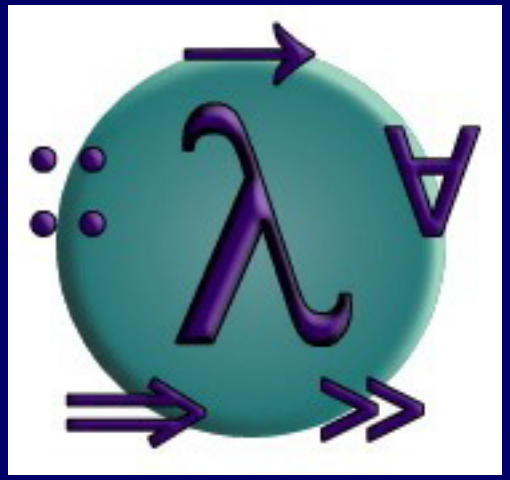
დაპროგრამება HASKELL ენაზე
(მულტიმედიური პრეზენტაცია)



არჩილ ფრანგიშვილი, ზურაბ წვერაიძე,
ოლეგ ნამიჩეიშვილი

წინასწარი შენიშვნები ავტორებისგან 1

- ჰასკელი წარმოადგენს ზოგადი დანიშნულების პროგრამების *წმინდა ფუნქციონალურ ენას*, რომელიც შეიცავს მრავალ უკანასკნელ ინოვაციას პროგრამების ენათა დამუშავების სფეროში.
- ჰასკელი უზრუნველყოფს მაღალი რიგის ფუნქციებს, არამკაცრ სემანტიკას, სტატიკურ პოლიმორფულ ტიპიზაციას, მონაცემთა ალგებრულ ტიპებს, რომლებსაც მომხმარებელი განსაზღვრავს, ნიმუშთან შედარებას, სიების აღწერას, მოდულურ სისტემას, შეტანისა და გამოტანის მონადურ მექანიზმს და მონაცემთა პრიმიტიული ტიპების მდიდარ ნაკრებს სიების, მასივების ნებისმიერი და ფიქსირებული სიზუსტის მთელი რიცხვების, ასევე მცურავწერტილიანი რიცხვების ჩათვლით.

წინასწარი შენიშვნები ავტორებისგან 2

- ჰასკელი არამკაცრი ფუნქციონალური ენების მრავალი წლის კვლევათა კულმინაციაც და კრისტალიზაციაც არის.

- ამ ენის ასათვისებლად, სალექციო [1] კურსთან ერთად, მნიშვნელოვანი როლი ენიჭება სათანადო ლაბორატორიულ სამუშაოთა [2] პრაქტიკუმს.

- ქართველი მკითხველისათვის ჩვენ მიერ მომზადებული ამ სახელმძღვანელოების ეფექტურ გამოყენებას საუნივერსიტეტო პრაქტიკაში კარგ დახმარებას გაუწევს, ალბათ, წინამდებარე საპრეზენტაციო სლაიდების ნაკრებიც.

წინასწარი შენიშვნები ავტორებისგან 3

- სალექციო, ლაბორატორიული და საპრეზენტაციო მთელი მასალის, ასევე მისი გადმოცემის სტილის შერჩევისას, არსებითად, გამოყენებულია ნოტინგემის უნივერსიტეტში (**England, University of Nottingham**) მიღებული მიდგომები, რომლებსაც პროფესორი *გრემ ჰატონი* (*Graham Hutton*) ამკვიდრებს.
- ჩვენი დროის ამ უდიდესი მეცნიერისა და პედაგოგის ხსენებული მიდგომები და იდეები ნასესხებია ბესტსელერად ქცეული წიგნიდან:

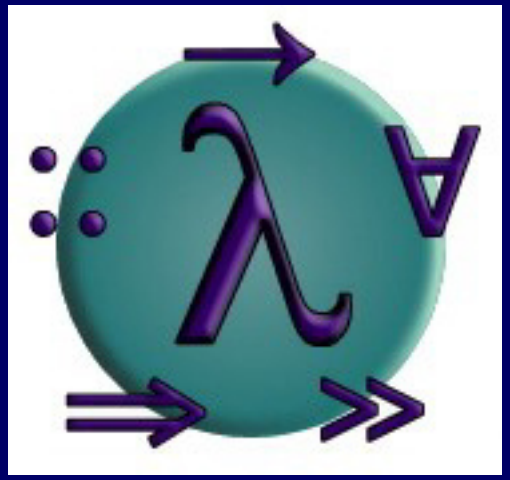
- *Graham Hutton*, **Programming in Haskell. Cambridge University Press, 2007, pp 200.**

ლიტერატურა

- 1. *არჩილ ფრანგიშვილი, ზურაბ წვერაიძე, ოლეგ ნამიჩეიშვილი* დაპროგრამება ჰასკელზე. – თბილისი: საგამომცემლო სახლი «ტექნიკური უნივერსიტეტი», 2012. – 288 გვ.

- 2. *არჩილ ფრანგიშვილი, ზურაბ წვერაიძე, ბადრი ბარდაველიძე, ოლეგ ნამიჩეიშვილი* ფუნქციონალური დაპროგრამების ენა ჰასკელი (ლაბორატორიული პრაქტიკუმი). – თბილისი: საგამომცემლო სახლი «ტექნიკური უნივერსიტეტი», 2012. – 149 გვ.

დაპროგრამება Haskell ენაზე



თავი 1 - შესავალი

პროგრამული უზრუნველყოფის კრიზისი

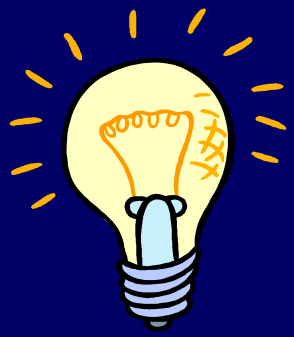
- როგორ გავართვათ თავი თანამედროვე კომპიუტერულ პროგრამათა ზომასა და სირთულეს?
- როგორ შეგვიძლია პროგრამათა დამუშავების დროისა და ღირებულების შემცირება?
- როგორ შეგვიძლია გავზარდოთ ჩვენი რწმენა, რომ პროგრამები კორექტულად დაასრულებს მუშაობას?

დაპროგრამების ენები

ერთ-ერთ გამოსავალს პროგრამული უზრუნველყოფის კრიზისიდან წარმოადგენს დაპროგრამების ახალი ენების შექმნა:

- შევეცადოთ პროგრამათა დაწერა მკაფიოდ, ლაკონურად და აბსტრაქციის მაღალ დონეზე;
- პროგრამული უზრუნველყოფის მრავალჯერადი გამოყენების კომპონენტებს უპირატესობა მივცეთ;
- წავახალისოთ ფორმალური შემოწმების გამოყენება;

- უზრუნველვყოთ სწრაფი პროტოტიპირების შესაძლებლობა;
- უზრუნველვყოთ ამოცანათა გადაწყვეტის მძლავრი საშუალებები.



ფუნქციონალური ენები ამ მიზნების მისაღწევად განსაკუთრებულად ელეგანტურ საშუალებებს გვაძლევს !!!

რა არის ფუნქციონალური დაპროგრამება?

არსებობს სხვადასხვა აზრი და ძნელია ზუსტი განსაზღვრა, მაგრამ ზოგადად შეიძლება ითქვას:

- ფუნქციონალური დაპროგრამება – დაპროგრამების სტილია, რომელშიც გამოთვლათა მთავარი მეთოდია არგუმენტებად წარმოდგენილი ფუნქციების გამოყენება;
- ფუნქციონალური ენა – ეს ისეთი ენაა, რომელიც ახორციელებს დაპროგრამების ფუნქციონალურ სტილს და ხელს უწყობს მას.

მაგალითი

1-დან 10-მდე მთელი რიცხვების შეკრება Java-ზე

```
total = 0;
for (i = 1; i ≤ 10; ++i)
   total = total+i;
```

გამოთვლის მეთოდია ცვლადისათვის მნიშვნელობის მინიჭება

მაგალითი

1-დან 10-მდე მთელი რიცხვების შეკრება Haskell-ზე

```
sum [1..10]
```

გამოთვლის მეთოდია ფუნქციის გამოყენება

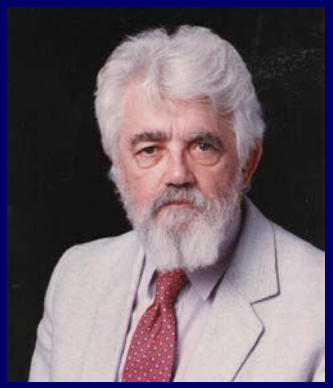
ისტორიული მიმოხილვა

1930 წელი:

ალონზო ჩერჩი ქმნის ლამბდა-აღრიცხვას – ფუნქციათა მარტივ, მაგრამ მძლავრ თეორიას.

ისტორიული მიმოხილვა

1950 წელი:



ჯონ მაკ-კარტი ქმნის პირველ ფუნქციონალურ Lisp ენას ლამბდა-აღრიცხვის გარკვეული ზეგავლენით, მაგრამ ცვლადისათვის მნიშვნელობათა მინიჭების შენარჩუნებით.

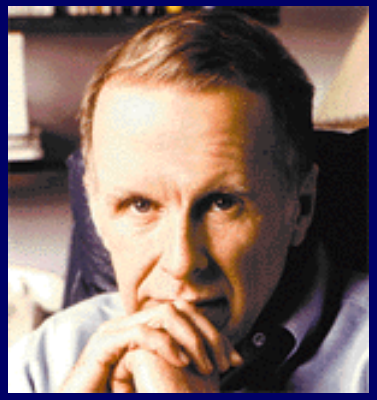
ისტორიული მიმოხილვა

1960 წელი:

პიტერ ლენდინი ქმნის პირველ წმინდა ფუნქციონალურ ISWIM ენას ლამბდა-აღრიცხვის მკაცრ საფუძველზე და ცვლადისათვის მნიშვნელობის მინიჭების გამოუყენებლად.

ისტორიული მიმოხილვა

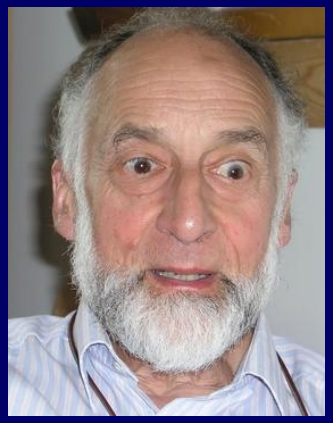
1970 წელი:



ბეკუსი ავითარებს ფუნქციონალურ ენაზე წარმოდგენებს, შემოაქვს მაღალი რიგის ფუნქციათა ცნება და აყალიბებს აზრებს პროგრამებზე ასეთი ენის გამოყენებისას.

ისტორიული მიმოხილვა

1970 წელი:



რობინ მილნერი და სხვები ქმნიან ML ენას – პირველ თანამედროვე ფუნქციონალურ ენას, რომელშიც წარმოდგენილია ტიპის გამოყვანა და პოლიმორფული ტიპები.

ისტორიული მიმოხილვა

1970 – 1980 წლები:

დევიდ ტერნერი ქმნის რიგ ზარმაც ფუნქციონალურ ენას, რომლებიც აგვირგვინებს Miranda სისტემას.

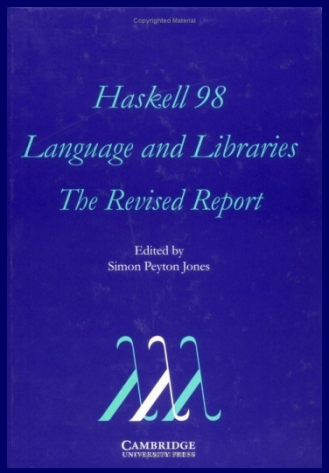
ისტორიული მიმოხილვა

1987 წელი:

კვლევათა საერთაშორისო კომიტეტი სტანდარტული ზარმაცი Haskell ფუნქციონალური ენის შექმნის ინიცირებას ახდენს

ისტორიული მიმოხილვა

2003 წელი:



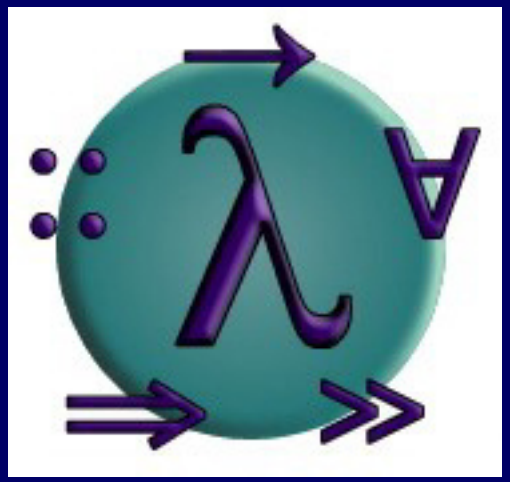
კომიტეტი აქვეყნებს Haskell 98 ანგარიშს, რომელმაც ენის სტაბილური ვერსია განსაზღვრა.

Haskell ენის გემოს მოსინჯვა

```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
               ys = [a | a ← xs, a ≤ x]
               zs = [b | b ← xs, b > x]
```

?

დაპროგრამება Haskell ენაზე



თავი 2 – პირველი ნაბიჯები

Hugs სისტემა

- Hugs-ი წარმოადგენს Haskell 98-ის რეალიზაციას და არის ყველაზე გავრცელებული Haskell სისტემა;
- Hugs-ის ინტერაქტიული ხასიათი კარგად მიესადაგება სწავლებისა და პროტოტიპირების მიზნებს;
- Hugs-ი მისაწვდომია web გვერდზე:

  www.haskell.org/hugs

Hugs-ის გაშვება

Unix სისტემაში Hugs-ი შეიძლება ოპერატიულად გაიშვას %-დან hugs ბრძანებით:

```
% hugs


__   __ __  __  ____   ___                          ________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                        Report bugs to: hugs-bugs@haskell.org
||   ||                        ________________________________________


>
```

«Hugs>სტრიქონი» ნიშნავს, რომ Hugs სისტემა მზადაა გამოსახულების შესაფასებლად

მაგალითად:

```
> 2+3*4
14

> (2+3)*4
20

> sqrt (3^2 + 4^2)
5.0
```

Prelude სტანდარტული ბიბლიოთეკა

Prelude.hs ფაილი მრავალ სტანდარტულ ფუნქციას შეიცავს. ზოგიერთ ნაცნობ რიცხვით ფუნციასთან ერთად, როგორიცაა + და *, ბიბლიოთეკა ასევე იძლევა არაერთ სასარგებლო ფუნქციას სიებზე სამუშაოდ.

- სიის პირველი ელემენტის გამოყოფა :

```
> head [1,2,3,4,5]
1
```

- პირველი ელემენტის მოსპობა სიაში:

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

- n-ური ელემენტის გამოყოფა სიაში:

```
> [1,2,3,4,5] !! 2
3
```

- სიის პირველი n ელემენტის გამოყოფა:

```
> take 3 [1,2,3,4,5]
[1,2,3]
```

- პირველი n ელემენტის განადგურება სიაში:

```
> drop 3 [1,2,3,4,5]
[4,5]
```

- სიის სიგრძის გამოთვლა:

```
> length [1,2,3,4,5]
5
```

- სიაში რიცხვთა ჯამის გამოთვლა:

```
> sum [1,2,3,4,5]
15
```

- სიაში რიცხვთა ნამრავლის გამოთვლა:

```
> product [1,2,3,4,5]
120
```

- ერთი სიის ბოლოში მეორე სიის დამატება:

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

- სიის რევერსი (შექცევა) :

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

ფუნქციის გამოყენება

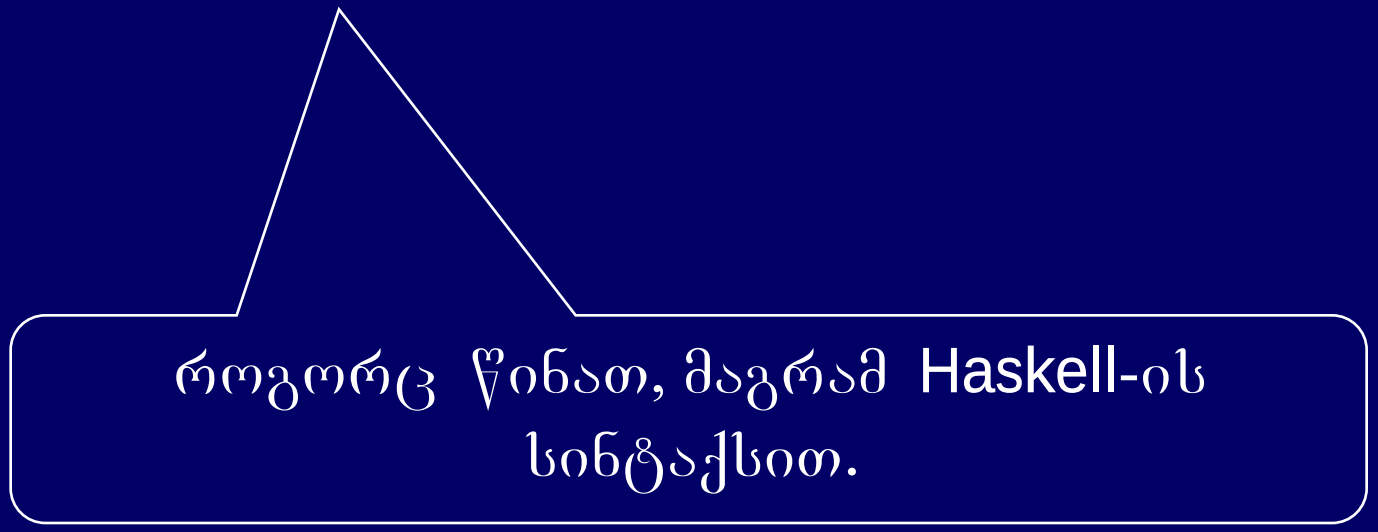
მათემატიკაში ფუნქციის გამოყენება აღინიშნება ფრჩხილების ხმარებით, ხოლო გამრავლება ხშირად აღინიშნება გვერდიგვერდ განლაგების ან ინტერვალის ხმარებით

```
f(a,b) + c d
```

გამოვიყენოთ f ფუნქცია a-სა და b-ს მიმართ და მივუმატოთ შედეგი c-სა და d-ს ნამრავლს.

Haskell-ში ფუნქციის გამოყენება აღინიშნება ხარვეზის (ინტერვალის) ხმარებით, ხოლო გამრავლება აღინიშნება * ნიშნის ხმარებით.

```
f a b + c*d
```



გარდა ამისა ითვლება, რომ ფუნქციის გამოყენებას აქვს უფრო მაღალი პრიორიტეტი, ვიდრე ყველა სხვა ოპერატორს.

```
f a + b
```

გაიგება როგორც (f a) + b, და არა როგორც f (a + b).

მაგალითები

მათემატიკა	Haskell ენა
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x)g(y)`	`f x * g y`

Haskell-ის სკრიპტები

- სტანდარტული prelude ბიბლიოთეკის ფუნქციათა დამატებით თქვენ შეგიძლიათ ასევე განსაზღვროთ საკუთარი ფუნქციებიც;

- ახალი ფუნქციები განისაზღვრება სკრიპტში – ტექსტურ ფაილში, რომელიც შედგება განსაზღვრებათა მიმდევრობისგან;

- შეთანხმებით, Haskell-ის სკრიპტებს, ჩვეულებრივ, აქვს **.hs** სუფიქსი (გაფართოება). ეს არ არის სავალდებულო, მაგრამ სასარგებლოა იდენტიფიკაციის მიზნებისათვის.

ჩემი პირველი სკრიპტი

Haskell-ზე სკრიპტის დამუშავებისას მიზანშეწონილია ორი გახსნილი ფანჯრის შენარჩუნება, ერთი მუშაობს სკრიპტის რედაქტორად, ხოლო მეორე – Hugs-ის გასაშვებად.

გაუშვით რედაქტორი, შეიტანეთ შემდეგი ორი ფუნქციის განსაზღვრება და შეინახეთ ეს **test.hs** ფაილში:

```
double x     = x + x

quadruple x = double (double x)
```

გახსნილი რედაქტორის პირობებში, მეორე ფანჯარაში სტარტი ეძლევა Hugs-ს ახალი სკრიპტით:

```
% hugs test.hs
```

ახლა იტვირთება Prelude.hs და test.hs და შეიძლება ფუნქციების გამოყენება ორივე სკრიპტიდან:

```
> quadruple 10
40

> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

გახსნილი Hugs-ის დროს დაბრუნდით რედაქტორში, დაუმატეთ ორი განსაზღვრება და შეინახეთ:

```
factorial n = product [1..n]

average ns  = sum ns `div` length ns
```

შენიშვნა:

- div ჩასმულია მარცხენა გამხსნელ (და არა მარჯვენა გამხსნელ) ფრჩხილებში;

- x `f` y მხოლოდ სინტაქსური შაქარია (f x y) - სათვის.

Hugs სისტემა ავტომატურად ვერ გამოავლენს სკრიპტის შეცვლას, ამიტომ გადატვირთვის ბრძანება წინ უნდა უსწრებდეს ახალ განსაზღვრებათა გამოყენებას:

```
> :reload
Reading file "test.hs"

> factorial 10
3628800

> average [1,2,3,4,5]
3
```

მოთხოვნები სახელებისადმი

- ფუნქციისა და არგუმენტის სახელები უნდა იწყებოდეს ქვედა რეგისტრის ასოდან. მაგალითად:

`myFun` `fun1` `arg_2` `x'`

- შეთანხმებით, სიის არგუმენტებს, ჩვეულებრივ, აქვს **s** სუფიქსი მათი სახელების ბოლოში:

`xs` `ns` `nss`

პროგრამათა დაპროექტების ტოპოლოგიური წესები

განსაზღვრებათა მომდევრობაში ყოველი მათგანი უნდა იწყებოდეს ზუსტად ერთსა და იმავე სვეტში:

```
a = 10
b = 20
c = 30
```

```
a = 10
 b = 20
c = 30
```

```
  a = 10
b = 20
  c = 30
```

პროგრამათა დაპროექტების ტოპოლოგიური წესები თავიდან გვაცილებს ცხადი სინტაქსის აუცილებლობას განსაზღვრებათა დაჯგუფების მისათითებლად.

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

შეიძლება

```
a = b + c
    where
      {b = 1;
       c = 2}
d = a * 2
```

არაცხადი დაჯგუფება

ცხადი დაჯგუფება

Hugs-ის სასარგებლო ბრძანებები

ბრძანება	მნიშვნელობა
:load *name*	*name* სკრიპტის ჩატვირთვა
:reload	მიმდინარე სკრიპტის გადატვირთვა
:edit *name*	*name* სკრიპტის რედაქტირება
:edit	მიმდინარე სკრიპტის რედაქტირება
:type *expr*	*expr*-ის ტიპის ჩვენება
:?	ყველა ბრძანების ჩვენება
:quit	Hugs-იდან გამოსვლა

სავარჯიშოები

(1) შეამოწმეთ Hugs-ით სლაიდები 2-8 და 14-17.

(2) გაასწორეთ სინტაქსური შეცდომები პროგრამაში ქვემოთ და ჩაატარეთ Hugs-ით მისი ტესტირება.

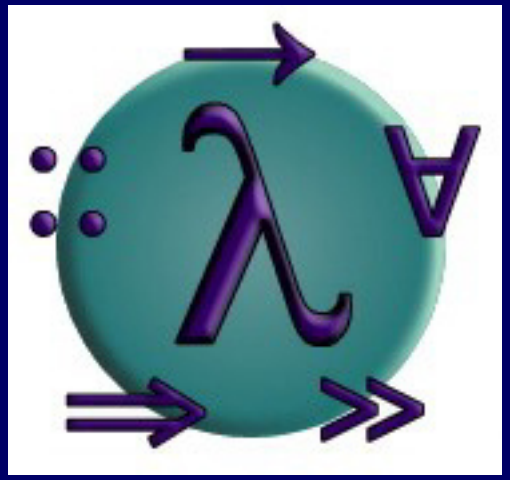
```
N = a `div` length xs
    where
       a = 10
      xs = [1,2,3,4,5]
```

(3) როგორ შეიძლება საბიბლიოთეკო **last** ფუნქცია, რომელიც სიის უკანასკნელ ელემენტს ირჩევს, ამ ლექციის ფუნქციებით იყოს განსაზღვრული?

(4) შეგიძლიათ კიდევ ერთი შესაძლო განსაზღვრება მოიფიქროთ?

(5) ამის მსგავსად მოიფიქრეთ: როგორ შეიძლება **init** საბიბლიოთეკო ფუნქცია, რომელიც სიაში უკანასკნელ ელემენტს სპობს, ამ ლექციის ფუნქციებით იყოს განსაზღვრული ორი სხვადასხვა გზით?

დაპროგრამება HASKELL ენაზე



თავი 3 – ტიპები და კლასები

რა არის ტიპი?

ტიპი არის დაკავშირებულ მნიშვნელობათა ერთობლიობის სახელი. მაგალითად, Haskell-ის საბაზო ტიპი

```
Bool
```

ორ ლოგიკურ მნიშვნელობას შეიცავს:

```
False
```

```
True
```

ტიპის შეცდომები

ფუნქციაში ერთი ან რამდენიმე არასწორი ტიპის არგუმენტთა გამოყენებას *ტიპის შეცდომა* ეწოდება:

```
> 1 + False
Error
```

1 რიცხვია, ხოლო False ლოგიკური მნიშვნელობაა, მაგრამ + მოითხოვს ორ რიცხვს.

ტიპები Haskell-ში

- თუ `e` გამოსახულების შეფასება `t` ტიპის მნიშვნელობაა, მაშინ `e`-ს `t` ტიპი აქვს. ეს ასე ჩაიწერება:

```
e :: t
```

- ნებისმიერ სწორად შედგენილ გამოსახულებას აქვს ტიპი, რომელიც ავტომატურად შეიძლება დადგინდეს კომპილაციისას *ტიპის გამოტანად* წოდებული პროცესის საშუალებით.

- თიპის ყველა შეცდომა კომპილაციის დროს ვლინდება, რაც ანიჭებს პროგრამებს მეტ უსაფრთხოებას და სისწრაფეს, რადგან ქრება ტიპების შემოწმების აუცილებლობა შესრულებისას.

- **Hgs**-ინტერპრეტატორში **:type** ბრძანება ადგენს გამოსახულების ტიპს ამ გამოსახულების შეუფასებლად:

```
> not False
True

> :type not False
not False :: Bool
```

საბაზო ტიპები

Haskell-ს *საბაზო ტიპების* გარკვეული რაოდენობა აქვს. ესენია:

- `Bool` - ლოგიკური სიდიდეები
- `Char` - განმხოლოებული სიმბოლოები
- `String` - სიმბოლური სტრიქონები
- `Int` - ფიქსირებული სიზუსტის მთელი
- `Integer` - ნებისმიერი სიზუსტის მთელი
- `Float` - რიცხვები მცურავი წერტილით

სიის ტიპები

სია არის ერთისა და იმავე ტიპის მნიშვნელობათა მიმდევრობა:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```

საზოგადოდ:

> [t] არის იმ სიის ტიპი, რომელიც t ტიპის ელემენტებს შეიცავს.

შენიშვნა:

- სიის ტიპი არაფერს გვეუბნება მის სიგრძეზე:

```
[False,True]        :: [Bool]

[False,True,False] :: [Bool]
```

- ელემენტთა ტიპები არ იზღუდება. მაგალითად, დასაშვებია სიათა სიებიც კი:

```
[['a'],['b','c']] :: [[Char]]
```

კორტეჟის ტიპები

კორტეჟი – სხვადასხვა ტიპის სიდიდეთა მიმდევრობაა:

```
(False,True)      :: (Bool,Bool)

(False,'a',True)  :: (Bool,Char,Bool)
```

საზოგადოდ:

(t1,t2,...,tn) – n-კორტეჟთა ტიპია, რომელთა კომპონენტებს აქვს ti ტიპი, სადაც i იღებს მნიშვნელობებს 1...n.

შენიშვნა:

- კორტეჟის ტიპი განსაზღვრავს მის ზომას:

```
(False,True)       :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- კომპონენტთა ტიპები არ იზღუდება:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b'])  :: (Bool,[Char])
```

ფუნქციის ტიპები

ფუნქცია არის ერთი ტიპის სიდიდეთა შეპირისპირება მეორე ტიპის სიდიდეებთან:

```
not     :: Bool → Bool

isDigit :: Char → Bool
```

საზოგადოდ:

> t1 → t2 ფუნქციების ტიპია, რომლებიც ასახავენ t1 ტიპის სიდიდეებს t2 ტიპის სიდიდეებად.

შენიშვნა:

- ისარი → შედის კლავიატურიდან ასე: `->`.

- არგუმენტისა და შედეგის ტიპები არ იზღუდება. მაგალითად, ფუნქციები რამდენიმე არგუმენტით ან შედეგით შესაძლებელია სიების ან კორტეჟების საშუალებით:

```
add          :: (Int,Int) → Int
add (x,y)     = x+y

zeroto       :: Int → [Int]
zeroto n      = [0..n]
```

კარირებული ფუნქციები

მრავალარგუმენტიანი ფუნქციები ასევე შესაძლებელია ჩავწეროთ შედეგებად დაბრუნებული ფუნქციების სახით:

```
add'     :: Int → (Int → Int)
add' x y = x+y
```

add' იღებს **x** მთელ რიცხვს და გვიბრუნებს **add' x** ფუნქციას. თავის მხრივ, ეს ფუნქცია იღებს **y** მთელ რიცხვს და გვიბრუნებს **x+y** შედეგს.

შენიშვნები:

- add და add' ერთსა და იმავე საბოლოო შედეგს იძლევა, მაგრამ add იღებს ორ არგუმენტს ერთდროულად, მაშინ როცა add' იღებს მათ რიგრიგობით:

```
add  :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

- ფუნქციებს, რომლებიც იღებს თავიანთ არგუმენტებს რიგრიგობით, *კარირებულს* უწოდებენ – ჰასკელ კარის პატივისცემის ნიშნად (იგი მუშაობდა ასეთ ფუნქციებთან).

- ორზე მეტი არგუმენტის მქონე ფუნქციები შეიძლება იყოს *კარირებული* ერთმანეთში ჩალაგებული ფუნქციების დაბრუნებით:
- .

```
mult       :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult იღებს **x** მთელს და გვიბრუნებს **mult x** ფუნქციას, რომელიც თავის მხრივ იღებს **y** მთელს და გვიბრუნებს **mult x y** ფუნქციას, უკანასკნელი იღებს **z** მთელს და გვიბრუნებს **x*y*z** შედეგს.

რით არის სასარგებლო კარირება?

კარირებული ფუნქციები უფრო მოხერხებულია გამოსაყენებლად, ვიდრე ფუნქციები კორტეჟებზე, რადგან პრაქტიკული ფუნქცია ხშირად შეიძლება იყოს ხელოვნურად გადაქცეული კარირებულ ფუნქციად ნაწილობრივი გამოყენებით.
მაგალითად:

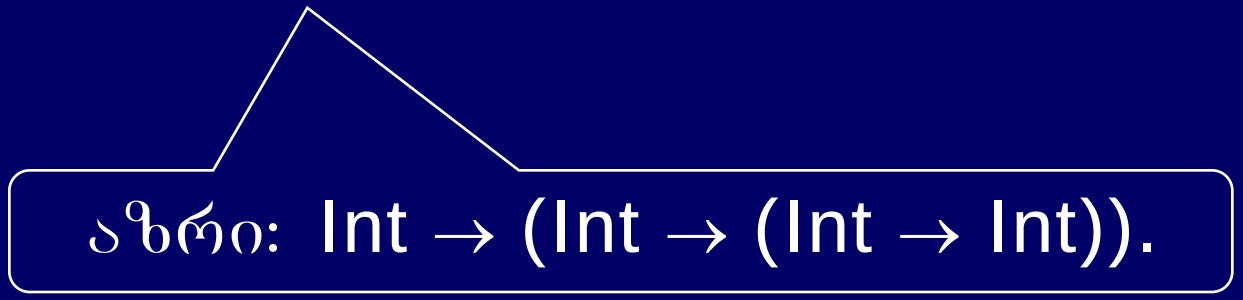
```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

შეთანხმებები კარირებისათვის

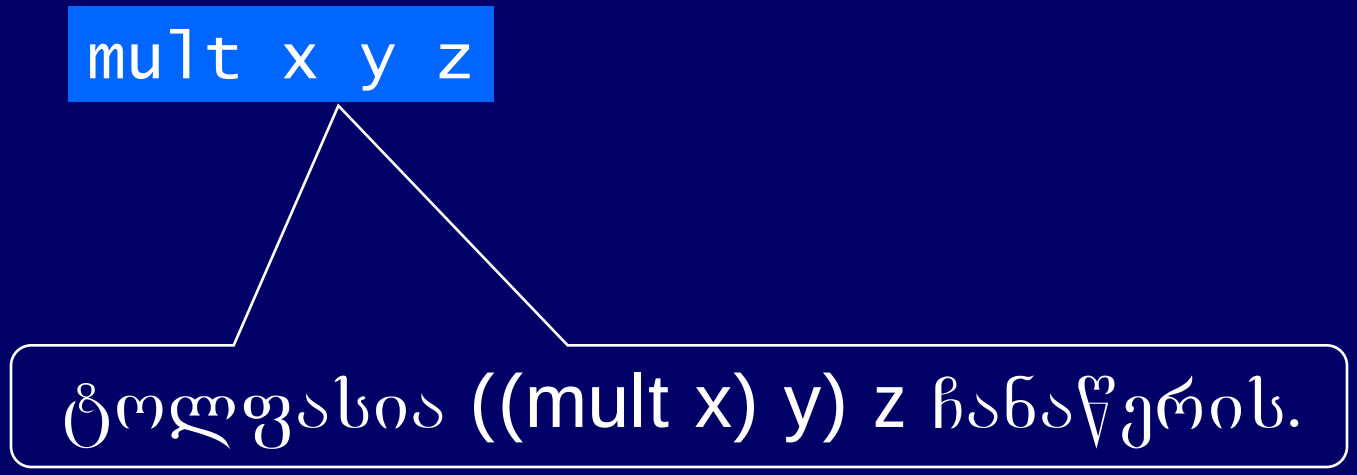
ზედმეტი ფრჩხილების თავიდან ასაცილებლად კარირებული ფუნქციების გამოყენებისას, მიღებულია ორი მარტივი შეთანხმება:

- ისარი $\rightarrow$ ასოციაციურია მარჯვნივ.

$$\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$$



- შედეგად, ბუნებრივია ფუნქციისათვის მარცხნიდან დაკავშირების გამოყენება.



თუ კორტეჟირება ცხადად არ მოითხოვება, ყველა ფუნქცია Haskell-ში, ჩვეულებრივ, კარირებული ფორმით განისაზღვრება.

პოლიმორფული ფუნქციები

ფუნქციას *პოლიმორფული* (მრავალფორმიანი) ეწოდება, თუ მისი ტიპი შეიცავს ცვლადის ერთ ან რამდენიმე ტიპს.

```
length :: [a] → Int
```

ნებისმიერი a ტიპისათვის length იღებს ტიპის მნიშვნელობათა სიას და გვიბრუნებს მთელ რიცხვს.

შენიშვნა:

- ცვლადების ტიპი შეიძლება დამუშავდეს სხვადასხვა ტიპისათვის სხვადასხვა ვითარებაში:

```
> length [False,True]
2

> length [1,2,3,4]
4
```

a = Bool

a = Int

- ცვლადების ტიპი უნდა იწყებოდეს ქვედა რეგისტრის ასოთი და, ჩვეულებრივ, მას აქვს a, b, c და ა.შ. სახელი.

- სტანდარტულ prelude ფაილში განსაზღვრულ ფუნქციათა შორის მრავალი პოლიმორფულია. მაგალითად:

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]

id   :: a → a
```

გადატვირთული ფუნქციები

პოლიმორფულ ფუნქციას *გადატვირთული* ეწოდება, თუ მისი ტიპი შეიცავს კლასის ერთ ან რამდენიმე შეზღუდვას.

```
sum :: Num a ⇒ [a] → a
```

ნებისმიერი რიცხვითი **a** ტიპისათვის **sum** იღებს **a** ტიპის მნიშვნელობათა სიას და გვიბრუნებს **a** ტიპის მნიშვნელობას.

შენიშვნა:

- პირობებით შეზღუდული ცვლადების ტიპი შეიძლება დამუშავდეს ნებისმიერი ტიპისათვის, რომლებიც შეზღუდვებს აკმაყოფილებს:

```
> sum [1,2,3]
6

> sum [1.1,2.2,3.3]
6.6

> sum ['a','b','c']
ERROR
```

a = Int

a = Float

Char არ არის რიცხვითი ტიპი

- Haskell-ს ტიპთა რიგი კლასი აქვს, მათ შორის :

  `Num` - რიცხვითი ტიპები

  `Eq` - ტოლობის ტიპები

  `Ord` - მოწესრიგებული ტიპები

- მაგალითად:

```
(+)  :: Num a ⇒ a → a → a
(==) :: Eq a  ⇒ a → a → Bool
(<)  :: Ord a ⇒ a → a → Bool
```

რჩევები და დარიგებები

- Haskell-ში ახალი ფუნქციის განსაზღვრა მიზანშეწონილია დაიწყოთ მისი ტიპის ჩაწერით;

- კარგი პრაქტიკაა სკრიპტის ფარგლებში ყოველი ახალი აღწერილი ფუნქციის ტიპის მითითება;

- იმ პოლიმორფულ ფუნქციათა ტიპების ფორმულირებისას, რომლებიც იყენებს რიცხვებს, ტოლობებს ან მოწესრიგებას, იზრუნეთ შეზღუდვათა აუცილებელი კლასის ჩართვაზე.

სავარჯიშოები

(1) როგორია შემდეგი მნიშვნელობების ტიპები?

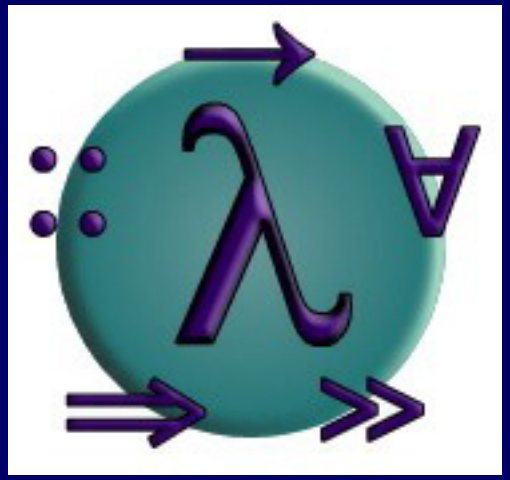
```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

([False,True],['0','1'])

[tail,init,reverse]
```

(2) როგორია შემდეგი ფუნქციების ტიპები?

```
second xs       = head (tail xs)

swap (x,y)      = (y,x)

pair x y        = (x,y)

double x        = x*2

palindrome xs = reverse xs == xs

twice f x       = f (f x)
```

(3) შეამოწმეთ თქვენი პასუხები Hugs-ით.

დაპროგრამება HASKELL ენაზე



თავი 4 – განმსაზღვრელი ფუნქციები

პირობითი გამოსახულებები

დაპროგრამების ენათა უმრავლესობის მსგავსად, Haskell-შიც ფუნქციები შეიძლება განისაზღროს *პირობითი გამოსახულებებით* :

```
abs  :: Int → Int
abs n = if n ≥ 0 then n else -n
```

იღებს მთელ «n» რიცხვს და გვიბრუნებს n-ს, თუ n არაუარყოფითი მთელია, ხოლო წინააღმდეგ შემთხვევაში «-n»-ს.

პირობითი გამოსახულებები შეიძლება ჩალაგებული იყოს ერთმანეთში:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
                  if n == 0 then 0 else 1
```

შენიშვნა:

- Haskell-ის პირობით გამოსახულებებში ყოველთვის უნდა არსებობდეს `else` შტო, რათა გამოირიცხოს შესაძლო არაცალსახა გაგება ამოცანებში ჩალაგებული პირობებით.

დაცული განტოლებები

ლოგორც პირობითი ოპერატორების ალტერნატივა, ფუნქციები შეიძლება განისაზღვროს *დაცული განტოლებებით*:

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

მეორდება წინა შემთხვევა, მაგრამ დამცველი განტოლებების გამოყენებით.

კითხვადობის გასაუმჯობესებლად დაცული განტოლებები შეიძლება გამოიყენებოდეს განსაზღვრისათვის რამდენიმე პირობის მონაწილეობით:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

შენიშვნები

- მთლიანობაში otherwise პირობა განსაზღვრულია prelude-ში otherwise = True ფორმით.

შესაბამისობა შაბლონთან

მრავალი ფუნქცია განსაკუთრებით მკაფიოდ განისაზღვრება მათ არგუმენტთა შაბლონების გამოყენებისას:

```
not       :: Bool → Bool
not False = True
not True  = False
```

not ასახავს False-ს True-დ, ხოლო True-ს False-ად.

ფუნქციები ხშირად შეიძლება იყოს განსაზღვრული მრავალი სხვადასხვა ხერხით შაბლონთან შედარებისას. მაგალითად:

```
(&&)             :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

ეს უფრო კომპაქტურადაც განისაზღვრება:

```
True && True = True
_    &&  _   = False
```

მაგრამ შემდეგი განსაზღვრება უფრო ეფექტურია, რადგან იგი არ მიმართავს მეორე არგუმენტის შეფასებას, როცა პირველ არგუმენტს False მნიშვნელობა აქვს:

```
True  && b = b
False && _ = False
```

შენიშვნა:

- ქვედა ხაზგასმის «_» სიმბოლო წარმოადგენს ჩასმის შაბლონს ნებისმიერი მნიშვნელობის არგუმენისათვის.

- შაბლონების შედარება *რიგრიგობით* ხდება. მაგალითად, შემდეგი განსაზღვრება ყოველთვის გვიბრუნებს False მნიშვნელობას:

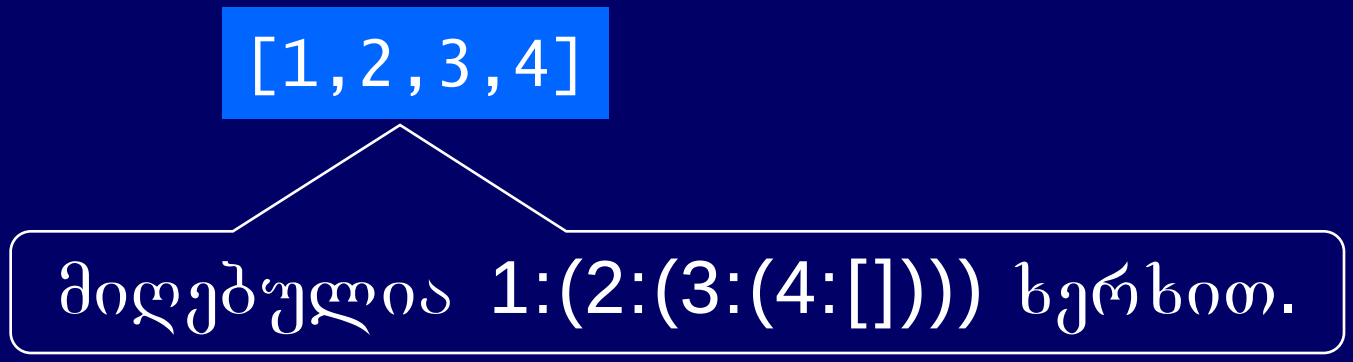
```
_    && _    = False
True && True = True
```

- შაბლონები კრძალავს ცვადების გამეორებას. მაგალითად, შემდეგი განსაზღვრება შეცდომას იძლევა:

```
b && b = b
_ && _ = False
```

შაბლონების სია

ყოველი არაცარიელი (გაუნადგურებელი) სია აიგება გამეორების **(:)** ოპერატორით, რომელსაც **“cons”** სახელი აქვს და ელემენტს სიის დასაწყისში ამატებს:



ფუნქციები სიებზე შეიძლება იყოს განსაზღვრული x:xs შაბლონებით.

```
head        :: [a] → a
head (x:_)   = x

tail        :: [a] → [a]
tail (_:xs) = xs
```

head და tail ფუნქციები ასახავს არაცარიელი სიის პირველ და ყველა დარჩენილ ელემენტს შესაბამისად.

შენიშვნა:

- x:xs შაბლონები გამოიყენება მხოლოდ არა-ცარიელი სიებისათვის:

```
> head []
შეცდომა
```

- x:xs შაბლონები უნდა განთავსდეს ფრჩხილებში, ვინაიდან რეალიზაციას პრიორიტეტი ენიჭება **(:)**-თან შედარებით. მაგალითად, შემდეგი განსაზღვრება მცდარია: :

```
head x:_ = x
```

შაბლონები მთელ რიცხვებზე

როგორც მათემატიკაში, ფუნქციები მთელ რიცხვებზე განისაზღვრება ე.წ. n+k შაბლონებით, სადაც n მთელი ცვლადია, ხოლო k>0 - მთელი მუდმივაა:

```
pred          :: Int → Int
Pred 0         = 0
pred (n+1)     = n
```

pred ასახავს მთელს, რომელიც წინ უძღვის შეტანილს.

შენიშვნა:

- n+k შაბლონები შეესაბამება მხოლოდ მთელ რიცხვებს, რომლებიც ≥ k-ზე.

```
> pred (-1)
შეცდომა
```

- n+k შაბლონებში უნდა იყოს გამოყენებული ფრჩხილები, ვინაიდან სამომხმარებლო პროგრამას პრიორიტეტი გააჩნია + ოპერაციასთან შედარებით. მაგალითად, შემდეგი განსაზღვრება შეცდომას იძლევა:

```
pred n+1 = n
```

ლამბდა-გამოსახულებები

ფუნქცია შეიძლება აიგოს მისი სახელის მიუთითებლად ლამბდა-გამოსახულების დახმარებით:

$\lambda x \rightarrow x+x$

უსახელო ფუნქცია, რომელიც შესასვლელზე იღებს x რიცხვს და გვიბრუნებს x+x შედეგს.

შენიშვნა:

- λ სიმბოლო წარმოადგენს ბერძნულ ასოს და იგი აიკრიფება კლავიატურაზე როგორც \.

- მათემატიკაში უსახელო ფუნქციები, ჩვეულებრივ, აღინიშნება $\mapsto$ სიმბოლოთი, მაგალითად, $x \mapsto x+x$.

- Haskell-ში λ სიმბოლოს გამოყენება უსახელო ფუნქციებისათვის დაკავშირებულია ლამბდა-აღრიცხვასთან - ფუნციათა თეორიასთან, რომელსაც ეფუძნება ეს ენა.

რატომ არის ლამბდა სასარგებლო?

ლამბდა-გამოსახულებები შეიძლება გამოვიყენოთ ფორმალური არსის მისაცემად კარირებით განსაზღვრული ფუნქციებისათვის.

მაგალითად:

```
add x y = x+y
```

შესაძლებელია:

```
add = λx → (λy → x+y)
```

ლამბდა–გამოსახულებები სასარგებლოა აგრეთვე იმ ფუნქციათა განსაზღვრისას, რომლებიც გვიბრუნებს ფუნქციებს როგორც შედეგებს.

მაგალითად,

```
const    :: a → b → a
const x _ = x
```

უფრო ბუნებრივად განისაზღვრება ასეთი ჩანაწერით:

```
const  :: a → (b → a)
const x = λ_ → x
```

ლამბდა-გამოსახულებათა გამოყენება შეიძლება ისეთი ფუნქციის დასახელების თავიდან ასაცილებლად, რომელსაც მხოლოდ ერთჯერ მიმართავენ.

მაგალითად,

```
odds n = map f [0..n-1]
         where
           f x = x*2 + 1
```

შეიძლება იყოს დაყვანილი გამოსახულებამდე:

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

სექციები

ორ არგუმენტს შორის დაწერილი ოპერატორი შეიძლება გარდავსახოთ ფუნქციად, სადაც ფრჩხილებს შორის მოთავსებული ეს ოპერატორი ხსენებული ორი არგუმენტის წინ დგას.

მაგალითად:

```
> 1+2
3

> (+) 1 2
3
```

ეს შეთანხმება საშუალებას იძლევა ოპერატორის ერთ-ერთი არგუმენტი ჩავრთოთ ფრჩხილებს შორის.

მაგალითად:

```
> (1+) 2
3

> (+2) 1
3
```

საერთოდ, თუ ⊕ არის ოპერატორი, მაშინ (⊕), (x⊕) და (⊕y) ფორმის ფუნქციებს *სექციები* ეწოდება.

რატომ არის სასარგებლო სექციები?

ზოგჯერ სასარგებლო ფუნქციები მარტივად შეიძლება იყოს აგებული სექციების საშუალებით. მაგალითად:

- `(1+)` - მოწესრიგების ფუნქცია
- `(1/)` - შებრუნების ფუნქცია
- `(*2)` - გაორკეცების ფუნქცია
- `(/2)` - განახევრების ფუნქცია

სავარჯიშოები

(1) განვიხილოთ **safetail** ფუნქცია, რომელიც **tail** ფუნქციის მსგავსად იქცევა, იმის გამოკლებით, რომ **safetail** ფუნქცია ასახავს ცარიელ სიას კვლავ ცარიელ სიად, მაშინ როცა **tail** ფუნქცია ამ შემთხვევაში შეცდომას იძლევა. **safetail** ფუნქციის საშუალებით განსაზღვრეთ:

(a) პირობითი გამოსახულება;
(b) დაცული განტოლებები;
(c) შაბლონთან შესაბამისობა.

კარნახი: საბიბლიოთეკო ფუნქცია null :: [a] → Bool შეიძლება გამოვიყენოთ შესამოწმებლად, თუ სია ცარიელია.

(2) მიუთითეთ **or** (ან) ლოგიკური (||) ოპერატორის სამი შესაძლო განსაზღვრება შაბლონთან შესაბამისობის გამოყენებით.

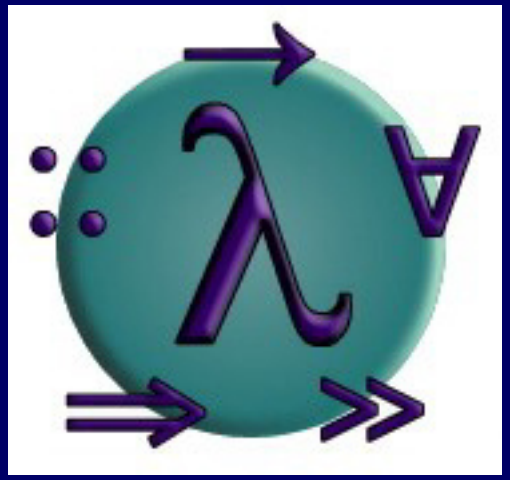
(3) განსაზღვრეთ ხელახლა (&&) ოპერატორის შემდეგი ვერსია, უპირატესად, პირობითი ოპერატორების (და არა შაბლონების) გამოყენებით :

```
True && True = True
_    && _    = False
```

(4) იგივე გააკეთეთ შემდეგი ვერსიისათვის:

```
True  && b = b
False && _ = False
```

დაპროგრამება Haskell ენაზე



თავი 5 – სიის კონსტრუქტორები

სიმრავლის კონსტრუქტორები

მათემატიკაში კონსტრუქტორის ცნება გამოიყენება ახალი სიმრავლეების ასაგებად ძველი სიმრავლეების საფუძველზე:

$$\{x^2 \mid x \in \{1...5\}\}$$

x^2 რიცხვების {1,4,9,16,25} სიმრავლე, სადაც x არის {1...5} სიმრავლის ელემენტი.

სიათა კონსტრუქტორები

ჰასკელში კონსტრუქტორის მსგავსი ცნება შეიძლება გამოვიყენოთ ახალი სიების ასაგებად ძველი სიების საფუძველზე:

```
[x^2 | x ← [1..5]]
```

x^2 რიცხვთა [1,4,9,16,25] სია, სადაც x არის [1..5] სიის ელემენტი.

შენიშვნა:

- x ← [1..5] გამოსახულებას *გენერატორი* ეწოდება, რადგან იგი გვეუბნება, თუ საიდან მიიღება x-ის მნიშვნელობები.

- კონსტრუქტორს მძიმეებით გამოყოფილი *რამდენიმე* გენერატორი შეიძლება გააჩნდეს. მაგალითად:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

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

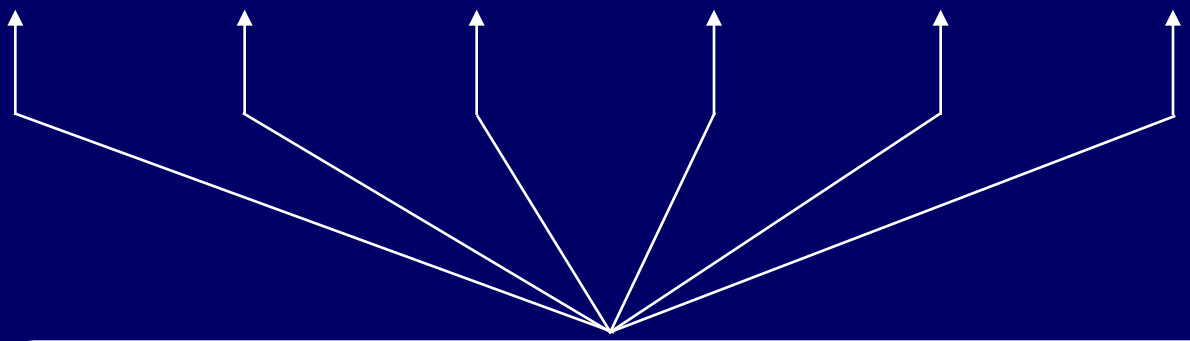
- გენერატორების თანამიმდევრიბის შეცვლისას იცვლება ელემენტების მიმდევრობაც საბოლოო სიაში:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

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

- რამდენიმე გენერატორი ჩალაგებული ციკლების მსგავსია, სადაც მომდევნო გენერატორები უფრო ღრმად ჩალაგებული ციკლებია, რომელთა მნიშვნელობები გაცილებით ხშირად იცვლება.

- მაგალითად:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

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



x ← [1,2,3] – მომდევნო გენერატორია და ამიტომ შედეგის თითოეულ წყვილში ყველაზე ხშირად x-კომპონენტი იცვლება.

დამოკიდებული გენერატორები

მომდევნო გენერატორები შეიძლება დამოკიდებული იყოს ცვლადებზე, რომლებიც უფრო წინა გენერატორებით შეყვანება.

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

რიცხვთა ყველა (x,y) წყვილის [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] სია, როცა x და y – ელემენტებია [1..3] სიიდან და ამასთან ერთად y ≥ x.

დამოკიდებული გენერატორის საშუალებით ჩვენ შეგვიძლია განვსაზღვროთ საბიბლიოთეკო *concat* ფუნქცია, რომელიც აერთიანებს ერთ სიას მეორესთან:

```
concat    :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

მაგალითად:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

მცვლები

სიის კონსტრუქტორებს შეუძლია *მცვლების* გამოყენება იმ მნიშვნელობების შესაზღუდავად, რომლებიც ნაწარმოებია წინა გენერატორებით

```
[x | x ← [1..10], even x]
```

ყველა x რიცხვის [2,4,6,8,10] სია, როცა x არის [1..10] სიის ლუწი ელემენტი.

მცველის საშუალებით შესაძლებელია ფუნქციის განსაზღვრა, რომელიც ასახავს დადებით მთელ რიცხვს თავისი *ფაქტორების* (*გამყოფების*) სიაში:

```
factors  :: Int → [Int]
factors n =
   [x | x ← [1..n], n `mod` x == 0]
```

მაგალითად:

```
> factors 15

[1,3,5,15]
```

დადებითი მთელი რიცხვი *მარტივია*, თუ მისი ფაქტორებია მხოლოდ 1 და თავად ეს რიცხვი. factors ფუნქციის გამოყენებით შესაძლებელია ახალი ფუნქციის განსაზღვრა, რომელიც ადგენს, თუ არის რიცხვი მარტივი:

```
prime  :: Int → Bool
prime n = factors n == [1,n]
```

მაგალითად:

```
> prime 15
False

> prime 7
True
```

მცველის საშუალებით შესაძლებელია ახლა ისეთი ფუნქციის განსაზღვრა, რომელიც გვიბრუნებს ყველა მარტივი რიცხვის სიას მოცემულ მნიშვნელობამდე:

```
primes  :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

მაგალითად:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

Zip ფუნქცია

სასარგებლო საბიბლიოთეკო zip ფუნქცია ასახავს ორ სიას მათი შესაბამისი ელემენტების წყვილთა ერთ სიად.

```
zip :: [a] → [b] → [(a,b)]
```

მაგალითად:

```
> zip ['a','b','c'] [1,2,3,4]

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

zip ფუნქციის საშუალებით შეიძლება განისაზღვროს ფუნქცია, რომელიც გვიბრუნებს სიის მომიჯნავე ელემენტების ყველა წყვილის სიას:

```
pairs   :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

მაგალითად:

```
> pairs [1,2,3,4]

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

pairs ფუნქციის საშუალებით შესაძლებელია განისაზღვროს ფუნქცია, რომელიც ადგენს, თუ არის სიის ელემენტები დახარისხებული:

```
sorted   :: Ord a ⇒ [a] → Bool
sorted xs =
   and [x ≤ y | (x,y) ← pairs xs]
```

მაგალითად:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

zip ფუნქციის საშუალებით შეიძლება განისაზღვროს ფუნქცია, რომელიც გვიბრუნებს სიაში წარმოდგენილი რაღაც მნიშვნელობის ყველა პოზიციის სიას (პოზიცია ინომრება ნულიდან!):

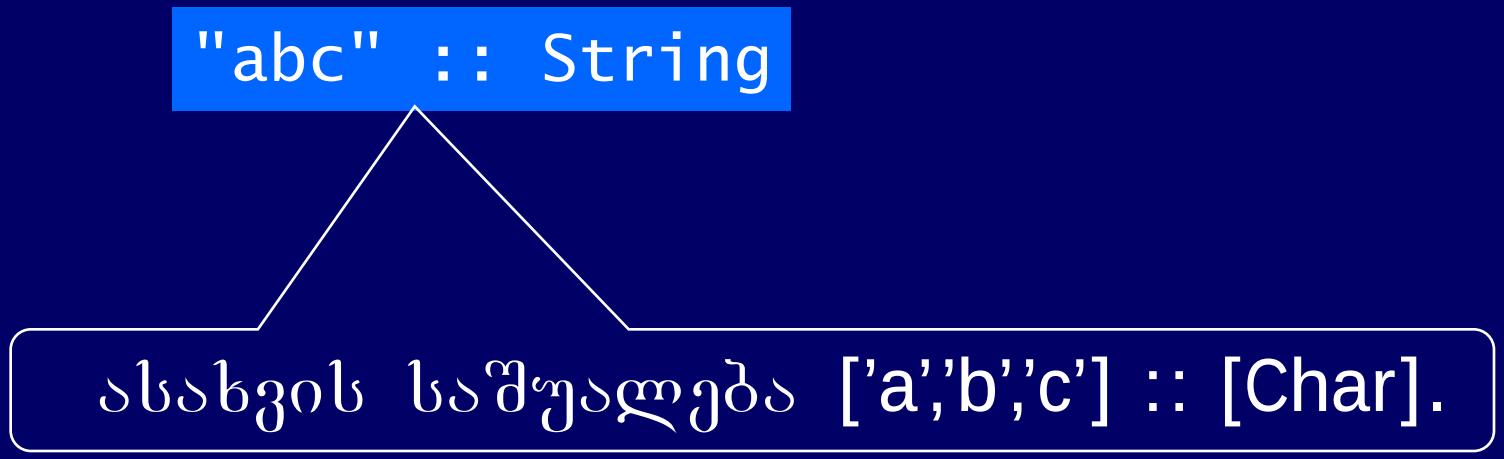
```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
   [i | (x',i) ← zip xs [0..n], x == x']
   where n = length xs - 1
```

მაგალითად:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

სტრიქონის კონსტრუქტორი

სტრიქონი არის ორმაგ ფრჩხილებში ჩასმული სიმბოლოების მიმდევრობა. ამის მიუხედავად, შინაგანად, სტრიქონები წარმოდგენილია როგორც სიმბოლოთა სიები:



რადგან სტრიქონები უბრალოდ სიების სპეციალური სახეა, სიებზე მომუშავე ნებისმიერი *პოლიმორფული* ფუნქცია შეიძლება სტრიქონებზეც გამოვიყენოთ:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

გარდა ამისა, სიის კონსტრუქტორი შეიძლება გამოვიყენოთ ფუნქციების განსაზღვრისათვის სტრიქონებზე. მაგალითად, ჩავწეროთ ფუნქცია, რომელიც ანგარიშობს ნუსხური ასოების რაოდენობას სტრიქონში:

```
lowers   :: String → Int
lowers xs =
   length [x | x ← xs, isLower x]
```

მაგალითად:

```
> lowers "Haskell"

6
```

სავარჯიშოები

(1) (x,y,z) დადებითი მთელი რიცხვების სამეულს პითაგორას სამეული ეწოდება, თუ $x^2 + y^2 = z^2$. სიის კონსტრუქტორის საშუალებით განსაზღვრეთ ფუნქცია:

```
pyths :: Int → [(Int,Int,Int)]
```

იგი გვიბრუნებს პითაგორას ყველა სამეულს, რომელთა კომპონენტები [1..n] სიიდან მოცემულ ზღვრულ n სიდიდეს არ აღემატება. მაგალითად:

```
> pyths 5
[(3,4,5),(4,3,5)]
```

(2) დადებითი მთელი რიცხვი *სრულყოფილია*, თუ იგი უდრის ყველა თავისი ფაქტორის (გამყოფის) ჯამს (თავად ამ რიცხვის გარეშე). სიის კონსტრუქტორის გამოყენებით განსაზღვრეთ ფუნქცია

```
perfects :: Int → [Int]
```

რომელიც გვიბრუნებს ყველა სრულყოფილი რიცხვის სიას მოცემულ მნიშვნელობამდე. მაგალითად:

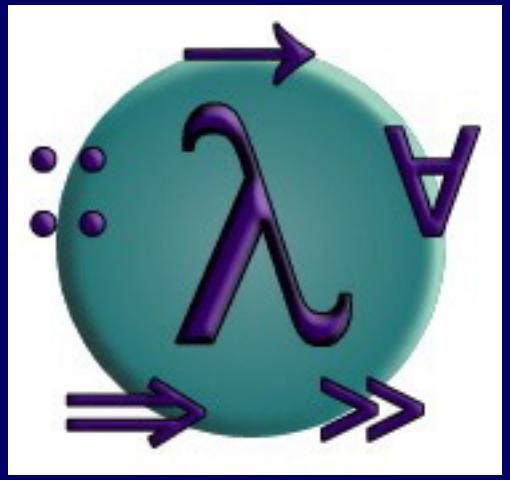
```
> perfects 500

[6,28,496]
```

(3) მთელ რიცხვთა ორი xs და ys ერთისა და იმავე n სიგრძის სიის *სკალარული ნამ-რავ-ლი* მოიცემა შესაბამისი მთელი რიცხვების ნამრავლების ჯამით:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

სიის კონსტრუქტორის გამოყენებით გან-საზღვრეთ ფუნქცია, რომელიც ორი სიის სკალარულ ნამრავლს ანგარიშობს.

დაპროგრამება HASKELL ენაზე



თავი 6 – რეკურსიული ფუნქციები

შესავალი

როგორც ვნახეთ, მრავალი ფუნქცია ბუნებრივად შეიძლება განისაზღვრებოდეს სხვა ფუნქციათა საშუალებით.

```
factorial  :: Int → Int
factorial n = product [1..n]
```

ფაქტორიალი გარდაქმნის ნებისმიერ მთელ n რიცხვს 1-დან n-მდე რიცხვების ნამრავლად.

გამოსახულებები ფასდება პროცესით, როცა ფუნქციათა გამოყენება მათი არგუმენტების მიმართ ეტაპობრივია. მაგალითად:

```
  factorial 4
=
  product [1..4]
=
  product [1,2,3,4]
=
  1*2*3*4
=
  24
```

რეკურსიული ფუნქციები

ჰასკელში ფუნქცია შეიძლება განისაზღვროს თავისივე საშუალებით. ასეთ ფუნქციას *რეკურსიულს* უწოდებენ.

```
factorial 0     = 1
factorial (n+1) = (n+1) * factorial n
```

ფაქტორიალი ასახავს ნულს ერთიანად და ნებისმიერ სხვა დადებით მთელ რიცხვს თავად ამ რიცხვისა და წინა რიცხვის ფაქტრიალის ნამრავლად.

მაგალითად:

```
  factorial 3
= 3 * factorial 2
= 3 * (2 * factorial 1)
= 3 * (2 * (1 * factorial 0))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
= 6
```

შენიშვნა:

- factorial 0 = 1 მისალები თანაფარდობაა, ვინაიდან 1 არის იგივეობა გამრავლებისათვის: 1*x = x = x*1.

- რეკურსიული განსაზღვრება უვარგისია უარყოფით მთელ რიცხვებზე, რადგან საბაზო შემთხვევა არასოსოდეს მიიღწევა :

```
> factorial (-1)

Error: Control stack overflow
```

შეცდომა: კონტროლის სტეკის გადავსება

რატომ არის რეკურსია სასარგებლო?

- ზოგიერთი ფუნქცია, ფაქტორიალის მსგავსად, უფრო ადვილად განისაზღვრება სხვა ფუნქციებით.

- მაგრამ, როგორც ვნახავთ, მრავალი ფუნქცია ბუნებრივად განისაზღვრება საკუთარი თავის მეშვეობით.

- რეკურსიით განსაზღვრულ ფუნქციათა თვისებები შეიძლება დამტკიცდეს მათემატიკური ინდუქციის მარტივი, მაგრამ მძლავრი მეთოდით.

რეკურსია სიებზე

ლეკურსია არ შემოიფარგლება რიცხვებით, იგი შეიძლება გამოიყენებოდეს ფუნქციათა განსაზღვრისათვისაც სიებზე.

```
product        :: [Int] → Int
product []      = 1
product (n:ns) = n * product ns
```

product ასახავს ცარიელ სიას ერთად, და არაცარიელ სიას მისი თავის ნამრავლით product ფუნქციის მნიშვნელობაზე სიის კუდისათვის.

მაგალითად:

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * (3 * product [4])
=
  2 * (3 * (4 * product []))
=
  2 * (3 * (4 * 1))
=
  24
```

რეკურსიის იმავე მოდელის გამოყენებით შესაძლებელია განისაზღვროს სიგრძის length ფუნქცია სიებზე.

```
length        :: [a] → Int
length []      = 0
length (_:xs) = 1 + length xs
```

length ფუნქცია წარმოადგენს ცარიელ სიას ნულად, ხოლო ნებისმიერ არაცარიელ სიას - მისი კუდის სიგრძის მომდევნო მნიშვნელობად (ე.ი. მემკვიდრედ).

მაგალითად:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

რეკურსიის მსგავსი მოდელის გამოყენებით ჩვენ შეგვიძლია განვსაზღვროთ სიებზე reverse ფუნქცია.

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

reverse ფუნქცია წარმოადგენს ცარიელ სიას ცარიელ სიად, ხოლო ნებისმიერ არაცარიელ სიას – reverse ფუნქციად კუდზე, რომელსაც ამ სიის თავი მიეწერება ბოლოში.

მაგალითად:

```
  reverse [1,2,3]
=
  reverse [2,3] ++ [1]
=
  (reverse [3] ++ [2]) ++ [1]
=
  ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

მრავალი არგუმენტი

ფუნქციები ორზე მეტი არგუმენტით ასევე შეიძლება განისაზღვროს რეკურსიით. მაგალითად:

- ორი სიის ელემენტთა «შეკვრა ელვა-საკეტით» (zip [1,2,3] ['a','b'] გვიბრუნებს [(1,'a'),(2,'b')])

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

- სიიდან პირველი n ელემენტის ამოღება:

```
drop                :: Int → [a] → [a]
drop 0      xs       = xs
drop (n+1) []        = []
drop (n+1) (_:xs) = drop n xs
```

- ორი სიის დამატება:

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

სწრაფი დახარისხება Quicksort

quicksort ალგორითმი მთელ რიცხვთა სიის დასახარისხებლად შეიძლება იყოს მოცემული შემდეგი ორი წესით:

- ცარიელი სია უკვე დახარისხებულია;
- ნებისმიერი არაცარიელი სია შეიძლება დახარისხდეს, თუ მის თავს მოვათავსებთ ორ სიას შორის. ისინი წარმოადგენს მოცემული არაცარიელი სიის კუდის იმ ელემენტების დახარისხების შედეგს, რომლებიც ან ნაკლებია ამ თავზე, ან მასზე მეტია შესაბამისად.

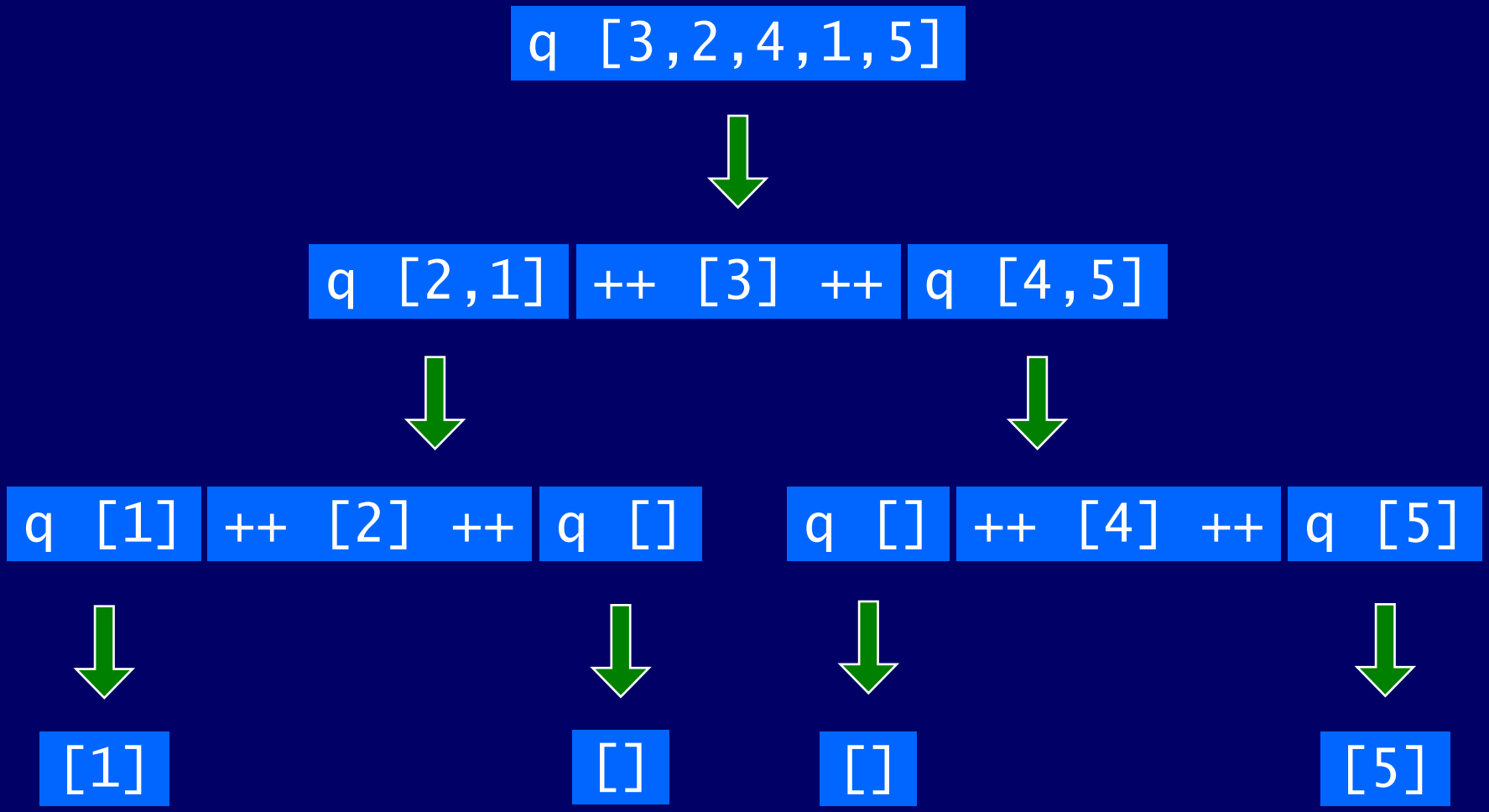
რეკურსიის გამოყენებით ეს წესები შეიძლება პირდაპირ ვაქციოთ კონკრეტულ რეალიზაციად:

```
qsort          :: [Int] → [Int]
qsort []        = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

შენიშვნა:

- ალბათ, ეს quicksort-ის უმარტივესი განხორციელებაა დაპროგრამების ნებისმიერ ენაზე!

მაგალითად (qsort წარმოდგენილია მხოლოდ q-თი):



სავარჯიშოები

(1) სტანდარტული prelude ფაილის გამოუყენებლად განსაზღვრეთ რეკურსიის საშუალებით შემდეგი საბიბლიოთეკო ფუნქციები:

- დაადგინეთ, თუ არის ჭეშმარიტი ყველა ლოგიკური მნიშვნელობა სიაში:

```
and :: [Bool] → Bool
```

- განახორციელეთ სიათა კონკატენაცია:

```
concat :: [[a]] → [a]
```

- შექმენით n ერთნაირი ელემენტის შემცველი სია:

```
replicate :: Int → a → [a]
```

- გამოყავით სიის n-ური ელემენტი:

```
(!!) :: [a] → Int → a
```

- დაადგინეთ, თუ არის მოცემული მნიშვნელობა სიის ელემენტი:

```
elem :: Eq a ⇒ a → [a] → Bool
```

(2) განსაზღვრეთ რეკურსიული ფუნქცია:

```
merge :: [Int] → [Int] → [Int]
```

იგი ახორციელებს ორი დახარისხებული სიის გაერთიანებას ერთი დახარისხებული სიის მისაღებად. მაგალითად:

```
> merge [2,5,6] [1,3,4]

[1,2,3,4,5,6]
```

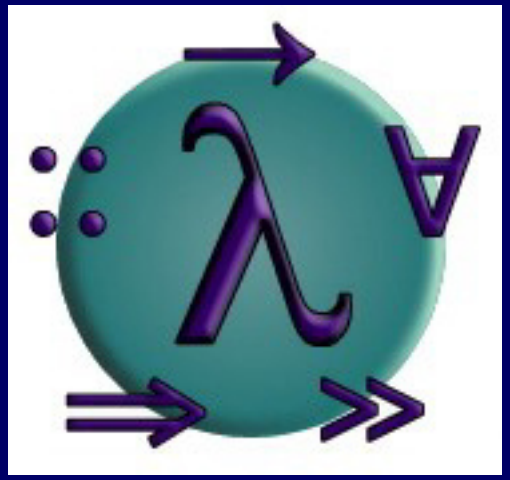
(3) განსაზღვრეთ რეკურსიული ფუნქცია:

```
msort :: [Int] → [Int]
```

იგი merge sort ფუნქციის რეალიზებას ახდენს, რომელიც შეიძლება იყოს მოცემული შემდეგი ორი წესის საფუძველზე:

- სიები, სადაც სიგრძე ≤ 1, უკვე დახარისხებულია;
- სხვა სიის დახარისხება ხდება ორი სიის გაერთიანებით, რომლებიც მიღებულია საწყისი სიის ორივე ნახევრის ცალ-ცალკე დახარისხების შედეგად.

დაპროგრამება HASKELL ენაზე



თავი 7 – მაღალი რიგის ფუნქციები

შესავალი

ფუნქციას, რომელიც იღებს ფუნქციას არგუმენტად ან გვიბრუნებს ფუნქციას შედეგის სახით, *მაღალი რიგის ფუნქცია* ეწოდება.

```
twice    :: (a → a) → a → a
twice f x = f (f x)
```

twice მაღალი რიგის ფუნაციაა, იმიტომ რომ იგი იღებს ფუნქციას როგორც თავის პირველ არგუმენტს.

რატომაა ისინი სასარგებლო?

- დაპროგრამების ჩვეულებრივი სტილი შეიძლება იყოს კოდირებული ფუნქციად თავად ენის ფარგლებში.
- ენათა სპეციფიკური სფეროები შეიძლება იქნეს განსაზღვრული მაღალი რიგის ფუნქციათა ნაკრებებით.
- მაღალი რიგის ფუნქციათა ალგებრული თვისებები შეიძლება გამოიყენებოდეს პროგრამათა დასაბუთებისათვის.

Map ფუნაცია

map სახელწოდების მაღალი რიგის საბიბლიოთეკო ფუნქცია ახორციელებს მითითებულ ფუნქციას სიის ყოველ ელემენტზე.

```
map :: (a → b) → [a] → [b]
```

მაგალითად:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

map ფუნქცია შეიძლება განისაზღვროს განსაკუთრებულად მარტივი ფორმით სიის კონსტრუქტორის გამოყენებისას:

```
map f xs = [f x | x ← xs]
```

გარდა ამისა, მტკიცებათა ჩატარების მიზნით, *map* ფუნქცია შეიძლება განისაზღვროს რეკურსიის საშუალებითაც:

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

Filter ფუნქცია

მაღალი რიგის საბიბლიოთეკო filter ფუნქცია ირჩევს სიიდან ყოველ ელემენტს, რომელიც პრედიკატს აკმაყოფილებს.

```
filter :: (a → Bool) → [a] → [a]
```

მაგალითად:

```
> filter even [1..10]

[2,4,6,8,10]
```

Filter ფუნქცია შეიძლება განისაზღვროს სიის კონსტრუქტორის საშუალებით:

```
filter p xs = [x | x ← xs, p x]
```

გარდა ამისა იგი შეიძლება განისაზღვროს რეკურსიითაც:

```
filter p []     = []
filter p (x:xs)
   | p x        = x : filter p xs
   | otherwise  = filter p xs
```

Foldr ფუნქცია

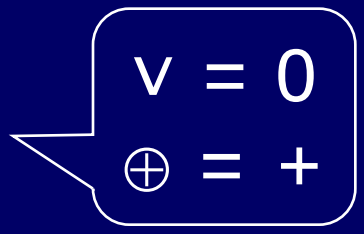
რიგი ფუნქციისა სიაზე შეიძლება იყოს განსაზღვრული რეკურსიის შემდეგი მარტივი მოდელით:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

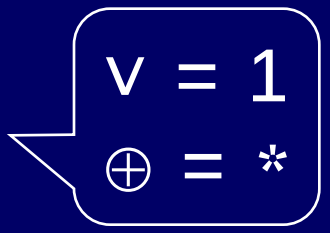
f ასახავს ცარიელ სიას რაღაც **v** მნიშვნელობად, ხოლო ნებისმიერ არაცარიელ სიას გარკვეულ ⊕ ოპერაციად, რომელიც ორ ოპერანდად შეიცავს სიის თავს და სიის კუდზე განხორციელებულ **f** ფუნქციას.

მაგალითად:

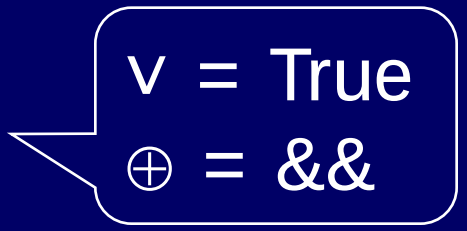
```
sum []     = 0
sum (x:xs) = x + sum xs
```




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




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



მაღალი რიგის foldr (fold right) საბიბლიოთეკო ფუნქცია რეკურსიის ამ მარტივი შაბლონის ინკაფსულირებას ახდენს ⊕ ფუნქციასთან და v მნიშვნელობასთან ერთად როგორც არგუმენტებთან.

ინკაფსულაცია (ლათ. *in capsula* - კოლოფში) – მონაცემებისა და მოქმედებების აბსტრაქცია და გადამალვა გარემოსაგან.

მაგალითად:

```
sum     = foldr (+) 0

product = foldr (*) 1

or      = foldr (||) False

and     = foldr (&&) True
```

Foldr თავად შეიძლება განისაზღვროს რეკურსიით:

```
foldr :: (a → b → b) → b → [a] → b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

მაგრამ უკეთესია განვიხილოთ foldr არარეკურსიულად, რისთვისაც ერთდროულად შევცვალოთ ყოველი (:) სიაში მოცემული ფუნქციით, ხოლო [] – მოცემული მნიშვნელობით.

მაგალითად:

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  1+(2+(3+0))
=
  6
```

ყოველი (:)-ის ჩანაცვლება (+)-ით და []-ის ჩანაცვლება 0-ით.

მაგალითად:

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
  1*(2*(3*1))
=
  6
```

ყოველი (:)-ის ჩანაცვლება (*)-ით და []-ის ჩანაცვლება 1-ით.

Foldr-ის გამოყენების სხვა მაგალითები

მიუხედავად იმისა, რომ foldr-ი რეკურსიის მარტივი შაბლონის ინკაფსულირებას ახდენს, შესაძლებელია მისი გამოყენება გაცილებით უფრო მეტი ფუნქციისათვის, ვიდრე ეს მოსალოდნელი იყო.

გავიხსენოთ სიგრძის length ფუნქცია:

```
length        :: [a] → Int
length []      = 0
length (_:xs) = 1 + length xs
```

მაგალითად:

```
  length [1,2,3]
=
  length (1:(2:(3:[])))
=
  1+(1+(1+0))
=
  3
```

ყოველი (:)-ის ჩანაცვლება (λ_ n → 1+n)-ით და []-ის ჩანაცვლება 0-ით.

ამრიგად, გვაქვს:

```
length = foldr (λ_ n → 1+n) 0
```

ახლა გავიხსენოთ **reverse** ფუნქცია:

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

მაგალითად:

```
  reverse [1,2,3]
=
  reverse (1:(2:(3:[])))
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

ყოველი (:)-ის ჩანაცვლება (λx xs → xs ++ [x])-ით და []-ის ჩანაცვლება []-ით.

ამრიგად, გვაქვს:

```
reverse =
    foldr (λx xs → xs ++ [x]) []
```

დაბოლოს, აღსანიშნავია, რომ დამატების (++) ფუნქციას აქვს განსაკუთრებულად კომპაქტური განსაზღვრება foldr-ის გამოყენებით:

```
(++ ys) = foldr (:) ys
```

ყოველი (:)-ის ჩანაცვლება (:)-ით და []-ის ჩანაცვლება ys-ით.

რატომაა სასარგებლო Foldr-ი?

- ზოგიერთი რეკურსიული ფუნქცია სიაზე, მაგალითად sum ფუნქცია, უფრო მარტივია განისაზღვროს foldr-ით.

- foldr-ით განსაზღვრულ ფუნქციათა თვისებები შეიძლება იყოს დამტკიცებული foldr-ის ალგებრული თვისებების გამოყენებით, როგორიცაა შერწყმა და «ნახევარბანანის»-ის (ნაყინიანი ბანანის) წესი.

- ოპტიმიზაციის გაუმჯობესებული პროგრამა შეიძლება უფრო მარტივი აღმოჩნდეს, თუ ცხადი რეკურსიის ნაცვლად გამოიყენება foldr ფუნქცია.

სხვა საბიბლიოთეკო ფუნქციები

საბიბლიოთეკო (.) ფუქნცია გვიბრუნებს ორი ფუნქციის კომპოზიციას როგორც ერთ ფუნქციას:

```
(.)    :: (b → c) → (a → b) → (a → c)
f . g  = λx → f (g x)
```

მაგალითად:

```
odd :: Int → Bool
odd  = not . even
```

საბიბლიოთეკო **all** ფუნქცია არკვევს, თუ აკმაყოფილებს მოცემულ პრედიკატს სიის ყოველი ელემენტი.

```
all      :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

მაგალითად:

```
> all even [2,4,6,8,10]

True
```

ამის დუალურად, საბიბლიოთეკო any ფუნქცია არკვევს, თუ აკმაყოფილებს პრედიკატს სიის ერთი ელემენტი მაინც.

```
any      :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

მაგალითად:

```
> any isSpace "abc def"

True
```

საბიბლიოთეკო takeWhile ფუნქცია გამოყოფს ელემენტებს სიიდან, ვიდრე პრედიკატი სამართლიანი რჩება.

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p []      = []
takeWhile p (x:xs)
   | p x            = x : takeWhile p xs
   | otherwise      = []
```

მაგალითად:

```
> takeWhile isAlpha "abc def"

"abc"
```

ამის დუალურად, dropWhile ფუნქცია ანადგურებს ელემენტებს სიაში, ვიდრე პრედიკატი სამართლიანი რჩება.

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p []      = []
dropWhile p (x:xs)
    | p x           = dropWhile p xs
    | otherwise     = x:xs
```

მაგალითად:

```
> dropWhile isSpace "   abc"

"abc"
```

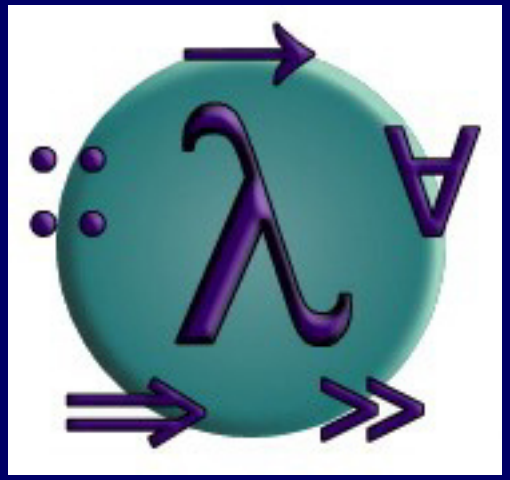
სავარჯიშოები

(1) რა სახელით უფრო არის ცნობილი მაღალი რიგის ფუნქცია, რომელიც შედეგის ფორმით კვლავ ფუნქციას გვიბრუნებს?

(2) გამოსახეთ **[f x | x ← xs, p x]** კონსტრუქტორი, რომელიც იყენებს **map** და **filter** ფუნქციებს.

(3) **foldr** ფუნქციის გამოყენებით ხელახლა განსაზღვრეთ **map f** და **filter p** ფუნქციები.

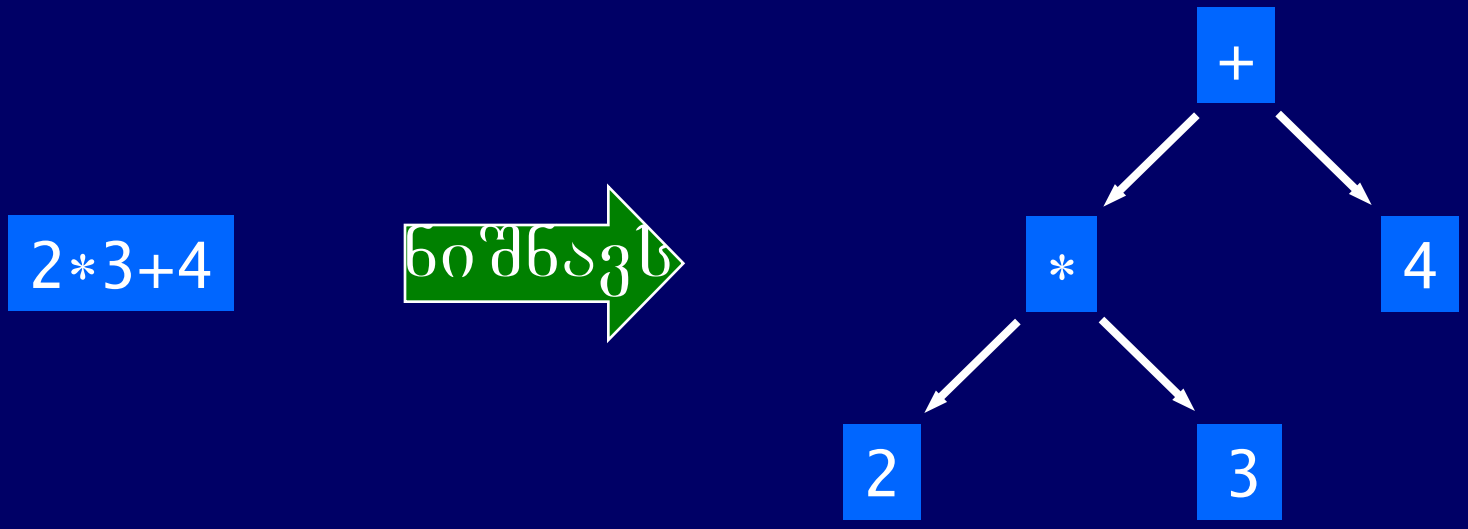
დაპროგრამება HASKELL ენაზე



თავი 8 – ფუნქციონალური პარსერები
(სინტაქსური ანალიზატორები)

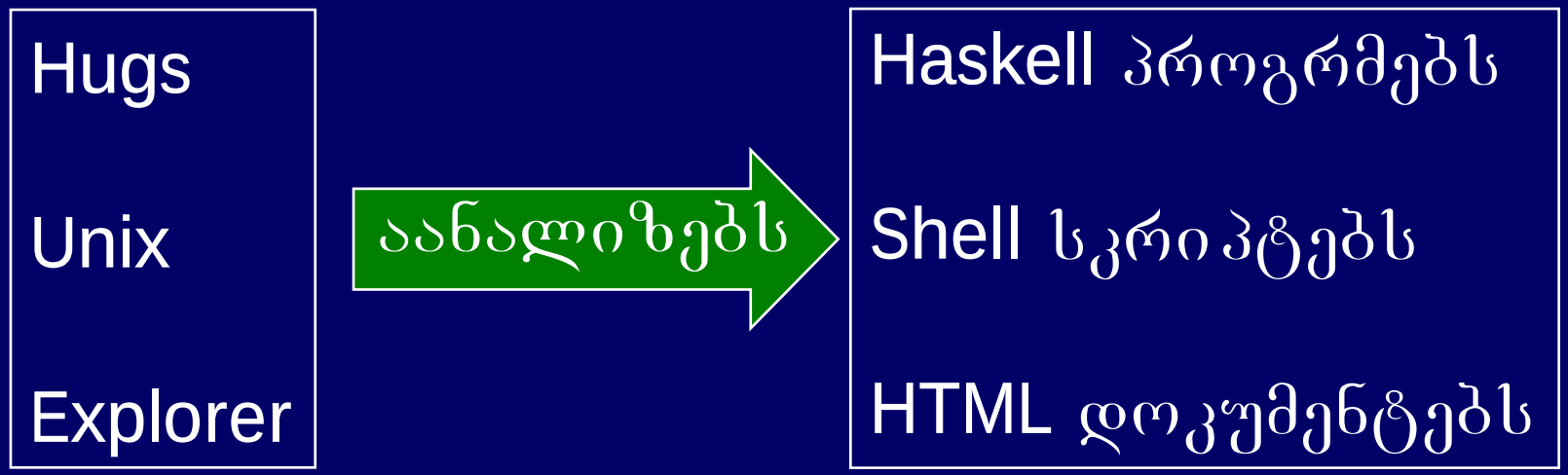
რა არის პარსერი?

პარსერი არის პროგრამა რომელიც აანალი-ზებს ტექსტის ფრაგმენტს მისი სინტაქსური სტრუქტურის დასადგენად.



სად გამოიყენება იგი?

თითქმის ყოველი რეალურად არსებული პროგრამა იყენებს ამა თუ იმ ფორმით პარსერს თავისი შეტანის წინასწარი ანალიზისათვის.



Parser ტიპი

ნებისმიერ ფუნქციონალურ ენაში, როგორიცაა ჰასკელი, პარსერები შეიძლება ბუნებრივად განიხილებოდეს ფუნქციებად.

```
type Parser = String → Tree
```

პარსერი არის ფუნქცია, რომელსაც შეუძლია სტრიქონის მიღება და ამა თუ იმ ფორმით ხის დაბრუნება.

მაგრამ პარსერი შეიძლება არ მოითხოვდეს ყველა თავის შემავალ სრიქონს, ამიტომ ჩვენ ასევე ვაბრუნებთ გამოუყენებელ შემავალ ინფორმაციასაც:

```
type Parser = String → (Tree,String)
```

სტრიქონი შეიძლება ითხოვდეს სინტაქსურად გარჩევას სხვადასხვა გზით და არა მხოლოდ ერთი გზით. ასეთ შემთხვევაში ჩვენ ვანზოგადებთ შედეგების სიის დასაბრუნებლად პარსერის ტიპს:

```
type Parser = String → [(Tree,String)]
```

დაბოლოს, პარსერს შეუძლია არც აწარმოოს ყოველთვის რაიმე ხე და გარკვეული მნიშვნელობის დაბრუნებით შემოიფარგლოს. ამიტომ *Tree* ტიპის განზოგადება მოხდება პარსერ ტიპის პარამეტრში:

```
type Parser a = String → [(a,String)]
```

შენიშვნა:

- სიმარტივისათვის განვიხილავთ მხოლოდ ისეთ პარსერს, რომელიც ან განიცდის მტყუნებას და გვიბრუნებს შედეგების ცარიელ სიას, ან აღწევს წარმატებას და გვიბრუნებს ერთელემენტიან სიას (სინგლეტონს).

ძირითადი პარსერები

- *item* პარსერი, რომელიც წარუმატებლად მთავრდება, თუ შემავალი სტრიქონი ცარიელია, და წარმატებით – პირველი სიმბოლოს სახით საშედეგო მნიშვნელობის როლში – წინააღმდეგ შემთხვევაში:

```
item :: Parser Char
item  = λinp → case inp of
                 []     → []
                 (x:xs) → [(x,xs)]
```

- *failure* პარსერი მუდამ წარუმატებლობას განიცდის შემავალი სტრიქონის შინაარსისაგან დამოუკიდებლად:

```
failure :: Parser a
failure  = λinp → []
```

- *return v* პარსერი ყოველთვის წარმატებით გვიბრუნებს შედეგის *v* მნიშვნელობას მთლიანი შემავალი სტრიქონის დაუმუშავებლად:

```
return  :: a → Parser a
return v = λinp → [(v,inp)]
```

- `p +++ q` პარსერი იქცევა როგორც `p` პარსერი, თუ ეს ხერხდება, და როგორც `q` პარსერი წინააღმდეგ შემთხვევაში:

```
(+++)  :: Parser a → Parser a → Parser a
p +++ q = λinp → case p inp of
                   []       → parse q inp
                   [(v,out)] → [(v,out)]
```

- *parse* ფუნქცია გამოიყენება სტრიქონის გასაანალიზებლად:

```
parse :: Parser a → String → [(a,String)]
parse p inp = p inp
```

მაგალითები

სინტაქსური ანალიზის (გარჩევის) ხუთი პრიმიტივის ქცევა შეიძლება ილუსტრირებული იქნეს რამდენიმე მარტივ მაგალითზე:

```
% hugs Parsing

> parse item ""
[]

> parse item "abc"
[('a',"bc")]
```

```
> parse failure "abc"
[]

> parse (return 1) "abc"
[(1,"abc")]

> parse (item +++ return 'd') "abc"
[('a',"bc")]

> parse (failure +++ return 'd') "abc"
[('d',"abc")]
```

შენიშვნა:

- Parsing საბიბლიოთეკო ფაილი მისაწვდომია ინტერნეტში ჰასკელზე დაპროგრამების საკუთარ გვერდზე.
- თექნიკური მიზეზების გამო წარუმატებლობის გამომწვლენი პირველი მაგალითი ფაქტობრივად იძლევა შეცდომას, რომელიც ეხება ტიპებს, მაგრამ ეს არ ხდება არატრივიალურ მაგალითებში.
- Parser ტიპი არის მონადა – მათემატიკური სტრუქტურა, რომელსაც დანამდვილებით მოაქვს სარგებლობა სხვადასხვა სახის გამოთვლათა მოდელირებისას.

მოწესრიგება

პარსერების მიმდევრობა შეიძლება გაერთიანდეს ერთ შედგენილ პარსერად **do** საკვანძო სიტყვის გამოყენებით.

მაგალითად:

```
p :: Parser (Char,Char)
p  = do x ← item
        item
        y ← item
        return (x,y)
```

შენიშვნა:

- ყოველი პარსერი უნდა იწყებოდეს ერთსა და იმავე სვეტში. ეს არის ტოპოლოგიური წესის მოთხოვნა.
- შუალედური პარსერების მიერ დაბრუნებული მნიშვნელობები გადაგდებული აღმოჩნდება გაუცხადებლად, მაგრამ საჭიროების შემთხვევაში მათ მიენიჭება სახელები ← ოპერატორის გამოყენებით.
- უკანასკნელი პარსერის მიერ მოცემული მნიშვნელობა არის ის მნიშვნელობა, რომელიც დაბრუნებულია პარსერთა მიმდევრობით როგორც ერთი მთლიანობა.

- თუ რომელიმე პარსერი ამ ობიექტების მიმდევრობაში წარუმატებელია, მაშინ მთლიანად მიმდევრობაც მთყუნებით მთავრდება. მაგალითად:

```
> parse p "abcdef"
[(('a','c'),"def")]

> parse p "ab"
[]
```

- **do** ნოტაცია არ არის სპეციფიკური **Parser** ტიპისათვის, იგი შეიძლება გამოყენებული იქნეს ნებისმიერ მონადურ ტიპთან.

ნაწარმოები პრიმიტივები

- სიმბოლოს სინტაქსური ანალიზი (გარჩევა, პარსინგი) პრედიკატის დაკმაყოფილების დასადგენად:

```
sat  :: (Char → Bool) → Parser Char
sat p = do x ← item
           if p x then
              return x
            else
              failure
```

- ციფრებისა და სპეციფიკური სიმბოლოების სინტაქსური ანალიზი (გარჩევა, პარსინგი):

```
digit :: Parser Char
digit  = sat isDigit

char  :: Char → Parser Char
char x = sat (x ==)
```

- პარსერის გამოყენება ნულჯერ ან მეტჯერ:

```
many  :: Parser a → Parser [a]
many p = many1 p +++ return []
```

- პარსერის გამოყენება თუნდაც ერთჯერ ან მეტჯერ:

```
many1  :: Parser a -> Parser [a]
many1 p = do v  ← p
             vs ← many p
             return (v:vs)
```

- სიმბოლოთა სპეციფიკური სტრიქონის სინტაქსური ანალიზი:

```
string        :: String → Parser String
string []      = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

მაგალითი

ახლა შეგვიძლია განვსაზღვროთ პარსერი, რომელიც იყენებს ერთი ან მეტი ციფრის სიას სტრიქონიდან:

```
p :: Parser String
p  = do char '['
        d  ← digit
        ds ← many (do char ','
                      digit)
        char ']'
        return (d:ds)
```

მაგალითად:

```
> parse p "[1,2,3,4]"
[("1234","")]

> parse p "[1,2,3,4"
[]
```

შენიშვნა:

- სინტაქსური ანალიზის უფრო რთულ ბიბლიოთეკებს შეუძლია შემავალი სტრიქონის შეცდომათა მიუთითება და/ან აცილება.

არითმეტიკული გამოსახულებები

განვიხილოთ გამოსახულებათა მარტივი ფორმა, რომელიც აგებულია (+) შეკრებისა და (*) გამრავლების ოპერაციათა გამოყენებით მრგვალ ფრჩხილებთან ერთად.

ასევე გასაგებია, რომ:

- * და + ასოციატიურია მარჯვნივ;
- * უფრო მაღალი პრიორიტეტისაა, ვიდრე +.

ფორმალურად, ასეთი გამოსახულებების სინტაქსი განსაზღვრულია თავისუფალი გრამატიკის შემდეგი კონტექსტით:

$$
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{term}\ \texttt{'+'}\ \mathit{expr} \mid \mathit{term} \\
\mathit{term} &\rightarrow \mathit{factor}\ \texttt{'*'}\ \mathit{term} \mid \mathit{factor} \\
\mathit{factor} &\rightarrow \mathit{digit} \mid \texttt{'('}\ \mathit{expr}\ \texttt{')'} \\
\mathit{digit} &\rightarrow \texttt{'0'} \mid \texttt{'1'} \mid \ldots \mid \texttt{'9'}
\end{aligned}
$$

მაგრამ, ეფექტურობის მოსაზრებებიდან გამომდინარე, მნიშვნელოვანია *expr*-ისა და *term*-ისათვის წესების გამარტივება (ესე იგი მათი დაშლა, ანუ ფაქტორიზაცია ელემენტარულ მდგენელებად):

$$expr \rightarrow term\ (\text{'+'}\ expr \mid \varepsilon)$$

$$term \rightarrow factor\ (\text{'*'}\ term \mid \varepsilon)$$

შენიშვნა:

- ε სიმბოლო აღნიშნავს ცარიელ სრიქონს.

ახლა ადვილია გრამატიკის წარმოდგენა პარსერად, რომელიც აფასებს გამოსახულებებს გრამატიკული წესების უბრალო გადაწერით სინტაქსური ანალიზის პრიმიტივების გამოყენებით.

მაშასადამე, გვაქვს:

```
expr :: Parser Int
expr  = do t ← term
           do char '+'
              e ← expr
              return (t + e)
            +++ return t
```

```
term :: Parser Int
term  = do f ← factor
           do char '*'
              t ← term
              return (f * t)
            +++ return f
```

```
factor :: Parser Int
factor  = do d ← digit
             return (digitToInt d)
           +++ do char '('
                  e ← expr
                  char ')'
                  return e
```

საბოლოოდ, თუ განვსაზღვრავთ, რომ

```
eval   :: String → Int
eval xs = fst (head (parse expr xs))
```

მაშინ პრაქტიკულად შევამოწმებთ ზოგიერთ მაგალითსაც:

```
> eval "2*3+4"
10

> eval "2*(3+4)"
14
```

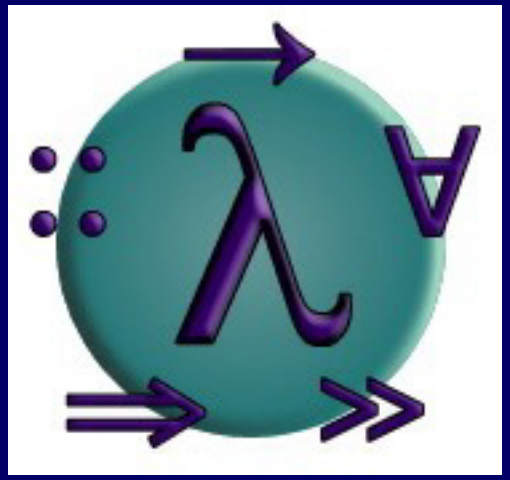
სავარჯიშოები

(1) რატომ ახდენს არითმეტიკული გამოსახულებებისათვის განკუთვნილი გრამატიკის საბოლოო გამარტივება არსებით გავლენას ამის შედეგად მიღებული პარსერის ეფექტურობაზე.

(2) გააფართოეთ პარსერი არითმეტიკული გამოსახულებებისათვის გამოკლებისა და გაყოფის მხარდასაჭერად, რისთვისაც გამოიყენეთ გრამატიკის შემდეგი ჩაწერის ფორმები:

```
expr → term ('+' expr | '-' expr | ε)

term → factor ('*' term | '/' term | ε)
```

დაპროგრამება HASKELL ენაზე

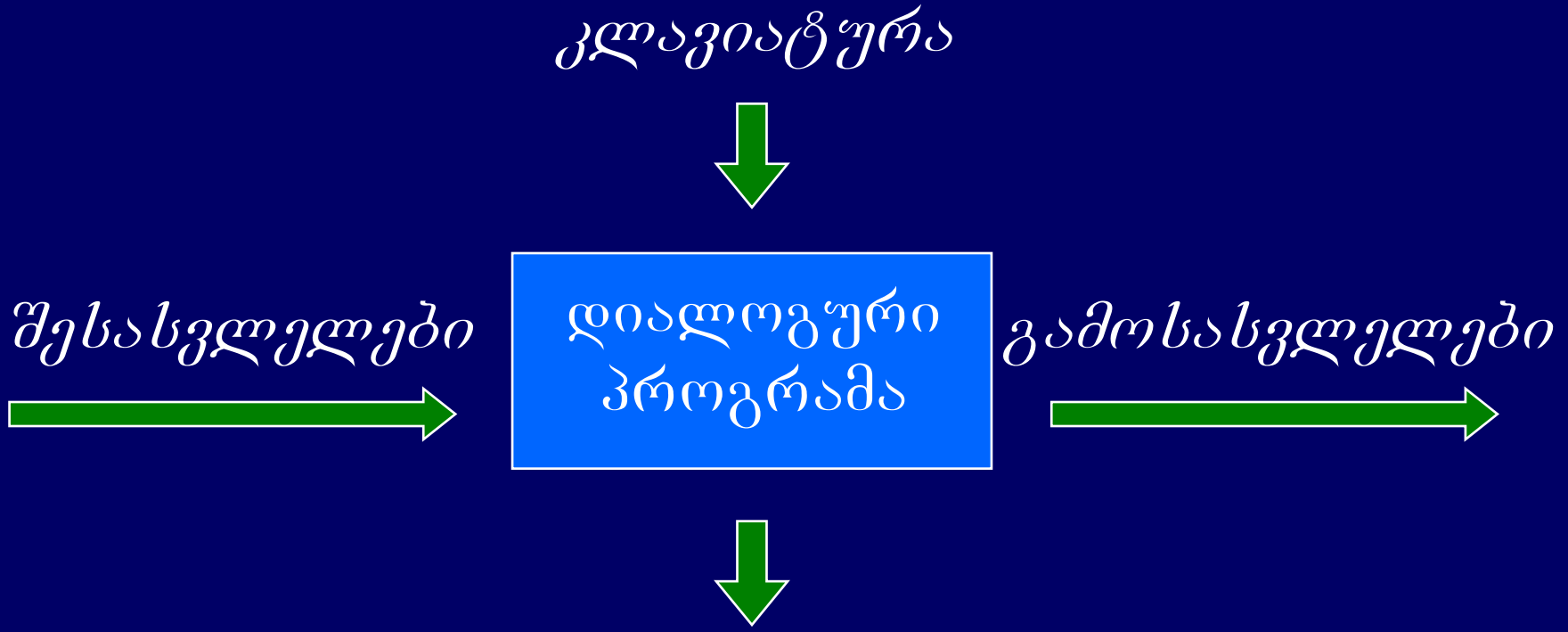


თავი 9 – ინტერაქტიური (დიალოგური) პროგრამები

შესავალი

აქამდე ვსწავლობდით, როგორ შეიძლება ჰასკელის გამოყენება პაკეტური პროგრამის ჩასაწერად, რომელიც იღებს ყველა თავის შემავალ მონაცემებს სტარტზე და იძლევა ყველა თავის გამომავალ მონაცემებს მუშაობის დასრულებისას.

მაგრამ ჩვენ გვჭირდება ასევე ჰასკელის გამოყენება *ინტერაქტიური* (*დიალოგური*) პროგრამის შესაქმნელად, რომელიც კითხულობს კლავიატურიდან და წერს ეკრანზე ინფორმაციას თავისი მუშაობის პროცესში.



პრობლემის არსი

ჰასკელის პროგრამები *სუფთა* მათემატიკური ფუნქციებია:

- ჰასკელის პროგრამებს *არ აქვს თანამდევი ეფექტები*.

მაგრამ კლავიატურიდან წაკითხვა და ეკრანზე ჩაწერა თანამდევი ეფექტებია:

- ინტერაქტიურ პროგრამებს *აქვს თანამდევი ეფექტები*.

პრობლემის გადაჭრა

ინტერაქტიური პროგრამები შეიძლება დაიწეროს ჰასკელზე ისეთი ტიპების გამოყენებით, რომლებიც გაარჩევს სუფთა გამოსახულებებს თანამდევი ეფექტების შემცველი არასუფთა *მოქმედებებისაგან* (***actions***).

```
IO a
```

ტიპი: მოქმედების, რომელიც გვიბრუნებს **a** ტიპის რაღაც მნიშვნელობას.

მაგალითად:

```
IO Char
```

ტიპი: მოქმედების, რომელიც გვიბრუნებს სიმბოლოს.

```
IO ()
```

ტიპი: მოქედების, რომელიც არ გვიბრუნებს შედეგის მნიშვნელობას.

შენიშვნა:

- () – უკომპონენტო კორტეჟთა ტიპი.

ძირითადი მოქმედებები

სტანდარტული ბიბლიოთეკა ითვალისწინებს მოქმედებათა გარკვეულ რაოდენობას პრიმიტივების შემდეგი ხის ჩათვლით:

- **getChar** მოქმედება კითხულობს სიმბოლოს კლავიატურიდან, ასახავს მას ეკრანზე და გვიბრუნებს სიმბოლოს როგორც შედეგის მნიშვნელობას:

```
getChar :: IO Char
```

- **putChar c** მოქმედება წერს ეკრანზე **c** სიმბოლოს და არ გვიბრუნებს შედეგის მნიშვნელობას:

```
putChar :: Char → IO ()
```

- **return v** მოქმედება უბრალოდ გვიბრუნებს **v** მნიშვნელობას რაიმე ურთიერთქმედების განხორციელების გარეშე:

```
return :: a → IO a
```

მოწესრიგება

მოქმედებათა მიმდევრობა შეიძლება გაერთიანდეს და იქცეს ერთ შედგენილ მოქმედებად **do** საკვანძო სიტყვის გამოყენებით.

მაგალითად:

```
a :: IO (Char,Char)
a  = do x ← getChar
        getChar
        y ← getChar
        return (x,y)
```

ნაწარმოები პრიმიტივები

- სტრიქონის წაკითხვა კლავიატურიდან:

```
getLine :: IO String
getLine  = do x ← getChar
              if x == '\n' then
                 return []
               else
                 do xs ← getLine
                    return (x:xs)
```

- სტრიქონის ჩაწერა ეკრანზე:

```
putStr        :: String → IO ()
putStr []      = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

- სტრიქონის ჩაწერა და ახალზე გადასვლა:

```
putStrLn    :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

მაგალითი

ახლა შესაძლებელია ისეთი მოქმედების განსაზღვრა, რომელიც ითხოვს სტრიქონის შეყვანას და ასახავს მის სიგრძეს:

```
strlen :: IO ()
strlen  = do putStr "Enter a string: "
             xs ← getLine
             putStr "The string has "
             putStr (show (length xs))
             putStrLn " characters"
```

მაგალითად:

```
> strlen

Enter a string: abcde
The string has 5 characters
```

შენიშვნა:

- რაიმე მოქმედების შეფასება ახორციელებს თავის თანამდევ ეფექტებს საბოლოო შედეგის მნიშვნელობის გადაგდებასთან ერთად.

ჯალათი

განვიხილოთ თამაშ «ჯალათის» («**hangman**») შემდეგი ვერსია:

- ერთი მოთამაშე ფარულად ბეჭდავს სიტყვას.
- მეორე მოთამაშე ცდილობს ამ სიტყვის გამოცნობას სავარაუდო სიტყვათა მიმდევრობის შეყვანით.
- ყოველი ვარაუდისათვის კომპიუტერი უთითებს იმ ასოებს საიდუმლო სიტყვაში, რომლებიც გვხვდება სავარაუდოში.

- თამაში მთავრდება, როცა სავარაუდო სიტყვა სწორია.

ჩვენ ვირჩევთ დაღმავალ მიდგომას თამაშ «ჯალათის» სარეალიზაციოდ ჰასკელზე და ვიწყებთ ასეთი ფრაგმენტით (აქ ინგლისურად ნახმარია ორი ფრაზა – ჩაიფიქრეთ სიტყვა და შეეცადეთ გამოიცნოთ იგი) :

```
hangman :: IO ()
hangman  =
   do putStrLn "Think of a word: "
      word ← sgetLine
      putStrLn "Try to guess it:"
      guess word
```

sgetLine მოქმედება კითხულობს ტექსტის სტრიქონს კლავიატურიდან და ასახავს ეკრან-ზე ყოველ სიმბოლოს როგორც ტირეს:

```
sgetLine :: IO String
sgetLine  = do x ← getCh
               if x == '\n' then
                  do putChar x
                     return []
                else
                  do putChar '-'
                     xs ← sgetLine
                     return (x:xs)
```

შენიშვნა:

- **getCh** მოქმედება კითხულობს სიმბოლოს კლავიატურიდან, მაგრამ არ ასახავს მას ეკრანზე.

- ეს სასარგებლო მოქმედება არ არის სტანდარტული ბიბლიოთეკის ნაწილი, მაგრამ არის **Hugs** სისტემის სპეციალური პრიმიტივი, რომელიც შეიძლება იქნეს იმპორტირებული სკრიპტში შემდეგნაირად:

```
primitive getCh :: IO Char
```

guess ფუნქცია – ძირითადი ციკლია, რომელიც ითხოვს და ამუშავებს სავარაუდო სიტყვებს თამაშის დასრულებამდე. აქ ნახმარია ინგლისური ფრაზა: «თქვენ იპოვეთ იგი!»

```
guess      :: String → IO ()
guess word =
   do putStr "> "
      xs ← getLine
      if xs == word then
         putStrLn "You got it!"
       else
         do putStrLn (diff word xs)
            guess word
```

diff ფუნქცია უთითებს იმ ასოებს ერთ სტრიქონში, რომლებიც გვხვდება მეორე სტრიქონშიც.

```
diff      :: String → String → String
diff xs ys =
   [if elem x ys then x else '-' | x ← xs]
```

მაგალითად:

```
> diff "haskell" "pascal"

"-as--ll"
```

სავარჯიშო

nim თამაშის განხორციელება ჰასკელზე, როცა ამ თამაშის წესები ასეთია:

- დაფა შეიცავს ფიფქების ხუთ სტრიქონს:

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

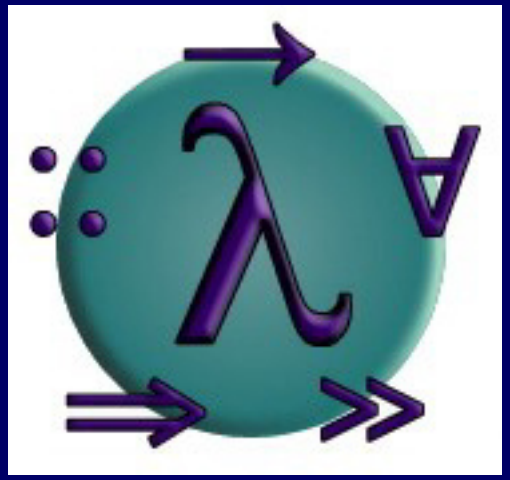
- ორი მოთამაშე რიგრიგობით ეხება და ანადგურებს ერთ ან რამდენიმე ფიფქს ერთადერთი სტრიქონის ბოლოდან.

- მოგებული რჩება ის მოთამაშე, რომელიც დაფიდან აიღებს უკანასკნელ ფიფქს ან ფიფქებს.

კარნახი :

წარმოადგინეთ დაფა სიად ხუთი მთელი რიცხვით, რომლებიც დაფაზე დარჩენილ ფიფქთა რაოდენობას იძლევა თითოეულ სტრიქონზე. მაგალითად, თამაშის დასაწყისში დაფა აისახება [5,4,3,2,1] სიით.

დაპროგრამება HASKELL ენაზე



თავი 10 – ტიპების გამოცხადება და კლასები

ტიპის გამოცხადებები

ჰასკელში ახალი სახელი არსებული ტიპისათვის შეიძლება იყოს განსაზღვრული *ტიპის გამოცხადების* (*დეკლარაციის*) გამოყენებით.

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

String – სინონიმია [Char] ტიპისათვის.

ტიპის გამოცხადებები შეიძლება გამოიყენებოდეს სხვა ტიპების წაკითხვის გასაიოლებლად. მაგალითად,

```
type Pos = (Int,Int)
```

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

```
origin     :: Pos
origin      = (0,0)

left       :: Pos → Pos
left (x,y) = (x-1,y)
```

ფუნქციის განსაზღვრებათა მსგავსად, ტიპის გამოცხადებებს ასევე შეიძლება გააჩნდეს *პარამეტრები*. მაგალითად,

```
type Pair a = (a,a)
```

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

```
mult       :: Pair Int → Int
mult (m,n) = m*n

copy       :: a → Pair a
copy x      = (x,x)
```

ტიპის გამოცხადებები შეიძლება ჩალაგებულიც იყოს ერთმანეთში:

```
type Pos   = (Int,Int)

type Trans = Pos → Pos
```

მაგრამ ისინი არ შეიძლება იყოს რეკურსიული:

```
type Tree = (Int,[Tree])
```

მონაცემთა გამოცხადებები

სრულიად ახალი ტიპი შეიძლება განისაზღვროს მისი მნიშვნელობების მითითებით *მონაცემთა გამოცხადების* დროს.

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

Bool არის ახალი ტიპი, ორი ახალი მნიშვნელობით False და True.

შენიშვნა:

- False და True მნიშვნელობებს ეწოდება *კონსტრუქტორები* მოცემული Bool ტიპისათვის.
- ტიპისა და კონსტრუქტორის სახელები უნდა იწყებოდეს ზედა რეგისტრის ასოთი.
- მონაცემთა გამოცხადებები თავისუფალი გრამატიკების კონტექსტის მსგავსია. პირველი აღწერს მნიშვნელობათა ტიპს, უკანასკნელი კი - ენის წინადადებებს.

ახალი ტიპების მნიშვნელობათა გამოყენება ჩაშენებული ტიპების მნიშვნელობათა მსგავსად შეიძლება. მაგალითად,

```
data Answer = Yes | No | Unknown
```

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

```
answers     :: [Answer]
answers      = [Yes,No,Unknown]

flip        :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

მონაცემთა გამოცხადებაში კონსტრუქტორებს ასევე შეიძლება გააჩნდეს პარამეტრები. მაგალითად,

```
data Shape = Circle Float
           | Rect Float Float
```

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

```
square          :: Float → Shape
square n         = Rect n n

area            :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

შენიშვნა:

- Shape-ს აქვს ფორმის მნიშვნელობები Circle r, სადაც r – მცურავწერტილიანი რიცხვია, და Rect x y, სადაც x და y – მცურავწერტილიანი რიცხვებია.

- Circle და Rect შეიძლება იყოს განხილული როგორც **ფუნქციები** Shape ტიპის მნიშვნელობათა ასაგებად:

```
Circle :: Float → Shape

Rect   :: Float → Float → Shape
```

განსაკვირი არ არის, რომ მონაცემთა გამოცხადებებს თავად შეიძლება გააჩნდეს ასევე პარამეტრები. მაგალითად,

```
data Maybe a = Nothing | Just a
```

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

```
safediv    :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

რეკურსიული ტიპები

ჰასკელში ახალი ტიპები შეიძლება გამოცხადდეს საკუთარი თავის მეშვეობით. სხვანაირად რომ ვთქვათ ტიპები შეიძლება **რეკურსიული** იყოს.

```
data Nat = Zero | Succ Nat
```

Nat ახალი ტიპია კონსტრუქტორებით
Zero :: Nat და Succ :: Nat → Nat.

შენიშვნები:

- Nat ტიპის მნიშვნელობა ან Zero, ან Succ n ფორმის არის, სადაც n :: Nat. სხვანაირად, Nat შეიცავს მნიშვნელობათა შემდეგ უსასრულო მიმდევრობას:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```

- შეგვიძლია მივიჩნიოთ, რომ Nat ტიპის მნიშვნელობები *ნატურალური რიცხვებია*, სადაც Zero წარმოადგენს 0-ს, ხოლო Succ იძლევა 1+ ფუნქციის მომდევნო ელემენტს.

- მაგალითად,

```
Succ (Succ (Succ Zero))
```

მნიშვნელობა წარმოადგენს ნატურალურ რიცხვს

```
1 + (1 + (1 + 0))  =  3
```

რეკურსიის გამოყენებით ადვილად განისაზღვრება ფუნქციები, რომლებიც ახორციელებს Nat და Int ტიპის მნიშვნელობათა შორის გარდასახვას:

```
nat2int           :: Nat → Int
nat2int Zero       = 0
nat2int (Succ n)   = 1 + nat2int n

int2nat           :: Int → Nat
int2nat 0          = Zero
int2nat (n+1)      = Succ (int2nat n)
```

ორი ნატურალური რიცხვის ჯამი შეიძლება ვიპოვოთ მათი გარდაქმნით მთელ რიცხვებად, შეკრებით და მერე შედეგის ხელახლა გარდასახვით ნატურალურ რიცხვად:

```
add     :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

მაგრამ რეკურსიის საშუალებით add ფუნქცია შეიძლება განისაზღვროს გარდაქმნის გამოუყენებლადაც:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

მაგალითად:

```
  add (Succ (Succ Zero)) (Succ Zero)
=
  Succ (add (Succ Zero) (Succ Zero))
=
  Succ (Succ (add Zero (Succ Zero))
=
  Succ (Succ (Succ Zero))
```

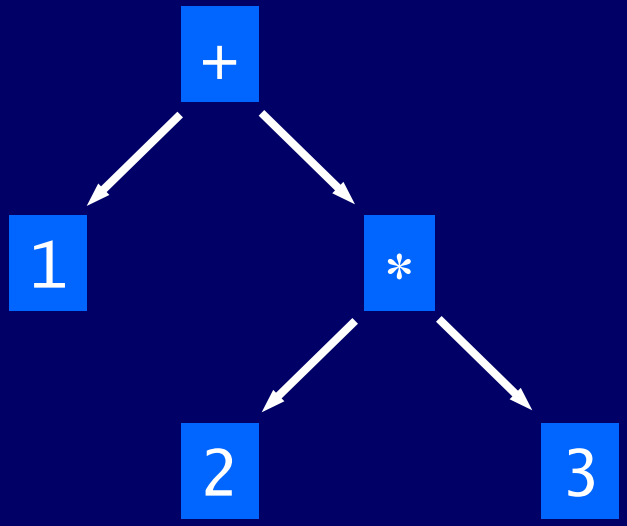
შენიშვნა

- რეკურსიული განსაზღვრება add-სათვის შეესაბამება შემდეგ წესებს:

  0+n = n და **(1+m)+n = 1+(m+n)**.

არითმეტიკული გამოსახულებები

განვიხილოთ რიცხვთა შეკრებითა და გამრავლებით აგებულ გამოსახულებათა მარტივი ფორმა.



რეკურსიის საშუალებით შესაფერი ახალი ტიპი მსგავსი გამოსახულებების წარმოსადგენად შეიძლება ასეთი გზით გამოცხადდეს:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

მაგალითად, გამოსახულება წინა სლაიდზე შემდეგნაირად იქნებოდა წარმოდგენილი:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

რეკურსიის გამოყენებით, ახლა ადვილია ფუნქციების განსაზღვრა, რომლებიც გამოსახულებათა დამუშავებას ახორციელებს. მაგალითად:

```
size            :: Expr → Int
size (Val n)     = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval            :: Expr → Int
eval (Val n)     = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

შენიშვნა:

- სამი კონსტრუქტორის ტიპია:

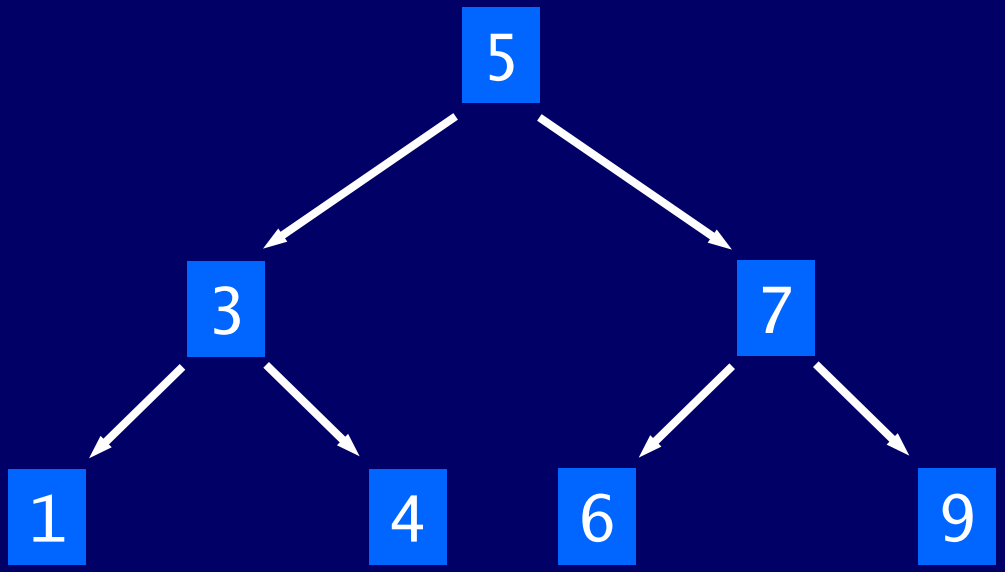
```
Val :: Int → Expr
Add :: Expr → Expr → Expr
Mul :: Expr → Expr → Expr
```

- მრავალი ფუნქცია გამოსახულებებზე შეიძლება განისაზღვროს კონსტრუქტორების ჩანაცვლებისას სხვა ფუნქციებით შესაფერი **fold** ფუნქციის ხარჯზე. მაგალითად:

```
eval = fold id (+) (*)
```

ბინარული ხეები

გამოთვლების დროს ხშირად სასარგებლოა მონაცემების შენახვა ორი მიმართულებით განშტოებულ სტრუქტურაში, რომელსაც *ბინარული ხე* ეწოდება.



რეკურსიის საშუალებით, სათანადო ახალი ტიპი მსგავს ბინარულ ხეთა წარმოსადგენად შეიძლება შემდეგნაირად გამოცხადდეს:

```
data Tree = Leaf Int
          | Node Tree Int Tree
```

მაგალითად, წინა სლაიდზე მოცემული ხე ასეთნაირად აღმოჩნდება ჩაწერილი:

```
Node (Node (Leaf 1) 3 (Leaf 4))
     5
     (Node (Leaf 6) 7 (Leaf 9))
```

ახლა შეგვიძლია განვსაზღვროთ ფუნქცია, რომელიც დაადგენს, თუ გვხვდება მოცემული მთელი რიცხვი ბინარულ ხეზე:

```
occurs                    :: Int → Tree → Bool
occurs m (Leaf n)          = m==n
occurs m (Node l n r)      = m==n
                             || occurs m l
                             || occurs m r
```

მაგრამ... უარეს შემთხვევაში, როცა მთელი რიცხვი არ გვხვდება, ამ ფუნციას მთელი ხის გავლა მოუწევს.

ახლა განვიხილოთ **flatten** ფუნქცია, რომელიც გვიბრუნებს ხეზე განთავსებული ყველა მთელი რიცხვის სიას :

```
flatten                  :: Tree → [Int]
flatten (Leaf n)          = [n]
flatten (Node l n r)      = flatten l
                            ++ [n]
                            ++ flatten r
```

ხეს ეწოდება *ძებნის ხე*, თუ იგი დაიყვანება მოწესრიგებულ სიამდე. ჩვენი მაგალითის ხე წარმოადგენს ძებნის ხეს, რადგან იგი დაიყვანება მოწესრიგებულ **[1,3,4,5,6,7,9]** სიამდე.

ძებნის ხეს აქვს არსებითი თვისება – ხეში მნიშვნელობის პოვნის მცდელობისას ყოველთვის შეგვიძლია ორ ქვეხეს შორის შევარჩიოთ ის, რომელშიც იგი შეიძლება შეგვხვდეს:

```
occurs m (Leaf n)                = m==n
occurs m (Node l n r) | m==n = True
                      | m<n  = occurs m l
                      | m>n  = occurs m r
```

ეს ახალი განსაზღვრება უფრო ეფექტურია, რადგან ხეზე – ქვევით მიმავალი გზის – გავლა მხოლოდ ერთხელ არის საჭირო.

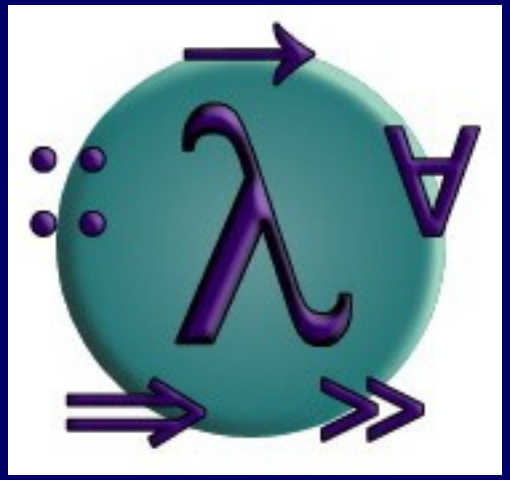
სავარჯიშოები

(1) რეკურსიისა და **add** ფუნქციის გამოყენებით განსაზღვრეთ ფუნქცია, რომელიც ორ ნატურალურ რიცხვს ამრავლებს.

(2) განსაზღვრეთ სათანადო **fold** ფუქცია გამოსახულებებისათვის და მოიყვანეთ მისი გამოყენების რამდენიმე მაგალითი.

(3) ბინარული ხე **სრულია**, თუ ყოველი კვანძის ორი ქვეხე ერთნაირი ზომისაა. განსაზღვრეთ ფუნქცია, რომელიც არკვევს ხის სისრულეს.

დაპროგრამება Haskell ენაზე



თავი 11 – ამოცანა «კაუნტდაუნი» («**countdown**»)

რა არის კაუნტდაუნი (Countdown)?

- პოპულარული ტელევიქტორინაა, რომელიც პირველად გაჩნდა ბრიტანეთის ტელევიზია-ზე 1982 წელს.

- ეფუძნება ფრანგული სატელევიზიო შოუ-თა-მაშის ორიგინალურ ვერსიას "Des Chiffres et Des Lettres" («ციფრები და ასოები»).

- შეიცავს რიცხვებთან თამაშს, რომელსაც ვუ-წოდებთ *კაუნტდაუნის ამოცანას*.

მაგალითი

გამოვიყენოთ შემდეგი რიცხვების «*წყარო*»

1 3 7 10 25 50

და არითმეტიკული *ოპერატორები*

\+ − * ÷

ამის საფუძველზე ავაგოთ გამოსახულება, რომლის მნიშვნელობა, ანუ «*მიზანი*», არის 765

წესები

- ყველა რიცხვი, შუალედური შედეგების ჩათვლით, უნდა იყოს *დადებითი ნატურალური* (1, 2, 3, ...).

- გამოსახულების აგებისას ყოველი რიცხვი წყაროდან შეიძლება იყოს გამოყენებული *მაქსიმუმ ერთხელ*.

- ტელევიზიაზე პრაგმატული მოსაზრებებით მიღებული სხვა წესები აქ *იგნორირებულია*.

ვთქვათ, ჩვენი მაგალითისათვის, ერთ-ერთი შესაძლო ამონახსნია

```
(25-10) * (50+1)  =  765
```

შენიშვნები:

- ამ მაგალითისათვის 780 ამონახსნი არსებობს.
- მიზნის შეცვლა რიცხვით 831 იძლევა მაგალითს, რომელსაც ამონახსნი არ აქვს.

გამოსახულებათა გამოთვლა

ოპერატორები:

```
data Op = Add | Sub | Mul | Div
```

ოპერატორის გამოყენება:

```
apply           :: Op → Int → Int → Int
apply Add x y    = x + y
apply Sub x y    = x - y
apply Mul x y    = x * y
apply Div x y    = x `div` y
```

ნების დართვა, თუ ორი დადებითი ნატურალური რიცხვის მიმართ ოპერატორის გამოყენების შედეგი ასევე ნატურალურია:

```
valid           :: Op → Int → Int → Bool
valid Add _ _    = True
valid Sub x y    = x > y
valid Mul _ _    = True
valid Div x y    = x `mod` y == 0
```

გამოსახულებები:

```
data Expr = Val Int | App Op Expr Expr
```

გამოსახულების სრული მნიშვნელობის დაბრუნება, თუ იგი დადებითი ნატურალური რიცხვია:

```
eval                :: Expr → [Int]
eval (Val n)         = [n | n > 0]
eval (App o l r) = [apply o x y | x ← eval l
                                , y ← eval r
                                , valid o x y]
```

წარმატებისას - ერთელემენტიანი,
ხოლო წარუმატებლობისას -
ცარიელი სიების დაბრუნება.

ამოცანის ფორმალიზება

სიიდან ნული ან მეტი ელემენტის არჩევის ყველა შესაძლო ვარიანტთა სიის დაბრუნება:

```
choices :: [a] → [[a]]
```

მაგალითად:

```
> choices [1,2]

[[],[1],[2],[1,2],[2,1]]
```

გამოსახულების ყველა მნიშვნელობათა სიის დაბრუნება:

```
values                 :: Expr → [Int]
values (Val n)          = [n]
values (App _ l r) = values l ++ values r
```

ნების დათვა, თუ გამოსახულება წარმოადგენს ამონახსნს წყაროს რიცხვთა მოცემული სიისა და მიზნისათვის :

```
solution          :: Expr → [Int] → Int → Bool
solution e ns n = elem (values e) (choices ns)
                    && eval e == [n]
```

პირდაპირი («უხეში ძალით») ამოხსნა

ორ არაცარიელ ნაწილად სიის გაყოფის ყველა შესაძლო გზათა (ვერსიათა) სიის დაბრუნება:

```
split :: [a] → [([a],[a])]
```

მაგალითად:

```
> split [1,2,3,4]

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

გვიბრუნებს სიას, სადაც წარმოდგენილია ყველა შესაძლო გამოსახულება, რომელთა მნიშვნელობები ზუსტად მოცემული რიცხვების სიაა:

```
exprs     :: [Int] → [Expr]
exprs []   = []
exprs [n]  = [Val n]
exprs ns   = [e | (ls,rs) ← split ns
                , l       ← exprs ls
                , r       ← exprs rs
                , e       ← combine l r]
```

საკვანძო ფუნქცია ამ ლექციაში.

ორი გამოსახულების კომბინირება თითოეული ოპერატორის გამოყენებით:

```
combine     :: Expr → Expr → [Expr]
combine l r =
   [App o l r | o ← [Add,Sub,Mul,Div]]
```

სიის დაბრუნება, სადაც ყველა შესაძლო გამოსახულებაა, რომლებიც კაუნტდაუნის ამოცანის მოცემული მაგალითის ამოხსნას იძლევა:

```
solutions     :: [Int] → Int → [Expr]
solutions ns n = [e | ns' ← choices ns
                    , e   ← exprs ns'
                    , eval e == [n]]
```

რამდენად სწრაფია ყველაფერი ეს?

სისტემა:	1.2 გჰც Pentium M ლეპტოპი
კომპილატორი:	GHC ვერსია 6.4.1
მაგალითი:	`solutions [1,3,7,10,25,50] 765`
ერთი ამონახსნი:	0.36 წამი
ყველა ამონახსნი:	43.98 წამი

შესაძლებელია გაუმჯობესება?

- განხილლვისას მრავალი გამოსახულება, ჩვეულებრივ, **დაუშვებელი** იქნება – მათი გამოთვლა არ უნდა ხდებოდეს.

- ჩვენს მაგალითში, დაახლოებით, მხოლოდ 5 მილიონი გამოსახულებაა დასაშვები 33 მილიონ შესაძლოთა შორის.

- გენერირების კომბინირება გამოთვლასთან დაუშვებელ გამოსახულებათა ***წინასწარი განთესვის*** საშუალებას მოგვცემს.

ორი ფუნქციის გაერთიანება

დასაშვები გამოსახულებები და მათი მნიშვნელობები:

```
type Result = (Expr,Int)
```

ვეცადოთ განვსაზღვროთ ფუნქცია, რომელიც გააერთიანებს გამოსახულებათა გენერირებასა და გამოთვლას :

```
results   :: [Int] → [Result]
results ns = [(e,n) | e ← exprs ns
                    , n ← eval e]
```

ასეთი ქცევა მიიღწევა განსაზღვრებით

```
results []  = []
results [n] = [(Val n,n) | n > 0]
results ns  =
   [res | (ls,rs) ← split ns
        , lx      ← results ls
        , ry      ← results rs
        , res     ← combine' lx ry]
```

სადაც

```
combine' :: Result → Result → [Result]
```

შედეგების გაერთიანება:

```
combine' (l,x) (r,y) =
   [(App o l r, apply o x y)
      | o ← [Add,Sub,Mul,Div]
      , valid o x y]
```

ახალი ფუნქცია, რომლითაც წყდება კაუნტდაუნის (*countdown*) ამოცანები:

```
solutions'     :: [Int] → Int → [Expr]
solutions' ns n =
   [e | ns'   ← choices ns
      , (e,m) ← results ns'
      , m == n]
```

რამდენად სწრაფია ეს ახლა?

მაგალითი:

```
solutions' [1,3,7,10,25,50] 765
```

ერთი ამინახსნი: 0.04 წმ

ყველა ამონახსნი: 3.47 წმ

ორივე შემთხვევაში, დაახლოებით, 10-ჯერ უფრო სწრაფად.

შესაძლებელია გაუმჯობესება?

- მრავალი გამოსახულება თავიანთი არსით ერთნაირი აღმოჩნდება მარტივი არითმეტიკული თვისებების გამო, როგორიცაა, მაგალითად, შემდეგი იგივეობები:

```
x * y  =  y * x
```

```
x * 1  =  x
```

- ასეთი თვისებების გათვალისწინება ძებნისა და ამონახსნთა სივრცეების მნიშვნელოვან *რედუქციას* გამოიწვევს.

თვისებათა გამოყენება

დასაშვები პრედიკატის გაძლიერება კომუტატიურობისა და იდენტურობის თვისებათა გათვალისწინებით:

```
valid           :: Op → Int → Int → Bool
valid Add x y   = x ≤ y
valid Sub x y   = x > y
valid Mul x y   = x ≤ y && x ≠ 1 && y ≠ 1
valid Div x y   = x `mod` y == 0 && y ≠ 1
```

როგორი სისწრაფეა ახლა?

მაგალითი: `solutions'' [1,3,7,10,25,50] 765`

ვარგისია: 250000 გამოსახულება

დაახლოებით, 20-ჯერ ნაკლები.

ამონახსნი: 49 გამოსახულება

დაახლოებით, 16-ჯერ ნაკლები.

ერთი ამონახსნი: 0.02 წმ

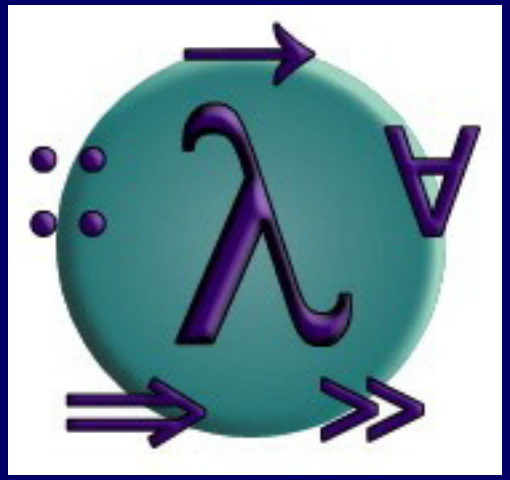
დაახლოებით, 2-ჯერ უფრო სწრაფად.

ყველა ამონახსნი: 0.44 წმ

დაახლოებით, 7-ჯერ უფრო სწრაფად.

მთლიანობაში ჩვენი პროგრამა, ჩვეულებრივ, ხსნის ამოცანებს სატელევიზიო ვიქტორინიდან მყისიერად, ხოლო ყველა ამონახსნს წამის განმავლობაში ადგენს.

დაპროგრამება Haskell ენაზე



თავი 12 – გადადებული (ზარმაცი) გამოთვლები

შესავალი

აქამდე ჩვენ არ მოგვცემია საშუალება დაწვრილებით გვესაუბრა გამოსახულებათა შეფასების თავისებურებებ-ბის შესახებ ჰასკელში.

სინამდვილეში გამოსახულებათა შეფასება ამ ენაში მარ-ტივი მეთოდით ხდება, რომლის, სხვათა შორის, მთავარი პრინციპები ასეთია:

- თავის შეკავება ზედმეტი, უსარგებლო გამოთვლებისაგან;
- პროგრამათა მეტი მოდულობის უზრუნველყოფა;
- დაპროგრამებისას უსასრულო სიებთან მუშაობის შესა-ძლებლობის შექმნა.

გამოთვლათა ასეთ მეთოდს გადადებული, ანუ ზარმაცი გამოთვლები ეწოდება, ხოლო თავად ჰასკელს – ზარმაცი ფუნქციონალური ენა.

გამოსახულებათა გამოთვლა 1

- ძირითადად, გამოსახულებები გამოითვლება ან გარდაიქმნება განსაზღვრებათა წარმატებული გამოყენებით, ვიდრე შემდგომი გამარტივება შეუძლებელი არ გახდება.
- მაგალითად, თუ გავითვალისწინებთ

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

განსაზღვრებას, მაშინ `square(3+4)` გამოსახულება შეიძლება შეფასდეს გარდაქმნათა შემდეგი მიმდევრობის გამოყენებით:

```
square  (3+4)
=
square  7
=
7 * 7
=
49
```

გამოსახულებათა გამოთვლა 2

მაგრამ შესაძლო გარდაქმნათა მიმდევრობის გამოყენება ერთადერთი გზით არ ხდება. მაგალითად:

```
square   (3+4)
=
(3+4)* (3+4)
=
7 *(3+4)
=
7*7
=
49
```

ახლა ჩვენ გამოვიყენეთ კვადრატში აყვანა შეკრების ოპერაციამდე, მაგრამ საბოლოო შედეგი ისეთივე მივიღეთ.

ფაქტი: ჰასკელში გამოსახულების შეფასება ორი სხვადასხვა (მაგრამ სასრული) გზით ერთსა და იმავე შედეგს იძლევა.

გარდაქმნათა სტრატეგიები

ყოველ ეტაპზე გამოსახულების გამოთვლის პროცესში დასაშვებია გაჩნდეს მრავალი ქვეგამოსახულება, რომლის რედუცირება განსაზღვრების გამოყენებით შეიძლება.

რედექსის, ანუ რედუცირებადი ქვეგამოსახულების (REDucible subEXpression) ასარჩევად ორი ზოგადი სტრატეგია არსებობს.

1. შიდა რედუქცია
   შიდა რედექსი ყოველთვის რედუცირებადია;

2. გარე რედუქცია
   გარე რედექსი ყოველთვის რედუცირებადია.

   როგორ ხდება ორი სტრატეგიის შედარება...?

დასრულებადობა 1

მოცემულია განსაზღვრება:

```
loop  = tail loop
```

შევაფასოთ `fst (1,loop)` გამოსახულება რედუქციის ამ ორი სტრატეგიის გამოყენებით:

1. შიდა რედუქცია

```
fst (1, loop)
=
fst (1,tail loop)
=
fst (1,tail (tail loop))
=
…
```

ეს სტრატეგია დასრულებადი არ არის.

დასრულებადობა 2

2. გარე რედუქცია

```
fst (1,loop)
=
1
```

ეს სტრატეგია იძლევა შედეგს ერთი ბიჯით.

ფაქტები

- გარე რედუქციამ შეიძლება მოგვცეს შედეგი მაშინაც კი, როცა შიდა რედუქცია დასრულებადი არ არის;

- თუ მოცემული გამოსახულებისათვის საერთოდ არსებობს რედუქციათა რომელიმე სასრული მიმდევრობა, მაშინ გარე რედუქცია ასევე სასრული იქნება იმავე შედეგით.

რედუქციათა რიცხვი 1

კვლავ განვიხილოთ შემდეგი რედუქციები:

შიდა

```
square (3+4)
=
Square 7
=
7 * 7
=
49
```

გარე

```
square (3+4)
=
(3+4)    * (3+4)
=
7*(3+4)
=
7 * 7
=
49
```

გარე ვერსია არაეფექტურია: 3+4 ქვეგამოსახულება მეორდება კვადრატში ახარისხების რედიცირებისას და ამიტომ შემდეგ ორჯერ გვიხდება გამარტივების განხორციელება.

ფაქტი: გარე რედუქციამ შეიძლება მოითხოვოს უფრო მეტი მოქმედება, ვიდრე შიდამ.

რედუქციათა რიცხვი 2

ეს პრობლემა შეიძლება გადაიჭრას მაჩვენებლებით გამოსახულებათა ერთობლივი გამოყენების საჩვენებლად გამოთვლაში.

```
Square (3+4)
=
(° * °)                 აქ ° არის (3+4)-ის მაჩვენებელი
=
(° * °)                 აქ ° არის 7-ის მაჩვენებელი
=
49
```

ეს რედუქციის ახალ სტრატეგიას იძლევა:

ძარმაცი გამოთვლა = გარე რედუქცია + ერთობლივად გამოყენება

ფაქტები:

- ზარმაცი გამოთვლა არასოდეს ითხოვს რედუქციათა უფრო მეტ ბიჯს, ვიდრე შიდა რედუქცია;
- ჰასკელი იყენებს ზარმაც (გადადებულ) გამოთვლას.

უსასრულო სიები 1

დასრულებადობის უპირატესობასთან ერთად, ზარმაცი გამოთვლა მნიშვნელობათა უსასრულო სიების დაპროგრამების საშუალებასაც იძლევა!

განვიხილოთ რეკურსიული განსაზღვრება

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

რეკურსიის რამდენიმეჯერ განხორციელება იძლევა:

```
 ones = 1:ones
      = 1:1:ones
      = 1:1:1:ones
      =   …
```

ამრიგად, მიიღება ერთიანების უსასრულო სია.

უსასრულო სიები 2

კვლავ განვიხილოთ **head ones** გამოსახულების შეფასება შიდა რედუქციისა და გამოთვლის გამოყენებით:

1. შიდა რედუქცია

```
head  ones = head (1:ones)
           = head (1:1:ones)
           = (1:1:1:ones)
           = ...
```

ამ შემთხვევაში გამოთვლა დაუსრულებლად გრძელდება.

2. ძარმაცი გამოთვლა

```
head  ones = (1:ones)
           = 1
```

ამ შემთხვევაში გამოთვლა იძლევა შედეგს 1-ის სახით.

უსასრულო სიები 3

ამრიგად, ზარმაცი გამოთვლების გამოყენებით, ones უსასრულო სიაში მხოლოდ პირველი მნიშვნელობაა ფაქტობრივად ნაწარმოები, რადგან მხოლოდ იგია საჭირო მთელი head ones გამოსახულების გამოსათვლელად.

შაზოგადოდ გვაქვს შემდეგი სლოგანი (დევიზი, ლოზუნგი):

ზარმაცი გამოთვლების საშუალებით გამოსახულებები გამოითვლება ზუსტად იმ მოცულობით, რაც აუცილებელია საბოლოო შედეგის მისაღებად.

ახლა ვხედავთ, რომ

```
ones =1 : ones
```

ნამდვილად განსაზღვრავს პოტენციურად უსასრულო სიას, რომელიც ფასდება ზუსტად გამოყენების შინაარსის მოთხოვნიდან გამომდინარე...

მოდულური დაპროგრამება

შეიძლება სასრული სიების გენერირება უსასრულო სიებიდან სამეტყველო ელემენტის გამოყენებით. მაგალითად:

```
? take 5 ones
[1,1,1,1,1]

? Take 5 [1..]
[1,2,3,4,5]
```

ზარმაცი გამოთვლები საშუალებას გვაძლევს გავხადოთ პროგრამები უფრო მოდულური მონაცემებში შინაარსობრივი ფორმის განცალკევებით:

```
Take 5 [1..]
|----| |----|
მართვა  მონაცემები
```

ზარმაცი გამოთვლების საშუალებით **მონაცემები** გამოითვლება ზუსთად **მართვის** კომპონენტის მოთხოვნის შესაბამისად.

მაგალითი : მარტივი რიცხვების გენერირება 1

ელემენტარული პროცედურა ყველა *მარტივი რიცხვის* უსასრულო სიის მისაღებად ასეთია:

1. 2, 3, 4, ... სიის ჩაწერა;
2. სიაში პირველი p სიდიდის მონიშვნა მარტივ რიცხვად;
3. სიიდან p-ს ჯერადი ყველა რიცხვის ამოშლა;
4. მე-2 ბიჯზე გადასვლა.

პირველი რამდენიმე ბიჯი შეიძლება შემდეგი ცხრილით გამოიხატოს:

2	3	4	5	6	7	8	9	10	11	12	...
	3		5	_	7		9		11	_	...
			5		7			_	11		...
					7				11		...
									11		...

მაგალითი : მარტივი რიცხვების გენერირება 2

ეს პროცედურა ცნობილია «ერატოსფენის საცრის» სახელწოდებით, ძველი ბერძენი მათემატიკოსის პატივსაცემად, რომელმაც პირველმა აღწერა ხსენებული ალგორითმი.

იგი შეიძლება პირდაპირ იყოს გადაყვანილი ჰასკელში

```
primes          :: [Int]
primes           = seive [z..]
seive          :: [Int] –›[Int]
seive (p:xs) = p:seive [x|x‹–xs, x`mod`p /=0]
```

და შემდეგნაირად შესრულდეს:

```
? Primes
[2,3,5,7,11,13,17,19,23,29,31,
 37,41,43,47,53,59,61, 67, …
```

მაგალითი : მარტივი რიცხვების გენერირება 3

თუ მარტივ რიცხვთა გენერირებისას უარს ვიტყვით სასრულობის შეზღუდვაზე, მივიღებთ მოდულურ განსაზღვრებას, სადაც შეიძლება იყოს შემოტანილი სხვადასხვა სასაზღვრო პირობა ამა თუ იმ ვითარებაში.

პირველი ათი მარტივი რიცხვის არჩევა:

```
?  take  10  primes
[2,3,5,7,11,13,17,19,23,29]
```

მარტივი რიცხვების არჩევა, რომლებიც ნაკლებია 15-ზე:

```
?  take  while ( <15 ) primes
[2,3,5,7,11,13]
```

ზარმაცი გამოთვლები მოხერხებული დაპროგრამებაა!

განსართობი სავარჯიშოები

1. განსაზღვრეთ
   `fibs :: [Integer]`
   პროგრამა, რომელიც წარმოქმნის ფიბონაჩის უსასრულო
   `[0,1,1,2,3,5,8,13,21,34, ...`
   მიმდევრობას შემდეგი მარტივი პროცედურის გამოყენებით:

   a) პირველი ორი რიცხვია 0 და 1;

   b) შემდეგი რიცხვი წინა ორის ჯამია;

   c) გადასვლა **b** ბიჯზე.

2. განსაზღვრეთ
   `fib :: Int -> Integer`
   ფუნქცია, რომელიც ანგარიშობს ფიბონაჩის მე-***n*** რიცხვს.