ნათელა არჩვაძე

ფუნქციონალური დაპროგრამება Haskell-ზე

ნათელა არჩვაძე

ფუნქციონალური დაპროგრამება Haskell-ზე

ფუნქციონალური დაპროგრამება

Haskell-ზე

ივანე ჯავახიშვილის სახელობის
თბილისის სახელმწიფო უნივერსიტეტი

ნათელა არჩვაძე

ფუნქციონალური დაპროგრამება Haskell-ზე

თბილისის უნივერსიტეტის გამომცემლობა

დღეისათვის `Haskell` წარმოადგენს ფუნქციონალური დაპროგრამების ყველაზე მძლავრ და დასრულებულ ინსტრუმენტს და ერთ-ერთი წამყვანი ენაა ფუნქციონალური დაპროგრამების შესასწავლად.

წინამდებარე სახელმძღვანელო შედგება შესავლისა და ორი ნაწილისაგან. პირველი ნაწილი `Haskell`-ზე პრაქტიკული დაპროგრამების შესწავლას ეხება; მეორე ნაწილში ფუნქციონალური დაპროგრამებისა და `Haskell`-ის ისეთი თეორიული საკითხებია განხილული, როგორებიცაა: პოლიმორფული ტიპები, ტიპების გამოყვანის მექანიზმი, ფუნქციათა კარირება, რეკურსიული ფუნქციების განსაზღვრა, გადატანილი - ზარმაცი გამოთვლები, მაღალი რიგის ფუნქციები და სხვა. თეორიული აღწერის გარდა, განსაკუთრებული ყურადღება ეთმობა პრაქტიკულ სავარჯიშოებს. მოყვანილია სავარჯიშოები და ამოცანები დამოუკიდებელი მუშაობისთვის და მათი ამოხსნის ნიმუშები.

წიგნი გათვალისწინებულია უნივერსიტეტის სტუდენტებისთვის, მკვლევრებისა და ყველა დაინტერესებული პირისთვის, რომელთაც შეუძლიათ გამოიყენონ აღნიშნული სისტემა სხვადასხვა ამოცანის ამოსახსნელად. ის დაეხმარება დაპროგრამებით დაინტერესებულ მკითხველთა ფართო წრეს; მისაღებია დამწყებთათვისაც, ვინაიდან არ მოითხოვს წინასწარ გამოცდილებას დაპროგრამების სფეროში.

წინამდებარე სახელმძღვანელო წარმოადგენს ლექციათა იმ კურსის შევსებულ და გადამუშავებულ ვარიანტს, რომელსაც ავტორი რამდენიმე წლის განმავლობაში კითხულობდა ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტში.

რედაქტორი პროფესორი კობა გელაშვილი

რეცენზენტები: პროფესორი მანანა ხაჩიძე
ასისტენტი პროფესორი ლიანა ლორთქიფანიძე

გამოცემულია ივანე ჯავახიშვილის სახელობის
თბილისის სახელმწიფო უნივერსიტეტის
საუნივერსიტეტო საგამომცემლო საბჭოს გადაწყვეტილებით.

ISBN 978-9941-13-709-9 (PDF)

შინაარსი

შესავალი

ფუნქციონალური პარადიგმის ქვაკუთხედს წარმოადგენს ფუნქცია. ცნება *ფუნქცია* ოთხასი წლის წინ დაიბადა და მას შემდეგ მათემატიკამ ფუნქციებთან ოპერირებისთვის გამოიგონა უამრავი თეორიული და პრაქტიკული აპარატი, დაწყებული ჩვეულებრივი დიფერენცირებიდან და ინტეგრირებიდან, დამთავრებული ფუნქციონალური ანალიზით, არამკაფიო სიმრავლეთა თეორიითა და კომპლექსური ცვლადის ფუნქციებით.

მათემატიკური ფუნქციები გამოხატავს კავშირს პროცესის პარამეტრებსა (შესასვლელი) და შედეგს (გამოსასვლელი) შორის. რადგან გამოთვლა ასევე პროცესია, რომელსაც აქვს შესასვლელი და გამოსასვლელი, ფუნქცია სავსებით შესაფერისი და ადეკვატური საშუალებაა გამოთვლების აღსაწერად. სწორედ ეს მარტივი პრინციპი არის ჩადებული ფუნქციონალური პარადიგმის საფუძველში და ფუნქციონალური სტილით დაპროგრამებაში.

ფუნქციონალური პროგრამა წარმოადგენს ფუნქციების განსაზღვრებების ერთობლიობას. ფუნქცია განისაზღვრება სხვა ფუნქციებით ან რეკურსიულად – თავისი თავით. პროგრამის შესრულების მომენტში ფუნქცია იღებს პარამეტრებს, ითვლის და აბრუნებს შედეგს, საჭიროების შემთხვევაში, ითვლის სხვა ფუნქციის მნიშვნელობებსაც. ფუნქციონალურ ენაზე დაპროგრამებისას პროგრამისტი არ აღწერს გამოთვლების თანმიმდევრობას. მისთვის აუცილებელია აღწეროს მხოლოდ სასურველი შედეგი ფუნქციების სისტემის სახით.

ფუნქციონალური დაპროგრამების აქტუალობა გამოიწვია იმან, რომ იგი ძალზე მოსახერხებელია კონკურენტული და პარალელური აპლიკაციების დასაწერად. ფუნქციონალურმა დაპროგრამებამ, ისევე როგორც ლოგიკურმა დაპროგრამებამ, დიდი გამოყენება პოვა ხელოვნურ ინტელექტსა და მის დანართებში. უნდა

აღინიშნოს, რომ ღრუბლოვანმა გამოთვლებმა ხელმისაწვდომი გახადა დიდი კომპიუტერული გამოთვლითი რესურსები, მაგრამ, ამასთან ერთად, მოიტანა მასშტაბურობის, წარმოებადობისა და პარალელიზმის მოთხოვნები, რასაც ობიექტებზე ორიენტირებული დაპროგრამება ვეღარ ერევა, განსაკუთრებით მაშინ, როდესაც საქმე ეხება კონკურენტულობასა და პარალელიზმს. ფუნქციონალური დაპროგრამება კარგად ესადაგება კონცეფციებს უცვლელი მდგომარეობების (Immutable state), დახურული (Closure) და მაღალი დონის (Higher-Order) ფუნქციების შესახებ, რომლებიც ძალზე შესაფერისია კონკურენტული და პარალელური აპლიკაციების დასაწერად.

ენა Haskell-ის დაბადების თარიღად მიიჩნევა 1987 წელი, როდესაც მკვლევართა საერთაშორისო კომიტეტი შეუდგა Haskell-ის, ფუნქციონალური დაპროგრამების სტანდარტული ზარმაცი ენის დამუშავებას, რომელსაც ეს სახელი ამერიკელი ლოგიკოსისა და მათემატიკოსის - ჰასკელ კარის (Haskell Curry, 1900-1982) პატივისცემის ნიშნად დაერქვა. 2003 წელს კომიტეტმა გამოაქვეყნა მოხსენება ჰასკელის შესახებ (Haskell Report), რომელმაც განსაზღვრა ამ ენის სტანდარტი. სტანდარტში ბოლო ცვლილებები 2014 წელს შევიდა.

გამოყენებული აღნიშვნები

Haskell-ის კოდის დაწერა შესაძლებელია ორი გზით: ინტერაქტიულად და საწყისი (source) ფაილის გამოყენებით. შემდგომ ჩვენ გავეცნობით ამ პროცესებს.

მოცემულ სახელმძღვანელოში შემოვიტანეთ შემდეგი აღნიშვნები:

1. თუ ჩვენ ვმუშაობთ ინტერპრეტატორში, იგივე REPL-ის გარემოში (Read-Eval-Print Loop, რაც ნიშნავს ციკლს: წაკითხვა-მნიშვნელობის გამოთვლა-დაბეჭდვა), მაშინ კოდის გარშემო გამოვიყენებთ მწვანე ფერის ფონს. მაგალითად, ინტერპრეტატორის დაყენებისა და გაშვების შემდეგ ჩვენ დავინახავთ შეტყობინებას ვერსიის შესახებ, განცხადებას და „მოწვევას სამუშაოდ“ – სტრიქონს Prelude>:

```
GHCi, version 7.10.3 :
http://www.haskell.org/ghc/ :? for help
Prelude>
```

2. თუ საჭიროა კოდის თანდათანობით აგება, მაშინ ვმუშაობთ საწყის ფაილებთან (ტექსტური ფაილი .hs გაფართოებით) და ვიყენებთ ცისფერი ფერის ფონს. მაგალითად, შემდეგი ტექსტი არის ფუნქცია sayHello-ის განმარტება, რომელსაც ეკრანზე გამოაქვს ტექსტი "Hello, x!" (x - ფუნქციის გამოძახებისას გადაცემული პარამეტრია):

```
sayHello :: String -> IO ()
sayHello x = putStrLn ("Hello, " ++ x ++ "!")
```

შევნიშნოთ, რომ ამ რეჟიმში არ ხდება კოდის შესრულება. კოდის შესასრულებლად ან უნდა გავუშვათ საწყისი კოდის ფაილი, ან გავხსნათ იგი REPL-ში.

სარჩევზე დაბრუნება

ნაწილი 1. HASKELL-ზე დაპროგრამების პრაქტიკა

თავი 1.1. მუშაობის საფუძვლები

HASKELL გარემო

Haskell-ის შესასწავლად, უპირველეს ყოვლისა, საჭიროა კომპიუტერზე დადგეს ინტერპრეტატორი, ანუ პროგრამა – ტრანსლატორი, რომელიც Haskell-ის კოდს გადაიყვანს ორობით კოდში და ამ კოდს შეასრულებს. Haskell მრავალმხრივ რეალიზებულია. მათგან ორი – ყველაზე გავრცელებულია. Hugs – ინტერპრეტატორი ძირითადად სასწავლო მიზნით გამოიყენება. რეალური პროგრამებისთვის Glasgow Haskell Compiler (GHCi) კომპილერი უფრო პოპულარულია. ის კომპილირებს მანქანურ კოდში, მხარს უჭერს პარალელურ დაპროგრამებას და აქვს პროგრამის ანალიზისა და გამართვის საშუალებები. აქედან გამომდინარე, ჩვენ მიერ შემდგომ GHCi იქნება გამოყენებული.

შემდეგ მისამართებზე მოყვანილია Haskell-ის ინტერპრეტატორის, სახელად GHCi-ს დოკუმენტაცია და დაინსტალირების ინსტრუქცია:

http://docs.haskellstack.org/en/stable/README/

https://github.com/bitemyapp/learnhaskell

Haskell-ი გვაძლევს კოდთან მუშაობის ორ გზას: პირველია კოდის შეტანა უშუალოდ GHCi-ის ინტერაქტიულ გარემოში ანუ REPL; მეორეა კოდის შეტანა საწყის ფაილში, შენახვა და შემდეგ საწყისი ფაილის ჩატვირთვა GHCi-ში. ჩვენ შევეხებით ორივე მეთოდს.

REPL არის შემოკლება შემდეგი ფრაზისა: *read-eval-print loop* (ციკლი: წაკითხვა-მნიშვნელობის გამოთვლა-დაბეჭდვა). REPL არის ინტერაქტიული დაპროგრამების გარემო, სადაც შეგვიძლია შევიტანოთ კოდი, რომლის მნიშვნელობაც გამოითვლება და დავინახავთ შედეგს. ეს პროცესი პირველად გაჩნდა ენა ლისპში, თუმცა ამჟამად საერთოა დაპროგრამების თანამედროვე ენებისთვის, მათ შორის Haskell–ისთვის.

ინტერპრეტატორის დაყენებისა და გაშვების შემდეგ ჩვენ დავინახავთ შეტყობინებას ვერსიის შესახებ, განცხადებას (თუ გსურთ დახმარება, აკრიფეთ სიმბოლო ?) და „მოწვევას სამუშაოდ“ (სტრიქონი Prelude>):

```
GHCi, version 7.10.3 :
http://www.haskell.org/ghc/ :? for help
Prelude>
```

რა თქმა უნდა, ვერსია შეიძლება იყოს სხვა.

ბრძანებათა სტრიქონში სიტყვა Prelude მიუთითებს, რომ ჩატვირთულია და მზად არის გამოსაყენებლად სტანდარტული ბიბლიოთეკა Prelude. როცა ჩავტვირთავთ სხვა მოდულებს ან საწყის ფაილებს, მაშინ ისინი აისახება დიალოგურ ფანჯარაში. შესაძლებელია Prelude-ის გამორთვაც, რასაც შემდეგ შევეხებით.

მოდულს Prelude ზოგჯერ უწოდებენ „სტანდარტულ პრელუდიას“, ვინაიდან მისი შინაარსი განისაზღვრება Haskell 98-ის სტანდარტით.

მოსაწვევის პასუხად შევიტანოთ რამდენიმე მარტივი არითმეტიკული გამოსახულება:

```
Prelude> 2 + 2
4
Prelude> 7 < 9
True
Prelude> 10 ^ 2
100
```

დავწეროთ ჩვენი პირველი პროგრამა Haskell-ზე, რომელსაც ეკრანზე გამოაქვს მისასალმებელი სტრიქონი. ფუნქციას დავარქვათ sayHello და გადავცეთ ერთი სტრიქონული არგუმენტი. ფუნქციის სახელთან მისი განმარტების დასაკავშირებლად გამოვიყენოთ ოპერატორი let (სახელთან დაკავშირების ოპერატორი). REPL გარემოში ავკრიფოთ ფუნქციის განმარტება:

```
Prelude> let sayHello x = putStrLn ("Hello, "
++ x ++ "!")
```

მაგალითში გამოყენებული გვაქვს putStrLn, რომელიც არის სტრიქონის ეკრანზე გამოტანის ფორმატირებული ოპერატორი. მაგალითად:

```
Prelude> putStrLn "Here's a newline -->\n <--
See?"
Here's a newline -->
 <-- See?
```

ახლა მოვახდინოთ ფუნქციის გამოძახება არგუმენტით `"Haskell"` და ვნახოთ შედეგი:

```
Prelude> sayHello "Haskell"
Hello, Haskell!
```

აქ გამოყენებულია ოპერატორი ++ კონკატენაცია, რომელიც ახდენს სტრიქონების შერწყმას.

GHCi-დან გამოსასვლელად გამოიყენეთ ბრძანება `:quit` ან `:q`.

მუშაობა საწყის ფაილებთან

მუშაობა საწყის ფაილებთან ისევე მოსახერხებელია, როგორც მუშაობა REPL-ის გარემოში. ფაილის გაფართოება სავალდებულოა იყოს `.hs`, რომელიც მიუთითებს, რომ ეს არის საწყისი კოდი Haskell-ზე. ამ რეჟიმს მაშინ ვიყენებთ, როცა საჭიროა კოდის თანდათანობით აგება. კოდის შექმნის შემდეგ ამ ფაილს ჩავტვირთავთ REPL-ში და შემდეგ ძირითადი პროცესი დადის ინტერაქტიულ პროცესზე: კოდის გაშვება, მოდიფიკაცია და ტესტირება.

საწყისი ფაილის შესრულებაც შესაძლებელია ორი გზით: ან გავუშვათ საწყისი კოდის ფაილი ან REPL-ში გავხსნათ იგი. პირველ შემთხვევაში გაიხსნება `gnci.exe` ინტერპრეტატორის ფანჯარა, სადაც უკვე ჩატვირთული იქნება ფაილის შინაარსი. მეორე შემთხვევაში კი გავხსნით REPL-ს და სპეციალური დირექტივის (`:load`) გამოყენებით ჩავტვირთავთ ფაილს.

გავხსნათ ახალი `Text Documen`. საჭიროა ფაილს დაერქვას სახელი, მაგალითად, `test.hs`. შევიტანოთ ამ ფაილში შემდეგი კოდი და შევინახოთ იგი იმავე სახელით (`test.hs`):

```
sayHello :: String -> IO ()
sayHello x = putStrLn ("Hello, " ++ x ++ "!")
```

აქ ნიშანი „::“ აღნიშნავს ტიპს, ანუ ჩვენ ვამბობთ, რომ „აქვს ტიპი“. ასე რომ, sayHello-ს აქვს ტიპი String -> IO (). ტიპებზე ჩვენ შემდგომ ვისაუბრებთ. შევნიშნოთ, რომ ეს სტრიქონი არ არის აუცილებელი, ვინაიდან Haskell თვითონ შეძლებს ამოიცნოს ჩვენ მიერ განსაზღვრული ფუნქციის ტიპი.

გავხსნათ WinGHCi და ჩავწეროთ შემდეგი ბრძანებები:

```
Prelude> :load test.hs
*Main>  sayHello "Haskell"
Hello, Haskell!
*Main>
```

პირველი ბრძანების :load ჩატვირთავს ფაილს test.hs, რის შემდეგად ფუნქცია sayHello უკვე ჩანს REPL-ში, ამიტომაც შეგიძლიათ გადასცეთ ნებისმიერი სტრიქონი-არგუმენტი (მაგალითად,"Haskell") და ნახოთ შედეგი. დააკვირდით, რომ ფაილის ჩატვირთვის შემდეგ მოსაწვევი სტრიქონი შეიცვალა. თუ გამოიყენებთ GHCi-ის ბრძანება :m-ს, რომელიც არის ბრძანება :module-ის შემოკლება, მაშინ დაბრუნდება Prelude> და ჩატვირთული ფაილი უკვე აღარ იქნება REPL-ის ხედვის წვდომის არეში:

```
*Main> :m
Prelude> sayHello "Haskell"

<interactive>:12:1: Not in scope: ‘sayHello’
(0.00 secs, 0 bytes)
Prelude>
```

სავარჯიშო

1. მოცემულია შემდეგი საწყისი კოდი:

```
half x = x / 2
square x = x * x
```

როგორ უნდა შეცვალოთ იგი, რათა გამოიყენოთ ეს კოდი REPL-ში? რა ტიპისაა თითოეული ფუნქცია (გამოიყენეთ `:type` დირექტივა!)?

2. დაწერეთ ფუნქცია `add a b`, რომელიც გადაამრავლებს `a` და `b` რიცხვებს. ფუნქცია განმარტეთ როგორც `REPL` გარემოში, ასევე საწყისი კოდის სახით.
3. დაწერეთ ერთი ფუნქცია, რომელსაც ექნება ერთი არგუმენტი და იმუშავებს ყველა გამოსახულებისთვის. დაარქვით ფუნქციას სახელი.

```
3.14 * (5 * 5)
3.14 * (10 * 10)
3.14 * (2 * 2)
3.14 * (4 * 4)
```

აქ გამოყენებულია მნიშვნელობა `pi`.

GHCI ბრძანებები

ჩვენ `REPL`-ში ხშირად გამოვიყენებთ `GHCi`-ის ბრძანებებს, ისეთებს, როგორიცაა, მაგალითად `:quit` ან `:info`. ეს არის სპეციალური ბრძანებები, რომლებიც იწყება სიმბოლო :-ით. `:quit` არ არის `Haskell`-ის კოდი, ის არის უბრალოდ `GHCi`-ის ფუნქცია.

GHCi-ის ბრძანებების გამოყენება შეიძლება შემოკლებითაც, ანუ მიეთითება მხოლოდ : და ბრძანების პირველი ასო. მაგალითად, :quit შეიძლება გამოვიძახოთ :q-თი, :info ბრძანება კი :i-ით.

არსებობს GHCi-ის სულ ოცამდე ბრძანება. აღვწეროთ ზოგიერთი მათგანი. შევნიშნოთ, რომ ქვემოთ მოყვანილ გამოსახულებებში გამოყენებულია დაპროგრამების ენების სინტაქსის აღწერის ბეკუს-ნაურის ფორმებში (Backus–Naur form) გამოყენებული მეტაენის სიმბოლოები: ოთხკუთხა ფრჩხილები ([])წარმოადგენს აღწერის მეტაენის სიმბოლოებს, რომელიც ნიშნავს „შეიძლება იყოს ან არ იყოს“, ანუ ეს პარამეტრი შეიძლება არც იყოს, ხოლო კუთხური ფრჩხილები (<>)– შეიცავს კონკრეტულ დასახელებას. ძირითადად ეს არის სტრიქონული პარამეტრი.

შემდეგი ბრძანება:

```
:load [<filenames>]
```

ჩატვირთავს პროგრამულ მოდულს მოცემული ფაილიდან; იმეორებს ინსტრუმენტების პანელის მოდულის ჩატვირთვის ღილაკის  ფუნქციას. კონკრეტულ გამოძახებას შეიძლება ჰქონდეს ასეთი სახე :

```
Prelude> :load "1.hs"     -- ფაილი, სახელით 1.hs
[1 of 1] Compiling Main      (1.hs, interpreted)
Ok, modules loaded: Main.
*Main> :load
Ok, modules loaded: none.
```

პირველ სტრიქონში მიეთითა კონკრეტული ფაილის სახელი (`"1.hs"`). ყურადღება მიაქციეთ, რომ ფაილის გახსნის შემდეგ მოსაწვევი სტრიქონის სახელი გახდა `*Main>`. შემდეგ ბრძანებაში არ მიეთითა ფაილის სახელი, ამიტომაც გაჩუმების, შეთანხმების პრინციპით აღებულ იქნა სულ ბოლოს გამოყენებული ფაილის სახელი. ჩვენს მაგალითში „`1.hs`“.

შემდეგი ბრძანება:

```
:also <filenames>
```

ჩატვირთავს დამატებით მოდულს მიმდინარე პროექტში.

მომდევნო ბრძანება:

```
:reload
```

იმეორებს `:load` ჩატვირთვის ბოლო შესრულებულ ბრძანებას. იგი საშუალებას იძლევა მოდული ხელახლა სწრაფად ჩავტვირთოთ იმ შემთხვევაში, თუ მისი რედაქტირება გარე ტექსტურ რედაქტორში ხდება.

შემდეგი ბრძანება:

```
:project <filename>
```

ჩატვირთავს და გამოიყენებს პროექტის ფაილს. შეიძლება მხოლოდ ერთი ფაილის ჩატვირთვა. პროექტი გამოიყენება დაცალკევებული კოდიანი ფაილების გასაერთიანებლად.

შემდეგი ბრძანება:

```
:edit [<filename>]
```

იძახებს გარე ტექსტურ რედაქტორს მითითებული ფაილის გასასწორებლად. თუ ფაილის სახელი მითითებული არაა, მაშინ რედაქტირებისთვის იძახებს ბოლო (ჩატვირთულ ან რედაქტირებულ) ფაილს. მოცემული ბრძანება, ფაქტობრივად, იმავე ფუნქციას ასრულებს, რასაც გარე ტექსტური რედაქტორის გამოძახების ინსტრუმენტების პანელის ღილაკი  .

მომდევნო ბრძანება:

```
:module <module>
```

მოცემულ მოდულს აცხადებს მიმდინარე მოდულად. ეს ბრძანება განკუთვნილია, უპირველეს ყოვლისა, სახელების კოლიზიათა გადასაჭრელად. მაგალითად, მოდული `Data.Ratio` გვაძლევს რაციონალურ რიცხვებთან (წილადებთან) მუშაობის საშუალებას. მაგალითად:

```
:module + Data.Ratio
Prelude> 3/5
0.6
it :: Fractional a => a
```

`REPL`-ში სახელის აკრეფა იწვევს მისი მნიშვნელობის გამოთვლას:

```
<expr>
```

ახდენს მოცემული – `expr` გამოსახულების გაშვებას შესასრულებლად.

მომდევნო ბრძანებას:

```
:type <expr>
```

ეკრანზე გამოაქვს მოცემული გამოსახულების ტიპი. ეს ბრძანება უმთავრესად გამოიყენება გამართვის მიზნით შესაქმნელი გამოსახულების (ცვლადის, ფუნქციის, რთული ობიექტის) ტიპის მისაღებად.

შემდეგ ბრძანებას:

```
Prelude>:?
```

ეკრანზე გამოაქვს კომპილერის ყველა ბრძანების სია და თითოეული მათგანის მოკლე აღწერა. იგივე მოქმედება შესრულდება  ღილაკზე ხელის დაჭერითაც.

შემდეგი ბრძანება:

```
Prelude>:set [<options>]
```

უზრუნველყოფს ბრძანებათა სტრიქონიდან GHCi-ის პარამეტრების მოცემის საშუალებას. რეკომენდებულია მუშაობა დავიწყოთ შემდეგი ბრძანებებით:

```
Prelude>:set +t
```

რომელიც ჩართავს გამოსახულების ტიპების ჩვენების ოფციას. მაგალითად:

```
Prelude> 2
2
it :: Num a => a
```

თავდაპირველად სასარგებლოა ვაჩვენოთ ყველა გამოსახულების ტიპი, მაგრამ, ტიპებში გარკვევის შემდეგ, შესაძლოა საჭირო გახდეს ამ ოფციის გამორთვა. ამისთვის საჭიროა გამოვიყენოთ ბრძანება `:unset`:

```
Prelude> :unset +t
Prelude> 2
2
```

ბრძანებაში `set` პარამეტრი `+s`, ანუ

```
:set +s
```

ჩართავს გამოთვლებზე დახარჯულ დროს და მეხსიერების რაოდენობის ჩვენებას. პარამეტრი `prompt`-ით შესაძლებელია `Prelude>` მოწვევის შეცვლა იმ სტრიქონით, რომელიც თქვენთვის სასურველია, მაგალითად:

```
:set prompt student>
```

`Prelude>` შეიცვლება `student>`-ით.

მომდევნო ბრძანებას:

```
:info <names>
```

ეკრანზე გამოაქვს მოცემული სახელის აღწერა/ინფორმაცია. `Haskell`-ში ოპერატორებს ენიჭება რიცხვითი მნიშვნელობები 1-დან (ყველაზე პატარა პრიორიტეტი) 9-მდე. იმის გასაგებად, თუ რა პრიორიტეტი აქვს ოპერატორს, შეიძლება გამოვიყენოთ `:info` – ინფორმაცია ბრძანების შესახებ:

```
Prelude> :info (+)
class Num a where
  (+) :: a -> a -> a
  ...
     -- Defined in ‘GHC.Num’
infixl 6 +
Prelude> :info (*)
class Num a where
  ...
  (*) :: a -> a -> a
  ...
     -- Defined in ‘GHC.Num’
infixl 7 *
```

ინფორმაცია „`infixl 6 +`“ მიუთითებს, რომ ოპერატორს (+) აქვს პრიორიტეტი 6 (სხვა მონაცემებს შემდეგ განვიხილავთ). „`infixl 7 *`“ განსაზღვრავს, რომ ოპერატორს (*) აქვს პრიორიტეტი 7. ამის გამო ვხედავთ, რომ `1 + 4 * 4` გამოითვლება, როგორც `1 + (4 * 4)` და არა, როგორც `(1 + 4) * 4`.

Haskell-ში განსაზღვრულია ასევე ოპერატორების ასოციურობა – ოპერატორის მრავალჯერადი გამოყენების შესაძლებლობა მარცხნიდან მარჯვნივ ან პირიქით. (+) და (*) მარცხნიდან ასოციურია და აღინიშნება როგორც `infixl`. მარჯვნიდან მარცხნივ ასოციურობა აღინიშნება `infixr`. ასეთია მთელი რიცხვის ახარისხების ოპერაცია.

```
Prelude> :info (^)
(^) :: (Num a, Integral b) => a -> b -> a
                        -- Defined in 'GHC.Real'
infixr 8 ^
```

შემდეგ ბრძანებას:

```
:browse <modules>
```

ეკრანზე გამოაქვს მოცემულ მოდულში განსაზღვრული ყველა ობიექტის (ფუნქცია, ცვლადი, ტიპი) სია.

შემდეგი ბრძანება:

```
:find <name>
```

გამოიძახებს მოცემული სახელის შემცველ მოდულს რედაქტირებისათვის. თუ მოცემული სახელი არცერთ მიმდინარე მოდულში არ არის, გამოვა შეტყობინება შეცდომის შესახებ: `ERROR – No current definition for name "<name>".`

მომდევნო ბრძანება:

```
:!<command>
```

გამოდის ოპერაციულ სისტემაში და ასრულებს მოცემულ ბრძანებას. ხაზგასმით უნდა აღინიშნოს, რომ ძახილის ნიშანსა და ოპერაციული სისტემის ბრძანებას შორის ხარვეზი დაუშვებელია.

შემდეგი ბრძანება:

```
:cd <directory>
```

ცვლის მიმდინარე კატალოგს.

შემდეგი ბრძანებით:

```
:quit
```

REPL-ში მუშაობას ვამთავრებთ. ეს იგივეა, რაც ოპერაციულ სისტემაში დაბრუნება.

ძირითადი გამოსახულებები და ფუნქციები

Haskell-ში ყველაფერი ან გამოსახულებაა, ან დეკლარაცია. გამოსახულებას შეიძლება ჰქონდეს მნიშვნელობა ან მნიშვნელობების კომბინაცია და/ან იყოს ფუნქციის გამოძახება არგუმენტზე.

თუ გამოსახულებებს შევიტანთ REPL-ში, გამოითვლება მნიშვნელობები:

```
Prelude> 1
1
Prelude> 1 / 2
0.5
Prelude> ((1 + 2) * 3) ^ 10
3486784401
```

ვიტყვით, რომ გამოსახულება არის *ნორმალურ ფორმაში,* თუ მისი შემდგომი გარდაქმნა შეუძლებელია. მაგალითად, 1+1-ის ნორმალური ფორმაა 2. ნორმალურ ფორმაზე დაყვანას ასევე უწოდებენ *რედუცირებას,* მიღებულ შედეგს კი – *რედექსს.*

`Haskell` ენის პროგრამები არიან გამოსახულებები, რომელთა გამოთვლებს მივყავართ მნიშვნელობებამდე. თითოეულ მნიშვნელობას აქვს *ტიპი.* ინტუიციურად, ტიპი შეიძლება გავიგოთ როგორც გამოსახულების დასაშვები მნიშვნელობების სიმრავლე. იმისათვის, რომ განვსაზღვროთ, რა ტიპი აქვს მოცემულ გამოსახულებას, საჭიროა გამოვიყენოთ ინტერპრეტატორის ბრძანება `:type`.

ამის გარდა, შესაძლოა მოვითხოვთ დაიბეჭდოს ინფორმაცია ყოველი გამოთვლილი გამოსახულების ტიპის შესახებ. ამისთვის საჭიროა გამოვიყენოთ ინტერპრეტატორის ბრძანება `:set` პარამეტრით `+t`.

```
Prelude> :set +t
Prelude> 'c'
'c'
it :: Char
Prelude> "foo"
"foo"
it :: [Char]
```

გამოსახულების მნიშვნელობის შემდეგ ტიპის დაბეჭდვა ძალზე სასარგებლოა.

ყურადღება გავამახვილოთ:

- სახელი `it` არის სპეციალური ცვლადი, რომელშიც `GHCi` ინახავს ბოლო გამოსახულების გამოთვლილ მნიშვნელობას.
- `x ::` ფორმის ტექსტს აქვს ასეთი შინაარსი: „გამოსახულება `x`-ს აქვს ტიპი `y` “.

- სტრიქონს `"foo"` აქვს ტიპი[Char], ანუ არის სიმბოლოების სია. სიტყვა `string` ხშირად გამოიყენება `[Char]`-ის ნაცვლად. ისინი სინონიმებია.

ცვლადი `it` GHCi-ის მოსახერხებელი სტრუქტურაა, რომელიც გვაძლევს საშუალებას გამოვიყენოთ ბოლო გამოთვლილი გამოსახულების მნიშვნელობა.

```
Prelude> "foo"
"foo"
it :: [Char]
Prelude> it ++ "bar"
"foobar"
it :: [Char]
```

აქ გამოყენებულია ოპერატორი ++ კონკატენაცია, რომელიც ახდენს არგუმენტების (სიების) შერწყმას. თუ გამოსახულების გამოთვლისას შეცდომა დაფიქსირდა, მაშინ GHCi არ ცვლის `it`-ის მნიშვნელობას.

```
Prelude> it
"foobar"
it :: [Char]
Prelude> it ++ 3

<interactive>:1:6:
    No instance for (Num [Char])
      arising from the literal `3' at
      <interactive>:1:6
    Possible fix: add an instance declaration
    for (Num [Char])
    In the second argument of `(++)', namely `3'
    In the expression: it ++ 3
```

```
    In the definition of `it': it = it ++ 3
Prelude> it
"foobar"
it :: [Char]
Prelude> it ++ "baz"
"foobarbaz"
it :: [Char]
```

მარტივი ტიპები

ენა Haskell-ის მარტივ 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 ტიპიზებულ*** დაპროგრამების ენას, რაც იმას ნიშნავს, რომ ტიპების არაცხადი გარდაქმნა არ ხდება. მიუხედავად ამისა, ხშირ შემთხვევაში პროგრამისტი არ არის ვალდებული განაცხადოს, თუ რომელ ტიპს ეკუთვნის მის მიერ შემოტანილი ცვლადი. ინტერპრეტატორს თვითონ აქვს შესაძლებლობა ***გამოიყვანოს*** მომხმარებლის მიერ გამოყენებული ცვლა-

დის ტიპი. თუმცა, თუ რაიმე მიზნისთვის საჭიროა გამოცხადდეს, რომ მნიშვნელობა ეკუთვნის რომელიღაც ტიპს, გამოიყენება კონსტრუქცია: *ცვლადი* `::` *ტიპი.* თუ ჩართულია ინტერპრეტატორის ოფცია `:set +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 მაგალითიდან ჩანს, რომ ტიპების – `Integer`, `Double` და `Char` მნიშვნელობები მოიცემა იმავე წესებით, როგორც ენა `C/C++`-ში.

ტიპების სისტემის განვითარებისა და მკაცრი ტიპიზაციის შედეგად `Haskell` ენის პროგრამები არის *უსაფრთხო.* გარანტირებულია, რომ `Haskell` ენის გამართულ პროგრამაში ტიპებიც შესაბამისად გამოიყენება. პრაქტიკულად, ეს ნიშნავს, რომ `Haskell` ენაზე პროგრამა შესრულებისას ვერ გამოიწვევს შეცდომას მეხსიერების შეღწევადობაზე `(Access violation)`. ასევე გარანტირებულია, რომ პროგრამაში ცვლადების გამოყენება საწყისი ინიციალი-

ზების გარეშე არ მოხდება. ამრიგად, პროგრამაში მრავალი შეცდომის არსებობა მოწმდება კომპილაციის და არა შესრულების ეტაპზე.

არითმეტიკა

ზოგადად, გამოსახულებები მოიცემა *ინფიქსური ფორმით* (in *infix* form), სადაც ოპერატორი მოთავსებულია ოპერანდებს შორის.

```
Prelude> 2 + 2
4
Prelude> 31337 * 101
3165037
Prelude> 7.0 / 2.0
3.5
```

ჩანაწერის ინფიქსური ფორმის გარდა, შესაძლებელია *პრეფიქსული ფორმის* (in *prefix form*) გამოყენება. ამისთვის საჭიროა ოპერატორი ჩაისვას მრგვალ ფრჩხილებში, ოპერანდები კი ხარვეზით გამოიყოს:

```
Prelude> 2 + 2
4
Prelude> (+) 2 2
4
```

Haskell-ში არის მთელი და მცოცავმძიმიანი რიცხვების ცნება. ნიშანი (^) არის რიცხვის მთელ ხარისხში აყვანის ოპერაცია, ხოლო ნიშანი (**) – ნამდვილ ხარისხში აყვანის ოპერაცია:

```
Prelude> 7 ^ 80
40536215597144386832065866109016673800875222251012083746192454448001
it :: Integer
Prelude> 625**0.5
25.0
Prelude> 625**(-0.5)
4.000000000000001e-2
```

საჭიროა ასევე აღინიშნოს, რომ დაპროგრამების 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 თანრიგობრიობით (თანამედროვე მანქანებზე იგი ტოლია $2^{31}-1$ = 2147483647), Haskell ენაში ტიპმა Integer-მა შეიძლება წარმოადგინოს ნებისმიერი სიგრძის მთელი რიცხვი.

```
Prelude> 2^500
32733906078961418700131896968275991522166420460430647894832913680961337964046745548832700923259041571508866841275600710092172565458853930533285275893760

Prelude> 2^5000
14124670321394260368352096670161473336688961751845411168136880858571181698427075125580891263167115263733560320843136608276420383806997933833597118572663992343105177785186539901187799964513170706937349821263132375255311121537284403595090053595486073341845340557556673680156558740546469964049905084969947235790090561757137661822821643421376589376
```

```
181520991556677126498651782204174061830939239176
861341383294018240225838692725596147005144243281
075275629495339093813198966735633606329691023842
454125835888656873133981287240980008838073668221
804264432910894030789020219440578198488267339768
238872279902157420307247570510423845868872596735
891805818727796435753018518086641356012851302546
726823009250218328018251907340245449863183265637
987862198511046362985461949587281119139907228004
385942880953958816554567625296086916885774828934
449941362416588675326940332561103664556982622206
834474219811081872404929503481991376740379825998
791411879802717583885498575115299471743469241117
070230398103378615232793710290992656444842895511
830355733152020804157920090041811951880456705515
468349446182731742327685989277607620709525878318
766488368348965015474997864119765441433356928012
344111765735336393557879214937004347568208665958
717764059293592887514292843557047089164876483116
615691886203812997555690171892169733755224469032
475078797830901321579940127337210694377283439922
280274060798234786740434893458120198341101033812
506720046609891160700284002100980452964039788704
335302619337597862052192280371481132164147186514
169090917191909376

Prelude>
```

ეს უკანასკნელი 1506 ციფრისგან შემდგარი მნიშვნელობაა.

Haskell-ში ლოგიკური მნიშვნელობებია True და False. დიდი ასოთი დაწყება აუცილებელია. ლოგიკური ოპერაციებია: && - ლოგიკური „და“, || - ლოგიკური „ან“.

```
Prelude> True && False
False
Prelude> False || True
True
```

ზოგიერთ ენაში რიცხვი 0 ასოცირდება როგორც ლოგიკური False, ხოლო არანულოვანი მნიშვნელობა – ლოგიკური True. Haskell-ში ასე არ არის:

```
Prelude> True && 1

<interactive>:1:8:
    No instance for (Num Bool)
      arising from the literal ‘1’ at
      <interactive>:1:8
    Possible fix: add an instance declaration
    for(Num Bool)
    In the second argument of ‘(&&)’, namely ‘1’
    In the expression: True && 1
    In the definition of ‘it’: it = True && 1
```

ამ შეცდომას იწვევს ის, რომ ლოგიკური ტიპი არ არის რიცხვითი Num ტიპის.

მოვიყვანოთ ზოგიერთი ტიპური შეტყობინება შეცდომის შესახებ:

- "No instance for (Num Bool)" გვეუბნება, რომ ინტერპრეტატორი GHCi ცდილობს გარდაქმნას მნიშვნელობა 1 როგორც Bool ტიპი, მაგრამ ეს არ შეიძლება.
- "arising from the literal '1' " მიუთითებს, რომ მთელი რიცხვის – 1-ის გამოყენება იწვევს პრობლემას.
- "In the definition of 'it'" – GHCi მიუთითებს, რომ it განსაზღვრულია.

შედარების ოპერატორები Haskell-ში მსგავსია C/C++ ენის ოპერატორებისა, გარდა ოპერატორისა „არ უდრის, რომელიც აღინიშნება ნიშნით (/=) :

```
Prelude> 1 == 1
True
Prelude> 2 < 3
True
Prelude> 4 >= 3.99
True
Prelude> 2 /= 3
True
```

გარდა ამისა, C/C++ -ის ტიპის ენებში გამოიყენება ლოგიკური უარყოფისთვის ნიშანი !, ხოლო Haskell-ში ფუნქცია not .

```
Prelude> not True
False
```

როგორც სხვა ენებში, Haskell-ის ინფიქსური ოპერატორებისთვისაც განისაზღვრება პრიორიტეტები. გამოსახულების ცხადი დაჯგუფებისთვის გამოიყენება მრგვალი ფრჩხილები.

Haskell-ში ოპერატორებს ენიჭება რიცხვითი მნიშვნელობები 1-დან (ყველაზე პატარა პრიორიტეტი) 9-მდე. იმის გასაგებად, თუ რა პრიორიტეტი აქვს ოპერატორს, შეიძლება გამოვიყენოთ `:info` – ინფორმაცია ბრძანების შესახებ:

```
Prelude> :info (+)
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  ...
     -- Defined in GHC.Num
infixl 6 +
Prelude> :info (*)
class (Eq a, Show a) => Num a where
  ...
  (*) :: a -> a -> a
  ...
     -- Defined in GHC.Num
infixl 7 *
```

ინფორმაცია "`infixl 6 +`" მიუთითებს, რომ ოპერატორს `(+)` აქვს პრიორიტეტი `6` (სხვა მონაცემებს შემდეგ განვიხილავთ). "`infixl 7 *`" განსაზღვრავს, რომ ოპერატორს `(*)` აქვს პრიორიტეტი `7`. ამიტომაც `1 + 4 * 4` გამოითვლება, როგორც `1 + (4 * 4)` და არა როგორც `(1 + 4) * 4`.

Haskell-ში განსაზღვრულია ასევე ოპერატორების ასოციურობა – ოპერატორის მრავალჯერადი გამოყენების შესაძლებლობა მარცხნიდან მარჯვნივ ან პირიქით. `(+)` და `(*)` მარცხნიდან ასოციურია და აღინიშნება როგორც `infixl`. მარჯვნიდან მარცხნივ

ასოციურობა აღინიშნება `infixr`. ასეთია მთელი რიცხვის ახარისხების ოპერაცია.

```
Prelude> :info (^)
(^) :: (Num a, Integral b) => a -> b -> a --
Defined in GHC.Real
infixr 8 ^
```

განსაზღვრული მნიშვნელობები - კონსტანტები

`Haskell`-ის სტანდარტულ ბიბლიოთეკაში `Prelude` განსაზღვრულია მათემატიკური კონსტანტები:

```
Prelude> pi
3.141592653589793
```

მაგრამ ყველა კონსტანტა, რა თქმა უნდა, `Prelude`-ში განმარტებული არაა. მაგალითად, ეილერის რიცხვი:

```
Prelude> e
<interactive>:1:0: Not in scope: ‘e’
```

საჭიროა ჩვენვე განვსაზღვროთ იგი. GHCi-ში არის `let` კონსტრუქცია, რომლის საშუალებითაც შეიძლება `e`-ს დროებითი განსაზღვრა:

```
Prelude> let e = exp 1
```

exp 1 არის ჩვენთვის Haskell ფუნქციის გამოყენების მაგალითი. მრგვალი ფრჩხილების გამოყენება არგუმენტისთვის საჭირო არ არის. ასე განსაზღვრული e შეიძლება გამოვიყენოთ არითმეტიკულ გამოსახულებებში. ოპერატორი ^ გამოიყენება მხოლოდ მაშინ, თუ ახარისხებისას ხარისხის მაჩვენებელი მთელი რიცხვია, ხოლო, თუ ხარისხის მაჩვენებელი ნამდვილი რიცხვია, გამოიყენება ოპერატორი **. მაგალითი:

```
Prelude> (e ** pi) - pi
19.99909997918947
Prelude> 2^(-5)
*** Exception: Negative exponent
Prelude> 2**(-5)
3.125e-2
```

ეს სინტაქსი დამოკიდებულია ghci-ის კონკრეტულ რეალიზაციაზე! დამატებითი ინფორმაცია შეიძლება მოვიძიოთ: Unix სისტემებისთვის ბიბლიოთეკები GNU readline, Windows-ისთვის, პრძანება doskey.

მუშაობა რაციონალურ რიცხვებთან

რაციონალურ რიცხვებთან სამუშაოდ შექმნილია ბიბლიოთეკა Data.Ratio, რომლის გამოსაყენებლად საჭიროა ჩავტვირთოთ იგი :module დირექტივით – ავკრიფოთ სტრიქონი :module +Data.Ratio ან :m +Data.Ratio. რაციონალური რიცხვების მისაღებად უნდა გამოვიყენოთ ოპერატორი(%), რომლის მარცხნივ უნდა იყოს მრიცხველი, მარჯვნივ – მნიშვნელი.

```
Prelude> :m +Data.Ratio
Prelude> 11 % 29
11%29
it :: Ratio Integer
```

შევნიშნოთ, რომ წილადებში მრიცხველს და მნიშვნელს აქვს მთელი ტიპი. ჩვენ თუ შევეცდებით მთელის ნაცვლად სხვა ტიპის მნიშვნელობის აღებას, დაფიქსირდება შეცდომა:

```
Prelude> 3.14 % 8

<interactive>:1:0:
    Ambiguous type variable `t' in the
    constraints:
      `Integral t' arising from a use of `%' at
<interactive>:1:0-7
      `Fractional t'
        arising from the literal `3.14' at
        <interactive>:1:0-3
    Probable fix: add a type signature that
    fixes these type variable(s)
Prelude> 1.2 % 3.4

<interactive>:1:0:
    Ambiguous type variable `t' in the
    constraints:
      `Integral t' arising from a use of `%' at
<interactive>:1:0-8
      `Fractional t'
        arising from the literal `3.4' at
        <interactive>:1:6-8
    Probable fix: add a type signature that
    fixes these type variable(s)
```

თუმცა თავდაპირველად სასარგებლოა ვაჩვენოთ ყველა გამოსახულების ტიპი, მაგრამ ტიპებში გარკვევის შემდეგ შესაძლოა საჭირო გახდეს ამ ოპციის გამორთვა. ამისთვის საჭიროა გამოვიყენოთ ბრძანება : `unset`.

```
Prelude> :unset +t
Prelude> 2
2
```

ამ შემთხვევაშიც შეგვიძლია მივიღოთ ინფორმაცია ტიპების შესახებ `ghci`-ის შემდეგი ბრძანებით `:type`.

```
Prelude> :type 'a'
'a' :: Char
Prelude> "foo"
"foo"
Prelude> :type it
it :: [Char]
```

ბრძანება : `type` ბეჭდავს ნებისმიერი გამოსახულების ტიპს, `it`-ისაც კი.

გაარკვიეთ, რატომ აქვს შემდეგ გამოსახულებებს სხვადასხვა ტიპი:

```
Prelude> 3 + 2
5
Prelude> :type it
it :: Integer
Prelude> :type 3 + 2
3 + 2 :: (Num t) => t
```

Haskell-ში არის რამდენიმე რიცხვითი ტიპი. GHCi გამოსახულებისთვის `3 + 2`, გაჩუმების პრინციპით იღებს ტიპს `Integer`. მაგრამ მეორე მაგალითისთვის ასახელებს, რომ ტიპი არის რიცხვითი.

სავარჯიშოები

1. დაწერეთ შემდეგი გამოსახულებები GHCi -ში. შეამოწმეთ, რა ტიპი აქვთ მათ? რას აკეთებს მოცემული ფუნქციები?

   - ✓ `5 + 8`
   - ✓ `3 * 5 + 8`
   - ✓ `2 + 4`
   - ✓ `(+) 2 4`
   - ✓ `sqrt 16`
   - ✓ `succ 6`
   - ✓ `succ 7`
   - ✓ `pred 9`
   - ✓ `pred 8`
   - ✓ `sin (pi / 2)`
   - ✓ `truncate pi`
   - ✓ `round 3.5`
   - ✓ `round 3.4`
   - ✓ `floor 3.7`
   - ✓ `ceiling 3.3`

ქვემოთ მოყვანილია მაგალითში გამოყენებული ფუნქციების ცნობარი:

ფუნქციის სახელი	დანიშნულება	მაგალითი
sqrt	ფესვის ამოღება	
succ	ერთიანის მიმატება	succ 4.567= 5.567
pred	ერთიანის გამოკლება	
sin	სინუსი	
truncate	მთელი ნაწილი	truncate (-2.67)= -2
round	დამრგვალება	round (-3.5)= -4
floor	უახლოესი უდიდესი მთელი, რომელიც არგუმენტს არ აღემატება	floor 3.7=3 floor (-3.3)= -4
ceiling	უახლოესი უმცირესი მთელი, რომელიც არგუმენტზე პატარა არ არის	ceiling 1.005=2 ceiling (-1.005)= -1
putStrLn	სტრიქონის ბეჭდვა (ფორმატირებული)	putStrLn "Here's a newline -->\n <-- See?"

თავი 1.2. სიები

სია არის ბაზური სტრუქტურა, რომელიც გამოიყენება ფუნქციონალურ ენებში. მისი საშუალებით შესაძლებელია წარმოვადგინოთ მონაცემთა მრავალფეროვანი სტრუქტურა, მარტივი მასივიდან დაწყებული რთულ სტრუქტურებამდე. სიების საშუალებით შეიძლება აღიწეროს მონაცემთა უსასრულო თანმიმდევრობა. ამ უკანასკნელს ადგილი აქვს მხოლოდ დაპროგრამების არამკაცრ ენებში, ანუ ისეთებში, რომლებიც მხარს უჭერენ გადადებულ, ზარმაც გამოთვლებს.

სიის ელემენტები თავსდება ოთხკუთხა ფრჩხილებში და გამოიყოფა ერთმანეთისგან მძიმით. ელემენტები აუცილებლად უნდა ეკუთვნოდეს ერთსა და იმავე ტიპს. სიის ტიპი, რომელიც შედგება `a` ტიპის ელემენტებისგან, აღინიშნება როგორც `[a]`.

```
Prelude> [1, 2, 3]
[1,2,3]
```

სიის სიგრძე ნებისმიერია. ცარიელი სია აღინიშნება როგორც `[]`.

```
Prelude> []
[]
Prelude> ["foo", "bar", "baz", "quux",
"fnord", "xyzzy"]
["foo","bar","baz","quux","fnord","xyzzy"]
```

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 (:)-ის და ცარიელი სიის გამოყენებით შეიძლება ავაგოთ ნებისმიერი სია:

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

ოპერატორი (:) მარჯვნიდან ასოციაციური ოპერაციაა, ამიტომ ზემოთ მოყვანილ გამოსახულებაში შეიძლება გამოვტოვოთ ფრჩხილები:

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

მოცემულ სიაში ბოლო ელემენტს ასე: [1,2]:3 ვერ დავამატებთ. ეს გამოიწვევს შეცდომას, რადგან : ოპერატორის მეორე არგუმენტი სია არ არის.

სიის ელემენტები შეიძლება იყოს ნებისმიერი მნიშვნელობები – რიცხვები, სიმბოლოები, კორტეჟები (შემდგომ ვისაუბრებთ), 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]]
```

აუცილებელია სიის ყველა ელემენტი იყოს ერთისა და იმავე ტიპის. თუ ამ წესს დავარღვევთ, გვექნება შეტყობინება შეცდომის შესახებ:

```
Prelude> [True, False, "testing"]

<interactive>:1:14:
    Couldn't match expected type `Bool'
    against inferred type `[Char]'
      Expected type: Bool
      Inferred type: [Char]
    In the expression: "testing"
    In the expression: [True, False,"testing"]
```

სიის ელემენტებში შეიძლება გამოვიყენოთ *ჩამოთვლის ნოტაცია* (`..`), `Haskell` თვითონ შეავსებს სიას:

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

აქ სიმბოლო `..` აღნიშნავს ჩამოთვლას `1`-ის მატებით ან კლებით. მაგალითში განხილულია ჩაკეტილი ინტერვალი, მაგრამ შესაძლებელია ღია ინტერვალების გამოყენებაც.

ჩამონათვალში შესაძლოა მითითებული იყოს ბიჯის ზომაც. პირველი ორი ელემენტის შემდეგ მოცემულია მნიშვნელობა, რომელიც ამთავრებს სიას:

```
Prelude> [1.0,1.25..2.0]
[1.0,1.25,1.5,1.75,2.0]
Prelude> [1,4..15]
[1,4,7,10,13]
Prelude> [10,9..1]
[10,9,8,7,6,5,4,3,2,1]
```

ჩვენ შეიძლება გამოვტოვოთ ჩამონათვალის ბოლო ელემენტი. თუ ტიპს არ აქვს ბუნებრივი ზედა ზღვარი, მაშიმ მან შეიძლება მოგვცეს უსასრულო სია. მაგალითად, თუ ჩავწერთ `[1..]`, GHCi-ის დიალოგურ ფანჯარაში გვექნება უსასრულო ჩამონათვალი, რომელიც ხელით უნდა გავაჩეროთ. სხვა ენებში, მაგალითად, C/C++ -ში ასე არ არის – მოხდება გადავსების შეცდომა. უსასრულო სიები Haskell-ში ხშირად არის სასარგებლო.

გაფრთხილდით, ნამდვილი რიცხვებისთვის `..` გამოყენების დროს, შეიძლება ასეთი ტიპის შეცდომები დაუშვათ:

```
Prelude> [1.0..1.8]
[1.0,2.0]
```

ყურადღება მიაქციეთ, რომ თქვენ მიერ განსაზღვრულ სიაში ინტერპრეტატორმა ზედმეტი ელემენტი ჩასვა.

ხშირად გამოყენებადი ფუნქციები სიებთან

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
Prelude> [3,1,3] ++ [3,7]
[3,1,3,3,7]
Prelude> [] ++ [False,True] ++ [True]
[False,True,True]
```

სიის კონსტრუქტორები

მათემატიკაში კონსტრუქტორის ცნება გამოიყენება ახალი სიმრავლეების ასაგებად ძველი სიმრავლეების საფუძველზე (სიმრავლის კონსტრუქტორი). მაგალითად,

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

არის x^2 რიცხვების $\{1,4,9,16,25\}$ სიმრავლე, სადაც x არის $\{1,2,3,4,5\}$ სიმრავლის ელემენტი.

Haskell-ში კონსტრუქტორის მსგავსი ცნება შეიძლება გამოვიყენოთ ახალი სიების ასაგებად ძველი სიების საფუძველზე:

```
[x^2 | x ← [1..5]]
```

ეს არის x^2 რიცხვების {1,4,9,16,25} სიმრავლე, სადაც x არის {1,2,3,4,5} სიის ელემენტი.

შევნიშნოთ, რომ x ← [1..5] გამოსახულებას გენერატორი ეწოდება, რადგან იგი გვეუბნება, თუ საიდან მიიღება x-ის მნიშვნელობები. ნიშანი ← (ისარი) აიკრიფება ორი სიმბოლოს გამოყენებით: <-. კონსტრუქტორს მძიმეებით გამოყოფილი რამდენიმე გენერატორი შეიძლება გააჩნდეს. მაგალითად:

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

გენერატორების თანამიმდევრობის შეცვლისას იცვლება ელემენტების მიმდევრობაც საბოლოო სიაში:

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

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

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

არსებობს დამოკიდებული გენერატორებიც, ანუ მომდევნო გენერატორები შეიძლება დამოკიდებული იყოს ცვლადებზე, რომლებიც უფრო წინა გენერატორებით შეიყვანება.

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

არის რიცხვთა ყველა `(x,y)` წყვილის `[(1,1),(1,2),(1,3),(2,2),(2,3), 3,3)]` სია, როცა `x` და `y` ელემენტებია `[1..3]` სიიდან და, ამასთან ერთად, $y \geq x$.

დამოკიდებული გენერატორის საშუალებით ჩვენ შეგვიძლია განვსაზღვროთ `Prelude` ბიბლიოთეკის ფუნქცია `concat`, რომელიც აერთიანებს ერთ სიას მეორესთან:

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

მაგალითად,

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

სიის კონსტრუქტორებს შეუძლიათ პრედიკატების გამოყენება იმ მნიშვნელობების შესაზღუდავად, რომლებიც ნაწარმოებია წინა გენერატორებით.

```
[x | x ← [1..10], even x]
```

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

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

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

მაგალითად:

```
Prelude> factors 15
[1,3,5,15]
```

დადებითი მთელი რიცხვი მარტივია, თუ მას მხოლოდ ორი ფაქტორი აქვს (1 და თავად ეს რიცხვი). `factors` ფუნქციის გამოყენებით შესაძლებელია ახალი ფუნქციის განსაზღვრა, რომელიც ადგენს, მარტივია თუ არა რიცხვი:

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

მაგალითად:

```
Prelude > prime 15
False
Prelude > prime 7
True
```

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

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

მაგალითად:

```
Prelude > primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

სიის კონსტრუქტორები გამოიყენება ფუნქციების განსაზღვრისთვისაც.

სტრიქონები და სიმბოლოები

სტრიქონული მნიშვნელობები ენა Haskell-ში, ისევე როგორც C/C++-ში, მოიცემა ორმაგ ბრჭყალებში. ისინი მიეკუთვნება ტიპს String.

```
Prelude>"hello"
"hello" :: String
```

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

```
Prelude>head "hello"
'h' :: Char
Prelude>tail "hello"
"ello" :: [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
```

შეცდომა დაფიქსირდება, თუ ფუნქცია `show` ვერ გარდაქმნის სტრიქონს რიცხვად.

როგორც სხვა ენებში (მაგალითად, `C/C++`-ში), არსებობს მმართველი სიმბოლოების (მაგალითად, `'\n','\t'`) გამოყენების შესაძლებლობა:

```
Prelude> putStrLn "Here's a newline -->\n<--
See?"
Here's a newline -->
<-- See?
```

ფუნქცია `putStrLn`-ის შესრულების შემდეგ სტრიქონი გამოჩნდება კომპიუტერის ეკრანზე. მასთან დაშვებულია ტაბულაციის `(\t)`, ახალი სტრიქონისა `(\n)` და ხმოვანი სიმბოლოს `(\a)` გამოყენება.

`Haskell`–ში სიმბოლო ერთმაგ ბრჭყალებში ჩაისმის:

```
Prelude> 'a'
'a'
```

ტექსტური სტრიქონი წარმოადგენს სიმბოლოების თანმიმდევრობას:

```
Prelude> let a = ['l', 'o', 't', 's', ' ',
'o', 'f', ' ', 'w', 'o', 'r', 'k']
Prelude> a
"lots of work"
Prelude> a == "lots of work"
True
```

ცარიელი სტრიქონი ჩაიწერება `""` და წარმოადგენს `[]`-ის სინონიმს.

```
Prelude> "" == []
True
```

რადგან სტრიქონი არის სიმბოლოების სია, ამიტომ ახალი სტრიქონის ასაგებად შეიძლება გამოვიყენოთ ოპერატორები `:` და `++`.

```
Prelude> 'a':"bc"
"abc"
Prelude> "foo" ++ "bar"
"foobar"
```

ასევე, სიებზე მომუშავე ნებისმიერი პოლიმორფული ფუნქცია შეიძლება სტრიქონებზეც გამოვიყენოთ:

```
Prelude > length "abcde"
5
Prelude > take 3 "abcde"
"abc"
Prelude > zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

თავი 1.3. ტიპები და ფუნქციები

ტიპების სისტემა HASKELL-ში

Haskell-ში ყოველ გამოსახულებას და ფუნქციას აქვს ტიპი. ამ ენის ტიპების სისტემას ახასიათებს სამი რამ: მკაცრი ტიპიზაცია, სტატიკურობა და ავტომატურად განსაზღვრის შესაძლებლობა. განვიხილოთ თითოეული მათგანი.

მკაცრი ტიპები

როცა ვამბობთ, რომ Haskell-ს აქვს მკაცრი ტიპიზაცია, ვგულისხმობთ, რომ ტიპების სისტემით გარანტირებულია, პროგრამა არ შეიცავდეს გარკვეული სახის შეცდომებს. მაგალითად, თუ არის მცდელობა, რომ ფუნქციას, რომელიც მუშაობს მთელ რიცხვებზე და მას გადაეცემა სტრიქონი, კომპილატორმა უნდა ამოაგდოს შეცდომა. ასეთ ენას ეწოდება მკაცრად ტიპიზებული. მკაცრი ტიპიზაციის კიდევ ერთი ასპექტია ის, რომ არ ხდება ერთი ტიპის მეორეში ავტომატურად გარდაქმნა. მაგალითად, თუ C/C++-ის კომპილერს გადავცემთ მთელ რიცხვს, როცა ფუნქცია ელოდება ნამდვილ პარამეტრს, მოხდება გარდაქმნა მთელი რიცხვისა ნამდვილად, მაშინ, როცა Haskell-ის კომპილერი მსგავს სიტუაციაში აფიქსირებს კომპილაციის შეცდომას.

მკაცრი ტიპიზაციის დიდი უპირატესობაა ის, რომ იგი აფიქსირებს კოდის რეალურ შეცდომებს მანამ, სანამ ეს შეცდომები გამოიწვევს პრობლემებს.

სტატიკური ტიპები

სტატიკური ტიპიზაცია ნიშნავს, რომ კომპილატორმა იცის თითოეული მნიშვნელობის და გამოსახულების ტიპი კომპილაციის დროს, კოდის შესრულებამდე (დინამიკური ტიპიზაციისგან განსხვავებით, როცა ტიპი დგინდება შესრულებისას). მაგალითად:

```
Prelude> True && "false"

<interactive>:1:8:
    Couldn't match expected type `Bool'
against inferred type `[Char]'
    In the second argument of `(&&)', namely
`"false"'
    In the expression: True && "false"
    In the definition of `it': it = True &&
"false"
```

ეს შეტყობინება შეცდომის შესახებ ეხება `(&&)` ოპერატორის მეორე პარამეტრს.

ტიპების გამოყვანა

`Haskell`-ის კომპილატორს შესაძლებლობა აქვს გამოიტანოს თითქმის ყველა გამოსახულების ტიპი. ამ პროცესს ეწოდება ტიპების გამოყვანა. ამ პროცესზე ვისაუბრებთ მოგვიანებით, ტიპების კლასების, პოლიმორფული ტიპების განმარტების შემდეგ.

ჩვენ უკვე შემოვიტანეთ შემდეგი ძირითადი ბაზური ტიპები:

- `Char`-ის მნიშვნელობები შეესაბამება `Unicode` სიმბოლოებს.

- ტიპი `Bool`-ის მნიშვნელობები წარმოადგენს ლოგიკურ მნიშვნელობებს: `True` და `False`.
- ტიპი `Int` გამოიყენება ფიქსირებული სიგრძის მთელი რიცხვების წარმოსადგენად. `Int`-ის დიაპაზონი საზოგადოდ `32` ბიტია (`32`-ბიტიან მანქანაზე) და `64` ბიტია (`64`-ბიტიან მანქანაზე). `Haskell`-ის სტანდარტში მოცემულია, რომ `Int` უფრო გრძელია, ვიდრე `28` ბიტი (რა თქმა უნდა, არსებობს ენაში `8, 16` და ა.შ. ბიტიანი მთელი რიცხვითი ტიპები).
- `Integer` მნიშვნელობა არის შეუზღუდავი ზომის. იგი არ გამოიყენება ისეთი სიხშირით, როგორც `Int`.
- მცოცავმძიმიანი რიცხვებისთვის გამოიყენება ტიპი `Double`. მისი მნიშვნელობა, როგორც წესი, არის `64` ბიტი. `Float` ტიპი არსებობს, მაგრამ მისი გამოყენება რეკომენდებული არაა, ვინაიდან იგი გაცილებით ნელია, ვიდრე `Double`.

როდესაც ტიპს განვსაზღვრავთ, ვიყენებთ ნოტაციას `expression :: MyType` და ვამბობთ, რომ `expression` აქვს ტიპი `MyType`. ინტერპრეტატორს შეუძლია ტიპის განსაზღვრა `:type`-ის გამოყენებით.

```
Prelude> :type 'a'
'a' :: Char
Prelude> 'a' :: Char
'a'
Prelude> [1,2,3] :: Int

<interactive>:1:0:
    Couldn't match expected type `Int' against
    inferred type `[a]'
    In the expression: [1, 2, 3] :: Int
    In the definition of `it': it = [1, 2, 3]
    :: Int
```

ზემოთ ჩამოთვლილი მარტივი ტიპების გარდა, Haskell-ში შეიძლება განვსაზღვროთ შედგენილი ტიპის მნიშვნელობებიც, ანუ რამდენიმე მნიშვნელობა (ერთი და იმავე ან სხვადასხვა ტიპის), რომლებიც გაერთიანებულია მრგვალ ფრჩხილებში და ერთმანეთისგან მძიმით გამოიყოფა. ასეთ მნიშვნელობას კორტეჟი ეწოდება. მაგალითად, სიბრტყეზე წერტილების მოსაცემად აუცილებელია ორი რიცხვი, რომლებიც მათ კოორდინატებს შეესაბამება. ენაში რიცხვების წყვილი შეიძლება განვსაზღვროთ ასე: ჩამოვთვალოთ კომპონენტები, გამოვყოთ მძიმეებით და მოვაქციოთ ფრჩხილებში:(5,3). არ არის აუცილებელი, რომ რიცხვების კომპონენტები იყოს ერთი და იმავე ტიპის. შეიძლება შევადგინოთ წყვილი, რომლის პირველი კომპონენტი შეიძლება იყოს სტრიქონი, მეორე - მთელი რიცხვი და ა.შ. ხშირად ორადგილიან კორტეჟს წყვილს უწოდებენ, სამადგილიან კორტეჟს - სამეულს 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 ცნებას. პირველი მათგანი წარმოადგენს მნიშვნელობას, მაშინ, როცა მეორე არის ტიპი.

კორტეჟი წარმოადგენს C/C++ ენის სტრუქტურის ანალოგს. მაგალითად, წიგნის შესახებ ინფორმაცია შეიძლება კორტეჟით წარმოვადგინოთ:

```
Prelude> (1964, "Labyrinths")
(1964,"Labyrinths")
```

კორტეჟში როგორც მნიშვნელობები გამოიყოფა მძიმით, ასევე მისი ტიპი გამოიყოფა მძიმით. ამასთან, რამდენ ადგილიანიცაა კორტეჟი, იმდენი სახელია მის ტიპშიც:

```
Prelude> :type (True, "hello")
(True, "hello") :: (Bool, [Char])
Prelude> (4, ['a', 'm'], (16, True))
(4,"am",(16,True))
```

არსებობს სპეციალური ტიპი (), რომელიც მოქმედებს როგორც ნულოვან-ელემენტიანი კორტეჟი. () - ეს არის C/C++ –ის void-ის ანალოგიური.

Haskell-ში ერთელემენტიანი კორტეჟის ცნება არ არსებობს. კორტეჟში ელემენტების რაოდენობების აღსანიშნავად წერენ *რაოდენობა-კორტეჟი,* მაგალითად, ორელემენტიან კორტეჟს ასე წერენ: 2-კორტეჟი.

```
Prelude> :type (False, 'a')
(False, 'a') :: (Bool, Char)
Prelude> :type ('a', False)
('a', False) :: (Char, Bool)
```

ჩანს, რომ გამოსახულებას `(False, 'a')` აქვს ტიპი `(Bool, Char)`, რომელიც განსხვავდება ტიპისგან `('a', False)`.

```
Prelude> :type (False, 'a', 'b')
(False, 'a', 'b') :: (Bool, Char, Char)
```

წყვილებთან სამუშაოდ ენა Haskell-ში არსებობს სტანდარტული 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
```

შევნიშნოთ, რომ 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'` ელემენტის ამოღებას კორტეჟიდან `(1, ('a', 23.12))`:

```
Prelude>fst (snd (1, ('a', 23.12)))
'a' :: Integer
```

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)
```

სავარჯიშოები

1. რა ტიპისაა შემდეგი გამოსახულებები?

   - ✓ `False`
   - ✓ `(["foo", "bar"], 'a')`
   - ✓ `[(True, []), (False, [['a']])]`

2. მოიყვანეთ არატრივიალური გამოსახულებების მაგალითები, რომლებიც ეკუთვნის ტიპებს:

   - ✓ `((Char,Integer), String, [Double])`
   - ✓ `[(Double,Bool,(String,Integer))]`
   - ✓ `([Integer],[Double],[(Bool,Char)])`
   - ✓ `[[[(Integer,Bool)]]]`
   - ✓ `(((Char,Char),Char),[String])`
   - ✓ `(([Double],[Bool]),[Integer])`
   - ✓ `[Integer, (Integer,[Bool])]`
   - ✓ `(Bool,([Bool],[Integer]))`
   - ✓ `[([Bool],[Double])]`
   - ✓ `[([Integer],[Char])]`

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

ფუნქციათა გამოძახება

მათემატიკაში ფუნქციის გამოყენება აღინიშნება ფრჩხილების ხმარებით, ხოლო გამრავლება ხშირად აღინიშნება გვერდიგვერდ განლაგების და ინტერვალის ხმარებით. მაგალითად: გამოსახულება

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

ნიშნავს – გამოვიყენოთ `f` ფუნქცია `a`-სა და `b`-ს მიმართ და შედეგს მივუმატოთ `c`-ს და `d`-ს ნამრავლი.

`Haskell`-ში ფუნქციის გამოყენება აღინიშნება ხარვეზის (ინტერვალის) ხმარებით, ხოლო გამრავლება აღინიშნება *-ით, ანუ წინა მაგალითს `Haskell`-ის სინტაქსით ექნება სახე: `f a b + c*d`.

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

```
f a + b
```

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

ფუნქციის გამოყენების მაგალითებია:

```
Prelude> odd 3
True
Prelude> odd 6
False
```

ფუნქციის არგუმენტების გამოსაყოფად მრგვალი ფრჩხილები ან მძიმეები არ გამოიყენება. `odd` არის `Prelude` ბიბლიოთეკის

პრედიკატი, რომელიც არკვევს, არის თუ არა ამ ფუნქციის არგუმენტი კენტი.

მოცემული ცხრილი გვიჩვენებს ფუნქციის ჩაწერის თავისებურებებს Haskell-ში:

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

განვიხილოთ ორარგუმენტიანი ფუნციების გამოძახების მაგალითები:

```
Prelude> compare 2 3
LT
Prelude> compare 3 3
EQ
Prelude> compare 3 2
GT
```

შედარების ფუნქციას (compare) აქვს უფრო დიდი პრიორიტეტი, ვიდრე სხვა ოპერატორებს, ამიტომ შემდეგი ორი გამოსახულება ერთი და იმავე მნიშვნელობისაა:

```
Prelude> (compare 2 3) == LT
True
Prelude> compare 2 3 == LT
True
```

ზემოთ მოყვანილ გამოსახულებაში ფრჩხილები აუცილებელი არაა, განსხვავებით ქვემოთ მოყვანილი გამოსახულებისა:

```
Prelude> compare (sqrt 3) (sqrt 6)
LT
```

ფრჩხილების გარეშე გამოდის, რომ ფუნქციას გადავცემთ ორ არგუმენტს.

შედგენილი ტიპები შედგება მარტივი ტიპებისგან. ჩვენ უკვე ვნახეთ, რომ სტრიქონი არის „Char-ის მასივი“ და დაიწერება, როგორც `[Char]`.

ფუნქცია `head` აბრუნებს სიის პირველ ელემენტს.

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

ფუნქცია `tail` აბრუნებს სიას პირველი ელემენტის გარეშე.

```
Prelude> tail [1,2,3,4]
[2,3,4]
Prelude> tail [2,3,4]
[3,4]
Prelude> tail [True,False]
[False]
Prelude> tail "list"
"ist"
Prelude> tail []
*** Exception: Prelude.tail: empty list
```

ვინაიდან სიაში მნიშვნელობები შეიძლება იყოს ნებისმიერი ტიპის, სიას უწოდებენ *პოლიმორფულს*. ეს აღინიშნება ასე: `[a]`.

ტიპებისა და ცვლადების სახელები განსხვავდება იმით, რომ ტიპების სახელები იწყება დიდი ასოთი.

როგორც უკვე აღვნიშნეთ, ბრძანება `:type` ბეჭდავს გამოსახულების ტიპს:

```
Prelude> :type [[True],[False,False]]
[[True],[False,False]] :: [[Bool]]
```

ფუნქციები `take` და `drop` ორარგუმენტიანია; პირველი არგუმენტია რიცხვი (მაგ. `n`), მეორე – სია. ფუნქცია `take` აბრუნებს სიის პირველ `n` ელემენტს, ხოლო ფუნქცია `drop` აბრუნებს სიას `n` ელემენტის გარეშე.

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

ფუნქციები `fst` და `snd` აბრუნებს კორტეჟის წყვილის პირველ და მეორე ელემენტებს, შესაბამისად:

```
Prelude> fst (1,'a')
1
Prelude> snd (1,'a')
'a'
```

განვიხილოთ ფუნქციისთვის გამოსახულების გადაცემა. `Haskell`-ში ფუნქციების გამოყენება მარცხნივ ასოციატიურია. მა-

გალითად, გამოსახულება `a b c d` ეკვივალენტურია `(((a b) c) d)`. თუ ჩვენ გვინდა ერთი გამოსახულება გამოვიყენოთ არგუმენტად სხვისთვის, მაშინ ცხადად უნდა მივუთითოთ ფრჩხილები.

```
Prelude> head (drop 4 "azerty")
't'
```

ფუნქციის ტიპი

განვიხილოთ, მაგალითად, ფუნქცია `lines`, რომელიც არგუმენტს (სტრიქონის ტიპის) ყოფს სტრიქონებად მმართველი სიმბოლოების შესაბამისად:

```
Prelude> :type lines
lines :: String -> [String]
```

ეს ჩანაწერი ასე იკითხება: „`lines`-ს აქვს ტიპი `String`, რომელსაც შეუსაბამებს სტრიქონების სიას“. ნიშანი `->` იკითხება „აბრუნებს“.

```
Prelude> lines "the quick\nbrown fox\njumps"
["the quick","brown fox","jumps"]
Prelude>  lines "aa\nbb\nbb"
["aa","bb","bb"]
```

გვერდითი ეფექტები ახასიათებს პროცედურული ენების ფუნქციებს. განვიხილოთ ფუნქცია, რომელიც არგუმენტად იღებს გლობალურ ცვლადს და ცვლის მას. თუ რომელიმე კოდს შეუძლია შეცვალოს ეს გლობალური ცვლადი, მაშინ ფუნქციის მნიშვნელობა

დამოკიდებული იქნება ამ ცვლადის მნიშვნელობაზე, რასაც გვერდითი ეფექტი ჰქვია.

Haskell-ის ფუნქციებს არ აქვთ გვერდითი მოვლენები და ამის გამო მათ ეწოდება წმინდა (სუფთა) ფუნქციები.

თუ ფუნქციას აქვს გვერდითი მოვლენები, ამის შესახებ შეგვიძლია გავიგოთ ფუნქციის ტიპიდან – ის იწყება სიტყვა IO -დან.

```
Prelude> :type readFile
readFile :: FilePath -> String
```

მარტივი ფუნქციების შექმნა

ჩვენ აქამდე ვიყენებდით ენა Haskell-ის სტანდარტულ ფუნქციებს. ვნახოთ, როგორ შეიძლება განისაზღვროს მომხმარებლის ფუნქციები. გავიხსენოთ ინტერპრეტატორის რამდენიმე ბრძანება (შესაძლებელია ამ ბრძანებების შემოკლება ერთ ასომდე), რომლებსაც გამოვიყენებთ პროგრამების შესასრულებლად:

- 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 (.hs გაფართოების), 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 რედაქტორი. 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 lab1.hs. დავუშვათ, ამ ფაილის სრული გზაა – c:\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 c:\labs\lab1.hs და ახლა x ცვლადის მნიშვნელობა იქნება განსაზღვრული:

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

ყურადღება მივაქციოთ, რომ ფაილის სახელის ჩაწერისას `:load` ბრძანების არგუმენტში სიმბოლო `\` 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 შევქმნათ რაიმე ფაილი და ჩავწეროთ მასში შემდეგი ტექსტი:

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

პირველი სტრიქონი `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, ფუნქციასაც 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 დაპროგრამებისას ფუნქციის ტიპის ცხადად მითითება ითვლება „კარგ ტონად“, რადგანაც ტიპის გამოცხადება ემსახურება რაღაც აზრით ფუნქციის დოკუმენტაციას და ეხმარება დაპროგრამების შეცდომების გამოვლენას.

მომხმარებლის 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'
```

შემდეგი მაგალითი: დავწეროთ ორი რიცხვის შეკრების ფუნქცია. შევნიშნოთ, რომ არგუმენტების ტიპს არ ვაკონკრეტებთ. ავკრიფოთ შემდეგი ტექსტი Notepad-ში, შევინახოთ სახელით add.hs; შემდეგ გავხსნათ ბრძანებით :load.

```
-- file: ch03/add.hs
add a b = a + b
```

```
Prelude> :load add.hs
[1 of 1] Compiling Main  ( add.hs, interpreted )
Ok, modules loaded: Main.
Prelude> add 1 2
3
```

:cd ბრძანებით შეგვიძლია კატალოგის შეცვლა:

```
Prelude> :cd /tmp
```

გარდა ამისა, შესაძლოა :load ბრძანებაში მივუთითოთ სრული გზა. მაგალითად,

```
::load „C:\\Documents and Settings
\\Administrator\\ Desktop\\ HASKELL\\
examples\\add1.hs“
```

Haskell-ში არ გვაქვს მინიჭების ოპერატორი. ცვლადს მნიშვნელობა უკავშირდება მხოლოდ ერთხელ და მისი შეცვლა არ შეიძლება.

```
-- file: ch02/Assign.hs
x = 10
x = 11
```

თუ ამ ფაილს გავხსნით ინტერპრეტატორში, იქნება შეცდომა:

```
Prelude> :load Assign
[1 of 1] Compiling Main ( Assign.hs, interpreted )

Assign.hs:4:0:
    Multiple declarations of `Main.x'
    Declared at: Assign.hs:3:0
                 Assign.hs:4:0
Failed, modules loaded: none.
```

პირობითი გამოსახულება

ენა Haskell-ში ფუნქციის განსაზღვრებისას შესაძლებელია გამოვიყენოთ 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*.

გასაღები სიტყვა `if` შედგება სამი კომპონენტისაგან:

- `Bool`-ის ტიპის გამოსახულება არის პრედიკატი.
- `then` – გამოითვლება, თუ პრედიკატის მნიშვნელობა არის ჭეშმარიტი.
- `else` გამოსახულება გამოითვლება, თუ პრედიკატის მნიშვნელობა არის `False`.

`then` და `else` გამოსახულებებს უნდა ჰქონდეთ ერთი და იგივე ტიპი.

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

კომპონენტისაგან:

- `<, >, <=, >=` – 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
```

ახლა ფუნქცია `signum` შეიძლება განვსაზღვროთ შემდეგნაირად:

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

შევნიშნოთ, რომ ფუნქცია `isPositive` შეიძლება განვსაზღვროთ მარტივადაც:

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

`Haskell`-ში, ისევე როგორც სხვა დაპროგრამების ენებში, არსებობს `if` პირობითი ოპერატორი. გავიხსენოთ, თუ როგორ მუშაობს ოპერატორი `drop`:

```
Prelude> drop 2 "foobar"
"obar"
Prelude> drop 4 "foobar"
"ar"
```

```
Prelude> drop 4 [1,2]
[]
Prelude> drop 0 [1,2]
[1,2]
Prelude> drop 7 []
[]
Prelude> drop (-2) "foo"
"foo"
```

როგორც ვხედავთ, `drop` აბრუნებს თავდაპირველ სიას, თუ ნომერი (პირველი არგუმენტი) ნაკლებია ან ტოლი ნულის. წინააღმდეგ შემთხვევაში, აგდებს ელემენტებს იმდენჯერ, სანამ არ მივა მოცემულ რიცხვამდე. განვსაზღვროთ ფუნქცია `myDrop`, რომელიც იყენებს `if` გამოსახულებას. გამოყენებული ფუნქცია `null` ამოწმებს, არის თუ არა სია ცარიელი.

```
-- file: ch02/myDrop.hs
myDrop n xs = if n <= 0 || null xs
              then xs
              else myDrop (n - 1) (tail xs)
```

შევინახოთ ფაილი სახელით `myDrop.hs`, შემდეგ კი ჩავტვირთოთ `ghci` ინტერპრეტატორში.

```
Prelude> :load myDrop.hs
[1 of 1] Compiling Main ( myDrop.hs, interpreted )
Ok, modules loaded: Main.
Prelude> myDrop 2 "foobar"
"obar"
```

```
Prelude> myDrop 4 "foobar"
"ar"
Prelude> myDrop 4 [1,2]
[]
Prelude> myDrop 0 [1,2]
[1,2]
Prelude> myDrop 7 []
[]
Prelude> myDrop (-2) "foo"
"foo"
```

ამ კოდში სიმბოლოები `--` ერთსტრიქონიან კომენტარს წარმოადგენს.

ვნახოთ, რომელ ოპერატორებს შეიცავს პრედიკატი. `null` ფუნქცია უჩვენებს, არის თუ არა სია ცარიელი, ხოლო ოპერატორი `(||)` ასრულებს ლოგიკურ „ან“-ს.

```
Prelude> :type null
null :: [a] -> Bool
Prelude> :type (||)
(||) :: Bool -> Bool -> Bool
```

ყურადღება მიაქციეთ იმას, რომ ფუნქცია `myDrop` იყენებს რეკურსიულ მიმართვას.

შესაძლოა, მთელი ფუნქცია დაწეროთ ერთ სტრიქონზეც, მაგრამ ის ძალზე ცუდად წაიკითხება:

```
-- file: ch02/myDrop.hs
myDropX n xs = if n <= 0 || null xs then xs
else myDropX (n - 1) (tail xs)
```

Haskell-ში ლოგიკური ფუნქცია ან (||) შეიძლება ასეც განვმარტოთ:

```
-- file: ch02/shortCircuit.hs
newOr a b = if a then a else b
```

თუ დავწერთ ასეთ მიმართვას: newOr True (length [1..] > 0), მაშინ მეორე პარამეტრის გამოთვლა არ მოხდება, ასე რომ, შეცდომაც არ დაფიქსირდება.

პოლიმორფიზმი HASKELL-ში

როცა წარმოვადგინეთ სიები, ვთქვით, რომ მათი ტიპი არის პოლიმორფული.

ვთქვათ, ვიყენებთ სიის ბოლო ელემენტის ამოღების ფუნქცია last-ს. არ აქვს მნიშვნელობა, თუ რა ტიპის პარამეტრი გადაეცემა მას.

```
Prelude> last [1,2,3,4,5]
5
Prelude> last "baz"
'z'
```

დავაბეჭდინოთ მისი ტიპი:

```
Prelude> :type last
last :: [a] -> a
```

აქ a არის ტიპის ცვლადი. ეს ჩანაწერი ასე იკითხება: last ფუნქციის პარამეტრია სია, რომლის ელემენტებია ნებისმიერი a ტიპის. ფუნქცია აბრუნებს იმავე a ტიპის მნიშვნელობას.

როდესაც გვინდა გამოვიყენოთ ფუნქცია last, ვთქვათ Char-ის სიებთან, მაშინ კომპილერი ცვლის Char-ით a-ს და შედეგად გვაძლევს last-ის ტიპს, როგორც [Char] -> Char.

ასეთი ტიპის პოლიმორფიზმს ეწოდება პარამეტრული პოლიმორფიზმი. დასახელება გულისხმობს: როგორც ფუნქციას აქვს პარამეტრები, რომლებსაც უკავშირდება რეალური მნიშვნელობები, ასევე Haskell-ის ტიპს შეიძლება ჰქონდეს პარამეტრი, რომელსაც შემდეგ დაუკავშირდება სხვა ტიპი.

Haskell-ის პარამეტრულმა პოლიმორფიზმმა გავლენა მოახდინა Java და C#-ის პროექტირებაზე (Java generics). C++-ის შაბლონებიც წააგავს Haskell-ის პარამეტრულ პოლიმორფიზმს.

ვნახოთ, მაგალითად, ფუნქცია take-ის ტიპი.

```
Prelude> :type take
take :: Int -> [a] -> [a]
```

თუ განვმარტავთ, რომ ისარი -> მარჯვნივ ასოციურია, მაშინ ცხადი ხდება, რომ ეს ჩანაწერი არის ეკვივალენტური ჩანაწერის:

```
take :: Int -> ([a] -> [a])
```

მრავალარგუმენტიანი ფუნქცია დაიყვანება კარირების ოპერაციით ერთ არგუმენტიანზე, რასაც შემდეგ განვიხილავთ.

სავარჯიშოები

1. Haskell-ში განსაზღვრულია last :: [a] -> a, რომელიც აბრუნებს სიის ბოლო ელემენტს. მოიყვანეთ ამ ფუნქციის განსაზღვრის ვერსია.

2. დაწერეთ ფუნქცია lastButOne, რომელიც დააბრუნებს სიის ბოლოდან მეორე ელემენტს.

მრავალი ცვლადის ფუნქცია და ფუნქციების განსაზღვრის რიგი

ჩვენ გამოვიძახეთ ფუნქციები, რომლებიც ერთ არგუმენტს იღებდნენ. რა თქმა უნდა, Haskell ენაში შესაძლებელია განვსაზღვროთ ფუნქციები, 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 -> ასოციაციურია მარჯვნიდან. ასე რომ, add ფუნქციის ტიპი შეიძლება ასე წავიკითხოთ Integer -> (Integer -> 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 ეკუთ-

ვნის `t1, t2, . . . , tn,` ტიპებს და აბრუნებს `a` ტიპის შედეგს, ჩაიწერება სახით `t1->t2->...->tn->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 არა აქვს მნიშვნელობა! ასე რომ, ფუნქცია `isPositive` შეიძლება განისაზღვროს როგორც `signum` ფუნქციის განსაზღვრამდე, ისე მის შემდეგ.

კომენტარები

ცხადია, რომ აუცილებელია პროგრამაში კომენტარების არსებობა. `Haskell` ენაში, ისევე, როგორც `C/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 `Haskell` ინტერპრეტატორის მიერ. მაგალითად,

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

Haskell-ში არსებობს რამდენიმე ჩადგმული გამოსახულება, რომელიც აადვილებს ფუნქციის აგებას და კოდს ხდის უფრო გასაგებს. ეს გამოსახულებები შეიძლება დაიყოს ორ ჯგუფად: გამოსახულებები, რომლებიც მხარს უჭერს ფუნქციის განსაზღვრის დეკლარაციულ სტილს (declarative style) და გამოსახულებები, რომლებიც მხარს უჭერს კომპოზიციურ სტილს (expression style). დეკლარაციული სტილით ფუნქციის განსაზღვრა უფრო წააგავს მათემატიკურ ნოტაციას. კომპოზიციური სტილის დროს ვაგებთ მარტივი გამოსახულებებიდან უფრო რთულ გამოსახულებებს, ვიყენებთ ამ გამოსახულებებთან სხვა გამოსახულებებს და ვაგებთ კიდევ უფრო დიდ გამოსახულებებს.

Haskell სრულად უჭერს მხარს ორივე სტილს. სტილის არჩევა, საზოგადოდ, პროგრამისტის გემოვნებაზეა დამოკიდებული.

ლოკალური ცვლადები

გავიხსენოთ სამკუთხედის ფართობის გამოთვლის ფორმულა სამკუთხედის სამი გვერდის მიხედვით, რომელიც ჰერონის ფორმულის სახელითაა ცნობილი:

$$S = \sqrt{p*(p-a)*(p-b)*(p-c)}$$

სადაც a, b და c – სამკუთხედის გვერდებია, ხოლო p არის ნახევარპერიმეტრი.

როგორ განვსაზღვროთ ეს ფუნქცია? ყველაზე მარტივად შეგვიძლია დავწეროთ ასე:

```
square a b c = sqrt(p a b c * (p a b c - a) *
(p a b c - b) * (p a b c - c))
p a b c = (a + b + c) / 2
```

ეს ჩანაწერი ნამდვილად სჯობს შემდეგ ჩანაწერს:

```
square a b c = sqrt ((a+b+c)/2 * ((a+b+c)/2 -
a) * ((a+b+c)/2 - b) * ((a+b+c)/2 - c))
```

ორივე ჩანაწერში ხდება გამოსახულების გამოთვლის დუბლირება. სასურველია განსაზღვრება ისევე გამოიყურებოდეს, როგორც მათემატიკური განსაზღვრება:

```
square a b c = sqrt (p * (p - a) * (p - b) * (p - c))
p = (a + b + c) / 2
```

საჭიროა `p`-მ იცოდეს, რომ `a`, `b` და `c` აიღება ფუნქციის არგუმენტებიდან. ასეთ შემთხვევებში გამოიყენება ლოკალური ცვლადები.

`Where` გამოსახულება

დეკლარაციული სტილისთვის გამოიყენება `where`-გამოსახულება, რომელიც ასე იწერება:

```
square a b c = sqrt (p * (p - a) * (p - b) * (p - c))
        where p = (a + b + c) / 2
```

ან ასე:

```
square a b c = sqrt (p * (p - a) * (p - b)
* (p - c)) where
p = (a + b + c) / 2
```

ფუნქციის განსაზღვრას მოსდევს სიტყვა `where`, რომელსაც შემოყავს ლოკალური ცვლადების სინონიმები. ამასთან, ფუნქციის არგუმენტები იმყოფებიან სახელთა ხილვადობის არეში. სინონიმები შეიძლება იყოს რამდენიმე:

```
square a b c = sqrt (p * pa * pb * pc)
where p = (a + b + c) / 2
pa = p - a
pb = p - b
pc = p - c
```

ლოკალური გამოსახულებების რიგი `where` გამოსახულებაში არ არის მნიშვნელოვანი.

`where` გამოსახულებაში შეიძლება განისაზღვროს ახალი ფუნქციები და ასევე, აღიწეროს მათი ტიპები:

```
add2 x = succ (succ x)
where succ :: Int -> Int
      succ x = x + 1
```

ფუნქციის ტიპი შეიძლება არც აღიწეროს, ინტერპრეტატორი თვითონ მიხვდება:

```
add2 x = succ (succ x)
where succ x = x + 1
```

თუმცა ზოგჯერ სასარგებლოა ფუნქციის ტიპის მითითება, როცა გამოიყენება ტიპების კლასები.

განვიხილოთ კიდევ ერთი მაგალითი – სიის ფილტრაციის ფუნქცია, რომელიც განსაზღვრულია `Prelude`-ში:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : rest else
rest
where rest = filter p xs
```

აქ განსაზღვრულია ლოკალური ცვლადი `rest`, რომელიც მიუთითებს ფუნქციის რეკურსიულ გამოძახებას სიის დარჩენილი ნაწილით.

`Where` გამოსახულებები განისაზღვრება ფუნქციის განსაზღვრაში თითოეული განტოლებისთვის:

```
even :: Int -> Bool
even Zero = res
     where res = True
even (Succ Zero) = res
     where res = False
even x = even res
 where (Succ (Succ res)) = x
```

რა თქმა უნდა, ამ მაგალითში `where` არ არის საჭირო, მაგრამ მოყვანილია იმის საჩვენებლად, თუ როგორ შეიძლება `where` განმარტება მიებას მოცემულ განტოლებას. მაგალითში განსაზღვრულია სამი ლოკალური ცვლადი ერთი და იმავე სახელით. `where` გამოსახულებები შეიძლება მოიცეს `where`-ს შიგნითვე, მაგრამ ჯობს ღრმად ჩადგმულ გამოსახულებებს მოვერიდოთ.

Let გამოსახულება

კომპოზიციური სტილით წერისას ლოკალური ცვლადების განსაზღვრისთვის გამოიყენება `let` დაკავშირება.

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

```
root s 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 მოიცავს: ჯერ ერთი, შეიძლება ადვილად დავუშვათ შეცდომა ერთი და იმავე გამოსახულების ორჯერ დაწერისას; მეორე, ამ პროგრამის დაწერისას საჭიროა შევადაროთ ორი გამოსახულება, რათა დავრწმუნდეთ, რომ ერთი და იგივეა; მესამე, პროგრამა უფრო გრძელი ხდება. დაბოლოს, ის ნაკლებად ეფექტურია, ვინაიდან კომპიუტერი ითვლის ერთსა და იმავე გამოსახულებას ორჯერ.

ამ პრობლემის ასაცილებლად ენაში შემოტანილია ლოკალური ცვლადის ცნება. ფუნქცია შეიძლება ჩაიწეროს 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)
```

შემდეგი მაგალითი: დავწეროთ კომპოზიციურ სტილში სამკუთხედის ფართობის გამოთვლის ფუნქცია. მას აქვს შემდეგი სახე:

```
square a b c = let p = (a + b + c) / 2
in sqrt (p * (p - a) * (p - b) * (p - c))
```

სიტყვები `let` და `in` – გასაღები სიტყვებია. ისინი შეიძლება გამოვიყენოთ გამოსახულების ნებისმიერ ადგილას:

```
square a b c = let p = (a + b + c) / 2
     in sqrt ((let pa = p - a in p * pa) *
             (let pb = p - b
                  pc  = p - c
              in pb * pc))
```

ამით ჩანს, რომ ისინი მიეკუთვნებიან კომპოზიციურ სტილს. `let` გამოსახულებები შეიძლება მონაწილეობდეს ნებისმიერ ქვეგამოსახულებაში, შეიძლება დაჯგუფდეს ფრჩხილებით, ხოლო `where` გამოსახულებები მიბმულია ფუნქციის განმარტების განტოლებებთან.

განვსაზღვროთ ფილტრაციის ფუნქცია `let`-ის მეშვეობით:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) =
       let rest = filter p xs
       in if p x then x : rest else rest
```

`let ... in ...` კონსტრუქციის 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 `n` რიცხვის მიხედვით ნატურალური რიცხვების სიას `[1,2,...,n]`. შემოვიტანოთ დამხმარე ფუნქცია `numsFrom`, რომელიც მოცემული `m` 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 განსაზღვრებაში იყენებს ცვლადს `n`, თუმცა მას იგი არ გადაეცემა პარამეტრად.

ოპერატორების განსაზღვრება

ბინარული ოპერატორები, როგორიცაა +, – და ა.შ., ენა Haskell-ში ისეთივე ფუნქციებს ასრულებს, რასაც ყველა დანარჩენი, მხოლოდ ერთი გამონაკლისით, – მათი გამოძახება შეიძლება ინფიქსური ნოტაციითაც. თუ ბინარულ ოპერატორს ფრჩხილებში მოვაქცევთ, მაშინ მისი გამოძახებისთვის შეიძლება გამოვიყენოთ პრეფიქსული ნოტაცია და მივმართოთ მას, როგორც ჩვეულებრივ ფუნქციას. ასე რომ, შემდეგი ჩანაწერები ეკვივალენტურია:

```
2 + 2
(+) 2 2
```

```
x < y
(<) x y
```

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

ნებისმიერი ფუნქცია, რომელიც ღებულობს ორ არგუმენტს, პირიქით, შეიძლება გამოვიყენოთ ინფიქსული სტილითაც. 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 გვხვდება მხოლოდ „სიმბოლოები“ (არა ასოები და არა ციფრები), მაშინ იგი ავტომატურად ითვლება ინფიქსურ ოპერატორად. განსაზღვრისას მისი სახელი აუცილებლად უნდა ჩაისვას ფრჩხილებში. მაგალითად, გან-

ვსაზღვროთ 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 გამოვიყენოთ ისევე, როგორც სხვა დანარჩენი:

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

პოლიმორფული ტიპები

ტიპი არის დაკავშირებულ მნიშვნელობათა ერთობლიობის სახელი. მაგალითად, `Haskell`-ის საბაზო ტიპი `Bool` შეიცავს ორ ლოგიკურ მნიშვნელობას: `False, True`.

ფუნქციაში ერთი ან რამდენიმე არასწორი ტიპის არგუმენტების გამოყენებას *ტიპის შეცდომა* ეწოდება.

```
Prelude> 1 + False
Error
```

შეცდომა გამოიწვია იმან, რომ 1 რიცხვია, ხოლო `False` ლოგიკური მნიშვნელობა, მაგრამ ოპერაცია + მოითხოვს ორ რიცხვს.

ტიპის ყველა შეცდომა ტრანსლაციის დროს ვლინდება, რაც ანიჭებს პროგრამებს მეტ უსაფრთხოებას და სისწრაფეს, რადგან ქრება ტიპების შემოწმების აუცილებლობა შესრულებისას.

ინტერპრეტატორში :type ბრძანება ადგენს გამოსახულების ტიპს ამ გამოსახულების შეუფასებლად:

```
Prelude> not False
True
Prelude> :type not False
not False :: Bool
```

ჩანაწერები:

v :: T ნიშნავს, რომ v-ს აქვს T ტიპი;

False :: Bool ნიშნავს, რომ False არის Bool ტიპის მნიშვნელობა;

True :: Bool მსგავსად ნიშნავს, რომ True-ც ლოგიკური მნიშვნელობისაა;

not :: Bool ->Bool ნიშნავს, რომ not არის ფუნქცია, რომელსაც Bool ტიპის მნიშვნელობის არგუმენტი ისევ იმავე ტიპის მნიშვნელობაში გადაჰყავს. საზოგადოდ, e :: T ჩანაწერი ნიშნავს, რომ e გამოსახულების შეფასება წარმოქმნის T ტიპის მნიშვნელობას. ასე რომ:

```
not(False)   ::      Bool
not(True)            ::      Bool
not(not False)       ::      Bool
```

როგორც უკვე ვახსენეთ, Haskell-ში მუშაობს ე.წ. ტიპის გამოყვანის მექანიზმი. ყოველ გამოსახულებას უნდა გააჩნდეს ტიპი, რომელიც დგინდება გამოსახულების გამოთვლამდე ტიპის გამოტანის/გამოყვანის სახელწოდებით ცნობილი პროცესის მეშვეობით. ამ პროცესის გაგების გასაღებს წარმოადგენს ტიპის დადგენის წესი

ფუნქციის გამოყენებისათვის. ეს წესი გვეუბნება, რომ, თუ f არის ფუნქცია, რომელიც A ტიპის არგუმენტებს ასახავს B ტიპის შედეგში და e არის A ტიპის გამოსახულება, მაშინ f e ფუნქციას B ტიპი აქვს: f :: A ->B e :: A

f e :: B

ჩვენ უკვე გავეცანით საბაზო ტიპებს. ესენია:

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

Haskell-ში განსაზღვრულია სიის ტიპები, როგორც ერთი და იმავე ტიპის მნიშვნელობების მიმდევრობა:

```
[False, True, False]::[Bool]
['a','b','c','d']::[Char]
```

საზოგადოდ, [t]არის იმ სიის ტიპი, რომელიც t ტიპის ელემენტებს შეიცავს. სიის ტიპი არაფერს გვეუბნება მის სიგრძეზე:

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

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

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

Haskell-ში განსაზღვრულია აგრეთვე კორტეჟის ტიპები, როგორც სხვადასხვა ტიპის სიდიდეთა მიმდევრობა:

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

საზოგადოდ: $(t_1, t_2, ..., t_n)$ – n-კორტეჟთა ტიპია, რომელთა კომპონენტებს აქვს t_i ტიპი, სადაც i იღებს მნიშვნელობებს 1...n.

კორტეჟის ტიპი განსაზღვრავს მის ზომას:

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

კომპონენტთა ტიპები არ იზღუდება:

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

Haskell-ში განსაზღვრულია ფუნქციის ტიპები. ფუნქცია არის ერთი ტიპის შეპირისპირება მეორე ტიპის სიდიდეებთან:

```
not     :: Bool -> Bool
isDigit :: Char -> Bool
```

საზოგადოდ, t1 -> t2 ფუნქციების ტიპია, რომლებიც ასახავენ t1 ტიპის სიდიდეებს t2 ტიპის სიდიდეებად. ისარი შედის კლავიატურიდან, როგორც ორი სიმბოლო: ->.

არგუმენტისა და შედეგის ტიპები არ იზღუდება. მაგალითად, ფუნქციები რამდენიმე არგუმენტით ან შედეგით შესაძლებელია სიებს ან კორტეჟებს ეხებოდეს:

```
add        :: (Int,Int) -> Int
add (x,y)   = x+y
zeroto     :: Int -> [Int]
zeroto n    = [0..n]
```

კარირებული ფუნქციები

მრავალარგუმენტიანი ფუნქციები ასევე შესაძლებელია ჩავწეროთ შედეგებად დაბრუნებული ფუნქციების სახით:

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

ეს ჩანაწერი ნიშნავს, რომ `add'` იღებს `x` მთელ რიცხვს და გვიბრუნებს `add' x` ფუნქციას. თავის მხრივ, ეს ფუნქცია იღებს `y` მთელ რიცხვს და გვიბრუნებს `x+y` შედეგს.

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

```
add  :: (Int,Int) -> Int
add' :: Int -> (Int -> Int)
```

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

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

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

ეს ჩანაწერი ნიშნავს, რომ mult იღებს x მთელს და გვიბრუნებს mult x ფუნქციას, რომელიც, თავის მხრივ, იღებს y მთელს და გვიბრუნებს mult x y ფუნქციას, უკანასკნელი იღებს z მთელს და გვიბრუნებს `x*y*z` შედეგს.

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

მაგალითად:

```
add' 1 :: Int -> Int
take 5 :: [Int] -> [Int]
drop 5 :: [Int] -> [Int]
```

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

- ისარი `->` ასოციაციურია მარჯვნივ. ეს კი იმას ნიშნავს, რომ გამოსახულებას `Int -> Int -> Int -> Int` აქვს აზრი `Int -> (Int -> (Int -> Int))`.
- შედეგად, ბუნებრივია ფუნქციისთვის მარცხნიდან დაკავშირების გამოყენება. მაგალითად, ჩანაწერი `mult x y z` ტოლფასია `t ((mult x) y) z` ჩანაწერის.

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

ენა Haskell-ში გამოიყენება ტიპების პოლიმორფული სისტემა. არსებითად, ეს ნიშნავს, რომ ენაში არსებობს ტიპების ცვლადები. ფუნქციას პოლიმორფული (მრავალფორმიანი) ეწოდება, თუ მისი ტიპი შეიცავს ცვლადის ერთ ან რამდენიმე ტიპს. მაგალითად:

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

ეს ნიშნავს, რომ ნებისმიერი `a` ტიპისთვის `length` იღებს ტიპის მნიშვნელობათა სიას და გვიბრუნებს მთელ რიცხვს.

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 იმავე 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 არიან, 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.

სტანდარტულ `prelude` ფაილში განსაზღვრულ ფუნქციათა შორის მრავალი პოლიმორფულია. სიებთან, წყვილებთან და კორტეჟებთან მომუშავე მრავალ ფუნქციას პოლიმორფული ტიპი აქვს. მაგალითად:

```
fst  :: (a,b) -> a
head :: [a] -> a
take :: Int -> [a] -> [a]
zip  :: [a] -> [b] -> [(a,b)]
id   :: a -> a
```

შევნიშნოთ, რომ ზოგიერთი ფუნქციის ტიპის განსაზღვრაში გამოყენებულია ტიპის ორი ცვლადი.

გადატვირთული ფუნქციები

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

```
sum :: Num a ⇒ [a] -> a
```

ნებისმიერი რიცხვითი `a` ტიპისათვის `sum` იღებს `a` ტიპის მნიშვნელობათა სიას და გვიბრუნებს `a` ტიპის მნიშვნელობას. შევ-

ნიშნოთ, რომ პირობებით შეზღუდული ცვლადების ტიპი შეიძლება დამუშავდეს ნებისმიერი ტიპისათვის, რომელიც შეზღუდვებს აკმაყოფილებს:

```
Prelude> sum [1,2,3]
6                          --a=Int
Prelude> sum [1.1,2.2,3.3]
6.6                        --a=Float
Prelude> sum ['a','b','c']
ERROR                   --Char არ არის რიცხვითი ტიპი
```

სავარჯიშოები

1. როგორია შემდეგი მნიშვნელობების ტიპები?

```
['a','b','c']
('a','b','c')
[(False,'0'),(True,'1')]
([False,True],['0','1'])
[tail,init,reverse]
```

2. როგორია შემდეგი ფუნქციების ტიპები?

```
second xs      = head (tail xs)
swap (x,y)     = (y,x)
pair x y       = (x,y)
double x       = x*2
palindrome xs  = reverse xs == xs
twice f x      = f (f x)
```

თქვენი პასუხები შეამოწმეთ Hugs-ით.

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. ფუნქციების განსაზღვრება ენა `Haskell`-ში.

დავალებები

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), აბრუნებს `True`-ს, თუ მეორე წრეწირი მთლიანად თავსდება პირველის შიგნით.
8. ფუნქცია `isRectangular`, რომელიც პარამეტრად ღებულობს სიბრტყეზე სამი წერტილის კოორდინატებს და აბრუნებს `True`-ს, თუ მათ მიერ შედგენილი სამკუთხედი არის მართკუთხა სამკუთხედი.
9. ფუნქცია `isTriangle`, რომელიც განსაზღვრავს, შეიძლება თუ არა მოცემულ `x, y` და `z` სიგრძის მონაკვეთებზე აიგოს სამკუთხედი.
10. ფუნქცია `isSorted`, რომელიც შესასვლელზე ღებულობს სამ რიცხვს და აბრუნებს `True`, თუ ეს რიცხვები დალაგებულია ზრდადობით ან კლებადობით.

თავი 1.4. ფუნქციების განსაზღვრა. რეკურსიული ფუნქციების განსაზღვრა

ფუნქციის განსაზღვრა ამორჩევის ოპერატორის გამოყენებით

ამორჩევის ოპერატორი case არის ენა C/C++-ის switch კონსტრუქციის ანალოგი. მისი გამოყენება რეკომენდებულია ისეთი ფუნქციების განსაზღვრისას, რომლებიც კონკრეტულ არგუმენტებზე კონკრეტულ მნიშვნელობებს ღებულობენ ანუ დისკრეტული ფუნქციების განსაზღვრისას.

დავუშვათ, საჭიროა ფუნქციის განსაზღვრა, რომელიც კონკრეტული მნიშვნელობის სამ არგუმენტზე (0,1 და 2) შედეგად იძლევა კონკრეტულ რიცხვებს (1,5 და 0 შესაბამისად), ხოლო, ყველა სხვა შემთხვევაში, ფუნქციის შედეგია რიცხვი -1. რა თქმა უნდა, ამ ფუნქციის ჩაწერა პირობითი 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 ქვედა ტირე (_) არის ენა

C/C++-ის default 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 (ანალოგიური სისტემა გამოიყენება ფართოდ გავრცელებულ ენა Python-შიც). ეს სისტემა იძლევა საშუალებას არ გამოვიყენოთ სპეციალური სიმბოლოები ოპერატორების გაერთიანებისთვის, მაგალითად, ისეთები, როგორიცაა {,} და : ენა C/C++-ში. სინამდვილეში, ენა Haskell-შიც შეიძლება ამ სიმბოლოების გამოყენება იმავე 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 სტრიქონში პოზიციას, საიდანაც იწყება შემდეგი ბრძანება. შემდგომ, ყოველი ახალი სტრიქონის წინ, 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 მას შემდეგნაირად აღიქვამს:

```
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 პრაქტიკული საქმიანობისას არ გირჩევთ.

კიდევ ერთი შენიშვნა. ვინაიდან Haskell ენის პროგრამისთვის ხარვეზებს აქვთ მნიშვნელობა, ამიტომ ყურადღებაა საჭირო ტაბულაციის სიმბოლოსთანაც. ინტერპრეტატორი თვლის, რომ ტაბულაციის სიმბოლო ტოლია 8 ხარვეზის. თუმცა ზოგიერთი ტექსტური რედაქტორი იძლევა ტაბულაციის სიმბოლოს განსაზღვრის საშუალებას, მაგალითად, Visual Studio- რედაქტორში გაჩუმების პრინციპით ტაბულაცია არის 4 ხარვეზი. ამან შეიძლება გამოიწვიოს შეცდომები, ამიტომაც ჯობს ენა Haskell-ზე დაპროგრამებისას არ გამოიყენოთ ტაბულაციის სიმბოლოები.

ფუნქციის ფრაგმენტული განსაზღვრა

ფუნქცია შეიძლება განისაზღვროს ფრაგმენტულადაც. ეს ნიშნავს, რომ ფუნქციის ერთი ვერსია შეიძლება განისაზღვროს პარამეტრების გარკვეული მნიშვნელობებისთვის, მეორე ვერსია –

სხვა მნიშვნელობისთვის. 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-ის ნებისმიერი არგუმენტისთვის. თუ ამ სტრიქონს საერთოდ არ მივუთითებთ, მივიღებთ შეცდომას არგუმენტისთვის, რომელიც არ არის ტოლი 0-ის, 1-ის ან 2-ის.

ფუნქციის ფრაგმენტული განსაზღვრა ხშირად გამოიყენება Haskell-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)
```

ფუნქციების განსაზღვრა სიების კონსტრუქტორების საშუალებით

სიის კონსტრუქტორი შეიძლება გამოვიყენოთ ფუნქციების განსაზღვრისათვის. განვიხილოთ მაგალითები. სასარგებლო საბიბლიოთეკო zip ფუნქციას ორი არგუმენტი აქვს (ორივე სია),

შედეგად კი იძლევა წყვილების სიას. ამ სიის პირველი წყვილი შეიცავს არგუმენტების პირველ ელემენტებს, მეორე წყვილი - არგუმენტების მეორე ელემენტებს და ა.შ.

```
zip :: [a] → [b] → [(a,b)]
```

მაგალითად,

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

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

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

მაგალითად,

```
Prelude> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

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

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

მაგალითად,

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

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

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

მაგალითად,

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

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

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

მაგალითად,

```
> lowers "Haskell"
6
```

ნიმუშთან შედარება

შესაძლებელია რეკურსიული ფუნქციები განისაზღვროს როგორც მთელ რიცხვებზე, ასევე სიებზეც. სიების 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 გამოყენებულ იქნეს ფუნქციის აღწერისას:

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

მოვიყვანოთ კიდევ ერთი მაგალითი. ფუნქცია, რომელიც არგუმენტად 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 (`pattern matching`). იგი წარმოადგენს ენის ძალზე ძლიერ კონსტრუქციას. ნიმუშები ჩაიწერება ფუნქციის არგუმენტებად და შედარდება ფუნქციაზე გადაცემულ ფაქტობრივ 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 გამოყენებულ იქნეს სიმბოლო `_`. იგი აღნიშნავს ნიმუშს, რომელსაც შეესაბამება ნებისმიერი მნიშვნელობა, თვითონ ეს მნიშვნელობა კი არცერთ ცვლადს არ უკავშირდება.

შემდეგი მაგალითები უჩვენებენ ნიმუშთან შედარების გამოყენების სხვადასხვა ვარიანტს:

ფუნქცია, რომელიც შეკრებს სიის პირველ ორ წევრს:

```
f1 (x:y:xs) = x + y
```

ფუნქციის განსაზღვრება, რომელიც head-ის ანალოგიურია:

```
my_head (x:xs) = x
```

ფუნქციის განსაზღვრება, რომელიც tail-ის ანალოგიურია. ჩვენ ვიყენებთ სიმბოლო _-ს იმიტომ, რომ სიის პირველი ელემენტის მნიშვნელობა არ არის საჭირო:

```
my_tail (_:xs) = xs
```

ფუნქცია, რომელიც იღებს პირველ წევრს სამეულიდან:

```
fst3 (x,_,_) = x
```

ნიმუშთან შედარება შეიძლება გამოვიყენოთ ოპერატორში case:

სიის სიგრძის განსაზღვრის კიდევ ერთი ფუნქცია:

```
my_length s = case s of
                  [] -> 0
                  (_:xs) -> 1 + my_length xs
```

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

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

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

მაგალითად, ფუნქცია `factorial`-ის ზემოთ მოყვანილ მაგალითში (ფუნქციის ფრაგმენტული განსაზღვრისას) 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 გამოყენებულ იქნეს გასაღები სიტყვა `otherwise` (ინგლისურად – წინააღმდეგ შემთხვევაში). მაგალითად, ფუნქცია, რომელიც განსაზღვრავს რიცხვის ნიშანს, ასე გამოიყურება:

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

ასეთი სტილით ფუნქციის განმარტება უფრო თვალსაჩინოა და Haskell-ის პროგრამებში ხშირად გამოიყენება (შესაბამისად, პირობითი ოპერატორი გამოიყენება იშვიათად). თვალსაჩინოებისთვის, განვსაზღვროთ ფუნქცია signum პირობითი ოპერატორების გამოყენებით:

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

საზოგადოდ, otherwise პირობა განსაზღვრულია Prelude ბიბლიოთეკაში otherwise = True ფორმით.

შესაბამისობა შაბლონთან

მრავალი ფუნქცია განსაკუთრებით მკაფიოდ განისაზღვრება მათი არგუმენტების შაბლონების გამოყენებით. ამ შემთხვევაში შაბლონად იგულისხმება კონსტანტური მნიშვნელობა:

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

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

ფუნქციები ხშირად შეიძლება იყოს განსაზღვრული მრავალი სხვადასხვა ხერხით შაბლონებთან შედარებისას. მაგალითად, ორარგუმენტიანი ფუნქცია `&&` (ლოგიკური და):

```
(&&)           :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

ეს უფრო კომპაქტურად განისაზღვრება:

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

მაგრამ შემდეგი განმარტება უფრო ეფექტურია, რადგან იგი არ მიმართავს მეორე არგუმენტის შეფასებას, როცა პირველ არგუმენტს `False` მნიშვნელობა აქვს.

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

ამ განმარტებებისას გამოყენებული სიმბოლო - ქვედა ტირე(`_`) წარმოადგენს ჩასმის შაბლონს ნებისმიერი მნიშვნელობის არგუმენტისთვის.

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

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

შაბლონები კრძალავს ცვლადების განმეორებას. მაგალითად, შემდეგი განმარტება იძლევა შეცდომას:

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

შაბლონები სიების ასაგებად

იმ ფუნქციების განსაზღვრისთვის, რომლებიც აბრუნებენ სიებს, ხშირად გამოიყენება ოპერატორი `:`. როგორც უკვე ვიცით, ყოველი არაცარიელი სია აიგება „cons“(`:`) ოპერატორის გამოყენებით, რომელიც ელემენტს სიის დასაწყისში ამატებს.

სიებზე მოქმედი ფუნქციები შეიძლება განსაზღვრულ იქნეს `x:xs` შაბლონებით. მაგალითად, ფუნქცია, რომელიც ღებულობს რიცხვების სიას და აბრუნებს ამ რიცხვების კვადრატებს, შეიძლება ასე განისაზღვროს:

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

შემდეგ მოცემულია `head` და `tail` ფუნქციების განმარტებები შაბლონების გამოყენებით. მიაქციეთ ყურადღება `_`-ის გამოყენებას, რომელიც შაბლონებში გამოხატავს ფრაზას „ნებისმიერი მნიშვნელობისთვის“:

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

შევნიშნოთ, რომ შაბლონები გამოიყენება მხოლოდ არაცარიელი სიებისთვის: `>head[]` არის შეცდომა. `x:xs` შაბლონები უნდა განთავსდეს ფრჩხილებში, ვინაიდან რეალიზაციას პრიორიტეტი ენიჭება `(:)`-თან შედარებით. მაგალითად, შემდეგი განსაზღვრება მცდარია:

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

შაბლონები მთელ რიცხვებზე

როგორც მათემატიკაში, ფუნქციები მთელ რიცხვებზე განისაზღვრება ე.წ. `n+k` შაბლონებით, სადაც `n` მთელი რიცხვია, ხოლო `k>0` – მთელი მუდმივაა.

```
pred        :: Int -> Int
pred 0      = 0
pred n      = n-1
```

`pred` ასახავს მთელს, რომელიც წინ უძღვის შეტანილს.

შევნიშნოთ, რომ `n+k` შაბლონები შეესაბამება მხოლოდ მთელ რიცხვებს, რომლებიც `>= k`-ზე. მაგალითად, `>pred (-1)` შეცდომაა. ამასთან, `n+k` შაბლონები უნდა გამოვიყენოთ ფრჩხილებში, ვინაიდან ფუნქციის გამოძახება უფრო მაღალპრიორიტეტულია, ვიდრე ოპერაცია მიმატება `(+)`. მაგალითად, შემდეგი განსაზღვრება შეცდომას იძლევა:

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

ფუნქცია შეიძლება აიგოს მისი სახელის მიუთითებლად ლამბდა-გამოსახულების დახმარებით:

```
λx -> x+x
```

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

λ სიმბოლო წარმოადგენს ბერძნულ ასოს და იგი აიკრიფება კლავიატურაზე, როგორც დახრილი ხაზი (\).

Haskell-ში λ სიმბოლოს გამოყენება უსახელო ფუნქციებისთვის დაკავშირებულია λ აღრიცხვასთან – ფუნქციათა თეორიასთან, რომელსაც ეფუძნება ეს ენა.

λ გამოსახულებები შეიძლება გამოვიყენოთ ფორმალური არსის მისაცემად კარირებით განსაზღვრული ფუნქციებისთვის. მაგალითად:

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

შესაძლებელია:

```
add = λx -> (λy -> x+y)
```

λ გამოსახულებები სასარგებლოა აგრეთვე იმ ფუნქციების განსაზღვრისას, რომლებიც გვიბრუნებენ ფუნქციებს, როგორც შედეგებს. მაგალითად:

```
const    :: a -> b -> a
const x _ = x
```

უფრო ბუნებრივად განისაზღვრება ასეთი ჩანაწერით:

```
const  :: a -> (b -> a)
const x = λ_ -> x
```

λ გამოსახულებების გამოყენება შეიძლება ისეთი ფუნქციების დასახელების თავიდან ასაცილებლად, რომელსაც მხოლოდ ერთხელ მიმართავენ. მაგალითად:

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

შეიძლება დავიყვანოთ გამოსახულებამდე:

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

სექციები

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

```
Prelude>  1+2
3
Prelude> (+) 1 2
3
```

ეს შეთანხმება საშუალებას იძლევა ოპერატორის ერთ-ერთი არგუმენტი ჩავრთოთ ფჩხილებს შორის. მაგალითად,

```
Prelude> (1+) 2
3
Prelude>  (+2) 1
3
```

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

სექციები მოსახერხებელია, რადგან, ზოგჯერ, სასარგებლო ფუნქციები მარტივად შეიძლება ავაგოთ სექციების საშუალებით. მაგალითად:

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

შეცდომების შესახებ

ჩვენ 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 სწორად მუშაობს მანამ, სანამ არ შევეცდებით გამოვთვალოთ უარყოფითი რიცხვის ფაქტორიალი. რთული მისახვედრი არაა, რომ ამ დროს რეკურსია უსასრულოდ გრძელდება, რადგან 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 ჩაიწერება:

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

რეკურსიული ფუნქციების განსაზღვრის ძირითადი კონცეფციები

როგორც უკვე დავრწმუნდით, მრავალი ფუნქცია სავსებით ბუნებრივად განისაზღვრება სხვა ფუნქციების გამოყენებით.

მაგალითად, შესაძლებელია განვსაზღვროთ ფუნქცია, რომელიც გვიბრუნებს არაუარყოფითი მთელი რიცხვის ფაქტორიალს საბიბლიოთეკო ფუნქციების გამოყენებით ერთსა და მოცემულ მნიშვნელობას შორის მოთავსებულ რიცხვთა ნამრავლის გამოსათვლელად:

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

Haskell-ში დაშვებულია აგრეთვე ფუნქციათა განსაზღვრა საკუთარი თავის გამოყენებით. ასეთ შემთხვევაში ფუნქციებს რეკურსიულს უწოდებენ. მაგალითად, ამ გზით შეიძლება განისაზღვროს `factorial` ფუნქციაც:

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

პირველი განტოლება გვეუბნება, რომ ნულის ფაქტორიალი ერთია და ამ განტოლებას საბაზო (საყრდენი) შემთხვევა (გამოსახულება) ეწოდება. მეორე განტოლება ამტკიცებს, რომ ნებისმიერი მკაცრად დადებითი მთელი რიცხვის ფაქტორიალი წარმოადგენს ამ რიცხვისა და მისი წინა რიცხვის ფაქტორიალის ნამრავლს. მას რეკურსიული შემთხვევა (გამოსახულება) ეწოდება.

ყურადღება მიაქციეთ იმ გარემოებას, რომ, თუმცა `factorial` ფუნქცია საკუთარი თავის საშუალებით განისაზღვრება, იგი უსასრულო ციკლს არ წარმოადგენს. სახელდობრ, `factorial` ფუნქციის ყოველი გამოყენება ამცირებს მთელრიცხვა არგუმენტს ერთით, ვიდრე იგი საბოლოო ჯამში ნულის ტოლი არ გახდება. აქ რეკურსია წყდება და გამრავლების ოპერაცია სრულდება. ნულის ფაქტორიალი, რომლითაც ერთიანი გვიბრუნდება, სავსებით ადეკვატურია ამ ვითარებაში, რადგან გამრავლებისას ერთიანი იგივეობას უნარჩუნებს გამოსახულებას. სხვანაირად რომ ვთქვათ, `1 * x = x` და `x * 1 = x` ნებისმიერი `x` რიცხვისთვის.

`factorial` ფუნქციის შემთხვევაში საწყისი განსაზღვრება საბიბლიოთეკო ფუნქციების საშუალებით უფრო მარტივია, ვიდრე რეკურსიის გამოყენებით. მაგრამ, როგორც ამას შემდგომ დავინახავთ, მრავალი ფუნქცია მარტივად და ბუნებრივად სწორედ რეკურსიის გამოყენებით განისაზღვრება. მაგალითად, მრა-

ვალი საბიბლიოთეკო ფუნქცია Haskell-ში განისაზღვრება რეკურსიით. გარდა ამისა, ფუნქციათა განსაზღვრა რეკურსიით მათი თვისებების დამტკიცების საშუალებას იძლევა მათემატიკური ინდუქციის მძლავრი მეთოდის გამოყენებით.

მთელ რიცხვებზე რეკურსიის მაგალითად განვიხილოთ ზემოთ გამოყენებული გამრავლების * ოპერატორი. ეფექტურობის მოსაზრებებიდან გამომდინარე, Haskell-ში ეს ოპერატორი გათვალისწინებულია როგორც პრიმიტივი - დაპროგრამების ენაში პროგრამათა შესრულების სიჩქარის გაზრდისათვის ჩაშენებული ოპერატორი. მაგრამ არაუარყოფითი მთელი რიცხვებისათვის იგი შეიძლება იყოს განსაზღვრული ასევე რეკურსიითაც, თავისი ორი არგუმენტიდან ნებისმიერ ერთზე, ვთქვათ მეორეზე:

```
  (*)           ::    Int → Int → Int
m * 0           =     0
m * (n + 1)     =     m + (m * n)
```

მაგალითად:

```
   4 * 3
=          { * ოპერატორის გამოყენება }
   4 + (4 * 2)
=          { * ოპერატორის გამოყენება }
   4 + (4 + (4 * 1))
=          { * ოპერატორის გამოყენება }
   4 + (4 + (4 + (4 * 0)))
=          { * ოპერატორის გამოყენება }
   4 + (4 + (4 + 0))
=          { + ოპერატორის გამოყენება }
   12
```

ამრიგად, აქ რეკურსიული განსაზღვრება * ოპერატორისათვის ფორმალურად გამოხატავს იმ იდეას, რომ გამრავლება დაიყვანება იტერაციულ (მრავალჯერად) შეკრებამდე.

რეკურსია სიებზე

რეკურსია არ შემოიფარგლება ფუნქციებით მთელ რიცხვებზე, იგი შეიძლება ასევე გამოყენებულ იქნეს ფუნქციათა განსაზღვრისათვის სიებზე. მაგალითად, `product` საბიბლიოთეკო ფუნქცია, რომელიც წინა პუნქტში ვიხმარეთ, შემდეგი სახით შეიძლება განისაზღვროს:

```
product             ::   Num a ⇒ [a] → a
product []    =     1
product (n : ns)    =    n * product ns
```

პირველი განტოლება ამტკიცებს, რომ ცარიელი სიის ნამრავლი ერთიანია, რაც სავსებით ადეკვატურია, რადგან ერთიანი გამრავლების ოპერაციაში უნარჩუნებს გამოსახულებას იგივეობას. მეორე განტოლება კი გვეუბნება, რომ ნებისმიერი არაცარიელი სიის ნამრავლი მიიღება პირველი რიცხვისა და რიცხვთა დარჩენილი სიის ნამრავლის ერთმანეთზე გამრავლების ოპერაციით. მაგალითად:

```
   product [2, 3, 4]
=        { product ფუნქციის გამოყენება }
   2 * product [3, 4]
=        { product ფუნქციის გამოყენება }
   2 * (3 * product [4])
=        { product ფუნქციის გამოყენება }
```

```
    2 * (3 * (4 * product []))
=           { product ფუნქციის გამოყენება }
    2 * (3 * (4 * 1))
=           { * ოპერატორის გამოყენება }
    24
```

გავიხსენოთ, რომ რეალურად სიას Haskell-ში აქვს ერთი ელემენტის ლოგიკური სტრუქტურა, რომელიც ამავე დროს cons ოპერატორს იყენებს. მაშასადამე, [2, 3, 4] ჩანაწერი მხოლოდ აბრევიატურაა 2 : (3 : (4 : [])) გამოსახულებისათვის და მეტი არაფერი. განვიხილოთ სიებზე რეკურსიის კიდევ ერთი მარტივი მაგალითი, რისთვისაც მივმართოთ length საბიბლიოთეკო ფუნქციას, რომელიც შეიძლება განისაზღვროს რეკურსიის ამავე შაბლონის გამოყენებით product ფუნქციის მსგავსად:

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

მაშასადამე, ცარიელი სიის სიგრძე ნულია, ხოლო ნებისმიერი არაცარიელი სიის სიგრძე მისი კუდის სიგრძის მომდევნო მნიშვნელობაა. ყურადღება მიაქციეთ (_) ჩასმის შაბლონის გამოყენებას რეკურსიულ გამოსახულებაში. ეს შაბლონი ასახავს იმ ფაქტს, რომ სიის სიგრძე დამოკიდებული არ არის მისი ელემენტების მნიშვნელობაზე.

ახლა განვიხილოთ საბიბლიოთეკო ფუნქცია, რომელიც სიის შებრუნებას ახორციელებს. რეკურსიის საშუალებით ეს ფუნქცია შეიძლება შემდეგნაირად განვსაზღვროთ:

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

მაშასადამე, ცარიელი სიის შებრუნება კვლავ ცარიელი სიაა, ხოლო ნებისმიერი არაცარიელი სიის შებრუნება ხორციელდება მისი კუდის შებრუნებით მიღებული სიის გაერთიანებით ერთელემენტიან სიასთან, რომელიც საწყისი სიის თავს წარმოადგენს. მაგალითად:

```
   reverse [1, 2, 3]
=         { reverse ფუნქციის გამოყენება }
   reverse [2, 3] ++ [1]
=         { reverse ფუნქციის გამოყენება }
   (reverse [3] ++ [2])++ [1]
=         { reverse ფუნქციის გამოყენება }
   ((reverse []++ [3])++ [2])++ [1]
=         { reverse ფუნქციის გამოყენება }
   (([] ++ [3])++ [2]) ++ [1]
=         { ++ ოპერატორის გამოყენება }
   [3, 2, 1]
```

თავის მხრივ, დამატების ++ ოპერატორი, რომელიც *reverse* ფუნქციის ზემოთ მოცემულ განსაზღვრებაში გამოიყენება, თავად შეიძლება იქნეს აღწერილი რეკურსიით თავის პირველ არგუმენტზე:

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

მაგალითად:

```
   [1, 2, 3] ++ [4, 5]
=          { ++ ოპერატორის გამოყენება }
   1 : ([2, 3] ++ [4, 5])
=          { ++ ოპერატორის გამოყენება }
   1 : (2 : ([3]++ [4, 5]))
=          { ++ ოპერატორის გამოყენება }
   1 : (2 : (3 : ([] ++ [4, 5])))
=          { ++ ოპერატორის გამოყენება }
   1 : (2 : (3 : [4, 5]))
=          { ჩაწერა სიის სახით }
   [1, 2, 3, 4, 5]
```

ამრიგად, რეკურსიული განსაზღვრება ++ ოპერატორისათვის ფორმალურად ასახავს იმ იდეას, რომ ორი სიის გადაბმა შეიძლება პირველი სიიდან ელემენტების ასლის გადაღებით, ვიდრე ეს სია არ ამოიწურება, რის შემდეგ მეორე სია დაემატება ამ ასლს ბოლოში.

განვიხილოთ ორი მაგალითი, რომელიც ეხება რეკურსიას დახარისხებულ სიებზე. უპირველეს ყოვლისა, განვიხილოთ ფუნქცია, რომელიც დახარისხებულ სიაში ახორციელებს ნებისმიერი მოწესრიგებული ტიპის ახალი ელემენტის ჩასმას კიდევ ერთი ახალი დახარისხებული სიის მისაღებად. ეს ფუნქცია შემდეგი სახით შეიძლება განისაზღვროს:

```
insert                      ::   Ord a ⇒ a → [a] → [a]
insert x []                 =    [x]
insert x (y : ys) | x ≤ y       =  x : y : ys
                  | otherwise   =  y : insert x ys
```

მაშასადამე, ახალი ელემენტის ჩასმა ცარიელ სიაში იძლევა ერთელემენტიან სიას, მაშინ, როცა არაცარიელი სიისათვის შედეგი დამოკიდებულია ახალი `x` ელემენტისა და სიის `y` თავის ურთიერთგანლაგებაზე. სახელდობრ, თუ $x \leq y$, მაშინ ახალი `x` ელემენტი მხოლოდ თავსდება სიის დასაწყისში და მეტი არაფერი, წინააღმდეგ შემთხვევაში `y` თავი საბოლოო სიის პირველი ელემენტი ხდება და შემდეგ იწყებენ ახალი ელემენტის ჩასმას მოცემული სიის კუდში. მაგალითად:

```
   insert 3 [1, 2, 4, 5]
=        { insert ფუნქციის გამოყენება }
   1 : insert 3 [2, 4, 5]
=        { insert ფუნქციის გამოყენება }
   1 : 2 : insert 3 [4, 5]
=        { insert ფუნქციის გამოყენება }
   1 : 2 : 3 : [4, 5]
=        { ჩაწერა სიის სახით }
   [1, 2, 3, 4, 5]
```

`insert` ფუნქციის გამოყენებით შესაძლებელია ახალი ფუნქციის განსაზღვრა, რომელიც ახორციელებს დახარისხებას ჩასმით (`insertion sort`). ამ განსაზღვრებაში უნდა იქნეს გათვალისწინებული, რომ ცარიელი სია უკვე დახარისხებულია, ხოლო ნებისმიერი არაცარიელი სიის დახარისხება ხდება მისი თავის ჩასმით კუდის დახარისხების შედეგად მიღებულ სიაში:

```
isort        ::   Ord a ⇒ [a] → [a]
isort []          =     []
isort (x : xs)    =     insert x (isort xs)
```

მაგალითად:

```
   isort [3, 2, 1, 4]
=        { isort ფუნქციის გამოყენება }
   insert 3 (insert 2 (insert 1 (insert 4 [])))
=        { insert ფუნქციის გამოყენება }
   insert 3 (insert 2 (insert 1 [4]))
=        { insert ფუნქციის გამოყენება }
   insert 3 (insert 2 [1, 4])
=        { insert ფუნქციის გამოყენება }
   insert 3 [1, 2, 4]
=        { insert ფუნქციის გამოყენება }
   [1, 2, 3, 4]
```

მრავალარგუმენტიანი ფუნქციები

მრავალარგუმენტიანი ფუნქციები შეიძლება განისაზღვროს რეკურსიის გამოყენებით ორ ან მეტ არგუმენტზე. მაგალითად, საბიბლიოთეკო `zip` ფუნქცია, რომელიც იღებს შესასვლელზე ორ სიას და გვიბრუნებს წყვილების შექმნილ სიას, შემდეგი სახით განისაზღვრება:

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

მაგალითად:

```
   zip ['a', 'b', 'c'] [1, 2, 3, 4]
=          { zip ფუნქციის გამოყენება }
   ('a', 1) : zip ['b', 'c'] [2, 3, 4]
=          { zip ფუნქციის გამოყენება }
   ('a', 1) : ('b', 2) : zip ['c'] [3, 4]
=          { zip ფუნქციის გამოყენება }
   ('a', 1) : ('b', 2) : ('c', 3) : zip [] [4]
=          { zip ფუნქციის გამოყენება }
   ('a', 1) : ('b', 2) : ('c', 3) : []
=          { ჩაწერა სიის სახით }
[('a', 1), ('b', 2), ('c', 3)]
```

ყურადღება მიაქციეთ, რომ `zip` ფუნქციის განსაზღვრებაში საჭიროა ორი საბაზო გამოსახულება, ვინაიდან არგუმენტთა ორი სიიდან ნებისმიერი შეიძლება აღმოჩნდეს ცარიელი.

განვიხილოთ რამდენიმე არგუმენტზე რეკურსიის კიდევ ერთი მაგალითი. სახელდობრ, საბიბლიოთეკო `drop` ფუნქცია, რომელიც სიაში ანადგურებს ელემენტების მოცემულ რაოდენობას ამ სიის დასაწყისიდან, შემდეგი სახით შეიძლება განისაზღვროს:

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

აქ კვლავ ორი საბაზო გამოსახულებაა საჭირო: ერთი ნულოვანი რაოდენობის ელემენტთა გასანადგურებლად სიაში, ხოლო მეორე – ცარიელ სიაში ერთი ან რამდენიმე ელემენტის განადგურების მცდელობისას.

ფუნქციები ასევე შეიძლება განისაზღვროს მრავალჯერადი რეკურსიით, რომელშიც ფუნქცია თავის საკუთარ განსაზღვრებას არაერთხელ იყენებს. მაგალითად, გავიხსენოთ ფიბონაჩის რიცხვთა `0, 1, 1, 2, 3, 5, 8, 13, ...` მიმდევრობა, რომელშიც პირველი და მეორე რიცხვია `0` და `1` შესაბამისად, ხოლო ყოველი მომდევნო რიცხვი წინა ორის ჯამს წარმოადგენს. `Haskell`-ში ფუნქცია, რომელიც ანგარიშობს ფიბონაჩის მე-`n` რიცხვს ნებისმიერი მთელი $n \geq 0$ რიცხვისათვის, შემდეგნაირად შეიძლება განისაზღვროს ორჯერადი რეკურსიის გამოყენებით:

```
fibonacci       ::  Int → Int
fibonacci 0     =   0
fibonacci1      =   1
fibonacci (n + 2)  = fibonacci n + fibonacci (n + 1)
```

განვიხილოთ სიის სწრაფი დახარისხების `Quicksort` სახელით ცნობილი მეთოდის რეალიზაცია. `Haskell` ენაში შესაბამისი ფუნქციის განსაზღვრა შემდეგი სახით შეიძლება:

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

მაშასადამე, ცარიელი სია უკვე დახარისხებულია, ხოლო ნებისმიერი არაცარიელი სია შეიძლება დახარისხდეს, თუ მის თავს მოვათავსებთ ორ სიას შორის. პირველი სიაა თავდაპირველი სიის თავზე ნაკლები კუდის ელემენტებისგან მიღებული დახარისხებული სია, ხოლო მეორე - მასზე მეტი კუდის ელემენტებისგან მიღებული დახარისხებული სია.

ურთიერთრეკურსია

ფუნქციების განსაზღვრა ასევე შეიძლება ურთიერთრეკურსიით (ინგლ.*mutual recursion*), როცა ორი ან მეტი ფუნქცია მთლიანად განსაზღვრულია ერთმანეთის საშუალებით. მაგალითად, განვიხილოთ `even` (ლუწი რიცხვი) და `odd` (კენტი რიცხვი) საბიბლიოთეკო ფუნქციები. ეფექტურობის გაზრდის მიზნით ეს ფუნქციები, ჩვეულებრივ, განისაზღვრება ორზე გაყოფის შედეგად მიღებული ნაშთის გამოყენებით. მაგრამ არაუარყოფითი მთელი რიცხვებისათვის მათი განსაზღვრა ურთიერთრეკურსიის საშუალებითაც შეიძლება:

```
even          ::      Int → Bool
even 0                =       True
even (n + 1)  =       odd n
odd           ::      Int → Bool
odd 0         =       False
odd (n + 1)   =       even n
```

მაშასადამე, ნული – მთელი რიცხვია, ხოლო ნებისმიერი მკაცრად დადებითი რიცხვი ლუწია, თუ მისი წინამავალი რიცხვი კენტია, და ნებისმიერი მკაცრად დადებითი რიცხვი კენტია, თუ მისი წინამავალი რიცხვი ლუწია. მაგალითად:

```
   even 4
=          { even ფუნქციის გამოყენება }
   odd 3
=          { odd ფუნქციის გამოყენება }
   even 2
=          { even ფუნქციის გამოყენება }
   odd 1
=          { odd ფუნქციის გამოყენება }
   even 0
=          { even ფუნქციის გამოყენება }
   True
```

ამის მსგავსად, ფუნქციები, რომლებიც ირჩევს ელემენტებს სიიდან ყველა ლუწ და კენტ პოზიციაში (ამ პოზიციის ათვლისას ნულიდან), შეიძლება შემდეგი სახით განისაზღვროს შესაბამისად:

```
evens          ::       [a] → [a]
evens []               =      []
evens (x : xs)         =      x : odds xs
odds           ::       [a] → [a]
odds []                =      []
odds ( : xs) =         evens xs
```

მაგალითად:

```
   evens "abcde"
=          { evens ფუნქციის გამოყენება }
   'a' : odds "bcde"
=          { odds ფუნქციის გამოყენება }
   'a' : evens "cde"
=          { evens ფუნქციის გამოყენება }
```

```
   'a' : 'c' : odds "de"
=          { odds ფუნქციის გამოყენება }
   'a' : 'c' : evens "e"
=          { evens ფუნქციის გამოყენება }
   'a' : 'c' : 'e' : odds []
=          { odds ფუნქციის გამოყენება }
   'a' : 'c' : 'e' : []
=          { ჩაწერა სიის სახით }
   "ace"
```

გავიხსენოთ, რომ სტრიქონები Haskell-ში რეალურად აგებულია, როგორც სიმბოლოთა სიები. ამრიგად, `"abcde"` ჩანაწერი მხოლოდ აბრევიატურაა `['a', 'b', 'c', 'd', 'e']` გამოსახულებისათვის და მეტი არაფერი.

რეკომენდაციები რეკურსიის საკითხებზე

რეკურსიული ფუნქციების განსაზღვრა გამოიყურება მარტივად, როცა ამას ვინმე სხვა აკეთებს; შეიძლება განუხორციელებელი მოგეჩვენოთ, თუ ამის გაკეთებას პირველად დამოუკიდებლად მოინდომებთ, მაგრამ მარტივი და ბუნებრივი ხდება პრაქტიკის მიღების შემდეგ. შემდგომ, სამი კონკრეტული მაგალითის საფუძველზე, განხილულია რეკომენდაციები ხუთეტაპიანი პროცესის სახით, რომელიც არსებობს რეკურსიულ ფუნქციათა აღწერისთვის.

მაგალითი I - *product*

პირველი მარტივი მაგალითის სახით აღვწეროთ, როგორ ხდება ეტაპობრივად ამ თავში ადრე მოცემული `product` სა-

ბიბლიოთეკო ფუნქციის განსაზღვრა, რომელიც ანგარიშობს რიცხვთა სიის ნამრავლს.

ეტაპი 1: ტიპის აღწერა

დაფიქრება ტიპების თაობაზე შეიძლება ძალიან სასარგებლო აღმოჩნდეს ფუნქციათა განსაზღვრისას. ასე რომ, ფუნქციის ტიპის განსაზღვრა საკუთრივ ფუნქციის განსაზღვრის დაწყებამდე კარგი და მისასალმებელი ჩვევაა. განსახილველი მაგალითის შემთხვევაში ვიწყებთ ფუნქციის ფორმით:

```
product       ::       [Int] → Int
```

ამ სტრუქტურიდან ჩანს, რომ `product` ფუნქცია იღებს შესასვლელზე მთელი რიცხვების სიას და გამოაქვს გამოსასვლელზე მთელრიცხვა მნიშვნელობა. ამ მაგალითის მსგავსად, სადაც საუბარია `Int` ტიპზე, ხშირად სასარგებლოა განხილვის დაწყება სწორედ ასეთი მარტივი ტიპით, რომელიც მოგვიანებით შეიძლება დაზუსტდეს ან განზოგადდეს აუცილებლობის შემთხვევაში.

ეტაპი 2: შემთხვევათა ჩამოთვლა

არგუმენტთა ტიპების უმრავლესობისათვის არსებობს რიგი სტანდარტული შემთხვევა და მათი განხილვა აუცილებელია. სახელდობრ, სიებისათვის სტანდარტული შემთხვევებია ცარიელი სია და არაცარიელი სიები. ამიტომ შეგვიძლია ჩავწეროთ შემდეგი სქემატური განსაზღვრება შაბლონთან შედარების გამოყენებით:

```
product []    =
product (n : ns)      =
```

არაუარყოფითი მთელი რიცხვებისათვის სტანდარტული შემთხვევებია `0` და *n* + 1, ლოგიკური მნიშვნელობებისათვის ასეთია `False` და `True` და ასე შემდეგ. მოგვიანებით, ტიპების მსგავსად, შეიძლება დაგვჭირდეს შემთხვევათა დაზუსტებაც, მაგრამ მათი განხილვა სასარგებლოა სტანდარტული შემთხვევებით დაიწყოს.

ეტაპი 3: მარტივ შემთხვევათა განსაზღვრა

მთელი რიცხვების ნამრავლი, როცა ამ რიცხვების რაოდენობა ნულს შეადგენს, იძლევა ერთს, ვინაიდან ერთიანი არ არღვევს გამრავლების ოპერაციაში გამოსახულების უცვლელობას. ამიტომ ცარიელი სიის შემთხვევაში ბუნებრივია განისაზღვროს, რომ:

```
product []    =       1
product (n : ns)    =
```

ამ მაგალითის მსგავსად, მარტივი შემთხვევა ხშირად იძენს ძირითადი, საბაზო შემთხვევის მნიშვნელობას.

ეტაპი 4: სხვა შემთხვევათა განსაზღვრა

როგორ შეიძლება გამოვიანგარიშოთ მთელი რიცხვების არაცარიელი სიის ნამრავლი? ამ ეტაპზე სასარგებლო იქნება ჯერ განვიხილოთ შემადგენელი ნაწილები, რომლებიც შეიძლება გამოვიყენოთ. ასეთია, მაგალითად, საკუთრივ ფუნქცია `(product)`, არგუმენტები `(n` და `ns)` და შესაბამისი ტიპის `(+, -, *` და ა.შ.) საბიბლიოთეკო ფუნქციები. ამ შემთხვევაში ჩვენ მხოლოდ ვამრავლებთ პირველ მთელ რიცხვს და მთელი რიცხვების დარჩენილი სიის ნამრავლს:

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

ამ მაგალითის მსგავსად, სხვა შემთხვევები ხშირად იძენს რეკურსიულ ხასიათს.

ეტაპი 5: განზოგადება და გამარტივება

მას შემდეგ, რაც ფუნქცია განისაზღვრება ზემოთ აღწერილი პროცესის გამოყენებით, ხშირად ნათელი ხდება, რომ იგი შეიძლება განზოგადდეს ან გამარტივდეს. მაგალითად, `product` ფუნქცია დამოკიდებული არ არის იმაზე, არგუმენტი მთელი რიცხვია თუ ათწილადი. ამიტომ მისი ტიპი შეიძლება განზოგადდეს. სავსებით დასაშვებია მთელი რიცხვებიდან ნებისმიერ რიცხვით ტიპზე გადასვლა:

```
product    ::    Num a ⇒ [a] → a
```

რაც შეეხება გამარტივებას, შემდგომ ვნახავთ, რომ `product` ფუნქციაში გამოყენებული რეკურსიის შაბლონი ინკაფსულირებულია (ე.ი. კიდევ რაღაცას შეიცავს საკუთარ თავში) `foldr` დასახელების საბიბლიოთეკო ფუნქციით, რომელსაც `product` ფუნქცია იყენებს. ამიტომ `product` ფუნქცია შეიძლება ხელახლა განისაზღვროს ერთი განტოლებით:

```
product       =       foldr (*) 1
```

ამრიგად, საბოლოო განსაზღვრებას `product` ფუნქციისთვის შემდეგი სახე აქვს:

```
product    ::    Num a ⇒ [a] → a
product    =     foldr (*) 1
```

ეს არის `product` ფუნქციის ზუსტი განმარტება. `prelude` ბიბლიოთეკაში ეს ფუნქცია `foldr` ფუნქციის ნაცვლად `foldl` ფუნქციას იყენებს. თუ რა განსხვავებაა ამ ორ ფუნქციას შორის, შემდგომ განვიხილავთ.

მაგალითი II - *drop*

ახლა უფრო არსებით მაგალითზე ვაჩვენოთ, როგორ შეიძლება ხუთეტაპიანი პროცესის გამოყენებით ავაგოთ საბიბლიოთეკო `drop` ფუნქციისათვის ადრე მოცემული განსაზღვრება. როგორც ვიცით, ეს ფუნქცია ანადგურებს სიაში ელემენტების მოცემულ რაოდენობას (ამ სიის დასაწყისიდან).

ეტაპი 1: ტიპის აღწერა

დავიწყოთ ტიპით, რომელიც გვეუბნება, რომ `drop` ფუნქცია იღებს შესასვლელზე მთელ რიცხვსა და გარკვეული `a` ტიპის მნიშვნელობათა სიას, ხოლო გამოსასვლელზე ახორციელებსა იმავე ტიპის სიდიდეთა ახალი სიის ფორმირებას:

```
drop    ::    Int → [a] → [a]
```

ყურადღება მიაქციეთ, რომ ამ ტიპის განსაზღვრებას რამდენიმე თავისებურება ახასიათებს, სახელდობრ: 1. პირველ არგუმენტად გამოყენებულია მთელი რიცხვი და არა უფრო ზოგადი რიცხვითი ტიპი, რაც სიმარტივის უზრუნველსაყოფად კეთდება; 2. ნაცვლად იმისა, რომ ფუნქცია შესასვლელზე იღებდეს თავის ორ არგუმენტს წყვილის სახით, მეტი მოქნილობისათვის შექმნილია კარინგის გამოყენების შესაძლებლობა; 3. მთელრიცხვა არგუმენტი განთავსებულია მეორე არ-

გუმენტამდე, რომელსაც ელემენტთა სიის სახე აქვს, რაც წაკითხვის მოხერხებულობას ემსახურება. მაგალითად: `drop n xs` გამოსახულება შეიძლება ასე იკითხებოდეს - „`n` ელემენტის ამოგდება `xs` სიიდან“; `4`. იქმნება ფუნქცია, რომელიც პოლიმორფულია ელემენტთა სიის ტიპით (გამოყენების უნივერსალობის უზრუნველსაყოფად).

ეტაპი 2: შემთხვევათა ჩამოთვლა

ვინაიდან მთელრიცხვა არგუმენტისათვის ორი (`0` და `n + 1`) სტანდარტული შემთხვევა არსებობს, ხოლო არგუმენტების სიისათვის ასევე ორ (`[]` და `x : xs`) განსაკუთრებულ მნიშვნელობასთან გვაქვს საქმე, ქვემოთ მოცემული სქემატური ჩანაწერი განსაზღვრებისათვის საერთო ჯამში ოთხ გამოსახულებას მოითხოვს:

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

ეტაპი 3: მარტივ შემთხვევათა განსაზღვრა

ლოგიკურად, ნებისმიერი სიის დასაწყისიდან მისი ნული ელემენტის ამოგდება იმავე სიას იძლევა. ასე რომ, პირველი ორი შემთხვევის განსაზღვრა უშუალოდ ხდება:

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

ცარიელი სიიდან ერთი ან რამდენიმე ელემენტის ამოგდების მცდელობა დაუშვებელია. ამიტომ მესამე შემთხვევა უნდა გამოვტოვოთ. მაგრამ ეს გამოიწვევს შეცდომის გაჩენას, თუ ასეთი სიტუაცია წარმოიქმნება. ამ დროს პრაქტიკაში მიიღება გადაწყვეტილება შეცდომის გაჩენის ასაცილებლად, რისთვისაც ცარიელი სიის დაბრუნება ხდება:

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

ეტაპი 4: სხვა შემთხვევათა განსაზღვრა

როგორ ამოვაგდოთ ერთი ან რამდენიმე ელემენტი არაცარიელი სიიდან? პასუხი: ერთით ნაკლები რაოდენობის ელემენტის მარტივი ამოღებით სიის კუდიდან:

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

ეტაპი 5: განზოგადება და გამარტივება

`drop` ფუნქცია დამოკიდებული არ არის იმ მთელი რიცხვის ზუსტ სახეზე, რომელსაც იგი პირველ არგუმენტად იღებს. ამიტომ ხსენებული რიცხვის ტიპი შეიძლება განზოგადდეს და ნებისმიერ `Integral` მთელრიცხვა ტიპად გამოცხადდეს, რომლის სტანდარტულ ეგზემპლარებს `Int` და `Integer` I ტიპები წარმოადგენს:

```
drop    ::   Integral b ⇒ b → [a] → [a]
```

მაგრამ ეფექტურობის მოსაზრებებით ასეთი განზოგადება ფაქტობრივად არ ხდება სტანდარტულ `prelude` ფაილში. გამარტივების თვალსაზრისით კი პირველი ორი განტოლება `drop` ფუნქციისათვის შეიძლება ერთ განტოლებად ჩაიწეროს, რომელიც გვეუბნება, რომ ნებისმიერი სიიდან ნულოვანი რაოდენობის ელემენტთა ამოგდება იმავე სიას იძლევა:

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

გარდა ამისა, `x` ცვლადი უკანასკნელ განტოლებაში შეიძლება შევცვალოთ ჩასმის შაბლონით, ვინაიდან ეს ცვლადი განტოლების ტანში არ გამოიყენება:

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

ამის მსგავსად, ჩნდება მოსაზრება, რომ მეორე განტოლებაში `n` სიდიდე უნდა იყოს ჩანაცვლებული `( _ )` ჩანაწერით, მაგრამ ეს დაუშვებელ და მცდარ სახეს აძლევს განსაზღვრებას, ვინაიდან `n + k` ფორმის შაბლონები მოითხოვს, რომ `n` სიდიდე წარმოადგენდეს ცვლადს. ამ შეზღუდვის თავიდან აცილება შესაძლებელი იქნება, თუ მეორე განტოლებაში მთლიანად ჩავანაცვლებთ `n + 1` შაბლონს `( _ )` ჩანაწერით. მაგრამ ეს ფუნქციის ქცევასაც შეცვლის. მაგალითად, `drop`

`(-1) []` გამოსახულების გამოთვლა ამ დროს ცარიელი სიის შემთხვევაშიც მოხდება, ახლა კი ეს შეცდომას იწვევს, ვინაიდან `( _ )` ჩანაწერთან შესაბამისობაში ნებისმიერი მთელი რიცხვის მოყვანა შეიძლება, ხოლო `(n + 1)` ჩანაწერს მხოლოდ ისეთი მთელი რიცხვები შეესაბამება, რომლებიც ერთზე ნაკლები არ არის `(≥ 1)`.

დასასრულ, საბოლოო განსაზღვრება `drop` ფუნქციისათვის სწორედ იმ სახეს იღებს, რომელიც მას `prelude` სტანდარტულ ფაილში აქვს:

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

მაგალითი III - `init`

ავაგოთ განსაზღვრება `init` საბიბლიოთეკო ფუნქციისათვის ხუთეტაპიანი პროცესის გამოყენებით. ეს ფუნქცია არაცარიელი სიის უკანასკნელ ელემენტს ანადგურებს.

ეტაპი 1: ტიპის აღწერა

დავიწყოთ ტიპის აღწერით, რომელიც გვეუბნება, რომ `init` ფუნქცია იღებს შესასვლელზე გარკვეული ტიპის მნიშვნელობათა სიას და აგებს ასეთივე მნიშვნელობების სხვა სიას:

```
init    ::    [a] → [a]
```

ეტაპი 2: შემთხვევათა ჩამოთვლა

ვინაიდან ცარიელი სია არ არის დასაშვები არგუმენტი `init` ფუნქციისათვის, განსაზღვრების ქვემოთ ნაჩვენები სქემატური ჩანაწერი შაბლონთან შედარების გამოყენებით მხოლოდ ერთი შემთხვევის მითითებას მოითხოვს:

```
init (x : xs)     =
```

ეტაპი 3: მარტივ შემთხვევათა განსაზღვრა

მაშინ, როცა ორ წინა მაგალითში ეს ეტაპი სავსებით ცხადი იყო, `init` ფუნქციის შემთხვევაში ოდნავ მეტი დაფიქრება დაგვჭირდება. ლოგიკურად, უკანასკნელი ელემენტის ამოღება ერთელემენტიანი სიიდან ცარიელ სიას იძლევა. ამიტომ შეგვიძლია მცველის შემოტანა ამ მარტივი შემთხვევისათვის:

```
init (x : xs) | null xs     =     []
              | otherwise   =
```

გავიხსენოთ, რომ საბიბლიოთეკო `null` ფუნქცია ადგენს ცარიელია სია, თუ არა.

ეტაპი 4: სხვა შემთხვევათა განსაზღვრა

როგორ შეიძლება უკანასკნელი ელემენტის ამოღება სიიდან, რომელშიც, სულ ცოტა, ორი ელემენტი მაინც არის - თავის მარტივი შენარჩუნებით და უკანასკნელი ელემენტის ამოღებით კუდიდან:

```
init (x : xs) | null xs     =    []
              | otherwise   =    x : init xs
```

ეტაპი 5: განზოგადება და გამარტივება

`init` ფუნქციისათვის ტიპი უკვე იმდენად ზოგადია, რამდენადაც ეს შესაძლებელია, თუმცა თავად განსაზღვრება შეიძლება გამარტივდეს შემდეგნაირად:

```
init          ::    [a] → [a]
init [ _ ]          =     []
init (x : xs)       =     x : init xs
```

ამ შემთხვევაშიც სწორედ ასეთია `init` ფუნქციის განსაზღვრება `prelude` სტანდარტულ ფაილში.

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 t_n ტოლია ერთნაირი მონეტების რაოდენობისა, რომლებიდანაც შეიძლება შედგეს ტოლგვერდა სამკუთხედი, რომლის თითოეულ გვერდზეც დაიდება n მონეტა. ცხადია, რომ $t_1 = 1$ და $t_n = n + t_{n-1}$).

8) პირამიდალური რიცხვების სია.
განმარტება: n-ური პირამიდალური რიცხვი p_n ტოლია ერთნაირი ბურთების რაოდენობისა, რომლებიდანაც შეიძლება აიგოს სწორი პირამიდა სამკუთხედის ძირით, რომლის თითოეულ გვერდზეც დაიდება n ბურთი. ცხადია, რომ $p_1 = 1$ და $p_n = n*(n+1)*(2*n+1)/6$.

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 აჯამვის 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 $2^n=2^{2k+1}=2 \cdot(2^k)^2$. ფუნქციამ twopow n არ უნდა გამოიყენოს ოპერატორი ^ ან სხვა ფუნქცია სტანდარტული ბიბლიოთეკიდან, რომელიც ითვლის ხარისხს. ფუნქციის რეკურსიული გამოძახებები უნდა იყოს log n-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 სიმბოლოთი. მაგალითად, `substitute 'e' 'i' "eigenvalue"` აბრუნებს `"iiginvalui"`.

სავარჯიშოები

1. განსაზღვრეთ ხარისხში აყვანის `↑` ოპერატორი არაუარყოფითი მთელი რიცხვებისათვის რეკურსიის იმავე მოდელით, რომელიც გამოყენებული იყო გამრავლების `*` ოპერატორისათვის, და უჩვენეთ, თუ როგორ ფასდება `2 ↑ 3` გამოსახულება თქვენი განსაზღვრების გამოყენებით.

2. ამ თავში მოცემულ განსაზღვრებათა გამოყენებით აჩვენეთ `length [1, 2, 3]`, `drop 3 [1, 2, 3, 4, 5]` და `init [1, 2, 3]` გამოსახულებათა მნიშვნელობების გამოთვლა.

3. განსაზღვრეთ შემდეგი საბიბლიოთეკო ფუნქციები რეკურსიის გამოყენებით, როგორიც არ უნდა იყოს მათი აღწერები სტანდარტულ `prelude` ფაილში:

- ასკვნის, თუ აქვს True მნიშვნელობა სიის ყველა ლოგიკურ სიდიდეს:

```
and  ::  [Bool] → Bool
```

- ახორციელებს სიების ერთ სიად გაერთიანებას:

```
concat  ::  [[a]] → [a]
```

- ქმნის n ერთნაირი ელემენტის შემცველ სიას:

```
replicate  ::  Int → a → [a]
```

- ირჩევს სიის n -ურ ელემენტს:

```
(!!)  ::  [a] → Int → a
```

- ასკვნის, თუ არის მოცემული მნიშვნელობა სიის ელემენტი:

```
elem  ::  Eq a ⇒ a → [a] → Bool
```

4. განსაზღვრეთ `merge :: Ord a ⇒ [a] → [a] → [a]` რეკურსიული ფუნქცია, რომელიც ახორციელებს ორი დახარისხებული სიის გაერთიანებას ერთი დახარისხებული სიის მისაღებად (მეთოდი ცნობილია როგორც დახარისხება შერწყმით).

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

შენიშვნა: განსაზღვრებაში დაუშვებელია სხვა ფუნქციათა გამოყენება დახარისხებული სიებისათვის, როგორიცაა, ვთქვათ, `insert` ან `isort` ფუნქციები; განსაზღვრება უნდა იყენებდეს ცხად რეკურსიას.

5. `merge` ფუნქციის საშუალებით განსაზღვრეთ `msort :: Ord a ⇒ [a] → [a]` რეკურსიული ფუნქცია, რომელიც `merge sort` ფუნქციის რეალიზებას ახდენს. აქ ცარიელი სია და ერთელემენტიანი სიები უკვე დახარისხებულია, ხოლო ნებისმიერი სხვა სიის დახარისხება ხდება ორი სიის გაერთიანებით, რომლებიც მიღებულია საწყისი სიის ორივე ნახევრის ცალ-ცალკე დახარისხების შედეგად. რჩევა: თავდაპირველად განსაზღვრეთ `halve :: [a] → [([a], [a])]`, რომელიც ყოფს სიას ორ ნახევრად. ამ ნაწილების სიგრძეები შეიძლება განსხვავდებოდეს ერთმანეთისაგან მაქსიმუმ ერთი ერთეულით. ამას ახორციელებს საბიბლიოთეკო `splitAt` ფუნქცია, რომლის ტიპია `Int → [a] → ([a], [a])`.

6. ხუთი ეტაპის გამოყენებით განსაზღვრეთ შემდეგი სამი საბიბლიოთეკო ფუნქცია: `sum` – ანგარიშობს რიცხვთა სიის ჯამს; `take` – ამოაქვს სიიდან მისი ელემენტების მოცემული რაოდენობა (ელემენტების ამოღება დასაწყისიდან ხდება); `last` – ირჩევს არაცარიელი სიის უკანასკნელ ელემენტს.

თავი 1.5. ტიპები. რეკურსიული ტიპები

ტიპების კლასები

ტიპი წარმოადგენს დაკავშირებული მნიშვნელობების ნაკრებს. ტიპის კლასი არის მეთოდებად წოდებული გარკვეული გადატვირთული ოპერაციების მხარდამჭერი ტიპების კოლექცია. Haskell-ს ტიპთა მრავალი კლასი აქვს, მათ შორის:

Num	– რიცხვითი ტიპები
Eq	– ტოლობის ტიპები
Ord	– მოწესრიგებული ტიპები

მაგალითად,

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

Eq - ტოლობის ტიპები

შეიცავს ტიპებს, რომელთა მნიშვნელობების შედარება შეიძლება ორი მეთოდის გამოყენებით:

```
(==) ::      a ->a ->Bool
(/=) ::      a ->a ->Bool
```

ყველა ძირითადი ტიპი: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` არის `Eq` კლასის ეგზემპლარი, როგორც სიისა და კორტეჟის ტიპი.

ფუნქციათა ტიპი არ არის `Eq` კლასის ეგზემპლარი .

მაგალითები:

```
Prelude> False == False
True
Prelude > 'a' == 'b'
False
Prelude > "abc" == "abc"
True
Prelude > [1, 2] == [1, 2, 3]
False
> ('a', False) == ('a', False)
True
```

`Ord` - მოწესრიგებული ტიპები

შეიცავს ტიპებს, რომლებიც `Eq` ტოლობის კლასის ეგზემპლარებია და მათი მნიშვნელობები მოწესრიგებულია 6 მეთოდით:

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

ყველა ძირითადი ტიპი: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` არის `Ord` კლასის ეგზემპლარი, როგორც სიისა და კორტეჟის ტიპი.

```
Prelude > False < True
True
Prelude > min 'a' 'b'
'a'
Prelude > "elegant" < "elephant"
True
Prelude > [1, 2, 3] < [1, 2]
False
Prelude > ('a', 2) < ('b', 1)
True
Prelude > ('a', 2) < ('a', 1)
False
```

Show - სანახაობრივი ტიპები

შეიცავს ტიპებს, რომელთა მნიშვნელობები შეიძლება გარდავქმნათ სიმბოლოების სტრიქონად შემდეგი მეთოდით:

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

ყველა ძირითადი ტიპი: `Bool`, `Char`, `String`, `Int`, `Integer` და `Float` არის `Show` კლასის ეგზემპლარი.

მაგალითები:

```
Prelude > show False
"False"
Prelude > show 'a'
" 'a' "
Prelude > show 123
"123"
Prelude > show [1, 2, 3]
"[1,2,3]"
Prelude > show ('a', False)
"('a',False)"
```

Read - ადვილწასაკითხი ტიპები

ეს კლასი Show კლასის საწინააღმდეგოა და შეიცავს ტიპებს, რომელთა მნიშვნელობების სიმბოლური სტრიქონების კონვერტირება შეიძლება მეთოდით:

```
read ::      String ->a
```

ყველა ძირითადი ტიპი: Bool, Char, String, Int, Integer, Float არის Read კლასის ეგზემპლარი, როგორც სიისა და კორტეჟის ტიპი.

```
Prelude > read "False" :: Bool
False
Prelude > read " 'a' " :: Char
'a'
Prelude > read "123" :: Int
123
Prelude > read "[1,2,3]" :: [Int ]
[1, 2, 3]
Prelude > read "('a',False)" :: (Char, Bool )
('a', False)
```

Num - რიცხვითი ტიპები

ეს კლასი შეიცავს ტიპებს, რომლებიც Eq და Show კლასის ეგზემპლარებია. ამ მნიშვნელობათა დამუშავება შეიძლება 6 მეთოდით:

```
(+)     ::      a ->a -> a
(-)       ::    a -> a -> a
(*)       ::    a -> a -> a
negate    ::    a ->a
abs       ::    a -> a
signum    ::    a ->a
```

მაგალითები:

```
Prelude> 1 + 2
3
Prelude > 1.1 + 2.2
3.3
Prelude > negate 3.3
-3.3
Prelude > abs (-3)
3
Prelude > signum (-3)
-1
```

Integral - მთელრიცხვთა ტიპები

ეს კლასი შეიცავს ტიპებს, რომლებიც Num რიცხვითი კლასის ეგზემპლარებია და მათი მნიშვნელობები მთელი რიცხვებია. მხარდაჭერილია ორი მეთოდი:

```
div       ::     a -> a -> a
mod       ::     a -> a -> a
```

მაგალითები:

```
Prelude> 7 `div` 2
3
Prelude> 7 `mod` 2
1
```

Fractional - წილადური ტიპები

ეს კლასი შეიცავს ტიპებს, რომლებიც Num კლასის ეგზემპლარებია და მათი მნიშვნელობები მთელი რიცხვები არ არის. მხარდაჭერილია ორი მეთოდი - წილად რიცხვთა გაყოფა და შექცევის (შებრუნების):

```
(/)       ::     a -> a -> a
recip     ::     a -> a
```

`Float` ძირითადი ტიპი წარმოადგენს `Fractional` კლასის ეგზემპლარს. მაგალითები:

```
Prelude>7.0 / 2.0
3.5
prelude> recip 2.0
0.5
```

დაბოლოს, უნდა აღინიშნოს, რომ `Haskell`-ში ახალი ფუნქციის განსაზღვრა მიზანშეწონილია დაიწყოთ მისი ტიპის ჩაწერით. ამასთან, იმ პოლიმორფული ფუნქციების ტიპების ფორმულირებისას, რომლებიც იყენებენ რიცხვებს, ტოლობებს, ან შედარების ნიშნებს, იზრუნეთ შეზღუდვების აუცილებელი კლასების ჩართვაზე.

სავარჯიშოები

დაასახელეთ ფუნქციების ტიპი პოლიმორფიზმის გათვალისწინებით, თუ ისინი განსაზღვრულია შემდეგი ფორმულებით:

1. `fun a=a+34, fun a=a/3 , fun x y =x `mod` y, fun a b=a>b, fun a b=a==b,fun a b=2*a+ 3*b, fun x y z=x/y +z, fun a b c =a==b&& a/=c.`
2. `fun x y =x `div` y, fun a b=a<b, fun a b=a/=b, fun x=5*x, fun x=5/x, fun v s =v-s*5, fun x y z=z+div x y, fun a b c=a+b `div` c.`
3. `fun a b c=a*b*c, fun x y=x/y, fun x y=x `div` y, fun a b=a>=b, fun a b=a==b, fun a c=a/3.0 +c , fun a b=(-b) +a*a, fun x y z =mod x y+z.`

4. `fun a b=a>=b, fun a b=a==b, fun b=(-b), fun b c=b `div` c, fun (y z)=y+z, fun x z=5*x *z, fun x y z=x- y `mod` z, fun a=1- a/3.0.`
5. `fun x y=x/y, fun x y=x `div` y, fun a b=a/=b, fun a b=a>=b, fun x a =a+5/x, fun a b c =a+b+c, fun a b c=a>b && a>c.`

ტიპის გამოცხადებები

Haskell-ში ახალი სახელი არსებული ტიპისათვის შეიძლება იყოს განსაზღვრული ტიპის გამოცხადების (დეკლარაციის) გამოყენებით.

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

ამ გამოცხადებით `String` გახდა `[Char]`-ის სინონიმი.

ტიპის გამოცხადებები შეიძლება გამოიყენებოდეს სხვა ტიპების წაკითხვის გასაიოლებლად. მაგალითად,

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

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

```
origin     :: Pos
origin      = (0,0)
left       :: Pos → Pos
left (x,y) = (x-1,y)
```

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

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

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

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

ტიპის გამოცხადებები შეიძლება ჩალაგებულიც იყოს ერთმანეთში:

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

მაგრამ ისინი არ შეიძლება იყოს რეკურსიული:

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

მომხმარებლის ტიპები

პროგრამისტს აქვს შესაძლებლობა სტანდარტული ტიპების გვერდით განსაზღვროს მონაცემების თავისი საკუთარი, სპეციფიკური ტიპები. ამისთვის საჭიროა გამოყენებულ იქნეს გასაღები სიტყვა `data`.

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

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

ამ ჩანაწერით ნათქვამია, რომ `Bool` არის ახლი ტიპი ორი ახალი მნიშვნელობით `False` და `True`. შევნიშნოთ, რომ `False` და `True` მნიშვნელობებს ეწოდება კონსტრუქტორები მოცემული `Bool` ტიპისათვის. ტიპისა და კონსტრუქტორის სახელები უნდა იწყებოდეს ზედა რეგისტრის ასოთი. მონაცემთა გამოცხადებები თავისუფალი გრამატიკების კონტექსტის მსგავსია. პირველი აღწერს მნიშვნელობათა ტიპს, უკანასკნელი კი - ენის წინადადებებს.

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

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

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

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

შევნიშნოთ, რომ `Shape`-ს აქვს ფორმის მნიშვნელობები: `Circle r`, სადაც `r` – მცოცავწერტილიანი რიცხვია და `Rect x y`, სადაც `x` და `y` – ასევე მცოცავწერტილიანი რიცხვებია.

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

```
Circle :: Float → Shape
Rect   :: Float → Float → Shape
```

გასაკვირი არ არის, რომ მონაცემთა გამოცხადებებს თავადაც შეიძლება გააჩნდეს პარამეტრები. მაგალითად,

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

ჩანაწერის გათვალისწინებით შეგვიძლია განვსაზღვროთ :

```
safediv    :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
safehead   :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

წყვილები

მაგალითისთვის განვიხილოთ წყვილების განმარტება, რომელიც ძალზე წააგავს სტანდარტულს:

```
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 ტიპების 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 გვაქვს ერთადერთი კონსტრუქტორი `Pair`. არ არის აუცილებელი მონაცემების კონსტრუქტორის სახელი ემთხვეოდეს ტიპის სახელს, თუმცა ჩვენს მაგალითში ეს ბუნებრივია. კონსტრუქტორის სახელის შემდეგ ჩვენ ისევ ვწერთ `a`-ს. ეს აღნიშნავს, რომ წყვილის კონსტრუირებისთვის საჭიროა ორი მნიშვნელობა: პირველი, რომელიც ეკუთვნის `a`-ს, მეორე – `b`-ს.

ამ განმარტებას შემოყავს ფუნქცია `Pair :: a -> b -> Pair a b`, რომელიც 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 შესაბამისი ფუნქციებისთვის დამახასიათებელია ნიმუშთან შედარებისას მათი გამოყენება. ასე რომ, ფუნქციები, რომლებითაც ვღებულობთ ჩვენი წყვილისთვის პირველ და მეორე ელემენტს, შეიძლება განიმარტოს შემდეგნაირად:

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

განხილული მაგალითი ბადებს კითხვას: 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 იმუშავებს და მეორეც, შეიძლება დამატებით დაედოს შეზღუდვები. მაგალითად, წარმოიდგინეთ, რომ საჭიროა ტიპი, რომელიც შექმნის წყვილს ერთი და იმავე 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.

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 ტი-*

ჰეგს და იგი შეესაბამება ენა C++-ის `enum` კონსტრუქციას. ასეთი ტიპები ძალზე სასარგებლოა. მაგალითად, სტანდარტული ტიპი `Bool` ასე განისაზღვრება:

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

მრავლობითმა კონსტრუქტორებმა ასევე შეიძლება მიიღონ პარამეტრები. შევნიშნოთ, რომ ტიპი `Color` საშუალებას გვაძლევს განვსაზღვროთ მხოლოდ სამი ფიქსირებული ფერი. გავაფართოვოთ იგი ისე, რომ 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 პოვ-

ნის ფუნქცია აბრუნებს ნაპოვნ ფესვებს; ფუნქცია, რომელიც ეძებს სიაში პირველ არაუარყოფით რიცხვს, აბრუნებს ამ რიცხვს და ა. შ. ამასთან, შეიძლება განტოლება არ იხსნებოდეს ან სიაში არ იყოს არაუარყოფითი რიცხვები და ა.შ. როგორ უნდა შევატყობინოთ მას, ვინც ფუნქცია გამოიძახა? ზოგჯერ შესაძლოა შეთანხმება, რომ რომელიმე სპეციალური მნიშვნელობა (მაგალითად, -1) აღნიშნავდეს, რომ არ გვაქვს 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 a = Nothing | Just a
```

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

ფუნქცია აბრუნებს `ax + b = 0` განტოლების ფესვს:

```
solve :: Double -> Double -> Maybe Double
solve 0 b = Nothing
solve a b = Just (-b / a)
```

ფუნქცია აბრუნებს სიის პირველ არაუარყოფით ელემენტს:

```
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 მოხდეს შედარება ნიმუშთან და თუ ჩვენ დაგვავიწყდება `Nothing` შემთხვევის დამუშავება, კომპილერმა შეიძლება მოგვცეს გაფრთხილება.

ტიპების კლასები მნიშვნელოვნად ამარტივებს მომხმარებლის ტიპებთან მუშაობას. როგორც უკვე ვიცით, ტიპების კლასი წარმოადგენს განსაზღვრულ სიმრავლეს ტიპებისა, რომლებსაც მთელი რიგი საერთო თვისებები აქვთ. მაგალითად, ტიპების კლასში `Eq` შედის ყველა ტიპი, რომელთა ობიექტებისთვისაც განსაზღვრულია ტოლობის მიმართება, ანუ, თუ `x` და `y` ეკუთვნის ერთსა და იმავე ტიპს, რომელიც შედის `Eq` 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.

ასევე მნიშვნელოვანი კლასია `Show` კლასი. მასში შედის ყველა ტიპი, რომელთა ობიექტები შეიძლება გარდაიქმნას სტრიქონში იმისათვის, 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 დაბეჭდავს. ეს, რა თქმა უნდა, მოუხერხებელია, ამიტომ ტიპების განსაზღვრისას შესაძლებელია ის მივაკუთვნოთ სასურველ კლასს. ამისთვის, ტიპის განსაზღვრის შემდეგ საჭიროა დაემატოს გასაღები სიტყვა `deriving` და ფრჩხი-

ლებში ჩამოვთვალოთ კლასები, რომლებსაც უნდა ეკუთვნოდეს ტიპი. მაგალითად:

-- ტიპი, რომელიც აღწერს დღის პერიოდს (დილა, შუადღე, საღამო, ღამე):

```
data DayTime = Morning
     | Afternoon
     | Evening
     | Night deriving (Eq, Show)
```

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

დავალებები

1. თანამედროვე web-მაღაზიაში ხშირად იყიდება წიგნები, ვიდეოკასეტები და კომპაქტდისკები. ამ მაღაზიის მონაცემთა ბაზა თითოეული სახეობის საქონლისთვის უნდა შეიცავდეს შემდეგ მახასიათებლებს:

   - ✓ წიგნებისთვის: დასახელება და ავტორი
   - ✓ ვიდეოკასეტებისთვის: დასახელება
   - ✓ კომპაქტდისკებისთვის: დასახელება, შემსრულებელი და კომპოზიციების რაოდენობა.

   a. შექმენით მონაცემთა ტიპი `Product`, რომელიც წარმოადგენს საქონლის ამ ტიპებს.
   b. განსაზღვრეთ ფუნქცია `getTitle`, რომელიც დააბრუნებს საქონლის დასახელებას.

c. ამ ფუნქციის საფუძველზე განსაზღვრეთ ფუნქცია `getTitles`, რომელიც საქონლის სიის მიხედვით დააბრუნებს მათი დასახელებების სიას.

d. განსაზღვრეთ ფუნქცია `bookAuthors`, რომელიც საქონლის სიის მიხედვით დააბრუნებს წიგნების ავტორების სიას.

e. განსაზღვრეთ ფუნქცია
```
lookupTitle :: String -> [Product] -> Maybe Product
```
რომელიც დააბრუნებს საქონელს მოცემული დასახელებით (ყურადღება მიაქციეთ ფუნქციის შედეგის ტიპს).

f. განსაზღვრეთ ფუნქცია
```
lookupTitles :: [String] -> [Product] -> [Product]
```
ის პარამეტრად იღებს დასახელებებისა 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 ტიპი, რომელიც წარმოადგენს გეომეტრიულ ფუნქციას. ფიგურა შეიძლება იყოს წრეწირი (ხასიათდება ცენტრის კოორდინატებით და რადიუსით), მართკუთხედი (ხასიათდება მარცხენა ზედა და ქვედა მარჯვენა კუთხეების კოორდინატებით), სამკუთხედი (წვეროების კოორდინატები) და ტექსტური ველი (მისთვის აუცილებელია შენახულ იქნეს მარცხენა ქვედა კუთხე, შრიფტი და სტრიქონი, რომელიც წარწერას წარმოადგენს). შრიფტი მოიცემა სამელემენტიანი სიმრავლიდან: `Courier`, `Lucida` და `Fixedsys`. განსაზღვრეთ შემდეგი ფუნქციები:

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

b. ფუნცია `getRectangles`, რომელიც სიიდან მხოლოდ მართკუთხედებს არჩევს.

c. ფუნქცია `getBound`, რომელიც მოცემული ფიგურისთვის აბრუნებს მართკუთხედს, რომელიც საზღვრავს ამ ფიგურას.

d. ფუნქცია `getBounds`, რომელიც ფიგურების მოცემული სიისთვის აბრუნებს მათი შემომსაზღვრელი მართკუთხედების სიას.

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

f. ფუნქცია `move`, რომელიც მოცემული ფიგურისთვის და ძვრის ვექტორისთვის აბრუნებს ახალ ფიგურას, რომელიც დაძრული იქნება მოცემული ვექტორით.

3. 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 წარმოად-

გენს უძრავ ობიექტს, მეორე – მის ფასს. განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს უძრავი ქონების ობიექტებზე ინფორმაციას. განსაზღვრეთ შემდეგი ფუნქციები:

a. `getHouses` - არჩევს მონაცემთა ბაზიდან მხოლოდ კერძო სახლებს.
b. `getByPrice` - არჩევს მონაცემთა ბაზიდან მხოლოდ უძრავი ქონების იმ ობიექტებს, რომელთა ფასი ნაკლებია მითითებულზე.
c. `getByLevel`, ირჩევს მონაცემთა ბაზიდან ბინებს, რომლებიც მოცემულ სართულზე მდებარეობს.
d. `getExceptBounds` – ირჩევს მონაცემთა ბაზიდან ბინებს, რომლებიც არ მდებარეობს პირველ და ბოლო სართულებზე.

შეიმუშავეთ მონაცემების ტიპი, რომელიც წარმოადგენს უძრავი ქონების ობიექტებზე სხვადასხვა მოთხოვნას: 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.

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

a. `isPeriodic` - ამოწმებს, რომ მისი არგუმენტი არის პერიოდული გამოცემა.
b. `getByTytle` - შენახული ობიექტებიდან (მონაცემთა ბაზიდან) იღებს ობიექტებს მოცემული დასახელებით.
c. `getByMonth` - იღებს მონაცემთა ბაზიდან პერიოდულ გამოცემებს, რომლებიც მოცემული წლის განმავლობაში გამოდის თვეში ერთხელ (შევნიშნოთ, რომ გაზეთები გამოდის თვეში რამდენჯერმე).
d. `getByMonths` - ისევე მუშაობს, როგორც წინა ფუნქცია, მხოლოდ არგუმენტად ღებულობს თვეების სიას.
e. `getAuthors` - აბრუნებს ავტორების სიას იმ გამოცემებისა, რომლებიც ბიბლიოთეკაში ინახება.

5. დაპროგრამების 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 იდენტიფიკატორის სახელისგან და მისი ტიპისგან. შეიმუშავეთ მონაცემთა ტიპი, რომელიც წარმოადგენს აღწერილ ინფორმაციას. განსაზღვრეთ შემდეგი ფუნქციები:

a. `isStructured`, რომელიც ამოწმებს, რომ მისი არგუმენტი არის რთული ტიპის.
b. `getType` - მოცემული სახელითა და იდენტიფიკატორების სიით (მონაცემთა ბაზა) აბრუნებს მოცემული სახე-

ლის იდენტიფიკატორის ტიპს (გაითვალისწინეთ, რომ ამ სახელის იდენტიფიკატორი ბაზაში შეიძლება არც იყოს).

c. `getFields` - მოცემული სახელის მიხედვით აბრუნებს იდენტიფიკატორის ველების სიას, თუ იგი არის სტრუქტურის ტიპის.

d. `getByType` - აბრუნებს მონაცემთა ბაზიდან მოცემული ტიპის იდენტიფიკატორების სახელების სიას.

e. `getByTypes` - წინა ფუნქციის ანალოგიურია, მაგრამ ერთი არგუმენტის ნაცვლად იღებს ტიპების სიას. ამ ფუნქციის საშუალებით, მაგალითად, შეიძლება მივიღოთ ყველა რიცხვითი ტიპის იდენტიფიკატორების სია.

6. განსაზღვრეთ სტრიქონებზე შემდეგი ოპერაციები:

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

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

a. `process`, რომელიც არგუმენტად იღებს მოქმედებას და სტრიქონს და აბრუნებს სტრიქონს, რომელიც მოდიფიცირებულია მოქმედების შესაბამისად.

b. `processAll` - წინა ფუნქციის ანალოგიურად, მხოლოდ ღებულობს მოქმედებების სიას და ასრულებს მათ მოცემულ სტრიქონზე თანმიმდევრულად.

c. deleteAll - იღებს ორ სტრიქონს და აგდებს მეორე სტრიქონიდან პირველი სტრიქონის ყველა სიმბოლოს. რეალიზაციისას აუცილებელია გამოიყენოთ ფუნქცია processAll.

7. ელექტრონულ ბლოკნოტში ინახება შემდეგი სახის ჩანაწერები: შეხსენება ნაცნობების დაბადების დღეების, ნაცნობების ტელეფონის ნომრები 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:

a. getByName - აბრუნებს ინფორმაციას ადამიანზე მისი სახელით (მისი ტელეფონის ნომერს და დაბადების თარიღს).
b. getByLetter - აბრუნებს ადამიანების სიას, რომელთა შესახებ ინფორმაცია არის ბლოკნოტში და რომელთა სახელი იწყება მოცემულ ასოზე.
c. getAssignment - აბრუნებს მოცემული დროის მიხედვით საქმეების სიას (ინფორმაციას დანიშნულ შეხვედრებზე და მეგობრების ტელეფონის ნომრებს, ვისაც უნდა მიულოცოს დაბადების დღე).

8. კლავიშები კლავიატურაზე შეიძლება იყოს ან მმართველი, ან ანბანურ-ციფრული. ანბანურ-ციფრულ კლავიშზე დაჭერა შეიძლება განხორციელდეს Shift-თან ერთად. მმართველი კლავიშებიდან ჩვენ გვაინტერესებს მხოლოდ CapsLock. ანბანურ-ციფრულ კლავიშზე ყოველი დაჭერა შეიცავს ინფორმაციას სიმბოლოს სახით. CapsLock-ის დაჭერის შემდეგ სიმბოლოები გადადის მაღალ

რეგისტრში (თუ ისინი არ არის Shift-თან ერთად დაჭერილი) CapsLock-ის შემდეგ დაჭერამდე. თუ CapsLock-ის რეჟიმი არ არის გააქტივებული, მაშინ Shift-თან ერთად დაჭერილი სიმბოლოები გადადის ზედა რეგისტრში. შეიმუშავეთ მონაცემთა ტიპი, რომელიც მოცემულ ინფორმაციას წარმოადგენს. კლავიშების თანმიმდევრული დაჭერა წარმოადგინეთ სიის სახით. ძირითადი ამოცანაა, 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:

a. `getAlNum` - აბრუნებს დაჭერილი სიიდან მხოლოდ ანბანურ-ციფრულ კლავიშებზე დაჭერას.
b. `getRaw` - აბრუნებს სტრიქონს, რომელიც შედგება დაჭერილი სიმბოლოებისგან კლავიშების `Shift` და `CapsLock`-ის გაუთვალისწინებლად.
c. `isCapsLocked` - ბოლო დაჭერის მიხედვით გაარკვევს, დარჩა თუ არა მის შემდეგ რეჟიმი `CapsLock` აქტიურ მდგომარეობაში.
d. `getString` - გადაჰყავს კლავიშებზე დაჭერა სტრიქონში.

ფუნქციის რეალიზებისას შესაძლოა გამოიყენოთ სტანდარტული ფუნქციები `toUpper` და `toLower`, რომლებსაც გადაჰყავთ სიმბოლო, შესაბამისად, ზედა ან ქვედა რეგისტრში. ეს ფუნქციები განსაზღვრულია მოდულში `Char`.მათი გამოყენება რომ შეძლოთ, პროგრამის დასაწყისში დაუმატეთ სტრიქონი:

```
import Char.
```

9. 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 და თემის დასახელებით. შეიმუშავეთ მონაცემთა ტიპი, რომელიც შეიცავს ინფორმაციას დავალებების შესახებ. სტუდენტის სასწავლო გეგმა წარმოადგენს წყვილების სიას, რომლის პირველი წევრი წარმოადგენს დავალებას, მეორე – კვირის ნომერს, როცა ეს დავალება ჩაბარდა. თუ დავალება არ შესრულდა, წყვილის მეორე ელემენტი უნდა იყოს ცარიელი (გამოიყენეთ ტიპი `Maybe`). განსაზღვრეთ შემდეგი ფუნქციები:

a. `getByTitle` – აბრუნებს დავალებას, რომელიც აუცილებელია ჩაბარდეს მოცემულ საგანში.
b. `getReferats` – აბრუნებს რეფერატების თემების სიას.
c. `getRest` – აბრუნებს ჩაუბარებელი საგნების სიას.
d. `getRestForWeek` – აბრუნებს დავალებების სიას, რომლებიც მოცემული კვირისთვის არ არის შესრულებული.
e. `getPlot` – ადგენს სიას, რომელიც შედგება წყვილებისგან, რომლის პირველი ელემენტი არის კვირის ნომერი, ხოლო მეორე – ამ კვირაში ჩაბარებული დავალებების რაოდენობა.

რეკურსიული ტიპების გამოცხადება

`Haskell`-ში ახალი ტიპები შეიძლება გამოცხადდეს საკუთარი თავის მეშვეობით. სხვანაირად რომ ვთქვათ, ტიპები შეიძლება რეკურსიული იყოს.

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

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

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

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

შეგვიძლია მივიჩნიოთ, რომ `Nat` ტიპის მნიშვნელობები ნატურალური რიცხვებია, სადაც `Zero` წარმოადგენს `0`-ს, ხოლო `Succ` იძლევა `1+` ფუნქციის მომდევნო ელემენტს. მაგალითად, `Succ (Succ (Succ Zero))` მნიშვნელობა წარმოადგენს ნატურალურ რიცხვს `1 + (1 + (1 + 0))=3`

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

```
nat2int          :: Nat → Int
nat2int Zero      = 0
nat2int (Succ n) = 1 + nat2int n
int2nat          :: Int → Nat
int2nat 0         = Zero
int2nat (n+1)     = Succ (int2nat n)
```

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

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

მაგრამ რეკურსიის საშუალებით `add` ფუნქცია შეიძლება განისაზღვროს გარდაქმნის გამოუყენებლადაც:

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

მაგალითად:

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

შევნიშნოთ, რომ რეკურსიული განსაზღვრება `add`-თვის შეესაბამება შემდეგ წესებს:

`0+n = n` და `(1+m)+n = 1+(m+n)`.

რეკურსიული ტიპები

მონაცემების ტიპების განსაზღვრისას მის მარჯვენა მხარეს შეიძლება გამოყენებულ იქნეს 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 ტიპებთან მუშაობს, თვითონაც არის რეკურსიული.

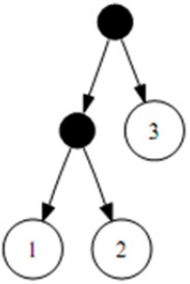
მაგალითად, განვსაზღვროთ ფუნქცია `treeSize`, რომელიც დააბრუნებს ხეში ფოთლების რაოდენობას. ის ჩაიწერება შემდეგნაირად:

```
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
```

სიები, როგორც რეკურსიული ტიპები

სიაც ასევე წარმოადგენს რეკურსიულ ტიპს. განვიხილოთ შემდეგი პოლიმორფული ტიპი:

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

`List` 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 ცარიელია, 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 Haskell ენის ნამდვილი კოდი!

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

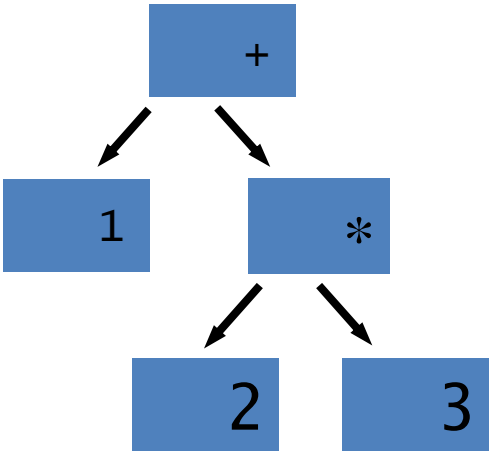
List ტიპის მნიშვნელობისთვის შეიძლება განვსაზღვროთ ყველა ის ფუნქცია, რომელიც განსაზღვრულია სიებისთვის. მოვიყვანოთ ფუნქციების head, tail და map მაგალითები:

```
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
```

განვიხილოთ რიცხვთა შეკრებითა და გამრავლებით აგებულ გამოსახულებათა მარტივი ფორმა.



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

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

მაგალითად, გამოსახულება წინა სურათზე შემდეგნაირად იქნებოდა წარმოდგენილი:

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

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

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

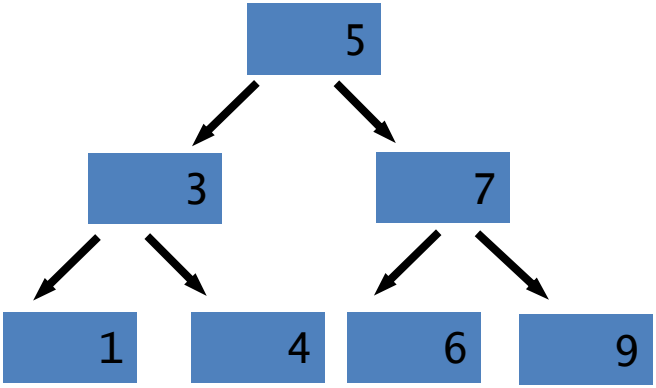
შევნიშნოთ, რომ აქ სამი ტიპის კონსტრუქტორია განსაზღვრული:

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

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

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

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



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

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

მაგალითად, წინა სურათზე მოცემული ხე ასეთნაირად აღმოჩნდება ჩაწერილი:

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

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

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

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

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

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

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

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

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

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

სავარჯიშოები

1. რეკურსიისა და `add` ფუნქციის გამოყენებით განსაზღვრეთ ფუნქცია, რომელიც ორ ნატურალურ რიცხვს ამრავლებს.
2. განსაზღვრეთ სათანადო `fold` ფუნქცია გამოსახულებებისათვის და მოიყვანეთ ამ ფუნქციის გამოყენების რამდენიმე მაგალითი.
3. ბინარული ხე სრულია, თუ ყოველი კვანძის ორი ქვეხე ერთნაირი ზომისაა. განსაზღვრეთ ფუნქცია, რომელიც არკვევს ხის სისრულეს.

სინტაქსური ხეები

პროგრამირებაში ფართოდ გამოიყენება ხის ტიპის სტრუქტურები. მაგალითად, 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 კონსტანტას `(Const)`, ან ორი გამოსახულების ჯამს, ან მათ 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 განვსაზღვროთ შემდეგნაირად:

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

ტიპი Expr შესაძლოა გავაფართოვოთ, თუ შემოვიტანთ გამოსახულებებში ცვლადების გამოყენების შესაძლებლობას:

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

კონსტრუქტორი `Var` აღწერს 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))
```

ვნახოთ რა შედეგს იძლევა დიფერენცირების ფუნქცია შემდეგ არგუმენტზე: $x + x^2$, ამასთან, `Expr`-ის განსაზღვრების შემდეგ საჭიროა დაემატოს `deriving(Show)` ტიპი:

```
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+(1Äx+xÄ1)`, რომელიც წარმოადგენს სწორ გამოსახულებას, ოღონდ, რასაკვირველია, საჭიროებს გამარტივებას.

ფუნქცია `diff`-ის სხვა შეზღუდვაა ის, რომ ფუნქციაში არ განირჩევა, 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, რეკომენდებულია ფუნქციის `parseExpr`-ის გამოყენება (განსაზღვრულია ფაილში `expr.hs`). თავიდან დაუმატეთ სტრიქონი

```
import Exp
```

ფუნქციას `parseExpr` აქვს შემდეგი ტიპი:

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

მოცემული სტრიქონისთვის იგი აბრუნებს ტიპი `Expr`-ის მნიშვნელობას:

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

დავალებები

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

   a. განსაზღვრეთ ფუნქცია `diff`, რომელიც დამატებით არგუმენტად ღებულობს ცვლადის სახელს, რომლის მიხედვითაც აუცილებელია მოხდეს დიფერენცირება.

   b. განსაზღვრეთ ფუნქცია `simplify`, რომელიც ამარტივებს `Expr` ტიპის გამოსახულებას შემდეგი წესების გამოყენებით:

      - `x + 0 = 0 + x = x`
      - `x · 1 = 1 · x = x`
      - `x · 0 = 0 · x = 0`
      - და ა.შ.

   c. განსაზღვრეთ ფუნქცია `toString`, რომელიც `Expr` ტიპის არგუმენტს შეუსაბამებს სტრიქონს. მაგალითად, ფუნქციის გამოყენების შედეგი გამოსახულებასთან `Add (Mult (Const 2) (Var "x")) (Var "y")` უნდა იყოს სტრიქონი `"2*x+y"`. გაითვალისწინეთ ფრჩხილების გამოყენების შესაძლებლობა. მაგალითად, გამოსახულება `Mult (Const 2) (Add (Var "x") (Var "y"))` უნდა გარდაიქმნას სტრიქონად `"2*(x+y)"`.

   d. 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 მნიშვნელობე-

ბის გათვალისწინებით. მაგალითად, გამოსახულება `eval (Add (Var "x") (Var "y")) [("x",1),("y",2)]` უნდა იძლეოდეს რიცხვ 3-ს.

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

   a. `lengthList`, რომელიც აბრუნებს `List` ტიპის სიის სიგრძეს.
   b. `nthList`, რომელიც აბრუნებს სიის `n`-ურ ელემენტს.
   c. `removeNegative`, რომელიც მთელი რიცხვების სიიდან (ტიპი `List Integer`) ამოაგდებს უარყოფით ელემენტებს.
   d. `fromList`, რომელიც გარდაქმნის `List` ტიპის სიას ჩვეულებრივ სიად.
   e. `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 მოვიპოვოთ ინფორმაცია ხიდან. განსაზღვრეთ მონაცემთა აღწერილი ტიპი და შემდეგი ფუნქციები:

a. add, რომელიც ხეს უმატებს გასაღებისა და მნიშვნელობის მოცემულ წყვილს.
b. Find - აბრუნებს სტრიქონის შესაბამის რიცხვს.
c. Exists - ამოწმებს, რომ ელემენტი მოცემული გასაღებით არის ხეში.
d. 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:

a. dirAll - აბრუნებს კატალოგის ყველა ფაილის სრული სახელების სიას, ქვეკატალოგების ჩათვლით.
b. Find - აბრუნებს გზას, რომელსაც მივყავართ მოცემული სახელის ფაილთან. მაგალითად, თუ კატალოგი შეიცავს ფაილებს a, b, c და b წარმოადგენს კატალოგს, რომელიც შეიცავს x და y, მაშინ ძებნის ფუნქციამ უნდა დააბრუნოს სტრიქონი "b/x".
c. 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`
- `~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:

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

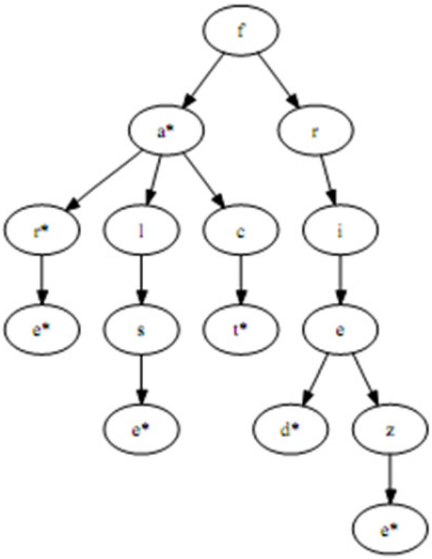
b. დავუშვათ მოცემულია ცვლადების სახელების სია და მათი `Bool` ტიპის მნიშვნელობები, მაგალითად, `[("x",True), ("y",False)]`. განსაზღვრეთ ფუნქცია `truthValue :: Prop -> [(String,Bool)] -> Bool`, რომელიც აბრუნებს `True`-ს, თუ ცვლადებს აქვთ სიით მოცემული მნიშვნელობები, წინააღმდეგ შემთხვევაში - `Fasle`-ს.

c. 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 `trees` გამოიყენება ლექსიკონების წარმოსადგენად. ხის ყოველი წვერო შეიცავს შემდეგ ინფორმაციას: სიმბოლოს, ლოგიკურ მნიშვნელობას და ქვეხეების სიას (ნებისმიერ

წვეროს შეიძლება ჰქონდეს ნებისმიერი რაოდენობის შვილობილი ხეები. `trees` -ხის მაგალითი მოყვანილია სურ. 1-ზე.
ლოგიკური მნიშვნელობა `True` აღნიშნავს სიტყვის ბოლოს, რომელიც იკითხება ხის წვეროდან. სურათზე ასეთი წვეროები აღნიშნულია 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 `trees`-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. `trees`-ხეები

- `completions`, რომელიც მოცემული სტრიქონის შესაბამისად აბრუნებს სიას იმ სიტყვებისა, რომლებიც იწყება მოცემული სტრიქონით (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:

```
data Number = Zero | Next Number
```

ანუ, რიცხვი ან ნულია (`Zero`), ან განისაზღვრება, როგორც რიცხვი, რომელსაც მოსდევს შემდეგი ციფრი. მაგალითად, რიცხვი 3 ჩაიწერება როგორც `Next (Next (Next Zero))`. ასეთი წარმოდგენისთვის განსაზღვრეთ შემდეგი ფუნქციები:

a. `fromInt` - მოცემული მთელი `Integer` ტიპის რიცხვისთვის აბრუნებს მისი მნიშვნელობის შესაბამის `Number` ტიპის მნიშვნელობას.
b. `toInt`, რომელიც გარდაქმნის `Number` ტიპის მნიშვნელობას შესაბამის მთელ რიცხვად.
c. `plus :: Number -> Number -> Number`, რომელიც უმატებს თავის არგუმენტებს.
d. `mult :: Number -> Number -> Number`, რომელიც ამრავლებს თავის არგუმენტებს.
e. `dec`, რომელიც აკლებს თავის არგუმენტს 1-ს. `Zero`-სთვის ფუნქციამ უნდა დააბრუნოს `Zero`.
f. `fact`, რომელიც ითვლის ფაქტორიალს.

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

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

a. `getSubordinate`, რომელიც აბრუნებს მოცემული თანამშრომლის დაქვემდებარებულების სიას.
b. `getAllSubordinate`, რომელიც აბრუნებს მოცემული თანამშრომლის ყველა დაქვემდებარებულის სიას, მათ შორის ირიბი დაქვემდებარებულებისასაც.
c. `getBoss` - აბრუნებს მოცემული თანამშრომლის უფროსს.
d. `getList` - აბრუნებს წყვილების სიას, რომლებიდანაც პირველი ელემენტია თანამშრომლის სახელი, მეორე – მისი დაქვემდებარებულების (ირიბი დაქვემდებარებულების ჩათვლით) რაოდენობა.

9. სიბრტყეზე არე წარმოადგენს ან მართკუთხედს, ან 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:

a. `contains`, რომელიც შეამოწმებს, მოცემული წერტილი არის თუ არა არეში.
b. `isRectangular`, ამოწმებს, რომ არე მოიცემა მხოლოდ მართკუთხედებით.
c. `isEmpty` - ამოწმებს, რომ არე არის ცარიელი, ანუ სიბრტყის არცერთი წერტილი მასზე არ გვხვდება.

10. ობიექტორიენტირებულ ენაზე კლასი შეიცავს მეთოდების ჯგუფს (მოცემულ დავალებაში კლასის მონაცემები – ველები არ განვიხილოთ). ამას გარდა, მას შეიძლება ჰქონდეს ერთადერთი მშობელი კლასი (მრავლობით მემკვიდრეობითობას არ განვიხილავთ). თუმცა არსებობს კლასები მშობლების გარეშე. მემკვიდრე-

ობითობის დროს მშობლის მეთოდებს ემატება შთამომავლის მეთოდები. განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს ინფორმაციას კლასების იერარქიის შესახებ. აღწერეთ შემდეგი ფუნქციები:

a. `getParent` - აბრუნებს მოცემული სახელის კლასის უშუალო წინაპარს.
b. `getPath` - აბრუნებს მოცემული კლასის ყველა წინაპარს (უშუალო წინაპარს, წინაპრის წინაპარს და ა.შ).
c. `getMethods` - აბრუნებს მოცემული კლასის მეთოდებს მემკვიდრეობითობის გათვალისწინებით.
d. `Inherit` - უმატებს კლასების იერარქიას მოცემული სახელის კლასს,რომელიც მემკვიდრეობით მიიღება მოცემული წინაპრიდან.

თავი 1.6. მაღალი 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
```

შევნიშნოთ, რომ ეს ფუნქციები იყენებს ერთსა და იმავე 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 აბსტრაჰირება? 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
```

ახლა ფუნქციები `sqrtList` და `logList` შეიძლება ასე განისაზღვროს:

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

ანდა, კარირების გათვალისწინებით:

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

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

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

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

შევნიშნოთ, რომ მაღალი რიგის ფუნქციათა ალგებრული თვისებები შეიძლება გამოიყენებოდეს პროგრამათა დასაბუთებისათვის.

ფუნქცია MAP

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 მსგავს ფუნქციებს, რომლებიც არგუმენტად იღებენ სხვა ფუნქციებს, უწოდებენ *მაღალი რიგის ფუნქციებს.* მათ ძალზე ხშირად იყენებენ ფუნქციონალური 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 თვალსაჩინო, ვიდრე ჩვეულებრივი რეალიზაციები.

მაგალითად,

```
Prelude > map (+1) [1,3,5,7]
[2,4,6,8]
Prelude > map isDigit ['a', '1', 'b', '2']
[False, True, False, True ]
Prelude > map reverse ["abc", "def", "ghi"]
["cba", "fed", "ihg"]
```

`map` ფუნქცია შეიძლება განისაზღვროს განსაკუთრებულად მარტივი ფორმით სიის კონსტრუქტორის გამოყენებისას:

```
map f xs = [f x | x ← xs]
```

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

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

ფუნქცია FILTER

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

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

გარდა ამ რეკურსიული განმარტებისა, `filter` ფუნქცია შეიძლება განისაზღვროს სიის კონსტრუქტორის საშუალებით:

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

გამოყენების მაგალითი:

```
Prelude > filter even [1..10]
[2,4,6,8,10]
```

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

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

`map` და `filter` ფუნქციები ხშირად ერთად გამოიყენება: `filter` ფუნქციას მიმართავენ სიიდან გარკვეული ელემენტების ასარჩევად, ხოლო შემდეგ ყოველი მათგანი გარდაიქმნება `map` ფუნქციის საშუალებით.

მაგალითად, ფუნქცია, რომელიც გვიბრუნებს სიიდან ლუწი მთელი რიცხვების კვადრატების ჯამს, განისაზღვრება ასე:

```
sumsqreven    ::       [Int] → Int
sumsqreven ns = sum (map (^2) (filter even ns))
```

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 განზოგადებას:

```
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 ახორციელებს სიის „შეხვევას“ გადაცემული პარამეტრების შესაბამისად.

რათა უკეთ გავიგოთ, თუ როგორ მუშაობს ფუნქცია `foldr`, 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 მნიშვნელობაზე. 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 : []))
1 + (2 + (3 + 0))
```

ანალოგიურად, გამრავლებისთვის:

```
[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 ასოციაციურობას. ასე რომ, მაგალითებიდან ჩანს, რომ ფუნქციის გამოყენება ჯგუფდება მარჯვნივ. განსაზღვრება 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
```

ასოციაციური ოპერაციისას, როგორიცაა შეკრება და გამრავლება, ფუნქციები `foldr` და `foldl` ეკვივალენტურია, თუმცა, თუ ოპერაცია არ არის ასოციაციური, 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`.

განვიხილოთ `foldr`-ის გამოყენების სხვა მაგალითებიც. მიუხედავად იმისა, რომ `foldr` რეკურსიის მარტივი შაბლონის ინკაფსულირებას ახდენს, შესაძლებელია მისი გამოყენება მოსალოდნელზე გაცილებით მეტი ფუნქციისათვის.

გავიხსენოთ სიგრძის `length` ფუნქცია:

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

მაგალითად:

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

ხდება ყოველი `(:)`-ის ჩანაცვლება `(λ_ n → 1+n)`-ით და `[]`-ის ჩანაცვლება 0-ით. ამრიგად, გვაქვს:

```
length = foldr (λ_ n → 1+n) 0
```

ახლა გავიხსენოთ `reverse` ფუნქცია:

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

მაგალითად:

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

ხდება ყოველი `(:)`-ის ჩანაცვლება `(λx xs → xs ++ [x])`-ით და `[]`-ის ჩანაცვლება `[]`-ით. ამრიგად, გვაქვს:

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

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

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

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

აღსანიშნავია, თუ რატომაა სასარგებლო `foldr` ფუნქცია. პირველი, – ზოგიერთი რეკურსიული ფუნქცია სიაზე, მაგალითად `sum` ფუნქცია, უფრო მარტივად განისაზღვრება `foldr`-ით; მეორე, `foldr`-ით განსაზღვრულ ფუნქციათა თვისებები შეიძლება იყოს დამტკიცებული `foldr`-ის ალგებრული თვისებების გამოყენებით და, ბოლოს, ოპტიმიზაციის გაუმჯობესებული პროგრამა შეიძლება უფრო მარტივი აღმოჩნდეს, თუ ცხადი რეკურსიის ნაცვლად გამოიყენება `foldr` ფუნქცია.

მაღალი 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 მოცემული ფუნქციის 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 (+)
```

საბიბლიოთეკო `(.)` ფუნქცია გვიბრუნებს ორი ფუნქციის კომპოზიციას, როგორც ერთ ფუნქციას:

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

მაგალითად:

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

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

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

მაგალითად,

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

მსგავსად, საბიბლიოთეკო `any` ფუნქცია არკვევს, თუ აკმაყოფილებს პრედიკატს სიის ერთი ელემენტი მაინც.

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

მაგალითად:

```
Prelude > any isSpace "abc def"
True
```

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

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

მაგალითად:

```
Prelude > takeWhile isAlpha "abc def"
"abc"
```

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

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

მაგალითად,

```
Prelude > dropWhile isSpace "   abc"
"abc"
```

სავარჯიშოები

1. გამოსახეთ `[f x | x ← xs, p x]` კონსტრუქტორი, რომელიც იყენებს `map` და `filter` ფუნქციებს.
2. `foldr` ფუნქციის გამოყენებით ხელახლა განსაზღვრეთ `map f` და `filter p` ფუნქციები.

მაღალი რიგის ფუნქციის გამოყენებისას ხშირად აუცილებელია განისაზღვროს ბევრი დამატებითი ფუნქცია. მაგალითად, ფუნქცია `getPositive`-ისათვის ჩვენ მოგვიწია განგვესაზღვრა დამატებითი ფუნქცია `isPositive`, 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 λ-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
```

ახლა მათი გამოყენება შეიძლება არგუმენტებად მაღალი 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 λ-აბსტრაქცია განისაზღვროს რამდენიმე ცვლადისთვის:

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

λ-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
```

λ-აბსტრაქციის გამოყენებით შესაძლოა განისაზღვროს ფუნქცია. მაგალითად, ჩანაწერი

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

ეკვივალენტურია

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

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 გამოყენებულ იქნეს არგუმენტების ნაწილთან (ვინაიდან ბინარულ ოპერატორებს ორი არგუმენტი აქვთ, ეს ნაწილი შედგება ერთი არგუმენტისგან). ბინარულ ოპერაციას, 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)
```

დავალებები

1. განსაზღვრეთ ფუნქცია მაღალი რიგის ფუნქციების გამოყენებით:

   a. ფუნქცია, რომელიც ითვლის ნამდვილი რიცხვების სიის ელემენტების საშუალო არითმეტიკულს ფუნქცია `foldr`-ის გამოყენებით. ფუნქციამ მხოლოდ ერთხელ უნდა გადახედოს სიას.

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

   c. ფუნქცია `countEven`, რომელიც აბრუნებს სიის ლუწ ელემენტებს.

   d. ფუნქცია `quicksort`, რომელიც ახდენს სიის სწრაფ დახარისხებას რეკურსიული ალგორითმით. იმისათვის, რომ დახარისხდეს სია `xs`, მისგან აიღება პირველი ელემენტი (აღვნიშნოთ 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.

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

თავი 1.7. 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 ჩვენ მიერ უკვე არის განსაზღვრული.

მოდული ცხადად *ექსპორტირებს* Tree, Leaf, Branch და leafList. მოდულიდან ექსპორტირებული სახელები მიეთითება გასაღები სიტყვა module-ის შემდეგ ფრჩხილებში. იმ შემთხვევაში, თუ სახელები მითითებული არ არის, მაშინ 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.

ერთი მოდული შეიძლება სხვა მოდულში იყოს *იმპორტირებული*. მაგალითად, მოდული `Tree` იმპორტირებულია `Main` 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 ერთი და იმავე 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
```

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 ლექსიკონს.

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

აქ მომხმარებელი ლექსიკონს ხედავს როგორც სიას და ამით სარგებლობს. შემდგომ შეიძლება მოგვინდეს ლექსიკონის შეცვლა. იმ შემთხვევაში, თუ სია დიდია, მასში ძებნის განხორციელება რთულია. გაცილებით უკეთესია ჰეშ-ცხრილების ან ძებნის ხეების გამოყენება. თუმცა, თუ `Dictionary`-ის ტიპის წარმოდგენა არის ღია, ჩვენ რისკის გარეშე ვერ შევძლებთ შევცვალოთ მომხმარებლის პროგრამის ფუნქციონირება.

გავხადოთ ტიპი `Dictionary` აბსტრაქტული, რათა მოდულის მომხმარებლისგან დავმალოთ მისი შიდა წარმოდგენა. განვსაზღვროთ 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):d“
emptyDict = Dictionary []
```

მონაცემთა აბსტრაქტული ტიპი წარმოადგენს მონაცემთა დამალვის მექანიზმს, რომელსაც ობიექტორიენტირებული დაპროგრამების ენებში უწოდებენ ინკაფსულაციას.

შეტანა-გამოტანის ოპერაციები

შეტანა-გამოტანის სისტემა `Haskell`-ში სრულად ფუნქციონირებს და არ ჩამოუვარდება შეტანა-გამოტანის სისტემას იმპერატიულ დაპროგრამებაში. იმპერატიულ ენებში პროგრამა წარმოადგენს *მოქმედებების* თანმიმდევრობას. ტიპური 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 დაპროგრამებისთვის 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 მარტივი, რომელიც განისაზღვრება სისტემური პრიმიტივების საშუალებით, ანდა სხვა მოქმედებების თანმიმდევრული კომპოზიცია. შეტანა-გამოტანის მონადა შეიცავს მარტივ მოქმედებებს, რომელთა გამოყენებით იქმნება შედგენილი მოქმედება, ანალოგიურად, იმპერატიულ ენებში ‘;’ -ის გამოყენებისას. მონადა, როგორც „წებო“, უკავშირებს მოქმედებებს პროგრამაში.

თოთოეული 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 `C`-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 კითხულობს სიმბოლოს და ბეჭდავს მას:

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

სახელი `main` აქ შემთხვევით არაა: ფუნქცია `main` მოდული `Main`-დან წარმოადგენს პროგრამის საწყის წერტილს `Haskell` ენაზე, მსგავსად ფუნქცია `main`-ისა `C`-ში. მისი ტიპი უნდა იყოს `IO ()`. წარმოდგენილი პროგრამა ასრულებს თანმიმდევრულად ორ მოქმედებას: კითხულობს სიმბოლოს, რომელიც ცვლადი `c`-ს მნიშვნელობა ხდება და შემდეგ ბეჭდავს ამ 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' -- შეცდომა!!!
```

შეცდომა გამოიწვია იმან, რომ `do`-ში მეორე ოპერატორი არის ლოგიკური მნიშვნელობა და არა მოქმედება. ჩვენ უნდა ავიღოთ ეს ლოგიკური მნიშვნელობა და შევქმნათ მოქმედება, რომელიც აბრუნებს ამ ბულის მნიშვნელობას, როგორც შედეგს. ამას ემსახურება ფუნქცია `return`:

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

ფუნქცია `return` ამთავრებს მოქმედებების თანმიმდევრობას. ამრიგად, `ready` განისაზღვრება ასე:

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

ახლა 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 შეჰყავს ჩვეულებრივი მნიშვნელობა შეტანა-გამოტანის მოქმედებაში. ისმის კითხვა, შეიძლება თუ არა, პირიქით, შეტანა-გამოტანის ოპერაცია შესრულდეს გამოსახულებაში? აღმოჩნდა, რომ არა. ისეთ ფუნქციას, როგორიცაა `f :: Int -> Int` არ შეუძლია შეასრულოს შეტანა-გამოტანის ოპერაცია, ვინაიდან `IO` არ არის დასაბრუნებელი მნიშვნელობის ტიპი.

ნაწარმოები პრიმიტივები

სტრიქონის წაკითხვა კლავიატურიდან:

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

სტრიქონის ჩაწერა ეკრანზე:

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

სტრიქონის ჩაწერა და ახალზე გადასვლა:

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

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

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

მაგალითად,

```
Prelude> strlen
Enter a string: abcde
The string has 5 characters
```

შევნიშნოთ, რომ რაიმე მოქმედების შეფასება ახორციელებს თავის თანამდევ ეფექტებს საბოლოო შედეგის მნიშვნელობის გამოტანასთან ერთად.

თამაში „ჯალათის“ რეალიზება

განვიხილოთ თამაში „ჯალათის“ (hangman) შემდეგი ვერსია:

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

ჩვენ ვირჩევთ დაღმავალ მიდგომას თამაში „ჯალათის“ სარეალიზაციოდ Haskell-ზე და ვიწყებთ ასეთი ფრაგმენტით (აქ ინგლისურად ნახმარია ორი ფრაზა – „ჩაიფიქრეთ სიტყვა“ და „შეეცადეთ გამოიცნოთ იგი“) :

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

sgetLine მოქმედება კითხულობს ტექსტის სტრიქონს კლავიატურიდან და ასახავს ეკრანზე ყოველ სიმბოლოს, როგორც ტირეს:

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

`getCh` მოქმედება კითხულობს სიმბოლოს კლავიატურიდან, მაგრამ არ ასახავს მას ეკრანზე. ეს სასარგებლო მოქმედება არ არის სტანდარტული ბიბლიოთეკის ნაწილი, მაგრამ არის `Hugs` სისტემის სპეციალური პრიმიტივი, რომელიც შეიძლება იქნეს იმპორტირებული სკრიპტში შემდეგნაირად:

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

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

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

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

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

მაგალითად:

```
Prelude > diff "haskell" "pascal"
"-as--ll"
```

სავარჯიშო

1. nim თამაშის განხორციელება Haskel-ზე, როცა ამ თამაშის წესები ასეთია:

- დაფა შეიცავს ფიფქების ხუთ სტრიქონს:

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

- ორი მოთამაშე რიგრიგობით ეხება და ანადგურებს ერთ ან რამდენიმე ფიფქს ერთადერთი სტრიქონის ბოლოდან.
- მოგებული რჩება ის მოთამაშე, რომელიც დაფიდან აიღებს უკანასკნელ ფიფქს ან ფიფქებს.

დახმარება :

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

შეტანა-გამოტანის სტანდარტული ოპერაციები

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

```
type FilePath = String -- ფაილების სახელი ფაილურ სისტემაში
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 ფუნქცია `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 ()
```

ამის გარდა, ძალზე სასარგებლოა შემდეგი ფუნქციები:

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

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

მაგალითი

დავწეროთ ფაილების ასლის გადამღები (კოპირების) პროგრამა. ის კითხულობს კლავიატურიდან ორი ფაილის სახელს (საწყისი და მიზნობრივი ფაილების) და ერთი ფაილის ასლი გადააქვს მეორეში.

a. ფუნქცია ბეჭდავს მოწვევას, კითხულობს ფაილის სახელს
b. 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`-ს, 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 ფუნქციას, რომელიც განსაზღვრულია მოდულში `System`:

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

ეს ფუნქცია აბრუნებს სტრიქონების სიას, რომელიც წარმოადგენს ბრძანების სტრიქონის პარამეტრებს, მსგავსად მასივისა `argv` C-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.

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 გამოვიყენოთ 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 --make Main.hs
```

იმ შემთხვევაში, თუ პროგრამა შეიცავს შეცდომებს, ინფორმაცია მათ შესახებ გამოვა ეკრანზე. თუ შეცდომები არაა, კომპილერი შექმნის შესასრულებელ ფაილს, რომელიც უნდა გავუშვათ შესასრულებლად.

დავალებები

1. დაწერეთ შემდეგი პროგრამები:
   a. პროგრამა კითხულობს ორ რიცხვს და აბრუნებს მათ ჯამს.
   b. პროგრამა ბეჭდავს მასზე გადაცემულ ბრძანების სტრიქონის არგუმენტებს.

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

d. პროგრამა, რომელიც იღებს ბრძანებათა სტრიქონში რიცხვ `n`-ს და ფაილის სახელს და ეკრანზე გამოაქვს ფაილის პირველი `n` სტრიქონი. გამოიყენეთ ფუნქცია `lines`, რომელიც არგუმენტს ყოფს სტრიქონების სიად სიმბოლო `\n`-ის მითითებით. მაგალითად, `lines "line1\nline2"` დააბრუნებს `["line1", "line2"]`. ასევე სასარგებლოა ფუნქცია `unlines`, რომელიც ასრულებს ოპერაციას პირიქით.

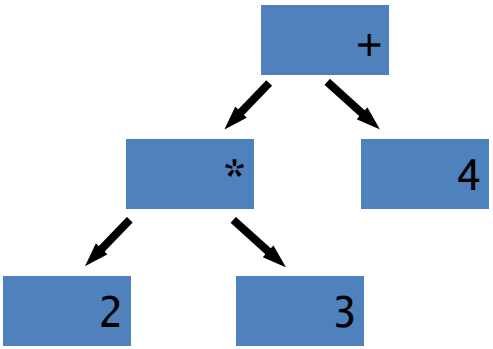
თავი 1.8. ფუნქციონალური პარსერები

სინტაქსური ანალიზატორები

პარსერი არის პროგრამა, რომელიც აანალიზებს ტექსტის ფრაგმენტს მისი სინტაქსური სტრუქტურის დასადგენად.

მაგალითად,

`2*3+4` ნიშნავს:



თითქმის ყოველი რეალურად არსებული პროგრამა იყენებს ამა თუ იმ ფორმით პარსერს თავისი შეტანის წინასწარი ანალიზისათვის. მაგალითად, `Hugs` სისტემა აანალიზებს `Haskell` პროგრამებს, `Unix` სისტემა - `Shell` სკრიპტებს, ხოლო `Explorer` - `HTML` დოკუმენტებს.

`Haskell`-ში პარსერები შეიძლება განვიხილოთ როგორც ფუნქციები, რომლებსაც შეუძლიათ არგუმენტად მიიღონ სტრიქონი და დააბრუნონ ამა თუ იმ ფორმის ხე:

```
type Parser = String → Tree
```

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

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

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

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

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

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

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

განვიხილოთ ძირითადი პარსერები.

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

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

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

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

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

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

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

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

`parse` ფუნქცია გამოიყენება სტრიქონის გასაანალიზებლად:

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

მაგალითები:

ვაჩვენოთ რამდენიმე მარტივ მაგალითზე, თუ როგორ ფუნქციონირებს სინტაქსური ანალიზის (გარჩევის) ხუთი პრიმიტივი:

```
% hugs Parsing
Prelude> parse item ""
[]
Prelude> parse item "abc"
[('a',"bc")]
Prelude> parse failure "abc"
[]
Prelude> parse (return 1) "abc"
[(1,"abc")]
Prelude> parse (item +++ return 'd') "abc"
[('a',"bc")]
Prelude> parse (failure +++ return 'd') "abc"
[('d',"abc")]
```

შევნიშნოთ, რომ `Parsing` საბიბლიოთეკო ფაილი მისაწვდომია ინტერნეტში `Haskell`-ზე დაპროგრამების საკუთარ გვერდზე. ამასთან, ტექნიკური მიზეზების გამო, წარუმატებლობის გამომვლენი პირველი მაგალითი ფაქტობრივად იძლევა შეცდომას, რომელიც ეხება ტიპებს, მაგრამ ეს არ ხდება არატრივიალურ მაგალითებში. გვინდა კიდევ ერთხელ გავუსვათ ხაზი, რომ `Parser` ტიპი არის მონადა – მათემატიკური სტრუქტურა, რომელსაც დანამდვილებით მოაქვს სარგებლობა სხვადასხვა სახის გამოთვლათა მოდელირებისას.

პარსერების მიმდევრობა შეიძლება გაერთიანდეს ერთ შედგენილ პარსერად `do` გასაღები სიტყვის გამოყენებით. მაგალითად:

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

შევნიშნოთ, რომ აუცილებელია ყოველი პარსერი დაიწყოს ერთსა და იმავე სვეტში. ეს არის ტოპოლოგიური წესის მოთხოვნა. ამასთან, შუალედური პარსერების მიერ დაბრუნებული მნიშვნელობები გადაგდებული აღმოჩნდება გაუცხადებლად (ანუ დაიკარგება), მაგრამ, საჭიროების შემთხვევაში, მათ მიენიჭებათ სახელები ← ოპერატორის გამოყენებით. უკანასკნელი პარსერის მიერ მოცემული მნიშვნელობა არის ის მნიშვნელობა, რომელიც დაბრუნებულია პარსერთა მიმდევრობით, როგორც ერთი მთლიანობა.

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

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

`do` ნოტაცია არ არის სპეციფიკური `Parser` ტიპისათვის, იგი შეიძლება გამოვიყენოთ ნებისმიერ მონადურ ტიპთან.

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

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

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

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

პარსერის გამოყენება ან არგამოყენება:

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

პარსერის გამოყენება თუნდაც ერთხელ ან მეტჯერ:

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

სიმბოლოთა სპეციფიკური სტრიქონის სინტაქსური ანალიზი:

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

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

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

მაგალითად,

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

რა თქმა უნდა, სინტაქსური ანალიზის უფრო რთულ ბიბლიოთეკებს შეუძლია შემავალი სტრიქონის შეცდომათა მითითება და-/ან აცილება.

განვიხილოთ გამოსახულებათა მარტივი ფორმა, რომელიც აგებულია (+) შეკრებისა და (*) გამრავლების ოპერაციათა გამოყენებით მრგვალ ფრჩხილებთან ერთად. გავითვალისწინოთ, რომ: * და + ასოციატიურია მარჯვნივ და * უფრო მაღალი პრიორიტეტისაა, ვიდრე +.

ფორმალურად, ასეთი გამოსახულებების სინტაქსი განსაზღვრულია თავისუფალი გრამატიკის შემდეგი კონტექსტით:

```
expr   → term '+' expr | term
term   → factor '*' term | factor
factor → digit | '(' expr ')'
digit  → '0' | '1' | ... | '9'
```

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

```
expr → term ('+' expr | ε)
term → factor ('*' term | ε)
```

შევნიშნოთ, რომ სიმბოლო ε აღნიშნავს ცარიელ სტრიქონს.

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

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

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

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

საბოლოოდ, თუ განვსაზღვრავთ, რომ:

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

მაშინ პრაქტიკულად შევამოწმებთ ზოგიერთ მაგალითსაც:

```
Prelude> eval "2*3+4"
10
Prelude> eval "2*(3+4)"
14
```

სავარჯიშოები

1. რატომ ახდენს არითმეტიკული გამოსახულებებისათვის განკუთვნილი გრამატიკის საბოლოო გამარტივება არსებით გავლენას ამის შედეგად მიღებული პარსერის ეფექტურობაზე?
2. გააფართოეთ პარსერი არითმეტიკული გამოსახულებებისათვის გამოკლებისა და გაყოფის მხარდასაჭერად, რისთვისაც უნდა გამოიყენოთ გრამატიკის შემდეგი ჩაწერის ფორმები:

```
expr → term ('+' expr | '-' expr | ε)
term → factor ('*' term | '/' term | ε)
```

თავი 1.9. ზარმაცი, იგივე გადადებული გამოთვლები

Haskell-ში გამოსახულებათა შეფასება მარტივი მეთოდით ხდება, რომლის მთავარი პრინციპები ასეთია:

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

გამოთვლათა ასეთ მეთოდს გადადებული, იგივე ზარმაცი გამოთვლები ეწოდება, ხოლო თავად Haskell-ს – ზარმაცი ფუნქციონალური ენა.

ძირითადად, გამოსახულებები გამოითვლება ან გარდაიქმნება განსაზღვრებათა გამოყენებით, ვიდრე შემდგომი გამარტივება შეუძლებელი არ გახდება.

მაგალითად, თუ გავითვალისწინებთ

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

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

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

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

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

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

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

ვისაუბროთ გარდაქმნათა სტრატეგიებზე. რედექსის ანუ რედუცირებადი ქვეგამოსახულების (REDu-cible subEXpression) ასარჩევად ორი ზოგადი სტრატეგია არსებობს.

1. შიდა რედუქცია. შიდა რედექსი ყოველთვის რედუცირებადია;
2. გარე რედუქცია. გარე რედექსი ყოველთვის რედუცირებადია.

ისმის შეკითხვა: როგორ ხდება ორი სტრატეგიის შედარება?

დასრულებადობა

```
loop  = tail loop
```

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

1. შიდა რედუქცია

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

ეს სტრატეგია დასრულებადი არ არის

2. გარე რედუქცია

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

ეს სტრატეგია იძლევა შედეგს ერთი ბიჯით.

ადგილი აქვს შემდეგ ფაქტებს:

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

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

გამოვთვალოთ რედუქციათა რიცხვი. კვლავ განვიხილოთ შემდეგი რედუქციები:

შიდა	გარე
square (3+4)	square (3+4)
= Square 7	=(3+4) * (3+4)
= 7 * 7	= 7*(3+4)
= 49	= 7*7
	=49

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

აქედან შეიძლება დავასკვნათ, რომ გარე რედუქციამ შეიძლება მოითხოვოს უფრო მეტი მოქმედება, ვიდრე შიდამ.

ეს პრობლემა შეიძლება გადაიჭრას მაჩვენებლებით გამოსახულებათა ერთობლივი გამოყენების საჩვენებლად გამოთვლაში.

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

ეს რედუქციის ახალ სტრატეგიას იძლევა:

ზარმაცი გამოთვლა = გარე რედუქცია + ერთობლივად გამოყენება

თუ დავაკვირდებით, შევამჩნევთ შემდეგ ფაქტებს:

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

უსასრულო სიები

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

განვიხილოთ რეკურსიული განსაზღვრება

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

რეკურსიის რამდენიმეჯერ განხორციელება იძლევა:

```
ones = 1:ones
     = 1:1:ones
     = 1:1:1:ones
     =   ...
```

ამრიგად მიიღება ერთიანების უსასრულო სია.

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

1. შიდა რედუქცია

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

ამ შემთხვევაში გამოთვლა დაუსრულებლად გრძელდება.

ზარმაცი გამოთვლა

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

ამ შემთხვევაში გამოთვლის შედეგია რიცხვი 1.

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

საზოგადოდ, არსებობს შემდეგი სლოგანი (დევიზი, ლოზუნგი):

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

ახლა ვხედავთ, რომ ones =1 : ones
ნამდვილად განსაზღვრავს პოტენციურად უსასრულო სიას, რომელიც ფასდება ზუსტად გამოყენების შინაარსის მოთხოვნიდან გამომდინარე.

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

```
Prelude> take 5 ones
[1,1,1,1,1]
Prelude> Take 5 [1..]
[1,2,3,4,5]
```

ზარმაცი გამოთვლები საშუალებას გვაძლევს მონაცემები გამოითვალის მართვის კომპონენტის მოთხოვნის შესაბამისად.

განვიხილოთ მაგალითი: მარტივი რიცხვების გენერირება. ეს პროცედურა ცნობილია „ერატოსფენის საცრის“ სახელწოდებით, ძველი ბერძენი მათემატიკოსის პატივსაცემად, რომელმაც პირველმა აღწერა ხსენებული Haskell-ალგორითმი. ეს ალგორითმი შეიძლება პირდაპირ ჩავწეროთ Haskell ენაზე:

```
primes         :: [Int]
primes         = seive [z..]
seive          :: [Int] ->[Int]
seive (p:xs) = p:seive [x|x<-xs, x`mod`p /=0]
```

მარტივი რიცხვების გამოთვლის ფუნქციის შესრულებას აქვს სახე:

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

ფუნქციის გამოძახებისას შეიძლება შემოვიტანოთ სხვადასხვა სასაზღვრო პირობა. მაგალითად, პირველი ათი მარტივი რიცხვის ან 15-ზე მცირე მარტივი რიცხვების არჩევა:

```
-- პირველი ათი მარტივი რიცხვის არჩევა
Prelude>    take  10  primes
[2,3,5,7,11,13,17,19,23,29]
-- მარტივი რიცხვების არჩევა, რომლებიც ნაკლებია 15-ზე
Prelude>  take  while ( <15 ) primes
[2,3,5,7,11,13]
```

ზარმაცი გამოთვლები მოხერხებული დაპროგრამებაა!

სავარჯიშოები

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

ა) პირველი ორი რიცხვია `0` და `1`;

ბ) შემდეგი რიცხვი წინა ორის ჯამია;

გ) გადასვლა ბ)-ზე.

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

ნაწილი 2. HASKELL ენის თეორია

ფუნქციონალური დაპროგრამების ისტორია

შევნიშნოთ, რომ იმპერატიულ დაპროგრამებას თეორიული საფუძველი ჩაეყარა ჯერ კიდევ XX საუკუნის 30-იან წლებში ალან ტიურინგისა (Alan Turing) და ჯონ ვონ ნეიმანის (John von Neumann) მიერ. თეორია, რომელიც ფუნქციონალური მიდგომის საფუძველია, ასევე დაიბადა 20-იან – 30-იან წლებში. მათ შორის, ვინც შეიმუშავა ფუნქციონალური დაპროგრამების მათემატიკური საფუძვლები, შეიძლება დავასახელოთ მოსეს შონფინკელი (Moses Schonfinkel) და ჰასკელ კარი (Haskell Curry), რომელმაც კომბინატორული ლოგიკა დაამუშავა, ასევე ალონზო ჩერჩი (Alonzo Church), რომელმაც შექმნა λ აღრიცხვა.

თეორია რჩებოდა თეორიად, სანამ წინა საუკუნის 50-იანი წლების დასაწყისში ჯონ მაკარტმა (John McCarthy) არ შეიმუშავა ენა Lisp, რომელიც გახდა პირველი ფუნქციონალური ენა და წლების განმავლობაში რჩებოდა ერთადერთად. ენა Lisp დღემდე გამოიყენება (მაგალითად, FORTRAN-ის მსგავსად), თუმცა ვეღარ აკმაყოფილებს ზოგიერთ თანამედროვე მოთხოვნას, რაც აიძულებს პროგრამების შემქმნელებს დიდი ძალისხმევა გადაიტანონ კომპილერზე. ამის აუცილებლობას იწვევს სულ უფრო მზარდი სირთულის პროგრამული უზრუნველყოფა.

ამ გარემოების გამო დიდ როლს თამაშობს ტიპიზაცია. XX საუკუნის 70-იანი წლების ბოლოსა და 80-იანი წლების დასაწყისში ინტენსიურად მუშავდებოდა ფუნქციონალური დაპროგრამების შე-

საბამისი ტიპიზაციის მოდელები. მათი უმრავლესობა შეიცავს ისეთ ძლიერ მექანიზმებს, როგორიცაა მონაცემთა აბსტრაქცია და პოლომორფიზმი. გაჩნდა მთელი რიგი ტიპიზებული ფუნქციონალური ენები: `ML`, `Scheme`, `Hope`, `Miranda`, `Clean` და ბევრი სხვა. დამატებით, მუდმივად იზრდებოდა დიალექტების რაოდენობაც. შეიქმნა სიტუაცია, როდესაც პრაქტიკულად ყველა ჯგუფი, რომელიც ფუნქციონალურ დაპროგრამებაში მუშაობდა, იყენებდა საკუთარ ენას, რამაც გააჩინა მთელი რიგი პრობლემები. ფუნქციონალური დაპროგრამების სფეროში წამყვანი მკვლევრების გაერთიანებულმა ჯგუფმა სიტუაციის გამოსწორების მიზნით გადაწყვიტა სხვადასხვა ენის ღირსებები გამოეყენებინა ახალი უნივერსალური ფუნქციონალური ენის შექმნისას. ასეთი ენის (რომელსაც დაერქვა `Haskell`, ჰასკელ კარის პატივსაცემად) პირველი რეალიზაცია განხორციელდა 90-იანი წლების დასაწყისში. ამჟამად ფუნქციონირებს სტანდარტი `Haskell-98`.

ფუნქციონალური ენების უმრავლესობისთვის, `Lisp`-ის ტრადიციებიდან გამომდინარე, რეალიზებულია ინტერპრეტატორი. ინტერპრეტატორები მოსახერხებელია პროგრამის სწრაფი გამართვისთვის. ამ დროს კომპილაციის გრძელი პროცესი გამოიტოვება, რითაც ჩქარდება დამუშავების ჩვეულებრივი ციკლი. თუმცა, მეორე მხრივ, ინტერპრეტატორები, კომპილერებთან შედარებით, რამდენჯერმე აგებენ კოდის შესრულების სიჩქარეში. ამიტომაც, ინტერპრეტატორის გვერდით არსებობს კომპილერები, რომლებიც გენერირებენ მანქანურ კოდს (მაგალითად, `Objective Caml`) ანდა კოდს (მაგალითად, `Glasgow Haskell Compiler`). ნიშანდობლივია ის, რომ პრაქტიკულად ყველა კომპილერი რეალიზებულია თვით ამ ენაზე.

ისტორიული მოვლენები, რომლებმაც გავლენა მოახდინეს ენა HASKELL-ის განვითარებაზე

Haskell-ის ბევრი მახასიათებელი არ არის ახალი - პირველად სხვა ენებში იქნა შემოთავაზებული. ქვემოთ განხილულია ის მოვლენები, რომლებიც აისახა Haskell- ის განვითარებაში:

- 1930-იან წლებში ალონზო ჩერჩის (Alonzo Church) მიერ იქნა დამუშავებული ფუნქციათა მარტივი, თუმცა მძლავრი მათემატიკური თეორია სახელწოდებით – ლამბდა აღრიცხვა (Lambda Calculus);
- პირველი ფუნქციონალური ენა Lisp („LISt Processor“-სიის დამუშავება)შეიქმნა 1950-იან წლებში ჯონ მაკარტნის (John McCarthy) მიერ. Lisp ენამ lambda-აღრიცხვის გარკვეული გავლენა განიცადა, მაგრამ დღემდე მნიშვნელობათა მინიჭება ცვლადებისათვის ენის ცენტრალურ ელემენტად ითვლება.
- 1960-იან წლებში პიტერ ლანდინმა (Peter Landin) შექმნა პირველი წმინდა ფუნქციონალური ენა სახელწოდებით ISWIM (ინგლისური ფრაზის აკრონიმია: „If you See What I Mean“ – „თუ ხედავთ რას ვგულისხმობ“), რომელიც მკაცრად ეფუძნებოდა lambda-აღრიცხვას და არ შეიცავდა მნიშვნელობათა მისანიჭებელ ცვლადებს.
- 1970-იან წლებში ჯონ ბეკუსმა (John Backus) დაამუშავა ფუნქციონალური დაპროგრამების FP („Functional Programming“) ენა, რომელიც განსაკუთრებით უსვამდა ხაზს მაღალი რიგის ფუნქციათა იდეას და ეკვაციონურ დასკვნებს პროგრამებზე.
- 1970-იან წლებში რობინ მილნერმა (Robin Milner) კოლეგებთან ერთად შექმნა ფუნქციონალური დაპროგრამების პირველი თანამედროვე ML („Meta-Language“) ენა, რომელ-

მაც წამოაყენა პოლიმორფული ტიპებისა და ტიპის გამოყვანის იდეა.

- 1970-იან – 1980-იან წლებში დევიდ ტერნერმა (David Turner) დაამუშავა ფუნქციონალური დაპროგრამების რიგი ზარმაცი ენა, რაც დასრულდა კომერციულად ნაწარმოები ენით Miranda (ინგლ. „admirable“ – შესანიშნავი, ჩინებული, საუცხოო).
- 1987 წელს მკვლევართა საერთაშორისო კომიტეტი შეუდგა Haskell-ის – ფუნქციონალური დაპროგრამების სტანდარტული ზარმაცი ენის – დამუშავებას, რომელსაც ეს სახელი ამერიკელი ლოგიკოსისა და მათემატიკოსის, ჰასკელ კარის (Haskell Curry, 1900-1982) პატივისცემის ნიშნად დაერქვა.
- 2003 წელს კომიტეტმა გამოაქვეყნა მოხსენება ჰასკელის შესახებ (Haskell Report), რომელიც განსაზღვრავს ამ ენის სტაბილურ ვერსიას და წარმოადგენს მისი შემქმნელების თხუთმეტწლიანი დაუღალავი შრომის ნაყოფს და კულმინაციას.

თავი 2.1. ფუნქციონალური ენების თვისებები

შეიძლება მოკლედ ჩამოვთვალოთ ფუნქციონალური ენების ძირითადი თვისებები:

- ✓ მოკლე და მარტივი კოდი;
- ✓ მკაცრი ტიპიზაცია;
- ✓ მოდულირება;
- ✓ ფუნქცია – ეს მნიშვნელობაა;
- ✓ სისუფთავე (გვერდითი ეფექტების არარსებობა);
- ✓ გადატანილი (ზარმაცი) გამოთვლები.

მოკლე და მარტივი კოდი

პროგრამა ფუნქციონალურ ენაზე, საზოგადოდ, უფრო მოკლეა და მარტივი, ვიდრე იგივე პროგრამა იმპერატიულ ენაზე. შევადაროთ პროგრამები C-ზე და აბსტრაქტულ ფუნქციონალურ ენაზე სწრაფი დახარისხების ჰოარეს ალგორითმის (ავტორი: Tony Hoare) მაგალითზე. ეს მაგალითი გახდა კლასიკური ფუნქციონალური ენების უპირატესობების საჩვენებლად.

ჰოარეს სწრაფი დახარისხების პროგრამას ენა C/C++-ზე აქვს შემდეგი სახე:

```
void quickSort (int a[], int l, int r)
{
      int i = l;
      int j = r;
      int x = a[(l + r) / 2];
      do
      {
```

```
            while (a[i] < x) i++;
            while (x < a[j]) j--;
            if (i <= j)
            {
                  int temp = a[i];
                  a[i++] = a[j];
                  a[j--] = temp;
      }
      }
      while (i <= j);
      if (l < j) quickSort (a, l, j);
      if (i < r) quickSort (a, i, r);
}
```

შემდეგი პროგრამა არის ჰოარეს სწრაფი დახარისხების პროგრამა აბსტრაქტულ ფუნქციონალურ ენაზე:

```
quickSort ([]) = []
quickSort ([h : t]) = quickSort (n | n   t, n
<= h) + [h] + quickSort (n | n   t, n > h)
```

იგი შეიძლება წავიკითხოთ ასე:

1. თუ სია ცარიელია, მაშინ შედეგიც იქნება ცარიელი სია.
2. წინააღმდეგ შემთხვევაში (ანუ სია როცა არ არის ცარიელი) გამოიყოფა თავი (პირველი ელემენტი) და კუდი (დარჩენილი ელემენტების სია, რომელიც შეიძლება იყოს ცარიელი). ამ შემთხვევაში შედეგად ვღებულობთ კონკატენაციას კუდის ყველა ელემენტისა, რომელიც ნაკლებია ან ტოლი თავის სიასთან, რომელიც შედგება თავისა და კუდის ყველა ელემენტისგან, რომელიც მეტია თავზე.

ჰოარეს სწრაფი დახარისხების პროგრამა ენა Haskell-ზე ასე გამოიყურება:

```
quickSort [] = []
quickSort (h : t) = quickSort [y | y <- t, y <
h] ++ [h] ++ quickSort [y | y <- t, y >= h]
```

ამ მარტივ მაგალითზეც ჩანს, თუ როგორ იგებს ფუნქციონალური დაპროგრამების სტილი როგორც კოდის რაოდენობაში, ასევე მის ელეგანტურობაში.

ამის გარდა, ყველა ოპერაცია მეხსიერებასთან სრულდება ავტომატურად. ნებისმიერი ობიექტის შექმნისას მას ავტომატურად გამოეყოფა მეხსიერება. მას შემდეგ, რაც ობიექტი თავის დანიშნულებას შეასრულებს, ის ავტომატურადვე განადგურდება დამლაგებლის მიერ, რომელიც არის ნებისმიერი ფუნქციონალური ენის ნაწილი.

კიდევ ერთი სასარგებლო თვისება, რომელიც იძლევა პროგრამის შემცირების საშუალებას, არის ნიმუშთან შედარების მექანიზმი. მისი საშუალებით აღიწერება ფუნქცია, როგორც ინდუციური განსაზღვრება. მაგალითად, ფიბონაჩის `N`-ური რიცხვის განსაზღვრის ფუნქციას აქვს შემდეგი სახე:

```
fibb (0) = 1
fibb (1) = 1
fibb (N) = fibb (N - 2) + fibb (N - 1)
```

როგორც ჩანს, ფუნქციონალური ენები ადის უფრო მაღალ აბსტრაქტულ დონეზე, ვიდრე ტრადიციული იმპერატიული ენები. ნიმუშთან შედარების მექანიზმს განვიხილავთ შემდგომ.

პრაქტიკულად ყველა თანამედროვე დაპროგრამების ენა წარმოადგენს ტიპიზებულ ენას (შესაძლებელია `JavaScript`-ისა და მისი დიალექტების გამოკლებით. არ არსებობს იმპერატიული ენა, რომელშიც არ იყოს ცნება „ტიპი“). ფუნქციონალურ ენებს ახასიათებს მკაცრი ტიპიზაცია, რომელიც უზრუნველყოფს უსაფრთხოებას. პროგრამა, რომელიც წინასწარ ამოწმებს ტიპებს, არ შეწყდება ოპერაციული სისტემის შეტყობინებით `"access violation"`. ეს განსაკუთრებით ეხება ისეთ ენებს, როგორიცაა `C/C++` და `Object Pascal`, სადაც მიმთითებლების გამოყენება ხშირად ხდება. ფუნქციონალურ ენებში შეცდომების დიდი ნაწილის გასწორება ხდება ინტერპრეტაციის ეტაპზე, ამიტომ გამართვის სტადია და პროგრამის დამუშავების მთლიანი დრო მცირდება. და კიდევ, მკაცრი ტიპიზაცია კომპილერს აძლევს უფრო ეფექტური კოდის გენერირების საშუალებას და ამით აჩქარებს პროგრამის შესრულების დროს.

თუ განვიხილავთ ჰოარეს სწრაფი დახარისხების მაგალითს, შეიძლება დავინახოთ, რომ უკვე ნახსენები განსხვავებების გარდა `C` ენაზე ვარიანტსა და აბსტრაქტულ ფუნქციონალურ ენაზე ვარიანტს შორის, არის კიდევ ერთი ძირითადი განსხვავება: ფუნქცია `C`-ზე ახდენს `int` ტიპის (მთელი რიცხვების) დახარისხებას, ხოლო აბსტრაქტულ ფუნქციონალურ ენაზე – ნებისმიერი ტიპის მნიშვნელობების სიის, რომელიც ეკუთვნის დალაგებული სიდიდეების კლასს. ამიტომ, ბოლო ფუნქციას შეუძლია დაახარისხოს მთელი რიცხვების სია, ასევე ნამდვილი რიცხვების სია და სტრიქონების სია. შეიძლება აღვწეროთ ახალი ტიპი. მისთვის განვსაზღვროთ შედარების ოპერაცია და შემდეგ გამოვიყენოთ ხელახალი კომპილაციის გარეშე `quickSort` ამ ახალი ტიპის მნიშვნელობების სიის-

თვისაც. ამ სასარგებლო თვისებას ეწოდება პარამეტრული ანუ ჭეშმარიტი პოლიმორფიზმი და მას მხარს უჭერს ფუნქციონალური ენების უმრავლესობა.

პოლიმორფიზმის კიდევ ერთი სახეობაა ფუნქციების გადატვირთვა, რომელიც სხვადასხვა ფუნქციას, მაგრამ რაღაცით მსგავსს, აძლევს ერთსა და იმავე სახელებს. გადატვირთული ოპერაციის ტიპური მაგალითია შეკრების ოპერაცია. მთელი რიცხვებისა და ნამდვილი რიცხვების შეკრების ფუნქციები სხვადასხვაა, მაგრამ მოხერხებულობისთვის ისინი ატარებენ ერთსა და იმავე სახელს. ზოგიერთი ფუნქციონალური ენა, პოლიმორფიზმის გარდა, მხარს უჭერს ოპერაციების გადატვირთვასაც.

ენა C++ არის ისეთი ცნება, როგორიცაა შაბლონი, რომელიც იძლევა საშუალებას განისაზღვროს პოლიმორფული ფუნქციები, მსგავსი quickSort-ის. C++-ის სტანდარტულ ბიბლიოთეკაში STL შედის ასეთი და კიდევ მრავალი სხვა პოლიმორფული ფუნქცია. მაგრამ C++-ის შაბლონები და Ada-ს გვაროვნული ფუნქციები ბადებს გადატვირთული ფუნქციების სიმრავლეს, რომელსაც კომპილერი ყოველ ჯერზე აკომპილირებს, რაც უარყოფითად მოქმედებს კომპილაციის დროსა და კოდის ზომაზე. ხოლო ფუნქციონალურ ენებში პოლიმორფული ფუნქცია quickSort-ის ერთი, ერთადერთი ფუნქციაა.

ზოგიერთი ენაში, მაგალითად, Ada-ში მკაცრი ტიპიზაცია პროგრამისტისგან ითხოვს ცხადად აღწეროს ყველა მნიშვნელობის და ფუნქციის ტიპი. ამის თავიდან ასაცილებლად, მკაცრად ტიპიზებულ ფუნქციონალურ ენებში ჩადგმულია სპეციალური მექანიზმი, რომელიც კომპილერს საშუალებას აძლევს განსაზღვროს კონსტანტის, გამოსახულების და ფუნქციის ტიპი კონტექსტიდან გამომდინარე. ამ მექანიზმს უწოდებენ ტიპების გამოყვანის მექანიზმს. ცნობილია რამდენიმე ასეთი მექანიზმი, თუმცა მათი უმრავლესობა წარმოადგენს სახესხვაობებს ჰინდლი-მილნერის ტიპი-

ზაციის მოდელისა, რომელიც XX საუკუნის 80-იანი წლების დასაწყისში დამუშავდა. ამრიგად, უმრავლეს შემთხვევებში შეიძლება არ მივუთითოთ ფუნქციის ტიპი.

მოდულირება

მოდულირების მექანიზმი საშუალებას იძლევა პროგრამა დავყოთ რამდენიმე დამოუკიდებელ ნაწილად (მოდულად) მათ შორის მკვეთრად განსაზღვრული კავშირებით. ამით მარტივდება დიდი პროგრამული სისტემების პროექტირებისა და შემდგომი მხარდაჭერის პროცესები. მოდულირების მხარდაჭერა არ წარმოადგენს კონკრეტულად ფუნქციონალური ენების თვისებას, თუმცა მას მხარს უჭერს ამ ენების უმრავლესობა. არსებობს ძალზე განვითარებული მოდულური იმპერატიული ენები. ასეთი ენებია, მაგალითად, Modula-2 და Ada-95.

ფუნქცია – ეს მნიშვნელობაა

ფუნქციონალურ ენებში (ისევე, როგორც, საზოგადოდ, დაპროგრამებასა და მათემატიკაში) ფუნქციები შეიძლება გადაეცეს სხვა ფუნქციებს არგუმენტად ან დაბრუნდეს როგორც შედეგი. ფუნქციებს, რომლებიც ფუნქციონალურ არგუმენტებს იღებს, უწოდებენ მაღალი რიგის ფუნქციებს ანუ ფუნქციონალებს. ყველაზე ცნობილი ფუნქციონალი არის ფუნქცია map. ეს ფუნქცია იყენებს მოცემულ ფუნქციას სიის ყველა ელემენტთან და აკომპლექტებს შედეგად სხვა სიას. მაგალითად, განვსაზღვროთ ფუნქცია, რომელსაც აჰყავს მთელი რიცხვი კვადრატში, ასე:

```
square (N) = N * N
```

შეიძლება გამოვიყენოთ ფუნქცია `map` ნებისმიერი სიის ყველა ელემენტის კვადრატში ასაყვანად:

```
squareList = map (square, [1, 2, 3, 4])
```

ამ ინსტრუქციის შესრულების შედეგი იქნება სია `[1, 4, 9, 16]`.

სისუფთავე

დაპროგრამებაში ცნება „სისუფთავე“ გამოიყენება გვერდითი ეფექტების არარსებობის მნიშვნელობით.

იმპერატიულ ენებში ფუნქციამ, მისი შესრულების პროცესში, შეიძლება წაიკითხოს ან შეცვალოს გლობალური ცვლადების მნიშვნელობები და შეასრულოს შეტანა-გამოტანის ოპერაციები. ამიტომ, თუ გამოვიძახებთ ერთსა და იმავე ფუნქციას ორჯერ, ერთი და იგივე არგუმენტებით, შეიძლება შედეგად გამოითვალოს ორი სხვადასხვა მნიშვნელობა. ასეთ ფუნქციას უწოდებენ ფუნქციას გვერდითი ეფექტებით.

ფუნქციის აღწერა გვერდითი ეფექტების გარეშე პრაქტიკულად შესაძლებელია ყველა ენაში, მაგრამ ზოგიერთი ენა მხარს უჭერს, ითხოვს გვერდით ეფექტებს. მაგალითად, მრავალ ობიექტორიენტირებულ ენაში კლასის წევრ ფუნქციას გადაეცემა ფარული არგუმენტი (ხშირად მას უწოდებენ `this` ან `self`), რომელსაც ეს ფუნქცია არაცხადად მოდიფიცირებს.

წმინდა ფუნქციონალურ ენაში მინიჭების ოპერატორი არ არსებობს. ობიექტები არ შეიძლება შეიცვალოს და განადგურდეს, შესაძლოა მხოლოდ ახალი შეიქმნას არსებულების დეკომპოზიციითა და სინთეზით. არასაჭირო ობიექტებზე ზრუნავს ენაში ჩადგმული

დამლაგებელი, რის გამოც წმინდა ფუნქციონალურ ენებში ყველა ფუნქცია თავისუფალია გვერდითი ეფექტებისგან. შესაძლოა ფუნქციონალურ ენაში არსებობდეს იმპერატიული ენებისთვის დამახასიათებელი ზოგიერთი თვისება, როგორიცაა, მაგალითად, გამონაკლისი სიტუაციები და მასივები.

ისმის კითხვა: რა უპირატესობა აქვთ წმინდა ფუნქციონალურ ენებს? გარდა პროგრამების გამარტივებული ანალიზისა, არსებობს კიდევ ერთი ძლიერი უპირატესობა – პარალელიზმი. ვინაიდან ფუნქცია გამოთვლისას იყენებს მხოლოდ თავის პარამეტრებს, ჩვენ შეგვიძლია გამოვთვალოთ დამოუკიდებელი ფუნქციები ნებისმიერი რიგით ან პარალელურად, ეს შედეგზე ასახვას ვერ პოვებს. ამასთან, პარალელიზმი შეიძლება განხორციელდეს არა მხოლოდ ენის კომპილატორის დონეზე, არამედ არქიტექტურის დონეზეც. ზოგიერთ ლაბორატორიაში უკვე შემუშავებულია და გამოიყენება ექსპერიმენტული კომპიუტერები, რომლებიც მსგავს არქიტექტურას ეყრდნობა. მაგალითისთვის შეიძლება დავასახელოთ `Lisp`-მანქანა.

გადატანილი გამოთვლები

დაპროგრამებაში ცნებები „გადატანილი გამოთვლები“ და „ზარმაცი (`lazy`) გამოთვლები“ ერთი და იგივეა.

ტრადიციულ დაპროგრამების ენებში (მაგალითად, `C++`-ში) ფუნქციის გამოძახება იწვევს ყველა არგუმენტის გამოთვლას. ფუნქციის გამოძახების ამ მეთოდს უწოდებენ *მნიშვნელობით გამოძახებას.* თუ ფუნქციაში რომელიღაც არგუმენტი არ გამოიყენება, მაშინ გამოთვლის შედეგი იკარგება. აქედან გამომდინარე, გამოთვლები ამაოდ ჩატარდა. რაღაც აზრით, მნიშვნელობით გამოძახების საწინააღმდეგოა გამოძახება საჭიროების მიხედვით. ამ შემთხვევაში არგუმენტი გამოიძახება, თუ საჭიროა შედეგის გამოთ-

ვლისათვის. ასეთი გამოთვლების მაგალითად შეიძლება დავასახელოთ კონიუნქციის ოპერატორი (&&) C++-დან, რომელიც არ ითვლის მეორე არგუმენტს, თუ პირველ არგუმენტს აქვს მცდარი მნიშვნელობა.

თუ ფუნქციონალური ენა მხარს არ უჭერს გადატანილ გამოთვლებს, მას უწოდებენ მკაცრ ენას. მართლაც, ასეთ ენებში გამოთვლების რიგი მკაცრად არის განსაზღვრული. მკაცრი ენების მაგალითად შეიძლება დავასახელოთ Scheme, Standard ML და Caml.

ენებს, რომლებიც იყენებენ გადატანილ გამოთვლებს, უწოდებენ არამკაცრს. Haskell — არამკაცრი ენაა, ისევე, როგორც Gofer და Miranda. არამკაცრი ენები ამასთანავე სუფთაცაა.

ძალზე ხშირად მკაცრი ენები შეიცავს ზოგიერთი ისეთი შესაძლებლობის მხარდაჭერას, რაც ახასიათებს არამკაცრ ენებს, მაგალითად, უსასრულო სიებს. Standard ML-ში სპეციალური მოდულია, რომელიც გადატანილ გამოთვლებს უჭერს მხარს. ხოლო Objective Caml, ამის გარდა, შეიცავს რეზერვირებულ სიტყვას lazy და კონსტრუქციას მნიშვნელობათა სიისთვის, რომელიც გამოითვლება აუცილებლობის მიხედვით.

ამოსახსნელი ამოცანები

ფუნქციონალური დაპროგრამების კურსებში, ტრადიციულად განხილული ამოცანებიდან, შეიძლება გამოვყოთ შემდეგი:

1. დარჩენილი პროცედურის მიღება

   თუ მოცემულია შემდეგი ობიექტები:

   P (x_1, x_2, ..., x_n) — რაღაც პროცედურა.

   $x_1 = a_1$, $x_2 = a_2$ — პარამეტრების ცნობილი მნიშვნელობები.

x_3, ..., x_n — პარამეტრების უცნობი მნიშვნელობები. მოითხოვება დარჩენილი პროცედურის მიღება P1 (x_3, ..., x_n). ეს ამოცანა იხსნება პროგრამების მხოლოდ ვიწრო კლასისთვის.

2. ფუნქციის მათემატიკური აღწერის მიღება
   ვთქვათ, გვაქვს P პროგრამა. მისთვის განსაზღვრულია შესასვლელი მნიშვნელობები და გამოსასვლელი მნიშვნელობები. მოითხოვება აიგოს ფუნქციის მათემატიკური აღწერა f : Dx_1, ..., Dx_n -> Dy_1, ..., Dy_m.
3. დაპროგრამების ენის სემანტიკის ფორმალური აღწერა.
4. მონაცემთა დინამიკური სტრუქტურების აღწერა.
5. პროგრამის „მნიშვნელოვანი“ ნაწილის აგება მონაცემთა სტრუქტურის აღწერით, რომლებსაც ამუშავებს ასაგები პროგრამა.
6. პროგრამის ზოგიერთი თვისების არსებობის დამტკიცება.
7. პროგრამების ეკვივალენტური ტრანსფორმაცია.

ყველა ეს ამოცანა საკმაოდ მარტივად იხსნება ფუნქციონალური დაპროგრამების საშუალებებით, მაგრამ მათი გადაწყვეტა იმპერატიულ ენებზე პრაქტიკულად შეუძლებელია.

თავი 2.2. მონაცემთა სტრუქტურები და ბაზური ოპერაციები

როგორც უკვე აღვნიშნეთ, დაპროგრამების ფუნქციონალური პარადიგმის საფუძველს წარმოადგენს მათემატიკური აზროვნების განვითარების ისეთი მიმართულებები, როგორიცაა კომბინატორული ლოგიკა და λ-აღრიცხვა. ეს უკანასკნელი უფრო მჭიდროდ არის დაკავშირებული ფუნქციონალურ დაპროგრამებასთან. სწორედ λ-აღრიცხვაა ფუნქციონალური დაპროგრამების თეორიული საფუძველი.

იმისათვის, რომ განვიხილოთ ფუნქციონალური დაპროგრამების თეორიული საფუძვლები, პირველ რიგში აუცილებელია შემოვიტანოთ ზოგიერთი შეთანხმება, შემოვიღოთ აღნიშვნები და ავაგოთ ფორმალური სისტემა.

ვთქვათ, მოცემულია რომელიღაც `A` პირველადი ტიპის ობიექტები. ამჟამად არ აქვს მნიშვნელობა, თუ კონკრეტულად რას წარმოადგენს გამოყოფილი ობიექტები. საზოგადოდ, ითვლება, რომ ამ ობიექტებზე განისაზღვრება ბაზისური ოპერაციების და პრედიკატების ერთობლიობა. ტრადიციულად, ობიექტებს უწოდებენ ატომებს. ეს მოდის მაკარტიდან (`Lisp`-ის ავტორისგან). თეორიულად, არანაირი მნიშვნელობა არ აქვს ბაზისური ოპერაციებისა და პრედიკატების რეალიზების საშუალებებს, უბრალოდ ხდება მათი ჩამოთვლა, თუმცა ყოველი ფუნქციონალური ენა თავისებურად ახდენს ბაზისური ოპერაციებისა და პრედიკატების რეალიზებას.

ტრადიციულად და, ამავე დროს, თეორიული აუცილებლობიდან გამომდინარე, ბაზისურ ოპერაციად ითვლება შემდეგი სამი ოპერაცია:

- წყვილის შექმნის ოპერაცია — `prefix (x, y) = x : y = [x | y]`. მას ხშირად უწოდებენ კონსტრუქტორს ანუ შემდგენელს.
- თავის გამოყოფის ოპერაცია — `head (x) = h (x)`. ეს არის პირველი სელექტორული ოპერაცია.
- კუდის გამოყოფის ოპერაცია — `tail (x) = t (x)`. ეს არის მეორე სელექტორული ოპერაცია.

თავისა და კუდის გამოყოფის სელექტორულ ოპერაციებს ხშირად უწოდებენ უბრალოდ სელექტორებს. ეს ოპერაციები დაკავშირებულია ერთმანეთთან შემდეგი სამი აქსიომით:

```
1. head (x : y) = x
2. tail (x : y) = y
3. prefix (head (x : y), tail (x : y)) = (x : y)
```

ყველა ობიექტის სიმრავლეს, რომელიც შეიძლება კონსტრუირდეს პირველადი ტიპის ობიექტებისგან ბაზისური ოპერაციების გამოყენების შედეგად, უწოდებენ `S` გამოსახულებას (აღნიშვნა — `Sexpr (A)`). მაგალითად:

```
a1 : (a2 : a3) in Sexpr
```

შემდგომი კვლევისთვის შემოდის ცნება *ფიქსირებული ატომი*, რომელიც აგრეთვე ეკუთვნის პირველად `A` ტიპს. ამ ატომს შემდგომ ვუწოდებთ „ცარიელ სიას“ და აღვნიშნავთ სიმბოლოებით `[]` (თუმცა ფუნქციონალური დაპროგრამების სხვადასხვა ენაში შეიძლება გამოყენებული იყოს ცარიელი სიის სხვა აღნიშვნებიც). ახლა აღვწეროთ ის, რაზეც ოპერირებს ფუნქციონალური დაპროგრამება — `List (A) Sexpr (A)`, საკუთრივი ქვესიმრავლე, რომელსაც ეწოდება „სია `A`-ზე“.

განმარტება:

1. ცარიელი სია [] in List (A)
2. x in A & y in List (A) => x : y in List (A)

სიის მთავარი თვისება არის: x in List (A) & x =/= [] => head (x) in A; tail (x) in List (A).

n-ელემენტიანი სიის აღსანიშნავად შეიძლება გამოყენებულ იქნეს სხვადასხვა ნოტაცია, თუმცა ჩვენ გამოვიყენებთ მხოლოდ ასეთს: [a1, a2, ..., an]. სიასთან ოპერაციების - head-ის და tail-ის მეშვეობით შეიძლება სიის თითოეულ ელემენტთან წვდომა, რადგანაც:

head ([a1, a2, ..., an]) = a1

tail ([a1, a2, ..., an]) = [a2, ..., an]
(როცა n > 0).

სიების გარდა შემოდის მონაცემების კიდევ ერთი ტიპი, რომელსაც ეწოდება „სიური სტრუქტურა A-ზე“ (აღნიშვნა – List_str (A)), ამასთან, შესაძლოა აგებულ იქნეს დამოკიდებულების შემდეგი სტრუქტურა: List (A) List_str (A) Sexpr (A). სიური სტრუქტურის განმარტებას ასეთი სახე აქვს: *განმარტება:*

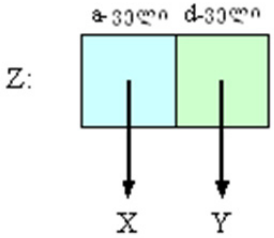
1°. a in A => a in List_str (A)

2°. List (List_str (A)) in List_str (A)

ანუ, ჩანს, რომ სიური სტრუქტურა – ეს არის სია, რომლის ელემენტები შეიძლება იყოს როგორც ატომები, ასევე სხვა სიური სტრუქტურები, მათ შორის ჩვეულებრივი სიები. სიური სტრუქტურის მაგალითი, რომელიც ამავე დროს არ არის მარტივი სია, არის შემდეგი გამოსახულება: [a1, [a2, a3, [a4]], a5]. სიური სტრუქტურისთვის შემოდის ისეთი ცნება, როგორიცაა *ჩადგმის დონე.*

განვიხილოთ სიებისა და სიური სტრუქტურების პროგრამული რეალიზაციები. ეს საჭიროა იმისთვის, რომ გავიგოთ, რა ხდება ფუნქციონალური პროგრამის მუშაობისას როგორც რომელიმე კონკრეტულ ფუნქციონალურ ენაზე, ასევე აბსტრაქტულ ენაზე.

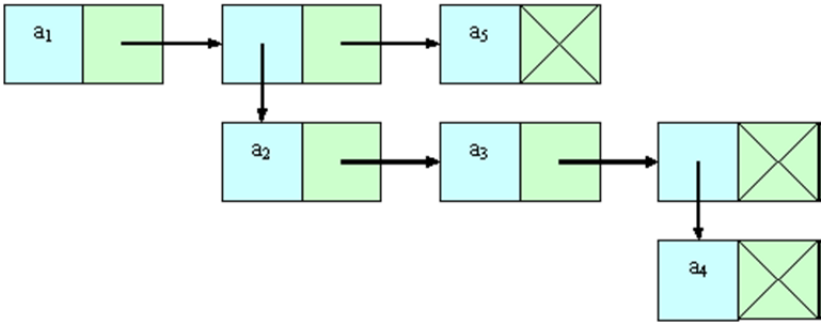
თითოეული ობიექტი მანქანის მეხსიერებაში იკავებს რაღაც ადგილს. ატომები წარმოადგენს მიმთითებლებს (მისამართებს) უჯრედებზე, რომლებშიც ობიექტი ინახება. ასეთ შემთხვევაში წყვილი `z = x : y` გრაფიკულად შეიძლება წარმოვადგინოთ, როგორც ნაჩვენებია შემდეგ სურათზე:



უჯრედის მისამართი, რომელიც შეიცავს მიმთითებლებს `x`-ზე და `y`-ზე, არის ობიექტი `z`. როგორც ნახატიდან ჩანს, წყვილი წარმოდგენილია ორი მისამართით – მიმთითებლით თავზე და კუდზე. ტრადიციულად პირველ მიმთითებელს უწოდებენ `a`-ველს, მეორე მიმთითებელს – `d`-ველს.

ობიექტები, რომელზეც `a`-ველი და `d`-ველი მიუთითებს, რომ უფრო მოსახერხებლად წარმოვადგინოთ, შემდგომ მათ ჩავწერთ უშუალოდ ველებში. ცარიელ სიას აღვნიშნავთ გადახაზული კვადრატით.

ამრიგად, სიური სტრუქტურა `[a1, [a2, a3, [a4]], a5]` შეიძლება წარმოდგეს შემდეგნაირად:



სურათზე კარგად ჩანს ჩადგმის დონეები: – `a1` და `a5` ატომებს აქვთ ჩადგმის დონე 1, ატომებს `a2` და `a3` – 2, ხოლო ატომს `a4` – შესაბამისად, 3.

აღვნიშნოთ, რომ ოპერაცია `prefix` ითხოვს მეხსიერებას, ვინაიდან წყვილის კონსტრუირების დროს გამოიყოფა მეხსიერება მიმთითებლებისთვის. მეორე მხრივ, ოპერაციებს `head` და `tail` არ სჭირდება მეხსიერება, ისინი უბრალოდ აბრუნებენ მისამართებს, რომლებიც შეიცავს, შესაბამისად, `a`-ველს და `d`-ველს.

თავდაპირველად განვიხილოთ ოპერაცია `prefix`-ის მუშაობა უფრო დატალურად, სამ ზოგად მაგალითზე:

1. `prefix (a1, a2) = a1 : a2` (ამასთან, შედეგი არ არის `List_str` (A-ის ელემენტი).
2. `prefix (a1, [b1, b2]) = [a1, b1, b2]`
3. `prefix ([a1, a2], [b1, b2]) = [[a1, a2], b1, b2]`

შემდეგი მაგალითი: სიის სიგრძის განსაზღვრის ფუნქცია `Length`.

მანამ, სანამ დავიწყებდეთ უშუალოდ ფუნქცია `Length`-ის რეალიზებას, ვნახოთ, თუ რას აბრუნებს იგი. ფუნქცია `Length`-ის შედეგი არის ელემენტების რაოდენობა იმ სიაში, რომელიც გადაეცემა მას პარამეტრად. აქ ორი შემთხვევაა – ფუნქციას გადაეცა ცარიე-

ლი სია და ფუნქციას გადაეცა არაცარიელი სია. პირველ შემთხვევაში, ცხადია, შედეგი უნდა იყოს 0. მეორე შემთხვევაში ამოცანა ორ ქვეამოცანად იყოფა, სია იყოფა თავად და კუდად ოპერაციების `head` და `tail` საშუალებით.

ვიცით, რომ `head` აბრუნებს სიის პირველ ელემენტს, ხოლო ოპერაცია `tail` აბრუნებს დანარჩენი ელემენტების სიას. თუ გვეცოდინება რისი ტოლია `tail` ოპერაციით მიღებული სიის სიგრძე, მაშინ თავდაპირველი სიის სიგრძე იქნება ეს სიგრძე ერთით გადიდებული. ახლა უკვე ადვილად შეგვიძლია დავწეროთ ფუნქცია `Length`-ის განმარტება:

```
Length ([]) = 0
Length (L) = 1 + Length (tail (L))
```

მაგალითი: ორი სიის შერწყმის ფუნქცია `Append`.

ორი სიის შერწყმა (ანუ გაერთიანება) შეიძლება რამდენიმე საშუალებით. პირველია დესტრუქციული მინიჭება, ანუ შევცვალოთ [] სიაზე მინიჭება მეორე სიის თავზე მიმთითებლით და ამით მივიღებთ შედეგს პირველ სიაში. მაგრამ ამ დროს იცვლება პირველი სია. ასეთი მიდგომები ფუნქციონალურ დაპროგრამებაში დაუშვებელია (თუმცა ზოგიერთ ენაში ეს დასაშვებია).

მეორე მიდგომა მდგომარეობს შემდეგში: მოვახდინოთ პირველი სიის კოპირება ზედა დონეზე და მოვათავსოთ კოპიოს ბოლო მიმთითებლის ნაცვლად მიმთითებელი მეორე სიის პირველ ელემენტზე. ეს მიდგომა კარგია იმით, რომ არ ასრულებს დესტრუქციულ მოქმედებას და არ აქვს გვერდითი ეფექტები, თუმცა მოითხოვს დამატებით მეხსიერებასა და დროს.

```
Append ([], L2) = L2
Append (L1, L2) = prefix (head (L1), Append
(tail (L1), L2))
```

ბოლო მაგალითი გვიჩვენებს, თუ როგორ შეიძლება თანდათანობითი კონსტრუირებით აიგოს ახალი სია, რომელიც იქნება ორი მოცემულის შერწყმა.

სავარჯიშო №1

1. ააგეთ ფუნქცია, რომელიც გამოითვლის შემდეგი მიმდევრობების n-ურ წევრს:

   a. $a_n = x_n$
   b. $a_n = \sum_{i=1}^{n} i$
   c. $a_n = \sum_{i=1}^{n} \sum_{j=1}^{i} j$
   d. $a_n = \sum_{i=1}^{p} n^{-i}$
   e. $a_n = e^n = \sum_{i=0}^{\infty} (n^i / i!)$

2. ახსენით `prefix` ოპერაციის შედეგი, რომელიც მოყვანილია მაგალითში. ახსნისას შეგიძლიათ გამოიყენოთ გრაფიკული მეთოდი.
3. ახსენით ფუნქცია `Append`-ის შედეგი. რატომ არ არის ფუნქცია დესტრუქციული?
4. ააგეთ ფუნქცია, რომელიც მუშაობს სიებთან:
   a. `GetN` — ფუნქცია, რომელიც მოცემული სიიდან n-ურ ელემენტს გამოყოფს.
   b. `ListSumm` — ორი სიის ელემენტების შეკრება. აბრუნებს სიას, რომელიც არის მოცემული სიების შესაბამისი ელემენტების ჯამი. გაითვალისწინეთ, რომ სიების სიგრძე შეიძლება იყოს სხვადასხვა.
   c. `OddEven` — ფუნქცია უცვლის ადგილებს მეზობელ ლუწ და კენტ ელემენტებს მოცემულ სიაში.
   d. `Reverse` — ფუნქცია, რომელიც აბრუნებს სიას (სიის პირველი ელემენტი ხდება ბოლო, მეორე – ბოლოდან მეორე და ა.შ. ბოლო ელემენტამდე).
   e. `Map` — ფუნქცია, რომელიც იყენებს მეორე ფუნქციას (რომელიც პარამეტრად გადაეცემა თავდაპირველ ფუნქციას) მოცემული სიის ყველა ელემენტთან.

თავი 2.3. ტიპები ფუნქციონალურ ენებში

როგორც ცნობილია, ფუნქციის არგუმენტები შეიძლება იყოს არა მხოლოდ ბაზური ტიპის ცვლადები, არამედ სხვა ფუნქციებიც. ამ შემთხვევაში ჩნდება მაღალი რიგის ფუნქციის ცნება. შემოვიღოთ ფუნქციონალური ტიპის ცნება (ანუ ტიპის, რომელიც აბრუნებს ფუნქციას). ვთქვათ, რომელიღაც `f`-ფუნქცია არის ერთი ცვლადის ფუნქცია `A` სიმრავლიდან, რომელიც ღებულობს მნიშვნელობებს `B` სიმრავლიდან, მაშინ განსაზღვრების თანახმად:

```
#(f) : A -> B
```

აქ ნიშანი `#(f)` აღნიშნავს „ფუნქცია `f`-ის ტიპი“. ამრიგად, ტიპს, რომელსაც აქვს სიმბოლო ისარი `->`, ეწოდება ფუნქციონალურ ტიპი. ზოგჯერ მისთვის გამოიყენება აღნიშვნა: B^A (შემდგომ გამოვიყენებთ მხოლოდ ისრიან ჩანაწერს, ვინაიდან ზოგიერთი ფუნქციის ტიპი ძალზე რთულად წარმოდგება ხარისხებით).

მაგალითად:

```
#(sin) : Real -> Real
#(Length) : List (A) -> Integer
```

მრავალარგუმენტიანი ფუნქციისთვის ტიპის განსაზღვრა შეიძლება გამოყვანილი იყოს ოპერაციით – დეკარტული ნამრავლით, მაგალითად:

```
#(add(x, y)) : Real x Real -> Real).
```

თუმცა ფუნქციონალურ დაპროგრამებაში ასეთმა საშუალებამ გამოყენება ვერ პოვა.

1924 წელს მ. შონფიკელმა მრავალარგუმენტიანი ფუნქცია წარმოადგინა როგორც ერთარგუმენტიანი ფუნქციების თანმიმდევრობა. ასეთ შემთხვევაში ფუნქციის ტიპი, რომელიც შეკრებს ორ ნამდვილ რიცხვს, წარმოდგება ასე: `Real -> (Real -> Real)`. ანუ ასეთი ფუნქციის ტიპი მიიღება სიმბოლო ისრის -> თანმიმდევრული გამოყენებით. განვსაზღვროთ ეს პროცესი შემდეგ მაგალითზე:

განვიხილოთ, მაგალითად, ფუნქცია `add (x,y)`-ის ტიპი.

დავუშვათ, ფუნქცია `add`-ის თითოეულ არგუმენტს უკვე აქვს მნიშვნელობა, ვთქვათ `x = 5, y = 7`. ამ შემთხვევაში, თუ ფუნქცია `add`-ს მოვაშორებთ პირველ არგუმენტს, მივიღებთ ახალ ფუნქციას — `add5`, რომელიც თავის ერთადერთ არგუმენტს უმატებს რიცხვ 5-ს. ამ ფუნქციის ტიპი მიიღება ადვილად და წარმოდგება ასე: `Real -> Real`. ახლა, თუ უკან დავბრუნდებით, უკვე გვესმის, რატომ არის `add` ფუნქციის ტიპი `Real -> (Real -> Real)`.

`add5` ტიპის ფუნქციები რომ არ დაეწერათ (როგორც წინა მაგალითში), მოიგონეს სპეციალური აპლიკაციური ჩაწერის ფორმა „ოპერატორი – ოპერანდი“. ამის წინაპირობა გახდა ახალი ხედვა ფუნქციისა ფუნქციონალურ დაპროგრამებაში. ტრადიციულად, ითვლებოდა, რომ გამოსახულება `f (5)` აღნიშნავს „f ფუნქციის გამოყენებას არგუმენტის მნიშვნელობასთან, რომელიც ტოლია 5-ის“ (ანუ, გამოითვლება მხოლოდ არგუმენტი). ასევეა ფუნქციონალურ დაპროგრამებაშიც.

ამრიგად, თუ ფუნქცია `f`-ს აქვს ტიპი A_1 `->` (A_2 `-> ( ... (`A_n `-> B) ... )`), მაშინ, მნიშვნელობის სრულად გამოსათვლელად `f (`a_1`,` a_2`, ...,` a_n`)`, აუცილებელია თანმიმდევრულად გამოვთვალოთ `( ... (f (`a_1`)` a_2`) ... )` a_n და გამოთვლის შედეგი იქნება `B` ტიპის ობიექტი.

შესაბამისად, გამოსახულებას, რომელშიც ყველა ფუნქცია განიხილება როგორც ერთარგუმენტიანი ფუნქცია და ერთადერთ ოპერაციას წარმოადგენს აპლიკაცია (გამოყენება), ეწოდება გამოსახულება ფორმით „ოპერატორი – ოპერანდი“. ასეთმა ფუნქციებმა მიიღეს სახელწოდება „კარირებული“, ხოლო თვითონ ფუნქციის დაყვანის ზემოთ აღწერილმა პროცესმა – „კარირება“ (კარი ჰასკელის სახელიდან გამომდინარე).

თუ გავიხსენებთ λ-აღრიცხვას, აღმოვაჩენთ, რომ მასში უკვე არის მათემატიკური აბსტრაქცია ჩანაწერების აპლიკაციური ფორმისთვის. მაგალითად:

```
f (x) = x2 + 5          <=>     λx.(x2 + 5)
f (x, y) = x + y        <=>     λy. λx.(x + y)
f (x, y, z) = x2 + y2 + z2  <=>  λz. λy. λx.(x2 +
y2 + z2)
```

და ა.შ...

რამდენიმე სიტყვა აბსტრაქტული ენის ნოტაციის შესახებ

ნიმუშები და კლოზები

აღვნიშნოთ, რომ აბსტრაქტული ფუნქციონალური ენის ნოტაციაში, რომელსაც ვიყენებით ფუნქციის მაგალითების დაწერისას, შესაძლებელი იყო გამოგვეყენებინა ისეთი კონსტრუქცია, როგორიცაა `if-then-else`. მაგალითად, ფუნქცია `Append`-ის აღწერისას მისი ტანი შეიძლება ჩაწერილიყო შემდეგნაირად:

```
Append (L1, L2) = if (L1 == [])        then L2
                  else head (L1) : Append (tail
                  (L1), L2)
```

თუმცა მოცემული ჩანაწერი ცუდად გასარჩევზე დაბრუნებაა, ამიტომაც მაგალითში უკვე გამოვიყენეთ ნოტაცია, რომელიც მხარს უჭერს ე.წ. „ნიმუშებს“.

განმარტება:

ნიმუში ეწოდება გამოსახულებას, რომელიც აგებულია მონაცემთა კონსტრუირების ოპერაციით და რომელიც გამოიყენება მონაცემებთან შესაბამისობისათვის. ცვლადები აღინიშნება დიდი ასოებით, კონსტანტები – პატარით.

ნიმუშის მაგალითებია:

`5` — რიცხვითი კონსტანტა.
`X` — ცვლადი.
`X : (Y : Z)` — წყვილი.
`[X, Y]` — სია.

ნიმუშმა აუცილებლად უნდა დააკმაყოფილოს ერთი მოთხოვნა, წინააღმდეგ შემთხვევაში მასთან შედარება არასწორად შესრულდება. ეს მოთხოვნა ასე ჟღერს: ნიმუშთან მონაცემების შედარებისას ცვლადისთვის მნიშვნელობის მინიჭება უნდა მოხდეს მხოლოდ ერთადერთი გზით, ანუ, მაგალითად, გამოსახულება `(1 + X ==> 5)` შეიძლება გამოვიყენოთ როგორც ნიმუში, რადგანაც `X` ცვლადის აღნიშვნა ხდება ერთადერთი გზით `(X = 4)`, ხოლო შემდეგი გამოსახულების `(X + Y ==> 5)` გამოყენება ნიმუშად არ შეიძლება, ვინაიდან `X` და `Y` ცვლადები აღინიშნება სხვადასხვანაირად (არაცალსახად).

ფუნქციონალურ დაპროგრამებაში ნიმუშის გარდა შემოდის ისეთი ცნება, როგორიცაა „კლოზი“ (ინგლისურიდან „`clause`“). განმარტებით, კლოზი არის:

`def f` p_1`, ...,` p_n `= expr`

სადაც:

`def` და `=` — აბსტრაქტული ენის კონსტანტებია.
`f` — განსაზღვრული ფუნქციის სახელია.
p_i — ნიმუშების თანმიმდევრობაა (ამასთან, `>= 0`).
`expr` — გამოსახულებაა.

ამრიგად, ფუნქციონალურ დაპროგრამებაში ფუნქციების განსაზღვრება არის უბრალოდ კლოზების თანმიმდევრობა (შესაძლოა, მხოლოდ ერთი ელემენტისგან შემდგარი). რათა გავამარტივოთ ფუნქციის განსაზღვრების ჩაწერა, შემდგომ სიტყვა `def`-ს გამოვტოვებთ.

მაგალითი. ნიმუშები და კლოზები ფუნქციაში `Length`.

```
Length ([]) = 0
Length (H:T) = 1 + Length (T)
```

ვთქვათ, ფუნქცია `Length`-ის გამოძახება ხდება პარამეტრით `[a, b, c]`. ამ დროს მუშაობას იწყებს ნიმუშთან შედარების მექანიზმი. სათითაოდ გადაისინჯება ყველა კლოზი და ხდება შედარებების მცდელობები. ამ შემთხვევაში წარმატებით მოხდება მხოლოდ მეორე კლოზთან შედარება (რადგანაც სია `[a, b, c]` არ არის ცარიელი).

ფუნქციის გამოძახების ინტერპრეტაცია მდგომარეობს შემდეგში: ხდება შედარება ზემოდან ქვემოთ ნიმუშებში და რიგით პირველი ნიმუშის პოვნა, რომელიც წარმატებით შედარდა ფაქტობრივ პარამეტრებს. ნიმუშის ცვლადების მნიშვნელობები, რომლებიც მათ მიენიჭათ შედარების შედეგად, ჩაისმის კლოზის მარჯვენა მხარეს (გამოსახულებაში `expr`), რომლის მნიშვნელობის გამოთვლაც მოცემულ კონტექსტში წარმოადგენს ფუნქციის გამოძახების მნიშვნელობას.

დაცვა

აბსტრაქტულ ნოტაციაში ფუნქციის დაწერისას დაშვებულია ე.წ. დაცვის გამოყენება, ანუ ნიმუშის ცვლადებზე შეზღუდვების გამოყენება. მაგალითად, დაცვის გამოყენებით ფუნქცია `Length`-ის განსაზღვრა შეიძლება იყოს შემდეგი:

```
Length (L) = 0 when L == []
Length (L) = 1 + Length (tail (L)) otherwise
```

განხილულ კოდში სიტყვა `when` (მაშინ) და `otherwise` (წინააღმდეგ შემთხვევაში) წარმოადგენს ენის დარეზერვებულ სიტყვებს. თუმცა, ამ სიტყვების გამოყენება არ არის დაცვის გამოყენებისთვის აუცილებელი პირობა. დაცვა შეიძლება განვახორციელოთ სხვადასხვა საშუალებით, მათ შორის λ-აღრიცხვით:

```
Append = λ[].(λL.L)
Append = λ(H:T).(λL.H : Append (T, L))
```

წარმოდგენილი ჩანაწერი ცუდი წასაკითხია, ამიტომ მას მხოლოდ უკიდურეს შემთხვევებში გამოვიყენებთ.

ლოკალური ცვლადები

როგორც უკვე აღვნიშნეთ, ლოკალური ცვლადების გამოყენება იწვევს გვერდით ეფექტს, ამიტომ იგი დაუშვებელია ფუნქციონალურ ენებში. თუმცა, ზოგიერთ შემთხვევაში, ლოკალური ცვლადების გამოყენება არის ოპტიმალური, რაც იძლევა გამოთვლების დროის და რესურსების ეკონომიის საშუალებას.

დავუშვათ, `f` და `h` ფუნქციებია და აუცილებელია გამოითვალოს გამოსახულება `h (f(X), f(X))`. თუ ენაში არ არის ჩადებული ოპტიმიზაციის მეთოდები, მაშინ ხდება `f (X)` გამოსახულების ორჯერ გამოთვლა. ეს რომ არ მოხდეს, შეიძლება მივმართოთ ასეთ საშუალებას: `(λv.h (v, v))(f (X))`. ბუნებრივია, რომ ამ შემთხვევაში გამოსახულება `f (X)` გამოითვლება პირველად და ერთხელ. იმისათვის, რომ λ–აღრიცხვა მინიმალურად გამოვიყენოთ, შემდგომ ასეთი სახის ჩანაწერით ვისარგებლებთ:

```
let v = f (X) in h (v, v)
```

სიტყვები `let`, `=` და `in` `Haskell` ენის დარეზერვებული სიტყვებია. ამ შემთხვევაში `v`-ს ვუწოდებთ ლოკალურ ცვლადს.

დაპროგრამების ელემენტები

პარამეტრების დაგროვება – აკუმულატორი

ფუნქციის შესრულებისას შეიძლება დადგეს მეხსიერების ხარჯვის სერიოზული პრობლემა. ავხსნათ ეს პრობლემა ფუნქციის მაგალითზე, რომელიც ითვლის რიცხვის ფაქტორიალს:

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

თუ მოვიყვანთ ამ ფუნქციის გამოთვლის მაგალითს არგუმენტზე 3, მაშინ დავინახავთ შემდეგ თანმიმდევრობას:

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

გამოთვლის ამ მაგალითზე ვხედავთ, რომ ფუნქციის რეკურსიული გამოძახებისას დიდი მოცულობით გამოიყენება კომპიუტერის მეხსიერება. მეხსიერება არგუმენტის მნიშვნელობის პროპორციულია, ამასთან არგუმენტიც შეიძლება იყოს დიდი. ჩნდება კითხვა, შესაძლებელია თუ არა ისე დაიწეროს ფაქტორიალის გამოთვლის ფუნქცია (და მისი მსგავსი ფუნქციები), რომ მეხსიერება მინიმალურად იქნეს გამოყენებული?

ამ შეკითვაზე დადებითი პასუხისთვის აუცილებელია განვიხილოთ აკუმულატორის (დამგროვებლის) ცნება.

განვიხილოთ მაგალითი – რიცხვის ფაქტორიალის გამოთვლის ფუნქცია აკუმულატორის გამოყენებით:

```
factorial_A (N) = f (N, 1)

f (0, A) = A
f (N, A) = f ((N - 1), (N * A))
```

ამ მაგალითში ფუნქცია `F`-ის მეორე პარამეტრი ასრულებს აკუმულატორი ცვლადის როლს. სწორედ იგი შეიცავს შედეგს, რომელიც ბრუნდება რეკურსიის დამთავრებისას. თვითონ რეკურსიას კი ამ დროს აქვს „კუდური“ რეკურსიის სახე, ამასთან, მეხსიერება გამოიყენება მხოლოდ ფუნქციის მნიშვნელობების დასაბრუნებელი მისამართების შენახვისათვის.

კუდური რეკურსია წარმოადგენს რეკურსიის სპეციალურ სახეს, რა დროსაც მხოლოდ ერთხელ ხდება რეკურსიული ფუნქციის გამოძახება და ეს გამოძახებაც სრულდება ყველა გამოთვლის შემდეგ.

კუდური რეკურსიის რეალიზაცია შეიძლება მოხდეს იტერაციის პროცესის საშუალებით. პრაქტიკაში ეს ნიშნავს, რომ ფუნქციონალური ენის „კარგ“ ტრანსლატორს უნდა „შეეძლოს“ აღმოაჩინოს კუდური რეკურსია და მოახდინოს მისი რეალიზება ციკლის სახით. თავის მხრივ, პარამეტრის დაგროვების მეთოდს ყოველთვის არ მივყავართ კუდურ რეკურსიამდე, თუმცა იგი ცალსახად გვეხმარება საერთო მეხსიერების მოცულობის შემცირებაში.

დამგროვებელი პარამეტრებით განსაზღვრებების აგების პრინციპები:

1. შემოდის ახალი ფუნქცია დამატებითი არგუმენტით (აკუმულატორით), რომელშიც გროვდება გამოთვლების შედეგები.
2. აკუმულატორი არგუმენტის საწყისი მნიშვნელობა მოიცემა ტოლობით, რომელიც აკავშირებს ძველ და ახალ ფუნქციებს.
3. საწყისი ფუნქციის ის ტოლობა, რომელიც შეესაბამება რეკურსიიდან გამოსავალს, იცვლება აკუმულატორით დაბრუნებით.
4. ტოლობა, რომელიც შეესაბამება რეკურსიულ განსაზღვრებას, გამოიხატება როგორც ახალ ფუნქციაზე მიმართვა, რომელშიც აკუმულატორი იღებს იმ მნიშვნელობას, რომელიც ბრუნდება საწყისი ფუნქციით.

ისმის შეკითხვა: შეიძლება გარდავქმნათ ნებისმიერი ფუნქცია აკუმულატორის გამოთვლის ფორმით? ბუნებრივია, რომ ამ შეკითხვის პასუხი უარყოფითია. დამგროვებელი პარამეტრიანი ფუნქციის აგების ხერხი არ არის უნივერსალური და იგი არ არის

კუდური რეკურსიის აგების გარანტია. მეორე მხრივ, განსაზღვრების აგება დამგროვებელი პარამეტრით არის შემოქმედებითი საქმე. ამ პროცესში აუცილებელია ზოგიერთი ევრისტიკის გამოყენება.

განსაზღვრება:

რეკურსიული განსაზღვრების ზოგად სახეს, რომელიც საშუალებას იძლევა ტრანსლაციისას უზრუნველყოს გამოთვლები მეხსიერების მუდმივ მოცულობაში იტერაციის საშუალებით, უწოდებენ იტერაციული სახის ტოლობებს.

$f_i\ (p_{ij}) = e_{ij}$

ამასთან, გამოსახულება e_{ij}-ს ედება შემდეგი შეზღუდვები:

1. e_{ij} – „მარტივი“ გამოსახულებაა, ანუ ის შეიცავს მხოლოდ მონაცემებზე ოპერაციებს და არ შეიცავს რეკურსიულ გამოძახებებს.
2. e_{ij}-ს აქვს სახე $f_k\ (v_k)$, ამასთან v_k – მარტივი გამოსახულებების თანმიმდევრობაა. ეს არის კუდური რეკურსია.
3. e_{ij} – პირობითი გამოსახულებაა პირობაში მარტივი გამოსახულებით, რომლის შტოები განისაზღვრება ამავე სამი პირობით.

სავარჯიშო №2

1. ააგეთ ფუნქცია, რომელიც მუშაობს სიებთან. საჭიროების შემთხვევაში გამოიყენეთ დამატებითი და ზემოთ განსაზღვრული ფუნქციები.

   a. `Reverse_all` — ფუნქცია, რომელიც შესასვლელზე იღებს სიურ სტრუქტურას და აბრუნებს მის ყველა ელემენტს და ასევე მას.

b. `Position` — ფუნქცია, რომელიც აბრუნებს მოცემული ატომის სიაში პირველად შესვლის ნომერს.

c. `Set` — ფუნქცია, რომელიც აბრუნებს სიას, რომელშიც მოცემული სიის ყველა ატომი მხოლოდ ერთხელ შედის.

d. `Frequency` — ფუნქცია, რომელიც აბრუნებს წყვილების სიას (სიმბოლო, სიხშირე). თითოეული წყვილი განისაზღვრება მოცემული სიის ატომით და ამ სიაში მისი შესვლის სიხშირით.

2. დაწერეთ ფუნქციები დამგროვებელი პარამეტრებით (თუ ეს შესაძლებელია) სავარჯიშო 1-ში მოყვანილი ფუნქციებისთვის.

თავი 2.4. HASKELL ენის საფუძვლები

გავეცნოთ Haskell ენის სინტაქსს. განვიხილოთ ენის ყველა მნიშვნელოვანი ცნება, მათი შესაბამისობა აბსტრაქტულ-ფუნქციონალური ენის ცნებებთან. ასევე, არსებული ტრადიციების შესაბამისად, მოვიყვანოთ მაგალითები Lisp-ზე.

მონაცემთა სტრუქტურები და მათი ტიპები

დაპროგრამების ნებისმიერი ენის ძირითადი საბაზისო ელემენტი არის სიმბოლო (ლექსემი). სიმბოლოს, ტრადიციულად, უწოდებენ ასოების, ციფრებისა და სპეციალური ნიშნების შეზღუდული ან შეუზღუდავი სიგრძის თანმიმდევრობას. ზოგიერთ ენაში დიდი და პატარა ასოები განსხვავდება, ზოგიერთში – არა. Lisp-ში – არ განსხვავდება, Haskell-ში – განსხვავება არის.

სიმბოლოები ყველაზე ხშირად იდენტიფიკატორების როლშია, როგორიცაა მუდმივების (კონსტანტების), ცვლადების, ფუნქციების სახელები. მუდმივების, ცვლადების და ფუნქციების მნიშვნელობები კი ნიშნაკების ტიპიზებული თანმიმდევრობაა. ასე რომ, რიცხვითი კონსტანტის მნიშვნელობა შეიძლება იყოს ასოების სტრიქონი და ა.შ. ფუნქციონალურ ენებში არსებობს საბაზისო განმარტება – *ატომი*. ატომებს უწოდებენ სიმბოლოებს და ციფრებს. ციფრებისაგან შემდგარი რიცხვი შეიძლება იყოს სამი სახის: მთელი, ფიქსირებული და მცოცავი მძიმით.

ფუნქციონალური დაპროგრამების შემდეგ ცნებას წარმოადგენს *სია*. აბსტრაქტულ მათემატიკურ ნოტაციაში და Haskell-შიც

გამოიყენება სიმბოლოები `[]`. Lisp-ში გამოიყენება მრგვალი ფრჩხილები – `()`.Lisp- ' სიის ელემენტები ერთმანეთისგან ხარვეზებით გამოიყოფა, რაც ნაკლებ თვალსაჩინოა, ამიტომაც Haskell-ში სიის ელემენტების გამოსაყოფად გადაწყდა მძიმის (,) გამოყენება. ასე რომ, სია `[a, b, c]` სწორი ჩანაწერია Haskell-ის სინტაქსის შესაბამისად. Lisp-ის ნოტაციით ის ჩაიწერება როგორც `(a b c)`. თუმცა Lisp-ის შემქმნელებმა დაუშვეს წერტილოვანი ჩაწერა წყვილებისთვის. ამრიგად, ზემოთ მოყვანილი სია ასეც შეიძლება ჩაიწეროს: `(a.(b.(c.NIL)))`.

სიური სტრუქტურები Lisp-შიც და Haskell-შიც აღიწერება ნოტაციის შესაბამისად – ერთი სია შეიცავს მეორეს.

როგორც შესავალში იყო აღნიშნული, ფუნქციონალურ ენებში მონაცემთა ტიპები განისაზღვრება ავტომატურად. ტიპების ავტომატურად განსაზღვრის მექანიზმი დევს Haskell-შიც. თუმცა, ზოგიერთ შემთხვევაში აუცილებელია ცხადად მივუთითოთ ტიპი, წინააღმდეგ შემთხვევაში შესაძლოა ინტერპრეტატორმა ის ვერ განსაზღვროს (ხშირ შემთხვევაში გამოდის შეტყობინება ან შეცდომა). Haskell-ში გამოიყენება სპეციალური სიმბოლო `::`(ორი ორი წერტილი), რომელიც ასე იკითხება: „აქვს ტიპი“, ანუ, თუ დავწერთ:

```
5 :: Integer
```

ეს წაიკითხება ასე: „რიცხვით კონსტანტა 5-ს აქვს ტიპი `Integer` (მთელი რიცხვი)“.

თუმცა Haskell მხარს უჭერს ისეთ გამორჩეულ საშუალებას, როგორიცაა პოლიმორფული ტიპები, ანუ ტიპების შაბლონებს. თუ, მაგალითად, ჩანაწერი `[a]`, აღნიშნავს ნებისმიერი ტიპის ატომების სიას, ამასთან, ატომების ტიპი უნდა იყოს ერთი და იგივე მთე-

ლი სიისთვის. მაგალითად, სიებს: `[1, 2, 3]` და `['a', 'b', 'c']` ექნება ტიპი `[a]`, ხოლო სია `[1, 'a']` იქნება სხვა ტიპის. ამ შემთხვევაში ჩანაწერში `[a]` სიმბოლო `a`-ს აქვს ტიპური ცვლადის მნიშვნელობა.

შეთანხმებები დასახელებებში

`Haskell`-ში მნიშვნელოვანია შეთანხმებები დასახელებებში, ვინაიდან ისინი შედიან ენის სინტაქსში (რაც, საზოგადოდ, არ არის იმპერატიულ ენებში). ყველაზე მთავარი შეთანხმებაა იდენტიფიკატორის დასაწყისში დიდი ასოს გამოყენება. ტიპების სახელები, მათ შორის პროგრამისტის მიერ განსაზღვრული, უნდა იწყებოდეს დიდი ასოთი. ფუნქციების, ცვლადებისა და მუდმივების სახელები უნდა იწყებოდეს პატარა ასოთი. იდენტიფიკატორის პირველი სიმბოლო შეიძლება იყოს სპეციალური ნიშანი, რომელთაგან ზოგიერთი ცვლის მის სემანტიკას.

სიებისა და მათემატიკური თანმიმდევრობების განსაზღვრა

`Haskell`, სამწუხაროდ, დაპროგრამების ერთადერთი ენაა, რომელიც შესაძლებლობას გაძლევს მარტივად და სწრაფად მოვახდინოთ სიების კონსტრუირება, რომლებიც განსაზღვრულია მარტივი ფორმულით. იგი ჩვენ უკვე გამოვიყენეთ სიის ჰუარეს სწრაფი დახარისხების მეთოდის დემონსტრაციისას (მაგალითი 3). სიების განსაზღვრის ყველაზე ზოგადი სახე ასეთია:

```
[ x | x <- xs ]
```

ეს ჩანაწერი შეიძლება ასე წავიკითხოთ: „ყველა ისეთი `x`-ის სია, რომელიც აღებულია `xs`-დან“. სტრუქტურას „`x xs`“ უწოდე-

ბენ გენერატორს. ასეთი გენერატორის შემდეგ (ის უნდა იყოს მხოლოდ ერთი და იდგეს პირველ ადგილას სიის განსაზღვრის ჩანაწერში) შეიძლება იყოს დაცვის რამდენიმე გამოსახულება, ერთმანეთისგან მძიმეებით გამოყოფილი. ასეთ დროს ამოირჩევა ყველა ისეთი x, რომლისთვისაც დაცვის ყველა გამოსახულების მნიშვნელობებიც იქნება ჭეშმარიტი. ანუ ჩანაწერი:

```
[ x | x <- xs, x > m, x < n ]
```

შეიძლება წავიკითხოთ ასე: „ყველა ისეთი x-ის სია, აღებული xs-დან, რომელიც აკმაყოფილებს პირობებს: x მეტია m-ზე და x ნაკლებია n-ზე“.

Haskell-ის შემდეგი მნიშვნელოვანი თავისებურებაა უსასრულო სიებისა და მონაცემთა სტრუქტურების მარტივი ფორმირება. უსასრულო სიები შეიძლება ფორმულირდეს როგორც განსაზღვრული სიების საფუძველზე, ასევე სპეციალური ნოტაციის საშუალებით. მაგალითად, ქვემოთ მოყვანილია უსასრულო სია, რომელიც ნატურალური რიცხვებისგან შედგება. მეორე სია წარმოადგენს კენტი ნატურალური რიცხვების სიას:

```
[1, 2 ..]
[1, 3 ..]
```

ორი წერტილის საშუალებით ასევე შესაძლებელია განისაზღვროს ნებისმიერი არითმეტიკული პროგრესია როგორც სასრული, ისე უსასრულო. თუ პროგრესია სასრულია, მაშინ დასახელდება პირველი და ბოლო ელემენტები. პროგრესიის სხვაობა გამოითვლება მოცემული მეორე და პირველი ელემენტის სხვაობით. ზემოთ მოყვანილ მაგალითებში პირველი პროგრესიის სხვაობაა 1, მეორისა – 2. ასე რომ, თუ საჭიროა განისაზღვროს კენტი რიცხვე-

ბის პროგრესია `1`-დან `10`-მდე, მაშინ საჭიროა ჩაიწეროს ასე: `[1, 3 .. 10]`. შედეგი იქნება სია `[1, 3, 5, 7, 9]`.

მონაცემთა უსასრულო სტრუქტურა შეიძლება განისაზღვროს უსასრულო სიების საფუძველზე, ასევე შესაძლოა რეკურსიული მექანიზმების გამოყენება. ამ შემთხვევაში რეკურსია გამოიყენება როგორც რეკურსიულ ფუნქციაზე მიმართვა. მონაცემთა უსასრულო სტრუქტურების შექმნის მესამე საშუალებაა უსასრულო ტიპების გამოყენება.

მაგალითად, განვიხილოთ, თუ როგორ განისაზღვროს ტიპი ორობითი ხეების წარმოსადგენად:

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

Branch       :: Tree a -> Tree a -> Tree a
Leaf         :: a -> Tree a
```

ამ მაგალითში ნაჩვენებია უსასრულო ტიპის განსაზღვრის საშუალება. ჩანს, რომ რეკურსიის გარეშე ეს არ მოხერხდა. თუმცა, თუ არ არის აუცილებლობა შეიქმნას მონაცემთა ახალი ტიპი, უსასრულო სტრუქტურა შეიძლება მივიღოთ ფუნქციის საშუალებით:

```
ones                = 1 : ones
numbersFrom n       = n : numberFrom (n + 1)
squares             = map (^2) (numbersFrom 0)
```

პირველი ფუნქცია განსაზღვრავს უსასრულო თანმიმდევრობას, შედგენილს მხოლოდ ერთიანებისგან. მეორე ფუნქცია აბრუნებს მთელ რიცხვებს, დაწყებულს მოცემული რიცხვიდან. მესამე ფუნქცია აბრუნებს ნატურალური რიცხვების კვადრატებს, დაწყებულს ნულიდან.

ფუნქციის გამოძახების მათემატიკური ნოტაცია ტრადიციულად გულისხმობდა პარამეტრის ჩასმას ფრჩხილებში. ეს ტრადიცია პრაქტიკულად ყველა იმპერატიულმა ენამ გააგრძელა. ფუნქციონალურ ენებში მიღებულია სხვა ნოტაცია – ფუნქციის სახელი გამოიყოფა მისი პარამეტრებისგან უბრალოთ ხარვეზით. Lisp-ში ფუნქცია `length`-ის გამოძახება მოცემულია `L` პარამეტრით, ჩაიწერება სიის სახით:`(length L)`. ასეთი ნოტაცია აიხსნება იმით, რომ ფუნქციონალურ ენებში ფუნქციათა უმრავლესობა კარირებულია.

Haskell-ში საჭირო არ არის ფუნქციის გამოძახება მოვათავსოთ ფრჩხილებში. მაგალითად, თუ განვსაზღვრავთ ორი რიცხვის შეკრების ფუნქციას ასე:

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

მაშინ მის გამოძახებას კონკრეტული პარამეტრებით (მაგალითად, 5 და 7) ექნება სახე:

```
add 5 7
```

აქ ჩანს, რომ Haskell-ის ნოტაცია ძლიერ არის მიახლოებული აბსტრაქტული მათემატიკური ენის ნოტაციასთან. თუმცა Haskell-ში, Lisp-გან განსხვავებით, არის ნოტაცია არაკარირებული ფუნქციების აღსაწერადაც, ანუ ისეთი ფუნქციების, რომელთა ტიპი არ შეიძლება წარმოდგეს სახით `A1 (A2 ... (An B) ...)`. ეს ნოტაცია, როგორც იმპერატიული დაპროგრამების ენები, იყენებს მრგვალ ფრჩხილებს:

```
add (x, y)   = x + y
```

შევნიშნოთ, რომ უკანასკნელი ჩანაწერი – ეს არის ერთი არგუმენტის ფუნქცია Haskell-ის მკაცრ ნოტაციაში. მეორე მხრივ, კარირებული ფუნქციებისთვის შესაძლებელია ნაწილობრივი გამოყენება. ანუ, ორარგუმენტიანი ფუნქციის გამოძახებისას გადავცეთ მხოლოთ ერთი არგუმენტი. ასეთი გამოძახების შედეგი იქნება ასევე ფუნქცია. ამ პროცესის საილუსტრაციოდ განვიხილოთ ფუნქცია inc, რომელიც უმატებს ერთიანს მოცემულ არგუმენტს:

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

ანუ, ამ შემთხვევაში ფუნქცია inc ერთი არგუმენტით უბრალოდ იძახებს ფუნქცია add-ს ორი არგუმენტით, რომელთაგანაც პირველია – 1. ეს არის ნაწილობრივი გამოყენების ცნების ინტუიციური განმარტება. განვიხილოთ კლასიკური მაგალითიც – ფუნქცია map (მისი აღწერა აბსტრაქტულ ფუნქციონალურ ენაზე უკვე განვიხილეთ). აი, ფუნქცია map-ის აღწერა ენა Haskell-ზე:

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

როგორც ვხედავთ, აქ გამოყენებულია ოპერაცია prefix-ის ინფიქსური ჩანაწერი – ორი წერტილი. მხოლოდ ასეთი ჩანაწერი გამოიყენება Haskell-ში წყვილის წარმოდგენისას. ზემოთ მოყვანილი განმარტების შემდეგ შეიძლება მოვახდინოთ ფუნქციის გამოძახება:

```
Prelude> map (add 1) [1, 2, 3, 4]
[2, 3, 4, 5]
```

რადგანაც დაპროგრამების ფუნქციონალური პარადიგმა დაფუძნებულია λ-აღრიცხვაზე, ამიტომ ბუნებრივია, რომ ყველა ფუნქციონალური ენა მხარს უჭერს ნოტაციას λ-აბსტრაქციის წარმოსადგენად. Haskell-ში შესაძლებელია ფუნქციის λ-აბსტრაქციის საშუალებით აღწერა. ამის გარდა, λ-აბსტრაქციის საშუალებით შესაძლებელია ანონიმური ფუნქციის აღწერა (მაგალითად, ერთეული გამოძახებისათვის). ქვემოთ მოყვანილია მაგალითი, სადაც განსაზღვრულია ფუნქციები add და inc λ-აღრიცხვის საშუალებით.

მაგალითი: ფუნქციები add და inc, განსაზღვრული λ-აბსტრაქციით.

```
add  = \x y -> x + y
inc  = \x -> x + 1
```

მაგალითი: ანონიმური ფუნქციის გამოძახება.

```
cubes = map (\x -> x * x * x) [0 ..]
```

მაგალითი გვიჩვენებს ანონიმური ფუნქციის გამოძახებას, რომელსაც გადაცემული პარამეტრი აჰყავს კუბში. ამ ინსტრუქციის შესრულების შედეგი იქნება მთელი რიცხვების კუბების უსასრულო სია, დაწყებული ნულიდან. აუცილებელია აღვნიშნოთ, რომ Haskell-ში გამოიყენება λ-გამოსახულების ჩაწერის გამარტივებული საშუალება, რადგანაც ფუნქცია add ზუსტ ნოტაციაში სწორად ასე უნდა დაგვეწერა:

```
add  = \x -> \y -> x + y
```

შევნიშნოთ, რომ λ-აბსტრაქციის ტიპი განისაზღვრება აბსოლუტურად ისევე, როგორც ფუნქციის ტიპი. `λx.expr` სახის λ-გამოსახულების ტიპი არის `T1 -> T2`, სადაც `T1` — არის ცვლადი `x`-ის ტიპი, ხოლო `T2` — `expr` გამოსახულების ტიპი.

ფუნქციის ჩაწერის ინფიქსური ფორმა

ზოგიერთი ფუნქცია შესაძლოა ჩაიწეროს ინფიქსური ფორმით. ეს ოპერაციები, როგორც წესი, მარტივი ბინარული ოპერაციებია. მაგალითად, ასე წარმოდგება სიების კონკატენაციისა და ფუნქციების კომპოზიციის ოპერაციები:

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

ფუნქციების კომპოზიციის ინფიქსური ოპერაცია:

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

რადგანაც ინფიქსური ოპერაციები წარმოადგენს `Haskell`-ის ფუნქციებს, ანუ ისინი კარირებულია, ამიტომ შესაძლებელია ასეთი ფუნქციების ნაწილობრივი გამოყენება. ამ მიზნისთვის `Haskell`-ში არსებობს სპეციალური ჩანაწერი, რომელსაც უწოდებენ „სექციას“.

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

ზემოთ მოყვანილია სამი სექცია. თითოეული მათგანი განსაზღვრავს სიების კონკატენაციის ინფიქსურ ოპერაციას მასზე გადაცემული არგუმენტების რაოდენობის შესაბამისად. სექციის ჩანაწერში მრგვალი ფრჩხილების გამოყენება სავალდებულოა.

თუ რომელიმე ფუნქცია იღებს ორ პარამეტრს, ასევე შეიძლება მისი ჩაწერა ინფიქსური ფორმით. თუმცა პარამერტებს შორის ფუნქციის სახელი აუცილებელია ჩაიწეროს სიმბოლოთი ` (შებრუნებული აპოსტროფი).

ახლად განსაზღვრული ინფიქსური ოპერაციისთვის შესაძლოა მოცემული იყოს გამოთვლის რიგი. ამისთვის `Haskell`-ში არის დარეზერვებული სიტყვა `infixr`, რომელიც განუსაზღვრავს ოპერაციას მის ნიშნადობას (შესრულების რიგს) ინტერვალში `0`-დან `9`-მდე. ამასთან, `9` არის ყველაზე მაღალნიშნადი ოპერაცია.

შევნიშნოთ, რომ `Haskell`-ში ყველა ფუნქცია არის არამკაცრი, ვინაიდან თითოეული მათგანი მხარს უჭერს გადატანილ გამოთვლებს. მაგალითად, თუ ფუნქცია ასეა განსაზღვრული:

```
bot   = bot
```

ასეთი ფუნქციის გამოძახება იწვევს შეცდომას და, ჩვეულებრივ, ასეთი შეცდომები ძნელი აღმოსაჩენია. საჭიროა განისაზღვროს კონსტანტური ფუნქცია, მაგალითად, ასე:

```
constant_1 x        = 1
```

მაშინ კონსტრუქციის (constant_1 bot) გამოძახებისას არანაირი შეცდომა არ მოხდება, რადგანაც ამ შემთხვევაში ფუნქცია bot არ გამოითვლება (გამოთვლა გადატანილია, გამოსახულება გამოითვლება მხოლოდ მაშინ, როცა ნამდვილად მოითხოვება). გამოთვლის შედეგი, რა თქმა უნდა, არის რიცხვი1.

სავარჯიშო №3

1. შეადგინეთ შემდეგი სასრული სიები (N — სიებში ელემენტების რაოდენობაა). ამისთვის გამოიყენეთ ან სიების გენერატორი, ან კონსტრუქტორი ფუნქციები:
   a. ნატურალური რიცხვების სია. N = 20.
   b. კენტი ნატურალური რიცხვების სია. N = 20.
   c. ლუწი ნატურალური რიცხვების სია. N = 20.
   d. ორის ხარისხების სია. N = 25.
   e. სამის ხარისხების სია. N = 25.
   f. ფერმას (Fermat's) სამკუთხა რიცხვების სია. N = 50.
   g. ფერმას (Fermat's) პირამიდალური რიცხვების სია. N = 50.
2. შეადგინეთ შემდეგი უსასრულო სიები. ამისთვის გამოიყენეთ ან სიების გენერატორი, ან კონსტრუქტორი ფუნქციები:
   a. ფაქტორიალების სია.
   b. ნატურალური რიცხვების კვადრატების სია.
   c. ნატურალური რიცხვების კუბების სია.
   d. ხუთიანის ხარისხების სია.
   e. ნატურალური რიცხვების მეორე სუპერხარისხების სია.

თავი 2.5. HASKELL-ის სინტაქსი და მოსამსახურე სიტყვები

ჩვენ უკვე აღვნიშნეთ, რომ Haskell ენის სინტაქსი ძალზე წააგავს აბსტრაქტული ფუნქციონალური ენის სინტაქსს. Haskell-ის მკვლევრებმა მიზნად დაისახეს იმ დროისთვის არსებული ფუნქციონალური ენების საუკეთესო თვისებების თავმოყრა ერთ ენაში და, ამავე დროს, ცუდის უარყოფა, რაც მათ მოახერხეს კიდეც.

ქვემოთ განვიხილავთ მოსამსახურე სიტყვებს, რომლებიც აქამდე არ გამოგვიყენებია. ასევე განვიხილავთ იმ ახალ შესაძლებლობებს, რომლებიც ფუნქციონალურ პარადიგმაში შევიდა იმის გამო, რომ Haskell-შია რეალიზებული.

დაცვა და ლოკალური ცვლადები

როგორც უკვე ვაჩვენეთ, დაცვა და ლოკალური ცვლადები გამოიყენება ფუნქციონალურ დაპროგრამებაში მხოლოდ ჩანაწერის გასამარტივებლად და ტექსტის გასაგებად. Haskell-ის სინტაქსში არის სპეციალური საშუალება, რაც უზრუნველყოფს დაცვის ორგანიზებასა და ლოკალური ცვლადების გამოყენებას.

ფუნქციის განსაზღვრისას დაცვის მექანიზმი მიეთითება ვერტიკალური ხაზის სიმბოლოს " | "-ის გამოყენებით:

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

ამ მაგალითში ფუნქცია `sign` იყენებს სამ დამცავ კონსტრუქციას, რომლებიდანაც თითოეული გამოიყოფა წინა განსაზღვრებისგან ვერტიკალური ხაზით. ასეთი კონსტრუქცია შეიძლება იყოს ნებისმიერი რაოდენობის. მათი გარჩევა განხორციელდება, ბუნებრივია, თანმიმდევრობით ზემოდან ქვემოთ და თუ არსებობს არაცარიელი გადაკვეთა დაცვის განსაზღვრებაში, მაშინ იმუშავებს ის კონსტრუქცია, რომელიც არის უფრო ადრე (მაღლა) ფუნქციის განსაზღვრების ჩანაწერში.

რათა გამარტივდეს პროგრამის დაწერა და გახდეს იგი უფრო წაკითხვადი და გასაგებად მარტივი იმ შემთხვევაში, როცა ფუნქციის განმარტება დაწერილია დიდი რაოდენობით კლოზების გამოყენებით, `Haskell`-ში არსებობს გასაღები სიტყვა „`case`“. ამ სიტყვის გამოყენებით შესაძლოა ფუნქციის განსაზღვრისას კლოზები არ ჩავწეროთ ისე, როგორც ეს „წმინდა“ ფუნქციონალურ ენაშია მიღებული, არამედ შევამციროთ ჩანაწერი. ფუნქციის განმარტების ზოგადი სახე გასაღები სიტყვა „`case`“-ის გამოყენებით შემდეგია:

```
Function X1 X2 ... Xk = case (X1, X2, ..., Xk) of
(P11, P21, ..., Pk1) -> Expression1
...
(P1n, P2n, ..., Pkn) -> Expressionn
```

ამრიგად, ფუნქცია, რომელიც აბრუნებს მოცემული სიის პირველ `n` ელემენტს, შეიძლება განისაზღვროს გასაღები სიტყვა „`case`“-ის გამოყენებით შემდეგნაირად:

```
takeFirst n l =      case (n, l) of
              (0, _) -> []
              (_, []) -> []
              (n, (x:xs)) -> (x) : (takeFirst
               (n - 1) xs)
```

ასეთი ჩანაწერი იქნება ფუნქციის ჩვეულებრივი განმარტების ეკვივალენტური:

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

განვმარტოთ ცნება „ჩასმის ნიღაბი“. Haskell-ში ნიღაბს აღნიშნავენ ქვედა ტირე სიმბოლოთი „ _ “, ისევე, როგორც Prolog-ში. ეს სიმბოლო ცვლის ნებისმიერ ნიმუშს და წარმოადგენს, თავის მხრივ, ანონიმურ ცვლადს. თუ კლოზის გამოსახულებაში არ არის აუცილებელი გამოვიყენოთ ნიმუშის ცვლადი, მაშინ შესაძლებელია იგი შეიცვალოს ჩასმის ნიღბით. ამასთან, ის გამოსახულება, რომელიც შეიძლება ჩაისვას ნიღბის ნაცვლად, არ გამოითვლება.

დამცავი კონსტრუქციების გამოყენების შემდეგი საშუალებაა კონსტრუქციის „if-then-else“-ის გამოყენება. Haskell-ში ეს შესაძლებლობა რეალიზებულია. ფორმალურად, ეს კონსტრუქცია შეიძლება მარტივად გადავიდეს გამოსახულებაში გასაღები სიტყვა „case“-ის გამოყენებით. შეიძლება ჩავთვალოთ, რომ გამოსახულება:

```
if Exp1 then Exp2 else Exp3
```

წარმოადგენს შემდეგი გამოსახულების შემოკლებას:

```
case   (Exp1) of
       (True) -> Exp2
       (False) -> Exp3
```

ბუნებრივია, რომ Exp1-ის ტიპი უნდა იყოს ლოგიკური, ხოლო გამოსახულებების - Exp2-ის და Exp3-ის ტიპები უნდა ემ-

თხვეოდეს ერთმანეთს (სწორედ მათი მნიშვნელობები უნდა დაბრუნდეს „`if-then-else`“ კონსტრუქციით).

ლოკალური ცვლადების გამოსაყენებლად `Haskell`-ში არსებობს ჩანაწერის ორი სახე. პირველი სრულად შეესაბამება უკვე განსაზღვრულ მათემატიკურ ნოტაციას:

```
let  y = a * b
     f x = (x + y) / y
in f c + f d
```

ლოკალური ცვლადის განსაზღვრის სხვა საშუალებაა მისი აღწერა გამოყენების შემდეგ. ამ შემთხვევაში გამოიყენება გასაღები სიტყვა „`where`“, რომელიც ჩაისმის გამოსახულების ბოლოს.

```
f x y| y > z = y - z
     | y == z       = 0
     | y < z = z - y
     where z = x * x
```

როგორც დავინახეთ, `Haskell` მხარს უჭერს ლოკალური ცვლადის განმარტების ორ საშუალებას – პრეფიქსულს (გასაღები სიტყვა „`let`“-ის საშუალებით) და პოსტფიქსულს (გასაღები სიტყვა „`where`“-ის საშუალებით). ორივე საშუალება თანაბარმნიშვნელოვანია, მათი გამოყენება მხოლოდ პროგრამისტის ჩვევაზეა დამოკიდებული. თუმცა, ჩვეულებრივ, პოსტფიქსური ჩანაწერი გამოიყენება გამოსახულებებში, სადაც არის დაცვა, მაშინ, როცა პრეფიქსული – ყველა დანარჩენ შემთხვევაში.

როგორც უკვე აღვნიშნეთ, დაპროგრამების ფუნქციონალური პარადიგმა მხარს უჭერს წმინდა ან პარამეტრიზებულ პოლიმორფიზმს. თუმცა თანამედროვე ენების უმრავლესობა მხარს უჭერს პოლიმორფიზმს ad hoc ანუ გადატვირთვას. მაგალითად, გადატვირთვა პრაქტიკულად მუდმივად გამოიყენება შემდეგი მიზნებისთვის:

✓ ლიტერალები 1, 2, 3 და ა.შ. (ანუ ციფრები) გამოიყენება როგორც მთელი რიცხვების ჩასაწერად, ასევე ნებისმიერი სიზუსტის რიცხვების ჩასაწერად.

✓ არითმეტიკული ოპერაციები (მაგალითად, შეკრება – ნიშანი „+“) გამოიყენება სხვადასხვა ტიპის მონაცემებთან (მათ შორის – არარიცხვითთანაც, მაგალითად, სტრიქონების კონკატენაციისთვის).

✓ შედარების ოპერაციები (Haskell-ში ორმაგი ტოლობის ნიშანი „==“) გამოიყენება სხვადასხვა ტიპის მონაცემების შესადარებლად.

ოპერაციის გადატვირთული მოქმედება განსხვავებულია სხვადასხვა ტიპისთვის, მაშინ, როცა პარამეტრიზებული პოლიმორფიზმისას მონაცემთა ტიპი არ არის მნიშვნელოვანი. Haskell-ში არის მექანიზმი გადატვირთის გამოსაყენებლად.

ad hoc პოლიმორფიზმის გამოყენების შესაძლებლობის განხილვა ყველაზე ადვილია შედარების ოპერაციის მაგალითზე. არსებობს ტიპების დიდი სიმრავლე, რომელთათვისაც შესაძლებელია და მიზანშეწონილია პოლიმორფიზმის გამოყენება, მაგრამ ზოგიერთი ტიპისთვის ეს ოპერაცია უსარგებლოა. მაგალითად, ფუნქციის შედარებები უაზრობაა, ფუნქცია აუცილებლად უნდა გამოითვალოს და შედარდეს ამ გამოთვლების შედეგები. თუმცა, მაგალი-

თად, შეიძლება აუცილებელი გახდეს სიების შედარება. ცხადია, აქ საუბარია სიების ელემენტების მნიშვნელობების შედარებაზე და არა მათი მიმთითებლების შედარებაზე (როგორც ეს გაკეთებულია Java-ში).

განვიხილოთ ფუნქცია element, რომელიც აბრუნებს ჭეშმარიტ მნიშვნელობას იმის შესაბამისად, არის თუ არა მოცემული ელემენტი მოცემულ სიაში. ფუნქცია აღიწერება ინფიქსური ფორმით.

```
x `element` []       = False
x `element` (y:ys)   = (x == y) || (x `element` ys)
```

აქ ჩანს, რომ ფუნქცია element-ს აქვს ტიპი (a -> [a] -> Bool), მაგრამ ოპერაცია „==“-ის ტიპი უნდა იყოს (a -> a -> Bool). ვინაიდან ტიპის ცვლადმა შეიძლება აღნიშნოს ნებისმიერი ტიპი (მათ შორის სიაც), მიზანშეწონილია გადავსაზღვროთ ოპერაცია „==“ ნებისმიერ ტიპთან სამუშაოდ.

ტიპების კლასები ამ პრობლემის გადაწყვეტას წარმოადგენს Haskell-ში. რათა განვიხილოთ ეს მექანიზმი მოქმედებაში, განვსაზღვროთ კლასის ტიპი, რომელიც შეიცავს ტოლობის ოპერატორს.

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

ამ კონსტრუქციაში გამოიყენება მოსამსახურე სიტყვები „class“ და „where“, ასევე ტიპის ცვლადი a. სიმბოლო „Eq“ წარმოადგენს განსაზღვრული კლასის სახელს. ეს ჩანაწერი შეიძლება ასე წავიკითხოთ: „ტიპი a არის Eq კლასის ეგზემპლარი, თუ ამ ტიპისთვის გადატვირთულია შესაბამისი ტიპის შედარების ოპერაცია „==“. აუცილებელია შევნიშნოთ, რომ შედარების

ოპერაცია უნდა იყოს განსაზღვრული ერთი და იმავე ტიპის ობიექტების წყვილზე.

იმ ფაქტის აღნიშვნა, რომ ტიპი `a` უნდა იყოს `Eq` კლასის ელემენტი, ჩაიწერება ასე `(Eq a)`. ამიტომაც ჩანაწერი `(Eq a)` არ წარმოადგენს ტიპის აღწერას, ის აღნიშნავს შეზღუდვას, რომელიც ედება `a` ტიპზე, და ამ შეზღუდვას `Haskell`-ში უწოდებენ *კონტექსტს*. კონტექსტები იწერება ტიპების განმარტების წინ და გამოიყოფა მისგან სიმბოლოებით „ => “:

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

ეს ჩანაწერი შეიძლება ასე წავიკითხოთ: „·”——,‘ `a` ტიპისთვის, რომელიც არის `Eq` კლასის ეგზემპლარი, განსაზღვრულია ოპერაცია „==“, რომელსაც აქვს ტიპი `(a -> a -> Bool)`“. ეს ოპერაცია უნდა გამოვიყენოთ ფუნქციაში `element`, ამიტომ შეზღუდვა ვრცელდება მასზეც. ამ შემთხვევაში აუცილებელია ცხადად მივუთითოთ ფუნქციის ტიპი:

```
element       :: (Eq a) => a -> [a] -> Bool
```

ამ ჩანაწერით ფიქსირდება განაცხადი, რომ ფუნქცია `element` განსაზღვრულია არა მონაცემთა ყველა ტიპისთვის, არამედ მხოლოდ მათთვის, რომელთათვისაც განსაზღვრულია შესაბამისი შედარების ოპერაცია.

თუმცა ახლა ჩნდება პრობლემა, განვსაზღვროთ, რომელი ტიპებია `Eq` კლასის ეგზემპლარები. ამისთვის არსებობს გასაღები სიტყვა „`instance`“. მაგალითად, იმის მისაწერად, რომ ტიპი `Integer` წარმოადგენს `Eq` კლასის ეგზემპლარს, აუცილებელია დავწეროთ:

```
instance Eq Integer where
     x == y  = x `integerEq` y
```

ამ გამოსახულებაში ოპერაცია „==“-ის განსაზღვრას უწოდებენ მეთოდის განსაზღვრას (როგორც ეს ობიექტორიენტირებულ პარადიგმაშია). ფუნქცია `integerEq` შეიძლება იყოს ნებისმიერი (და არა მხოლოდ ინფიქსური), მთავარია, მას ჰქონდეს ტიპი (`a a Bool`). ამ შემთხვევას ყველაზე მეტად მიესადაგება ორი ნატურალური რიცხვის შედარების პრიმიტიული ფუნქცია. თავის მხრივ, დაწერილი გამოსახულება შეიძლება ასე წავიკითხოთ: „ტიპი `Integer` წარმოადგენს `Eq` კლასის ეგზემპლარს, ხოლო შემდეგ მოდის მეთოდის აღწერა, რომელიც ადარებს ორ `Integer` ტიპის ობიექტს“.

ამრიგად, შესაძლოა შედარების ოპერაცია განისაზღვროს უსასრულო რეკურსიული ტიპებისთვისაც. მაგალითად, უკვე განხილული `Tree` ტიპისთვის განსაზღვრებას ექნება შემდეგი სახე:

```
instance (Eq a) => Eq (Tree a) where
     Leaf a == Leaf b                = a == b
     (Branch l1 r1) == (Branch l2 r2)= (l1 ==
                              l2) && (r1 == r2)
     _ == _                   = False
```

ბუნებრივია, თუ ენაში განსაზღვრულია კლასის ცნება, მაშინ უნდა იყოს განსაზღვრული მემკვიდრეობითობის კონცეფციაც. თუმცა `Haskell`-ში კლასის ქვეშ გვესმის უფრო აბსტრაქტული რამ, ვიდრე, ჩვეულებრივ, ობიექტორიენტირებულ ენებში. `Haskell`-ში ასევეა მემკვიდრეობითობა და მისი კონცეფცია ისევე დახვეწილად არის განმარტებული, როგორც კლასი. მაგალითად, ზემოთ განმარტებული `Eq` კლასიდან მემკვიდრეობით შეიძლება მივიღოთ კლასი

Ord, რომელშიც განსაზღვრულია მონაცემთა შედარების ოპერაციები. მისი განმარტება გამოიყურება შემდეგნაირად:

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

Ord კლასის ყველა ეგზემპლარი განსაზღვრავს, გარდა ოპერაციებისა „ნაკლებია“, „მეტია“, „ნაკლებია და ტოლია“, „მეტია და ტოლია“, „მინიმუმ“ და „მაქსიმუმ“, კიდევ შედარების ოპერაციას „==“, რადგანაც მისი კლასი Ord მემკვიდრეობითაა მიღებული Eq კლასიდან.

ყველაზე გასაოცარია ის, რომ Haskell მხარს უჭერს მრავლობით მემკვიდრეობას. თუ ვიყენებთ რამდენიმე ბაზურ კლასს, მათ უბრალოდ შესაბამის სექციაში ჩამოვთლით (მძიმეებით გამოვყოფთ). ამასთან, კლასის ეგზემპლარები, რომლებიც რამდენიმე ბაზური კლასიდან მემკვიდრეობით არის მიღებული, ცხადია მხარს უჭერს ბაზური კლასების ყველა მეთოდს.

შედარება სხვა ენებთან

კლასები, მართალია, დაპროგრამების მრავალ ენაში არსებობს, თუმცა Haskell-ში მისი ცნება რამდენადმე განსხვავებულია.

- ✓ Haskell-ში გაყოფილია კლასის განსაზღვრა მისი მეთოდების განსაზღვრისაგან, მაშინ, როცა, ისეთი ენები, როგორცაა C++ და Java ერთად განსაზღვრავს მონაცემთა სტრუქტურას და მისი დამუშავების მეთოდებს.

- ✓ მეთოდების განსაზღვრა Haskell-ში შეესაბამება ვირტუალურ ფუნქციებს C++-ში. კლასის თითოეულმა ეგზემპლარმა უნდა გადასაზღვროს კლასის მეთოდები.
- ✓ ყველაზე მეტად Haskell-ის კლასები წააგავს Java-ს ინტერფეისებს. როგორც ინტერფეისის განსაზღვრება, კლასებიც Haskell-ში წარმოადგენს ობიექტების გამოყენების ოქმებს თვითონ ობიექტების განსაზღვრის ნაცვლად.
- ✓ Haskell მხარს არ უჭერს ფუნქციების გადატვირთვას, რომელიც C++-შია გამოყენებული, როცა ერთი და იმავე სახელის ფუნქციები დასამუშავებლად ღებულობს სხვადასხვა ტიპის მონაცემებს.
- ✓ ობიექტების ტიპი Haskell-ში არ შეიძლება არაცხადად იყოს გამოყვანილი. Haskell-ში არ არსებობს ბაზური კლასი ყველა კლასისთვის (ისეთი, როგორიცაა, მაგალითად, TObject კლასი ენა Object Pascal-ში).
- ✓ C++ და Java კომპილირებულ კოდს უმატებს იდენტიფიცირებულ ინფორმაციას (მაგალითად, ვირტუალური ფუნქციების განლაგების ცხრილებს). Haskell-ში ასე არ არის. ინტერპრეტაციის (კომპილაციის) დროს ყველა ინფორმაცია გამოდის ლოგიკურად.
- ✓ არ არსებობს შეღწევადობაზე კონტროლის ცნება – არ არის ღია და დახურული მეთოდები. ამის ნაცვლად Haskell გვთავაზობს პროგრამების მოდულარიზაციის მექანიზმს.

სავარჯიშო №4

1. ჩაწერეთ Haskell-ის ნოტაციით ფუნქციები, რომლებიც მუშაობს სიებთან. შესაძლებლობისამებრ გამოიყენეთ დაცვისა და ლოკალური ცვლადების ფორმალიზმები:

a. `getN` — ფუნქცია მოცემული სიიდან `N`-ური ელემენტის გამოსაყოფად.
b. `listSumm` — ორი სიის შეკრების ფუნქცია. აბრუნებს სიას, რომელიც შედგება სია-პარამეტრების ელემენტების ჯამისგან. გაითვალისწინეთ, რომ გადაცემული სიები შეიძლება იყოს სხვადასხვა სიგრძის.
c. `oddEven` — მოცემულ სიაში მეზობელი კენტი და ლუწი ელემენტების გადაადგილების ფუნქცია.
d. `reverse` — ფუნქცია, რომელიც აბრუნებს სიას (სიის პირველი ელემენტი ხდება ბოლო ელემენტი, მეორე – ბოლოდან მეორე და ა.შ. ბოლო ელემენტამდე).
e. `map` — ფუნქცია მასზე პარამეტრად გადაცემულ მეორე ფუნქციას იყენებს მოცემული სიის ყველა ელემენტთან.

2. ჩაწერეთ `Haskell`-ის ნოტაციაში ფუნქციები, რომლებიც მუშაობს სიებთან. აუცილებლობის შემთხვევაში ისარგებლეთ დამატებითი, ასევე წინა სავარჯიშოში განსაზღვრული ფუნქციებით. შესაძლებლობების მიხედვით გამოიყენეთ დაცვისა და ლოკალური ცვლადების ფორმალიზმები:

a. `reverseAll` — ფუნქცია, რომელიც შესასვლელზე იღებს სიურ სტრუქტურას და შეაბრუნებს მის ყველა ელემენტს, ასევე თავის თავსაც.
b. `position` — ფუნქცია, რომელიც აბრუნებს ნომერს, თუ მოცემული ატომი სიაში პირველად როდის შევა.
c. `set` — ფუნქცია, რომელიც აბრუნებს სიას, რომელშიც მოცემული სიის ყველა ატომი მხოლოდ ერთხელ შედის.
d. `frequency` — ფუნქცია, რომელიც აბრუნებს წყვილების სიას (სიმბოლო, სიხშირე). თითოეული წყვილი განისაზღვრება მოცემული სიის ატომით და ამ სიაში მისი შესვლის სიხშირით.

3. აღწერეთ ტიპების შემდეგი კლასები. აუცილებლობის შემთხვევაში ისარგებლეთ კლასების მემკვიდრეობითობის მექანიზმით:

   a. `Show` — კლასი, რომლის ობიექტების ეგზემპლარები შეიძლება შევიტანოთ ეკრანიდან.
   b. `Number` — კლასი, რომელიც აღწერს სხვადასხვა ბუნების რიცხვებს.
   c. `String` — კლასი, რომელიც აღწერს სტრიქონებს (სიმბოლოების სიებს).

4. განსაზღვრეთ ტიპები – წინა დავალებაში აღწერილი კლასების ეგზემპლარები. შესაძლებლობების მიხედვით კლასის თითოეული ეგზემპლარისთვის განსაზღვრეთ მეთოდები, რომლებიც მუშაობენ ამ კლასის ობიექტებთან:

   a. `Integer` — მთელი რიცხვების ტიპი.
   b. `Real` — ნამდვილი რიცხვების ტიპი.
   c. `Complex` — კომპლექსური რიცხვების ტიპი.
   d. `WideString` — სტრიქონების ტიპი, როგორც ორბაიტიანი სიმბოლოების თანმიმდევრობა `UNICODE`-ის კოდირებაში.

თავი 2.6. მოდულები და მონადები HASKELL-ში

ენა არ არის მოდულის ცნების გარეშე, თუ არ ჩავთვლით ყველაზე დაბალი დონის ენებს და მათაც კი ამ ბოლო დროს შეიძინეს მაღალი დონის ენის თვისებები. მოდულის ცნება შემოვიდა იმ დროიდან, როცა დაპროგრამება, როგორც ხელოვნება (ან ხელობა), ჯერ ჯიდევ ვითარდებოდა. იგი გაჩნდა პროგრამის ლოგიკურ ნაწილებად დაყოფის აუცილებლობიდან, რომელთაგანაც თითოეულს ამუშავებდა ცალკეული მომხმარებელი ან მკვლევართა კოლექტივი.

Haskell-ში ასევე არსებობს მოდულის ცნება, თუმცა ამ ენაში უფრო მეტად მნიშვნელოვანია „მონადის“ ცნება. განვიხილოთ ორივე – „მოდული“ და „მონადა“.

მოდულები

Haskell-ში მოდულებს ორმაგი დანიშნულება აქვთ. ერთი მხრივ, მოდულები აუცილებელია სახელთა არის კონტროლისთვის (ისევე, როგორც ყველა სხვა ენაში); მეორე მხრივ, მოდულების საშუალებით შეიძლება შეიქმნას მონაცემთა აბსტრაქტული ტიპები.

მოდულის განმარტება Haskell-ში საკმაოდ მარტივია. მისი სახელი შეიძლება იყოს ნებისმიერი სიმბოლო, მხოლოდ სახელი იწყება დიდი ასოთი. მოდულის სახელი დამატებით არ არის დაკავშირებული ფაილურ სისტემასთან (ისე, როგორც ეს Pascal-ში და Java-შია), ანუ მოდულის შემცველი ფაილის სახელი შეიძლება

არ იყოს მოდულის სახელის მსგავსი. უფრო ზუსტად, ფაილში შეიძლება იყოს რამდენიმე მოდული, რადგანაც მოდული – ეს ყველაზე მაღალი დონის დეკლარაციაა.

როგორც ცნობილია, ზედა დონეზე `Haskell`-ის მოდულები შეიძლება შეიცავდეს დეკლარაციების სიმრავლეს (აღწერებსა და განსაზღვრებებს), როგორიცაა ტიპები, კლასები, მონაცემები, ფუნქციები. თუმცა დეკლარაციის ერთი სახე მოდულში უნდა იდგეს აუცილებლად პირველ ადგილას (თუ, რა თქმა უნდა, დეკლარაციის ეს სახე საერთოდ გამოიყენება). საქმე ეხება მოდულში სხვა მოდულის ჩართვას, რისთვისაც გამოიყენება მოსამსახურე სიტყვა `import`. დანარჩენი განსაზღვრებები შეიძლება იყოს ნებისმიერი თანმიმდევრობით.

მოდულის განმარტება იწყება მოსამსახურე სიტყვით `module`. მაგალითად, მოდული `Tree` შემდეგნაირად განიმარტება:

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

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

fringe            :: Tree a -> [a]
fringe (Leaf x)         = [x]
fringe (Branch left right)     = fringe left ++
                                 fringe right
```

ამ მოდულში აღწერილია ტიპი (`Tree`) და ფუნქცია (`fringe`). შევნიშნოთ, რომ ტიპის სახელი ემთხვევა მოდულის სახელს. ეს არ არის სახიფათო, რადგან ამ შემთხვევაში ისინი სახელთა სხვადასხვა არეში არიან. მოცემულ შემთხვევაში მოდული `Tree` ცხადად ექსპორტირებს ტიპს `Tree` (თავის ქვეტიპებთან, `Leaf`-თან და `Branch`-თან ერთად) და ფუნქცია `fringe`-ს. თუ რომელიმე ობიექტის დასახელებას ფრჩხილებში არ მოვუთითებთ, მაშინ მისი

ექსპორტირება არ მოხდება და მაშინ ეს ობიექტი არ გამოჩნდება მიმდინარე მოდულის გარედან.

ერთი მოდულის გამოყენება მეორიდან შესაძლებელია შემდეგნაირად:

```
module Main where
import Tree (Tree(Leaf, Branch), fringe)
main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

მოცემულ მაგალითში ჩანს, რომ მოდული `Main` იმპორტირებს მოდულს `Tree`. ამასთან, `import` დეკლარაციაში ცხადად არის აღწერილი, თუ რომელი ობიექტები იმპორტირდება მოდულიდან `Tree`. თუ ამ აღწერას გამოვტოვებთ, მაშინ იმპორტირდება ყველა ობიექტი, რომელსაც მოდული ექსპორტირებს, ანუ ამ შემთხვევაში შეგვეძლო უბრალოდ დაგვეწერა `: import Tree.`

ზოგჯერ ისე ხდება, რომ ერთი მოდული იმპორტირებს რამდენიმე სხვას (შევნიშნოთ, რომ ეს ჩვეულებრივი სიტუაციაა), მაგრამ, ამასთან, იმპორტირებულ მოდულში არსებობს ობიექტები ერთი და იმავე სახელებით. ბუნებრივია, რომ ამ შემთხვევაში ჩნდება სახელთა კონფლიქტი. ამის თავიდან ასაცილებლად, `Haskell`-ში არსებობს სპეციალური მოსამსახურე სიტყვა `qualified`, რომლის საშუალებითაც განისაზღვრება ის იმპორტირებული მოდულები, რომელთა ობიექტის სახელები იღებს სახეს: <მოდულის სახელი>-.<ობიექტის სახელი>, ანუ იმისათვის, რომ მივმართოთ მოდულიდან ობიექტს, მისი სახელის წინ აუცილებელია დავწეროთ მოდულის სახელი:

```
module Main where
import qualified Tree
main = print (Tree.fringe (Tree.Leaf 'a'))
```

ასეთი სინტაქსის გამოყენება პროგრამისტის გემოვნებაზეა, თუმცა ამით პროგრამის სიდიდე იზრდება. ზოგს მოსწონს მოკლე მნემონიკური სახელების გამოყენება და ისინი იყენებენ კვალიფიკატორებს (მოდულების სახელებს) მხოლოდ აუცილებლობის შემთხვევაში.

მონაცემთა აბსტრაქტული ტიპები

Haskell-ში მოდული ერთადერთი საშუალებაა ე.წ. მონაცემთა აბსტრაქტული ტიპების შესაქმნელად. აბსტრაქტულ ტიპებში დაფარულია ტიპის წარმოდგენა, ღიაა მხოლოდ სპეციფიკური ოპერაციები შექმნილ ტიპებზე, რომელთა სიმრავლე სრულიად საკმარისია ამ ტიპთან სამუშაოდ. მაგალითად, თუმცა ტიპი `Tree` საკმაოდ მარტივ ტიპს წარმოადგენს, მაინც უმჯობესია გავხადოთ იგი აბსტრაქტულ ტიპად, ანუ დავმალოთ, რომ ის შედგება `Leaf`-გან და `Branch`-გან. ეს ასე ხორციელდება:

```
module TreeADT (Tree, leaf, branch, cell,
left, right, isLeaf)  where

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

leaf                 = Leaf
branch               = Branch
cell (Leaf a)        = a
left (Branch l r)    = l
right (Branch l r)   = r
isLeaf (Leaf _)      = True
isLeaf _             = False
```

ჩანს, რომ შიდა ტიპ `Tree`-სთან პროგრამისტს შეუძლია მიაღწიოს მხოლოდ სპეციალური ფუნქციების გამოყენებით. აქედან გამომდინარე, თუ ამ მოდულის შემქმნელი შეეცდება შეცვალოს ტიპის წარმოდგენა (მაგალითად, მოახდინოს მისი ოპტიმიზაცია), მან უნდა შეცვალოს ის ფუნქციებიც, რომლებიც მოქმედებს `Tree` ტიპის ველებზე. თავის მხრივ, თუ პროგრამისტი გამოიყენებს თავის პროგრამაში ტიპს `Tree`, მას არაფრის შეცვლა არ მოუწევს, რადგანაც პროგრამა რჩება მოქმედი.

მოდულების გამოყენების სხვა ასპექტები

შემდგომ განხილული გვაქვს `Haskell`-ში მოდულების დამატებითი ასპექტები:

- ✓ იმპორტის დეკლარაციაში (`import`) შეიძლება ამორჩევით დავმალოთ ზოგიერთი სახელი ექსპორტირებული ობიექტიდან (მოსამსახურე სიტყვა `hiding`-ის საშუალებით). ეს სასარგებლოა იმპორტირებული მოდულიდან ზოგიერთი ობიექტის ცხადად ამოსაგდებად.
- ✓ იმპორტისას შეიძლება განვსაზღვროთ მოდულის ფსევდონიმი მისგან ექსპორტირებული ობიექტების კვალიფიკაციისათვის. ამისთვის გამოიყენება გასაღები სიტყვა `as`; გამოიყენება მოდულების სახელის შესამოკლებლადაც.
- ✓ ყველა პროგრამა არაცხადად იმპორტირებს მოდულს `Prelude`. თუ გავაკეთებთ ამ მოდულის ცხად იმპორტს, მაშინ მის დეკლარაციაში შესაძლებელია დაიფაროს ზოგიერთი ობიექტი, რათა მოხდეს შემდგომ მათი გადასაზღვრა.
- ✓ ყველა `instance` დეკლარაცია არაცხადად ექსპორტირდება და იმპორტირდება ყველა მოდულის მიერ.

- ✓ კლასის მეთოდები, ისევე როგორც კლასების ქვეტიპები, შეიძლება ჩამოვთვალოთ ფრჩხილებში, შესაბამისი კლასის სახელის შემდეგ, ექსპორტის/იმპორტის დეკლარაციის დროს.

მონადები

ფუნქციონალური დაპროგრამების ბევრი დამწყები პროგრამისტი შეცბუნებულია Haskell-ში მონადის ცნებით. მონადები ენაში ძალზე ხშირად გვხვდება. მაგალითად, შეტანა-გამოტანის სისტემა მონადის ცნებაზეა დაფუძნებული. ასევე სტანდარტული ბიბლიოთეკა შეიცავს მთელ რიგ მოდულებს, რომლებიც ეძღვნება მონადებს. აუცილებელია აღვნიშნოთ, რომ Haskell-ში მონადის ცნება ეფუძნება კატეგორიების თეორიას, თუმცა, რომ არ ჩავუღრმავდეთ აბსტრაქტულ მათემატიკას, შემდგომ წარმოვადგენთ მონადის ინტუიციურ გაგებას.

მონადები წარმოადგენენ ტიპებს, რომლებიც განსაზღვრავენ შემდეგი მონადური კლასებიდან ერთ-ერთის ეგზემპლარებს. ეს კლასებია: Functor, Monad და MonadPlus. არცერთი მათგანი არ შეიძლება იყოს სხვა კლასის წინაპარი, ანუ მონადური კლასიდან მემკვიდრეობითობა არ ხდება. მოდულში Prelude სამი მონადაა განსაზღვრული: IO, [] и Maybe, ანუ სია ასევე არის მონადა.

მათემატიკურად მონადა განისაზღვრება წესების ერთობლიობით, რომლებიც აკავშირებს მონადებზე მოქმედ ოპერაციებს. ეს წესები ინტუიციურად გვაგებინებს, როგორ უნდა იყოს გამოყენებული მონადა და როგორია მისი შინაგანი სტრუქტურა. დასაკონკრეტებლად განვიხილოთ კლასი Monad, რომელშიც განსაზღვრულია ორი ბაზური ოპერაცია და ერთი ფუნქცია:

```
class Monad m where
      (>>=)   :: m a -> (a -> m b) -> m b
      (>>)    :: m a -> m b -> m b
      return  :: a -> m a
      fail    :: String -> m a

      m >> k  = m >>= \_ -> k
```

ორი ოპერაცია `(>>=)` და `(>>)` — არის დაკავშირების ოპერაციები. ისინი ახდენენ ორი მონადური მნიშვნელობის კომბინაციას მაშინ, როცა ფუნქცია `return` გარდაქმნის გადაცემულ რომელიღაც `a` ტიპის მნიშვნელობას `m a` ტიპის მონადურ მნიშვნელობაში. ოპერაცია `(>>)`-ის სიგნატურა გვეხმარება გავიგოთ დაკავშირების ოპერაცია: გამოსახულება `(m a >>= \v m b)` კომბინირებს მონადურ მნიშვნელობას `m a`, რომელიც შეიცავს `a` ტიპის ობიექტს ფუნქციასთან, რომელიც ოპერირებს `v` ტიპის მნიშვნელობებით და აბრუნებს `m b` ტიპის შედეგს. კომბინაციის შედეგი კი იქნება `m b` ტიპის მონადური მნიშვნელობა. ოპერაცია `(>>)` გამოიყენება მაშინ, როცა ფუნქცია არ იყენებს მნიშვნელობებს, მიღებულს პირველი მონადური ოპერანდით.

დაკავშირების ოპერაციის ზუსტი მნიშვნელობა, რა თქმა უნდა, დამოკიდებულია მონადის კონკრეტულ რეალიზაციასთან. ასე, მაგალითად, ტიპი `IO` განსაზღვრავს ოპერაციას `(>>=)`, როგორც მისი ორი ოპერანდის თანმიმდევრულ შესრულებას. პირველი ოპერანდის შესრულების შედეგი გადაეცემა მეორეს. დანარჩენი ორი ჩადგმული მონადური ტიპისთვის (სიები და `Maybe`) ეს ოპერაცია განსაზღვრულია როგორც ნული ან მეტი პარამეტრის გადაცემა ერთი გამოთვლითი პროცესიდან მეორეზე.

`Haskell`-ში განსაზღვრულია სპეციალური მოსამსახურე სიტყვა, რომელიც ენის დონეზე მხარს უჭერს მონადების გამოყენე-

ბას. ეს არის სიტყვა `do`, რომლის აზრიც შეიძლება გავიგოთ მისი გამოყენების შემდეგი წესების მიხედვით:

```
do e1 ; e2            = e1 >> e2
do p <- e1 ; e2       = e1 >>= \p -> e2
```

პირველი გამოსახულება სრულდება ყოველთვის (მნიშვნელობების გადატანა პირველი ოპერანდიდან მეორეში არ ხდება). მეორე გამოსახულებაში შეიძლება იყოს შეცდომა, ამ დროს ხდება `fail` ფუნქციის გამოძახება, რომელიც ასევე განსაზღვრულია კლასში `Monad`. ამიტომაც მოსამსახურე სიტყვა `do` – უფრო ზუსტი განსაზღვრება მეორე შემთხვევისთვის ასე გამოიყურება:

```
do p <- e1 ; e2       = e1 >>= (\v -> case v of
                               p -> e2
                               _ -> fail "s")
```

სადაც `s` – ეს არის სტრიქონი, რომელსაც შეუძლია განსაზღვროს ოპერატორი `do`-ს ადგილმდებარეობა პროგრამაში, `a`-ს შეიძლება ჰქონდეს რაიმე სემანტიკური დატვირთვა. მაგალითად, `IO` მონადაში მოქმედება `('a' getChar)` იძახებს ფუნქციას `fail` იმ შემთხვევაში, თუ წაკითხული სიმბოლო არ არის სიმბოლო `'a'`. ეს მოქმედება წყვეტს პროგრამის შესრულებას, რადგან `IO` მონადაში განსაზღვრული ფუნქცია `fail` თავის მხრივ იძახებს სისტემურ ფუნქციას `error`.

კლასი `MonadPlus` გამოიყენება ისეთი მონადისათვის, რომელშიც არის ნულოვანი ელემენტი და ოპერაცია `"+"`. ამ კლასის აღწერა შემდეგნაირად გამოიყურება:

```
class (Monad m) => MonadPlus m where
      mzero   :: m a
      mplus   :: m a -> m a -> m a
```

მონადის ამ კლასში ნულოვანი ელემენტი ექვემდებარება შემდეგ წესებს:

```
m >>= \x -> mzero    = mzero
mzero >>= m  = mzero
```

მაგალითად, სიებისთვის ნულოვანი ელემენტი არის ცარიელი სია `[]`, ხოლო ოპერაცია „+“ - სიების კონკატენაცია. ამიტომაც, სიების მონადა წარმოადგენს კლასი `MonadPlus`-ის ეგზემპლარს. მეორე მხრივ, მონადა `IO`-ს არ აქვს ნულოვანი ელემენტი, ამიტომ იგი არის მხოლოდ `Monad`-ის კლასის ეგზემპლარი.

ჩადგმული მონადები

მიღებული ცნობების კონკრეტიზაციისათვის აუცილებელია განვიხილოთ მაგალითი. რადგანაც სიები არის მონადები და, ამავე დროს, შესწავლილია საკმაოდ დეტალურად, ამიტომ ისინი წარმოადგენენ საუკეთესო მასალას მონადის მექანიზმის პრაქტიკული გამოყენების განხილვისათვის.

სიებისთვის დაკავშირების ოპერაციას აკისრია ფუნქცია, გააერთიანოს იმ ოპერაციების ერთობლიობა, რომლებიც სრულდება სიის თითოეულ წევრთან. სიებთან გამოყენებისას ოპერაცია `(>>=)`-ის სიგნატურა იღებს შემდეგ სახეს:

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

ეს აღნიშნავს, რომ მოცემულია ტიპის მნიშვნელობისა და ფუნქციისგან შემდგარი სია, რომელიც დაიყვანს `a` ტიპის მნიშვნელობას `b` ტიპის მნიშვნელობების სიაზე. დაკავშირება იყენებს ფუნქციას მოცემული `a` ტიპის მნიშვნელობების სიის თითოეულ ელემენ-

ტზე და აბრუნებს მიღებულ სიას b ტიპის მნიშვნელობებით. ეს ოპერაცია უკვე ცნობილია – სიების განსაზღვრა სწორედ ასე მუშაობს. ამრიგად, შემდეგი სამი გამოსახულება აბსოლუტურად ერთნაირია:

```
-- გამოსახულება 1 --------------------------------
[(x, y) | x <- [1, 2, 3], y <- [1, 2, 3], x /=
y]

-- გამოსახულება 2 ------------------------------
do      x <- [1, 2, 3]
        y <- [1, 2, 3]
        True <- return (x /= y)
        return (x, y)

-- გამოსახულება 3 ------------------------------
[1, 2, 3] >>= (\x -> [1, 2, 3] >>= (\y -> return
(x /= y) >>=
        (\r -> case r of
                True    -> return (x, y)
                _       -> fail "")))
```

რომელ გამოსახულებას გამოიყენებს პროგრამის დაწერისას, ეს პროგრამისტის გადასაწყვეტია.

სავარჯიშო №5

1. რომელი ინტუიციური ცნებები შეიძლება ჩაიდოს ცნება „მონადაში“?
2. რა პრაქტიკული აზრი აქვს მონადის გამოყენებას ფუნქციონალურ დაპროგრამებაში?

სარჩევზე დაბრუნება

თავი 2.7. შეტანა-გამოტანის ოპერაციები HASKELL-ში

Haskell-ში, როგორც დაპროგრამების სხვა ენებში, არსებობს შეტანა-გამოტანის ოპერაციების ჩადგმული სისტემა, რომელიც სრულად უჭერს მხარს დაპროგრამების ფუნქციონალურ პარადიგმას.

ჩვენ უკვე ვიცით, რომ შეტანა-გამოტანის ოპერაციები აგებულია ენა Haskell-ის ისეთი ცნებისგან, როგორიცაა მონადა. ამავე დროს Haskell-ში შეტანა-გამოტანის სისტემის გასაგებად არ არის აუცილებელი გესმოდეთ ცნება *მონადის* თეორიული საფუძვლები. მონადები შეიძლება განვიხილოთ როგორც კონცეპტუალური ჩარჩოები, რომლებშიც მოთავსებულია შეტანა-გამოტანის სისტემა. შეიძლება ითქვას, რომ კატეგორიების თეორიის გაგება შეტანა-გამოტანის სისტემის ოპერაციების გამოყენებისათვის ისევე აუცილებელია, როგორც ჯგუფების თეორიის გაგება არითმეტიკული ოპერაციების შესასრულებლად.

შეტანა-გამოტანის ოპერაცია ნებისმიერ ენაში ეფუძნება მოქმედების ცნებას. მოქმედება შეიძლება იყოს მარტივი ან შედგენილი სხვა მოქმედებების თანმიმდევრობისაგან. მონადა IO შეიცავს ოპერაციებს, რომლებიც იძლევა საშუალებას შეიქმნას რთული მოქმედებები მარტივისგან ანუ მონადა ამ შემთხვევაში შეიძლება განვიხილოთ როგორც წებო, რომელიც შეკრავს მოქმედებებს პროგრამაში.

შეტანა-გამოტანის ყოველი მოქმედება აბრუნებს რაღაც მნიშვნელობებს. იმისათვის, რომ ეს მნიშვნელობები ბაზურისაგან განსხვავდებოდეს, ამ მნიშვნელობების ტიპები თითქოსდა შეხვეულია `IO` ტიპით (აუცილებელია გვახსოვდეს, რომ მონადა წარმოადგენს კონტეინერულ ტიპს). მაგალითად, ფუნქცია `getChar`-ის ტიპი ასეთია:

```
getChar      :: IO Char
```

ამ მაგალითში ნაჩვენებია, რომ ფუნქცია `getChar` ასრულებს რაღაც მოქმედებებს, რომელიც აბრუნებს `Char` ტიპის მნიშვნელობას. მოქმედებებს, რომლებიც არაფერ საინტერესოს არ აბრუნებენ, აქვთ ტიპი `IO ()`. ანუ სიმბოლო `()` აღნიშნავს ცარიელ ტიპს (სხვა ენებში `void`). ასე რომ, ფუნქცია `putChar` აქვს ტიპი:

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

ერთმანეთს მოქმედებები დაკავშირების ოპერატორის საშუალებით უკავშირდება. ანუ სიმბოლოები `>>=` აგებენ მოქმედებების თანმიმდევრობას. როგორც ცნობილია, ამ ფუნქციის ნაცვლად შეიძლება გამოვიყენოთ მოსამსახურე სიტყვა `do`, რომლის საშუალებით შეიძლება დააკავშიროთ ფუნქციის გამოძახებები, მონაცემების განსაზღვრა (სიმბოლო „ `<-` “ ის საშუალებით) და ლოკალური ცვლადების განმარტებების სიმრავლე (მოსამსახურე სიტყვა `let`).

მაგალითად, პროგრამა, რომელიც კითხულობს სიმბოლოს კლავიატურიდან და გამოაქვს ეკრანზე, შეიძლება ასე განვსაზღვროთ:

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

ამ მაგალითში ფუნქციის სახელად სიტყვა `main` შემთხვევით არ არის ამორჩეული. `Haskell`-ში, ისევე როგორც `C/C++` ენაში ფუნქციის სახელი `main` გამოიყენება პროგრამაში შესასვლელი წერტილის აღსანიშნავად. ამასთან, `Haskell`-ში `main` ფუნქციის ტიპი უნდა იყოს მონადა `IO`-ს ტიპი (ჩვეულებრივ გამოიყენება `IO ()`). გარდა ამ ყველაფრისა, შესასვლელი წერტილი ფუნქცია `main`-ის სახით უნდა იყოს განსაზღვრული მოდულში სახელით `Main`.

ვთქვათ, არის ფუნქცია `ready`, რომელმაც უნდა დააბრუნოს `True`, თუ დაჭერილია კლავიშა „`y`“ და `False` – დანარჩენ შემთხვევებში არ შეიძლება უბრალოდ დაწეროთ:

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

ვინაიდან ამ შემთხვევაში შედარების ოპერაციის შედეგი იქნება `Bool` ტიპის მნიშვნელობა და არა `IO Bool`. იმისათვის, რომ მონადური მნიშვნელობა დავაბრუნოთ, არსებობს სპეციალური ფუნქცია `return`, რომელიც მონაცემთა მარტივი ტიპიდან შექმნის მონადურს. ანუ, წინა მაგალითში ბოლო სტრიქონი, სადაც განმარტებულია ფუნქცია `ready`, ასე უნდა გამოიყურებოდეს „`return (c == 'y')`“.

შემდეგ მაგალითში განმარტებულია უფრო რთული ფუნქცია, რომელიც კითხულობს სიმბოლოების სტრიქონს კლავიატურიდან:

მაგალითად, განვიხილოთ ფუნქცია `getLine`.

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

აუცილებელია გვახსოვდეს, რომ იმ მომენტში, როცა პროგრამისტი გადადის მოქმედებების სამყაროში (შეტანა-გამოტანის ოპერაციის გამოყენებით), უკან დასაბრუნებელი გზა არ არის. ანუ, თუ ფუნქცია არ იყენებს მონადურ ტიპს, მას არ შეუძლია შეტანა-გამოტანის განხორციელება და პირიქით, თუ ფუნქცია აბრუნებს მონადურ ტიპ `IO`-ს, მაშინ ის უნდა დაექვემდებაროს მოქმედებების პარადიგმას `Haskell`-ში.

დაპროგრამება მოქმედებების საშუალებით

`Haskell`-ის ტერმინებში შეტანა-გამოტანის მოქმედებები ჩვეულებრივ მნიშვნელობებს წარმოადგენს. ანუ, მოქმედება შეიძლება გადავცეთ ფუნქციას პარამეტრად, ჩავრთოთ მონაცემთა სტრუქტურაში და, საერთოდ, გამოვიყენოთ იქ, სადაც შეიძლება `Haskell`-ის მონაცემების გამოყენება. ამ აზრით შეტანა-გამოტანის ოპერაციების სისტემა წარმოადგენს სრულად ფუნქციონალურს. მაგალითად, შეიძლება ჩამოიწეროს მოქმედებების სია:

```
todoList     :: [IO ()]
todoList     = [putChar 'a',
                do putChar 'b'
                   putChar 'c',
                do c <- getChar
                   putChar c]
```

ეს სია არ იწვევს არანაირ მოქმედებას, ის უბრალოდ შეიცავს მათ აღწერას. მაგრამ, რომ შევასრულოთ ეს სტრუქტურა, ანუ გამოვიწვიოთ ყველა მისი მოქმედება, აუცილებელია რომელიღაც ფუნქცია (მაგალითად, `sequence_`) :

```
sequence_     :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (a:as)     = do    a
                         sequence as
```

ეს ფუნქცია იქნება სასარგებლო ფუნქცია `putStr`-ის დაწერისას, რომელსაც გამოაქვს სტრიქონი ეკრანზე:

```
putStr        :: String -> IO ()
putStr s      = sequence_ (map putChar s)
```

ამ მაგალითიდან ცხადად ჩანს განსხვავება ენა `Haskell`-ის შეტანა-გამოტანის სისტემისა იმპერატიული ენების სისტემისაგან. თუ რომელიმე იმპერატიულ ენაში იქნებოდა ფუნქცია `map`, ის შეასრულებდა უამრავ მოქმედებას. სანაცვლოდ, `Haskell`-ში უბრალოდ იქმნება მოქმედებების სია (სტრიქონის თითოეული სიმბოლოსთვის ერთი), რომელიც შემდეგ შესასრულებლად მუშავდება ფუნქციით `sequence_` .

გამონაკლისი სიტუაციების დამუშავება

რა უნდა გავაკეთოთ, თუ შეტანა-გამოტანის ოპერაციის შესრულების პროცესში წარმოიშვა არაორდინარული სიტუაცია? მაგალითად, ფუნქცია `getChar`-მა აღმოაჩინა ფაილის ბოლო. ამ დროს ფიქსირდება შეცდომა. როგორც ყველა განვითარებული დაპროგრამების ენა, `Haskell`-იც ასეთი მიზნებისთვის გვთავაზობს გამო-

ნაკლისი სიტუაციების დამუშავების მექანიზმს. ამისთვის არ გამოიყენება სპეციალური სინტაქსი, მაგრამ არის სპეციალური ტიპი `IOError`, რომელიც შეიცავს შეტანა-გამოტანის პროცესში წარმოშობილ ყველა შეცდომას.

გამონაკლისი სიტუაციების დამუშავებას ასეთი სახე აქვს - `(IOError -> IO a)`. ამასთან, ფუნქცია `catch` ასოცირებს (აკავშირებს) გამონაკლის სიტუაციას მოქმედებათა ერთობლიობასთან.

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

ამ ფუნქციის არგუმენტს წარმოადგენს მოქმედება (პირველი არგუმენტი) და გამონაკლისი სიტუაციის დამუშავება (მეორე არგუმენტი). თუ მოქმედება წარმატებით შესრულდა, მაშინ უბრალოდ ბრუნდება შედეგი გამონაკლისი სიტუაციების დამუშავების აღგზნების გარეშე. თუ მოქმედების შესრულების პროცესში აღმოჩნდა შეცდომა, მაშინ იგი გადაეცემა დამმუშავებელს `IOError` ტიპის ოპერანდის სახით, რის შემდეგაც მუშაობს თვითონ დამამუშავებელი.

ამრიგად, შეიძლება დავწეროთ უფრო რთული ფუნქციები, რომლებშიც გათვალისწინებული იქნება შეცდომები:

```
getChar'    :: IO Char
getChar'    = getChar `catch` eofHandler
            where eofHandler e = if isEofError e
            then return \'n\ else ioError e

getLine'    :: IO String
getLine'    = catch getLine'' (\err -> return
            ("Error: " ++ show err))
            where getLine'' = do c <- getChar'
                      id c == '\n' then return ""
                           else do l <-  getLine'
                       return (c : l)
```

ამ პროგრამიდან ჩანს, რომ შეიძლება ერთმანეთში ჩაიდოს შეცდომების დამამუშავებლები. ფუნქცია `getChar'`-ში ხდება შეცდომის გამოჭერა, რომელიც გაჩნდა სიმბოლოს `'\n'` აღმოჩენისას. თუ შეცდომა სხვაა, მაშინ ფუნქცია `ioError`-ის დახმარებით ის მიემართება უფრო შორს და მას დაიჭერს დამმუშავებელი, რომელიც განსაზღვრულია `getLin` ფუნქციაში.

`Haskell`-ში გათვალისწინებულია აგრეთვე გამონაკლისი სიტუაციების დამუშავება *შეთანხმების პრინციპით,* რომელიც ყველაზე ზედა დონეზეა. თუ შეცდომა არ იქნა გამოჭერილი არცერთი დამამუშავებლის მიერ, რომელიც პროგრამაშია განსაზღვრული, მაშინ მას გამოიჭერს დამამუშავებელი შეთანხმებით, რომელიც ეკრანზე გამოიტანს შეტყობინებას შეცდომის შესახებ და გააჩერებს პროგრამას.

ფაილები, არხები და დამამუშავებლები

ფაილებთან სამუშაოდ `Haskell` გვთავაზობს ყველა იმ შესაძლებლობას, რასაც დაპროგრამების სხვა ენები. თუმცა ამ შესაძლებლობების უმრავლესობა განსაზღვრულია მოდულში `IO` და არა `Prelude`-ში, ამიტომ ფაილებთან სამუშაოდ აუცილებელია ცხადად გავუკეთოთ იმპორტი მოდულს `IO`.

ფაილის გახსნას ახდენს დამამუშავებელი (მას აქვს ტიპი `Handle`). დამამუშავებლის დახურვა ინიცირებს ფაილის დახურვას. დამამუშავებელი შეიძლება ასოცირდეს არხთან, ანუ ურთიერთქმედების პორტებთან, რომლებიც აკავშირებს პირდაპირ ფაილთან. ჰასკელში არის სამი ასეთი არხი – `stdin` (შეტანის სტანდარტული არხი, `stdout` (გამოტანის სტანდარტული არხი) და `stderr` (სტანდარტული არხი შეცდომების შესახებ შეტყობინებების გამოსატანად).

ამრიგად, ფაილების გამოსაყენებლად შეიძლება ვისარგებლოთ შემდეგი საშუალებებით:

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

შემდგომ მოყვანილია პროგრამის მაგალითი, რომელიც აკოპირებს ერთ ფაილს მეორეში:

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

getAndOpenFile :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
    do putStr prompt
       name <- getLine
       catch (openFile name mode)
             (\_ -> do putStrLn ("Cannot open
             "++ name ++ "\n")
                    getAndOpenFile prompt mode)
```

აქ გამოიყენება ერთი საინტერესო და სასარგებლო ფუნქცია – `hGetContents`, რომელიც იღებს მასზე არგუმენტად გადაცემული ფაილის შინაარსს და აბრუნებს მას ერთი გრძელი სტრიქონის სახით.

შენიშვნები

გამოდის, რომ `Haskell`-ში თავიდან აღმოაჩინეს იმპერატიული დაპროგრამება...

რაღაც აზრით, დიახ ასეა. `Haskell`-ში მონადა `IO` არის ჩადგმული პატარა იმპერატიული ქვეენა, რომლის საშუალებით შესაძლებელია შეტანა-გამოტანის ოპერაციების შესრულება. ამ ქვეენაზე პროგრამის დაწერა წააგავს ჩვეულებრივი აზრით იმპერატიულ ენებზე დაწერას. მაგრამ არსებობს არსებითი განსხვავებაც: `Haskell`-ში არ არის სპეციალური სინტაქსი პროგრამულ კოდში იმპერატიული ფუნქციის შეტანისა, ყველაფერი სრულდება ფუნქციონალური პარადიგმის დონეზე. ამავე დროს, გამოცდილ პროგრამისტებს მინიმუმამდე დაყავთ იმპერატიული კოდი, იყენებენ რა `IO` მონადას მხოლოდ თავისი პროგრამების ზედა დონეზე, რადგანაც `Haskell`-ში იმპერატიული და ფუნქციონალური სამყაროები მკვეთრად განსხვავდება ერთმანეთისგან. `Haskell`-გან განსხვავებით, იმ იმპერატიულ ენებში, რომლებშიც არის ფუნქციონალური ქვეენები, არ არის მკვეთრი განსაზღვრა ამ სამყაროებს შორის.

თავი 2.8. ფუნქციების კონსტრუირება

კონსტრუირებისთვის გამოიყენება სხვადასხვა ფორმალიზმი. ერთ-ერთია ე.წ. სინტაქსურად ორიენტირებული კონსტრუქცია. მის გამოსაყენებლად გავეცნოთ მეთოდს, რომელიც თავის დროზე შემოგვთავაზა ჰოარმა.

ქვემოთ გავეცნოთ მეტაენას, რომელიც გამოიყენება მონაცემთა სტრუქტურის აღსაწერად (აბსტრაქტულ სინტაქსში):

1. დეკარტული ნამრავლი: თუ $C_1, \ldots, C_n$- ტიპებია, ხოლო C – ტიპია, რომელიც შედგება შემდეგი სახის n–ური სიმრავლისგან: $<c_1, \ldots, c_n>$, $c_i \to C_i$, $i = 1,n$, მაშინ ამბობენ, რომ C - დეკარტული ნამრავლია $C_1, \ldots, C_n$ ტიპების და აღნიშნავენ ასე: $C = C_1 \times \ldots \times C_n$. ამასთან, იგულისხმება, რომ განსაზღვრულია C ტიპისთვის სელექტორები $s_1, \ldots, s_n$, რაც ჩაიწერება როგორც $s_1, \ldots, s_n$ = selectors C.

ასეთივე სახით ჩაიწერება კონსტრუქტორი g: g = constructor C. კონსტრუქტორი - ეს ფუნქციაა, რომელსაც აქვს ტიპი $(C_1 \to \ldots (C_n \to C) \ldots)$, ანუ $c_i \to$ Ci-თვის , $i = 1,n$: g c1 ... cn = <c1, ..., cn>.

ჩავთვალოთ, რომ სამართლიანია შემდეგი ტოლობა:

$$Ax \to C : \text{constructor } C\ (s_1, x) \ldots (s_n, x) = x$$

ამ ტოლობას უწოდებენ ტექტონიკურობის აქსიომას. ზოგჯერ ამ აქსიომას შემდეგნაირად წერენ:

$$s_i\ (\text{constructor } C\ c_1 \ldots c_n) = c_i$$

2. სპეციალური გაერთიანება: თუ $C_1, \ldots, C_n$ – ტიპებია, ხოლო C – ტიპია, რომელიც შედგება $C_1, \ldots, C_n$, ტიპე-

ბის გაერთიანებისგან „სპეციალურობის“ პირობის შესრულების პირობებში, მაშინ C-ს C_1, ..., C_n ტიპებს უწოდებენ სპეციალურ გაერთიანებას. ეს ფაქტი აღინიშნება ასე: C = C_1 + ... + C_n. სპეციალურობის პირობა ნიშნავს, რომ, თუ C-დან ავიღებთ რომელიღაც ელემენტს c_i, მაშინ ცალსახად განისაზღვრება ამ ელემენტის ტიპი C_i. სპეციალურობა შეიძლება განვსაზღვროთ პრედიკატებით P_1, ..., P_n ისეთებით, რომ:

(x -> C) & (x -> C_i) ==> (P_i x = 1) & (Aj =/= i : P_j x = 0)

სპეციალური გაერთიანება უზრუნველყოფს ასეთი ელემენტების არსებობას. ეს ფაქტი მიეთითება შემდეგი ჩანაწერით: P_1, ..., P_n = predicates C. არსებობს ასევე ტიპის ნაწილები, რომლებიც ასე აღინიშნება: N_1, ..., N_n = parts C.

როგორც ვხედავთ, მეტაენაში გამოიყენება ტიპების ორი კონსტრუქტორი - : და +. შემდეგ განვიხილავთ ახალი ტიპის განსაზღვრის რამდენიმე მაგალითს.

განვიხილოთ მაგალითი – ტიპი List (A)-ს ფორმალური აღწერა.

```
List (A) = NIL + (A x List (A))
null, nonnull = predicates List (A)
NIL, nonNIL = parts List (A)
head, tail = selectors List (A)
prefix = constructor List (A)
```

თუ შევხედავთ ტიპის ამ აღწერას (უფრო განსაზღვრებას), შეგვიძლია აღვწეროთ ფუნქციის გარეგნული სახე, რომელიც დაამუშავებს List (A) ტიპის სტრუქტურას:

თითოეული ფუნქცია უნდა შეიცავდეს მინიმუმ ორ კლოზს, პირველი ამუშავებს NIL-ს, მეორე – nonNIL-ს, შესაბამისად. List (A) ტიპის ამ ორ ნაწილს აბსტრაქტულ ჩანაწერში შეესაბამება სელექტორები [] და (H : T).

განვიხილოთ მაგალითი – List_str (A) ტიპის ფორმალური აღწერა:

```
List_str (A) = A + List (List_str (A))
atom, nonAtom = predicates List_str (A)
```

ფუნქციებს List_str (A) უნდა ჰქონდეთ, უკიდურეს შემთხვევაში, შემდეგი კლოზები მაინც:

```
1°. A -> when (atom (A))
2°. [] -> when (null (L))
3°. (H : T) -> head (L), tail (L)
    3.1°. atom (head (L))
    3.2°. nonAtom (head (L))
```

მაგალითი – მონიშნული წვეროებით ხეებისა და ტყეების ფორმალური აღწერა.

```
Tree (A) = A x Forest (A)
Forest (A) = List (Tree (A))
root, listing = selectors Tree (A)
ctree = constructor Tree (A)
```

მაგალითი – მონიშნული წვეროებითა და გვერდებით ხეების ფორმალური აღწერა.

```
MTree (A, B) = A x MForest (A, B)
MForest (A, B) = List (Element (A, B))
Element (A, B) = B x MTree (A, B)
mroot, mlist = selectors MTree (A, B)
null, nonNull = predicates MForest (A, B)
arc, mtree = selectors Element (A, B)
```

მტკიცდება, რომ ნებისმიერი ფუნქცია, რომელიც მუშაობს ტიპთან `MTree (A, B)`, საკმარისია შეიცავდეს მხოლოდ ჩამოთვლილ ექვს ოპერაციას იმისდა მიუხედავად, თუ როგორ არის ფუნქცია რეალიზებული. ეს დებულება შეიძლება შემოწმდეს დიაგრამის საშუალებით (უფრო სწორად, ეს ჰიპერგრაფია), რომელზეც ნათლად ჩანს, რომ ტიპი `MTree (A, B)`-ის ნებისმიერ ნაწილთან შეიძლება „მიღწევა“ მხოლოდ ამ ექვსი ოპერაციის გამოყენებით.

ფუნქციის კონსტრუირებისთვის, რომელიც ამუშავებს `MTree` მონაცემთა სტრუქტურას, აუცილებელია შემოვიტანოთ რამდენიმე დამატებითი ცნება და აღნიშვნები. საწყისი წვერო, წვერო `MForest` და წვერო `MTree` (რომელიც გამოდის `Element`-დან) აღინიშნება, შესაბამისად, როგორც `S0`, `S1` და `S2`. ამ წვეროების დასამუშავებლად აუცილებელია სამი ფუნქცია – `f0`, `f1` და `f2`, ამასთან, `f0` – საწყისი ფუნქციაა, დანარჩენი ორი კი – რეკურსიული.

`f0`-ის კონსტრუირება მარტივია – ამ ფუნქციას ერთი პარამეტრი აქვს `T`, რომელიც შეესაბამება საწყის წვეროს, `S0`-ს. შემდეგი ორი ფუნქცია კი რთულად კონსტრუირდება.

ფუნქცია `f1` იღებს შემდეგ პარამეტრებს:

`A` – წვეროს ჭდე;

`K` – პარამეტრი, რომელიც შეიცავს ხის გადახედილი ნაწილის დამუშავების შედეგს.

`L` – ტყე, რომლის დამუშავებაც აუცილებელია.

```
f1 A K L = g1 A K when null L
f1 A K L = f1 A (g2 (f2 A (arc (head L)) (mtree
(tail L)) K) A (arc L) K) (tail L) otherwise
```

ეს ფუნქცია რეალიზებს ხის გადახედვის რეჟიმს „თავიდან სიღრმისკენ“.

ფუნქცია `f2` იღებს შემდეგ პარამეტრებს (და ეს უკვე ცხადია მისი გამოძახებიდან `f1` ფუნქციის მეორე კლოზში):

`A` – წვეროს ჭდე;

`B` – ფერდის ჭდე;

`T` – ქვეხე დამუშავებისთვის;

`K` – ხის გადახედილი ნაწილის დამუშავების შედეგი.

```
f2 A B T K = f1 (mroot T) (g3 A B K) (mlist T)
```

აუცილებელია აღვნიშნოთ, რომ ეს არის ფუნქციის ზოგადი სახე `MTree` მონაცემთა სტრუქტურის დასამუშავებლად. დამატებითი `g1`, `g2` და `g3` ფუნქციების რეალიზება დამოკიდებულია კონკრეტულ ამოცანაზე. ახლა შეიძლება ავაგოთ `f0` ფუნქციის ზოგადი სახე:

```
f0 T = f1 (root T) k (mlist T)
```

სადაც `k – K` პარამეტრის საწყისი მნიშვნელობაა.

ფუნქციის კონსტრუირების მეთოდიკის უფრო ღრმა გაგებისთვის განვიხილოთ `B`-ხეებთან მუშაობის ფუნქციის კონკრეტული რეალიზაცია. დავუშვათ `BTree` მონაცემთა სტრუქტურისთვის არსებობს საბაზისო ოპერაციების ერთობლიობა, ხოლო თვითონ ხე წარმოდგენილია სიების სახით (წარმოდგენას არ აქვს განსაკუთრებული მნიშვნელობა). ბაზისური ოპერაციები შემდეგია:

```
1°. cbtree A Left Right = [A, Left, Right]
2°. ctree = []
3°. root T = head T
4°. left T = head (tail T)
```

```
5°. right T = head (tail (tail T))
6°. empty T = (T == [])
```

მაგალითი – ხეში ელემენტის ჩასმის ფუნქცია `insert`.

```
insert (A:L) T = cbtree (A:L) ctree ctree when
(empty T)
insert (A:L) T = cbtree (root T) (insert (A:L)
(left T)) (right T) when (A < head (root T))
insert (A:L) T = cbtree (A:(L:tail (root T)))
(left T) (right T) when (A == head (root T))
insert (A:L) T = cbtree (root T) (left T)
(insert (A:L) (right T)) otherwise
```

ეს არის რეალიზაცია აბსტრაქტულ დონეზე.

მაგალითი – `B`-ხეებში ელემენტის ძებნის ფუნქცია `access`.

```
access A Emptic = []
access A ((A1:L) Left Right) = access A Left
when (A < A1)
access A ((A1:L) Left Right) = access A Right
when (A > A1)
access A ((A:L) Left Right) = L
access A (Root Left Right) = access A Right
otherwise
```

ამ მაგალითში მოყვანილია ორი კონსტრუქცია – აბსტრაქტული ელემენტი `Emptic`, რომელიც წარმოადგენს ცარიელ ხეს, ასევე ნიშანი : , რომლის საშუალებით აბსტრაჰირდება დეკარტეს ნამრავლი, რომელიც აქ გამოიყენება სიური წარმოდგენის ნაცვლად. მაგრამ უნდა გვახსოვდეს, რომ ეს მხოლოდ აბსტრაქტული ფუნქციონალური ენაა.

წარმოდგენილ ორივე მაგალითში არსებობს ერთი პრობლემა. დაწერილი ფუნქციების გამოყენებისას ხდება ძალზე დიდი რაოდენობით ზედმეტი კოპირება მეხსიერების ერთი ადგილიდან მეორეში. არსებითად, ეს არის ახალი ხის აგება ახალი ელემენტებით (საუბარი ეხება ფუნქციას `insert`). ეს შეიძლება თავიდან ავიცილოთ დესტრუქციული მინიჭებით.

სავარჯიშო №6

1. მოახდინეთ `insert` ფუნქციის კონსტრუირება, რომელიც ჩასვამს ელემენტს `B-`ხეში, გამოიყენებს რა დესტრუქციულ მინიჭებას.

თავი 2.9. ფუნქციების თვისებების დამტკიცება

ამოცანა: ვთქვათ გვაქვს ფუნქციების ერთობლიობა $f = \langle f_1, \ldots, f_n \rangle$, რომელიც განსაზღვრულია არეებზე $D = \langle D_1, \ldots, D_n \rangle$. საჭიროა დამტკიცდეს, რომ მნიშვნელობების ნებისმიერი d ერთობლიობისთვის ადგილი აქვს შემდეგ თვისებას, ანუ:

$$\forall d \in D : P(F(d)),$$

სადაც P – განხილული თვისებაა. მაგალითად:

1. $\forall d \in D : f(d) \geq 0$
2. $\forall d \in D : f(d) = f(f(d))$
3. $\forall d \in D : f(d) = f_1(d)$

შემოდის პრინციპული შეზღუდვა განსახილველ თვისებებზე – თვისებები არის მხოლოდ ტოტალური, ანუ მართებულია მთელი D არისთვის.

შემდგომ განვიხილავთ D განსაზღვრის არის ზოგიერთ სახეს.

D – წრფივად დალაგებული სიმრავლეა

ვიტყვით, რომ სიმრავლე წრფივად დალაგებულია, თუ სიმრავლის ყოველი ორი ელემენტისთვის შეიძლება ითქვას, რომ ერთი მეტია ან ტოლი მეორეზე ან პირიქით:

$$\forall d_1, d_2 \in D : (d_1 \geq d_2) \vee (d_1 \leq d_2).$$

მაგალითისთვის მოვიყვანოთ მთელი რიცხვების სიმრავლე. წრფივად დალაგებული სიმრავლე ფუნქციონალურ დაპროგრამებაში გვხვდება ძალზე იშვიათად. ავიღოთ თუნდაც უმარტივესი

სტრუქტურა, რომელიც ყველაზე ხშირად მუშავდება ფუნქციონალურ დაპროგრამებაში – სია. სიებისთვის უკვე ძალზე ძნელია განვსაზღვროთ რიგის მიმართება.

ფუნქციის თვისების დასამტკიცებლად წრფივად დალაგებული სიმრავლისთვის საკმარისია განვახორციელოთ ინდუქცია მონაცემებზე, ანუ საკმარისია დავამტკიცოთ ორი პუნქტი:

1. $P(f(\varnothing))$ – ინდუქციის ბაზისი;
2. $\forall d_1, d_2 \in D:,\ d_1 \le d_2 : P(f(d_1)) \rightarrow$ ინდუქციის ბიჯი.

იმის გამო, რომ სტრუქტურები იშვიათად ქმნის წრფივად დალაგებულ სიმრავლეს, უფრო ეფექტური საშუალებაა ინდუქციის მეთოდის გამოყენება D ტიპის აგებისას.

D – განისაზღვრება, როგორც ინდუქციური კლასი

ჩვენთვის უკვე ცნობილია, რომ ინდუქციური კლასი განისაზღვრება ბაზისური კლასის შემოტანით (ეს არის რომელიღაც $d_i = 0,n$ in D კონსტანტების ერთობლიობა ან საწყისი $A_i = 0,n$: d in A_i => d in D ტიპების ერთობლიობა). ინდუქციური კლასი განისაზღვრება ასევე ინდუქციის ბიჯით – მოცემულია კონსტრუქტორები $g_1, \ldots, g_m$, რომლებიც განსაზღვრულია A_i და D-ზე და სამართლიანია, რომ:

$$(a_i \in A_i) \wedge (x_i \in D) \Rightarrow g_k(a_i, x_j) \in D, k = 1..m$$

1. P (f (d)) აუცილებელია დავამტკიცოთ კლასის ბაზისისათვის;
2. ინდუქციის ნაბიჯი: P (f (d)) = P (f (g_i (d))).

მაგალითად, სიებისთვის (ტიპი List (A)) ფუნქციების თვისებების დასამტკიცებლად, საკმარისია დამტკიცდეს თვისება ორი შემდეგი შემთხვევისთვის:

1. P (f ([])).
2. ∀a∈A, ∀L∈List(A):P(f(L)))⇒P(f(a:L))

ფუნქციების თვისებების დამტკიცება S-გამოსახულებებზე (ტიპი S-expr (A)) შეიძლება განვიხილოთ შემდეგი ინდუქციის მაგალითზე:

1. ∀a∈A:P(f(a)).
2. ∀x,y ∈S – exp(A):P(f(x))∧P(f(y)) ⇒P(f(x:y))

მაგალითი: დავამტკიცოთ, რომ ∀L ∈List(A):L*[]=L.

ამ თვისების დასამტკიცებლად შეიძლება განვიხილოთ მხოლოდ List (A) ტიპის განსაზღვრებები და თვითონ ფუნქცია append (ინფიქსურ ჩანაწერში გამოვიყენოთ სიმბოლო *).

1. L = [] : [] * [] = [] = L. ინდუქციის ბაზისი დამტკიცდა.

2. ∀L∈List(A):L*[]=L. ახლა გამოვიყენოთ კონსტრუქტორი: a : L. აუცილებელია დავამტკიცოთ, რომ (a : L) * [] = a : L. ეს ხდება ფუნქცია append-ის განსაზღვრების მეორე კლოზის გამოყენების საშუალებით:

(a:L)*[]=a:(L*[])=a:(L)=a:L.

მაგალითი: დავამტკიცოთ, რომ ფუნქცია append ასოციაციურია

ანუ საჭიროა დავამტკიცოთ, რომ ნებისმიერი სამი სიისათვის L1, L2 და L3 ადგილი აქვს ტოლობას (L1 * L2) * L3 = L1 * (L2 * L3). დამტკიცებისას ინდუქციას გამოვიყენებთ პირველ ოპერანდთან, ანუ სია L1-თან:

1°. L_1 = []:

```
([] * L2) * L3 = (L2) * L3 = L2 * L3.
[] * (L2 * L3) = (L2 * L3) = L2 * L3.
```

2°. ვთქვათ, სიებისთვის L_1, L_2 და L_3 ფუნქცია append ასოციაციურია. აუცილებელია დავამტკიცოთ სიებისთვის (a : L_1), L_2 და L_3:

```
((a : L1) * L2) * L3 = (a : (L1 * L2)) * L3 =
a : ((L1 * L2) * L3).
(a : L1) * (L2 * L3) = a : (L1 * (L2 * L3)).
```

როგორც ჩანს, უკანასკნელი ორი გამოსახულება ტოლია, რაც გულისხმობს, რომ სიებისთვის L1, L2 და L3 ასოციაციურობა დამტკიცებულია.

მაგალითი 25. ფუნქცია reverse-ის ორი განსაზღვრების იგივეობის დამტკიცება.

განმარტება 1:

```
reverse [] = []
reverse (H : T) = (reverse T) * [H]
```

განმარტება 2:

```
reverse' L = rev L []

rev [] L = L
rev (H : T) L = rev T (H : L)
```

ჩანს, რომ სიების შებრუნების ფუნქციის პირველი განმარტება – ეს ჩვეულებრივი რეკურსიული განმარტებაა. მეორე განმარტება იყენებს დამგროვებელ პარამეტრს. საჭიროა დავამტკიცოთ, რომ:

∀L∈List(A):reverse L=reverse`L.

1. ბაზისი – L = []:

```
reverse [] = [].
reverse' [] = rev [] [] = [].
```

2. ბიჯი – ვთქვათ, `L` სიისთვის ფუნქციების `reverse` და `reverse'` იგივეობა დამტკიცებულია. აუცილებელია დამტკიცდეს იგი სიისთვის `(H : L)`:

```
reverse (H : L) = (reverse L) * [H] =
(reverse' L) * [H].
reverse' (H : L) = rev (H : L) [] = rev L (H :
[]) = rev L [H].
```

ახლა აუცილებელია დავამტკიცოთ ბოლო ორი გამოყვანილი გამოსახულების ტოლობა `A` ტიპის ნებისმიერი სიისთვის. ესეც ინდუქციით ხდება:

2.1. ბაზისი – `L = []`:

```
(reverse' []) * [H] = (rev [] []) * [H] = [] *
[H] = [H].
rev [] [H] = [H].
```

2.2. ბიჯი – `L = (A : T)`:

```
(reverse' (A : T)) * [H] = (rev (A : T) []) *
[H] = (rev T (A : [])) * [H] = (rev T [A]) *
[H].
rev (A : T) [H] = rev L (A : H).
```

აქ მოხდა ცუდ უსასრულობაზე გადავარდნა. თუ გავაგრძელებთ დამტკიცებას ინდუქციის გამოყენებით ახალგამოყვანილ გამოსახულებებზე, მაშინ ეს გამოსახულებები სულ უფრო და უფრო გართულდება. მაგრამ ეს არ არის სასოწარკვეთის მიზეზი, რადგან მაინც შეიძლება დავამტკიცოთ. უბრალოდ, საჭიროა, როგორც წინა მაგალითში, მოვიფიქროთ „ინდუქციური ჰიპოთეზა“.

ინდუქციური ჰიპოთეზა: `(reverse' L1) * L2 = rev L1 L2.` ეს ინდუქციური ჰიპოთეზა წარმოადგენს განზოგადებულ გამოსახულებას `(reverse' L) * [H] = rev L [H].`

ინდუქციის ბაზისი ამ ჰიპოთეზისთვის ცხადია. ინდუქციის ბიჯის გამოყენება 2.2 პუნქტში მყოფ გამოსახულებასთან ასე გამოიყურება:

```
(reverse' (A : T)) * L2 = (rev (A : T) []) *
L2 = (rev T [A]) *
            L2 = ((reverse' T) * [A]) * L2 =
          = (reverse' T) * ([A] * L2) =
          (reverse' T) * (A : L2).

rev (A : T) L2 = rev T (A : L2) = (reverse' T)
* (A : L2).
```

რისი დამტკიცებაც მოითხოვებოდა.

დასკვნა: ზოგად შემთხვევებში ფუნქციის თვისებების დასამტკიცებლად ინდუქციის მეთოდით შეიძლება საჭირო გახდეს ზოგიერთი ევრისტიკის გამოყენება, უფრო ზუსტად, ინდუქციური ჰიპოთეზების შემოტანა. ევრისტიკული ბიჯი არის დებულების ფორმულირება, რომელიც არსაიდან არ გამომდინარეობს. ასე რომ, ფუნქციის თვისებების დამტკიცება არის ხელოვნება.

თავი 2.10. ფუნქციონალური დაპროგრამების ფორმალიზაცია λ-აღრიცხვის საფუძველზე

შესწავლის ობიექტია: ფუნქციების განსაზღვრებების სიმრავლე.

დაშვებები: ჩავთვალოთ, რომ ნებისმიერი ფუნქცია განსაზღვრულია რომელიღაც (λ-გამოსახულებით.

კვლევის მიზანი: ორი ფუნქციის - <f_1> და <f_2> აღწერის მიხედვით განვსაზღვროთ მათი იგივეობა განსაზღვრის მთელ არეზე – $\forall x: f_1(x) = f_2(x)$ (აქ გამოყენებულია ასეთი ნოტაცია: f – განსაზღვრული ფუნქცია, ამ ფუნქციის აღწერა (λ-აღრიცხვის ტერმინებში).

პრობლემა ისაა, რომ, ჩვეულებრივ, ფუნქციის აღწერისას მოიცემა ამ ფუნქციის ინტენსიონალი, ხოლო შემდეგ მოითხოვება დამყარდეს ექსტენსიონალური ტოლობა. ექსტენსიონალური ფუნქციის ქვეშ იგულისხმება მისი გრაფიკი (ან ცხრილი <არგუმენტი, მნიშვნელობა> წყვილების სიმრავლის სახით). ფუნქციის ინტენსიონალის ქვეშ იგულისხმება მოცემულ არგუმენტზე ფუნქციის მნიშვნელობის გამოთვლის წესი.

ისმის კითხვა: როგორ გავითვალისწინოთ ჩადგმული ფუნქციების სემანტიკა მათი ექსტენსიონალების შედარებისას (რადგანაც ამ ფუნქციების ცხადი აღწერა არ არის ცნობილი)? პასუხების ვარიანტებია:

1. შეიძლება შევეცადოთ გამოვსახოთ ჩადგმული ფუნქციების სემანტიკა λ-აღრიცხვის მექანიზმის გამოყენებით. ეს პროცესი შეიძლება დავიყვანოთ იმ შემთხვევამდე, როცა ყველა ჩადგმული ფუნქცია არ შეიცავს არაინტერპრეტირებულ ოპერაციებს.

2. ამბობენ, რომ <f1> და <f2> სემანტიკურად ტოლია (ეს ფაქტი აღინიშნება როგორც |= f1 = f2), თუ f1 (x) = f2 (x) არაინტერპრეტირებული იდენტიფიკატორების ნებისმიერი ინტერპრეტაციისას.

ფორმალური სისტემის ცნება

ფორმალური სისტემა წარმოადგენს ოთხეულს:

P = <V, Φ, A, R>, სადაც

V — ანბანი.

Φ — სწორად აგებული ფორმულების სიმრავლე.

A — აქსიომები (ამასთან A in Φ).

R — გამოყვანის წესი.

განხილულ ამოცანაში ფორმულები ასეთია (t1 = t2), სადაც t1 და t2 &mdashl აქვთ λ-გამოსახულების სახე. თუ რომელიმე ფორმულა გამოყვანადია ფორმალურ სისტემაში, მაშინ ეს ფაქტი ასე ჩაიწერება: (|- t1 = t2).

ამბობენ, რომ ფორმალური სისტემა კორექტულია, თუ (|- t1 = t2) => (|= t1 = t2).

ამბობენ, რომ ფორმალური სისტემა სრულია, თუ (|= t1 = t2) => (|- t1 = t2).

ცნება „კონსტრუქციის“ სემანტიკური განმარტება (აღნიშვნა – Exp):

1. v∈Ld ⇒ v∈Exp
2. v∈Ld, E∈Exp⇒λv.E∈Exp
3. E,E`∈Exp⇒(E E`)∈Exp
4. E∈Exp⇒(E)∈Exp
5. სხვა არანაირი Exp არ არის.

შენიშვნა: Id — იდენტიფიკატორების სიმრავლე.

ამბობენ, რომ v თავისუფალია M Exp-ში, თუ:

1. M = v.

2. M = (M_1M_2), და v თავისუფალია M_1-ში ან M_2-ში.

3. M = λv'.M', და и v != v', და v თავისუფალია M'-ში.

4. M = (M'), და v თავისუფალია в M'.

იდენტიფიკატორების სიმრავლე v-ს, რომელიც თავისუფალია M-ში, აღნიშნავენ როგორც FV (M).

ამბობენ, რომ v დაკავშირებულია M in Exp-ში, თუ:

1. M = λv'.M', და v = v'.

2. M = (M_1M_2), და v დაკავშირებულია M_1-ში ან M_2-ში (ანუ ერთი და იგივე იდენტიფიკატორი შეიძლება იყოს თავისუფალი და დაკავშირებული Exp-ში).

3. M = (M'), და v დაკავშირებულია M'-ში.

თავისუფალი და დაკავშირებული იდენტიფიკატორები

M = v. v — თავისუფალია. M = λx.xy. x — დაკავშირებულია, y — თავისუფალია. M = (λv.v)v. v შედის ამ გამოსახულებაში როგორც თავისუფალი, ისევე დაკავშირებული. M = VW. V და W — თავისუფალია.

ჩასმის წესები: E გამოსახულებაში E'გამოსახულების ჩასმა ცვლადი x-ის ყველა თავისუფალ შესვლასთან ერთად აღინიშნება ასე E[x <- E']. ჩასმის დროს ზოგჯერ ხდება სახელების კონფლიქტი, ანუ ცვლადების კოლიზია. კოლიზიების მაგალითებია:

(λx.yx)[y <- λz.z] = λx.(λz.z)x = λx.x — კორექტული ჩასმა.

`(λx.yx)[y <- xx] = λx.(xx)x` — ცვლადების სახელების კოლიზია.

`(λz.yz)[y <- xx] = λz.(xx)z` — კორექტული ჩასმა.

ბაზისური ჩასმების ზუსტი განსაზღვრებები: `x[x <- E'] = E'`

1. `y[x <- E'] = y`
2. `(λx.E)[x <- E'] = λx.E`
3. `(λy.E)[x <- E'] = λy.E[x <- E']`, იმ პირობით, რომ `y in FV (E')`
4. `(λy.E)[x <- E'] = (λz.E[y <- z])[x <- E']`, იმ პირობით, რომ `y !in FV (E')`
5. $(E_1E_2)[x <- E'] = (E_1[x <- E']E_2[x <- E'])$

ფორმალური სისტემის აგება

ამრიგად, მზად ვართ, გადავიდეთ ფორმალური სისტემის აგებამდე, რომელიც აღწერს ფუნქციონალურ დაპროგრამებას λ-აღრიცხვის ტერმინებში.

ფორმულების აგების წესები ასე გამოიყურება: `Exp = Exp.`

აქსიომები:

`(α): |- λx.E = λy.E[x <- y].`

`(β): |- (λx.E)E' = E[x <- E'].`

`(ρ): |- t = t`, იმ შემთხვევაში,თუ `t` – იდენტიფიკატორებია.

გამოყვანის წესები:

(μ): $t_1 = t_2 \Rightarrow t_1t_3 = t_2t_3$

(ν): $t_1 = t_2 \Rightarrow t_3t_1 = t_3t_2$

(σ): $t_1 = t_2 \Rightarrow t_2 = t_1$

(τ): $t_1 = t_2,\ t_2 = t_3 \Rightarrow t_1 = t_3$

(ξ): $t_1 = t_2 \Rightarrow \lambda x.t_1 = \lambda x.t_2$

მაგალითი – გამოვიყვანოთ შემდეგი ფორმულა:

$(\lambda x.xy)(\lambda z.(\lambda u.zu))v = (\lambda v.yv)v$

$(\lambda x.xy)(\lambda z.(\lambda u.zu))v = (\lambda v.yv)v$

(μ): $(\lambda x.xy)(\lambda z.(\lambda u.zu)) = (\lambda v.yv)$

(β): $z.(\lambda u.zu))y = (\lambda v.yv)$

(α): $u.yu = \lambda v.yv$

ეს მიგვითითებს, რომ ფორმულა სწორია.

ფუნქციონალური დაპროგრამების ფორმალიზაციის მეორე ვარიანტში შეიძლება ვისარგებლოთ არა სიმეტრიული თვისების დამოკიდებულებით „=“, არამედ არასიმეტრიული დამოკიდებულებით „->“.

მეორე ფორმალურ სისტემაში ფორმულის გამოყვანის წესი აბსოლუტურად იგივეა, რაც პირველ ვარიანტში. თუმცა აქსიომები ღებულობენ სხვა სახეს:

(α'): $\vdash \lambda x.M \to \lambda y.M[x \leftarrow y]$

(β'): $\vdash (\lambda x.M)N \to M[x \leftarrow N]$

(ρ'): $\vdash M \to M$

გამოყვანის წესი ფორმალური სისტემის მეორე ვარიანტში ერთია:

(π): $t_1 \to t_1',\ t_2 \to t_2' \Rightarrow t_1t_2 \to t_1't_2'$

არსებითად, გამოყვანის ამ წესით ვგებულობთ, რომ ნებისმიერ გამოსახულებაში შეიძლება გამოიყოს შემავალი ქვეგამოსახულება და შეიცვალოს იგი.

განსაზღვრება:

- ✓ გამოსახულებას λx.M ტიპისას ეწოდება α-რედექსი.
- ✓ გამოსახულებას (λx.M)N ტიპისას ეწოდება β-რედექსი.
- ✓ გამოსახულებას, რომელიც არ შეიცავს β-რედექსებს, ეწოდება გამოსახულება ნორმალური ფორმით.

რამდენიმე თეორემა (დამტკიცების გარეშე):

- ✓ |- E1 = E2 => E1 -> E2 | E2 -> E1.
- ✓ E -> E1 & E -> E2 => არსებობს F : E1 -> F & E2 > F (ჩერჩ-როსელის თეორემა).
- ✓ თუ E აქვს ნორმალური ფორმები E1 და E2, მაშინ ისინი განსხვავდებიან მხოლოდ დაკავშირებული ცვლადების აღნიშვნებით.

რედუქციის სტრატეგია

1. ნორმალური რედუქციური სტრატეგია. რედუქციის თითოეულ ბიჯზე ირჩევა ტექსტურად ყველაზე მარცხენა β-რედექსი. დამტკიცებულია, რომ ნორმალური რედუქციული სტრატეგიის გამოყენებით მიიღება გამოსახულების ნორმალური ფორმა, თუ ის არსებობს.

2. აპლიკაციური რედუქციული სტრატეგია. რედუქციის თითოეულ ბიჯზე ამოირჩევა β-რედექსი, რომელიც არ შეიცავს თავის შიგნით სხვა β-რედექსებს. შემდეგ ვაჩვენებთ, რომ აპლიკაციური რედუქციული სტრატეგია ყოველთვის არ იძლევა გამოსახულების ნორმალური ფორმის მიღების საშუალებას.

მაგალითი – გამოსახულების M = (λy.x)(EE) რედუქცია, სადაც E = λ x.xx

1. НРС: (λy.x)(EE) = (λy.x)[y <- EE] = x.

2. APC: (λy.x)(EE) = (λy.x)((λx.xx)(λx.xx)) = (λy.x)((λx.xx)(λx.xx)) = ...

ამ მაგალითში ჩანს, როგორ შეიძლება აპლიკაციურმა რედუქციულმა სტრატეგიამ მიგვიყვანოს ცუდ უსასრულობამდე. M გამოსახულების ნორმალური ფორმის მიღება აპლიკაციური რედუქციის სტრატეგიის გამოყენების დროს შეუძლებელია.

შენიშვნა: წითელი ფერით გამოყოფილია β რედექსი, რომელიც შემდეგი ბიჯით რედუცირდება.

მაგალითი – გამოსახულების M = (λx.xyxx)((λz.z)w) რედუქცია

1. HPC: (λx.xyxx)((λz.z)w) = ((λz.z)w)y ((λz.z)w)((λz.z)w) = wy((λz.z)w)((λz.z)w) = wyw((λz.z)w) = wyww.

2. APC: (λx.xyxx)((λz.z)w) = (λx.xyxx)w = wyww.

დაპროგრამებაში ნორმალური რედუქციული სტრატეგია შეესაბამება გამოძახებას სახელით, ანუ გამოსახულების არგუმენტი არ გამოითვლება მანამ, სანამ გამოსახულების ტანში არ გაჩნდება მასზე მიმართვა. აპლიკაციურ რედუქციულ სტრატეგიას შეესაბამება მნიშვნელობით გამოძახება.

შესაბამისობა ფუნქციონალური პროგრამის გამოთვლებსა და რედუქციას შორის

ინტერპრეტატორის მუშაობა აღიწერება რამდემინე ნაბიჯით:

1. გამოსახულებაში აუცილებელია გამოიყოს რაღაც მიმართვა რეკურსიულ ან ჩადგმულ ფუნქციებს შორის სრულად განსაზღვრული არგუმენტებით. თუ ჩადგმულ ფუნქციაზე გამოყოფილი მიმართვა არსებობს, მაშინ ხდება მისი შესრულება და დაბრუნება პირველ ნაბიჯზე.

2. თუ პირველ ნაბიჯზე გამოყოფილია რეკურსიულ ფუნქციაზე მიმართვა, მაშინ მის ნაცვლად ჩაისმება ფუნქციის ტანი ფაქტობრივი პარამეტრებით (რადგანაც ისინი უკვე აღნიშნულია). შემდეგ ხდება გადასვლა პირველი ბიჯის დასაწყისზე.

3. თუ მეტი მიმართვა არ არის, მოხდება გაჩერება.

ფუნქციონალურ დაპროგრამებაში გამოთვლები, პრინციპში, იმეორებს რედუქციის ნაბიჯებს, მაგრამ დამატებით შეიცავს ჩადგმული ფუნქციების გამოთვლას.

განსაზღვრული ფუნქციის წარმოდგენა λ-გამოსახულების სახით

ნაჩვენებია, რომ ფუნქციის ნებისმიერი განსაზღვრება შეიძლება წარმოდგეს λ-გამოსახულების სახით, რომელიც არ შეიცავს რეკურსიას. მაგალითად:

```
fact = λn.if n == 0 then 1 else n * fact (n - 1)
```

იგივე განსაზღვრება შეიძლება აღვწეროთ რომელიღაც ფუნქციონალის გამოყენებით:

```
fact = (λf.λn.if n == 0 then 1 else n * f (n -
1)) fact
```

წარმოდგენილ გამოსახულებაში მსხვილი შრიფტით გამოყოფილია ფუნქციონალი `F`. ამრიგად, ფაქტორიალის გამოთვლის ფუნქცია შეიძლება ჩავწეროთ ასე: `fact = F fact`. ნებისმიერი რეკურსიული განსაზღვრება `f` ფუნქციისა შეიძლება წარმოდგეს ასეთი სახით:

```
f = F f
```

ეს გამოსახულება შეიძლება განვიხილოთ როგორც განტოლება, რომელშიც რეკურსიული ფუნქცია `f` წარმოადგენს `F` ფუნქციონალის უძრავ წერტილს. შესაბამისად, ფუნქციონალური ენის ინტერპრეტატორი შეიძლება განვიხილოთ როგორც რიცხვითი მეთოდი ამ განტოლების ამოსახსნელად.

შეიძლება გამოვთქვათ ვარაუდი, რომ ეს რიცხვითი მეთოდი (ანუ ინტერპრეტატორი), თავის მხრივ, შეიძლება რეალიზებულ იქნეს რომელიღაც `Y` ფუნქციის საშუალებით, რომელიც `F` ფუნქციონალისთვის პოულობს მის უძრავ წერტილს. შესაბამისად, განსაზღვრავს საძებნ ფუნქციას – `f = Y F`.

`Y` ფუნქციის თვისებები განისაზღვრება ტოლობით:

```
Y F = F (Y F)
```

თეორემა (დამტკიცების გარეშე):

ნებისმიერ λ-გამოსახულებას აქვს უძრავი წერტილი.

λ-აღრიცხვა საშუალებას იძლევა ნებისმიერი ფუნქცია გამოხატულ იქნეს წმინდა λ-გამოსახულებით ჩადგმული ფუნქციების გამოყენების გარეშე. მაგალითად:

1.

```
prefix = λxyz.zxy
head = λp.p(λxy.x)
tail = λp.p(λxy.y)
```

2. პირობითი გამოსახულების მოდელირება:

```
True = λxy.x
False = λxy.y
if B then M else N = BNM, სადაც B in {True, False}.
```

თავი 2.11. პროგრამების ტრანსფორმაცია

P პროგრამის ქვეშ რომელიღაც L ენაზე იგულისხმება რაღაც ტექსტი L-ზე. ფუნქციონალური პროგრამის ქვეშ კი – კლოზების ნაკრები. L ენის სემანტიკა განისაზღვრება, თუ მოცემულია ამ ენის ინტერპრეტატორი. ინტერპრეტატორი განისაზღვრება ფორმულით:

```
Int (P, d) = d'
```

სადაც:

P – პროგრამაა;
d – საწყისი მონაცემები;
d' – გამოსასვლელი მონაცემები.

თუ ინტერპრეტატორი Int წარმოდგენილია კარირებული f ფუნქციის სახით ისე, რომ f P d = d', მაშინ განსაზღვრება f = M f, უფრო ზუსტად M ფუნქციონალს უწოდებენ L ენის დენოტაციურ სემანტიკას. ამ შემთხვევაში აზრი აქვს λ-გამოსახულებას: f P = λd.M' : D -> D'. ამასთან, ნაწილობრივი ფუნქცია f P შეიძლება განვიხილოთ, როგორც ერთარგუმენტიანი ფუნქცია f_P-ეს არის ფუნქცია, რომელიც ახდენს პროგრამა P-ს რეალიზებას. როგორც უკვე ვაჩვენეთ, შეიძლება აიგოს რეკურსიული განმარტება შემდეგი სახის: $f_P = M_P\ f_P$. ასეთი სახე აქვს თავდაპირველად ყველა ფუნქციას ფუნქციონალურ ენაზე, მაგრამ ეს ჩანაწერი შეიძლება გავიგოთ ორნაირად:

- ✓ ეს განმარტება შეიძლება გავიგოთ როგორც სიმბოლოების სტრიქონი, რომელიც ეძლევა შესასვლელზე ინტერპრეტა-

ტორს. ფუნქცია, რომელსაც ინტერპრეტატორი ითვლის ტექსტის მიხედვით $f = M f$, აღინიშნება როგორც f_{int}.

- ✓ $f = M f$ – არის f ფუნქციის წმინდა მათემატიკური განსაზღვრება. ამ განტოლების ამოხსნა აღინიშნება როგორც f_{mat}.

ისმის კითხვა: რა კავშირშია ეს ორი ფუნქცია – f_{int} და f_{mat}? უნდა ვიმედოვნოთ, რომ კორექტული ინტერპრეტატორი სწორად ითვლის f_{mat}-ს.

განმარტება:

ამბობენ, რომ ფუნქცია f_1 ნაკლებად განსაზღვრულია, ვიდრე ფუნქცია f_2 (აღინიშნება როგორც $f_1 \subseteq f_2$), თუ $\forall x: f_1\ x{=}y \Rightarrow f_2\ x{=}y$. იმ შემთხვევაში, როცა ორი ფუნქციისთვის ერთდროულად სრულდება $f_1 \subseteq f_2$და $f_2 \subseteq f_1$, მაშინ ადგილი აქვს ფუნქციების იგივეობას.

როგორც წესი, $f_{int} \subseteq f_{mat}$ – ეს ხდება იმიტომ, რომ ჩვეულებრივი ინტერპრეტატორი ახორციელებს რედუქციის აპლიკაციურ სტრატეგიას. თუმცა შემდგომ ჩვენ ვიგულისხმებთ ფუნქციების f_{int} და f_{mat} იგივეობას, ამიტომ პროგრამების ტექსტს განვიხილავთ როგორც ფუნქციის მათემატიკურ განმარტებას. მაშინ ფუნქციონალური პროგრამების ეკვივალენტური გარდაქმნა არის მათემატიკურად განსაზღვრული ფუნქციის, უბრალოდ, სპეციალური სახის ეკვივალენტური გარდაქმნა.

პროგრამების ტრანსფორმაცია არის სინტაქსური გარდაქმნა, რომლის დროსაც საერთოდ არ ხდება პროგრამის სემანტიკასთან შეხება, ვინაიდან პროგრამა გაიგება როგორც სიმბოლოების ნაკრები. ფაქტი, რომ ერთი ტექსტი f_1 მიიღება სინტაქსური ტრანსფორმაციის შედეგად მეორე f_2 ტექსტიდან, ჩაიწერება ასე: $f_1 \mid\!- f_2$.

ამბობენ, რომ გარდაქმნა კორექტულია, თუ $f_1 \subseteq f_2$.

ამბობენ, რომ გარდაქმნა ეკვივალენტურია, თუ $f_1 \equiv f_2$.

შემოდის კიდევ რამდენიმე სპეციალური აღნიშვნა:

გვაქვს კლოზების საწყისი ნაკრები (ანუ გარდაქმნის ობიექტების) და ეს ნაკრები აღნიშნავს ან DEF, ან SPEC.

1. კლოზები, რომლებიც აღწერენ საწყისი კლოზებიდან ასახვას, აღინიშნება როგორც INV.

2. ზოგიერთი ტოლობა, რომელიც გამოხატავს ფუნქციის შიდა (დარეზერვირებულ) თვისებას, აღინიშნება როგორც LOW.

განმარტება:

ვთქვათ, F (X) არის გამოსახულება (ტოლობა), რომელიც შეიცავს X ტერმს, მაშინ F [X <- M]-ს უწოდებენ F-ის მაგალითს (M – გამოსახულებაა).

გარდაქმნის სახეები

1°. კონკრეტიზაცია (instantiation) — INS.

$$\frac{E_1(X) = E_2(X)}{E_1[X \leftarrow N] = E_2[X \leftarrow N]}$$

2°. გარდაქმნა სახელის გარეშე

$$\frac{M(Y) = N(Y),\ E_1 = E_2[X \leftarrow M']M' = M[Y \leftarrow G]}{E_1 = E_2[X \leftarrow N'],\ N' = N[Y \leftarrow G]}$$

3°. გაშლა (unfolding) — UNF.

$$\frac{M(Y) = N(Y),\ E_1 = E_2(M'),\ M' = M[Y \leftarrow G]}{E_1 = E_2(N'),\ N' = N[Y \leftarrow G]}$$

4°. შეფუთვა (folding) - FLD.

$$\frac{M(Y) = N(Y),\ E_1 = E_2(N'),\ N' = N[Y \leftarrow G]}{E_1 = E_2(M'),\ M' = M[Y \leftarrow G]}$$

5°. კანონი (law) - LAW.

$$\frac{M(Y) = N(Y), E_1 = E_2(M'), M' = M[Y \leftarrow G]}{E_1 = E_2(N'), N' = N[Y \leftarrow G]}$$

6°. აბსტრაქცია და აპლიკაცია (abstraction & application) - ABS.

$$\frac{M[X \leftarrow G] = (\lambda XM)G, E_1 = E_2(M[X \leftarrow G]}{E_1 = E_2((\lambda XM)G)}$$

მაგალითი 30. ფუნქცია length-ის გარდაქმნა.

```
length [] = 0                                  1 (DEF)
length (H:T) = 1 + length T                      2 (DEF)
length_2 L1 L2 = length L1 + length L2 3 (SPEC)

length_2 [] L    = length [] + length L        4 (INS 3)
        = 0 + length L                        5 (UNF 1)
        = length L                            6 (LAW +) (*)

length_2 (H:T) L= length (H:T) + length L     7 (INS 3)
        = (1 + length T) + length L             8 (UNF 2)
        = 1 + (length T + length L)             9 (LAW +)
        = 1 + length_2 T L                    10 (FLD 3) (**)
```

ახლა ავიღოთ ორი მიღებული კლოზი (*) და (**) შევადგინოთ მათგან ახალი რეკურსიული განმარტება ახალი ფუნქციისა, რომელიც არ იყენებს ძველი ფუნქციის გამოძახებას:

```
length_2 [] L = length L
length_2 (H:T) L = 1 + length_2 T L
```

უნდა აღვნიშნოთ, რომ ახალი კლოზების არჩევა ახალი განმარტებისთვის ითხოვს დამატებით კვლევას და არ სრულდება ავტომატურად.

ფუნქციის განსაზღვრის ასეთი ტრანსფორმაცია ხშირად მიგვიყვანს ფუნქციის სირთულის შემცირებამდე. მაგალითად, ფუნქციისთვის, რომელიც ითვლის ფიბონაჩის N-ურ რიცხვს, შეიძლება ავაგოთ განსაზღვრება, რომლის გამოთვლის სიზუსტე წრფივად არის N-ზე დამოკიდებული, და არა ფიბონაჩის წესების მიხედვით, როგორც ეს ჩვეულებრივ განსაზღვრებაშია.

მაგრამ ტრანსფორმაცია უნდა შესრულდეს მოფიქრებულად, რადგანაც შეიძლება მივიდეთ FLD და UNF ბიჯების უსასრულო ციკლამდე. რათა ტრანსფორმაციისას არ მივიდეთ აბსურდამდე, ყურადღება უნდა მივაქციოთ, რომ გარდაქმნის პროცესში მიღებული გამოსახულებების ზოგადობა არ გაიზარდოს, ანუ ტრანსფორმაცია უნდა განხორციელდეს ზოგადიდან კერძოსკენ.

ინფორმატიკის მეორე კანონი

- ✓ არსებობს ამოუხსნადი ამოცანები.
- ✓ არ არსებობს ეფექტური რეალიზაცია დეკლარაციული ტიპის ენებისთვის, თუ ისინი უნივერსალურია.

ტრანსფორმაციული სინთეზის კონცეფცია: საშუალებას აძლევს პროგრამისტს დაწეროს ფუნქციის განსაზღვრება ისე, რომ არ იზრუნოს მის ეფექტურობაზე.

თუმცა დამტკიცდა, რომ სპეციფიკაციების ენის მიხედვით არ შეიძლება გამომუშავდეს (ანუ ტრანსფორმირდეს საწყისი ტექსტი) ფუნქციის ვარიანტი, რომელიც მუშაობს ეფექტურად. თუ სპეციფიკაციის ენად განვიხილავთ ფუნქციონალურ ენას, მაშინ ცხადი ხდება, რომ პროგრამისტმა თავად უნდა იზრუნოს თავისი პროგ-

რამის ეფექტურობაზე – ტრანსფორმაციული სინთეზის კონცეფცია აქ არ გამოდგება.

ნაწილობრივი გამოთვლები

ვთქვათ, P და S — ორი ენაა, რომელიც მუშაობს სიმბოლოების სტრიქონებთან (ეს არ არღვევს განხილვის ზოგადობას), ხოლო P და S — სინტაქსურად სწორი პროგრამების ერთობლიობაა. D — ყველა შესაძლო სიმბოლოების თანმიმდევრობის დომენია.

```
P :: D -> (D* -> D)
```

თუ p — პროგრამაა P-დან, მაშინ:

```
P (p) :: D* -> D
P (p) (<d1, ..., dn>) = d, და d in D
P (r) (<y1, ..., yn>) = P (p) (<d1, ..., dm, y1, ..., yn>)
```

ბოლო ტოლობაში ცვლადებით yi აღნიშნულია უცნობი მონაცემები. პროგრამა p-თვის ეს n ცვლადი წარმოადგებს დარჩენილ კოდს.

ნაწილობრივი გამომთვლელი MIX ეწოდება პროგრამას P-დან (თუმცა ნაწილობრივი გამომთვლელი შეიძლება რეალიზებულ იქნეს ნებისმიერ ენაზე), ისეთს, რომ:

$$\forall p \in P, p(x_1, \ldots, x_m, x_{m+1}, \ldots, x_n) : p(MIX)(< p, d_1, \ldots, d_m >) = r..$$

ანუ MIX – არის პროგრამა, რომელიც ღებულობს რა საწყის პროგრამას და მონაცემებს მისი ცნობილი პარამეტრებისთვის, გვაძლევს საწყისისთვის დარჩენილ პროგრამას.

ენა S-ის ინტერპრეტატორი ეწოდება პროგრამას INT in P, ისეთს, რომ:

$$\forall s \in S, < d_1, \ldots, d_n > \in D^* : P(INT)\ (< s, d_1, \ldots, d_n >) = S(s)\ (< d_1, \ldots, d_n >) .$$

S ენის კომპილერი ეწოდება პროგრამას COMP in P, ისეთს, რომ:

$$P(COMP)\ (< s >) = TARGET$$
$$P(TARGET)\ (< d_1, \ldots, d_n >) = S(s)\ (< d_1, \ldots, d_n >) .$$

ანუ:

$$P(P(COMP)\ (< s >)\ (< d_1, \ldots, d_n >) = S(s)\ (< d_1, \ldots, d_n >) .$$

S ენის კომპილატორების კომპილატორი ეწოდება პროგრამას COCOM in P, ისეთს, რომ:

$$P(COCOM)\ (< INT >) = COMP$$
$$P(P(P(COCOM)\ (< INT >))\ (< s >))\ (< d_1, \ldots, d_n >) = S(s)\ (< d_1, \ldots, d_n >) .$$

ფუტამორების პროექციები:

- TARGET = P (MIX) (<INT, s>)
- COMP = P (MIX) (<MIX, INT>)
- COCOM = P (MIX) (<MIX, MIX>)

ეს სამი დებულება თეორემებია, რომლებიც საკმაოდ ადვილად შეიძლება დამტკიცდეს ზემოთ მოყვანილი განმარტებებით.

საცნობარო მასალა

ფუნქციონალური დაპროგრამების ენები

ზოგიერთი ფუნქციონალური ენა აღვწეროთ მოკლედ. დამატებითი ინფორმაცია იხილეთ ქვემოთ ჩამოთვლილ რესურსებში.

Lisp (List processor). ითვლება, რომ არის პირველი ფუნქციონალური ენა. არატიპიზებულია; შეიცავს იმპერატიულ თვისებებსაც, თუმცა, საზოგადოდ, წაახალისებს ფუნქციონალური დაპროგრამების სტილს. გამოთვლებისას გამოიყენება გამოძახება მნიშვნელობით. არსებობს ენის ობიექტორიენტირებული დიალექტი CLOS.

ISWIM (If you See What I Mean). ფუნქციონალური ენა-პროტოტიპი. დამუშავებულია პიტერ ლენდიმის (Peter J. Landin) მიერ XX საუკუნის 60-იან წლებში იმის სადემონსტრაციოდ, თუ როგორი შეიძლება იყოს ფუნქციონალური ენა. ენასთან ერთად ლენდიმმა შექმნა სპეციალური ვირტუალური მანქანა, რომელიც ასრულებდა პროგრამებს ISWIM-ზე და მიიღო სახელწოდება SECD-მანქანა. ISWIM ენის სინტაქსს იყენებს მრავალი ფუნქციონალური ენა. ISWIM სინტაქსს ჰგავს ML-ის სინტაქსი, განსაკუთრებით Caml-ის სინტაქსი.

Scheme. Lisp-ის დიალექტი, რომელიც დანიშნულია სამეცნიერო კვლევებისთვის computer science დარგში. Scheme-ის შექმნისას ყურადღება გამახვილდა ელეგენტურობასა და სიმარტივეზე. ამის გამო ენა უფრო პატარა გამოვიდა, ვიდრე Common Lisp.

ML (Meta Language). მკაცრი ენების ოჯახი განვითარებული ტიპების პოლიმორფული სისტემით და პარამეტრიზებული მოდულებით. ML ისწავლება დასავლეთის მრავალ უნივერსიტეტში (ზოგან დაპროგრამების პირველ ენადაც კი).

Standard ML. ერთ-ერთი პირველი ტიპიზებული ფუნქციონალური დაპროგრამების ენაა; შეიცავს ზოგიერთ იმპერატიულ თვისებას, ისეთს, როგორიცაა მიმთითებლები (გამოიყენება მნიშვნელობების შესაცვლელად) და ამიტომაც არ არის სუფთა ენა. ძალზე საინტერესოდ არის რეალიზებული მოდულურობა. გამოთვლებისას იყენებს გამოძახებას მნიშვნელობით; აქვს ტიპების ძლიერი პოლიმორფული სისტემა. ენის ბოლო სტანდარტია Standard ML–97, რომლისთვისაც არსებობს სინტაქსის, ასევე ენის სტატიკური და დიმანიკური სემანტიკის ფორმალური მათემატიკური აღწერები.

Caml Light და Objective Caml. როგორც Standard ML, მიეკუთვნება ML ოჯახს. Objective Caml განსხვავდება Caml Light-გან იმით, რომ მხარს უჭერს კლასიკურ ობიექტორიენტირებულ დაპროგრამებას. როგორც Standard ML, ისიც მკაცრია, მაგრამ აქვს ჩადგმული მექანიზმი გადატანილი გამოთვლებისთვის.

Miranda. შემუშავებულია დევიდ ტერნეტის მიერ, როგორც სტანდარტული ფუნქციონალური ენა, რომელიც იყენებს გადატანილ გამოთვლებს; აქვს მკაცრი ტიპების პოლიმორფული სისტემა. მსგავსად ML-ისა, ისწავლება მრავალ უნივერსიტეტში. დიდი ზეგავლენა იქონია Haskell ენის მკვლევრებზე.

Haskell. ერთ-ერთი ყველაზე გავრცელებული არამკაცრი ენაა; აქვს ტიპიზაციის ძალზე განვითარებული სისტემა; უფრო ცუდად მუშაობს მოდულებთან. ენის ბოლო სტანდარტია — Haskell 98.

Gofer (GOod For Equational Reasoning). Haskell-ის გამარტივებული დიალექტი; გამიზნულია ფუნქციონალური დაპროგრამების შესასწავლად.

Clean. სპეციალურად დანიშნულია პარალელური და დანაწილებული დაპროგრამებისთვის; სინტაქსით მიაგავს Haskell-ს. სუფთაა; იყენებს გადატანილ გამოთვლებს; კომპილატორთან ერთად მოყვება ბიბლიოთეკები (I/O libraries), რაც საშუალებას იძლევა დაპროგრამდეს გრაფიკული ინტერფეისის Win32-თვის ან MacOS-თვის.

ინტერნეტრესურსები ფუნქციონალურ დაპროგრამებაში

www.haskell.org – საიტი, რომელიც ეხება ფუნქციონალურ დაპროგრამებას ზოგადად და ენა Haskell – კონკრეტულად; შეიცავს სხვადასხვა საცნობარო ინფორმაციას, ინტერპრეტატორების სიას და Haskell-ის კომპილერს (ამჟამად ყველა ინტერპრეტატორი და კომპილერი არის უფასო). ამის გარდა, საიტზე ნახავთ უამრავი რესურსის მისამართს ფუნქციონალური დაპროგრამების თეორიისა და სხვა ენების შესახებ(Standard ML, Clean).

cm.bell-labs.com/cm/cs/what/smlnj – Standard ML of New Jersey. ძალზე კარგი კომპილერია. უფასო დისტრიბუციაში კომპილერის გარდა ასევე შედის MLYacc და MLLex უტილიტები და ბიბლიოთეკა Standard ML Basis Library. შეიძლება გაეცნოთ კომპილერისა და ბიბლიოთეკის დოკუმენტაციასაც.

www.harlequin.com/products/ads/ml/–Harlequin MLWorks, კომერციული კომპილერი Standard ML-ის. თუმცა, არაკომერციული მიზნით შესაძლოა ისარგებლოთ უფასო, ცოტათი შეზღუდული ვერსიით.

caml.inria.fr — ინსტიტუტი INRIA. Caml Light და Objective Caml ენების მკვლევართა ჯგუფის საშინაო საიტი. შესაძლოა უფასოდ გადმოქაჩოთ Objective Caml, რომელიც შეიცავს ინტერპრეტატორს, კომპილერს ბაიტკოდში და მანქანურ კოდში, Yacc და Lex Caml-ისთვის, გამმართველს და პროფაილერს, დოკუმენტაციას, მაგალითებს. კომპილირებული კოდის ხარისხი ამ კომპილერისა იმდენად კარგია, რომ სიჩქარით Standard ML-აც კი სჯობნის New Jersey-დან.

www.cs.kun.nl/~clean/ — შეიცავს ენა Clean-ის კომპილატორის დისტრიბუციას. ეს კომპილერი კომერციულია, მაგრამ შესაძლოა მისი უფასო გამოყენება არაკომერციული მიზნებისთვის. იქიდან გამომდინარე, რომ კომპილერი ფასიანია, მისი ხარისხიც მისაღებია (ძალზე სწრაფია), აქვს დამუშავების გარემო, მოჰყვება კარგი დოკუმენტაცია და სტანდარტული ბიბლიოთეკები.

სარჩევზე დაბრუნება

გამოყენებული ლიტერატურა

1. Graham Hutton. Programming in Haskell. 2007. ISBN:9780521871723 http://ebooks.cambridge.org/ebook.jsf?bid=CBO9780511813672
2. Душкин Р. В. Функциональное программирование на языке Haskell. — М.: ДМК-Пресс, 2007. — 608 стр. ISBN 5-94074-335-8
3. Хювёнен Э., Сеппенен И. Мир Lisp'а. В 2-х томах. М.: Мир, 1990.
4. Бердж В. Методы рекурсивного программирования. М.: Машиностроение, 1983.
5. Филд А., Харрисон П. Функциональное программирование. М.: Мир, 1993.
6. Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.
7. Джонс С., Лестер Д. Реализация функциональных языков. М.: Мир, 1991.
8. Henson M. Elements of functional languages. Dept. of CS. University of Sassex, 1990.
9. Fokker J. Functional programming. Dept. of CS. Utrecht University, 1995.
10. Thompson S. Haskell: The Craft of Functional Programming. 2-nd edition, Addison-Wesley, 1999.
11. Bird R. Introduction to Functional Programming using Haskell. 2-nd edition, Prentice Hall Press, 1998.

სარჩევზე დაბრუნება

პასუხები თვითშემოწმებისთვის

სავარჯიშო №1

პასუხებში ხშირად წარმოდგენილი გვაქვს რამდენიმე შესაძლო ვარიანტიდან ერთ-ერთი.

1. ფუნქციები, რომლებიც ითვლის მიმდევრობის N-ურ წევრს:

a. Power:

```
Power (X, 0) = 1
Power (X, N) = X * Power (X, N - 1)
```

b. Summ_T:

```
Summ_T (1) = 1
Summ_T (N) = N + Summ_T (N - 1)
```

c. Summ_P:

```
Summ_P (1) = 1
Summ_P (N) = Summ_T (N) + Summ_P (N - 1)
```

d. Summ_Power:

```
Summ_Power (N, 0) = 1
Summ_Power (N, P) = (1 / Power (N, P)) +
Summ_Power (N, P - 1)
```

e. Exponent:

```
Exponent (N, 0) = 1
Exponent (N, P) = (Power (N, P) / Factorial
(P)) + Exponent (N, P - 1)

Factorial (0) = 1
Factorial (N) = N * Factorial (N - 1)
```

2. ოპერაცია prefix-ის მუშაობის მაგალითი შეიძლება წარმოვადგინოთ სამი მიდგომით (ისევე, როგორც ეს განხილულია მაგალითში). შესაძლებელია ოპერაცია prefix-ის წარმოდგენა ინფიქსური ჩანაწერის, სიმბოლო :-ის გამოყენებით.

a. ოპერაციის მუშაობის პირველი მაგალითია თვით ოპერაციის განმარტება. მის განხილვას არ აქვს აზრი, რადგან prefix სწორედ ასე განიმარტება. b. prefix (a1, [b1, b2]) = prefix (a1, b1 : (b2 : [])) = a1 : (b1 : (b2 : [])) = [a1, b1, b2]

(ეს გარდაქმნები მოყვანილია სიის განმარტების მიხედვით).

c. prefix ([a1, a2], [b1, b2]) = prefix ([a1, a2], b1 : (b2 : [])) = ([a1, a2]) : (b1 : (b2 : [])) = [[a1, a2], b1, b2].

3. Append ფუნქციის მუშაობის მაგალითად განვიხილოთ ორი სიის შერწყმა, რომლიდანაც თითოეული შედგება ორი ელემენტისგან: [a, b] და [c, d]. რომ არ გადავტვირთოთ ახსნით, ოპერაცია prefix-სთვის გამოვიყენოთ ინფიქსური ფორმის ჩანაწერი:

Append ([a, b], [c, d]) = a : Append ([b], [c, d]) = a : (b : Append ([], [c, d])) = a : (b : ([c, d])) = a : (b : (c : (d : []))) = [a, b, c, d].

4. ფუნქციები, რომლებიც მუშაობენ სიებთან:

a. GetN:

```
GetN (N, []) = _
GetN (1, H:T) = H
GetN (N, H:T) = GetN (N - 1, T)
```

b. ListSumm:

```
ListSumm ([], L) = L
ListSumm (L, []) = L
ListSumm (H1:T1, H2:T2) = prefix ((H1 + H2),
ListSumm (T1, T2))
```

c. OddEven:

```
OddEven ([]) = []
OddEven ([X]) = [X]
OddEven (H1:[H2:T]) = prefix (prefix (H2, H1),
OddEven (T))
```

d. Reverse:

```
Reverse ([]) = []
Reverse (H:T) = Append (Reverse (T), [H])
```

e. Map:

```
Map (F, []) = []
Map (F, H:T) = prefix (F (H), Map (F, T))
```

სავარჯიშო №2

1. შემდეგი აღწერები განსაზღვრავს მოთხოვნილ ფუნქციებს. ზოგიერთ პუნქტში გამოყენებულია დამატებითი ფუნქციები, რომელთათვის გვერდის ავლა ძნელად წარმოგვიდგენია.

a. Reverse_all:

```
Reverse_all (L) = L when atom (L)
Reverse_all ([]) = []
Reverse_all (H:T) = Append (Reverse_all (T),
Reverse_all (H)) otherwise
```

b. Position:

```
Position (A, L) = Position_N (A, L, 0)

Position_N (A, (A:L), N) = N + 1
Position_N (A, (B:L), N) = Position_N (A, L, N + 1)
Position_N (A, [], N) = 0
```

c. Set:

```
Set ([]) = []
Set (H:T) = Include (H, Set (T))

Include (A, []) = [A]
Include (A, A:L) = A:L
Include (A, B:L) = prefix (B, Include (A, L))
```

d. Frequency:

```
Frequency L = F ([], L)

F (L, []) = L
F (L, H:T) = F (Corrector (H, L), T)

Corrector (A, []) = [A:1]
Corrector (A, (A:N):T) = prefix ((A:N + 1), T)
Corrector (A, H:T) = prefix (H, Corrector (A, T))
```

2. სავარჯიშო №1-ის ყველა ფუნქციისთვის შეიძლება ავაგოთ განმარტებები დამგროვებელი პარამეტრებით. მეორე მხრივ, შესაძლოა ზოგიერთი ახლადგანმარტებული ფუნქცია არ იყოს ოპტიმიზებული.

a. Power_A:

```
Power_A (X, N) = P (X, N, 1)

P (X, 0, A) = A
P (X, N, A) = P (X, N - 1, X * A)
```

b. Summ_T_A:

```
Summ_T_A (N) = ST (N, 0)

ST (0, A) = A
ST (N, A) = ST (N - 1, N + A)
```

c. Summ_P_A:

```
Summ_P_A (N) = SP (N, 0)

SP (0, A) = A
SP (N, A) = SP (N - 1, Summ_T_A (N) + A)
```

d. Summ_Power_A:

```
Summ_Power_A (N, P) = SPA (N, P, 0)

SPA (N, 0, A) = A
SPA (N, P, A) = SPA (N, P - 1, (1 / Power_A
(N, P)) + A)
```

e. Exponent_A:

```
Exponent_A (N, P) = E (N, P, 0)

E (N, 0, A) = A
E (N, P - 1, (Power_A (N, P) / Factorial_A
(P)) + A)
```

სავარჯიშო №3

1. სასრული სიები აიგება ან შეზღუდვების საშუალებით, რომლებიც ჩაიდება სიების გენერატორში, ან დამატებითი შემზღუდავი პარამეტრების გამოყენებით.

a. `[1 .. 20]`

b. `[1, 3 .. 40]` ან `[1, 3 .. 39]`

c. `[2, 4 .. 40]`

d. 2-ის ხარისხების სია ყველაზე მარტივად შეიძლება აიგოს ფუნქციის საშუალებით (აქ `reverse` — სიის შეტრიალების ფუნქციაა):

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

reverse (powerTwo 25)
```

e. 3-ის ხარისხების სია ყველაზე მარტივად შეიძლება აიგოს ფუნქციის საშუალებით (აქ `reverse` — სიის შეტრიალების ფუნქციაა):

```
powerThree 0 = []
powerThree n = (3 ^ n) : powerThree (n - 1)

reverse (powerThree 25)
```

f. წინა ორი სავარჯიშოსგან განსხვავებით, აქ შეიძლება ვისარგებლოთ ფუნქციით `map`, რომელიც მოცემულ ფუნქციას იყენებს სიის ყველა ელემენტზე:

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

map t_Fermat [1 .. 50]
```

g. ფერმას `(Fermat's)` პირამიდული `50` რიცხვის სიის აგებაც ასევე დაფუძნებულია `map` ფუნქციის გამოყენებაზე:

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

map p_Fermat [1 .. 50]
```

2. უსასრულო სიები აიგება ან შეუზღუდავი გენერატორების საშუალებით, ანდა კონსტრუქტორი ფუნქციების საშუალებით პარამეტრების შეზღუდვების გარეშე.

a. ფაქტორიალების უსასრულო სია:

```
numbersFrom n = n : numbersFrom (n + 1)
factorial n = f_a n 1
f_a 1 m = m
f_a n m = f_a (n - 1) (n * m)
map factorial (numbersFrom 1)
```

b. ნატურალური რიცხვის კვადრატების უსასრულო სია:

```
square n = n * n
map square (numbersFrom 1)
```

c. ნატურალური რიცხვის კუბების უსასრულო სია:

```
cube n = n ^ 3
map cube (numbersFrom 1)
```

d. ხუთის ხარისხების უსასრულო სია:

```
powerFive n = 5 ^ n
map powerFive (numbersFrom 1)
```

e. ნატურალური რიცხვების მეორე სუპერხარისხების უსასრულო სია:

```
superPower n 0 = n
superPower n p = (superPower n (p - 1)) ^ n

secondSuperPower n = superPower n 2
map secondSuperPower (numbersFrom 1)
```

სავარჯიშო №4

1. ყველა ქვემოთ მოყვანილი აღწერა Haskell-ზე არის ერთ-ერთი მრავალი შესაძლებლობიდან:

a. getN:

```
getN          :: [a] -> a
getN n []             = _
getN 1 (h:t) = h
getN n (h:t) = getN (n - 1) t
```

b. listSumm:

```
listSumm                    :: Ord (a) => [a] -> [a]
listSumm [] l               = l
listSumm l []               = l
listSumm (h1:t1) (h2:t2) = (h1 + h2) :
                              (listSumm t1 t2)
```

c. oddEven:

```
oddEven                        :: [a] -> [a]
oddEven []                     = []
oddEven [x]              = [x]
oddEven (h1:(h2:t))  = (h2:h1) : (oddEven t)
```

d. reverse:

```
append        :: [a] -> [a] -> [a]
append [] l   = l
append (h:t) l2       = h : (append t l2)
```

```
reverse                 :: [a] -> [a]
reverse []              = []
reverse (h:t)           = append (reverse t [h])
```

e. map:

```
map             :: (a -> b) -> [a] -> [b]
map f []                = []
map f (h:t)   = (f h) : (map f t)
```

2. ყველა ქვემოთ მოყვანილი აღწერა Haskell-ზე არის ერთ-ერთი მრავალი შესაძლებლობიდან:

a. reverseAll:

```
atom            :: ListStr (a) -> Bool
atom a           = True
atom _           = False

reverseAll            :: ListStr (a) -> ListStr (a)
reverseAll l = l
reverseAll []       = []
reverseAll (h:t)    = append (reverseAll t)
(reverseAll h)
```

b. position:

```
position                :: a -> [a] -> Integer
position a l = positionN a l 0

positionN               :: a -> [a] -> Integer ->
Integer
positionN a (a:t) n   = (n + 1)
positionN a (h:t) n   = positionN a t (n + 1)
positionN a [] n      = 0
```

c. set:

```
set          :: [a] -> [a]
set []       = []
set (h:t)            = include h (set t)

include              :: a -> [a] -> [a]
include a [] = [a]
include a (a:t)      = a : t
include a (b:t)      = b : (include a t)
```

d. frequency:

```
frequency            :: [a] -> [(a : Integer)]
frequency l  = f [] l

f            :: [a] -> [a] -> [(a : Integer)]
f l []       = l
f l (h:t)            = f (corrector h l) t

corrector            :: a -> [a] -> [(a :
Integer)]
corrector a []       = [(a : 1)]
corrector a (a:n):t  = (a : (n + 1)) : t
corrector a h:t      = h : (corrector a t)
```

3. ყველა ქვემოთ მოყვანილი აღწერა Haskell-ზე არის ერთ-ერთი მრავალი შესაძლებლობიდან:

a. Show:

```
class Show a where
     show    :: a -> a
```

b. Number:

```
class Number a where
      (+)     :: a -> a -> a
      (-)     :: a -> a -> a
      (*)     :: a -> a -> a
      (/)     :: a -> a -> a
```

c. String:

```
class String a where
      (++)            :: a -> a -> a
      length  :: a -> Integer
```

4. ყველა ქვემოთ მოყვანილი აღწერა Haskell-ზე არის ერთ-ერთი მრავალი შესაძლებლობიდან:

a. Integer:

```
instance Number Integer where
      x + y   = plusInteger x y
      x - y   = minusInteger x y
      x * y   = multInteger x y
      x / y   = divInteger x y

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

b. Real:

```
instance Number Real where
      x + y   = plusReal x y
...
```

c. Complex:

```
instance Number Complex where
     x + y   = plusComplex x y
...
```

d. WideString:

```
instance String WideString where
     x ++ y  = wideConcat x y
     length x        = wideLength x

wideConcat x y       = append x y
wideLength x = length x
```

სავარჯიშო №5

1. უპირველესად გვინდა ვთქვათ, რომ მონადა არის კონტეინერული ტიპი. მართლაც, სია – კონტეინერული ტიპია, რადგანაც სიის შიგნით არის სხვა ტიპის ელემენტები. სწორედ ეს არის ნაჩვენები მონადური ტიპის განმარტებაში:

```
class Monad m where
     (>>=)   :: m a -> (a -> m b) -> m b
     (>>)    :: m a -> m b -> m b
     return  :: a -> m a
     fail    :: String -> m a
```

ჩანაწერი „m a“ თითქოსდა გვიჩვენებს, რომ ტიპი a (აუცილებელია გვახსოვდეს, რომ კლასებისა და მონაცემთა სხვა ტიპე-

ბის აღწერისას სიმბოლოები `a`, `b` და ა.შ. აღნიშნავს ტიპების ცვლადებს) შემოსაზღვრულია მონადური ტიპით `m`. თუმცა რეალური ფიზიკური შემოსაზღვრა შესაძლებელია მხოლოდ ტიპისთვის „სია“, რადგანაც მისი აღნიშვნა კვადრატული ფრჩხილებით უკვე ტრადიციაა. `Haskell`-ის მკაცრ ნოტაციაში შეიძლება დაგვეწერა ასეთი რამ `List (1 2 3 4 5)` – ეს არის სია `[1, 2, 3, 4, 5]`.

2. ფუნქციონალურ ენებში მონადის გამოყენება, ფაქტობრივად, უკან დაბრუნებაა იმპერატიულისკენ. დაკავშირების ოპერაციების (`>>=` და `>>`) საშუალებით ხდება დაკავშირებული გამოსახულებების თანმიმდევრული შესრულება გამოთვლების შედეგების გადაცემით ან გადაცემის გარეშე. ანუ მონადები – ეს იმპერატიული ბირთვია ფუნქციონალური ენების შიგნით. ერთი მხრივ, ეს ეწინააღმდეგება ფუნქციონალური დაპროგრამების თეორიას, მაგრამ, მეორე მხრივ, ზოგიერთი ამოცანა იხსნება მხოლოდ იმპერატიული ენების პრინციპებით. და კიდევ, `Haskell` იძლევა სიების შექმნის უნიკალურ შესაძლებლობას, მაგრამ იმის ხარჯზე, რომ ტიპი „სია“ შესრულებულია მონადის სახით.

სავარჯიშო №6

1. ფუნქცია `insert`-ის ერთ-ერთი შესაძლო ვარიანტი დესტრუქციული მინიჭებით:

```
-- ასე გაკეთება არ შეიძლება. Lisp-ში არის
დესტრუქციული მინიჭების
-- გამოყენების შესაძლებლობა
```

```
replace_root A T - ფუნქცია ამატებს ელემენტს ხის
ფესვში replace_left K (Root x Emptic x Right)
=> (Root x (K x Emptic x Emptic) x Right)
replace_right K (Root x Left x Emptic) =>
(Root x Left (K x Emptic x Emptic))

-- ფუნქცია insert

insert K Emptic = cbtree K ctree ctree
insert (A:L) ((A1:L1) x Left x Right) = insert
(A:L) Left when ((A < A1) & nonEmpty Left)
insert (A:L) ((A1:L1) x Emptic x Right) =
replace_left (A:L) ((A1:L1) x Emptic x Right)
when (A < A1)
insert (A:L) ((A1:L1) x Left x Right) = insert
(A:L) Right when ((A > A1) & nonEmpty Right)
insert (A:L) ((A1:L1) x Left x Emptic) =
replace_right (A:L) ((A1:L1) x Left x Emptic)
when (A > A1)
insert A T = replace_root A T otherwise
```

გამომცემლობის რედაქტორი მარინე ვარამაშვილი

გარეკანის დიზაინი ნინო ებრალიძე

დაკაბადონება ნინო ვაჩეიშვილი

გამოცემის მენეჯერი მარიკა ერქომაიშვილი

0179, თბილისი, ი. ჭავჭავაძის გამზირი 14

14, Ilia Tchavtchavadze Ave., Tbilisi 0179

Tel: 995(32) 2 25 14 32

www.press.tsu.ge