თბილისის სახელმწიფო უნივერსიტეტი

ზუსტი და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

ინტერდისციპლინური (მათემატიკა, კომპიუტერული მეცნიერებები)
ქვემიმართულება: მათემატიკური ლოგიკა და დისკრეტული სტრუქტურები

ბაკალავრიატი

დისკრეტული სტრუქტურების გაფართოვება: პროგრამული ენა ჰასკელი

K. Doets, J. van Eijck. The Haskell Road to Logic, Maths and Programming. Texts in Computer Series, 2004; Website: haskell.org.; Graham Hutton.Programming in Haskell. 2005; Tutorials:Learn You a Haskell for Great Good! http://tryhaskell.org/; http://www.haskell.ru/

საფუძველზე

ლექციათა კურსი

რევაზ გრიგოლია და ტატიანა კისელიოვა

თბილისი
2011

ლექცია № 1

1 სამუშაოს მიზანი

ენის ინტერპრეტატორის *Haskell*-ის მუშაობის უნარების ათვისება. *Haskell*-ის ტიპის ენის საფუძვლებზე წარმოდგენის მიღება. მარტივი ფუნქციების განსაზღვრის ათვისება

2 **Hugs**-ის ინტერპრეტატორთან მუშაობის საფუძვლები

ლაბორატორიული სამუშაოების ჩასატარებლად გამოიყენება *Haskell*-ის ტიპის ენის ინტერპრეტატორი. არსებობს ინტერპრეტატორების რამოდენიმე რეალიზაცია; წარმოდგენილ კურსში გამოიყენება ინტერპრეტატორი *Hugs* (ეს დასახელება არის სიტყვების **"Haskell users' Gofer system"** აბრევიატურა**, Gofer —** არის პროგრამირების ენის დასახელება, რომელიც იყო *Haskell*-ის ენის ერთერთი წინამორბედი).

Hugs-ის ინტერპრეტატორის გაშვების შემდეგ ეკრანზე ჩნდება დამმუშავებელი გარემოს დიალოგური ფანჯარა, ავტომატურად იტვირთება ტიპების წინასწარ განსაზღვრის და სტანდარტული ფუნქციების განსაზღვრის ფაილი *Haskell*-ის ენაზე (*Prelude.hs*)**,** და გამოიყვანება სტანდარტული შემოთავზება სამუშაოზე. ამ შემათავაზებას აქვს სახე *Prelude* >**;** ზოგადად, > სიმბოლოს წინ გამოიყვანება ბოლო ჩატვირთული მოდულის სახელი.

შემოთავაზების გამოტანის შემდეგ შეიძლება ავკრიფოთ ენის გამოსახულება ან ინტერპრეტატორის ბრძანება. ინტერპრეტატორის ბრძანება განსხვავდება *Haskell*-ის გამოსახულებისგან იმით, რომ იწყება სიმბოლო ორიწერტილით (:)**.** მაგალითად, ინტერპრეტატორის ბრძანებაა : *quit***,** რომელის შესრულებაც იწვევს ინტერპრეტატორის მუშაობის დამთავრებას. ინტერპრეტატორის ბრძანებები შეიძლება შემოკლდეს ერთ ასომდე. ასე, რომ, ბრძანებები : *quit* და : *q* ექვივალენტურია. ბრძანება : *set* გამოიყენება ინტერპრეტატორის სხვადასხვა ოფციების დასაყენებლად. ბრძანება :? -ს გამოჰყავს ინტერპრეტატორის ყველა ბრძანების სია. შემდგომში ჩვენ სხვა ბრძანებებსაც განვიხილავთ.

3 ტიპები

Haskell ენის პროგრამები არიან გამოსახულებები, რომელთა გამოთვლებს მივყავართ მნიშვნელობებამდე. თითოეულ მნიშვნელობას აქვს ტიპი. ინტუიციურად, ტიპი შეიძლება გავიგოთ როგორც გამოსახულების დასაშვები მნიშვნელობების სიმრავლე. იმისათვის, რომ განვსაზღროთ, რა ტიპი აქვს მოცემულ გამოსახულებას, საჭიროა გამოვიყენოთ ინტერპრეტატორის ბრძანება : *type* (ან : *t*). ამას გარდა, შესაძლოა გამოვიყენოთ ბრძანება : *set* + *t*, რათა ინტერპრეტატორმა ავტომატურად დაბეჭდოს ყოველი გამოთვლილი გამოსახულების ტიპი. ენა *Haskell* -ის ტიპებს წარმოადგენს:

- ტიპები *Integer* და *Int* - გამოიყენება მთელი რიცხვების წარმოსადგენად, ამასთან ტიპი *Integer* -ის სიგრძე არ არის შეზღუდული.
- ტიპები *Float* და *Double* გამოიყენება ნამდვილი რიცხვების წარმოსადგენად.
- ტიპი *Bool* შეიცავს ორ მნიშვნელობას: *True* და *False* და დანიშნულია ლოგიკური გამოსახულებების შედეგის წარმოსადგენად.
- ტიპი *Char* გამოიყენება სიმბოლოების წარმოსადგენად.

ტიპების სახელები *Haskell* ენაში ყოველთვის იწყება დიდი (მთავრული) ასოებით.

ენა *Haskell* წარმოადგენს ძლიერ ტიპიზირებულ პროგრამირების ენას. მიუხედავად ამისა, ხშირ შემთხვევებში პროგრამისტი არ არის ვალდებული განაცხადოს, თუ რომელ ტიპს ეკუთვნის მის მიერ შემოტანილი ცვლადი. ინტერპრეტატორს თვითონ აქვს შესაძლებლობა გამოიყვანოს მომხმარებლის მიერ გამოყენებული ცვლადის ტიპი. თუმცა, თუ რაიმე მიზნისთვის საჭიროა გამოცხადდეს, რომ მნიშვნელობა ეკუთვნის რომელიღაც ტიპს, გამოიყენება კონსტრუქცია: ცვლადი :: ტიპი. თუ ჩართულია ინტერპრეტატორის ოფცია +*t*, მაშინ ინტერპრეტატორი ამავე ფორმატში დაბეჭდავს მნიშვნელობებს. ქვემოთ მოყვანილია ინტერპრეტატორთან მუშაობის სესია. იგულისხმება, რომ ტექსტი მოსაწვევი *Prelude* > -ის შემდეგ, შეყვანილია მომხმარებლის მიერ, ხოლო მისი მომდევნო ტექსტი არის სისტემის პასუხი.

Prelude >: *set* + *t*

$Prelude > 1$
$1 :: Integer$
$Prelude > 1.2$
$1.2 :: Double$
$Prelude > a$
$a :: Char$
$Prelude > True$
$True :: Bool$

მოცემული ოქმიდან ჩანს, რომ ტიპები $Integer, Double$ და $Char$ -ის მნიშვნელობები მოიცემა იგივე წესებით, როგორც ენა C -ში.

ტიპების სისტემის განვითარებისა და მკაცრი ტიპიზაციის შედეგად *Haskell* ენის პროგრამები არის უსაფრთხო ტიპების მიხედვით. გარანტირებულია, რომ *Haskell* ენის სწორ პროგრამაში ტიპები სწორად გამოიყენება. პრაქტიკულად, ეს ნიშნავს, რომ *Haskell* ენაზე პროგრამა შესრულებისას ვერ გამოიწვევს შეცდომას მეხსიერების შეღწევადობაზე (*Access violation*). ასევე, გარანტირებულია, რომ პროგრამაში ცვლადების გამოყენება საწყისი ინიციალიზების გარეშე არ მოხდება. ამრიგად, პროგრამაში მრავალი შეცდომის არსებობა მოწმდება კომპილაციის ეტაპზე და არა შესრულების ეტაპზე.

4 არითმეტიკა

ინტერპრეტატორი *Hugs* შეიძლება გამოყენებული იყოს არითმეტიკული გამოსახულებების გამოსათვლელად. ამასთან, შესაძლოა გამოყენებული იყოს ოპერატორები: $+, -, *, /$ (მიმატება, გამოკლება, გამრავლება, გაყოფა) პრიორიტეტების ჩვეულებრივი წესებით. ამასთან, შეიძლება გამოყენებული იყოს ოპერატორი ^ (ხარისხში აყვანა). ამრიგად, მუშაობის სეანს შეიძლება ჰქონდეს შემდეგი სახე:

$Prelude > 2 * 2$
$4 :: Integer$
$Prelude > 4 * 5 + 1$
$21 :: Integer$
$Prelude > 2\hat{}3$
$8 :: Integer$

ამასთან, შეიძლება გამოვიყენოთ სტანდარტული მათემატიკური ფუნქციები $sqrt$ (კვადრატული ფესვი), sin, cos, exp და ა.შ. პროგრამირების სხვა ენებისგან განსხვავებით, *Haskell* -ში ფუნქციის გამოძა-

ხებისას არ არის აუცილებელი არგუმენტის ფრჩხილებში ჩასმა. ამრიგად, შეიძლება მარტივად დაიწეროს *sqrt*2, და არა *sqrt*(2). მაგალითი:

Prelude > *sqrt* 2
1.4142135623731 :: *Double*
Prelude > 1 + *sqrt* 2
2.4142135623731 :: *Double*
Prelude > *sqrt* 2 + 14 2.4142135623731 :: *Double*
Prelude > *sqrt* (2 + 1)
1.73205080756888 :: *Double*

ამ მაგალითიდან შეიძლება დავასკვნათ, რომ ფუნქციის გამოძახებას უფრო მაღალი პრიორიტეტი აქვს, ვიდრე არითმეტიკულ ოპერაციებს, ასე, რომ გამოსახულება *sqrt* 2 + 1 ინტერპრეტირდება როგორც (*sqrt* 2) + 1, და არა როგორც *sqrt* (2 + 1). გამოთვლების ზუსტი რიგის მისათითებლად საჭიროა გამოყენებული იყოს ფრჩხილები. ფუნქციის გამოძახებას ყოველთვის უფრო მაღალი პრიორიტეტი აქვს, ვიდრე ნებისმიერ ბინარულ ოპერაციას.

საჭიროა ასევე აღინიშნოს, რომ პროგრამირების ბევრი სხვა ენისგან განსხვავებით, მთელრიცხვებიანი გამოსახულება *Haskell* -ზე გამოითვლება თანრიგების შეუზღუდავი რიცხვით. (შეეცადეთ გამოთვალოთ გამოსახულება 2^5000.) *C* ენისგან განსხვავებით, სადაც *int* ტიპის მაქსიმალური მნიშვნელობა შეზღუდულია მანქანის თანრიგებრიობით (თანამედროვე მანქანებზე იგი ტოლია $2^{31} - 1 = 2147483647$), *Haskell* ენაში ტიპმა *Integer* -მა შეიძლება წარმოადგინოს ნებისმიერი სიგრძის მთელი რიცხვი.

5 კორტეჟები

ზემოთ ჩამოთვლილი მარტივი ტიპების გარდა ასკელლ-ში შეიძლება განვსაზღვროთ შედგენილი ტიპის მნიშვნელობებიც. მაგალითად, სიბრტყეზე წერტილების მოსაცემად აუცილებელია ორი რიცხვი, რომლებიც მათ კოორდინატებს შეესაბამება. ენა *Haskell* -ში რიცხვების წყვილი შეიძლება მოვცეთ ასე: ჩამოვთალოთ კომპონენტები, გამოვყოთ მძიმეებით და ავიღოთ ფრჩხილებში: (5,3). არ არის აუცილებელი, რომ რიცხვების კომპონენტები იყოს ერთიდაიგივე ტიპის. შეიძლება შევადგინოთ წყვილი, რომლის პირველი კომპონენტი შეიძლება იყოს სტრიქონი, მეორე მთელი რიცხვი და ა.შ.

ზოგადად, თუ a და b *Haskell* ენის ნებისმიერი ტიპია, მაშინ იმ წყვილის ტიპი, რომელშიც პირველი ელემენტი ეკუთვნის ტიპს a, ხოლო მეორე - ტიპს b, აღინიშნება ასე: (a, b). მაგალითად, წყვილს $(5, 3)$ აქვს ტიპი $(Integer, Integer)$; წყვილი $(1, a)$ ეკუთვნის ტიპს $(Integer, Char)$. შეიძლება მოვიყვანოთ უფრო რთული მაგალითი: წყვილი $((1, a), 1.2)$ ეკუთვნის ტიპს $((Integer, Char), Double)$. შეამოწმეთ ეს ინტერპრეტატორის საშუალებით.

ყურადღება მივაქციოთ იმას, რომ თუმცა შემდეგი სახის კონსტრუქციები (1,2) და $(Integer, Integer)$ მსგავსად გამოიყურება, ენა *Haskell* -ში ისინი აღნიშნავენ სხვადასხვა ცნებებს. პირველი მათგანი წარმოადგენს მნიშვნელობას, მაშინ როცა მეორე არის ტიპი.

წყვილებთან სამუშაოდ ენა *Haskell* -ში არსებობს სვანდარტული ფუნქციები fst და snd,რომლებიც, შესაბამისად, აბრუნებენ სიის პირველ და მეორე ელემენტებს. ამ ფუნქციების დასახელება წარმოშობილია ინგლისური სიტყვებიდან *first* (პირველი) და *second* (მეორე). ამრიგად, ისინი შეიძლება გამოვიყენოთ შემდეგნაირად:

$Prelude > fst(5, True)$
$5 :: Integer$
$Prelude > snd(5, True)$
$True :: Bool$

ანალოგიურად, გარდა წყვილებისა, შეიძლება განვსაზღვროთ სამეულები, ოთხეულები და ა.შ. მათი ტიპები ჩაიწერება შესაბამისი სახით:

$Prelude > (1, 2, 3)$
$(1, 2, 3) :: (Integer, Integer, Integer)$
$Prelude > (1, 2, 3, 4)$
$(1, 2, 3, 4) :: (Integer, Integer, Integer, Integer)$

მონაცემების ასეთ სტრუქტურას უწოდებენ კორტეჟს. კორტეჟში შეიძლება შენახული იყოს ფიქსირებული რაოდენობის სხვადასხვა მონაცემი. ფუნქციები fst და snd განსაზღვრულია მხოლოდ წყვილებისთვის და არ მუშაობს სხვა კორტეჟებთან. თუ მათ გამოვიყენებთ, მაგალითად, სამეულთან, ინტერპრეტატორი შეგვატყობინებს შეცდომის შესახებ.

კორტეჟის ელემენტი შეიძლება იყოს ნებისმიერი ტიპის, მათ შორის სხვა კორტეჟიც. იმ კორტეჟის ელემენტებთან წვდომისთვის, რომლებიც წყვილებს წარმოადგენენ, შეიძლება გამოვიყენოთ fst

და *snd* ფუნქციების კომბინაცია. შემდეგი მაგალითი გვიჩვენებს a ელემეტის ამოღებას კორტეჟიდან $(1, ('a', 23.12))$:

$Prelude > fst(snd(1, ('a', 23.12)))$
$a :: Char$

6 სიები

კორტეჟისაგან განსხვავებით, სიამ შეიძლება შეინახოს ელემენტების ნებისმიერი რაოდენობა. ენა *Haskell* -ში სია განისაზღვრება ასე: კვადრატულ ფრჩხილებში ჩამოითვლება ელემენტები და ერთმანეთისგან მძიმით გამოიყოფა. ელემენტები აუცილებლად უნდა ეკუთვნოდეს ერთიდაიგივე ტიპს. სიის ტიპი, რომელიც შედგება a ტიპის ელემენტებისგან, აღინიშნება როგორც $[a]$.

$Prelude > [1, 2]$
$[1, 2] :: [Integer]$
$Prelude > ['1', '2', '3']$
$['1', '2', '3'] :: [Char]$

სიაში შეიძლება არცერთი ელემენტი არ შედიოდეს. ცარიელი სია აღინიშნება როგორც $[]$. ოპერატორი $:$ (ორი წერტილი) გამოიყენება სიის თავში ელემენტის დასამატებლად. მისი მარცხენა არგუმენტი უნდა იყოს ელემენტი, მარჯვენა - სია:

$Prelude > 1 : [2, 3]$
$[1, 2, 3] :: [Integer]$
$Prelude > 5 : [1, 2, 3, 4, 5]$
$[5, 1, 2, 3, 4, 5] :: [Char]$
$Prelude > False : []$
$[False] :: [Bool]$

ოპერატორი $(:)$ და ცარიელი სიისგან შეიძლება ავაგოთ ნებისმიერი სია:

$Prelude > 1 : (2 : (3 : []))$
$[1, 2, 3] :: Integer$

ოპერატორი $(:)$ მარჯვნიდან ასოციატიურია, ამიტომ ზემოთ მოყვანილ გამოსახულებაში შეიძლება გამოვტოვოთ ფრჩხილები:

$Prelude > 1 : 2 : 3 : []$
$[1, 2, 3] :: Integer$

სიის ელემენტები შეიძლება იყოს ნებისმიერი მნიშვნელობები - რიცხვები, სიმბოლოები, კორტეჟები, სხვა სიები და ა.შ.

$Prelude > [(1,'a'), (2,'b')]$
$[(1,'a'), (2,'b')] :: [(Integer, Char)]$
$Prelude > [[1, 2], [3, 4, 5]]$
$[[1, 2], [3, 4, 5]] :: [[Integer]]$

ენა *Haskell* -ში სიებთან სამუშაოდ არსებობს ფუნქციათა დიდი რაოდენობა. ჩვენ მხოლოდ ზოგიერთ მათგანს განვიხილავთ.

- ფუნქცია *head* აბრუნებს სიის პირველ ელემენტს.
- ფუნქცია *tail* აბრუნებს სიას პირველი ელემენტის გარეშე.
- ფუნქცია *length* აბრუნებს სიის სიგრძეს.

ფუნქციები *head* და *tail* განსაზღვრულია არაცარიელი სიებისთვის. იმ შემთხვევაში, თუ ისინი გამოიყენება ცარიელ სიებთან, ინტერპრეტატორს გამოაქვს შეტყობინება შეცდომის შესახებ. ამ ფუნქციებთან მუშაობის მაგალითებია:

$Prelude > head[1, 2, 3]$
$1 :: Integer$
$Prelude > tail[1, 2, 3]$
$[2, 3] :: [Integer]$
$Prelude > tail[1]$
$[] :: Integer$
$Prelude > length[1, 2, 3]$
$3 :: Int$

შევნიშნოთ, რომ ფუნქცია *length* ეკუთვნის ტიპს *Int* და არა *Integer* -ს.

სიების გაერთიანებისთვის (კონკატენაციისთვის) *Haskell* -ში განსაზღვრულია ოპერატორი ++.

$Prelude > [1, 2] + +[3, 4]$
$[1, 2, 3, 4] :: Integer$

7 სტრიქონები

სტრიქონული მნიშვნელობები ენა *Haskell* -ში, ისევე როგორცჩ

Prelude > "*hello*"
"*hello*" :: *String*

სტრიქონი წარმოადგენს სიმბოლოების სიას. ასე, რომ გამოსახულებები "*hello*", ['*h*','*e*','*l*','*l*','*o*'] და '*h*' : '*e*' : '*l*' : '*l*' : '*o*' : [ ] აღნიშნავს ერთი და იმავეს, ხოლო ტიპი *String* წარმოადგენს [*Char*] -ის სინონიმს. ყველა ფუნქცია, რომელიც მუშაობს სიებთან, შეიძლება გამოვიყენიოთ სტროქონებთანაც:

Prelude > *head*"*hello*"
h :: *Char*
Prelude > *tail*"*hello*"
"*hello*" :: [*Char*]
Prelude > *length*"*hello*"
5 :: *Int*
Prelude > "*hello*" + +", *world*"
"*hello*, *world*" :: [*Char*]

რიცხვითი მნიშვნელობების გარდაქმნისთვის სტრიქონებად და პირიქით, არსებობს ფუნქციები *read* და *show*:

Prelude > *show* 1
"1" :: [*Char*]
Prelude > "*Formula*" + +*show* 1
"*Formula*1" :: [*Char*]
Prelude > 1 + *read* "12"
13 :: *Integer*

იმ შემთხვევაში, თუ ფუნქცია *show* ვერ გარდაქმნის სტრიქონს რიცხვად, გამოდის შეცდომა.

8 ფუნქციები

ჩვენ აქამდე ვიყენებდით ენა *Haskell* -ის სტანდარტულ ფუნქციებს. ვნახოთ, როგორ შეიძლება განისაზღვროს მომხმარებლის ფუნქ-

ციები. განვიხილოთ ინტერპრეტატორის რამდენიმე ბრძანება (გავიხსენოთ, რომ შესაძლებელია ამ ბრძანებების შემოკლება ერთ ასომდე).

- ბრძანება : *load* საშუალებას იძლევა ჩაიტვირთოს *Haskell* პროგრამა მოცემული ფაილიდან.
- ბრძანება : *edit* უშვებს ბოლო ჩატვირთული ფაილის რედაქტირების პროცესს.
- ბრძანება : *reload* თავიდან კითხულობს ბოლოს ჩატვირთულ ფაილს.

მომხმარებლის მიერ განსაზღვრული ფუნქცია უნდა იყოს ფაილში, რომელიც ჩაიტვირთება *Hugs* ინტერპრეტატორით ბრძანება : *load* -ის საშუალებით. ჩატვირთული ფაილის რედაქტირებისთვის შეიძლება გამოვიყენოთ ბრძანება : *edit*. ის გაუშვებს გარე რედაქტორს (შეთანხმების პრინციპით - ეს არის *Notepad*). რედაქტირების პროცესის დამთავრების შემდეგ აუცილებელია დავხუროთ რედატორი. თუმცა, ფაილი შეიძლება შევცვალოთ უშუალოდ *Windows* ოპერაციული სისტემის გარსიდან. ამ შემთხვევაში, იმისთვის რომ ინტერპრეტატორმა თავიდან წაიკითხოს ფაილი, საჭიროა ცხადად გამოვიძახოთ ბრძანება : *reload*. განვიხილოთ მაგალითი. შევქმნათ რომელიღაც კატალოგში ფაილი *lab*1.*hs*. დავუშვათ, ამ ფაილის სრული გზაა - `:\labs\lab1.hs` . *Hugs* ინტერპრეტატორში შევასრულოთ შემდეგი ბრძანება:

```
Prelude>:load "c:\\labs\\lab1.hs"
```

თუ ჩატვირთვა წარმატებით დამთავრდა, ინტერპრეტატორის მოსაწვევი იცვლება *Main* > -ით. საქმე იმაშია, რომ თუ არ მივუთითებთ მოდულის სახელს, ითვლება, რომ ის არის *Main*.

```
Main>:edit
```

აქ უნდა გაიხსნას რედაქტორის ფანჯარა, რომელშიც უნდა შევიტანოთ პროგრამის ტექსტი. შევიტანოთ:

```
x = [1,2,3]
```

შევინახეთ ფაილი და დავხუროთ რედაქტორი. ინტერპრეტატორი *Hugs* ჩატვირთავს ფაილს `:\labs\lab1.hs` და ეხლა x ცვლადის მნიშვნელობა იქნება განსაზღვრული:

```
Main>x
[1,2,3] :: [Integer]
```

ყურადღება მივაქციოთ, რომ ფაილის სახელის ჩაწერისას : *load* ბრძანების არგუმენტში სომბოლო \ დუბლირდება. ისევე, როგორც ენა C -ში, ენა *Haskell* -შიც სიმბოლო \ –ით იწყება მოსამსახურე სიმბოლოები ('\n' და ა.შ.). თვითონ სიმბოლო \ –ის გამოყენებისთვის საჭიროა კიდევ ერთი \ –ის მითითება, ისევე, როგორც C ენაშია. გადავიდეთ ფუნქციის განსაზღვრაზე. ზემოთ აღწერილი პროცესის შესაბამისად შევქმენათ რაიმე ფაილი და ჩავწეროთ მასში შემდეგი ტექსტი:

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

პირველი სტრიქონი *square* (*square* :: *Integer*− > *Integer*) მიუთითებს, რომ ჩვენ შემოგვაქვს ფუნქცია *square* –ს განსაზღვრება, რომლის პარამეტრია *Integer* ტიპის და აბრუნებს *Integer* ტიპის მნიშვნელობას. მეორე სტრიქონი (*square* $x = x * x$) წარმოადგენს უშუალოდ ფუნქციის აღწერას. ფუნქცია *square* ღებულობს ერთ არგუმენტს და აბრუნებს მის კვადრატს. ფუნქციები ენა *Haskell* –ში წარმოადგენენ „პირველი კლასის“ მნიშვნელობებს. ეს ნიშნავს, რომ ისინი არიან „თანაბარუფლებიანი“ ისეთი მნიშვნელობების, როგორიცაა მთელი და ნამდვილი რიცხვები, სიმბოლოები, სტრიქონები, სიები და ა.შ. ფუნქციები შეიძლება გადაეცეს სხვა ფუნქციებს არგუმენტად, დაბრუნდეს როგორც მნიშვნელობები და ა.შ. ისევე როგორც ყველა მნიშვნელობას *Haskell* –ში, ფუნქციასა აქვს ტიპი. ფუნქციის ტიპი, რომელიც იღებს a ტიპის მნიშვნელობას და აბრუნებს b ტიპის მნიშვნელობას, აღინიშნება ასე: $a- > b$.

შექმნილი ფაილი ჩავტვირთოთ ინტერპრეტატორში და შევასრულოთ შემდეგი ბრძანებები:

```
Main>:type square
square :: Integer -> Integer
Main>square 2
4 :: Integer
```

შევნიშნოთ, რომ *square* ფუნქციის ტიპის გამოცხადება არ იყო აუცილებელი: ინტერპრეტატორს თვითონ შეუძლია აუცილებელი ინფორმაციის გამოყვანა ფუნქციის ტიპის შესახებ მისი აღწერიდან. თუმცა, ჯერ ერთი, გამოყვანილი ტიპი იქნებოდა უფრო დიდი,

ვიდრე *Integer− > Integer*, მეორეც, *Haskell* ენაზე პროგრამირებისას ფუნქციის ტიპის ცხადად მითითება ითვლება „კარგ ტონად“, რადგანაც ტიპის გამოცხადება ემსახურება რაღაც აზრით ფუნქციის დოკუმენტაციას და ეხმარება პროგრამირების შეცდომების გამოვლენას.

მომხმარებლია მიერ განსაზღვრული ფუნქციებისა და ცვლადების სახელები უნდა იწყებოდეს პატარა (ქვედა რეგისტრის) ლათინური ასოებით. სახელებში სხვა სიმბოლოები შეიძლება იყოს დიდი ან პატარა ლათინური ასოები, ციფრები ან სიმბოლო _ და ′ (ხაზგასმა და აპოსტროფი). ასე, რომ ქვემოთ ჩამოთვლილია ცვლადების სწორი სახელები:

```
var
var1
variableName
variable_name
var'
```

9 პირობითი გამოსახულებები

ენა *Haskell* −ში ფუნქციის განსაზღვრებისას შესაძლებელია გამოყენებული იყოს პირობითი გამოსახულებები. ჩავწეროთ ფუნქცია *signum*, რომელიც თვლის გადაცემული არგუმენტის ნიშანს:

```
signum :: Integer -> Integer
signum x = if x > 0 then 1
           else if x < 0 then -1
                         else 0
```

პირობითი გამოსახულება ჩაიწერება ასე:

if პირობა *then* გამოსახულება *else* გამოსახულება.

ყურადღება გავამახვილოთ იმაზე, რომ თუმცა გარეგნულად ეს გამოსახულება ჰგავს *C* ან *Pascal* ენების ოპერატორს, ენა *Haskell* −ში აუცილებლად უნდა იყოს როგორც *then*, ასევე *else* ნაწილები. გამოსახულებები პირობითი ოპერატორის *then* და *else* ნაწილებში აუცილებლად ერთიდაიგივე ტიპის უნდა იყოს. პირობა პირობითი ოპერატორის განსაზღვრებაში წარმოადგენს ნებისმიერ *Bool* −ის ტიპის გამოსახულებას. ასეთი გამოსახულებების მაგალითად გამოდგება შედარება. შედარებისას შეიძლება გამოვიყენოთ შემდეგი ოპერატოები:

- <, >, <=, >= — ამ ოპერატორებს იგივე აზრი აქვთ, რაც ენა *C* -ში (ნაკლებია, მეტია, ნაკლებია და ტოლია, მეტია და ტოლია).
- == — ტოლობაზე შედარების ოპერატორი.
- /= — უტოლობაზე შედარების ოპერატორი.

Bool –ის ტიპის გამოსახულებები შეიძლება გავაერთიანოთ ლოგიკური ფუნქციებით && და || („და“ და „ან“), და უარყოფის ფუნქციით *not*. დასაშვები პირობების მაგალითებია:

```
x >= 0 && x <= 10
x > 3 && x /= 10
(x > 10 || x < -10) && not (x == y)
```

რა თქმა უნდა, შესაძლებელია განვსაზღვროთ ფუნქციები, რომლებიც აბრუნებენ *Bool* –ის ტიპის მნიშვნელობებს და გამოვიყენოთ ისინი პირობებად. მაგალითად, შესაძლებელია განვსაზღვროთ ფუნქცია *isPositive*, რომელიც აბრუნებს *True*–ს, თუ მისი არგუმენტი არის არაუარყოფითი და *False* წინააღმდეგ შემთხვევაში:

```
isPositive :: Integer -> Bool
isPositive x = if x > 0 then True else False
```

ეხლა ფუნქია *signum* შეიძლება განისაზღვროს შემდეგნაირად:

```
signum :: Integer -> Integer
signum x = if isPositive x then 1
           else if x < 0 then -1
                         else 0
```

შევნიშნოთ, რომ ფუნქცია *isPositive* შეიძლება განისაზღვროს უფრო მარტივად:

```
isPositive x = x > 0
```

10 მრავალცვლადიანი ფუნქციები და ფუნქციების განსაზღვრის რიგი

აქამდე ჩვენ განვსაზღვრავდით ერთარგუმენტიან ფუნქციებს. რა თქმა უნდა, ასკელლ ენაში შესაძლებელია განვსაზღვროთ ფუნქ-

ციები, რომლებიც იღებენ ნებისმიერი რიცხვის არგუმენტებს. ფუნქცია *add*–ის განსაზღვრას, რომელიც იღებს ორ მთელ რიცხვს და აბრუნებს მათ ჯამს, აქვს სახე:

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

ფუნქცია *add*–ის ტიპი გამოიყურება „უცნაურად“. *Haskell* ენაში ითვლება, რომ ოპერაცია $\rightarrow$ ასოციატიურია მარჯვნიდან. ასე, რომ *add* ფუნქციის ტიპი შეიძლება ასე წავიკითხოთ $Integer \rightarrow (Integer \rightarrow Integer)$, ანუ კარირების წესების მიხედვით, *add* ფუნქციის გამოყენების შედეგი ერთ არგუმენტთან იქნება ფუნქცია, რომელიც მიიღებს *Integer* ტიპის ერთ პარამეტრს. საზოგადოდ, ფუნქციის ტიპი, რომელიც ღებულობს n არგუმენტს, რომლებიც ეკუთვნის $t1, t2, ..., tn$, ტიპებს და აბრუნებს a ტიპის შედეგს, ჩაიწერება სახით $t1 \rightarrow t2 \rightarrow ... \rightarrow tn \rightarrow a$.

საჭიროა გავაკეთოთ კიდევ ერთი შენიშვნა ფუნქციების განსაზღვრის რიგის შესახებ. წინა პარაგრაფში ჩვენ განვსაზღვრეთ ორი ფუნქცია – *signum* და *isPositive*, ამათგან ერთი მათგანი იყენებდა თავის განსაზღვრებაში მეორეს. ისმის კითხვა, რომელი მათგანი უნდა განისაზღვროს ადრე? თითქოსდა *isPositive*–ის განსაზღვრება უნდა უსწრებდეს *signum*–ის განსაზღვრებას, მაგრამ *Haskell* ენაში ფუნქციების განსაზღვრის რიგს არ აქვს მნიშვნელობა! ასე, რომ ფუნქცია *isPositive* შეიძლება განისაზღვროს როგორც *signum* ფუნქციის განსაზღვრამდე, ისე მის შემდეგ.

11 ლაბორატორიული დავალებები

მოიყვანეთ არატრივიალური გამოსახულებების მაგალითები, რომლებიც ეკუთვნის ტიპებს:

1. $((Char, Integer), String, [Double])$
2. $[(Double, Bool, (String, Integer))]$
3. $([Integer], [Double], [(Bool, Char)])$
4. $[[[(Integer, Bool)]]]$
5. $(((Char, Char), Char), [String])$

6. $(([Double], [Bool]), [Integer])$
7. $[Integer, (Integer, [Bool])]$
8. $(Bool, ([Bool], [Integer]))$
9. $[([Bool], [Double])]$
10. $[([Integer], [Char])]$

ამ მაგალითის მოთხოვნა გამოსახულებების არატრივიალურობის შესახებ ნიშნავს, რომ გამოსახულებებში მონაწილე სიები უნდა შეიცავდნენ ერთ ელემენტზე მეტს.

2. განსაზღვრეთ შემდეგი ფუნქციები:

1) ფუნქცია $max3$, რომელიც სამი მთელი რიცხვიდან აბრუნებს მათ შორის უდიდესს.

2) ფუნქცია $min3$, რომელიც სამი მთელი რიცხვიდან აბრუნებს მათ შორის უმცირესს.

3) ფუნქცია $sort2$, რომელიც ორი მთელი რიცხვიდან აბრუნებს წყვილს, რომელშიც პირველ ადგილას დგას ამ ორი რიცხვიდან უმცირესი, მეორეზე კი – უდიდესი.

4) ფუნქცია $bothTrue :: Bool -> Bool -> Bool$, რომელიც აბრუნებს $True$–ს, მაშინ და მხოლოდ მაშინ, როცა ორივე არგუმენტი არის $True$. ფუნქციის განსაზღვრისათვის არ გამოიყენოთ ლოგიკური ოპერაციები (&&, || და ა.შ.)

5) ფუნქცია $solve2 :: Double -> Double -> (Bool, Double)$, რომელიც ორი რიცხვის მიხედვით, რომლებიც წარმოადგენენ $ax + b = 0$ წრფივი განტოლების კოეფიციენტებს, აბრუნებს წყვილს, რომლის პირველი ელემენტი არის $True$, თუ არსებობს ამონახსნი და $False$ – წინააღმდეგ შემთხვევაში; წყვილის მეორე ელემენტი კი არის ან ფესვის მნიშვნელობა, ან 0.0.

6) ფუნქცია $isParallel$, რომელიც აბრუნებს $True$–ს, თუ ორი მონაკვეთი, რომლებიც წარმოადგენენ ფუნქციის არგუმენტებს, არის პარალელური (ან დევს ერთ წრფეზე). მაგალითად, მნიშვნელობა გამოსახულებისა $isParallel\ (1,1)(2,2)(2,0)(4,2)$ არის $True$, ვინაიდან მონაკვეთები $(1,1) - (2,2)$ და $(2,0) - (4,2)$ პარალელურია.

7) ფუნქცია $isIncluded$, რომლის არგუმენტებია სიბრტყეზე ორი წრეწირის პარამეტრები (ცენტრის კოორდინატები და რადიუსები); ფუნქცია აბრუნებს $True$–ს, თუ მეორე წრეწირი მთლიანად თავსდება პირველის შიგნით.

8) ფუნქცია ისლეცტანგულარ, რომელიც პარამეტრად ღებულობს სიბრტყეზე სამი წერტილის კოორდინატებს და აბრუნებს *True*–ს, თუ მათ მიერ შედგენილი სამკუთხედი არის მართკუთხა სამკუთხედი.

12 საკონტროლო შეკითხვები

1. რით განსხვავდება ინტერპრეტატორის ბრძანებები *Haskell*–ის გამოსახულებებისგან?
2. ენა *Haskell*–ის ძირითადი ტიპები.
3. კორტეჟებთან მუშაობის ფუნქციები.
4. სიებთან მუშაობის ფუნქციები.
5. ცვლადებისა და ფუნქციების დასაშვები სახელები.
6. ინტერპრეტატორის ბრძანებები პროგრამების ფაილებთან სამუშაოდ.
7. პირობითი გამოსახულებები ენა *Haskell*–ში.
8. ფუნციების განსაზვრება ენა *Haskell*–ში

ლექცია № 2

1 სამუშაოს მიზანი

რეკურსიული ფუნქციების განსაზღვრა. „ნიმუშთან შედარების“ მექანიზმის გაცნობა. სიებთან მუშაობის უნარ–ჩვევების შეძენა.

2 კომენტარები

ცხადია, რომ აუცილებელია პროგრამაში კომენტარების არსებობა. *Haskell* ენაში, ისევე, როგორც $C++$–ში, არსებობს ორი ტიპის კომენტარი: სტრიქონული და ბლოკის. სტრიქონული კომენტარი იწყება სიმბოლოებით $--$ და გრძელდება სტრიქონის ბოლომდე. ანალოგიურად, $C++$–ში სტრიქონული კომენტარი იწყება // სიმბოლოებით. ბლოკური კომენტარი იწყება სიმბოლოებით {– და გრძელდება სიმბოლოებამდე –}. ანალოგიურად, $C++$–ში კომენტარები შემოსაზღვრულია სიმბოლოებით /* და */. იგულისხმება, რომ კომენტარი იგნორირდება ასკელლ ინტერპრეტატორის მიერ. მაგალითად,

```
fx = x -- ეს არის კომენტარი
gxy = {- ესეც კომენტარია, მხოლოდ გრძელი კომენტარია - }
x + y
```

3 რეკურსია

პროგრამირების იმპერატიულ ენებში ძირითად კონსტრუქციას წარმოადგენს ციკლი. ენა ასკელლ–ში ციკლების ნაცვლად გამოიყენება რეკურსია. ფუნქციას ეწოდება რეკურსიული, თუ ის იძახებს თავის თავს (ანუ, უფრო ზუსტად, განსაზღვრულია თავისივე ტერმინებში).

რეკურსიული ფუნქციები არსებობს იმპერატიულ ენებშიც, მაგრამ ასე ფართოდ არ გამოიყენება. ერთ–ერთი რეკურსიული ფუნქციას წარმოადგენს ფაქტორიალი:

```
factorial :: Integer -> Integer
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

(შევნიშნოთ, რომ ჩვენ ვწერთ $factorial\ \ (n-1)$ და არა $factorial\ \ n1$ ოპერაციების პრიორიტეტების შესაბამისად.) რეკურსიის გამოყენებამ შეიძლება გამოიწვიოს სირთულეები. რეკურსიის კონცეფცია შეგვახსენებს ინდუქციური დამტკიცების ხერხს, რომელიც მათემატიკაში გამოიყენება. ფაქტორიალის განმარტებაში გამოვყავით „ინდუქციის ბაზა“ (შემთხვევა $n == 0$) და „ინდუქციის ბიჯი“ (გადასვლა $factorial\ \ n$-დან $factorial\ \ (n-1)$ -ზე). ამ კომპონენტების გამოყოფა მნიშვნელოვანი ნაბიჯია რეკურსიული ფუნქციების განსაზღვრისას.

4 ამორჩევის ოპერაცია და გამართვის წესები

ჩვენ უკვე განვიხილეთ $n == 0$ პირობითი ოპერატორი. ეხლა განვიხილოთ ამორჩევის ოპერატორი *case*, რომელიც არის ენა C-ის სწიტცჰ კონსტრუქციის ანალოგი. დავუშვათ, საჭიროა ფუნქციის განსაზღვრა, რომელიც აბრუნებს 1-ს, თუ მას გადავცემთ არგუმენტს 0-ს; 5-ს, თუ არგუმენტი ტოლია 1; 2-ს, თუ არგუმენტი ტოლია 2-ის და -1-ს, ყველა სხვა შემთხვევაში. რა თქმა უნდა, ამ ფუნქციის ჩაწერა ოპერატორი if-ის საშუალებითაც არის შესაძლებელი, თუმცა განსაზღვრება იქნება გრძელი და ბუნდოვანი. ასეთ შემთხვევებში გამოიყენება *case*:

```
f x = case x of
        0 -> 1
        1 -> 5
        2 -> 2
        _ -> -1
```

მოყვანილი მაგალითიდან ცხადია *case* ოპერატორის სინტაქსი. შევნიშნოთ, რომ სიმბოლო _ არის ენა C-ის დეფაულტ კონსტრუქციის ანალოგი. თუმცა, უნდა განისაზღვროს წესები, როგორ არკვევს ენა *Haskell*-ის ინტერპრეტატორი, თუ სად იწყება ერთი შემთხვევა და სად - მეორე.

ენა *Haskell*-ში არსებობს ტექსტის სტრუქტურირების ორგანზომილებიანი სისტემა (ანალოგური სისტემა გამოიყენება ფართოდ გავრცელებულ ენა *Python*-შიც). ეს სისტემა იძლევა საშუალებას არ გამოვიყენოთ სპეციალური სიმბოლოები ოპერატორების გაერთიანებისთვის, მაგალითად, ისეთები, როგორიცაა {,} და : ენა *C*-ში. სინამდვილეში, ენა *Haskell*-შიც შეიძლება ამ სიმბოლოების გამოყენება იგივე აზრით. ზემოთ მოყვანილი განსაზღვრება შეიძლება ასეც ჩაიწეროს:

```
f x = case x of
         { 0 -> 1; 1 -> 5;
   2 -> 2;
        _ -> -1 }
```

ასეთი ჩაწერა ცხადად განსაზღვრავს ოპერატორების დაჯგუფებას, თუმცა შეიძლება მათ გარეშეც. ზოგადი წესები ასეთია: გასაღები სიტყვების *where*, *let*, *do* და *of* შემდეგ ინტერპრეტატორი სვამს გახსნილ ფრჩხილს ({) და იმახსოვრებს სვეტს, რომელშიც შემდეგი ბრძანება არის ჩაწერილი. შემდგომში, ყოველი ახალი სტრიქონის *win*, რომელიც გასწორებულია დამახსოვრებული სიდიდით, ჩაისმება გამყოფი სიმბოლო ;. თუ შემდეგი სტრიქონი ნაკლებად არის გასწორებული (ანუ მისი პირველი სიმბოლო არის დამახსოვრებული პოზიციის მარცხნივ), მაშინ ჩაისმება დახურული ფრჩხილი (}). თუ ამ წესს გამოვიყენებთ ზემოთ აღწერილ f ფუნქციასთან, მივიღებთ, რომ ინტერპრეტატორი აღიქვამს მას შემდეგნაირად:

```
f x = case x of{
        ;0 -> 1
        ;1 -> 5
        ;2 -> 2
        ;_ -> -1
}
```

ნებისმიერ შემთხვევაში შეიძლება ცხადად მიუთითოთ სიმბოლოები {,} და ;, თუმცა ამ დროს ტექსტი ნაკლებად „წაკითხვადია“ და მათ გამოყენებას ლაბორატორიულ მეცადინეობებზე არ გირჩევთ.

კიდევ ერთი შენიშვნა. ვინაიდან *Haskell* ენის პროგრამისთვის ცარიელ უჯრედებს აქვთ მნიშვნელობა, ამიტომ საჭიროა ყურადღება ტაბულაციის სიმბოლოსთანაც. ინტერპრეტატორი თვლის, რომ ტაბულაციის სიმბოლო ტოლია 8 ცარიელი უჯრედისა (სიმბოლოსი). თუმცა, ზოგიერთი ტექსტური რედაქტორი იძლევა ტაბულაციის

სიმბოლოს განსაზღვრის საშუალებას და ხდის მას სხვა რიცხვის ტოლად (მაგალითად, *VisualStudio*- რედაქტორში გაჩუმების პრინციპით ტაბულაცია არის 4 ცარიელი უჯრედი (სიმბოლო)). ამან შეიძლება გამოიწვიოთ შეცდომები, ამიტომაც სჯობს ენა *Haskell*-ზე პროგრამირებისას არ გამოიყენოთ ტაბულაციის სიმბოლოები.

5 უბან-უბან მოცემული ფუნქციები

ფუნქცია შეიძლება განისაზღვროს უბან-უბან. ეს ნიშნავს, რომ ფუნქციის ერთი ვერსია შეიძლება განისაზღვროს განსაზღვრული პარამეტრებისთვის, მეორე ვერსია - სხვა პარამეტრებისთვის. ასე, რომ წინა პარაგრაფში მოყვანილი f ფუნქცია შეიძლება განისაზღვროს შემდეგნაირადაც:

```
f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1
```

ამ შემთხვევაში მნიშვნელობა აქვს ფუნქციის განსაზღვრის რიგს. თუ ჩვენ თავდაპირველად ჩავწერთ განსაზღვრებას `f _ = -1`, მაშინ f ფუნქცია დააბრუნებს მნიშვნელობას 1-ს ნებისმიერი არგუმენტისთვის. თუ ამ სტრიქონს საერთოდ არ მივუთითებთ, მივიღებთ შეცდომას არგუმენტისთვის, რომელიც არ არის ტოლი 0-ის, 1-ის ან 2-ის. ფუნქციის განსაზღვრის ასეთი სშუალება ხშირად გამოიყენება ასკელლ-ში. ის ნაწილობრივ იძლევა საშუალებას არ გამოვიყენოთ ოპერატორები *if* და *case*. ასე, რომ ფუნქცია ფაქტორიალი შეიძლება განისაზღვროს ასეთი სტილითაც:

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

6 ნიმუშთან შედარება

ისევე, როგორც შესაძლებელია რეკურსიული ფუნქციები განისაზღვროს მთელ რიცხვებზე, ასევე შესაძლებელია განისაზღვროს სიებზეც.

ამ შემთხვევაში, „რეკურსიის ბაზა“ იქნება ცარიელი სია ([]). განვსაზღვროთ სიის სიგრძის გამოთვლის ფუნქცია (ვინაიდან სახელი *length* უკვე დაკავებულია სტანდარტულ ბიბლიოთეკაში, ფუნქციას დავარქვათ *len*):

```
len [] = 0
len s  = 1 + len (tail s)
```

გავიხსენოთ, რომ სია, რომლის პირველი ელემენტი (სიის თავი) არის x, ხოლო დანარჩენ ელემენტებს (სიის კუდი) წარმოადგენს xs, ჩაიწერება როგორც $x : xs$. ამგვარი კონსტრუქცია შესაძლოა გამოყენებული იყოს ფუნქციის აღწერისას:

```
len []       = 0
len (x:xs)   = 1 + len xs
```

მოვიყვანოთ კიდევ ერთი მაგალითი. ფუნქცია, რომელიც შესასვლელზე ღებულობს რიცხვების წყვილს და აბრუნებს მათ ჯამს, შეიძლება ასე განისაზღვროს:

```
sum_pair p = fst p + snd p
```

თუმცა, როგორ მოვიქცეთ, თუ საჭიროა განისაზღვროს ფუნქცია, რომელიც ღებულობს რიცხვების სამეულს და აბრუნებს მათ ჯამს? ჩვენ არ გვაქვს *fst* და *snd* ფუნქციების მსგავსი ფუნქციები სამეულებიდან ელემენტების ამოსაღებად. აღმოჩნდა, რომ ასეთი ფუნქციები შეიძლება ჩაიწეროს შემდეგნაირად:

```
sum_pair (x,y) = x + y
sum_triple (x,y,z) = x + y + z
```

ასეთ ხერხს უწოდებენ ნიმუშთან შედარებას (პატტერნ მატჩინგ). იგი წარმოადგენს ენის ძალზე ძლიერ კონსტრუქციას. „ნიმუშები“ ჩაიწერება ფუნქციის არგუმენტებად და „შედარდება“ ფუნქციაზე გადაცემულ ფაქტიურ პარამეტრებს.

როდესაც ხდება ნიმუშთან შედარება, მასში მონაწილე ცვლადები ღებულობენ შესაბამის მნიშვნელობებს. თუ ეს მნიშვნელობები ფუნქციის გამოთვლისთვის არ არის საჭირო (მაგალითად, ფუნქცია `my_tail` შემდეგ მაგალითში), მაშინ ზედმეტი სახელების შემოტანის ნაცვლად შეიძლება გამოყენებული იყოს სიმბოლო `_`. იგი აღნიშნავს ნიმუშს, რომელსაც შეესაბამება ნებისმიერი მნიშვნელობა, თვითონ ეს მნიშვნელობა კი არცერთ ცვლადს არ უკავშირდება. შემდეგი მაგალითები უჩვენებენ ნიმუშთან შედარების გამოყენების სხვადასხვა ვარიანტებს:

```
--  funqcia, romelic Sekrebs siis pirvel or  \geoac wevrs
f1 (x:y:xs) = x + y
-- funqciis gansazRvreba, romelic head-is analogia
my_head (x:xs) = x

-- funqciis gansazRvreba, romelic tail-is analogiuria.
-- Cven viyenebT simbolo _-s imitom, rom siis
-- pirveli elementis mniSvneloba ar aris saWiro
my_tail (_:xs) = xs
-- funqcia, romelic iRebs pirvel wevrs sameulidan
fst3 (x,_,_) = x
```

ნიმუშთან შედარება შეიძლება გამოყენებული იყოს ოპერატორში *case*:

```
-- siis sigrZis gansazRvris kidev erTi funqcia
my_length s = case s of
[] -> 0
(_:xs) -> 1 + my_length xs
```

შეიძლება საკმაოდ რთული ნიმუშების განსაზღვრა. განვსაზღვროთ ფუნქცია, რომელიც იღებს რიცხვთა წყვილებს და აბრუნებს მათი სხვაობების ჯამს (ანუ $f[(x1,y1),(x2,y2),...,(xn,yn)] = (x1-y1)+(x2-y2)+...+(xn-yn)$):

```
f []          = 0
f ((x,y):xs) = (x - y) + f xs
```

7 სიების აგება

იმ ფუნქციების განსაზღვრისთვის, რომლებიც აბრუნებენ სიებს, ხშირად გამოიყენება ოპერატორი :. მაგალითად, ფუნქცია, რომელიც ღებულობს რიცხვების სიას და აბრუნებს ამ რიცხვების კვადრატებს, შეიძლება ასე განისაზღვროს:

```
square []     = []
square (x:xs) = x*x : square xs
```

8 ზოგიერთი სასარგებლო ფუნქციები

ლაბორატორიული ფუნქციების შესრულებისას შესაძლოა დაგჭირდეთ *Haskell* ენის შემდეგი სტანდარტული ფუნქციები:

- `even` --- აბრუნებს `True`-ს ლუწი არგუმენტისთვის და `False`-ს კენტებისთვის.
- `odd` --- წინას ანალოგური, მხოლოდ არგუმენს ამოწმებს კენტობაზე.

9 დავალებები

1. განსაზღვრეთ ფუნქცია, რომელიც შესასვლელზე ღებულობს მთელ რიცხვს n-ს და აბრუნებს სიას, რომელიც შედგება n ელემენტისგან, დალაგებულს ზრდადობით.

1. ნატურალური რიცხვების სია.
2. კენტი ნატურალური რიცხვების სია.
3. ლუწი ნატურალური რიცხვების სია.
4. ნატურალური რიცხვის კვადრატების სია.
5. ფაქტორიალების სია.
6. 2-ის ხარისხების სია.
7. სამკუთხედურის რიცხვების სია.

   (განმარტება: n-ური სამკუთხედური რიცხვი tn ტოლია ერთნაირი მონეტების რაოდენობისა, რომლებიდანაც შეიძლება შედგეს ტოლგვერდა სამკუთხედი, რომლის თითოეულ გვერდზეც დაიდება n მონეტა. ცხადია, რომ $t_1 = 1$ და $t_n = n + tn - 1$).
8. პირამიდალური რიცხვების სია.

   (განმარტება: n-ური პირამიდალური რიცხვი პნ ტოლია ერთნაირი ბურთების რაოდენობისა, რომლებიდანაც შეიძლება აიგოს სწორი პირამიდა სამკუთხედის ძირით, რომლის თითოეულ გვერდზეც დაიდება n ბურთი. ცხადია, რომ $p_1 = 1$ და $= tn + pn - 1$).

2. განსაზღვრეთ შემდეგი ფუნქციები:

1. ფუნქცია, რომელიც შესასვლელზე ღებულობს ნამდვილ რიცხვებს და ითვლის მათ საშუალო არითმეტიკულს. შეეცადეთ, რომ ფუნქციამ მხოლოდ ერთხელ გადახედოს სიას.

2. ფუნქცია გამოყოფს მოცემული სიის n წევრს.

3. ორი სიის ელემენტების ალამვის ფუნქცია. აბრუნებს სიას, რომელიც შედგება პარამეტრი სიების ელემენტების ჯამისგან. გაითვალისწინეთ, რომ გადაცემული სიები შეიძლება იყოს სხვადასხვა სიგრძის.

4. ფუნქცია, რომელიც აადგილებს მოცემულ სიაში მეზობელ ლუწ და კენტ ელემენტებს.

5. ფუნქცია *twopow n*, რომელიც ითვლის 2^n შემდეგი მოსაზრებებიდან გამომდინარე. დავუშვათ საჭიროა ავიყვანოთ 2 ხარისხში. თუ n ლუწია, ანუ $n = 2k$, მაშინ $2^n = 2^{2k} = (2^k)^2$. თუ n კენტია, ანუ $n = 2k + 1$, მაშინ $2^n = 2^{2k+1} = 2(2^k)^2$. ფუნქციამ *twopow* არ უნდა გამოიყენოს ოპერატორი ^ ან სხვა ფუნქცია სტანდარტული ბიბლიოთეკიდან, რომელიც ითვლის ხარისხს. ფუნქციის რეკურსიული გამოძახებები უნდა იყოს *logn*-ის პროპორციული.

6. ფუნქცია *removeOdd*, რომლიც მოცემული მთელი რიცხვების სიიდან ამოშლის ყველა კენტ რიცხვს. მაგალითად, *removeOdd* $[1, 4, 5, 6, 10]$ უნდა დააბრუნოს $[4, 10]$.

7. ფუნქცია *removeEmpty*, რომელიც ამოაგდებს ცარიელ სტრიქონებს სტრიქონების მოცემული სიიდან. მაგალითად, *removeEmpty* [””,”*Hello*”,””,””,”*World*!”] უნდა დააბრუნოს [”*Hello*”,”*World*!”].

8. ფუნქცია $countTrue :: [Bool] -> Integer$, რომელიც აბრუნებს სიის იმ ელემენტების რაოდენობას, რომლებიც არის $True$-ს ტოლი.

9. ფუნქცია *makePositive*, რომელიც უცვლის ნიშანს რიცხვების სიის ყველა უარყოფით ელემენტს. მაგალითად, *makePositive* $[-1, 0, 5, -10, -20]$ გვაძლევს $[1, 0, 5, 10, 20]$.

10. ფუნქცია $delete :: Char -> String -> String$, რომელიც იღებს შესასვლელზე სტრიქონს და სიმბოლოს და აბრუნებს სტრიქონს,

რომლიდანაც ამოშლილია მოცემული სიმბოლო. მაგალითად, *delete* $'l'$ "*Hello world*!" უნდა დააბრუნოს "*Heo word*!".

11. ფუნქცია *substitute* :: *Char*− > *Char*− > *String*− > *String*, რომელიც ცვლის მოცემულ სიმბოლოს მეორე სომბოლოთი. მაგალითად, *substitute* $'e'$ $'i'$ "*eigenvalue*" აბრუნებს "*iiginvalui*".

10 საკონტროლო შეკითხვები

1. გასწორების წესები.
2. შედარება ნიმუშთან.
3. ამორჩევის ოპერაცია.
4. ფუნქციის ფრაგმენტული მოცემა.

ლექცია № 3

1 *let*-დაკავშირება

ფუნქციის აღწერისას ხშირად აუცილებელია გამოვიყენოთ დროებითი ცვლადები შუალედური მნიშვნელობების შესანახად. გავიხსენოთ, თუ როგორ ვითვლით $ax2 + bx + c = 0$ კვადრატული განტოლების ფესვებს $x_{1,2} = (-b \pm \sqrt{b^2 - 4ac})/2a$ შეიძლება ჩაიწეროს ფუნქცია კვადრატული განტოლების ფესვების გამოსათვლელად:

```
roots a b c =
    ((-b + sqrt (b*b - 4*a*c)) / (2*a),
     (-b - sqrt (b*b - 4*a*c)) / (2*a))
```

ასეთი სტილით პროგრამის დაწერა რამდენიმე პრობლემას შეიცავს. ჯერ ერთი, შეიძლება ადვილად დავუშვათ შეცდომა ერთიდაიგივე გამოსახულების ორჯერ დაწერისას. მეორე, ამ პროგრამის დაწერისას საჭიროა შევადაროთ ორი გამოსახულება, რომ დავრწმუნდეთ, რომ ერთიდაიგივეა. მესამე, პროგრამა უფრო გრძელი ხდება. და ბოლოს, ის ნაკლებად ეფექტურია, ვინაიდან კომპიუტერი ითვლის ერთიდაიგივე გამოსახულებას ორჯერ.

ამ პრობლების ასაცილებლად ენაში შემოტანილია ლოკალური ცვლადის ცნება. ფუნქცია შეიძლება ჩაიწეროას ასე:

```
roots a b c =
    let det = sqrt (b*b - 4*a*c)
    in ((-b + det / (2*a),
         (-b - det / (2*a))
```

ლოკალური ცვლადი *det* მიღწევადია მხოლოდ *roots* ფუნქციის განსაზღვრებაში.

შეიძლება რამდენიმე ლოკალური ფუნქცია განვსაზღვროთ:

```
roots a b c =
    let det = sqrt (b*b - 4*a*c)
```

```
        twice_a = 2*a
    in ((-b + det) / twice_a,
        (-b - det) / twice_a)
```

შევნიშნოთ, რომ კონსტრუქციაში *let...in...* გამოიყენება გასწორების წესები: ხარვეზისგან განსხავებული პირველი სიმბოლო, რომელიც მოსდევს სიტყვა *let*-ს, ასახელებს სვეტს, რომლის მიხედვითაც უნდა გასწორდეს შემდგომი განსაზღვრებები. თუ გამოვიყენებთ სიმბოლოებს ″,″ და ′;′, მაშინ გასწორების წესები აღარ იქნება აუცილებელი და მაშინ ფუნქცია *roots* შეიძლება ასე ჩაიწეროს:

```
roots a b c =
    let { let = sqrt (b*b - 4*a*c); twice_a = 2*a }
    in ((-b + det) / twice_a,
        (-b - det) / twice_a)
```

let...in... კომსტრუქციის გარდა, ზოგჯერ მოსახერხებელია *...where...* კონსტრუქციის გამოყენება. ამ კონსტრუქციისას ლოკალური ცვლადების გამოყენება მოსდევს ძირითად ფუნქციას:

```
roots a b c =
    ((-b + det) / twice_a,
     (-b - det) / twice_a)
    where det = sqrt (b*b - 4*a*c)
          twice_a = 2*a
```

შევნიშნოთ, რომ ნაცვლად ლოკალური ცვლადის შემოტანისა, შეიძლება გაგვესაზღვრა გლობალური ფუნქცია:

```
det a b c= sqrt (b*b - 4*a*c)

twice_a a = 2*a

roots a b c =
    ((-b + det a b c) / twice_a a,
     (-b - det a b c) / twice_a a)
```

თუმცა, ამ მიდგომის ნაკლი თვალსაჩინოა: შემოვიტანეთ გლობალურ სახელთა არეში ორი დამხმარე ფუნქცია (ეს კი იმას ნიშნავს, რომ ჩვენ აღარ გვექნება უფლება გამოვიყენოთ, მაგალითად, სახელი *det* სხვა ფუნქციისთვის), ასევე გამოსახულებების $\sqrt{b^2 - 4ac}$ და $2a$ გამოსათვლელად უნდა გადავცეთ ფუნქციას შესაბამისი

პარამეტრები, მაშინ, როცა ლოკალურ ცვლადებს თავისუფლად შეუძლიათ იმ ფუნქციის პარამეტრების გამოყენება, რომლისთვისაც ისინი არიან განსაზღვრული.

კონსტრუქციებში *let* და *where* შეიძლება განისაზღვროს არა მხოლოდ ცვლადები, არამედ ფუნქციებიც. განვიხილოთ, მაგალითად, ფუნქცია, რომელიც აბრუნებს მოცემული ნ რიცხვის მიხედვით ნატურალური რიცხვების სიას [1,2,...,ნ]. შემოვიტანოთ დამხმარე ფუნქცია ნუმსრომ, რომელიც მოცემული მ რიცხვის მიხედვით აბრუნებს სიას $[m, m+1, m+2, ..., n]$ და განვსაზღვროთ იგი, როგორც ლოკალური ფუნქცია:

```
numsTo n =
    let numsFrom m = if m == n then [m] else m:numsFrom (m + 1)
    in numsFrom 1
```

შევნიშნოთ, რომ ფუნქცია *numsFrom* თავის განსაზღრებაში იყენებს ცვლადს n, თუმცა მას იგი არ გადაეცემა პარამეტრად.

2 სიგნალიზაცია შეცდომების შესახებ

ჩვენს მიერ განსაზღვრული ფუნქციები შეიძლება არ იყოს გამოთვლადი არგუმენტების ზოგიერთი მნიშვნელობისთვის. გავიხსენოთ ფუნქცია ფაქტორიალის განმარტება:

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

ეს ფუნქცია სწორედ მუშაობს მანამ, სანამ არ შევეცდებით გამოვთვალოთ უარყოფითი რიცხვის ფაქტორიალი. რთული არ არის მივხვდეთ, რომ ამ დროს რეკურსია უსასრულოდ გრძელდება, იმიტომ რომ ბაზური შემთხვევა არასდროს მიიღწევა. ასეთი შეცდომების შესახებ შეტყობინების მარტივი საშუალებაა სტანდარტული ფუნქცია *error*-ის გამოყენება. ეს ფუნქცია არგუმენტად იღებს სტრიქონს, მისი გამოთვლა კი იწვევს პროგრამის დამთავრებას და ამ სტრიქონის გამოტანას ეკრანზე. ამრიგად, ფუნქცია ასე ჩავწეროთ:

```
factorial 0 = 1
factorial n = if n > 0 then
                  n * factorial (n - 1)
              else
                  error "factorial: negative argument"
```

3 დამცავი პირობები

ნიმუშთან შედარება იძლევა ფუნქციის განსაზღვრების დიდ შესაძლობლებებს, თუმცა მისი საშუალებით არა მხოლოდ ფუნქციისთვის გადასაცემი პარამეტრების სტრუქტურის გამოყოფა და ამ ელემენტების შედარება არის შესაძლებელი პარამეტრების კონსტანტურ მნიშვნელობებთან. თუმცა, ზოგჯერ ეს არ არის საკმარისი: აუცილებელია შესასვლელ პარამეტრებს დაედოთ უფრო რთული პირობები.

მაგალითად, ფუნქცია *factorial*-ის ზემოთ მოყვანილ მაგალითში ჩვენ გამოვიყენეთ ნიმუშთან შედარებისა და პირობითი ოპერატორის კომბინაცია. ნიმუშთან შედარება გამოიყურება უფრო ნათლად და ეკონომიურად. შეიძლება თუ არა მსგავსი სინტაქსი გამოვიყენოთ პირობისთვისაც? დიახ, თუ გამოვიყენებთ დამცველ პირობებს. მათი გამოყენებით ფაქტორიალის გამოთვლის ფუნქცია ასე ჩაიწერება:

```
factorial 0 = 1
factorial n | n < 0  = error "factorial: negative argument"
            | n >= 0 = n * factorial (n - 1)
```

მოყვანილი მაგალითიდან ჩანს გამოსახულების ჩაწერის სინტაქსი. შევნიშნოთ, რომ ბოლო პირობის ნაცვლად შეიძლება გამოყენებული იყოს გასაღები სიტყვა *otherwise* (ინგლისურად-წინააღმდეგ შემთხვევაში). მაგალითად, ფუნქცია, რომელიც განსაზღვრავს რიცხვის ნიშანს, ასე გამოიყურება:

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

ფუნქციია ასეთი სტილით განმარტება უფრო თვალსაჩინოა და *Haskell*-ის პროგრამებში ხშირად გამოიყენება (სესაბამისად, პირობითი ოპერატორი გამოიყენება იშვიათად). თვალსაჩინოებისთვის, განვსაზღვროთ ფუნქცია სიგნუმ პირობითი ოპერატორების გამოყენებით:

```
signum x = if x < 0 then
               -1
           else
               if x == 0 then
                   0
               else
                   -1
```

4 პოლიმორფული ტიპები

ენა *Haskell*-ში გამოიყენება ტიპების პოლომორფული სისტემა. არსებითად, ეს ნიშნავს, რომ ენაში არსებობს ტიპების ცვლადები. განვიხილოთ, ჩვენთვის უკვე ნაცნობი ფუნქცია *tail*,რომელიც გვიბრუნებს სიის პირველ ელემენტს. როგორია ან ფუნქციის ტიპი? ის ერთნაირად გამოიყენება როგორც მთელი რიცხვების სიისთვის, ასევე სიმბოლოებისა და სტრიქონების სიებისთვისაც:

```
Prelude>tail [1,2,3]
[2,3]
Prelude>tail ['a','b','c']
['b','c']
Prelude>tail ["list", "of", "lists"]
["of", "lists"]
```

ფუნქცია *tail*-ს აქვს პოლიმორფული ტიპი: $[a]->[a]$. ეს ნიშნავს, რომ იგი არგუმენტად იღებს ნებისმიერ სიას და აბრუნებს იგივე ტიპის სიას. აქ a აღნიშნავს ტიპის ცვლადს, ანუ იგულისხმება, რომ მის მაგივრად შეიძლება ჩაისვას ნებისმიერი კონკრეტული ტიპი. ამრიგად, ჩანაწერი $[a]->[a]$ იძლევა ტიპების მთელ კლასს, რომლის წარმომადგენლებიც არის, მაგალითად, $[Integer]->[Integer], [Char]->[Char], [[Char]]->[[Char]]$ და ა.შ.

ანალოგიურად, ფუნქციას *tail*, რომელიც აბრუნებს სიის პირველ ელემენტს, აქვს ტიპი $[a]->a$. ტიპების ამ ოჯახის წარმომადგენლები არიან $[Integer]->Integer, [Char]->Char$ და ა.შ.

სიებთან, წყვილებთან და კორტეჟებთან მომუშავე მრავალ ფუნქციას პოლიმორფული ტიპი. ასე, ფუნქციას fst-ს აქვს ტიპი $(a,b)->a$ (შევნიშნოთ, რომ ამ ტიპის განსაზღვრაში გამოპყენებულია ტიპის ორი ცვლადი).

5 მომხმარებლის ტიპები

პროგრამისტს აქვს შესაძლებლობა სტანდარტული ტიპების გვერდით განსაზღვროს მონაცემების თავისი საკუთარი, სპეციფიური ტიპები. ამისთვის საჭიროა გამოყენებული იყოს გასაღები სიტყვა *data*.

5.1 წყვილები

მაგალითისთვის განვიხილოთ წყვილების განმარტება, რომელიც ზალზე წააგავს სტანდარტულს: დ

```
data Pair a b = Pair a b
```

დეტალურად განვიხილოთ ეს კოდი. გასაღები სიტყვა *data* გვიჩვენებს, რომ ჩვენ ვგეგმავთ მონაცემების ტიპის განსაზღვრას. მას მოსდევს ამ ტიპის დასახელება, ჩვენ შემთხვევაში *Pair* (გავიხსენოთ, რომ ტიპების დასახელება ყოველთვის დიდი, მთავრული ასოთი იწყება). *a* და *b* წარმოადგენენ ტიპებბის ცვლადებს, რომლებიც ტიპების პარამეტრებს აღნიშნავენ. ამრიგად, ჩვენ აღვწეროთ მონაცემთა სტრუქტურა, რომელიც პარამეტრიზებულია ორი ტიპით *a* და *b*. (ეს ძალზე წააგავს ენა $C++$-ის შაბლონებს). ტოლობის ნიშნის შემდეგ ჩვენ მივუთითებთ ამ ტიპის მონაცემების კონსტრუქტორს. ამ შემთხვევაში ჩვენ გვაქვს ერთადერთი კონსტრუქტორი *Pair*. (არ არის აუცილებელი მონაცემების კონსტუქტორის სახელი ემთხვეოდეს ტიპის სახელს, თუმცა ჩვენს მაგალითში ეს ბუნებრივია). კონსტრუქტორის სახელის შემდეგ ჩვენ ისევ ვწერთ *a*. ეს აღნიშნავს, რომ წყვილის კონსტრუირებისთვის საჭიროა ორი მნიშვნელობა: პირველი, რომელიც ეკუთვნის *a*-ს, მეორე - *b*-ს. ამ განმარტებას შემოყავს ფუნქცია $Pair :: a- > b- > Pairab$, რომელივ გამოიყენება *Pair* ტიპის წყვილების ასაგებად. ჩავტვირთოთ ეს კოდი ინტერპრეტატორში, ვნახოთ, როგორ იქმნება წყვილები:

```
Main>:t Pair
Pair :: a -> b -> Pair a b
Main>:t Pair 'a'
Pair 'a' :: a -> Pair Char a
Main>:t Pair 'a' "Hello"
Pair 'a' "Hello" :: Pair Char [Char]
```

მონაცემების კონსტრუქტორების შესაბამის ფუნქციებს ახასიათებთ ის თვისება, რომ შეიძლება მათი გამოყენება ნიმუშთან შედარებისას. ასე, რომ ფუნქციები, რომლებითაც ვღებულობთ ჩვენი წყვილისთვის პირველ და მეორე ელემენტს, შეიძლება განიმარტოს შემდეგნაირად:

```
pairFst (Pair x y) = x
pairSnd (Pair x y) = y
```

გამხილული მაგალითი იწვევს ასეთ შეკითხვას: რისთვისაა საჭირო საკუთარი *Pair* ტიპის განსაზღვრა, როცა არის წყვილების განსაზღვრის სტანდარტული საშუალება? ჯერ ერთი, *Pair* ტიპის გამოყენებით შეიძლება განისაზღვროს ფუნქციათა ნაკრები, რომელიც მხოლოდ ამ ტიპთან მუშაობს და გამოიყოს ეს ფუნქციები იმ ფუნქციებისგან, რომლებიც „ზოგადად“ მუშაობენ წყვილებთა. მეორეც, წყვილების მისაღებას შეიძლება დაედოს შეზღუდვები, რომელიც ვერ მოხერხდება სტანდარტული ტიპისთვის. მაგაკითად, წარმოიდგინეთ, რომ საჭიროა ტიპი, რომელიც შექმნის წყვილს ერთიდაიგივე ტიპის ელემენტებისგან. ეს ტიპი შეიძლება ასე განისაზღვროს:

```
data SamePair a = SamePair a a
```

აქ ტიპს აქვს ერთი პარამეტრი, თუმცა მონაცემების კონსტრუქტორი იღებს ერთი და იმავე ტიპის ორ პარამეტრს.

5.2 მრავლობითი კონსტრუქტორები

წინა მაგალითში ჩვენ განვიხილეთ მონაცემთა ტიპი ერთი კონსტრუქტორით. ასევე, შესაძლებელია და ხშირად ძალზე სასარგებლოა განისაზღვროს ტიპი რამდენიმე კონსტრუქტორით. კონსტრუქტორები ერთმანეთისგან გამოიყოფა სიმბოლოთი '|'. განვიხილოთ ტიპი *Color*, რომელიც წარმოადგენს ფერს შესაძლო მნიშვნელობებით *Red*, *Green* და *Blue*. იგი შეიძლება ასე განისაზღვროს:

```
data Color = Red | Green | Blue
```

აქ *Color* არის ტიპის დასახელება, ხოლო *Red*, *Green* და *Blue* მონაცემების კონსტრუქტორები. შევნიშნოთ, რომ ეს ტიპი არ ღებულობს პარამეტრებს. ასეთ ტიპებს უწოდებენ ჩამოთვლად ტიპებს და იგი შეესაბამება ენა $C++$-ის ენუმ კონსტრუქციას. ასეთი ტიპები ზალზე სასარგებლოა. მაგალითად, სტანდარტული ტიპი *Bool* ასე განისაზღვრება:

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

მრავლობითმა კონსტრუქტორებმა, ასევე, შეიძლება მიიღონ პარამეტრები. შევნიშნოთ, რომ ტიპი ჩოლორ საშუალებას გვაძლევს განვსაზღვროთ მხოლოდ სამი ფიქსირებული ფერი. გავაფართოვოთ იგი ისე, რომ მას შეეძლოს განსაზღვროს ნებისმიერი ფერი, მოცემული სამი მთელი რიცხვით, რომლებიც შეესაბამება

წითელი, მწვანე და ლურჯი ფერების დონეებს (სტანდარტული *rgb* წარმოდგენა):

```
data Color = Red | Green | Blue | RGB Int Int Int
```

აქ ტიპი *Color* სტანდარტული ფერების *Red*, *Green* და *Blue* გარდა (ეს სია, რა თქმა უნდა, შეიძლება გაფართოვდეს), შეიძლება განისაზღვროს *RGB* კონსტრუქტორის საშუალებით, რომელიც ღებულობს სამ მთელ რიცხვს *rgb* კომპონენტების განსასაზღვრად. მაშინ, მაგალითად, ფუნქცია, რომელიც გამოყოფს ფერის *Red* კომპონენტს, შეიძლება ასე ჩაიწეროს:

```
redComponent :: Color -> Int
redComponent Red = 255
redComponent (RGB r _ _) = r
redComponent _ = 0
```

მრავალკონსტრუქტორიანი ტიპები, ასევე, შეიძლება იყოს პოლიმორფული. განვიხილოთ შემდეგი პრობლემა. დავუშვათ, ფუნქცია აბრუნებს რაიმე მნიშვნელობას, ან იძლევა შეტყობინებას შეცდომის შესახებ. მაგალითად, წრფივი განტოლების ფესვების პოვნის ფუნქცია აბრუნებს ნაპოვნ ფესვებს; ფუნქცია, რომელიც ეძებს სიაში პირველ არაუარყოფით რიცხვს, აბრუნებს ამ რიცხვს და ა.შ. ამასთან, განტოლების ამოხსნა შეიძლება არ არსებობდეს, სიაში არ იყოს არაუარყოფითი რიცხვები და ა.შ. როგორ მოხდეს ამის შეტყობინება იმისთვის, ვინც ფუნქცია გამოიძახა? ზოგჯერ, შესაძლოა შეთანხმება, რომ რომელიმე სპეციალური მნიშვნელობა (მაგალითად, -1) ნიშნავდეს, რომ არ არის შედეგი (*C* ენის სტანდარტული ბიბლიოთეკის ბევრი ფუნქცია სწორედ ასე იქცევა). თუმცა, ეს ყოველთვის არ არის შესაძლებელი: წრფივი განტოლების ამოხსნის შემთხვევაში ასეთი მნიშვნელობა არ არსებობს. პრობლემა იხსნება სტანდარტული ტიპის *Maybe*-ის საშუალებით, რომელიც ასე განისაზღვრება:

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

ტიპი *Maybe* (ინგლისურად, „შესაძლებელია“) პარამეტრიზირებულია ტიპური ცვლადით *a* და წარმოდგება ორი კონსტრუქტორით: *Nothing* (ინგლისურად, „არაფერი“) და *Just* (ინგლისურად, „ზუსტად“) აზრობლივი შედეგისთვის. მაშინ ჩვენი ფუნქციები შეიძლება ასე ჩაიწეროს:

```
--funqcia abrunebs ax + b = 0 gantolebis fesvs
solve :: Double -> Double -> Maybe Double
solve 0 b = Nothing
solve a b = Just (-b / a)

--funqcia abrunebs siis pirvel arauaryofiT elements
findPositive :: [Integer] -> Maybe Integer
findPositive [] = Nothing
findPositive (x:xs) | x > 0     = Just x
                    | otherwise = findPositive xs
```

Maybe ტიპის გამოყენებას აქვს მთელი რიგი უპირატესობები. მისი გამოყენებით ცხადად მივუთითებთ, რომ ფუნქციამ შეიძლება დააბრუნოს „არანაირი შედეგი“. ამასთან, ფუნქციის დასაბრუნებელი მნიშვნელობის დამუშავებისას საჭიროა ცხადად იყოს შედარება ნიმუშთან, და ტუ ჩვენ დაგვავიწყდება *Nothing* შემთხვევის დამუშავება, კომპილერმა შეიძლება მოგვცეს გაფრთხილება.

5.3 ტიპების კლასები

ტიპების კლასები მნიშვნელოვნად ამარტივებს მომხმარებლის ტიპებთან მუშაობას. შემდგომში მათ დეტალურად შევისწავლით. ტიპების კლასი წარმოადგენს ტიპების განსაზღვრულ სიმრავლეს, რომლებსაც მთელი რიგი საერთო თვისებები აქვთ. მაგალითად, ტიპების კლასში *Eq* შედის ყველა ტიპი, რომლების ობიექტებისთვისაც განსაზღვრულია ტოლობის მიმართება, ანუ, თუ x და y ეკუთვნის ერთიდაიგივე ტიპს, რომელიც შედის ქ კლასში, მაშინ შეიძლება გამოვთვალოთ გამოსახულება $x == y$ და $x/ = y$. ყველა მარტივი ტიპი, ასევე სიები და კორტეჟები შედის ამ კლასში, თუმცა, მაგალითად, ფუნქციისთვის მიმართება ტოლობა არ არის განსაზღვრული და ამიტომ ტიპი ფუნქცია არ შედის *Eq* კლასში.

შემდეგი მნიშვნელოვანი კლასია *Show* კლასი. მასში შედის ყველა ტიპი, რომლის ობიექტები შეიძლება გარდაქმნილი იყოს სტრიქონში იმისათის, რომ მოხდეს მათი ეკრანზე ასახვა. მარტივი ტიპები, კორტეჟები და სიები შედის ამ კლასში, ამიტომ ინტერპრეტატორს შეუძლია დაბეჭდოს, მაგალითად, სტრიქონი. ფუნქცია არ შედის ამ კლასში. შეთანხმების პრინციპით მომხმარებლის ტიპები არ შედის არცერთ კლასში, ამიტომ მათი მნიშვნელობების შედარება არ შეიძლება და ვერც ინტერპრეტატორი დაბეჭ-

დავს. ეს, რა თქმა უნდა, მოუხერხებელია, ამიტომ ტიპების განსაზღვრისას შესაძლებელია ის მივაკუთვნოთ სასურველ კლასს. ამისთვის, ტიპის განსაზღვრის შემდეგ საიროა დაემატოს გასაღები სიტყვა *deriving* და ფრჩხილებში ჩამოვთვალოთ კლასები, რომლებსაც უნდა ეკუთვნოდეს ტიპი. მაგალითად:

```
-- tipi, romelic warmoadgens  dRis  periodia
data DayTime =  Morning
              | Afternoon
              | Evening
              | Night deriving (Eq, Show)
```

დავალებებში ტიპების აღწერისას მიაკუთვნეთ ისინი კლასებს *Eq* და *Show*. ამით გაიადვილებთ სამუშაოს.

6 დავალებები

1. თანამედროვე *web*–მაღაზიაში ხშირად იყიდება წიგნები, ვიდეოკასეტები და კომპაქტ–დისკები. ამ მაღაზიის მონაცემთა ბაზა თითოეული სახეობის საქონლისთვის უნდა შეიცავდესშემდეგ მახასიათებლებს:

- წიგნებისთვის: დასახელება და ავტორი
- ვიდეოკასეტებისთვის: დასახელება
- კომპაქტ–დისკებისთვის: დასახელება, შემსრულებელი და კომპოზიციების რაოდენობა.

1) შექმენით მონაცემთა ტიპი *Product*, რომელიც წარმოადგენს საქონლის ამ ტიპებს.

2) განსაზღვრეთ ფუნქცია *getTitle*, რომელიც დააბრუნებს საქონლის დასახელებას.

3) ამ ფუნქციის საფუძველზე განსაზღვრეთ ფუნქცია *getTitles*, რომელიც საქონლის სიის მიხედვით დააბრუნებს მათ დასახელებების სიას.

4) განსაზღვრეთ ფუნქცია *bookAuthors*, რომელიც საქონლის სიის მიხედვით დააბრუნებს წიგნების ავტორების სიას.

5) განსაზღვრეთ ფუნქცია

```
lookupTitle :: String -> [Product] -> Maybe Product
```

რომელიც დააბრუნებს საქონელს მოცემული დასახელებით (ყურადღება მიაქციეთ ფუნქციის შედეგის ტიპს).

6) განსაზღვრეთ ფუნქცია

```
lookupTitles :: [String] -> [Product] -> [Product]
```

ის პარამეტრად იღებს დასახელებების სიას და საქონლის სიას და თითოეული დასახელებისთვის იღებს მეორე სიიდან შესაბამის საქონელს. ის დასახელება, რომელსაც საქონელი არ შეესაბამება, იგნორირდება. ფუნქციის განსაზღვრისას სავალდებულოა გამოიყენოთ ფუნქცია *lookupTitle*.

2. განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს კარტს კარტის თამაშისას. თითოეულ კარტს აქვს ერთი მასტი ოთხი შესაძლოდან. კარტი შეიძლება იყოს დაბალი (ორიდან ათის ჩათვლით) ან სურათი (ვალეტი, დამა, ბაბუა, ტუზი). განსაზღვრეთ შემდეგი ფუნქციები:

1) ფუნქცია $isMinor$, რომელიც ამოწმებს, რომ მისი არგუმენტი არის დაბალი კარტი.

2) ფუნქცია $sameSuit$, რომელიც ამოწმებს, რომ მასთვის გადაცემული კარტები ერთი მასტისაა.

3) ფუნქცია $beats :: Card -> Card -> Bool$, რომელიც ამოწმებს, რომ კარტი, რომელიც პირველ არგუმენტად გადაეცემა, ჭრის კარტს, რომელიც მეორე არგუ,მენტად გადაეცემა.

4) ფუნქცია $beats2$ ანალოგიურია $beats$–ის, მხოლოდ დამატებით არგუმენტად იღებს კოზირის მასტს.

5) ფუნცია $beatsList$,რომელიც არგუმენტად იღებს კარტების სიას, კარტს და კოზირის მასტს და აბრუნებს იმ კარტის სიას პირველი არგუმენტიდან, რომლებიც ჭრის მოცემულ კარტს კოზირის გათვალისწინებით.

6) ფუნქცია, რომელიც კარტის მოცემული სიის მიხედვით აბრუნებს რიცხვების სიას, რომლების წარმოადგენს მოცემული კარტის შესაძლო ქულათა ჯამს, რომელიც ითვლება თამაშის „თორმეტი ერთში“ წესების მიხედვით: პატარა კარტი ითვლება ნომინალის მიხედვით, ვალეთი, დამა და ბაბუა ითვლება 10 ქულად, ხოლო ტუზი ან 1 ან 11 ქულად. ფუნქციამ უნდა დააბრუნოს ყველა შესაძლო ვარიანტები.

3. განსაზღვრეთ ტიპი, რომელიც წარმოადგენს გეომეტრიულ ფუნქციას. ფიგურა შეიძლება იყოს ან წრეწირი (ხასიათდება ცენტრის კოორდინატებით და რადიუსით), მართკუთხედი (ხასიათდება მარცხენა ზედა და ქვედა მარჯვენა კუთხეების კოორდი-

ნატებით), სამკუთხედი (წვეროების კოორდინატები) და ტექსტური ველი (მისთვის აუცილებელია შენახული იქნას მარცხენა ქვედა კუთხე, შრიფტი და სტრიქონი, რომელიც წარწერას წარმოადგენს). შრიფტი მოიცემა სამ ელემენტიანი სიმრავლიდან: *Courier*, *Lucida* და *Fixedsys*. განსაზღვრეთ სემდეგი ფუნქციები:

1) ფუნქცია *area*, რომელიც აბრუნებს ფიგურის ფართობს. ტესტური ველის ფართობი დამოკიდებულია შრიფტში ასოების სიმაღლესა და სიგანეზე. ვინაიდან ჩვენს მიერ არჩეული შრიფტები არის მონოსიგანის (ანუ ყველა ასოს სიგანე ერთიდაიგივეა), ჩვენთვის აუცილებელი იქნება განისაზღვროს დამხმარე ფუნქცია, რომელიც მოცემული შრიფტისთვის დააბრუნებს მის გაბარიტებს.

2) ფუნცია *getRectangles*, რომელიც სიიდან მხოლოდ მართკუთხედებს არჩევს.

3) ფუნქცია *getBound*, რომელიც მოცემული ფიგურისთვის აბრუნებს მართკუთხედს, რომელიც საზღვრავს ამ ფიგურას.

4) ფუნქცია *getBounds*, რომელიც ფიგურების მოცემული სიისთვის აბრუნებს მათი შემომსაზღვრელი მართკუთხედების სიას.

5) ფუნქცია *getFigure*, მოცემული ფიგურების სიისთვის და წერტილის კოორდინატების მიხედვით აბრუნებს პირველ ფიგურას, რომლისთვისაც წერტილი ხვდება მის შემომსაზღვრელ მართკუთხედში. გამოიყენეთ ტიპი *Maybe* მნიშვნელობის დასაბრუნებლად.

6) ფუნქცია *move*, რომელიც მოცემული ფიგურისთვის და ძვრის ვექტორისთვის აბრუნებს ახალ ფიგურას, რომელიც დაძრული იქნება მოცემული ვექტორით.

4. უძრავი ქონების სააგენტოში იყიდება ბინები, ოთახები და კერძო სახლები. ბინა ხასიათდება სართულით, ფართობით და სახლის სართულების რაოდენობით. ოთახი ხასიათდება ამის გარდა კიდევ ფართობით (დამატებით მთელი ბინის ფართობისა). კერძო სახლი ხასიათდება მხოლოდ ფართობით. მონაცემთა ბაზაში ინახება მნიშვნელობების წყვილები, რომელთაგან პირველი წარმოადგენს უძრავ ობიექტს, მეორე–მის ფასს. განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს უძრავი ქონების ობიექტებზე ინფორმაციას. განსაზღვრეთ შემდეგი ფუნქციები:

1) *getHouses*, არჩევს მონაცემთა ბაზიდან მხოლოდ კერძო სახლებს.

2) *getByPrice*, არჩევს მონაცემთა ბაზიდან მხოლოდ უძრავი ქონების იმ ობიექტებს, რომელთა ფასი ნაკლებია მითითებულზე.

3) *getByLevel*, ირჩევს მონაცემთა ბაზიდან ბინებს, რომლებიც მოცემულ სართულზე მდებარეობს.

4) *getExceptBounds*, ირჩევს მონაცემთა ბაზიდან ბინებს, რომლებიც არ მდებარეობს პირველ და ბოლო სართულებზე.

შეიმუშავეთ მონაცემების ტიპი, რომელიც წარმოადგენს უძრავი ქონების ობიექტებზე სხვადასხვა მოთხოვნებს: უძრავი ქონების სასურველი ტიპის ობიექტს, მინიმალურ ფართობს, მაქსიმალურ ფასს, შეზღუდვას სართულზე. შექმენით ფუნქცია *query*, რომელიც მოთხოვნების სიის მიხედვით მონაცემთა ბაზიდან ამოიღებს უძრავი ქონების იმ ობიექტებს, რომლებიც აკმაყოფილებს ყველა მოთხოვნას.

5. ბიბლიოთეკაში ინახება წიგნები, გაზეთები და ჟურნალები. წიგნი ხასიათდება ავტორის გვარით და დასახელებით, ჟურნალი–დასახელებით, გამოშვების თვე და წელით, გაზეთი–დასახელებით და გამოშვების თარიღით. მონაცემთა ბაზა წარმოადგენს ამ ობიექტების სიას. შექმენით მონაცემთა ტიპი, რომელიც წარმოადგენს ბიბლიოთეკაში შესანახ ობიექტებს. განსაზღვრეთ შემდეგი ფუნქციები:

1) *isPeriodic*, ამოწმებს, რომ მისი არგუმენტი არის პერიოდული გამოცემა.

2) *getByTytle*, შენახული ობიექტებიდან (მონაცემთა ბაზიდან) იღებს ობიექტებს მოცემული დასახელებით.

3) *getByMonth*, იღებს მონაცემთა ბაზიდან პერიოდულ გამოცემებს, რომლებიც გამოდის თვეში ერთხელ და მითითებულ წელს (შევნიშნოთ, რომ გაზეთები გამოდის თვეში რამდენჯერმე).

4) *getByMonths*, ისევე მუშაობს, როგორც წინა ფუნქცია, მხოლოდ არგუმენტად ღებულობს თვეების სიას

5) *getAuthors*, აბრუნებს ავტორების სიას იმ გამოცემებისა, რომლებიც ბიბლიოთეკაში ინახება.

6. პროგრამირების ზოგიერთ ენაში არის მონაცემთა შემდეგი ტიპები:

- მარტივი ტიპები: მთელი, ნამდვილი და სტრიქონი
- რთული ტიპები: სტრუქტურები. სტრუქტურას აქვს დასახელება და შედგება რამდენიმე ველისგან, რომელთაგან თითოეულს, თავის მხრივ აქვს დასახელება და მარტივი ტიპი.

პროგრამის იდენტიფიკატორების მონაცემთა ბაზა წარმოადგენს წყვილების სიას, რომელიც შედგება იდენტიფიკატორისა და მსის ტიპისგან. შეიმუშავეთ მონაცემთა ტიპი, რომელიც წარმოადგენს აღწერილ ინფორმაციას. განსაზღვრეთ შემდეგი ფუნქციები:

1) *isStructured*, რომელიც ამოწმებს, რომ მისი არგუმენტი არის რთული ტიპის.

2) *getType*, მოცემული სახელითა და იდენტიფიკატორების სიით (მონაცემთა ბაზა) აბრუნებს მოცემული სახელის იდენტიფიკატორის ტიპს (გაითვალისწინეთ, რომ ამ სახელის იდენტიფიკატორი ბაზაში სეიძლება არც იყოს).

3) *getFields*, მოცემული სახელის მიხედვით აბრუნებს იდენტიფიკატორის ველების სიას, თუ იგი არის სტრუქტურის ტიპის.

4) *getByType*, აბრუნებს მონაცემთა ბაზიდან მოცემული ტიპის იდენტიფიკატორების სახელების სიას.

5) *getByTypes*, წინა ფუნქციის ანალოგიურია, მაგრამ ერთი არგუმენტის ნაცვლად იღებს ტიპების სიას. ამ ფუნქციის საშუალებით, მაგალითად, შეიძლება მივიღოთ ყველა რიცხვითი ტიპის იდენტიფიკატორების სია.

7. განსაზღვრეთ სტრიქონებზე შემდეგი ოპერაციები:

- გასუფთავება: სტრიქონიდან ყველა სიმბოლოს ამოგდება
- ამოგდება: მოცემული სიმბოლოს ამოგდება სტრიქონიდან (ყველა შესვლა)
- შეცვლა: ერთი სიმბოლოს ყველა შესვლის შეცვლა მეორეთი
- დამატება: მოცემული სიმბოლოს დამატება სტრიქონის დასაწყისში.

შეიმუშავეთ მონაცემთა ტიპი, რომელსაც ახასიათებს ოპერაციები სტრიქონებზე. განსაზღვრეთ შემდეგი ფუნქციები:

1) *process*, რომელიც არგუმენტად იღებს მოქმედებას და სტრიქონს და აბრუნებს სტრიქონს, რომელიც მოდიფიცირებულია მოქმედების შესაბამისად.

2) *processAll*, წინა ფუნქციის ანალოგიურად, მხოლოდ ღებულობს მოქმედებების სიას და ასრულებს მათ მოცემულ სტრიქონზე თანმიმდევრულად

3) *deleteAll*, იღებს ორ სტრიქონს და აგდებს მეორე სტრიქონიდან პირველი სტრიქონის ყველა სიმბოლოს. რეალიზაციისას აუცილებელია გამოიყენოთ ფუნქცია *processAll*.

8. ელექტრონულ ბლოკნოტში ინახება შემდეგი სახის ჩანაწერები: შეხსენება ნაცნობების დაბადების დღეების, ნაცნობების ტელეფონები და დანიშნული შეხვედრები. შეხსენება შედგება ნაცნობის სახელისგან და თარიღისგან (დღე და თვე). ჩანაწერი ტელეფონზე უნდა შეიცავდეს ადამიანის სახელს და მის ტელეფონს. ინფორმაცია შეხვედრის სესახებ შეიცავს შეხვედრის თარიღს

(დღე, თვე, წელი) და მოკლე აღწერას (შესაძლოა წარმოდგეს სტრიქონით). შეიმუშავეთ მონაცემთა ტიპი, რომელიც ასეთი ტიპის ჩანაწერს წარმოადგენს. ბლოკნოტი არის ჩანაწერების სია.განსაზღვრეთ შემდეგი ფუნქციები:

1) *getByName*, აბრუნებს ინფორმაციას ადამიანზე მისი სახელით (მის ტელეფონს და დაბადების თარიღს).

2) *getByLetter*, აბრუნებს ადამიანების სიას, რომელთა შესახებ ინფორმაცია არის ბლოკნოტში და რომელთა სახელი იწყება მოცემულ ასოზე.

3) *getAssignment*, აბრუნებს მოცემული დროს მიხედვით საქმეების სიას (ინფორმაციას დანიშნულ შეხვედრებსა და მეგობრების ტელეფონის ნომრებს, ვისაც უნდა მოილოცოს დაბადების დღე).

9. კლავიშები კლავიატურაზე შეიძლება იყოს ან მმართველი, ან ანბანურ–ციფრული. ანბანურ–ციფრულ კლავიშზე დაჭერა შეიძლება განხორციელდეს კლავიშა *Shift*–თან ერთად. მმართველი კლავიშებიდან ჩვენ გვაინტერესებს მხოლოდ კლავიშა *CapsLock*. ანბანურ–ციფრულ კლავიშზე ყოველი დაჭერა შეიცავს ინფორმაციას სიმბოლოს სახით. *CapsLock*–ის დაჭერის შემდეგ სიმბოლოები გადადის მაღალ რეგისტრში (თუ ისინი არ არის კლავიშა *Shift*–თან ერთად დაჭერილი) *CapsLock*–ის შემდეგ დაჭერამდე. თუ *CapsLock*–ის რეჟიმი არ არის გააქტივებული, მაშინ კლავიშა *Shift*–თან ერთად დაჭერილი სიმბოლოები გადადის ძედა რეგისტრში. შეიმუშავეთ მონაცემთა ტიპი, რომელიც მოცემულ ინფორმაციას წარმოადგენს. კლავიშების თანმიმდევრული დაჭერა წარმოდგინეთ სიის სახით. ძირითადი ამოცანა მდგომარეობს მასში, რომ დამუშავდეს ფუნქცია, რომელსაც გადაჰყავს ეს თანმიმდევრობა სიმბოლოების სტრიქონში. მაგალითად, დაჭერების თანმიმდევრობამ

```
Shift+'h' 'e' CapsLock 'l' 'l' Shift+'o' CapsLock
```

უნდა შედეგად მოგვცეს სტრიქონი *HeLLo*. განსაზღვრეთ შემდეგი ფუნქციები:

1) *getAlNum*, აბრუნებს დაჭერილი სიიდან მხოლოდ ანბანურ–-ციფრულ კლავიშებზე დაჭერას.

2) *getRaw*, აბრუნებს სტრიქონს, რომელიც შედგება დაჭერილი სიმბოლოებისგან კლავიშების *Shift* და *CapsLock*–ის გაუთვალისწინებლად.

3) *isCapsLocked*, ბოლო დაჭერის მიხედვით გაარკვევს, დარა თუ არა მის შემდეგ რეჟიმი *CapsLock* აქტიურ მდგომარეობაში.

4) *getString*, გადაჰყავს კლავიშებზე დაჭერა სტრიქონში.

ფუნქციის რეალიზებისას შესაძლოა გამოიყენოთ სტანდარტული ფუნქციები ტოპპერ და ტოოწერ, რომლებსაც გადაჰყავთ სიმბოლო შესაბამისად ზედა ან ქვედა რეგისტრში. ეს ფუნქციები განსაზღვრულია მოდულში *Char*. რათა შეძლოთ მათი გამოყენება, პროგრამის დასაწყისში დაუმატეთ სტრიქონი: *importChar*.

10. სწავლისას სტუდენტმა სემესტრის განმავლობაში უნდა ჩააბაროს განსაზღვრული რაოდენობის ლაბორატორიული სამუშაოები, გრაფიკული დავალებები და რეფერატები. ლაბორატორიული სამუშაო ხასიათდება საგნის დასახელებით და ნომრით, გრაფიკული სამუშაო–საგნის დასახელებით, რეფერატი–საგნის დასახელებით და რეფერატის თემის დასახელებით. შეიმუშავეთ მონაცემთა ტიპი, რომელიც შეიცავს ინფორმაციას დავალებების შესახებ. სტუდენტის სასწავლო გეგმა წარმოადგენს წყვილების სიას, რომლის პირველი წევრი წარმოადგენს დავალებას, მეორე – კვირის ნომერს, რომელშიც ეს დავალება ჩაბარდა. თუ დავალება არ სესრულდა, წყვილის მეორე ელემენტი უნდა იყოს ცარიელი (გამოიყენეთ ტიპი *Maybe*). განსაზღვრეთ შემდეგი ფუნქციები:

1) *getByTitle* – აბრუნებს დავალებას, რომელიც აუცილებელია ჩაბარდეს მოცემულ საგანში.

2) *getReferats* – აბრუნებს რეფერატების თემების სიას.

3) *getRest* – აბრუნებს ჩაუბარებელი საგნების სიას.

4) *getRestForWeek* – აბრუნებს დავალებების სიას, რომლებიც მოცემული კვირისთვის არ არის შესრულებული.

5) *getPlot* – ადგენს სიას, რომელიც შედგება წყვილებისგან, რომლიე პირველი ელემნტი არის კვირის ნომერი, ხოლო მეორე – ამ კვირაში ჩაბარებული დავალებების რაოდენობა.

7 საკონტროლო შეკითხვები

1. განსაზღვრეთ ლოკალური ცვლადები.
2. დამცავი პირობა.
3. პოლიმორფიზმი.
4. მომხმარებელთა მონაცემების ტიპების განსაზღვრა.

ლექცია № 4

1 ოპერატორების განსაზღვრა

ბინარული ოპერატორები, ისეთები როგორიცაა +, და ა.შ. ენა *Haskell*–ში ისეთივე ფუნქციებს წარმოადგენს, როგორიცაა ყველა დანარჩენი, მხოლოდ ერთი გამონაკლისით, მათი გამოძახება შეიძლება ინფიქსურ აღვნიშნებშიც. თუ ბინარულ ოპერატორს ფრჩხილებში ავიღებთ, მაშინ მისი გამოძახებისთვის შეიძლება გამოვიყენოთ პრეფიქსული ნოტაცია და მივმართოთ მას, როგორც ჩვეულებრივ ფუნქციას. ასე, რომ შემდეგი ჩანაწერები ექვივალენტურია:

```
2 + 2
(+) 2 2

x < y
(<) x y

x /= y
(/=) x y
```

პირიქითაც, ნებისმიერი ფუნქცია, რომელიც ღებულობს ორ არგუმენტს, შეიძლება გამოყენებული იყოს ინფიქსული სტილით. ამისთვის, მისი სახელი უნდა მოვათავსოთ უკუღმა ბრჭყალში (სიმბოლო). მაგალითად, თუ განსაზღვრულია ფუნქცია:

```
func x y = (x + y) / (x - y)
```

მაშინ მისი გამოძახება შეიძლება ორი სახით:

```
func 5 2
5 `func` 2
```

შემდეგი, თუ ფუნქციის სახელში გხვდება მხოლოდ „სიმბოლოები“ (არა ასოები და არა ციფრები), მაშინ იგი ავტომატურად ითვლება

ინფიქსურ ოპერატორად. განსაზღვრისას მისი სახელი აუცილებლად უნდა ჩაისვას ფრჩხილებში. მაგალითად, განვსაზღვროთ ოპერატორი „მიახლოებით ტოლია“, რომელიც ამოწმებს, არის თუ არა რიცხვი განსხვავებული უფრო მეტად, ვიდრე $0,001$:

```
(~=) x y = abs (x - y) < 0.001
```

ამ განსაზღვრების შემდეგ ეს ოპერატორი შეიძლება გამოყენებული იყოს ისევე, როგორც სხვა დანარჩენი:

```
testApproxEqual x y = if x ~= y then "equal"
                                else "not equal"
```

2 რეკურსიული ტიპები

მონაცემების ტიპების განსაზღვრისას მის მარჯვენა მხარეს შეიძლება გამოყენებული იყოს ამ კონსტრუქციით განსაზღვრული ტიპი. ეს იძლევა საშუალებას განისაზღვროს მონაცემების რეკურსიული ტიპები. ერთ–ერთი ძირითადი რეკურსიული ტიპია ხე. ბინარული ხე, რომლის ფოთლებზე არის a ტიპის ელემენტები, შეიძლება განვსაზღვროთ ასე:

```
data Tree a  =  Leaf a
             | Branch (Tree a) (Tree a)
```

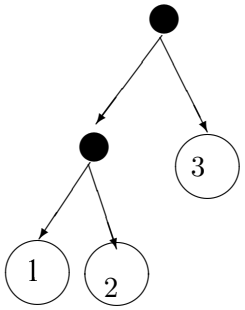
ამ განმარტებით ხე (*Tree*) წარმოადგენს ან ფოთოლს (*Leaf*), ანუ წვეროს, რომელსაც არ ჰყავს შთამომავალი, ან შტოს (*Branch*), ანუ წვეროს, რომელსაც აქვს მარცხენა და მარჯვენა ქვეხე. შევნიშნოთ, რომ მოტანილ განმარტებებში *Leaf* და *Branch* არის მონაცემთა კონსტრუქტორები, ხოლო *Tree a*, რომელიც გვხვდება განმარტების მარჯვენა და მარცხენა მხარეს, არის ტიპის სახელი. რეკურსიული ტიპის მუშაობა პრაქტიკულად არ განსხვავდება ჩვეულებრივი ტიპის მუშაობისგან, იმ გამონაკლისით, რომ პრაქტიკულად ყველა ფუნქცია, რომელიც რეკურსიულ ტიპებთამ მუშაობს, თვითონაც არის რეკურსიული. მაგალითად, განვსაზღროთ ფუნქცია *treeSize*, რომელიც დააბრუნებს ხეში ფოთლების რაოდენობას. ის ჩაიწერება შემდეგნაირად:

```
treeSize (Leaf _) = 1
treeSize (Branch l r) = treeSize l + treeSize r
```

ეს ფუნქცია შეიძლება გამოვიყენოთ შემდეგნაირად:

```
Main>treeSize (Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3))
3
```

აქ იგი გამოვიყენეთ შემდეგი სახის ხისათვის:



განვიხილოთ ხეებთან მუშაობის სხვა ფუნქცია, რომელიც ემსახურება ხის ყველა ფოთლის მიღებას:

```
leafList (Leaf x)            =  [x]
leafList (Branch left right) =  leafList left ++
                                leafList right
```

3 სიები, როგორც რეკურსიული ტიპები

სიაც, ასევე, წარმოადგენს რეკურსიულ ტიპს. განვიხილოთ შემდეგი პოლიმორფული ტიპი:

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

ტიპის მნიშვნელობა არის ან ცარიელი სია (*Nil*), ან შეიცავს *a* ტიპის ელემენტს ან *List a* ტიპის მნიშვნელობას. ძნელი არ არის, შევნიშნოთ ანალოგია სიებთან, რომლებიც, ასევე, ან არის ცარიელი, ან შეიცავს *a* ტიპის თავს და კუდს, რომელიც ასევე სიას წარმოადგენს. მსგავსება უფრო თვალსაჩინო გახდება, თუ *Cons* კონსტრუქტორს ჩავწერთ ინფიქსური სახით:

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

ამრიგად, სიის ტიპი შეიძლება განსაზღვრულიყო ასეთი სახით:
-- ეს არ არის ასკელლ ენის ნამდვილი კოდი

```
data [a] = [] | a : [a]
```

ტიპის მნიშვნელობისთვის შეიძლება განვსაზღვროთ ყველა ის ფუნქცია, რომლებიც განსაზღვრულია სიებისთვის. მოვიყვანოთ ფუნქციების *head*, *tail* და *map* მაგალითები:

```
headList (Cons x _) = x
headList Nil        = error "headList: empty list"

tailList (Cons _ y) = y
tailList Nil        = error "tailList: empty list"
```

მოვახდინოთ ამ ფუნქციების მუშაობის დემონსტრირება:

```
Main>headList (Cons 1 (Cons 2 Nil))
1
Main>tailList (Cons 1 (Cons 2 Nil))
Cons 2 Nil
```

4 სინტაქსური ხეები

პროგრამირებაში ფართოდ გამოიყენება ხის ტიპის სტრუქტურები. მაგალითად, ნებისმიერ კომპილატორის მიერ პროგრამის გრამატიკული გარჩევის შედეგს წარმოადგენს სინტაქსური ხე. მოვიყვანოთ ასეთი ხის მაგალითი გამოსახულებისთვის, რომელიც შეიცავს მუდმივებს, შეკრებისა და გამრავლების სიმბოლოებს:

```
data Expr =   Const Integer
            | Add Expr Expr
            | Mult Expr Expr
```

ამ განსაზღვრებიდან ცხადია, რომ გამოსახულება (*Expression*) წარმოადგენს ან მთელრიცხვა მუდმივს (*Constant*), ან ორი გამოსახულების ჯამს ან ნამრავლს. მაგალითად, გამოსახულების $1 + 2?(3 + 4)$ შესაბამის *Expr* ტიპის მნიშვნელობას ექნება სახე:

```
Add (Const 1) (Mult (Const 2) (Add (Const 3) (Const 4)))
```

გამოსახულების გამოთვლის ფუნქცია შეიძლება განვსაზღროთ შემდეგნაირად:

```
eval :: Expr -> Integer
eval (Const x) = x
eval (Add x y) = eval x + eval y
eval (Mult x y) = eval x * eval y
```

შესაძლოა გავაფართოვოთ, თუ შემოვიტანთ გამოსახულებებში ცვლადების გამოყენების შესაძლებლობას:

```
data Expr =   Const Integer
            | Var String
            | Add Expr Expr
            | Mult Expr Expr
```

კონსტრუქტორი Var განსაზღვრავს ცვლადს მოცემული სახელით. ასეთი $Expr$ ტიპი საშუალებას გვაძლევს განვსაზღვროთ, მაგალითად, გამოსახულების დიფერენცირების ფუნქცია:

```
diff :: Expr -> Expr
diff (Const _) = Const 0
diff (Var x) = Const 1
diff (Add x y) = Add (diff x) (diff y)
diff (Mult x y) = Add (Mult (diff x) y) (Mult x (diff y))
```

შევამოწმოთ ამ ფუნქციის მუშაობა გამოსახულების $x + x^2$ (არ დაგავიწყდეთ დაუმატოთ ფუნქცია $deriving(Show)$ ტიპი $Expr$–ის განსაზღვრების შემდეგ):

```
Main>diff (Add (Var "x") (Mult (Var "x") (Var "x")))
Add (Const 1) (Add (Mult (Const 1) (Var "x"))
 (Mult (Var "x") (Const 1)))
```

ამრიგად, დიფერენცირების შედეგად ჩვენ მივიღეთ გამოსახულება $1 + (1x + x1)$, რომელიც წარმოადგენს სწორ გამოსახულებას, ოღონდ, რასაკვირველია, საჭიროებს გამარტივებას. ფუნქცია $diff$–ის სხვა შეზღუდვაა ის, რომ ფუნქციაში არ განიჩევა, თუ რომელი ცვლადით ხდება დიფერენცირება. შესაბამისად, ფუნქცია უნდა ღებულობდეს დამატებით პარამეტრს–დიფერენცირების ცვლადის სახელს.

Expr ტიპის მნიშვნელობის განსაზღვრა საკმაოდ მოუხერხებელია. პრინციპში, შეიძლება დაიწეროს ფუნქცია, რომელიც გარდაქმნის "$1+x*y$" ტიპის სტრიქონს *Expr* ტიპის შესაბამის მნიშვნელობად. თუმცა, ასეთი ფუნქციის დაწერა საკმაოდ რთულია და ამიტომაც, რეკომენდირებულია ფუნქციის *parseExpr*–ის გამოყენება (განსაზღვრულია ფაილში *expr.hs*). თავიდან დაუმატეთ სტრიქონი

```
import Expr
```

ფუნქციას *parseExpr*–ს აქვს შემდეგი ტიპი:

```
parseExpr :: String -> Expr
```

მოცემული სტრიქონისთვის იგი აბრუნებს მის წარმოდგენას, როგორც ტიპი *Expr*–ის მნიშვნელობას:

```
Main>parseExpr "1+x"
Add (Const 1) (Var "x")
```

5 დავალებები

1. მუშაობა *Expr* ტიპთან. გამოიყენეთ ზემოთ განსაზღვრული ტიპი *Expr* და მოახდინეთ შემდეგი ფუნქციების რეალიზება (ტესტირებისთვის გამოიყენეთ ფუნქცია პარსეხპრ).

1) განსაზღვრეთ ფუნქცია $diff$, რომელიც დამატებით არგუმენტად ღებულობს ცვლადის სახელს, რომლის მიხედვითაც აუცილებელია მოხდეს დიფერენცირება.

2) განსაზღვრეთ ფუნქცია $simplify$, რომელიც ამარტივებს ხპრ ტიპის გამოსახულებას შემდეგი წესების გამოყენებით:

$x + 0 = 0 + x = x$
$x \cdot 1 = 1 \cdot x = x$
$x \cdot 0 = 0 \cdot x = 0$
და ა.შ.

3) განსაზღვრეთ ფუნქცია *toString*, რომელსაც გარდაქმნის *Expr* ტიპის გამოსახულებას სტრიქონად. მაგალითად, ფუნქციის გამოყენების შედეგი გამოსახულებასთან $Add(Mult(Const2)(Var"x"))(Var"y")$ უნდა იყოს სტრიქონი "$2*x+y$". გაითვალისწინეთ ფრჩხილების გამოყენების შესაძლებლობა. მაგალითად, გამოსახულება $Mult(Const2)(Add(Var"x")(Var"y"))$ უნდა გარდაიქმნას სტრიქონად "$2 * (x + y)$"

4) განსაზღვრეთ ფუნქცია *eval*, რომელიც იღებს ორ პარამეტრს: *Expr* ტიპის გამოსახულებას და $(String, Integer)$ ტიპის წყვილების

სიას, რომლებიც იძლევა შესაბამისობას სახელებსა და მათ მნიშვნელობებს შორის. ფუნქციამ უნდა გამოითვალოს გამოსახულების მნიშვნელობა გამოსახულების მოცემული მნიშვნელობების გათვალისწინებით. მაგალითად, გამოსახულება $eval(Add(Var"x")(Var"y"))[("x", 1), ("y", 2)]$ უნდა იძლეოდეს რიცხვს 3–ს.

2. ფუნქციები *List* ტიპთან სამუშაოდ. ადრე შემოტანილი *List* ტიპისთვის განსაზღვრეთ შემდეგი ფუნქციები:

1) *lengthList*, რომელიც აბრუნებს *List* ტიპის სიის სიგრძეს.

2) *nthList*, რომელიც აბრუნებს სიის n-ურ ელემენტს.

3) *removeNegative*, რომელიც მთელი რიცხვების სიიდან (ტიპი *List Integer*) ამოაგდებს უარყოფით ელემენტებს.

4) *fromList*, რომელიც გარდაქმნის *List* ტიპის სიას ჩვეულებრივ სიად.

5) *toList*, რომელიც გარდაქმნის ჩვეულებრივ სიას *List* ტიპის სიად.

3. ბინარულ ხეებთან მუშაობის ფუნქციები. განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს ძებნის ბინარულ ხეებს. განხილული ხეებისაგან განსხვავებით, ძებნის ხეებში მონაცემები შეიძლება იყოს არა მარტო ფოთლებში, არამედ ხის შუალედურ კვანძებში. გამოვიყენოთ ხეები ასოციატიური მასივის წარმოსადგენად, რომელიც გასაღებების მნიშვნელობებს (რომლებიც სტრიქონებად არის წარმოდგენილი) შეუსაბამებს მთელ რიცხვებს. თითოეული წვეროსთვის (რომელიღაც გასაღებით) მარცხენა ქვეხე უნდა შეიცავდეს ელემენტებს გასაღების ნაკლები მნიშვნელობით, ხოლო მარჯვენა–უფრო მეტით. სტრიქონსა და რიცხვს შორის შესაბამისობის მოძებნისას აუცილებელია ამ ინფორმაციის გათვალისწინება, ვინაიდან იგი იძლევა საშუალებას უფრო ეფექტურად იყოს ამოღებული ინფორმაცია ხიდან. განსაზღვრეთ მონაცემთა აღწერილი ტიპი და შემდეგი ფუნქციები:

1) *add*, რომელიც ხეს უმატებს გასაღებისა და მნიშვნელობის მოცემულ წყვილს.

2) *find*, აბრუნებს სტრიქონის შესაბამის რიცხვს.

3) *exists*, ამოწმებს, რომ ელემენტი მოცემული გასაღებით არის ხეში.

4) *toList*, გარდაქმნის მოცემულ ძებნის ხეს სიად, რომელიც დალაგებულია გასაღებების მნიშვნელობების მიხედვით.

4. შექმენით მონაცემთა ტიპი, რომელიც შეიცავს ფაილური სისტემის კატალოგს. ითვლება, რომ თითოეული ფაილი შეიცავს ან მონაცემებს, ან თვითონ წარმოადგენს კატალოგს. კატალოგი შეიცავს სხვა ფაილებს (რომლებიც, თავის მხრივ, შეიძლება იყოს

კატალოგი) მათ სახელებთან და ზომასთან (ბაიტებში) ერთად. ფაილების შინაარსს ყურადღება არ ექცევა: მონაცემთა ტიპი უნდა წარმოდგეს მხოლოდ სახელით, ზომით და კატალოგის სტრუქტურით. განსაზღვრეთ შემდეგი ფუნქციები:

1) *dirAll*, აბრუნებს კატალოგის ყველა ფაილის სრული სახელების სიას, ქვეკატალოგების ჩათვლით.

2) *find*, აბრუნებს გზას, რომელსაც მივყავართ მოცემული სახელის ფაილთან. მაგალითად, თუ კატალოგი შეიცავს ფაილებს a, b და c, და b წარმოადგენს კატალოგს, რომელიც შეიცავს x და y, მაშინ ძებნის ფუნქციამ უნდა დააბრუნოს სტრიქონი "b/x".

3) *du*, მოცემული კატალოგისთვის აბრუნებს ბაიტების რაოდენობას, რომელიც უჭირავს მის ფაილებს (ქვეკატალოგის ფაილების ჩათვლით).

5. დებულებას დავარქმევთ ლოგიკურ ფორმულას, რომელსაც აქვს ერთ–ერთი შემდეგი ფორმებიდან:

ცვლადის სახელი(სტრიქონი)

$p\&q$

$p|q$

$\overline{p}$

სადაც p და q არის დებულებები. მაგალითად, დებულებებია შემდეგი ფორმულები:

x

$x|y$

$x\&(x|\ y)$

შეიმუშავეთ მონაცემთა ტიპი *Prop*, რომელიც წარმოადგენს ამ სახის დებულებებს. განსაზღვრეთ შემდეგი ფუნქციები:

1) $vars :: Prop -> [String]$, რომელიც აბრუნებს ცვლადების სახელების სიას (განმეორებების გარეშე), რომლებიც გვხვდება დებულებებში.

2) დავუშვათ მოცემულია ცვლადების სახელების სია და მათი *Bool* ტიპის მნიშვნელობები, მაგალითად, $[("x", True), ("y", False)]$. განსაზღვრეთ ფუნქცია $truthValue :: Prop -> [(String, Bool)] -> Bool$, რომელიც განსაზღვრავს არის თუ არა დებულება ჭეშმარიტი, თუ ცვლადებს აქვთ სიით მოცემული მნიშვნელობები.

3) განსაზღვრეთ ფუნქცია $tautology :: Prop -> Bool$, რომელიც აბრუნებს $True$–ს, თუ დებულება არის ჭეშმარიტი ცვლადების ნებისმიერი მნიშვნელობისთვის, რომლებიც მასში გვხვდება (მაგალითად, ეს სრულდება დებულებისთვის $(x|\ x)$).

6. ლექსიკური ხეები (ტრიე-ხეები) გამოიყენება ლექსიკონების წარმოსადგენად. ხის ყოველი წვერო შეიცავს შემდეგ ინფორმა-

ციას: სიმბოლოს, ლოგიკურ მნიშვნელობას და ქვეხეების სიას (ნებისმიერ წვეროს შეიძლება ჰქონდეს ნებისმიერი რაოდენობის შვილობილი ხეები. *trie*-ხის მაგალითი მოყვანილია სურ. 1–ზე. ლოგიკური მნიშვნელობა $True$ აღნიშნავს სიტყვის ბოლოს, რომლიც იკითხება ხის წვეროდან. სურათზე ასეთი წვეროებიაღნიშნულია სიმბოლო $*$–ით. ამრიგად, მოცემული ხით წარმოდგენილია სიტყვები $fa, false, far, fare, fact, fried, frieze$. განსაზღვრეთ შემდეგი ფუნქციები:

exists, რომელიც ამოწმებს, მოცემული ხე არის თუ არა *trie*-ხეში.

insert, რომელიც მოცემული ხისთვის და სიტყვისთვის აბრუნებს ახალ ხეს, რომელიც შეიცავს ამ სიტყვას. თუ სიტყვა უკვე არის ხეში, მაშინ იგი უნდა დაბრუნდეს ცვლილების გარეშე. სურ. 1. *trie*-ხეები

completions, რომელიც მოცემული სტრიქონის შესაბამისად აბრუნება ყველა სიტყვის სიას, რომელთა დასაწყისიც არის მოცემული სტრიქონი (მაგალითად. სურ. 1–ზე მოცემული ხისთვის სტრიქონზე "fri" უნდა დაბრუნდეს სია ["$fried$","$frieze$"]).

7. თეორიულად შესაძლებელია, თუმცა არაეფექტურია განისაზღვროს მთელი რიცხვი მონაცემთა რეკურსიული ტიპით შემდეგნაირად:

$dataNumber = Zero|NextNumber$ ანუ, რიცხვი ან ნულია ($Zero$), ან განისაზღვრება, როგორც რიცხვი, რომელსაც მოსდევს შემდეგი ციფრი. მაგალითად, რიცხვი 3 ჩაიწერება როგორც $Next(Next(NextZero))$. ასეთი წარმოდგენისთვის განსაზღვრეთ შემდეგი ფუნქციები:

1) $fromInt$, მოცემული მთელი $Integer$ ტიპის რიცხვისთვის აბრუნებს მისი მნიშვნელობის შესაბამის $Number$ ტიპის მნიშვნელობას.

2) $toInt$, რომელიც გარდაქმნის $Number$ ტიპის მნიშვნელობას შესაბამის მთელ რიცხვად.

3) $plus :: Number- > Number- > Number$, რომელიც უმატებს თავის არგუმენტებს.

4) $mult :: Number- > Number- > Number$, რომელიც ამრავლებს თავის არგუმენტებს.

5) *dec*, რომელიც აკლებს თავის არგუმენტს 1–ს. $Zero$–სთვის ფუნქციამ უნდა დააბრუნოს $Zero$.

6) $fact$, რომელიც ითვლის ფაქტორიალს.

8. დაწესებულებაში თანამდებობების იერარქია წარმოადგენს ხის სტრუქტურას. თითოეული თანამშრომელი ხასიათდება უნიკალური სახელით და ჰყავს რამდენიმე დაქვემდებარებული. განსაზღვრეთ

მონაცემთა ტიპი, რომელიც წარმოადგენს ასეთ იერარქიას და დაწერეთ შემდეგი ფუნქციები:

1) *getSubordinate*, რომელიც აბრუნებს მოცემული თანამშრომლის დაქვემდებარებულების სიას.

2) *getAllSubordinate*, რომელიც აბრუნებს მოცემული თანმშრომლის ყველა დაქვემდებარებულის სიას, მათ შორის ირიბი დაქვემდებარებულებებისაც.

3) *getBoss*, აბრუნებს მოცემული თანამშრომლის უფროსს.

4) *getList*, აბრუნებს წყვილების სია, რომლებიდანაც პირველი ელემენტია თანამშრომლის სახელი, მეორე – მისი დაქვემდებარებულებების (ირიბი დაქვემდებარებულების ჩათვლით) რაოდენობა.

9. სიბრტყეზე არე წარმოადგენს ან მართკუთხედს, ამ წრეს, ან ამ არეების გაერთიანებას ან გადაკვეთას. მართკუთხედი ხასიათდება მარცხენა ქვედა და მარჯვენა ზედა წვეროების კოორდინატებით, წრე – ცენტრის კოორდინატებით და რადიუსით. შეადგინეთ მონაცემთა სტრუქტურა, რომელიც წარმოადგენს აღწერილ არეს. განსაზღვრეთ შემდეგი ფუნქციები:

1) *contains*, რომელიც შეამოწმებს, მოცემული წერტილი არის თუ არა არეში.

2) *isRectangular*, ამოწმებს, რომ არე მოიცემა მხოლოდ მართკუთხედებით.

3) *isEmpty*, ამოწმებს, რომ არე არის ცარიელი, ანუ სიბრტყის არცერთი წერტილი მასზე არ გვხვდება.

10. ობიექტ–ორიენტირებულ ენაზე კლასი შეიცავს მეთოდების ჯგუფს (მოცემულ დავალებაში კლასის მონაცემები – ველები არ განვიხილოთ). ამას გარდა, მას შეიძლება ჰქონდეს ერთადერთი მშობელი კლასი (მრავლობითი მემკვიდრეობითობას არ განვიხილავთ). თუმცა, არსებობს კლასები მშობლების გარეშე. მემკვიდრეობითობის დროს მშობლის მეთოდებს ემატება შთამომავლის მეთოდები. განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს ინფორმაციას კლასების იერარქიის შესახებ. აღწერეთ შემდეგი ფუნქციები:

1) *getParent*, აბრუნებს მოცემული სახელის კლასის უშუალო წინაპარს.

2) *getPath*, აბრუნებს მოცემული კლასის ყველა წინაპრის (უშუალო წინაპარს, წინაპრის წინაპარს და ა.შ).

3) *getMethods*, აბრუნებს მოცემული კლასის მეთოდებს მემკვიდრეობითობის გათვალისწინებით.

4) *inherit*, უმატებს კლასების იერარქიას მოცემული სახელის კლასს,რომელიც მემკვიდრეობით მიიღება მოცემული წინაპროდან.

ლექცია № 5

1 უმაღლესი რიგის ფუნქციები

განვიხილოთ ორი მაგალითი. ვთქვათ, მოცემულია რიცხვების სია. საჭიროა დაიწეროს ორი ფუნქცია, პირველი ფუნქცია აბრუნებს ამ რიცხვებიდან კვადრატული ფესვების სიას, მეორე – მათი ლოგარითმების სიას. ეს ფუნქციები შეიძლება ასე განისაზღვროს:

```
sqrtList [] = []
sqrtList (x:xs) = sqrt x : sqrtList xs

logList [] = []
logList (x:xs) = log x : logList xs ,
```

რომ ეს ფუნქციები იყენებს ერთიდაიგივე მიდგომას და მთელი განსხვავება მათ შორის არის ის, რომ ახალი სიის ელემენტის გამოთვლისთვის პირველი ფუნქცია იყენებს კვადრატული ფესვის ამოღების ფუნქციას, მეორე კი – ლიგარითმს. შეიძლება მოვახდინოთ ელემენტის გარდაქმნის ფუნქციის აბსტრაგირება? აღმოჩნდა, რომ შეიძლება. გავიხსენოთ, რომ *Haskell*–ში ფუნქციები წარმოადგენს „პირველი კლასის“ ელემენტებს: ისინი შეიძლება გადავცეთ პარამეტრებად სხვა ფუნქციებს. განვსაზღვროთ ფუნქცია $transformList$, რომელიც იღებს ორ პარამეტრს: გარდაქმნის ფუნქციას და გარდასაქმნელ სიას.

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

ეხლა ფუნქციები *sqrtList* და *logList* შეიძლება ასე განისაზღვროს:

```
sqrtList l = transformList sqrt l
logList l = transformList log l
```

ან, კარირების გათვალისწინებით:

```
sqrtList = transformList sqrt
logList = transformList log
```

1.1 ფუნქცია *map*

სინამდვილეში ფუნქცია, რომელიც სრულად შეესაბამება *transformList*, უკვე განსაზღვრულია სტანდარტულ ბიბლიოთეკაში და ეწოდება *map* (ინგლისურიდან, ასახვა). მას აქვს შემდეგი ტიპი:

```
map :: (a -> b) -> [a] -> [b]
```

ეს ნიშნავს, რომ მისი პირველი არგუმენტი არის $a->b$ ტიპის ფუნქცია, რომელიც ნებისმიერი a ტიპის მნიშვნელობას ასახავს b ტიპის მნიშვნელობაში (საზოგადოდ, ეს ტიპები შეიძლება ემთხვეოდეს ერთმანეთს). ფუნქციის მეორე არგუმენტია a ტიპის მნიშვნელობების სია. მაშინ ფუნქციის შედეგი იქნება b ტიპის მნიშვნელობების სია. *map*–ის მსგავსი ფუნქციები, რომლებიც არგუმენტად იღებს სხვა ფუნქციებს, უწოდებენ მაღალი რიგის ფუნქციებს. მათ ძალზე ხშირად იყენებენ ფუნციონალური პროგრამების დაწერისას. მათი საშუალებით შეიძლება ცხადად გამოიყოს ალგორითმის რეალიზაციის დეტალები (მაგალითად, კონკრეტული გარდაქმნის ფუნქცია *map*–ში) მისი მაღალდონიანი სტრუქტურისაგან (სიის თითოეული ელემენტის გარდაქმნა). ალგორითმები, რომლებიც წარმოდგენილია მაღალი დონის ფუნქციების გამოყენებით, როგორც წესი უფრო კომპაქტურია და ცხადი, ვიდრე ჩვეულებრივი რეალიზაციები.

1.2 ფუნქცია *filter*

შემდეგი მაღალი რიგის ფუნქცია, რომელიც ხშირად გამოიყენება, არის ფუნქცია *filter*. მოცემული პრედიკატის (ფუნქცია, რომელიც აბრუნებს ბულის მნიშვნელობას) და სიის მიხედვით ის აბრუნებს იმ ელემენტების სიას, რომლებიც აკმაყოფილებს მოცემულ პრედიკატს:

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

მაგალითად, ფუნქცია, რომელიც სიიდან იღებს მის დადებით ელემენტებს, განისაზღვრება ასე:

```
getPositive = filter isPositive
isPositive x = x > 0
```

1.3 ფუნქციები *foldr* და *foldl*

უფრო რთული მაგალითია ფუნქციები *foldr* და *foldl*. განვიხილოთ ფუნქციები, რომლებიც აბრუნებენ სიის ელემენტების ჯამს და ნამრავლს:

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

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

აქ შეიძლება დავინახოთ საერთო ელემენტები: საწყისი მნიშვნელობა (0 ჯამისთვის და 1–გამრავლებისთვის) და ფუნქცია, რომელიც კომბინირებს მნიშვნელობებს. ფუნქცია *foldr* ცხადად წარმოადგენს ასეთი სქემის განზოგადოებას:

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

ფუნქცია *foldr* პირველ არგუმენტად იღებს კომბინირებულ ფუნქციას (შევნიშნოთ, რომ მან შეიძლება მიიღოს სხვადასხვა ტიპის არგუმენტები, მაგრამ შედეგის ტიპი უნდა ემთხვეოდეს მეორე არგუმენტის ტიპს). ფუნქცია *foldr*–ის მეორე არგუმენტი არის საწყისი მნიშვნელობა კომბინაციისას. მესამე არგუმენტად გადაეცემა სია. ფუნქცია ანხორციელებს სიის „შეხვევას“ გადაცემული პარამეტრების შესაბამისად. რათა უკეთ გავიგოთ, თუ როგორ მუშაობს ფუნქცია ფოლდრ, ავწეროთ მისი განსაზღვრება ინფიქსური ნოტაციის გამოყენებით:

```
foldr f z []       = z
foldr f z (x:xs)   = x `f` (foldr f z xs)
```

წარმოვადგინოთ ელემენტების სია $[a, b, c, ..., z]$ ოპერატორი :–ის გამოყენებით. *foldr* ფუნქციის გამოყენების წესი ასეთია: ყველა ოპერატორი : იცვლება f ფუნქციის გამოყენებით ინფიქსური სახით ($`f`$), ხოლო ცარიელი სტრიქონის სიმბოლო [] იცვლება კომბინაციის საწყის მნიშნელობაზე. გარდაქმნის ნაბიჯები შეიძლება ასე წარმოვადგინოთ (ჩავთვალოთ, რომ საწყისი მნიშვნელობა ტოლია *init*)

```
[a,b,c,...,z]
a : b : c : ... : []
a : (b : (c : (... (z : [])...)))
a `f` (b `f` (c `f` (... (z `f` init)))))
```

ფუნქცია *foldr*–ის საშუალებით სიის ელემენტების შეკრება და გამრავლება ასე განისაზღვრება:

```
sumList = foldr (+) 0
multList = foldr (*) 1
```

ვნახოთ, როგორ გამოითვლება ამ ფუნქციის მნიშვნელობა მაგალითზე – სიაზე

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

ფუნქციის დასახელება მოდის ინგლისური სიტყვიდან *fold* – დაკეცვა, დალაგება (მაგალითად, ქაღალდის ფურცლების). ასო *r* ფუნქციის დასახელებაში მოდის სიტყვიდან *right* (მარჯვენა) და უჩვენებს დაკეცვისთვის გამოყენებული ფუნქციის ასოციატიურობას. ასე, რომ მაგალითებიდან ჩანს, რომ ფუნქციის გამოყენება ჯგუფდება მარჯვნივ. განვსაზღვრება ფუნქცია *foldl*–ის, სადაც *l* უჩვენებს, რომ ოპერაციის გამოყენება ჯგუფდება მარცხნივ, მოყვანილია ქვემოთ:

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

ასოციატიური ოპერაციისას, როგორიცაა შეკრება და გამრავლება, ფუნქციები *foldr* და *foldl* ექვივალენტურია, თუმცა, თუ ოპერაცია არ არის ასოციატიური, მათი შედეგები იქნება განსხვავებული:

```
Main>foldr (-) 0 [1,2,3]
2
Main>foldl (-) 0 [1,2,3]
-6
```

მართლაც, პირველ შემთხვევაში გამოითვლება სიდიდე $1 - (2 - (3 - 0)) = 2$, ხოლო მეორეში–სიდიდე $((0 - 1) - 2) - 3 = -6$.

1.4 უმაღლესი რიგის სხვა ფუნქციები

სტანდარტულ ბიბლიოთეკაში განსაზღვრულია ფუნქცია *zip*. ის გარდაქმნის ორ სიას წყვილების სიად:

```
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b):zip as bs
zip _ _ = []
```

გამოყენების მაგალითი:

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

შევნიშნოთ, რომ საშედეგო სიის სიგრძე ტოლია საწყისი სიებიდან ყველაზე მოკლე სიის. ამ ფუნქციის გაფართოებაა მაღალი დონის ფუნქცია *zipWith*, რომელიც „აერთებს“ ორ სიას მოცემულიფუნქციის საშუალებით:

```
zipWith                  :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)   = z a b : zipWith z as bs
zipWith _ _      _        = []
```

ამ ფუნქციის საშუალებით შეგვიძლია ადვილად განვსაზღვროთ, მაგალითად, ორი სიის ელემენტების ერთმანეთთან შეკრების ფუნქცია:

```
sumList xs ys = zipWith (+) xs ys
```

ანუ, კარირების გათვალისწინებით:

```
sumList = zipWith (+)
```

2 *lambda* - აბსტრაქციები

მაღალი რიგის ფუნქციის გამოყენებისას ხშირად აუცილებელია განისაზღვროს ბევრი დამატებითი ფუნქცია. მაგალითად, ფუნქცია *getPositive*–ის განსაზღვრისას ჩვენ მოგვიწია გაგვესაზღვრა დამატებითი ფუნქცია *isPositive*, რომელიც საჭიროა იმისთვის, რომ

დაგვედგინა, არგუმენტი არის თუ არა დადებითი. პროგრამის მოცულობის ზრდასთან ერთად დამხმარე ფუნქციის სახელების მოფიქრების აუცილებლობა სულ უფრო გვიშლის ხელს. თუმცა, ენა *Haskell*–ში შესაძლებელია განისაზღვროს უსახელო ფუნქციები *lambda*–აბსტრაქციის კონსტრუქციების გამოყენებით. მაგალითად, უსახელო ფუნქცია, რომელიც თავის არგუმენტს აიყვანს კვადრატში, მიუმატებს ერთიანს და გაამრავლებს 2–ზე, ჩაიწერება ასე:

```
\x -> x * x
\x -> x + 1
\x -> 2 * x
```

ეხლა მათი გამოყენება შეიძლება არგუმენტებად უმაღლესი რიგის ფუნქციებში. მაგალითად, ფუნქცია, რომელსაც სიის ელემენტები აჰყავს კვადრატში, შეიძლება ასე ჩაიწეროს:

```
squareList l = map (\x -> x * x) l
```

ფუნქცია *getPositive* შეიძლება განისაზღვროს შემდეგნაირად:

```
getPositive = filter (\x -> x > 0)
```

შესაძლოა *lambda*–აბსტრაქცია განისაზღვროს რამდენიმე ცვლადისთვის:

```
\x y -> 2 * x + y
```

ლამბდა–აბსტრაქცია შეიძლება გამოვიყენოთ როგორც ჩვეულებრივი ფუნქცია, მაგალითად, გამოვიყენოთ არგუმენტებთან:

```
Main>(\x -> x + 1) 2
3
Main>(\x -> x * x) 5
25
Main>(\x -> 2 * x + y) 1 2
4
```

ლამბდა–აბსტრაქციის გამოყენებით შესაძლოა განისაზღვროს ფუნქცია. მაგალითად, ჩანაწერი

```
square = \x -> x * x
```

ექვივალენტურია

```
square x = x * x
```

3 სექციები

ფუნქციები შეიძლება გამოვიყენოთ ნაწილობრივ, ანუ არ მივცეთ მნიშვნელობა ყველა მის არგუმენტს. მაგალითად, თუ ფუნქცია *add* განსაზღვრულია როგორც

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

მაშინ შეიძლება განისაზღვროს ფუნქცია *inc*, რომელიც თავის არგუმენტს ზრდის ერთით შემდეგნაირად:

```
inc = add 1
```

აღმოჩნდა, რომ ბინარული ოპერატორები, როგორც ენაში ჩადგმული, ასევე მომხმარებლის მიერ განსაზღვრული, ასევე შეიძლება გამოყენებული იყოს არგუმენტების ნაწილთან (ვინაიდან ბინარულ ოპერატორებს ორი არგუმენტი აქვთ, ეს ნაწილი შედგება ერთი არგუმენტისგან). ბინარულ ოპერაცია, რომელიც გამოიყენება ერთ არგუმენტთან, უწოდებენ სექციას. მაგალითი:

```
(x+) = \y -> x+y
(+y) = \x -> x+y
(+)  = \x y -> x+y
```

ფრჩხილები აქ აუცილებელია. ფუნქციები *add* და *inc* შეიძლება ასე განისაზღვროს:

```
add = (+)
inc = (+1)
```

სექციები განსაკუთრებით სასარგებლოა, როცა გამოიყენება არგუმენტებად მაღალი რიგის ფუნქციებში. გავიხსენოთ ფუნქციის განსაზღვრება სიის დადებითი ელემენტების მისაღებად:

```
getPositive = filter (\x -> x > 0)
```

სექციის გამოყენებით ის ჩაიწერება უფრო კომპაქტურად:

```
getPositive = filter (>0)
```

სიის ელემენტების გაორმაგებას ახდენს ფუნქცია:

```
doubleList = map (*2)
```

4 დავალებები

1. განსაზღვრეთ ფუნქცია მაღალი რიგის ფუნქციების გამოყენებით:

1) ფუნქცია, რომელიც ითვლის ნამდვილი რიცხვების სიის ელემენტების საშუალო არითმეტიკულს ფუნქცია $foldr$–ის გამოყენებით. ფუნქციამ მხოლოდ ერთხელ უნდა გადახედოს სიას.

2) ფუნქცია, რომელიც ითვლის ორი სიის სკალარულ ნამრავლს (გამოიყენეთ ფუნქციები $foldr$ და $zipWith$).

3) ფუნქცია $countEven$, რომელიც აბრუნებს სიის ლუწ ელემენტებს.

4) ფუნქცია $quicksort$, რომელიც ახდენს სიის სწრაფ დახარისხებას რეკურსიული ალგორითმით. იმისათვის, რომ დახარისხდეს სია xs, მისგან აიღება პირველი ელემენტი (ავღნიშნოთ იგი x–ით). დანარჩენი სია იყოფა ორ ნაწილად: სია, რომელიც შედგება xs–ის ელემენტებისგან, რომლებიც ნაკლებია x–ზე და სია ელემენტებისგან, რომლებიც მეტია x–ზე. ეს სიები დახარისხდება (აქ ვლინდება რეკურსია, ვინაიდან ისინი ხარისხდება იმავე ალგორითმით), ხოლო შემდეგ მათგან დგება შემდეგი სახის საშედეგო სია: $as++[x]++bs$, სადაც as და bs არის შესაბამისად პატარა და დიდი ელემენტების დახარისხებული სიები.

5) განსაზღვრეთ წინა პუნქტში მოყვანილი ფუნქცია $quicksort$, რომელიც სიას დაალაგებს ზრდის მიხედვით. განაზოგადეთ იგი: ვთქვათ იგი ღებულობს კიდევ ერთ არგუმენტს --- შემდეგი ტიპის შედარების ფუნქციას $a- > a- > Bool$ და ახარისხებს სიას მის შესაბამისად.

2. დაუბრუნდით ლაბორატორიული მეცადინეობა $N3$–ის დავალებებს და შეასრულეთ ისინი უმაღლესი რიგის ფუნქციების გამოყენებით. შეეცადეთ მთლიანად ამოადგოთ ფუნქციის განსაზღვრებიდან სიაზე ცხადი გავლა.

ლექცია № 6

1 მოდულები

პროგრამა ენა *Haskell*–ზე შედგება მოდულებისგან. მოდულები ორ მიზანს ემსახურება – სახელთა არის მართვას და მონაცემთა აბსტრაქტული ტიპების შექმნას. მოდულებს აქვთ სახელი, რომელიც იწყება დიდი ასოთი. ინტერპრეტატორ *Hugs* –ში მოდულის ტექსტი უნდა იყოს ცალკე ფაილში, რომლის სახელი უნდა ემთხვეოდეს მოდულის სახელს. ამ ფაილს უნდა ჰქონდეს გაფართოება *.hs*.

პრაქტიკულად, მოდული წარმოადგენს ერთ დიდ განაცხადს, რომელიც იწყება გასაღები სიტყვით *module*. მოვიყვანოთ მაგალითად მოდული, სახელად *Tree*.

```
module Tree ( Tree(Leaf,Branch), leafList) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

leafList (Leaf x)            =  [x]
leafList (Branch left right) =  leafList left ++
                                leafList right
```

ტიპი *Tree* და ფუნქცია *leafList* შემოვიტანეთ ლაბორატორიულ მეცადინეობაზე 4.

მოდული ცხადად ექსპორტირებს $Tree, Leaf, Branch$ და $leafList$. მოდულიდან ექსპორტირებული სახელები გასაღები სიტყვა *module*–ის შემდეგ ფრჩხილებში. ეს ჩამოთვლა მითითებული არ არის, მაშინ შეთანხმების პრინციპით ექსპორტირდება ყველა სახელი. შევნიშნოთ, რომ ტიპისა და მისი კონსტრუქტორი სახელები უნდა იყოს დაჯგუფებული, როგორც კონსტრუქციაში $Tree(Leaf, Branch)$. შემოკლების მიზნით შეიძლება გამოიყენოთ ჩანაწერი $Tree(..)$. ასევე, შესაძლოა ექსპორტირდეს მონაცემთა კონსტრუქტორების მხოლოდ ნაწილი.

მოდული *Tree* ეხლა შეიძლება იყოს იმპორტირებული რომელიღაც სხვა მოდულში:

```
module Main where
import Tree (Tree(Leaf,Branch), leafList)

...
```

აქ ჩვენ ცხადად მივუთითეთ იმპორტირებულების სია. თუ მას გამოვტოვებთ, მაშინ იმპორტირდება ყველაფერი, რაც იყო ექსპორტირებული მოდულიდან. ცხადია, რომ თუ ორი იმპორტირებული მოდული შეიცავს სხვადასხვა ცნებას ერთიდაიგივე სახელით, მაშინ ჩნდება პრობლემა. ამ პრობლემის თავიდან ასაცილებლად ენაში არსებობს გასაღები სიტყვა *qualified*, რომლის საშუალებით განისაზღვრება ის იმპორტირებული მოდულები, რომელთა ობიექტის სახელები ღებულობს სახეს: „მოდული. ობიექტი“. მაგალითად, მოდულისთვის *Tree*:

```
module Main where
import qualified Tree

leafList = Tree.leafList
```

2 მონაცემთა აბსტრაქტული ტიპები

მოდულების გამოყენება საშუალებას გვაძლევს განვსაზღვროთ მონაცემთა აბსტრაქტული ტიპები, ანუ, ტიპები, რომელთა შიდა სტრუქტურა დამალულია მომხმარებლისგან. მაგალითად, განვიხილოთ მარტივი ლექსიკონი, რომელიც მოცემული სიტყვის მიხედვით აბრუნებს მის მნიშვნელობას:

```
module Dictionary where
data Dictionary = Dictionary [(String,String)]

getMeaning :: Dictionary -> String -> Maybe String
getMeaning [] _ = Nothing
getMeaning ((word,meaning):xs) w | w == word = Just meaning
                                 | otherwise = Nothing
```

ფუნქცია *getMeaning* მოცემული ლექსიკონისა და სიტყვის მიხედვით აბრუნებს ნაპოვნ მნიშვნელობას (იყენებს რა ტიპს *Maybe*). თვითონ ლექსიკონი მოცემულია წყვილების მიხედვით.

როგორ შევქმნათ ლექსიკონი? ამ მოდულის მომხმარებელს შეუძლია განსაზღვროს *addWord*, რომელიც ლექსიკონში ამატებს წყვილს „სიტყვა–მნიშვნელობა“ და აბრუნებს მოდიფიცირებულ ლესიკონს.

```
import Dictionary
addWord (Dictionary dict) word meaning = Dictionary ((word,meaning):dict
```

აქ მომხმარებელი ლექსიკონს ხედავს როგორც სიას და ამით სარგებლობს. შემდეგომში შეიძლება მოგვინდეს ლექსიკონის შეცვლა. სია – მონაცემების საკმაოდ არაეფექტური სტრუქტურაა ძებნისთვის, თუ ის ხდება დიდი. გაცილებით უკეთესია ჰეშ–ცხრილების ან ძებნის ხეების გამოყენება. თუმცა, თუ *Dictionary*–ის ტიპის წარმოდგენა არის ღია, ჩვენ ვერ შევძლებთ შევცვალოთ რისკის გარეშე მომხმარებლის პროგრამის ფუნქციონირება.

გავხადოთ ტიპი *Dictionary* აბსტრაქტული, რათა მოდულის მომხმარებლისგან დავმალოთ მისი შიდა წარმოდგენა. განვსაზღროთ მოდულში მნიშვნელობა *emptyDict*, რომელიც წარმოადგენს ცარიელ ლექსიკონს და ფუნქცია *addWord*. მაშინ მომხმარებელს შეუძლია მიმართოს *Dictionary*–ის ტიპის მნიშვნელობას მხოლოდ დაშვებული ფუნქციებით:

```
module Dictionary (Dictionary, getMeaning, addWord, emptyDict) where
data Dictionary = Dictionary [(String,String)]

getMeaning :: Dictionary -> String -> Maybe String
getMeaning [] _ = Nothing
getMeaning ((word,meaning):xs) w | w == word = Just meaning
                                 | otherwise = Nothing

addWord (Dictionary dict)  word meaning = Dictionary ((word,meaning):dic

emptyDict = Dictionary []
```

მონაცემთა აბსტრაქტული ტიპი წარმოადგენს მონაცემთა დამალვის მექანიზმს, რომელსაც ობიექტ–ორიენტირებული პროგრამირების ენებში უწოდებენ ინკაპსულაციას.

3 ტიპების სინონიმები

ენაში არის შესაძლებლობა განისაზღვროს ტიპების სინონიმები, ანუ ხშირად გამოყენებული ტიპების სახელები. ისინი იქმნება

გასაღები სიტყვა *type*–ით. მოვიყვანოთ რამდენიმე მაგალითი:

```
type String = [Char]
type Person = (Name, Address)
type Name = String
type Address = Maybe String
```

ტიპების სინონიმები ახალ ტიპებს არ განსაზღვრავს, მხოლოდ ახალ სახელს არქმევს არსებულ ტიპს. მაგალითად, ტიპი $Person -> Name$ ექვივალენტურია ტიპის $(String, MaybeString) -> String$. თუმცა მათ იყენებენ, ვინაიდან, ჯერ ერთი ისინი გვეხმარებიან მივცეთ ტიპებს მოკლე სახელები, და მეორეც, ამაღლებენ კოდის გაგების დონეს.

4 შეტანა-გამოტანის ოპერაციები

შეტანა–გამოტანის სისტემა *Haskell*–ში სრულად ფუნქციონირებს და არ ჩამოუვარდება შეტანა–გამოტანის სისტემას იმპერატიულ პროგრამირებაში. იმპერატიულ ენებში პროგრამა წარმოადგენს მოქმედებების თანმიმდევრობას. ტიპიური მოქმედებაა წაკითხვა და გლობალური ცვლადების განსაზღვრა, ფაილში ჩაწერა, კლავიატურიდან წაკითხვა და ა.შ. ასეთი მოქმედებები *Haskell*–ის ნაწილიცაა, თუმცა ისინი მკვეთრად გამოიყოფა ენის ფუნქციონალური ბირთვისგან. შეტანა–გამოტანის სისტემა *Haskell*–ში აგებულია მონადის კონცეფციის გარშემო. თუმცა შეტანა–გამოტანის პროგრამირებისთვის მონადის გაგება საჭიროა არაუმეტეს, ვიდრე ზოგადი ალგებრის ცოდნა ელემენტარული არითმეტიკული ოპერაციების შესასრულებლად. ამიტომ ჩვენ განვიხილავთ შეტანა–გამოტანის სისტემას მონადებთან მიბმის გარეშე. *Haskell* ენის საშუალებით მოქმედება განისაზღვრება, მაგრამ არ შესრულდება. მოქმედების განსაზღვრა არ ნიშნავს, რომ ის სრულდება. მოქმედების შესრულება ხდება გამოსახულების გამოთვლის გარეთ. მოქმედება არის ან ატომარული, რომელიც განისაზღვრება სისტემური პრიმიტივების საშუალებით, ანდა თანმიმდევრიული კომპოზიციები სხვა მოქმედებებისა. შეტანა–გამოტანის მონადა შეიცავს პრიმიტივებს, რომლებიც საშუალებას იძლევა შევქმნათ შედგენილი მოქმედება, ანალოგიურად, იმპერატიულ ენებში ′;′ –ის გამოყენებისას. მონადა როგორც „წებო“ უკავშირებს მოქმედებებს პროგრამაში.

4.1 შეტანა-გამოტანის ბაზური ოპერაციები

თითოეული მოქმედება აბრუნებს მნიშვნელობას. ტიპების სისტემაში ეს მნიშვნელობა „მონიშნულია“ ტიპით *IO*,რომელიც გაარჩევს მოქმედებას სხვა ოპერაციებისგან. მაგალითად, განვიხილავთ ფუნქციას *getChar*:

```
getChar :: IO Char
```

IO Char გვიჩვენებს, რომ *getChar* გამოძახებისას რაღაც მოქმედებას ასრულებს, რომელიც გვიბრუნებს სიმბოლოს. მოქმედებები, რომლებიც არ აბრუნებენ შედეგს, იყენებენ ტიპს *IO* (). სიმბოლო () აღნიშნავს ცარიელ ტიპს (მსგავსია ტიპი *void*–ის ენა –ში). მაგალითად, ფუნქცია *putChar*:

```
putChar :: Char -> IO ()
```

ის იღებს სიმბოლოს და არაფერს საინტერესოს არ აბრუნებს.

მოქმედებები ერთმანეთს უკავშირდება ოპერატორის >>= საშუალებით. თუმცა ჩვენ გამოვიყენებთ ე.წ. *do*-ნოტაციას. გასაღები სიტყვა *do* იწყებს ოპერატორების თანმიმდევრობას, რომლებიც სრულდება რიგის მიხედვით. ოპერატორი შეიძლება იყოს ან მოქმედება, ან ნიმუში, რომელიც დაკავშირებულია მოქმედებასთან <–ის საშუალებით. *do*-ნოტაცია იყენებს გასწორების იმავე წესებს, როგორსაც გასაღები სიტყვა *let* ან *where*. აი მარტივი პროგრამა, რომელიც კითხულობა სიმბოლოს და ბეჭდავს მას:

```
main :: IO ()
main = do c <- getChar
          putChar c
```

სახელი *main* აქ შემთხვევით არაა: ფუნქცია *main* მოდული *Maindan* წარმოადგენს პროგრამის საწყის წერტილს *Haskell* ენაზე, მსგავსად ფუნქცია *main*–ისა –ში. მისი ტიპი უნდა იყოს *IO* (). წარმოდგენილი პროგრამა ასრულებს თანმიმდევრულად ორ მოქმედებას: კითხულობს სიმბოლოს, კავშირებს შედეგს ცვლად *c*–ს და შემდეგ ბეჭდავს სიმბოლოს. როგორ დავაბრუნოთ მოქმედებათა თანმიმდევრობის მნიშვნელობა? მაგალითად, საჭიროა განვსაზღვროთ ფუნცია *ready*, რომელიც კითხულობს სიმბოლოს და აბრუნებს *True*–ს, თუ იგი ტოლია $'y'$–ის:

```
ready :: IO Bool
ready = do c <- getChar
           c == 'y' -- !!!
```

ეს არ მუშაობს იმიტომ, რომ მეორე ოპერატორი *do*–ში არის ბულის მნიშვნელობა და არა მოქმედება. ჩვენ უნდა ავიღოთ ბულის მნიშვნელობა და შევქმნათ მოქმედება, რომელიც არაფერს აკეთებს, მაგრამ აბრუნებს ამ ბულის მნიშვნელობას, როგორც შედეგი. ამას ემსახურება ფუნქცია *return*:

```
return :: a -> IO a
```

ფუნქაცია *return* ამთავრებს მოქმედებების თანმიმდევრობას. ამრიგად, *ready* განისაზღვრება ასე:

```
ready :: IO Bool
ready = do c <- getChar
           return (c == 'y')
```

ეხლა განვსაზღვროთ შეტანა–გამოტანის უფრო რთული ფუნქციები. ფუნქცია *getLine*, აბრუნებს სტრიქონს, რომელსაც კითხულობს კლავიატურიდან სტრიქონის დამამთავრებელ სიმბოლომდე:

```
getLine :: IO String
getLine = do c <- getChar
             if c == '\n'
                then return ""
                else do l <- getLine
                        return (c:l)
```

ფუნქცია *return*–ს შეყავს ჩვეულებრივი მნიშვნელობა შეტანა–გამოტანის მოქმედებაში. ისმის შეკითხვა, შეიძლება თუ არა პირიქით, შეტანა–გამოტანის ოპერაცია შესრულდეს გამოსახულებაში? აღმოჩნდა, რომ არა. ისეთი ფუნქციას, როგორიცაა $f :: Int -> Int$ არ შეუძლია შეასრულოს შეტანა–გამოტანის ოპერაცია, ვინაიდან *IO* არ არის დასაბრუნებელი მნიშვნელობის ტიპი.

4.2 შეტანა-გამოტანის სტანდარტული ოპერაციები

განვიხილოთ შემდეგი მოქმედებები და ტიპები ფაილურ შეტანა–გამოტანასთან სამუშაოდ (ისინი განსაზღვრულია მოდულში *IO*):

```
type FilePath = String   --
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

ფაილის გახსნისთვის გამოიყენება ფუნქცია *openFile*, რომელსაც გადაეცემა ფაილის სახელი და რეჟიმი, რომელშიც იგი უნდა გაიხსნას. ამასთან, იქმნება ფაილის დესკრიპტორი (ტიპი *Handle*), რომელიც აუცილებელია შემდგომ დაიხუროს ფუნქციის *hClose*–ის საშუალებით. ფაილიდან სიმბოლოს და სტრიქონის წასაკითხად გამოიყენება ფუნქციები:

```
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
```

ფაილში ჩასაწერად გამოიყენება ფუნქციები:

```
hPutChar :: Handle -> Char -> IO ()
hPutStr  :: Handle -> String -> IO ()
```

კლავიატურიდან წასაკითხად და ეკრანზე გამოსატანად გამოიყენება შემდეგი ფუნქციები:

```
getChar :: IO Char
getLine :: IO String
putChar :: Char -> IO ()
putStr  :: String -> IO ()
```

ამას გარდა, ძალზე სასარგებლოა შემდეგი ფუნქციები:

```
hGetContents :: Handle -> IO String
```

ის კითხულობს მთლიან ფაილს როგორც ერთ დიდ სტრიქონს. ერთი შეხედვით, ეს ფუნქცია ძალზე არაეფექტურია, თუმცა სინანდვილეში, იმის გამო, რომ იყენებს გადატანილ გამოთვლებს, ფაილიდან წაიკითხება იმდენი სიმბოლო, რამდენიც აუცილებელია, და მეტი არა.

4.3 მაგალითები

დავწეროთ ფაილების ასლის გადამღები (კოპირების) პროგრამა. ის კითხულობს კლავიატურიდან ორი ფაილის სახელს, საწყისისა და მიზნობრივს, და ერთი ფაილის ასლი გადააქვს მეორეში. -- ფუნქცია ბეჭდავს მოწვევას, კითხულობს ფაილის სახელს -- და ხსნის მას მითითებულ რეჟიმში

```
--   ,
--
getAndOpenFile prompt mode = do putStr prompt
```

```
                                    name <- getLine
                                    openFile name mode

main = do fromHandle <- getAndOpenFile "Copy from: " ReadMode
          toHandle <- getAndOpenFile "Copy to" WriteMode
          contents <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose toHandle
          putStr "Done."
```

მიუხედავად იმისა, რომ ვიყენებთ ფუნქცია *hGetContents*, ფაილის მთელი შინაარსი არ იქნება მეხსიერებაში, ვინაიდან იგი წაიკითხება საჭიროების მიხედვით და ჩაიწერება დისკზე. ეს საშუალებას იძლევა მოვახდინოთ ისეთი დიდი ფაილების კოპირება, რომელთა მოცულობა აჭარბებს კომპიუტერის ოპერატიული მეხსიერების მოცულობას. საწყისი ფაილი არაცხადად დაიხურება, როცა მოხდება მისგან ბოლო სიმბოლოს წაკითხვა. ბრძანების სტრიქონის პარამეტრებთან შეღწევადობისთვის პროგრამა იყენებს შემდეგ ფუნქციას, რომელიც განსაზღვრულია მოდულში *System*:

```
getArgs :: IO [String]
```

ეს ფუნქცია აბრუნებს სტრიქონების სიას, რომელიც წარმოადგენს ბრძანების სტრიქონის პარამეტრებს, მსგავსად მასივისა *argv* –ის პროგრამებში. მაშინ კოპირების პროგრამა შეიძლება ასე განვსაზღვროთ:

```
main = do args <- getArgs
          copyFile
          putStr "Done."

copyFile [from, to] = do fromHandle <- openFile from ReadMode
                         toHandle <- openFile to WriteMode
                         contents <- hGetContents fromHandle
                         hPutStr toHandle contents
                         hClose toHandle
copyFile _ = error "Usage: copy <from> <to>"
```

ეს პროგრამა იღებს საწყისი და მიზნობრივი ფაილების სახელებს ბრძანების სტრიქონიდან. ფუნქცია *copyFile* ბეჭდავს შეტყობინებას შეცდომის შესახებ, თუ პროგრამას გადაეცა არგუმენტების არასწორი რაოდენობა.

5 შესასრულებელი პროგრამის შექმნა

აქამდე ჩვენ ვასრულებდით ენა *Haskell*–ზე დაწერილ პროგრამებს ინტერპრეტატორის გამოყენებით. თუმცა არსებობს შესაძლებლობა შევქმნათ ცალკეული პროგრამები, რომელთა შესრულებას არ სჭირდება ინტერპრეტატორის გარემო. ამისთვის გამოვიყენოტ კომპილერი Glasgow Haskell Compiler, რომელიც გამოიძახება ბრძანებით *ghc*. იმისათვის, რომ მოდელების ნაკრები კომპილირდეს შესასრულებელ პროგრამაში, უნდა იყოს განსაზღვრული მოდული სახელით *Main*, რომელშიც აუცილებელია განსაზღვრული იყოს ფუნქცია $main :: IO()$. ეს მოდული უნდა მოვათავსოთ ფაილში *Main.hs*. კომპილაციისთვის ბრძანებათა სტრიქონში უნდა შევიტანოთ შემდეგი ბრძანება: $ghc --makeMain.hs$ იმ შემთხვევაში, თუ პროგრამა სეიცავს შეცდომებს, ინფორმაცია მათ შესახებ გამოვა ეკრანზე. თუ შეცდომები არაა, კომპილერი შექმნის შესასრულებელ ფაილს, რომელიც უნდა გავუშვათ შესასრულებლად.

6 დავალებები

1. დაწერეთ შემდეგი პროგრამა:

1) პროგრამა კითხულობს ორ რიცხვს და აბრუნებს მათ ჯამს.

2) პროგრამა ბეჭდავს მასზე გადაცემულ ბრძანების სტრიქონის არგუმენტებს.

3) პროგრამა, რომელიც იღებს ბრძანებათა სტრიქონში ფაილის სახელს და ბეჭდავს მას ეკრანზე.

4) პროგრამა, რომელიც იღებს ბრძანებათა სტრიქონში რიცხვ n–ს და ფაილის სახელს და ეკრანზე გამოაქვს ფაილის პირველი n სტრიქონი (გამოიყენეთ ფუნქცია *lines*, რომელიც ყოფს სტრიქონს სტრიქონების სიად სტრიქონის ბოლოს სიმბოლოთი, ანუ, მაგალითად, $lines"line1\ nline2"$ დააბრუნებს $["line1","line2"]$. ასევე სასარგებლოა ფუნქცია უნლინეს, რომელიც ასრულებს ოპერაციას პირიქით).

2. შეასრულეთ პროგრამა პირველი ლაბორატორიული მეცადინეობიდან. ფუნქციის პარამეტრები უნდა გადასცეთ კლავიატურიდან.

1. შესავალი.

1930 წელს ***ალონზო ჩერჩმა*** (Alonzo Church) განავითარა *ლამბდა კალკულისი,* როგორც ფუნქციათა თეორია, რომელიც შეიცავს წესებს ფუნქციების მანიპულაციისათვის სუფთა სინტაქსური მანერით.

მიუხედავა იმისა, რომ ლამბდა კალკულუსი აღმოცენდა როგორც მათემატიკური ლოგიკის მიმართულება მათემატიკის საფუძვლებისათვის, მან მიიყვანა შესანიშნავ განშტოებამდე პროგრამული ენების თეორიისათვის.

ყველა ფუნქციონალური პროგრამული ენები შეიძლება წარმოვიდგინოთ როგორც ლამდა კალკულუსის სინტაქსური ვერსია ისეთნაირად, რომ როგორც მათი სემანტიკები ასევე მათი გამოყენებები შესაძლოა გაანალიზდეს ლამბდა კალკულუსის კონტექსტში.

ამ კალკულუსს შეიძლება დაერქვას მსოფლიოში უმცირესი პროგრამული ენა. ეს კალკულუსი შედგება ერთადერთი გარდამქნელი წესის (ცვლადების ჩასმის) და ერთადერთი ფუნქციური აღნიშვნის სქემით. ეს შემოიღო **ალონზო ჩერჩმა** , როგორც ეფექტური გამოთვლის ცნების ფორმალიზაციის გზა. ეს კალკულუსი არის უნივერსალური იმ აზრით, რომ ყოველი გამოთვლადი ფუნქცია შეიძლება გამოისახოს და შეფასდეს ამ ფორმალიზმის გამოყენებით. მაშასადამე ეს ექვივალენტურია **ტიურინგის მანქანებისა** (Alan Turing).

ფუნქციის ასეთნაირი განსაზღვრიდან წარმოიქმნება რამოდენიმე კითხვები:

• რა არის იდენტიფიკატორი “cube”-ის მნიშვნელობა?

• როგორ უნდა წარმოვადგინოთ ობიექტი, რომელიც დაკავშირებულია “cube”-თან?

• შეიძლება თუ არა ეს ფუნქცია განსაზღვრულ იქნას მისი სახელის მინიჭების გარეშე?

ჩერჩის ლამბდა აღნიშვნა იძლევა საშუალებას განისაზღვროს ანონიმური ფუნქციები, ე. ი. ფუნქციები სახელის გარეშე:

- **λn.n³** განსაზღვრვს ფუნქციას, რომელიც ასახავს ყოველ n-ს დომეინიდან n^3-ში.

ვიტყვით, რომ გამოსახულება, რომელიც წარმოდგენილია **λn.n³** -ით, არის მნიშვნელობა დაკავშირებული იდენტიფიკატორ "cube"-თან. ფუნქციის პარამეტრების რიცხვი და რიგი გამოსახულია λ სიმბოლოსა და გამოსახულების შორის.

მაგალითად, გამოსახულება n^2+m ორაზრო-ვანია ფუნქციის წესის განსაზღვრის თანახმად :

$(3,4) \mid\rightarrow 3^2+4 = 13$ ან $(3,4) \mid\rightarrow 4^2+3 = 19$.

ლამბდა აღრიცხვა წყვეტს ამ ორაზროვნებას პარამეტრების რიგის ჩვენებით:

$$\lambda\ n.\lambda\ m.n^2+m \quad \text{ან} \quad \lambda\ m.\lambda\ n.n^2+m.$$

ფუნქციონალური პროგრამირების ენების უმრავლესობა იძლევა საშუალებას ანონიმური ფუნქციების გამოყენებას როგორც მნიშვნელობებისა;

მაგალითად ფუნქცია $\lambda\ n.n^3$ წარმოდგენილია როგორც

- (lambda (n) (* n n n)) სქემის სახით და
- fn n => n*n*n სტანდარტულ მათემატიკურ ენაში

2. ლამბდა კალკულუსის სინტაქსი

ლამბდა კალკულუსს გააჩნია საკმარისი სიძლიერე იმისათვის, რომ წარმოადგინოს ყველა გამოთვლითი ფუნქციები. ლამბდა გამოსახულებები წარმოგვიდგება ოთხ მრავალსახეობაში:

1. ცვლადები, რომლებიც ჩვეულებრივ წარმოდგენილია ნებისმიერი პატარა ასოებით.

2. წინასწარ განსაზღვრული კონსტანტები, რომლებიც მოქმედებენ როგორც მნიშვნელობები და ოპერაციები, რომლებიც დაშვებულია სუფთა და **გამოყენებით ლამბდა კალკულუსში .**

3. ფუნქციური გამოყენებები (კომბინაციები).

4. ლამბდა აბსტრაქციები (ფუნქციური განსაზღვრები).

<expression> ::= <variable> ; იდენტიფიკატორები პატარა ასოებით

| <constant> ; წინასწარ განსაზღვრული ობიექტები

| (<expression> <expression>) ; კომბინაციები

| (<variable> . <expression>) ; აბსტრაქციები.

კალკულუსის ცენტრალური ცნება არის ”**expressio**n” (გამოსახულება).

”**name**“ (სახელი), რომელსაც აგრეთვე ეწოდება ”variable“ (ცვლადი) , არის იდენტი-ფიკატორი, რომელიც ჩვენი მიზნებისთვის არის ასოები a, b, c, ... (ან x, y, z)

გამოსახულება განისაზღვრება რეკურსი-ულად შემეგნაირად:

- <expression>:=<name>|<function>| <application>

- <function> := λ <name>.<expression>
- <application> := <expression><expression>

ფუნქციის მაგალითს წარმოადგენს:

$$\lambda x.x$$

ეს გამოსახულება განსაზღვრავს იგივურ ფუნქციას. სახელი λ-ს შემდეგ არის ფუნქციის არგუმენტის იდენტიფიკატორი. გამოსახულებას წერტილის შემდეგ (ამ შემთხვევაში ერთადერთი x) ეწოდება განსაზღვრის ”ტანი” (**”body"**).

ფუნქციები გამოიყენება გამოსახულებებზე. გამოყენებას (ქმედება) (application) წარმოადგენს

$$(\lambda x.x)y$$

ეს არის იგივური ფუნქცია გამოყენებული y-ზე.

ფუნქციების **გამოყენებები (ქმედებები)** განისაზღვრება არგუმენტ x-ის ჩანაცვლებით (ამ შემთხვევაში y-ით) ფუნქციის განსაზღვრის ტანში, ე. ი.

$$(\lambda x.x)y = [y/x]x = y$$

ამ გარდაქმნაში აღნიშვნა [y/x] გამოიყენება იმის საჩვენებლად, რომ x-ის ყველა შემავლობა ჩანაცვლებულია y-ით გამოსახულებაში მარჯვნივ.

არგუმენტების სახელები ფუნქციის განსაზღვრაში თავის მხრივ არ ატარებენ არავითარ აზრს. ესენი არიან მხოლოდ ”ადგილის დამკავებლები”, ე. ი. ესენი გამოიყენება იმისათვის, რომ ვუჩვენოთ როგორ შეგვიძლია არგუმენტების ჩანაცვლება მათი შეფასებისას. ამიტომ

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

3. შეკუმშვა (Reduction)

ფუნქციონალური პროგრამა შედგება E გამოსახულებებისგან (რომლებიც წარმოადგენენ ალგორითმს ან შესავალს). ეს გამოსახულება E აკმაყოფიკლებს ზოგიერთ გარდაქმნის წესებს.

შეკუმშვა (reduction) შედგება E -ს P ნაწილის ჩანაცვლებით სხვა P'-ის გამოსახულებით მოცემული გარდაქმნის წესების თანახმად. სქემატური ჩანაწერით

$$E[P] \to E[P'],$$

იმ პირობით, რომ $P \to P'$ თანახმად მოცემული წესებისა.

შეკუმშვის ეს პროცესი მეორდება მანამდე სანამ არ მივიღებთ გამოსახულებას, რომელზედაც შეკუმშვა ვერ გამოიყენება. ეს ეგრეთ წოდებული E გამოსახულების ნორმალური ფორმა E^* შედგება ფუნქციონალური პროგრამის გამოსავალისგან.

მაგალითი:

$$(7 + 4)*(8 + 5 * 3) \to 11 * (8 + 5 * 3)$$

$$\to 11 * (8 + 15)$$

$$\to 11 * 23$$

$$\to 253.$$

ამ მაგალითში შეკუმშვის წესები შედგება რიცხვების ჯამის და ნამრავლის 'ცხრილებისგან'.

4. გამოყენებები (ქმედებები) და აბსტრაქციები

პირველი ძირითადი ოპერაცია λ-კალკულუსში არის **ქმედება** (**გამოყენება**) (application). გამოსახულება

$$F\,.\,A$$

ან

$$FA$$

აღნიშნავს მონაცემ *F*-ს განხილულს როგორც ალგორითმს გამოყენებულს მონაცემ *A*-ზე განხილულს როგორც შესავალს.

ჩვენ შეგვიძლია განვიხილოთ გამოსახულება *F F,* სადაც *F* გამოიყენება თავის თავზე.

მეორე ძირითადი ოპერაცია არის ***აბსტრაქცია*** (abstraction). თუ M ≡ M[x] არის გამოსახულება, რომელიც შეიცავს (’დამოკიდებულია’) x (-ზე), მაშინ λx.M[x] აღნიშნავს ფუნქციას $x \to M[x]$.

ქმედება და აბსტრაქცია მუშაობს ერთად შემდეგი ინტუიციურ ფორმულაში.

$$(\lambda x.2*\ x + 1)3 = 2*\ 3 + 1\ (= 7).$$

ე. ი. , (λx.2* x + 1)3 აღნიშნავს ფუნქციას $x \to 2 * x + 1$ გამოყენებულს არგუმენტ 3 -ზე, რომელიც გვაძლევს 2*3+1, რაც უდრის 7. ზოგადად ჩვენ გვაქვს (λx.M[x])N = M[N].

ბოლო ტოლობა მოხერხებულია ჩაიწეროს როგორც

(λx.M)N = M[x := N], (β)

სადაც [x := N] აღნიშნავს N ის ჩანაცვლებას x-ის ნაცვლად.

5. თავისუფალი და ბმული ცვლადები

ვიტყვით, რომ x ”ბმულია” ფუნქცია λ x.x -ში ვინაიდან მისი შემავლობა განსაზღვრის ტანში წინსწრებია λ x -ით. სახელს, რომლის წინ არ გვხდება λ ეწოდება ”თავისუფალი ცვლადი”. გამოსახულებაში

(λ x .x y)

ცვლადი x ბმულია, ხოლო y თავისუფალი. გამოსახულებაში

(λx.x)(λy.yx).

x -ი ტანის პირველ გამოსახულებაში მარცხნიდან პირველ λ-დან არის ბმული. y - ი ტანის მეორე გამოსახულებაში λ-დან არის ბმული, ხოლო x თავისუფალი.

- მნიშვნელოვანია აღვნიშნოთ, რომ x მეორე გამოსახულებაში სრულიად დამოუკიდებელია x -გან პირველი გამოსახულებიდან.

ფორმალურად ვიტყვით, რომ ცვლადი <name> თავისუფალია რომელიღაც გამოსახულებაში, თუ ერთერთი შემდეგი სამი შემთხვევიდან სრულდება:

- <name> თავისუფალია <name> -ში.
- <name> თავისუფალია λ<name1 > . <exp> -ში, თუ იდენტიფიკატორი <name>≠<name$_1$ > და <name> თავისუფალია <exp> -ში.
- <name> თავისუფალია E_1E_2 , თუ <name> თავისუფალია E_1 ან თუ ის თავისუფალია E_2-ში.

აღსანიშნავია, რომ ერთიდაიგივე იდენტიფიკატორი შესაძლოა იყოს როგორც თავისუფალი ასევე ბმული ერთდაიგივე გამოსახულებაში.

გამოსახულებაში

$$(\lambda x.xy)(\lambda y.y)$$

პირველი y თავისუფალია ფრჩხილებში მოთავსებული პირველ (მარცხენა) გამოსახულებაში. ის ბმულია ქვეგამოსახულებაში მარჯვნიდან.

6. ჩანაცვლება

იგივური ფუნქცია შეგვიძლია აღვნიშნოთ I- ით, რომელიც იგივეა რაც $(\lambda x.x)$.

თავის თავზე გამოყენებული იგივური ფუნქცია არის გამოსახულება

$$II \equiv (\lambda x.x)(\lambda x.x)$$

ამ გამოსახულებაში პირველი x პირველ გამოსახულებაში დამოუკიდებელია x-გან მეორე გამოსახულებიდან. გადავწეროთ ზემოთმოყვანილი გამოსახულება შემდეგნაირად

$$II \equiv (\lambda x.x)(\lambda z.z)$$

იგივური ფუნქცია გამოყენებული თავის თავზე

$$II \equiv (\lambda x.x)(\lambda z.z).$$

ამიტომ გვაძლევს

$$[\lambda z.z/x]x = \lambda z.z \equiv I$$

ე. ი. ისევ იგივურ ფუნქციას.

უნდა ვიყოთ ფრთხილი, როცა ვაწარმოებთ ჩანაცვლებას, რათა ავიცილოთ თავიდან თავისუფალი ცვლადების ჩანაცვლება ბმულებით. გამოსახულებაში

$$(\lambda x.(\lambda y.xy))y.$$

ფუნქცია მარცხნივ შეიცავს ბმულ y -ს, როდესაც y მარჯვნივ არის თავისუფალი.

არასწორი ჩანაცვლება (თუ არ განვმიჯნეთ თავისუფალი და ბმული ცვლადები) გვაძლევს მცდარ შედეგს

$$(\lambda y.\ y\ y).$$

უბრალოდ, თუ შევუცვალეთ სახელი ბმულ y-ს t-თი, მაშინ ვღებულობთ

$$(\lambda x.(\lambda t.xt))y = (\lambda t.yt),$$

რომელიც გვაძლევს განსხვავებულ შედეგს, მაგრამ მართებულს.

ამიტომ, თუ ფუნქცია λx. <exp> გამო-ყენებულია E-ზე, ჩვენ ვანაცვლებთ <exp> -ში x -ს ყველა *თავისუფალ* შემავლობას E -თი. თუ ჩანაცვლება იღებს E-ს თავისუფალ ცვლადს გამოსახულებაში, სადაც ეს ცვლა-დი შედის ბმულად, ჩვენ ვუცვლით სახელს ამ ბმულ ცვლადს სანამ განვახორციელებთ ჩანაცვლებას. მაგალითად, გამოსახულებაში

$$(\lambda x.(\lambda y.(x(\lambda x.xy))))y$$

ჩვენ განვიხილავთ არგუმენტ x-ს y-თან.

ტანში

$$(\lambda y.(x(\lambda x.xy)))$$

მხოლოდ პირველი x არის თავისუფალი და შეიძლება მისი ჩანაცვლება. მაგრამ ჩანაცვლებამდე ჩვენ უნდა გადავარქვათ სახელი ცვლად y -ს, რომ ავიცილოთ თავიდან ბმული ცვლადის აღრევა თავისუფალთან:

$$[y/x](\lambda t.(x(\lambda x.xt))) = (\lambda t.(y(\lambda x.xt))).$$

გამოსახულების ჩანაცვლება (თავისუფალი) ცვლადისთვის ლამბდა გამოსახულებაში აღინიშნება როგორც **E[v→E1]** და განისაზღვრება შემდეგნაირად:

- a) $v[v \to E_1] = E_1$ ნებისმიერი v ცვლადისთვის
- b) $x[v \to E_1] = x$ ნებისმიერი x ცვლადისთვის $x \neq v$
- c) $c[v \to E_1] = c$ ნებისმიერი c კონსტანტასათვის
- d) $(E_{rator}\ E_{rand})[v \to E_1] = ((E_{rator}[v \to E_1])(E_{rand}[v \to E_1]))$
- e) $(\lambda v\ .\ E)[v \to E_1] = (\lambda v\ .\ E_1)$
- f) $(\lambda x\ .\ E)[v \to E_1] = \lambda x\ .\ (E[v \to E_1])$ როცა $x \neq v$ and $x \notin FV(E_1)$
- g) $(\lambda x\ .\ E)[v \to E_1] = \lambda z\ .\ (E[x \to z][v \to E_1])$ როცა $x \neq v$ and $x \in FV(E_1)$, სადაც where $z \neq v$ and $z \notin FV(E\ E_1)$

7. λ-კალკულუსი

1. განსაზღვრა. λ-ტერმების სიმრავლე Λ აიგება ცვლადების უსასრულო სიმრავლის-გან $V = \{v, v', v'', ...\}$ ქმედებების და (ფუნქცი-ების) აბსტრაქციების გამოყენებით.

$$x \in V \Rightarrow x \in \Lambda,$$

$$M,N \in \Lambda \Rightarrow (MN) \in \Lambda,$$

$$M \in \Lambda, x \in V \Rightarrow (\lambda xM) \in \Lambda.$$

variable ::= `v' | variable `''

λ-term ::= variable |`(' λ-term λ-term `)' | `(λ' variable λ-term `)'

2. მაგალითი . შემდეგი გამოსახულებები წარმოადგენენ λ-ტერმებს.

$$v';$$

$$(v'v);$$

$$(\lambda v(v'v));$$

$$((\lambda v(v'v))v'');$$

$$(((\lambda v(\lambda v'(v'v)))v'')v''').$$

3. შეთანხმება. (i) x, y, z, ... აღნიშნავენ ნებისმიერ ცვლადს; M,N, L, ... აღნიშნავენ ნებისმიერ λ-ტერმს. ყველაზე უფრო დაშორებულ ფრჩხილებს არ ვწერთ.

(ii) $M \equiv N$ აღნიშნავს, რომ M და N არის ერთდაიგივე ტერმი ან მიიღება ერთი მეორესგან ბმული ცვლადების სახელების შეცვლით. ე. ი.

$$(\lambda xy)z \equiv (\lambda xy)z;$$

$$(\lambda xx)z \equiv (\lambda yy)z;$$

$$(\lambda xx)\, z \equiv z;$$

$$(\lambda xx) \equiv (\lambda xy)z:$$

(iii) ვიყენებთ შემოკლებებს

$$FM_1 \ \dots \ M_n \equiv (\dots((FM_1)M_2) \dots \ M_n)$$

და

$$\lambda x_1 \dots x_n.M \equiv \lambda x_1(\lambda x_2(\dots (\lambda x_n(M))\dots)).$$

ტერმები მეორე მაგალითიდან ეხლა ჩაიწერება შემდეგნაირად.

y;

yx;

$\lambda x.yx$;

$(\lambda x.yx)z$;

$(\lambda xy.yx)zw$.

შევნიშნოთ, რომ $\lambda x.yx$ არის $(\lambda x(yx))$ და არა $((\lambda x.y)x)$.

4. განსაზღვრა. (i) თავისუფალი ცვლადების სიმრავლე M-ში, აღნიშვნაში $FV(M)$, განისაზ-ღვრება ინდუქციურად შემდეგნაირად.

$$FV(x) = \{x\};$$

$$FV(MN) = FV(M) \cup FV(N);$$

$$FV(\lambda x.M) = FV(M) - \{x\}.$$

ცვლადი M-ში ბმულია, თუ ის არ არის თავისუფალი. შევნიშნოთ, რომ ცვლადი ბმულია თუ ის იმყოფება λ-ას არეში.

(ii) M არის *ჩაკეტილი* λ-ტერმი (ან კომბი-ნატორი), თუ $FV(M) = \varnothing$. ჩაკეტილ λ-ტერმთა სიმრავლე აღინიშნება Λ^0-თი.

(iii) შედეგი N-ის ჩანაცვლებით x-ის ნაცვლად M-ში, აღნიშვნაში

$M[x := N]$, განისაზღვრება შემდეგნაირად.

- $x[x := N] \equiv N$;
- $y[x := N] \equiv y$, თუ $x \equiv y$;
- $(M_1M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$;
- $(\lambda y.M_1)[x := N] \equiv \lambda y.(M_1[x := N])$.

7. განსაზღვრა. (i) λ-კალკულუსის მთავარი სქემა აქსიომა არის

$$(\lambda x.M)N = M[x := N] \qquad (\beta)$$

ნებისმიერი $M,N \in \Lambda$.

(ii) მას ემატება აგრეთვე ლოგიკური აქსიომები და გამოყვანის წესები.

ტოლობა :

- $M = M$;
- $M = N \Rightarrow N = M$;
- $M = N, N = L \Rightarrow M = L$

თავსებადობის წესები:

$M = M' \Rightarrow MZ = M'Z$;

$M = M' \Rightarrow ZM = ZM'$;

$M = M' \Rightarrow \lambda x.M = \lambda x.M'$.

(iii) თუ $M = N$ დამტკიცებადია , მაშინ ზოგჯერ ჩვენ ვწერთ `

$\lambda \vdash M = N$.