП по езици

Семейството на Lisp:

Scheme (вкл. Guile, Racket), Common Lisp, Clojure, Shen, ...

Съвременни ФЕ с развити типови системи:

строго функционални – Haskell, Clean, Miranda, Elm;
предимно функционални – ML, OCaml, Alice, F#, GeomLab;
трансформационни – Pure.

Функционално паралелно и разпределено програмиране:

Erlang.

Конкатенативни (стекови):

Joy, Factor, PostScript.

Работа с масиви или редици, неявен паралелизъм:

предимно и по специфичен начин функционални –
APL, J, K, Nial, SequenceL, U.

Хибридни – Ф и релационни или обектноориентирани:

Scala, Dylan, Oz, Mercury, Pop-11, Io.

Универсални, за системно и приложно програмиране:

доразвити с елементи на ФП – C++, Java, C#;
(ново)създадени с възможности за ФП – D, Go, Rust.

Съвременни динамични:

JavaScript, Lua, Ruby, REBOL, (донякъде) Python.

Haskell в контекста на ФП

Водещ език в теорията на ФП, множество приложения в практиката и образованието.

Строго функционален – всеки израз произвежда стойност и няма други последствия.

Строго и статично типизиран – функциите имат типове, неправилното им прилагане е нарушение на синтаксиса.

Богата типова система, пълноценни определяеми типове, различни форми на обобщаване – параметричност, класове.

Всяка функция има един аргумент и един резултат.

Функциите са нестроги (още един смисъл на думата!), а пресмятанията – лениви.

Определенията наподобяват тези в математиката – използват се съпоставяне с образци, условия, варианти. Могат да се тълкуват и като тъждества – поощрява се преобразуването чрез замени при формално третиране на програмите.

Родословие на Haskell

CPL (началото и средата на 1960-те). Императивен език, но с множество начини за образуване на изрази, вкл. от блокове. Локални определения с `where`, успоредност и рекурентност. Извличане на контекст с лексично свързване на локалните променливи. Елементи на програмиране с функции от висок ред.

ISWIM (1966). Каквото в **CPL** плюс: стремеж към уподобяване на математически определения; локални определения и с `let`; отстъпи за влагане; стремеж към структурност чрез функции от висок ред; формално третиране на програмите, преобразуване чрез замени.

PAL (1967). Каквото в **ISWIM** плюс: стойности-енторки; съпоставяне с образец при определяне на функции.

Pop-2, днес **Pop-11** (средата на 1960-те). Императивно-функционален език. Основен принос е частичното прилагане на функции.

FP (1977). Безаргументен стил с функции от висок ред. Формални доказателства за програми.

HOPE (края на 1970-те). Съпоставяне с образец при определяне на функции. Конструктивни типове.

ML (края на 1970-те). Параметрични конструктивни типове, автоматично определяне на типове.

Miranda (средата на 1980-те, предшественици са **SASL** и **KRC**). Ленивост, списъчни определители и др. Оказва най-голямо влияние върху **Haskell**.

Полезни източници онлайн

- Главен сайт. Но по-добре този
- Изпробване онлайн
- Обзор
- *A gentle introduction to Haskell*
- *Why functional programming matters*
- Определение на езика
- Уикикнига
- *Learn you a Haskell* – учебник
- *Real world Haskell* – учебник
- Hoogλe – търсачка-справочник
- *A history of Haskell: being lazy with class*
- D. Turner. *Some history of functional programming languages*
- P. Hudak. *Conception, evolution, and application of functional programming languages*

Печатни учебници

- G. Hutton, *Programming in Haskell*, 2^{nd} ed, Cambridge University Press, 2016.
- S. Thompson, *Haskell: the craft of functional programming*, 3^{rd} ed, Addison Wesley, 2011.
- R. Bird, *Thinking functionally with Haskell*, Cambridge University Press, 2014.
- R. Bird, J. Gibbons, *Algorithm design with Haskell*, Cambridge University Press, 2020.

Реализации на Haskell

Hugs – интерпретатор, най-вече за учебни и експериментални цели.

Облекчена и допълнена с текстовия редактор SciTE версия на Hugs се предлага за използване в курса.

GHC или Haskell platform – професионален компилатор и интерпретатор + инфраструктура (без текстов редактор ;))

(Минималният вариант – само GHC – е **100-кратно по-голям** от Hugs!)

Начало на работа със *SciTE* и *Hugs*

1. Изтеглете този архив.
2. Цялото съдържание на архива е в директория `hugs` в него. Разархивирайте на произволно място `...`, при това ще се образува посочената директория: `...\hugs\`.
3. В тази директория намерете файла `SciTE.exe` и направете препратка (shortcut) към него, например на работния плот (Desktop).
4. Стартирайте редактора *SciTE* чрез препратката.
5. Създайте **файл с име, завършващо на** `.hs` (стандартно за езика) – този файл трябва да съдържа програмата на Haskell. Файлът може да се намира където и да е.
6. Стартиране или спиране на интерпретатора с програмата от файла става чрез менюто `Tools`.

Основни действия със *SciTE* и *Hugs*

Стартиране на интерпретатора – с `Go` или `Run in true console` от меню `Tools` или съответно с `F5` или `Ctrl+1` (програмата автоматично се записва във файла и изпълнява оттам)

Веднъж стартираният интерпретатор продължава да работи, докато не получи командата `:q` или `:quit`

Може да се прекрати и със `Stop Executing` от `Tools` (или `Ctrl+Break`)

В истинска конзола (самостоятелен прозорец, `Ctrl+1`), може да се прекрати и с `Ctrl+Z` → `Enter`

`Ctrl+F6` премества курсора от полето за редактиране в псевдоконзолата (под-прозорец) или обратно

`Shift+F5` изчиства съдържанието на псевдоконзолата

`F8` скрива или показва псевдоконзолата

`Vertical Split` от `Options` поставя псевдоконзолата отдясно или отдолу

За програмата и изпълнението ѝ

Програмата на Haskell съдържа описания, но не пресмятания.

Интерпретаторът изпълнява

- ◇ пресмятания на изрази на Haskell, както и
- ◇ свои команди,

но не допуска добавяне или променяне на описания.

Следните са най-важните команди на интерпретатора:

- ◇ `:q` или `:quit` – напускане;
- ◇ `:t` или `:type` – определяне на типа на израз (аргумент на командата);
- ◇ `:l` или `:load` – прочитане („зареждане“) на програмен файл или модул (аргумент на командата);
- ◇ `:r` или `:reload` – повторно прочитане на програмен файл (след промяна)
- ◇ `:?` – справка за командите на интерпретатора.

Всички команди започват с `:` – така се отличават от изразите – и могат да се съкращават до първата буква от имената си.

Състав на програмата

Програмата на Haskell е съвкупност от определения.

Всяко определение приписва стойности на едно или повече имена (свързва имена със стойности).

Това включва и определенията на функции: всяко определение на функция свързва име с функционна стойност.

Подредбата на определенията е без значение и няма отношение към евентуалните цитирания едно друго.

Освен определения програмата може да съдържа описания на типове за определяните имена, както и указания за образуване и взаимодействие между модули.

Строеж на изрази

Изразите са конструкциите, които образуват стойности.
Състоят се най-вече от прилагания на функции и на операции.

Прилагането на функции е префиксно: функцията предшества аргумента. Всеки израз от вида

$$E_1 \;\; E_2\,,$$

където E_1 и E_2 са изрази, се тълкува като прилагане на функция – стойността на E_1 – към аргумент, чиято стойност е тази на E_2. Израз от вида

$$E_1 \;\; E_2 \;\; E_3 \;\; \ldots \;\; E_n$$

отговаря на редица от прилагания, отляво надясно:

$$(\ldots((E_1 \;\; E_2) \;\; E_3) \;\; \ldots) \;\; E_n$$

Операциите, напр. +, -, *, <, && и др. също са функции, но имат небуквени имена, записват се инфиксно (между двата си аргумента) и прилагането им е след това на префиксните изрази. Например

```
f a b < g p q r + h z
```

е равнозначно на

```
(f a b) < ((g p q r) + (h z))
```

На всяка операция се приписва относителен ред на прилагане, който я съотнася към останалите. Така в горния пример + има по-голяма „сила на свързване“ от < по отношение на формирането на подизрази.

Операцията - се използва и префиксно, с един аргумент (тогава тя е обръщане на аритметичния знак, а не изваждане).

Функции и аргументи

Всяка функция има точно един параметър. Например

```
f x = 5*x+1
```

е определение на функция и `f 4` е израз със стойност 21.

Функциите могат *привидно* да имат повече от един параметъра. Например

```
f x y z = x-y/z
```

задава функция, която, приложена към аргумент, например x_*, на свой ред дава функция. Последната, да я наречем f', ако бъде приложена към аргумент, например y_*, също дава функция – да я наречем f''. Накрая, ако f'' бъде приложена към аргумент, например z_*, резултатът е число. Така, макар че `f` има само един параметър (`x`), условно, за удобство, казваме, че тя има три параметъра, а изрази като `f 7` и `f 7 4` също само условно наричаме *частични прилагания* на `f`.

Функции и операции

По начина на записването им операциите +, - и пр. изглеждат като функции с два аргумента, но това не е така.

Заграден в скоби, знакът на операция обозначава същата функционна стойност, като при това прилагането е префиксно. Например `(/)` е функцията „делене“, а изразът `(/) 3 5` има същата стойност като `3 / 5`, но тя се получава чрез последователно прилагане, `((/) 3) 5`, и изобщо `(/) 3` е правилен, самостоятелен израз – частичното, а всъщност същинското прилагане на `(/)`.

За кое да е `x` частичното прилагане `(/) x` може да се запише като `(x/)`. Частично прилагане на `(/)` към „втория“ ѝ аргумент записваме така: `(/x)`. (Първата функция дели `x`, а втората дели на `x`.) Такива записи се наричат *срезове* (*sections*), съответно ляв и десен, и са приложими почти за всяка операция. (Изключение е `(-x)`, което се тълкува не като десен срез на изваждането, а като `-x`).

Коя да е функция може да се прилага инфиксно като операция, като името ѝ се огради в обратни кавички: за всякакви изрази E_1 и E_2 изразът E_1 `` `f` `` E_2 е равнозначен на `f` E_1 E_2.

Списъци

Списък е крайна или безкрайна редица от стойности от един и същи тип.

Празен списък се означава с `[]`, което е и *конструктор на празен списък*. Всеки друг списък се образува от даден списък и още една стойност: в образувания списък тази стойност е начален член, а даденият списък – остатък след него. Образуването става чрез операцията `:`, *конструктор на непразен списък*.

Например изразът `3:1:4:[]`, равнозначен на `3:(1:(4:[]))`, е списък от три цели числа. Той е образуван от първи член 3 и остатък, който е списък от две цели числа. Последният има първи член 1 и остатък – списък от едно цяло число, на свой ред образуван чрез прилагане на `:` към `4` и празния списък `[]`.

Горният списък се записва по-удобно във вида `[3,1,4]`.

За кой да е непразен списък функциите `head` и `tail` дават първия член (главата) и остатъка му. Вторият член се получава чрез прилагане на `head` към резултата от `tail` и т. н.

Предвид начина на образуване на списък и достъпа до членовете му, всеки списък е всъщност стек.

Прогресии

Едно от синтактичните удобства във връзка със списъци е задаването на аритметични прогресии. Прогресиите могат да бъдат крайни или безкрайни, а разликата между съседни членове или се подразбира 1, или се задава изрично. Такива редици могат да се образуват от всеки тип стойности, за които съществува линейна наредба.

Примери:

`[3..8]`
    списъкът `[3,4,5,6,7,8]`

`['c','f'..'t']`
    низът `"cfilor"` (прогресия със стъпка 3)

`[1,1.1..1.5]`
    списъкът `[1.0,1.1,1.2,1.3,1.4,1.5]` (стъпка 1,1)

`[1..]`
    списък от естествените числа $1, 2, 3, \ldots$

`[0,2..]`
    списък от четните числа $0, 2, 4, \ldots$ (стъпка 2)

Списъчни определители

Списъчните определители (list comprehensions) са също едно от синтактичните удобства на езика. Чрез тях съдържанието на списък се задава чрез свойства и преобразования (вместо посредством изброяване). Определителите предоставят разнообразни възможности чрез кратък и изразителен запис.

Примери:

```
[x : show y | x <- "ab", y <- [20,5,800]]
```
списъкът `["a20","a5","a800","b20","b5","b800"]`

```
[(n,s) | n <- as, n > 0, mod n 5 == 1, let s = n^2, s < m]
```
списък от онези положителни и с остатък 1 по модул 5 числа от списъка `as`, заедно с квадратите им, за които квадратите са по-малки от `m`

Примери: съпоставяне с множество клаузи

Функцията `firstword` извлича първата „дума“ (подниз без шпации) от низ. Дава празен низ, ако аргументът не съдържа дума.

```
firstword ""          = ""
firstword (' ':cs)    = firstword cs
firstword (c:' ':_)   = [c]
firstword (c:cs)      = c : firstword cs
```

Функцията `tighten` сгъстява низ от „думи“: шпациите в началото и в края на низа се премахват, а между всеки две думи остава по една шпация. Ако низът не съдържа думи, резултатът е празен низ.

```
tighten ""              = ""
tighten (' ':cs)        = tighten cs
tighten (c:' ':' ':cs) = tighten (c:' ':cs)
tighten (c:' ':c':cs)  = c : ' ' : tighten (c':cs)
tighten (c:cs)          = c : tighten cs
```

Функцията `wordlist` образува списък от „думите“ в низ (празен, ако низът не съдържа думи).

```
wordlist ""          = []
wordlist (' ':cs)    = wordlist cs
wordlist [c]         = [[c]]
wordlist (c:' ':cs) = [c] : wordlist cs
wordlist (c:c':cs)  = (c:rs):rss
  where rs:rss = wordlist (c':cs)
```

Условни и охраняеми изрази

Условен израз е такъв от вида

$$\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2$$

Пресмята се булевият израз E_0 и според това дали стойността му е `True` или `False` – E_1 или E_2, стойността на който става и тази на условния израз. E_1 и E_2 трябва да имат един и същи тип на стойностите.

Охраняем израз е израз от вида

$$\texttt{| } E_0 \texttt{ = } E_1$$

Пресмятането му се състои в това на булевия израз E_0 и ако стойността му е `True` – на E_1. Охраняемите изрази не съществуват самостоятелно, а като част от определения и обикновено са по няколко последователно; условията се проверяват в реда, в който се срещат. Пример:

```
sign x | x < 0 = -1 | x > 0 = 1 | True = 0
```

Съпоставящи изрази

Съпоставящ израз е израз от вида

```
case израз of
  образец1 -> израз1
  образец2 -> израз2
  ...
```

Стойността на *израз* се съпоставя последователно с всеки от образците докато се стигне до успех и тогава се пресмята съответният на образеца израз. Получената при това стойност е и тази на съпоставящия израз.

Например изразът

```
case xs of (_:_:_) -> True; _ -> False
```

дава стойност `True` или `False` според това дали списъкът `xs` съдържа поне два члена.

Локални определения в изрази и определения

Израз от вида

`let` *определения* `in` *израз*

пресмята *израз*, като при това използва посочените определения. Те са в сила само за това пресмятане.

Следната конструкция осигурява локалност на определения в друго определение:

определение `where` *локални определения*

Примери: варианти на определяне на функцията map

Рекурентно, с условие и с head и tail за извличане на челен член и на остатък на списъка:

```
map1 f xs = if xs == [] then []
            else f (head xs) : map1 f (tail xs)
```

Рекурентно, с head и tail, но с охраняеми изрази вместо if:

```
map2 f xs
  | xs == [] = []
  | True     = f (head xs) : map2 f (tail xs)
```

Рекурентно, със съпоставяне вместо явни условия, head и tail:

```
map3 _ []     = []
map3 f (x:xs) = f x : map3 f xs
```

Чрез списъчен определител – нито рекурентност, нито условия или анализ на аргумента:

```
map4 f xs = [f x | x <- xs]
```

Примери: варианти на определяне на функцията `filter`

Рекурентно, със съпоставяне и с условие:

```
filter1 _ []     = []
filter1 p (x:xs) = if p x then x : filter1 p xs
                          else filter1 p xs
```

Като горното, но с локалноопределена *в израз* стойност:

```
filter2 _ []     = []
filter2 p (x:xs) = let ys = filter2 p xs
                   in if p x then x : ys else ys
```

Като горното, но локалното определение е такова *спрямо основното*:

```
filter3 _ []     = []
filter3 p (x:xs) = if p x then x : ys else ys
  where ys = filter3 p xs
```

Други варианти на `filter`

Рекурентно, с охраняеми изрази вместо условието и съпоставянето. Поради ленивостта локалноопределените `x` и `ys` се пресмятат само когато списъкът е непразен:

```
filter4 p xs
  | xs == [] = []
  | p x      = x : ys
  | True     = ys
  where x  = head xs
        ys = filter4 p (tail xs)
```

Чрез списъчен определител:

```
filter5 p xs = [x | x <- xs, p x]
```

Примери: определения на `reverse`

Просто, но неефективно:

```
rev1 []     = []
rev1 (x:xs) = rev1 xs ++ [x]
```

С помощна, локалноопределена функция и натрупване. Аргументът на `f` е двойка от списъци; тя премества един по един членовете на първия списък във втория:

```
rev2 xs = f (xs,[])
  where f ([],bs)   = bs
        f (a:as,bs) = f (as,a:bs)
```

Пример: синоними в образец

Функцията `uniq` премахва повторения на последователни членове в списък. В образеца `x:xs@(y:_)` частите `xs` и `(y:_)` отговарят на една и съща стойност – остатъка на списъка аргумент след `x`. Двете части са синоними и това се посочва чрез `@`.

```
uniq (x:xs@(y:_)) = if x /= y then x:rs else rs
  where rs = uniq xs
uniq xs = xs
```

Пример: определяне на операция

++ е операция, която образува списък чрез съединяване на два дадени. Може да бъде определена „като функция“:

```
(++) [] ys     = ys
(++) (x:xs) ys = x : (++) xs ys
```

или „като операция“:

```
[] ++ ys     = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Същото важи за всяка инфиксна операция.

Някои стандартнобиблиотечни функции над списъци

`null` *списък*	празен ли е?
`length` *списък*	дължина
`reverse` *списък*	обръщане
`head` *списък*	първи член
`tail` *списък*	остатък
`last` *списък*	последен член
`init` *списък*	всички без последния член
`take` *цяло списък*	префикс
`drop` *цяло списък*	суфикс
`takeWhile` *предикат списък*	префикс по условие
`dropWhile` *предикат списък*	суфикс по условие
`filter` *предикат списък*	подбор по условие

Почленно преобразуване:

```
map функция списък
zipWith функция списък списък
```

`zipWith` е подобна на `map`, но прилага *функция* над два списъка – над всеки два съответни члена.

Следната функция е частен случай на `zipWith` – успоредно съединяване на два списъка в списък от двойки:

```
zip = zipWith (,)
```

Обратната на `zip` функция

```
unzip списък
```

по списък от двойки образува двойка от списъци – от първите и от вторите членове.

Примери: възможни определения на take и zipWith

```
take 0 _      = []
take _ []     = []
take n (x:xs) = x : take (n-1) xs

zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Функции за разлагане на двойки

```
fst (x,_) = x

snd (_,x) = x
```

Пример: използване на ленивостта в определения

Списък от естествените числа, образуван чрез локалноопределена неограничено рекурентна функция:

```
nats1 = f 1 where f n = n : f (n+1)
```

Непосредствено рекурентно определение:

```
nats2 = 1 : map (1+) nats2
```

Пример за лениво пресмятане на израз

В червено е текущо прилаганото действие (съгласно определенията вдясно).
В синьо са „висящите“ функции / операции.

```
nats = 1 : map (1+) nats

map _ []     = []
map f (x:xs) = f x : map f xs

take 0 _      = []
take _ []     = []
take n (x:xs) = x : take (n-1) xs
```

```
take 3 (map (^2) nats) ≡
take 3 (map (^2) (1 : map (1+) nats)) ≡
take 3 ((^2) 1 : map (^2) (map (1+) nats)) ≡
(^2) 1 : take (3-1) (map (^2) (map (1+) nats)) ≡
1 : take (3-1) (map (^2) (map (1+) nats)) ≡
1 : take 2 (map (^2) (map (1+) nats)) ≡
1 : take 2 (map (^2) (map (1+) (1 : map (1+) nats))) ≡
1 : take 2 (map (^2) ((1+) 1 : map (1+) (map (1+) nats))) ≡
1 : take 2 ((^2) ((1+) 1) : map (^2) (map (1+) (map (1+) nats))) ≡
1 : (^2) ((1+) 1) : take (2-1) (map (^2) (map (1+) (map (1+) nats))) ≡
1 : (^2) 2 : take (2-1) (map (^2) (map (1+) (map (1+) nats))) ≡
1 : 4 : take (2-1) (map (^2) (map (1+) (map (1+) nats))) ≡
1 : 4 : take 1 (map (^2) (map (1+) (map (1+) nats))) ≡
1 : 4 : take 1 (map (^2) (map (1+) (map (1+) (1 : map (1+) nats)))) ≡
1 : 4 : take 1 (map (^2) (map (1+) ((1+) 1 : map (1+) (map (1+) nats)))) ≡
1 : 4 : take 1 (map (^2) ((1+) ((1+) 1) : map (1+) (map (1+) (map (1+) nats)))) ≡
1 : 4 : take 1 ((^2) ((1+) ((1+) 1)) : map (^2) (map (1+) (map (1+) (map (1+) nats)))) ≡
1 : 4 : ((^2) ((1+) ((1+) 1))) : take (1-1) (map (^2) (map (1+) (map (1+) (map (1+) nats)))) ≡
1 : 4 : ((^2) ((1+) 2)) : take (1-1) (map (^2) (map (1+) (map (1+) (map (1+) nats)))) ≡
1 : 4 : ((^2) 3) : take (1-1) (map (^2) (map (1+) (map (1+) (map (1+) nats)))) ≡
1 : 4 : 9 : take (1-1) (map (^2) (map (1+) (map (1+) (map (1+) nats)))) ≡
1 : 4 : 9 : take 0 (map (^2) (map (1+) (map (1+) (map (1+) nats)))) ≡
1 : 4 : 9 : [] ≡
1 : 4 : [9] ≡
1 : [4,9] ≡
[1,4,9]
```

Анонимни и именувани функции

Анонимна функция се задава чрез израз от вида

\име -> израз

където *име* посочва параметъра на функцията, а *израз* – тялото на функцията – е това, което се пресмята от нейно име.

Израз от вида

\име$_1$ име$_2$ -> израз

се тълкува като

\име$_1$ -> (\име$_2$ -> израз)

и за удобство се нарича „функция с два параметъра“. Подобно и за функции „с по-голям брой параметри“.

Всяко определение на функция

име име$_1$ име$_2$ ... = израз

се тълкува като

име = \име$_1$ име$_2$ ... -> израз

Навсякъде вместо име на параметър може да се използва образец.

Образуване на функции (стандартнобиблиотечни средства)

`id x = x`	идентитет
`const x _ = x`	константа
`f $ x = f x`	прилагане
`f . g = \x -> f (g x)`	композиция
`flip f x y = f y x`	размяна на местата на аргументите
`curry f x y = f (x,y)`	разлагане по аргумент
`uncurry f (x,y) = f x y`	обратното на горното

Операцията `$` е дясносвързваща, което позволява изрази от вида

$$f_1\ (f_2\ \ldots\ (f_{n-1}\ (f_n\ x))\ldots)$$

да се записват (без скоби) като

$$f_1\ \$\ f_2\ \$\ \ldots\ \$\ f_{n-1}\ \$\ f_n\ x$$

Освен това функцията `($)` и срезове от вида `($` *стойност*`)` са полезни като аргументи на други функции.

foldr и foldl

`foldr` f u *списък*

Обхождане отдясно наляво с последователно прилагане на f – схема за краен списък: $x_1 \diamond \ldots \diamond x_{n-1} \diamond x_n \diamond u$.

`foldl` f u *списък*

Обхождане отляво надясно с последователно прилагане на f: $u \diamond x_1 \diamond x_2 \diamond \ldots \diamond x_n$.

(С $\diamond$ е означено действието на f.)

Примерни определения:

```
foldr _ u []     = u
foldr f u (x:xs) = f x (foldr f u xs)

foldl _ u []     = u
foldl f u (x:xs) = foldl f (f u x) xs
```

foldr1 и foldl1

`foldr1` и `foldl1` са варианти на същите действия за непразни списъци и без начална стойност.

Примерни определения:

```
foldr1 _ [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)

foldl1 _ [x]        = x
foldl1 f (x:x':xs) = foldl1 f (f x x' : xs)
```

Забележки

- Въпреки имената на функциите, поради ленивостта на пресмятанията при `foldr` и `foldr1` те се изпълняват *отляво надясно*, а при `foldl` и `foldl1` – *отдясно наляво* – отвън навътре за всеки израз.
- Вследствие на горното `foldr` и `foldr1` могат да се прилагат и за безкрайни списъци; за `foldl` и `foldl1` това няма смисъл.
- Изобщо `foldr` и `foldr1` са „строго по-мощни“ в смисъл, че `foldl` и `foldl1` могат да се изразят чрез тях, но обратното изразяване не е възможно.
- Типовете на функциите са съответно:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b
foldr1, foldl1 :: (a -> a -> a) -> [a] -> a
```

  и тъй като тези на `foldr` и `foldl` са по-общи, тези две функции допускат и по-разнообразни пресмятания.

Примери

Следните функции са от стандартната библиотека и всяка от тях може да бъде определена чрез функциите от семейството `fold`.

```
sum = foldl (+) 0
product = foldl (*) 1
maximum = foldl1 max
minimum = foldl1 min
length = foldl (\n _ -> 1+n) 0      или
length = foldl (const . (1+)) 0
xs ++ ys = foldr (:) ys xs
concat = foldr (++) []
reverse = foldl (\zs x -> x:zs) []      или
reverse = foldl (flip (:)) []
map f = foldr (\x zs -> f x : zs) []      или
map f = foldr ((:).f) []
filter p = foldr (\x zs -> if p x then x:zs else zs) []
```

Стандартнобиблиотечни функции, производни на горните

`and = foldr (&&) True`	всички са истина?
`or  = foldr (||) False`	поне едно е истина?
`all p = and . map p`	всички изпълняват предиката `p`?
`any p = or . map p`	поне едно изпълнява `p`?
`elem = any . (==)`	дадена стойност е член на списък?
`notElem = all . (/=)`	обратно на горното

Пример: декартово произведение

Функцията намира декартовото произведение (списък от списъци) на наредени множества – списъци.

```
cartesian =
  foldr (\as bss -> concatMap (\a -> map (a:) bss) as) [[]]
```

`concatMap` е стандартнобиблиотечна функция, равнозначна на прилагане на `concat` след `map`.

Друго решение, само чрез `foldr` (Christopher S. Strachey, 1966):

```
cartesian = foldr f [[]] where
  f as zss = foldr g [] as where
    g a yss = foldr h yss zss where
      h zs xss = (a:zs):xss
```

Пример: списък от подмножества

Функцията образува списък от списъци-подмножества на множеството от членове на даден списък-множество – т. нар. степенно множество.

```
powerset :: [a] -> [[a]]    -- в случая е необходимо!
powerset = foldr f [[]] where
  f x yss = foldr g yss yss where
    g ys qss = (x:ys):qss

powerset "abcd"
      ⟶ ["abcd","abc","abd","ab","acd","ac","ad",
         "a","bcd","bc","bd","b","cd","c","d",""]
```

Пример: обединение на интервали

Интервал е двойка $|m, n|$ от цели числа $m \leq n$ и подразбираме, че той съдържа целите числа x, за които $m \leq x \leq n$.

Показаната функция намира обединението на множество (списък) от интервали. Обединяват се всеки два интервала, които имат поне едно общо число, т. е. такива, които се влагат един в друг, пресичат се или се допират. Резултатът е подреден списък от интервали, всеки два от които са взаимно чужди (не могат да се обединят), а всички те заедно съдържат точно същото множество от числа, което се съдържа и в дадените интервали.

```
ranges :: [(Int,Int)] -> [(Int,Int)]
ranges = foldr (foldr f . (:[])) []
  where f (a,b) ((c,d):xs)
          | b < c = (a,b):(c,d):xs
          | a > d = (c,d):(a,b):xs
          | True  = (min a c, max b d) : xs
```

Пример: транспониране на правоъгълна таблица

Под думата „таблица“ в случая се разбира списък от равнодълги списъци. Транспонирането е както при матрици в алгебрата: „обръщане“ около главния диагонал, т. е. мислената ос между първия член на първия списък и последния член на последния.

```
transp []     = []
transp ([]:_) = []
transp ass    = map head ass : transp (map tail ass)
```

По-късо определение с помощта на `foldr`:

```
transp []  = []
transp ass = foldr (zipWith (:)) (repeat []) ass
```

(`repeat []` образува безкрайния списък `[]:[]:...`).

```
transp ["abcde","fghij","klmno","pqrst"]
            ⟶ ["afkp","bglq","chmr","dins","ejot"]
```

Пример: завъртане на таблица на прав ъгъл срещу часовника

```
rotate = transp . map reverse
```

или още по-добре:

```
rotate = reverse . transp
```

```
rotate ["abcde","fghij","klmno","pqrst"]
          ⟶ ["ejot","dins","chmr","bglq","afkp"]
```

Пример: обхождане на таблица по спирала

Функцията образува списък от членовете на таблица, като ги обхожда по спирала – започвайки от първия елемент на първия ред и движейки се отвън навътре по посока на часовника до изчерпване на членовете.

```
spiral []       = []
spiral (xs:xss) = xs ++ spiral (rotate xss)

spiral ["abcde","fghij","klmno","pqrst"]
                              ⟶ "abcdejotsrqpkfghinml"
```

`scanr`, `scanl`, `scanr1`, `scanl1`

Функциите `scanr`, `scanr1`, `scanl` и `scanl1` извършват същите пресмятания както `foldr`, `foldr1`, `foldl` и `foldl1`, но при това съхраняват и междинните за последните резултати, образувайки от тях списък.

Междинните резултати се подреждат съответно отдясно наляво и отляво надясно.

`scanl` и `scanl1` могат да се прилагат и върху безкрайни списъци, докато за `scanr` и `scanr1` това няма смисъл. (В това отношение при функциите `fold` е обратно!)

Пример: пресмятане и разлагане на полином

Следната функция пресмята полином по схемата на Хорнър:

$$c_0x^n + c_1x^{n-1} + \cdots + c_{n-1}x + c_n = (\ldots((c_0x + c_1)x + c_2)x + \cdots)x + c_n\,.$$

Параметри на `poly` са списъкът от коефициенти `cs` и променливата `x`:

```
poly cs x = foldl1 (\v c -> v*x+c) cs
```

Например пресмятането на израза `poly [1,-2,3,5] 4` дава

$$x^3 - 2x^2 + 3x + 5\,|_{x=3} = 49$$

Като заменим обръщението към `foldl1` със `scanl1`:

```
poly cs x = scanl1 (\v c -> v*x+c) cs
```

за кой да е полином $p(x)$ и стойност x^* получаваме разлагането $p(x) = (x-x^*)q(x)+p(x^*)$.

Например пресмятането на израза `poly [1,-2,3,5] 4` сега дава `[1,2,11,49]` (т. к. $x^3-2x^2+3x+5 = (x-4)(x^2+2x+11)+49$).

Пример: намиране на най-голям сбор от последователни членове

Задачата е да намерим най-голям неотрицателен сбор от последователни числа в краен списък. Например за списъка `[-1,3,5,-2,1,2,-2,1]` търсеният сбор е $9 = 3+5+-2+1+2$. Ако списъкът съдържа само отрицателни числа, сборът е 0 и се образува от празен отрязък на дадения списък.

Образуваме търсения сбор последователно за всеки префикс на списъка. За целта за всеки префикс образуваме и най-голям неотрицателен сбор измежду неговите суфикси. Така намираме

```
maxsum = fst . foldl f (0,0)
  where f (best,curr) x = (max best curr', curr')
          where curr' = max (curr + x) 0
```

или

```
maxsum = maximum . scanl (\s x -> max (s + x) 0) 0
```

Пример: представяне на граф

Дъга на ориентиран граф с върхове – стойности от тип a задаваме като двойка `(a,a)`. По списък от дъгите на такъв граф функцията `mkGraph` образува представяне на графа чрез списък с членове от тип `(a,[a])`. На всеки връх отговаря по един член на списъка и в него стойността на върха е първа в двойката, а втори е списък от съседите на върха.

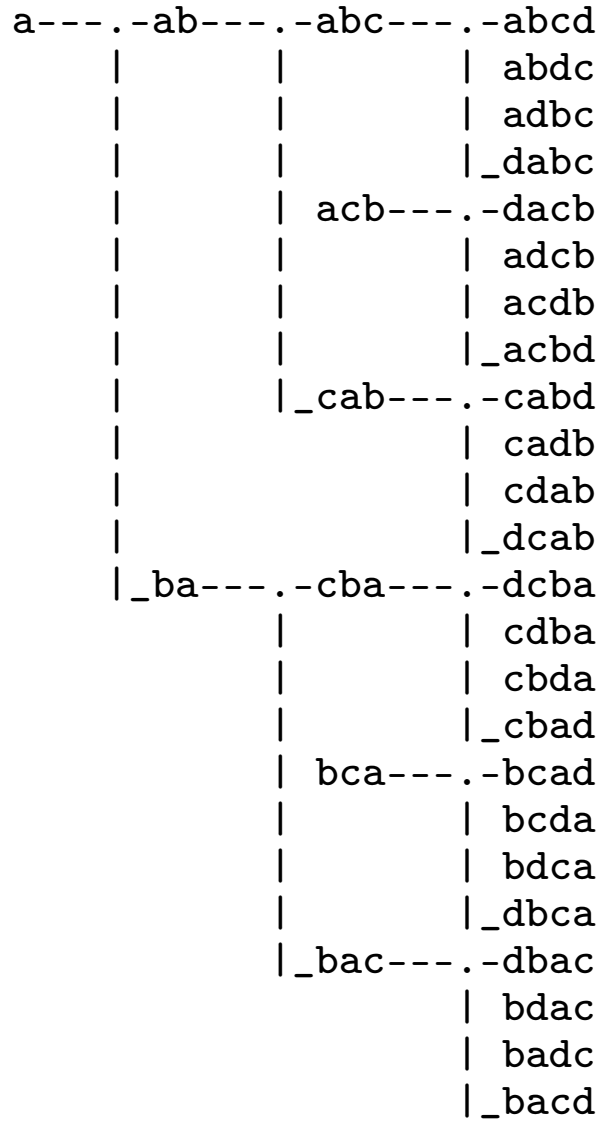
```
mkGraph g = map f $ foldr addif []
                    (let (as,bs) = unzip g in as ++ bs)
  where addif x xs = if elem x xs then xs else x:xs
        f x = (x, [b | (a,b) <- g, a == x])
```

Например

```
mkGraph [(1,2),(3,4),(3,1),(2,3),(1,3),(2,4),(2,1)]
        ⟶ [(2,[3,4,1]),(3,[4,1]),(4,[]),(1,[2,3])]
```

Редица от пермутации с максимална близост

Всеки две съседни пермутации в породената редица се отличават една от друга с размяна на само два техни елемента и то съседни. На схемата са дадени редици от пермутации за редове от 1 до 4.



Функция за пораждане на редица от пермутации

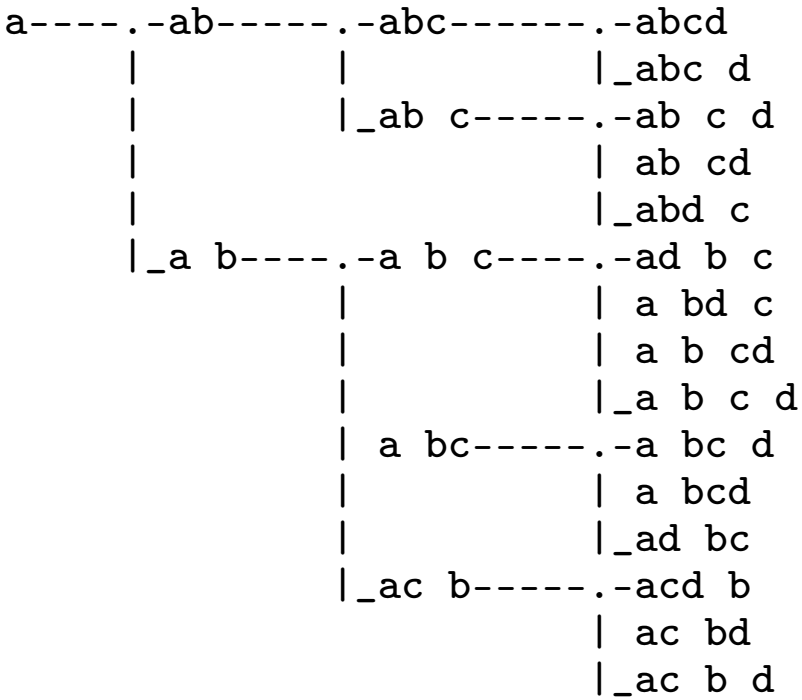
Пораждат се пермутациите на кое да е множество-списък. Редът на образуване е различен от този на схемата с това, че подвижният елемент е не последният, а първият.

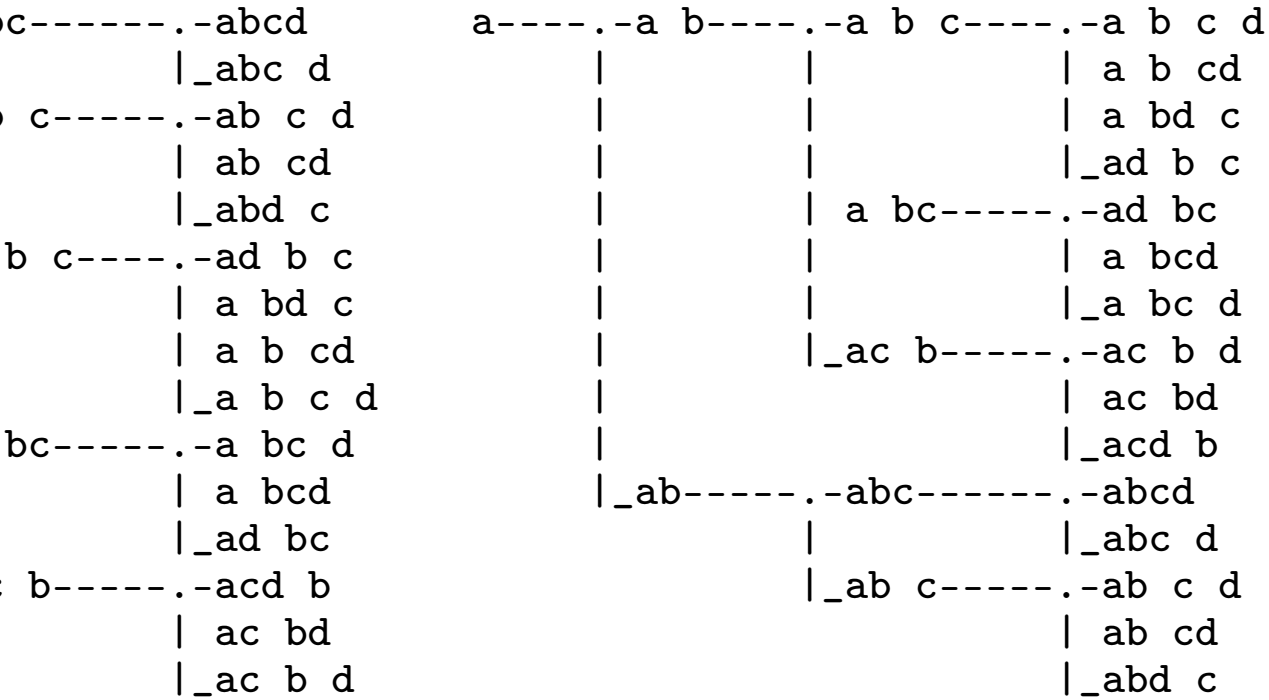
```
grayperm []     = [[]]
grayperm (a:as) =
    concat $ zipWith ($) (cycle [id,reverse])
                         (map insertAlong (grayperm as))
  where
    insertAlong xs =
        (a:xs) : case xs of []   -> []
                            x:xs -> map (x:) (insertAlong xs)
```

Редица от разбивания на множество с максимална близост

Всеки две съседни разбивания в породената редица се отличават едно от друго с преместване на един член на множеството в друго подмножество.

На схемата са дадени редици от разбивания за множества с от 1 до 4 члена. Показани са два начина за пораждане на редиците: започвайки от едно подмножество и от възможно най-много подмножества.





Функция за пораждане на редица от разбивания

Пораждат се разбиванията на кое да е множество-списък по някой от двата начина от схемата. Като определим

```
swing = cycle [id,reverse]
```

за пораждане на двата вида редици можем да се обръщаме съответно с

```
graysets swing списък
```

и

```
graysets (tail swing) списък.
```

```
graysets mode []     = [[]]
graysets mode (a:as) =
    map (map reverse) (foldl ascend [[[a]]] as)
  where
    ascend asss a =
      concat $ zipWith ($) mode (map (atEach a) asss)
    atEach a []       = [[[a]]]
    atEach a (as:ass) =
      ((a:as):ass) : map (as:) (atEach a ass)
```

Усилен вариант на `foldr`

Задача за мотивиране: вмъкване на стойност в подреден в ненамаляващ ред списък, възможно безкраен.

Не е трудно да се напише непосредствено рекурсивно решение, но се опитваме да избегнем това чрез използване на `foldr`.

Забелязваме, че функцията аргумент на `foldr` не бива да отговаря за вмъкването – иначе то става многократно!

Следното е решение, но само за крайни списъци:

```
insert a = foldr f [a] where
  f x rs@(r:rs') = if x <= r then x:rs else r:x:rs'
```

Обобщаваме `foldr`:

```
efoldr _ u [] = u
efoldr f u xs = f xs (efoldr f u (tail xs))
```

Определяме `insert` чрез `efoldr` – ако за поредното x от списъка е вярно $x < a$, мястото на a се търси с нови повиквания на `efoldr`, а при $x \geq a$ резултатът се образува непосредствено и вторият аргумент на `f`, който именно съдържа повикване на `efoldr`, не се използва:

```
insert a = efoldr f [a] where
  f xs@(x:_) rs = if x >= a then a:xs else x:rs
```

Определяне на `foldl` чрез `foldr`

Един вариант е следният:

```
rfl f u = foldr (flip f) u . reverse
```

При него списъкът се обхожда два пъти – с `reverse` и с `foldr`.

По-пестеливо решение:

```
rfoldl f u xs = foldr (\x->(.(`f`x))) id xs u
```

Прилагането на `foldr` образува, започвайки от `id`, композиция от частични прилагания на `f`, по едно за всеки член на списъка. Приложена към `u`, тази образувана от `foldr` функция дава същия резултат като `foldl`.

Пример за частично прилагане (без `u`) на тялото на функцията:

```
foldr (\x->(.(`f`x))) id [a,b,c] ≡
((id.(`f`c)).(`f`b)).(`f`a)
```

При пълно прилагане на `rfoldl` вместо композиция се образува влагане на прилагания на частичните прилагания на `f`. Например

```
rfoldl f u [a,b,c]
```

води последователно до

```
foldr (\x->(.(`f`x))) id [a,b,c] u ≡
(\x->(.(`f`x))) a (foldr (\x->(.(`f`x))) id [b,c]) u ≡
(.(`f`a)) (foldr (\x->(.(`f`x))) id [b,c]) u ≡
(foldr (\x->(.(`f`x))) id [b,c] . (`f`a)) u ≡
foldr (\x->(.(`f`x))) id [b,c] ((`f`a) u)
```

после аналогично до

```
foldr (\x->(.(`f`x))) id [c] ((`f`b) ((`f`a) u))
```

и накрая до

```
foldr (\x->(.(`f`x))) id [] ((`f`c) ((`f`b) ((`f`a) u)))
```

което вече се привежда в

```
id ((`f`c) ((`f`b) ((`f`a) u))) ≡
(`f`c) ((`f`b) ((`f`a) u)) ≡
f ((`f`b) ((`f`a) u)) c
```

(Оттук нататък продължаването на пресмятането зависи от `f`.)

Образуване на безкрайни списъци

Три функции:

```
iterate f x = x : iterate f (f x)

repeat x = x : repeat x
(или  repeat x = iterate id x)

cycle xs = xs ++ cycle xs
```

Други възможни определения:

```
iterate f x = xs where xs = x : map f xs
repeat x = xs where xs = x : xs
cycle xs = zs where zs = xs ++ zs
```

Пример: пресмятане на e^x

Приближено пресмятане на e^x като частична сума на реда

$$1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

```
expn x n = sum $ take n $ map fst $ iterate f (1,0)
  where f (v,i) = (v*x/i',i') where i' = i+1
```

Пример: редицата от n-те степени на числата $1, 2, 3, \ldots$

За да получим редицата от n-те степени ($n \geq 0$) започваме с редицата $1\,1\,1\ldots$. Ако $n > 0$, в текущата редица премахваме всеки $n+1$-и член и образуваме следваща редица, започваща с 1, всеки следващ член на която е сбор на този преди него и този, който е на същото място в предходната редица. Ако $n > 1$, в последната редица премахваме всеки n-и член, по същия начин образуваме от нея нова редица и т. н. – за всяка следваща редица периодът на премахване е с 1 по-къс. Така след $1\,1\,1\ldots$ образуваме всичко n редици, последната от които е търсената.

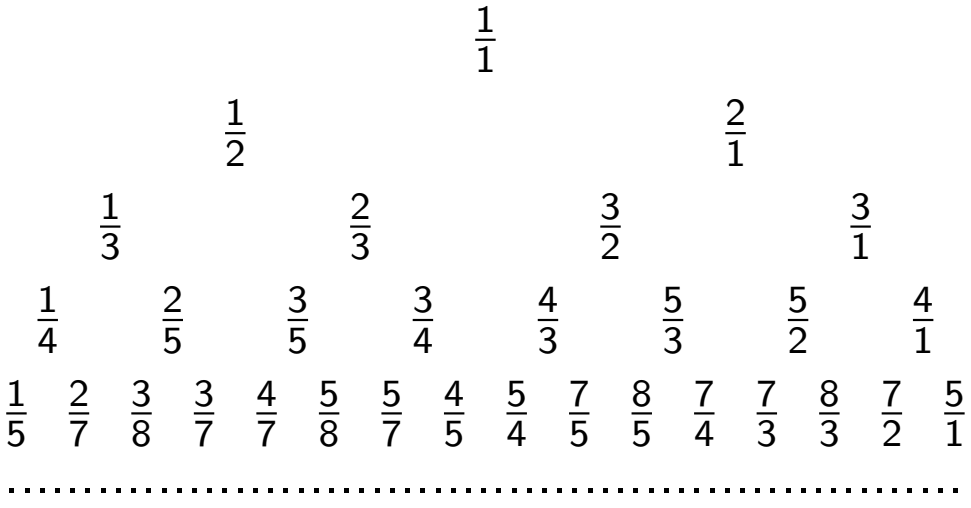
Образуване за $n = 4$:

1	1	1	1	~~1~~	1	1	1	1	~~1~~	1	1	1	1	~~1~~	1	1	...
1	2	3	~~4~~		5	6	7	~~8~~		9	10	11	~~12~~		13	14	...
1	3	~~6~~			11	17	~~24~~			33	43	~~54~~			67	81	...
1	~~4~~				15	~~32~~				65	~~108~~				175	~~256~~	...
1					16					81					256		...

```
powers n = f 1 where
  f k | k <= n = scanl1 (+) $ g $ f (k+1)
      | True   = repeat 1
    where g xs = as ++ g bs
            where (as,_:bs) = splitAt k xs
```

Пример: дърво на Щерн-Броко

Дървото на Щерн-Броко (Stern-Brocot) е двоично дърво, съдържащо всички положителни дроби в несъкратим вид, всяка по веднъж.



Дървото на числителите на дробите може да се получи по следния начин. Първият слой съдържа 1. Ако даден слой е редицата L, първата половина на следващия слой е също L, а втората половина е почленният сбор на L и обърнатата L. Така след 1 е 1 2, след това 1 2 3(=1+2) 3(=2+1) и т. н.

Дървото на знаменателите е огледално симетрично на това на числителите, т. е. всеки негов слой е обърнатата редица от числители за същия слой.

Посочените зависимости дават удобен начин да построим дървото като безкрайна редица от слоевете му – редици с растяща дължина. Самите дроби можем да представим като двойки от числител и знаменател.

Други зависимости позволяват да намерим координатите – номер на слой и пореден номер в слоя – на коя да е дроб и обратно, да намерим дробта с дадени координати.

За допълнителна информация виж напр. статията в Уикипедия и препратките в нея.

Дърво на Щерн-Броко – реализация

```
-- Редица от редовете от дроби (като двойки) на дървото:
sbtree = zipWith zip nums (map reverse nums) where
  nums = iterate (\ns -> ns ++ zipWith (+) ns (reverse ns)) [1]

-- Дроб -> координати:
fr2rc (n,d) = f (n,d) (0,0) where
  f (n,d) (r,c)
    | n < d = f (n,d-n) rc
    | n > d = f (n-d,d) (r',c'+1)
    | True  = (r,c)
    where rc@(r',c') = (r+1,2*c)

-- Координати -> дроб:
rc2fr (r,c) = f (r,c) (1,1) where
  f (r,c) (n,d)
    | r == 0       = (n,d)
    | c`mod`2 == 0 = f rc (n,n+d)
    | True         = f rc (n+d,d)
    where rc = (r-1,c`div`2)
```

Примери: редиците на Фибоначи и на простите числа

Редицата от числата на Фибоначи:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Редицата от простите числа:

```
primes = map head $ iterate sieve [2..] where
  sieve (p:ns) = filter ((0/=).(`mod`p)) ns
```

Пример: триъгълник и числа на Каталан

Триъгълникът се състои от безкрайно много безкрайни редици. j-то число в ред i за $i, j = 0, 1, 2, \ldots$ е равно на броя редици от i члена -1 и $i+j$ члена 1 с неотрицателни частични сборове.

Първите членове на всяка редица образуват редицата от числата на Каталан C_n за $n = 0, 1, 2, \ldots$.

```
1   1   1   1   1   1   1   1 ...
    1   2   3   4   5   6   7 ...
        2   5   9  14  20  27 ...
            5  14  28  48  75 ...
               14  42  90 165 ...
                   42 132 297 ...
                      132 429 ...
                          429 ...
                              ...
```

Всяка редица започва с втория член на предишната, а всяко следващо число в нея е сбор на това отляво и това над него. Така триъгълникът и редицата от числата на Каталан могат да се зададат с определенията

```
catrows = iterate (scanl1 (+) . tail) (repeat 1)
cats = map head catrows
```

Пример: редицата от числа на Хаминг

Числа на Хаминг (регулярни числа) са числата с прости делители само 2, 3 и 5. Списъкът `hs` се образува чрез безповторно сливане на три други списъка – числата на Хаминг, кратни на 2, тези кратни на 3 и кратните на 5, които на свой ред се образуват от `hs`.

```
hs = 1 : map (*2) hs || map (*3) hs || map (*5) hs
  where xs@(x:xs') || ys@(y:ys')
                   | x < y = x : xs' || ys
                   | x > y = y : xs  || ys'
                   | True  = x : xs' || ys'
```

Обобщение: функция, която по краен списък от цели положителни числа образува списък от числата, произведения на дадените.

```
hams ns = rs where
  rs = 1 : foldr1 (||) (map ((`map`rs).(*)) ns)
  -- операцията || се определя както по-горе
```

Безаргументни определения на функции

Определенията на много функции могат да бъдат преобразувани в безаргументен вид, който за някои цели е за предпочитане. Някои от дадените вече определения на стандартнобиблиотечни функции са изцяло или частично безаргументни. Например за функциите `all` и `any` дадохме определенията

```
all p = and . map p
any p = or . map p
```

където неявен е само аргументът-списък, но валидни са и следните изцяло безаргументни определения:

```
all = (and.).map
any = (or.).map
```

Представянето в безаргументен вид се основава на т. нар. η-привеждане, състоящо се в следното. Ако `F` и `G` са изрази, чиято стойност е функция и за всеки (уместен) аргумент `x` е вярно `F x == G x`, смятаме, че `F` и `G` представят една и съща функция и следователно са взаимнозаменими изрази. В частност, ако `F` е функционен израз (и `x` не се среща като свободна променлива в него), функцията `\x -> F x` е същата като `F`.

За да направим възможно η-привеждане, в редица случаи се налага да прибягваме до преобразования като следните (за какви да е функционни изрази `F` и `G`, изрази `E`, `X` и `Y`) и операция •:

```
F (G X)    ⟹   (F.G) X
X • Y      ⟹   (•) X Y
E • F X    ⟹   ((E•).F) X
F X • E    ⟹   ((•E).F) X
F X Y      ⟹   flip F Y X
```

Когато `F` е име на функция, например `f`, вместо горното преобразование по-подходящо може да е следното:

```
f X Y      ⟹   (`f`Y) X
```

Пример: привеждане на `map` в безаргументен вид

Освобождаването от втория параметър на `map` (списъка) е непосредствено. Освобождаването от първия параметър (функцията), както и преди това намирането на безаргументен вид на безименната функция-параметър на `foldr` изискват някои преобразования.

```
map f xs = foldr (\x rs -> f x : rs) [] xs
map f = foldr (\x rs -> f x : rs) []
map f = foldr (\x rs -> (:) (f x) rs) []
map f = foldr (\x -> (:) (f x)) []
map f = foldr (\x -> ((:).f) x) []
map f = foldr ((:).f) []
map f = (`foldr`[]) ((:).f)
map f = (`foldr`[]) (((:).) f)
map f = ((`foldr`[]).((:).)) f
map = (`foldr`[]).((:).)
```

Синоними на типове

`type` *име параметри* = *израз-тип*

Конструкцията създава синоним на конкретен или обобщен тип – тя е подобна на `typedef` в езиците C и C++, но е по-мощна, тъй като допуска параметризиране.

Името е въвежданият синоним. Параметрите (незадължителни) обозначават типове. Ако ги има, синонимът отговаря на обобщен тип (семейство типове). Конкретни типове-представители на семейството се посочват, като на мястото на параметрите се поставят конкретни типове.

Името-синоним започва с главна буква. Параметрите („променливи типове“) са имена, започващи с малки букви.

Въвежданият синоним е навсякъде и във всичко равнозначен на прякото посочване на типа със съответния израз.

Примери

`type String = [Char]`

Низ (така е определено в езика).

`type List a = [a]`

Синоним на познатия (обобщен) списъчен тип.

`type Table a b = [(a,b)]`

Таблица на съответствия между стойности от тип `a` и стойности от тип `b` (асоциативен списък).

`type Graph a = Table a (List a)`

Граф: на всеки връх отговаря списък от върхове-съседи.

Определения на типове

`data` *име параметри* = *конструктор*$_1$ | *конструктор*$_2$ | ...

Името посочва име на определяния тип. Параметрите (незадължителни) обозначават типове. Ако определяният тип има параметри, той е обобщен (т. е. е семейство типове). Конкретни типове-представители на семейството се получават, като на мястото на параметрите се поставят конкретни типове.

Конструкторите са един или повече и наподобяват функции, образуващи стойностите на типа. Всеки конструктор може да се придружава от един или повече свои параметри, задаващи типовете на аргументите му, или да няма параметри.

Конструкторите без аргументи са на практика константите на определяния тип. Типовете, на които всички конструктори са безаргументни, са аналогични на изброявания в други езици: между конструкторите съществува наредба и от тях могат да се образуват аритметични прогресии.

Имената на определяния тип и на конструкторите му започват с главни, а другите участващи имена – с малки букви.

Примери

Типът на булевите стойности в Haskell:

```
data Bool = False | True
```

Изразът `False < True` има стойност `True`.

Тип, подобен на вградения в Haskell списъчен тип:

```
data List a = Em | Cons a (List a)
```

Функция, която намира дължина на такъв списък:

```
listlen :: List a -> Int
listlen Em           = 0
listlen (Cons _ xs) = 1 + listlen xs
```

Тип двоично дърво с върхове от (какъв да е) тип a:

```
data BTree a = Et | Bt a (BTree a) (BTree a)
```

Функция, която обхожда такова дърво в инфиксен ред:

```
btwalk :: BTree a -> [a]
btwalk Et            = []
btwalk (Bt x t1 t2) = btwalk t1 ++ x : btwalk t2
```

Разширени конструктори

Разширената форма на конструктор позволява автоматично да се образуват функции, обратни на него в смисъл, че чрез тях извличаме стойностите на аргументите му (както `head` и `tail` по отношение на `:`).

За целта снабдяваме аргументите с имена. Разширеното задаване на конструктор има вида

Име-конструктор {*име-арг*$_1$ `::` *Тип*$_1$`,` *име-арг*$_2$ `::` *Тип*$_2$`, ...}`

име-арг$_1$ и т. н. според броя на аргументите са имената на функциите за извличане на съответните стойности.

Например, ако типът `BTree` се допълни както следва

```
data BTree a = Et | Bt {root :: a, left :: (BTree a),
                                   right :: (BTree a)}
```

функцията за обхождане може да се определи във вида

```
btwalk Et = []
btwalk t  = btwalk (left t) ++ root t : btwalk (right t)
```

Класове

Понятие за клас.
Типове-представители на клас.
Описателят `deriving`

Класът `Eq`

Типът `Ordering` и класът `Ord`

Класовете `Show` и `Read`

Класът `Bounded`

Класът `Enum` (аритметични прогресии)

Класът `Ix`

Числови класове и типове

.....