საქართველოს ტექნიკური უნივერსიტეტი

არჩილ ფრანგიშვილი, ზურაბ წვერაიძე,

†ბადრი ბარდაველიძე, ოლეგ ნამიჩეიშვილი

ფუნქციონალური დაპროგრამების ენა ჰასკელის ლაბორატორიული პრაქტიკუმი

რეკომენდებულია სტუ-ის
სარედაქციო-საგამომცემლო საბჭოს
მიერ. 09.03.2012, ოქმი №1

თბილისი

2012

უაკ 004.4

მოცემული წიგნი საქართველოში პირველი გამოცემაა, რომელშიც განხილულია დაპროგრამების ფუნქციონალური ენა ჰასკელის ლაბორატორიული სამუშაოები. ის მიზნად ისახავს ჩვენს ქვეყანაში ისეთი ინფორმატიკოსის, ფიზიკოსის, კიბერნეტიკოსის, ხელოვნური ინტელექტის სპეციალისტის, კრიპტოლოგისა და მათემატიკოსის მომზადების ხელშეწყობას, რომელიც თავისი პრაქტიკული საქმიანობისას ფუნქციონალური დაპროგრამების პარადიგმას მიმართავს. გასაგებია, რომ გამოყენებით საფუძვლებს ამ სფეროში ვასწავლით ჰასკელის მეშვეობით, ვინაიდან დღეს იგი ფუნქციონალური დაპროგრამების განსაკუთრებით მძლავრი და დასრულებული ინსტრუმენტია, ხოლო განვითარების ტემპით კომერციულად ყველაზე გავრცელებულ ენებსაც კი უსწრებს.

წიგნი გამოადგება კომბინატორული ლოგიკის, λ-აღრიცხვისა და პარალელური გამოთვლების შემსწავლელებსაც.

იგი ასევე საინტერესო იქნება ყველა იმ პირისათვის, ვინც სერიოზულად ეკიდება ახალ კომპიუტერულ ტექნოლოგიებს, ხელოვნური ინტელექტისა და საექსპერტო სისტემების პრობლემებს.

სამეცნიერო რედაქტორი პროფ. დავით გორდეზიანი

რეცენზენტები: პროფესორი კოტე კამკამიძე,

პროფესორი ჰამლეტ მელაძე

ISBN 978-9941-20-059-5

http://www.gtu.ge/publishinghouse/

სარჩევი

წინასიტყვაობა

ტრადიციულად ევროპის ქვეყნების, რუსეთისა და იაპონიის უნივერსიტეტებში ფუნქციონალური დაპროგრამების საფუძვლები იკითხებოდა მე-20 საუკუნის მეორე ნახევარში შექმნილი Lisp ენის მაგალითზე, ხოლო ლაბორატორიული სამუშაოები ტარდებოდა μ-Lisp ვერსიით [1]. Lisp-ის დამუშავების დროიდან ფუნქციონალური დაპროგრამების მრავალი ახალი თეორიული მექანიზმი, ფორმალიზმი და მეთოდოლოგია გაჩნდა, რაც Haskell-98 უნიფიცირებული სტანდარტის შექმნით დაგვირგვინდა, ხოლო შემდეგ იგი დაპროგრამების ფუნქციონალური ენა გახდა [2].

Haskell-98 სტანდარტი დღემდე რჩება ფუნქციონალური დაპროგრამების განვითარების «ნიშანსვეტად». ამიტომ ამ საკითხით დაინტერესებული ადამიანი ახალი სტანდარტისა და ახალი ენის საფუძვლებს მაინც უნდა იცნობდეს. მიუხედავად იმისა, რომ ინგლისურ ენაზე რამდენიმე ნაშრომია გამოცემული, Haskell ენის გამოყენება მაინც ფერხდება, ვინაიდან მცირე რაოდენობითაა ხელმისაწვდომი ლიტერატურა ისეთ «უსაზღვრო» წყაროშიც კი, როგორიც ინტერნეტია.

წინამდებარე პრაქტიკუმი აგრძელებს სასწავლო-მეთოდიკური ლიტერატურის სერიას, რომელიც განკუთვნილია ჩვენს ქვეყანაში უნივერსიტეტების ტექნიკური, ზუსტი და საბუნებისმეტყველო სპეციალობების შემსწავლელ სტუდენტთა ხელშესაწყობად, განსაკუთრებით ინფორმატიკის, ფიზიკის, კიბერნეტიკის, გამოყენებითი მათემატიკის, ხელოვნური ინტელექტისა და კრიპტოლოგიის სფეროებიდან.

ფუნქციონალური დაპროგრამების პარადიგმა ეფუძნება «ფუნქციის» მათემატიკურ ცნებას, რაც ეფექტური გამოთვლითი პროგრამების შექმნის საშუალებას იძლევა. გარდა ამისა, ფუნქციონალური დაპროგრამებით ხერხდება ეფექტური გამოთვლების ჩატარება სიმბოლოებისა და არა რიცხვების მეშვეობით. ამიტომ, ამ ფაქტმა ცხადი ასახვა პოვა ხელოვნურ ინტელექტში [3], [4], [5].

ფუნქციონალური დაპროგრამების თეორიას ჯერ კიდევ მეოცე საუკუნის ოციან წლებში ჩაეყარა საფუძველი ისეთი მძლავრი გამოთვლითი ფორმალიზმების დამუშავების შემდეგ, როგორიცაა კომბინატორული ლოგიკა და λ-აღრიცხვა [6]. მოგვიანებით λ-აღრიცხვა ყველა შექმნილი ფუნქციონალური ენის ბაზისი გახდა, დაწყებული Lisp ფუნქციონალური ენით და დამთავრებული უახლესი Haskell-98-ით.

მიუხედავად იმისა, რომ, ჯერჯერობით, პროგრამული უზრუნველყოფის დამუშავებისა და შექმნის სამყაროში ყველაზე მაღალი პოზიცია ობიექტზე ორიენტირებული დაპროგრამების პარადიგმას უკავია, ფუნქციონალური დაპროგრამების პრინციპები ფუძემდებლური ხდება. დაპროგრამების ყველაზე თანამედროვე ტექნოლოგიებიც კი გვერდს ვერ უვლის ფუნქციის ცნებას, ამიტომ ობიექტზე ორიენტირებული დაპროგრამება ფუნქციონალური დაპროგრამების მრავალი პრინციპის გამოყენებას თავს ვერ არიდებს.

მრავალი ალგორითმის უფრო ეფექტურად რეალიზება სუფთა ფუნქციათა საშუალებით შეიძლება. პირველ რიგში, ეს ეხება დახარისხებასა და ძებნას. ზოგიერთი ამოცანა კი, როგორიცაა ერთი ფორმალიზებული ენის

გარდაქმნა მეორე ენად ან კონსტრუქციათა სინტაქსური გარჩევა, მხოლოდ ფუნქციონალური დაპროგრამების მეთოდთა გამოყენებითაა შესაძლებელი.

ამიტომ, ფუნქციონალური დაპროგრამების პრინციპების საშუალებით იხსნება ისეთი ამოცანები, როგორიცაა სიმბოლური გამოთვლები, მათემატიკური წინადადების კომპიუტერული მტკიცება, დასკვნა ცოდნის საფუძველზე, ფორმალიზებული და ბუნებრივი ენების დამუშავება, ცოდნის გამოვლენა მონაცემთა ბაზებში, ურთიერთქმედი პროგრამული სისტემების (აგენტების) შექმნა, აგრეთვე ზოგიერთი სხვა ამოცანა. ყველა ეს პრობლემა ტრადიციულია ხელოვნური ინტელექტის სფეროსათვის [7], [8].

ლაბორატორიული პრაქტიკუმი შედგება ორი ნაწილისაგან. პაქტიკუმის პირველი ნაწილი ეთმობა HUGS 98 ინსტრუმენტული საშუალების აღწერას, ხოლო მეორე ნაწილი განკუთვნილია ლაბორატორიული სამუშაოების შესასრულებლად «ფუნქციონალური დაპროგრამება Haskell ენაზე» საგნის ფარგლებში ხსენებული ინსტრუმენტული საშუალებით.

იგი შედგება ექვსი ლაბორატორიული დავალებისაგან, რომლებიც დაყოფილია ნაწილებად, გადასაწყვეტი ამოცანების სირთულის შესაბამისად.

ავტორები მადლობას უხდიან კოლეგებს მნიშვნელოვანი დახმარებისა და სასარგებლო რჩევებისათვის, რამაც მნიშვნელოვნად შეუწყო ხელი ამ პრაქტიკუმის შექმნას.

შესავალი

Haskell ენის სპეციფიკაციამ, რომელიც 1998 წელს შეიქმნა, არნახული მხარდაჭერა პოვა სამეცნიერო წრეებში და, პირველ რიგში, ევროპელ და იაპონელ სწავლულებს შორის. ენას ასე ჰქვია ფუნქციონალური დაპროგრამების ერთ-ერთი ფუძემდებლის, ამერიკელი მათემატიკოსისა და ლოგიკოსის ჰასკელ კარის (Haskell Brooks Curry , 1900-1982) პატივისცემის ნიშნად. ამასთან დაკავშირებით სულ რაღაც ხუთ-ექვს თვეში რამდენიმე კვლევითმა ჯგუფმა და კომერციულმა საწარმომ შექმნა Haskell ენის რეალიზაციები, როგორც ინტერპრეტატორების, ისე კომპილატორების სახით. მათ შორის მხოლოდ ზოგიერთია უფასო, ხოლო დანარჩენი კომერციული პროგრამული პროდუქტია.

ყველაზე საინტერესო ინსტრუმენტული საშუალება, რომელიც მსოფლიოს მრავალ უნივერსიტეტში გამოიყენება ფუნქციონალური დაპროგრამების შესწავლისას, არის HUGS 98. იგი შეიცავს Haskell ენის 1998 წლის სტანდარტის (შემდეგ Haskell-98) თვით ინტერპრეტატორს, აგრეთვე დაპროგრამების ინტეგრირებულ გარემოს.

გარდა ამისა, HUGS 98 ინსტრუმენტული გარემო აბსოლუტურად უფასო პროგრამული საშუალებაა და მისი მიღება თავისუფლად შეიძლება ინტერნეტით (www.haskell.org). ეს დამატებით უწყობს ხელს ხსენებული ინსტრუმენტული გარემოს გავრცელებას, როგორც სწავლების შესანიშნავ საშუალებას, თუმცა მას არაერთი ნაკლი აქვს Haskell-98 ენის კომერციულ რეალიზაციასთან შედარებით. ამის მიუხედავად, ლაბორატორიულ სამუშაოთა ეს კრებული HUGS 98 გარემოს იყენებს.

I ნაწილი: HUGS 98-თან მუშაობის საფუძვლები

HUGS 98 ინტეგრირებული გარემოს გაშვებისას ეკრანზე ჩნდება პროგრამათა დაპროექტების ამ საშუალების (ინგლ. *integrated development environment*) დიალოგური ფანჯარა.

შემდეგ ავტომატურად იტვირთება Haskell ენის ტიპების წინასწარი განსაზღვრებებისა და სტანდარტულ ფუნქციათა აღწერის სპეციალური Prelude.hs ფაილი.

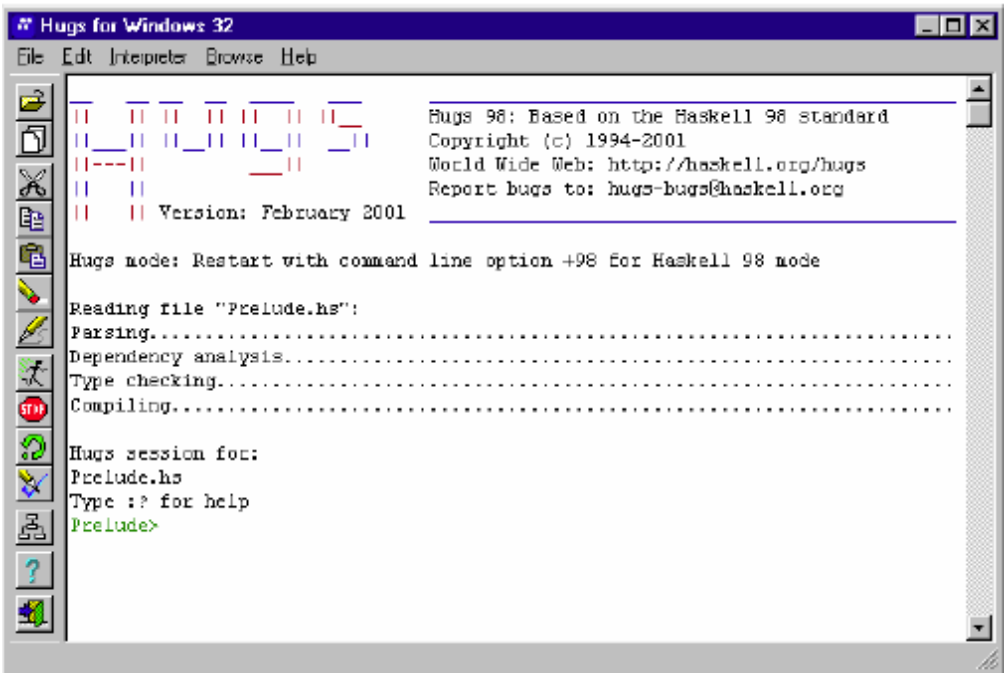
ბოლოს კი გამოჩნდება სტანდარტული მოწვევა სამუშაოდ.

პროგრამათა დაპროექტების ინტეგრირებული გარემოს დიალოგური ფანჯარა შეიცავს სისტემის მთავარ მენიუს.

აქვეა ღილაკების ნაკრები სწრაფი შეღწევისათვის განსაკუთრებით ხშირად გამოსაყენებელ ბრძანებებსა და კონსოლში, სადაც მიმდინარეობს ინტერპრეტატორთან მუშაობა.

საგანგებოდ უნდა აღინიშნოს, რომ ინტეგრირებული გარემო არ იძლევა პროგრამის კოდების შემცველი ფაილების შექმნისა და რედაქტირების საშუალებას, ამიტომ საჭიროა ნებისმიერი ტექსტური რედაქტორის გამოყენება, რომელსაც შეუძლია, ჩვეულებრივ, TXT სტანდარტთან მუშაობა.

ასეთ რედაქტორად მინიშნების გარეშე შეთანხმების საფუძველზე, ჩვეულებრივ, გამოიყენება ნებისმიერი ვერსიის Windows-ის კარგად ცნობილი Notepad ბლოკნოტი.



სურ. 1. HUGS 98 ინტეგრირებული გარემოს მთავარი ფანჯარა

HUGS 98-ის მთავარი ფანჯარა (სურ. 1) უზრუნველყოფს ინტეგრირებული გარემოს ყველა ბრძანების მისაწვდომობას, რომლებიც პროგრამათა ინტერპრეტაციისათვის და გასამართად გამოიყენება.

გარდა ამისა, ინტეგრირებული გარემო საშუალებას იძლევა გამოვიძახოთ რედაქტირებისათვის გახსნილი მოდულები Windows ოპერაციული სისტემის ნებისმიერი ვერსიის Notepad ტექსტური რედაქტორის სტანდარტულ ფანჯარაში.

1. HUGS 98-ის ინსტრუმენტების პანელი

ინსტრუმენტების პანელზე, რომელიც მთავარი დიალოგური ფანჯრის მარცხენა სვეტია, განთავსებულია ღილაკები. მათი მეშვეობით შეიძლება იმ ბრძანების გამოძახება, რომელიც განსაკუთრებით ხშირად გამოიყენება პროგრამის დამუშავების პროცესში (ცხადია, HUGS 98 ინტეგრირებული გარემოს დამპროექტებლის და არა საბოლოო მომხმარებლის თვალსაზრისით). ქვემოთ მოცემულია ყველა იმ ღილაკის მოკლე აღწერა, რომლებიც ინსტრუმენტების პანელზეა განთავსებული. ისინი სულ თოთხმეტია

მოდულის ჩატვირთვა გარე ფაილიდან. ეს ღილაკი საშუალებას იძლევა ავირჩიოთ და გავხსნათ ფაილი, საიდანაც ხდება ინტერპრეტატორის მიერ ამ ფაილში ნაპოვნი ყველა მოდულის ჩატვირთვა.

მოდულის მენეჯერის გამოძახება. მოდულის მენეჯერი საშუალებას იძლევა დავუმატოთ, გავაუქმოთ და გავასწოროთ ინტეგრირებული გარემოს მეხსიერებაში ჩატვირთული პროგრამული მოდული.

მონიშნული ტექსტის ამოჭრა. ეს ოპერაცია ტექსტის რედაქტირების სტანდარტული ფუნქციაა. ახორციელებს რედაქტორიდან მონიშნული ტექსტის ამოღებას და მის მოთავსებას ოპერაციული სისტემის გაცვლის ბუფერში.

მონიშნული ტექსტის პირის შექმნა გაცვლის ბუფერში. ეს ღილაკი ტექსტის რედაქტირების სტანდარტულ ფუნქციას აღნიშნავს. ამ

ღილაკის მეშვეობით შესაძლებელია მონიშნული ტექსტის პირის შექმნა და მისი მოთავსება ოპერაციული სიტემის გაცვლის ბუფერში.

ტექსტის ჩასმა გაცვლის ბუფერიდან. ეს ოპერაცია ტექსტის რედაქტირების სტანდარტული ფუნქციაა. ტექსტში, რომლის რედაქტირებაც ხდება, ათავსებს ოპერაციული სისტემის გაცვლის ბუფერის შიგთავსს.

არჩეული ტექსტის გასუფთავება. ეს ოპერაცია ტექსტის რედაქტირების სტანდარტული ფუნქციაა, იგი ახორციელებს მონიშნული ტექსტის ამოღებას რედაქტორიდან, მაგრამ არ ათავსებს ამ ტექსტს ოპერაციული სისტემის გაცვლის ბუფერში.

ტექსტის გარე რედაქტორის ამოქმედება. ამ ღილაკის მეშვეობით ხორციელდება ოპერაციულ სისტემაში დარეგისტრირებული გარე ტექსტური რედაქტორის გაშვება. მაგალითად, Windows-ის ოჯახისთვის ამ ღილაკის დაჭერისას ხდება სტანდარტული Notepad პროგრამის გაშვება.

გაშვება «main» გამოსახულების შესასრულებლად. ეს ღილაკი ჩატვირთულ მოდულში main ფუნქციას ასრულებს (ცხადია, თუ ასეთი ფუნქცია აღმოჩნდება მოდულში). თუ main ფუნქცია არც ერთ ჩატვირთულ მოდულში არ მოიძებნა, გამოჩნდება შეტყობინება შეცდომის შესახებ: ERROR — Undefined variable "main".

პროგრამის შესრულების გაჩერება. ამ ღილაკის მეშვეობით ხდება ნებისმიერი გაშვებული ფუნქციის შესრულების გაჩერება. მაგალი-

თად, იგი გამოიყენება გამოთვლის უსასრულო სიის შესაწყვეტად.

მიმდინარე პროექტის ყველა ფაილის გადატვირთვა. ამ ღილაკის მეშვეობით ხორციელდება ყველა ფაილის გადატვირთვა იმისათვის, რომ მოთავსდეს ინტერპრეტატორის მეხსიერებაში პროექტის კოდთან დაკავშირებული ყველა ცვლილება.

ინტერპრეტატორის პარამეტრების დაყენება. ეკრანზე გამოაქვს Haskell ენის ინტერპრეტატორის პარამეტრთა დაყენების დიალოგური ფანჯარა. ინტერპრეტატორის პარამეტრების შესახებ საუბარი მოგვიანებით გვექნება.

კლასების იერარქიის გამოტანა ეკრანზე. ამ ღილაკის მეშვეობით ეკრანზე ჩნდება მიმდინარე პროექტის კლასების იერარქია, რომელიც ნაჩვენებია მართკუთხედების სიმრავლის და მათ შორის კავშირების სახით. მართკუთხედებს აწერია სახელები (კლასები), ხოლო მათ შორის კავშირებით ასახულია მემკვიდრეობითობის ურთიერთდამოკიდებულება.

ცნობარის გამოძახება. ამ ღილაკის მეშვეობით ეკრანზე გამოდის საცნობო ინფორმაციის სტანდარტული დიალოგური ფანჯარა. იგულისხმება, რომ ყველა საცნობო ფაილი მოთავსებულია კატალოგში, სადაც დაყენებულია ინტეგრირებული გარემო (ეს ფაილები არ შედის HUGS 98-ის სტანდარტულ კომპლექტში).

პროგრამიდან გამოსვლა. ღილაკის მეშვეობით ხდება HUGS 98 ინტეგრირებული გარემოდან ოპერაციულ სისტემაში გამოსვლა.

2. HUGS 98-ის კონსოლის ბრძანებები

HUGS 98 ინტეგრირებული გარემოს კონსოლი დამხმარე კონსტრუქციების მცირე ნაკრებს იძლევა ამ გარემოს მუშაობის სამართავად. მათ შორის მრავალი ბრძანება ინსტრუმენტების პანელის ღილაკთა მოქმედებების და მთავარი მენიუს ზოგიერთი პუნქტის დუბლირებას ახდენს. მაგრამ ნებისმიერ შემთხვევაში ამ ბრძანებებს შეუძლია გამოცდილ მომხმარებელს დამუშავების პროცესის მნიშვნელოვნად დაჩქარების საშუალება მისცეს.

ყოველი ბრძანება იწყება სიმბოლო «ორწერტილით» – « : », რათა ერთმანეთისაგან განვასხვაოთ ჩაშენებული ბრძანებები დამპროექტებლის მიერ დაწერილი ფუნქციებისაგან. გარდა ამისა, ინტეგრირებული გარემო ყოველი ბრძანების ერთ ასომდე შეკვეცის საშუალებას იძლევა, როდესაც საკმარისია სიმბოლო «ორწერტილის» და ბრძანების მხოლოდ პირველი ასოს აკრეფა.

სულ ცხრამეტი ასეთი ბრძანება არსებობს. ქვემოთ თითოეული მათგანი დაწვრილებითაა აღწერილი.

:load [<filenames>]

ჩატვირთავს პროგრამულ მოდულს მოცემული ფაილიდან (ფაილების სახელების განცალკევება ცარიელი პოზიციით შეიძლება). იმეორებს ინსტრუმენტების პანელის მოდულის ჩატვირთვის ღილაკის ფუნქციას. თუ ფაილის სახელი მითითებული არ არის, მაშინ გადმოიტვირთება ყველა მოდული (Prelude.hs) სტანდარტულის გარდა. ბრძანების

ხელმეორედ გამოყენებისას ინტერპრეტატორის მეხსიერებიდან გადმოიტვირთება ყველა ადრე ჩატვირთული მოდული.

:also <filenames>

ჩატვირთავს დამატებით მოდულს მიმდინარე პროექტში. ფაილების სახელები განცალკევებული უნდა იყოს ცარიელი პოზიციით (ერთზე მეტი ფაილის მითითების შემთხვევაში).

:reload

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

:project <filename>

ჩატვირთავს და გამოიყენებს პროექტის ფაილს. შეიძლება მხოლოდ ერთი ფაილის ჩატვირთვა. პროექტის ფაილები გამოიყენება დაცალკევებული კოდიანი ფაილების გასაერთიანებლად. ბრძანების ხელახლა გამოყენებისას ყველა ფაილი (როგორც პროექტის, ისე ჩვეულებრივი) გადმოიტვირთება ინტერპრეტატორის მეხსიერებიდან.

:edit [<filename>]

იძახებს გარე ტექსტურ რედაქტორს მითითებული ფაილის გასასწორებლად. თუ ფაილის სახელი მითითებული არ არის, მაშინ რედაქტირებისათვის იძახებს ბოლო (ჩატვირთულ ან რედაქტირებულ) ფაილს. მოცემული ბრძანება ფაქტობრივად იმავე ფუნქციას ასრულებს, რასაც

გარე ტექსტური რედაქტორის გამოძახების ინსტრუმენტული პანელის ღილაკი.

:module <module>

მოცემულ მოდულს აცხადებს მიმდინარე მოდულად ფუნქციების შესასრულებლად. ეს ბრძანება, უპირველეს ყოვლისა, განკუთვნილია სახელების კოლიზიათა გადასაჭრელად.

<expr>

მოცემული გამოსახულების გაშვება შესასრულებლად. მაგალითად, `main` ბრძანება შესასრულებლად გაუშვებს შესაბამის `main` ფუნქციას, რითაც მოხდება ინსტრუმენტების პანელის ერთ-ერთი ღილაკის დუბლირება.

:type <expr>

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

:?

ეკრანზე გამოაქვს ბრძანებათა სია და მათი მოკლე აღწერა.

:set [<options>]

უზრუნველყოფს ბრძანებათა სტრიქონიდან ინტეგრირებული გარემოს პარამეტრების მოცემის საშუალებას. იმეორებს `HUGS 98`-ის პარამეტრთა

შერჩევის დიალოგური ფანჯრის მოქმედებას (HUGS 98-ს მოგვიანებით საგანგებოდ აღვწერთ).

:names [pat]

ეკრანზე გამოაქვს ობიექტის იდენტიფიკატორი, რომელიც სახელების მიმდინარე სივრცეშია (თუ არ არის მოცემული სხვა სივრცე).

:info <names>

ეკრანზე გამოაქვს ობიექტის მოცემული სახელის აღწერა. მაგალითად, ფუნქციისათვის მოცემული ფუნქციის სახელთან ერთად გამოაქვს მისი ტიპი.

:browse <modules>

ეკრანზე გამოაქვს მოცემულ მოდულში განსაზღვრული ყველა ობიექტის (ფუნქცია, ცვლადი, ტიპი) სია. მოდულის სახელი უნდა იყოს გამოცალკევებული ცარიელი პოზიციით (იმ შემთხვევაში, თუ მითითებულია ერთზე მეტი მოდულის სახელი).

:find <name>

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

:!<command>

გამოდის ოპერაციულ სისტემაში და ასრულებს მოცემულ ბრძანებას. ხაზგასმით უნდა აღინიშნოს ის გარემოება, რომ სიმბოლო «ძახილის ნიშანსა» და ოპერაციული სიტემის ბრძანების სახელს შორის ცარიელი პოზიცია (ე.ი. შუალედი) დაუშვებელია.

:cd <directory>

ცვლის მიმდინარე კატალოგს, რომელთანაც მუშაობს HUGS 98 ინტეგრირებული გარემო.

:gc

იძულებით იწვევს ნაგვის მოგროვების პროცესის დაწყებას. ამის შემდეგ ეკრანზე გამოაქვს სტატისტიკა მეხსიერების შეგროვებული და აღდგენილი უჯრედების შესახებ.

:version

ეკრანზე გამოაქვს ინფორმაცია Haskell ენის დაყენებული ინტერპრეტატორისა და HUGS 98 ინტეგრირებული გარემოს შესახებ.

:quit

ოპერაციულ სისტემაში გასვლა. იმეორებს ინსტრუმენტთა პანელის სათანადო ღილაკის დანიშნულებას.

3. გამართვის დამატებითი შესაძლებლობები

HUGS 98 ინტეგრირებული გარემო რამდენიმე დამატებით შესაძლებლობას აძლევს დამპროექტებელს პროგრამის გასამართავად, რაც მრავალ შემთხვევაში აადვილებს დამუშავებას და საშუალებას იძლევა უფრო ფართო კუთხით შევხედოთ შექმნილ გამოყენებით პროდუქტს. ასეთი დამატებითი შესაძლებლობების რიცხვს მიეკუთვნება კლასების, ობიექტების დარეგისტრირებული სახელების, ტიპების კონსტრუქტორთა და კლასთა იერარქიის დათვალიერება.

თითოეულ ამ დამატებით შესაძლებლობას შემდგომში დაწვრილებით განვიხილავთ, მაგრამ წინასწარ საჭიროა იმ აღნიშვნათა მნიშვნელობების ახსნა-განმარტება, რომლებიც გამოიყენება დათვალიერების ყველა ინსტრუმენტში. ასეთ აღნიშვნათა რიცხვს მიეკუთვნება მართკუთხა ფერადფერადი პიქტოგრამები, რომლებშიც ჩაწერილია სხვადასხვა ასო. სულ ცხრა პიქტოგრამაა:

• ლურჯი მართკუთხედი C ასოთი — კლასის აღნიშვნა (ინგლ. class).

• წითელი მართკუთხედი I ასოთი — კლასის ეგზემპლარის აღნიშვნა (ინგლ. instance).

• ვარდისფერი მართკუთხედი M ასოთი — კლასის წევრის აღნიშვნა (ინგლ. member).

• ლურჯი მართკუთხედი N ასოთი — ფუნქციის სახელის აღნიშვნა (ინგლ. name).

• წითელი მართკუთხედი `P` ასოთი — პრიმიტივის აღნიშვნა (ინგლ. `primitive`). პრიმიტივი განსხვავდება ჩვეულებრივი ფუნქციისაგან იმით, რომ მისი განსაზღვრება ფაილში კი არ არის, არამედ ჩაშენებულია ინტერპრეტატორში. პრიმიტივები წარმოდგენილია მხოლოდ `Prelude.hs` ფაილში.

• მწვანე მართკუთხედი `C` ასოთი — ტიპის კონსტრუქტორის აღნიშვნა (ინგლ. `constructor`).

• ლურჯი მართკუთხედი `D` ასოთი — მონაცემთა ტიპის აღნიშვნა (ინგლ. `data`).

• წითელი მართკუთხედი `S` ასოთი — ჩაშენებული ტიპის აღნიშვნა.

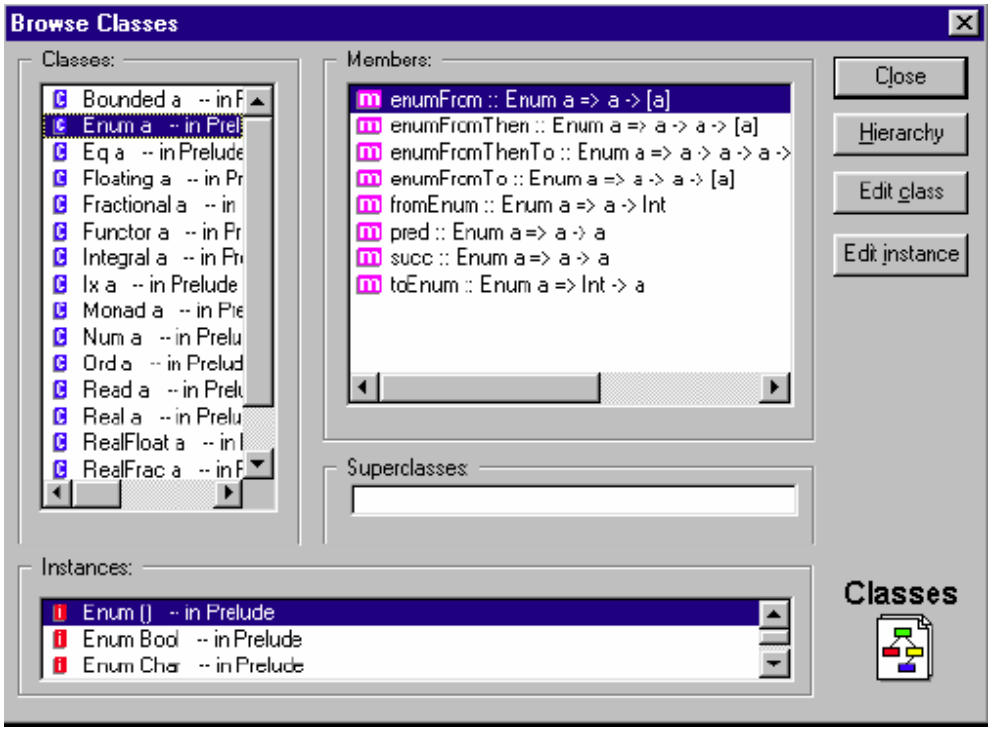
• ვარდისფერი მართკუთხედი `N` ასოთი — ახალი ტიპის აღნიშვნა (ინგლ. `new type`).

3.1. კლასის დათვალიერება

კლასის დასათვალიერებელი ინსტრუმენტის საშუალებით დამპროექტებელს შეუძლია შეისწავლოს:

- შექმნილ კლასთა სია,
- ყოველი კლასის წევრ ფუნქციათა სია (თუ ასეთი არის),
- თითოეული კლასის ეგზემპლართა სია (თუ ასეთი არის).

ეს ინსტრუმენტი გამოსახულია მე-2 სურათზე.



სურ. 2. დიალოგური ფანჯარა კლასების დასათვალიერებლად

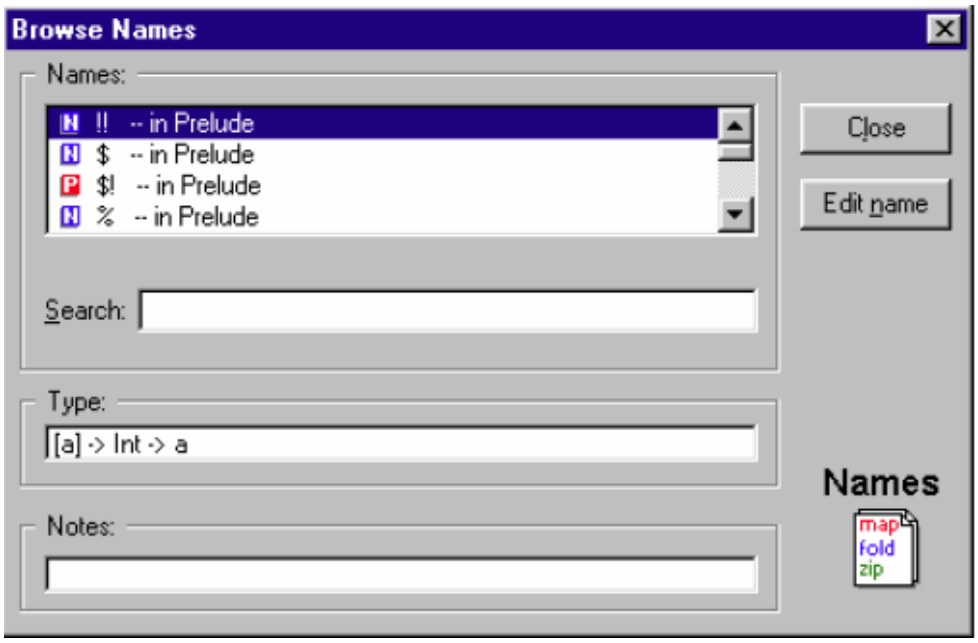
მარცხენა სვეტში მოცემულია შექმნილი კლასების სია. რომელიმე გარკვეული კლასის გამოყოფისას მარჯვენა ველში ჩნდება კლასის წევრ ფუნქციათა სია, ხოლო ქვედა ველში – კლასის ეგზემპლარების სია. «Superclasses» დასახელების ველში კი იწერება მონიშნული კლასის საბაზო კლასები (ცხადია, თუ ასეთი არის).

ეს დიალოგური ფანჯარა იძლევა ავტომატური გადასვლის საშუალებას მონიშნული კლასის ან კლასის მონიშნული ეგზემპლარის აღწერაზე მათი რედაქტირების დროს (შესაბამისად «Edit class» და «Edit instance» ღილაკების გამოყენებისას). გარდა ამისა, შეიძლება გადა-

სვლა კლასების იერარქიის დათვალიერებაზე («Hierarchy» ღილაკზე დაჭერით).

3.2. ობიექტების დარეგისტრირებული სახელების დათვალიერება

ობიექტების დარეგისტრირებული სახელების დასათვალიერებელი ინსტრუმენტის საშუალებით პროგრამისტს შეუძლია იმ სახელების სიის შესწავლა, რომლებიც გვხვდება ყველა ჩატვირთულ მოდულში. სახელებს მიეკუთვნება: ფუნქციის სახელი, პრიმიტივის (ფუნქციის, რომლის რეალიზაცია «ჩაკერებულია» ინტერპრეტატორში) სახელი, მონაცემთა კონსტრუქტორის სახელი და კლასის წევრი ფუნქციის სახელი. ეს ინსტრუმენტი გამოსახულია მე-3 სურათზე.



სურ. 3. სახელების დათვალიერების დიალოგური ფანჯარა

ზედა ველში მოცემულია სახელების სია და შესაბამისი პიქტოგრამები, რომლებიც სახელის ბუნებას აღნიშნავს. ძებნის სტრიქონის საშუალებით შეიძლება განხორციელდეს ინკრემენტული ძებნა მთელ სიაში: მორიგი ასოს შეყვანისას, სიაში კურსორი გადადის პირველივე სახელზე, რომელიც სიმბოლოთა შეყვანილი თანამიმდევრობით იწყება. ორ ქვედა ველში მოცემულია დამატებითი ინფორმაცია მონიშნული სახელის შესახებ — მისი ტიპი და კომენტარები (თუ ასეთი არის აღწერაში).

ეს დიალოგური ფანჯარა ასევე საშუალებას აძლევს დამპროექტებელს სწრაფად გადავიდეს მონიშნული სახელის რედაქტირებაზე «Edit name» ღილაკზე დაჭერით.

3.3. ტიპების კონსტრუქტორთა დათვალიერება

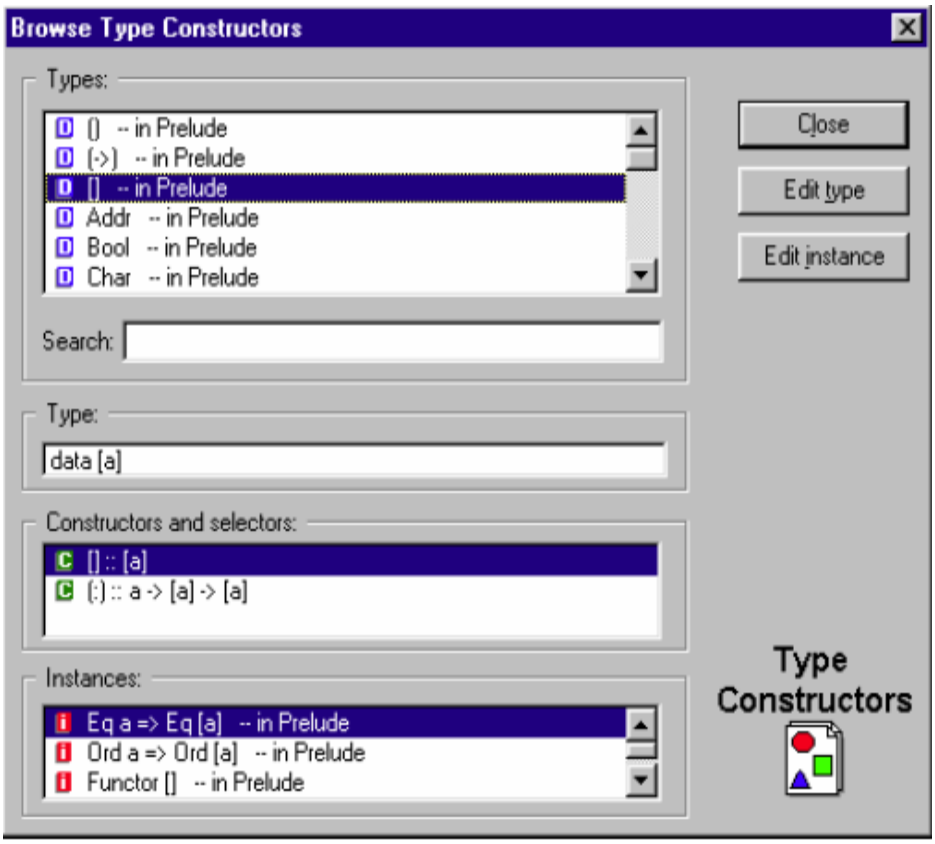
ტიპების კონსტრუქტორთა დათვალიერების ინსტრუმენტით დამპროექტებელს შეუძლია შეისწავლოს იმ კონსტრუქტორების სია, რომლებიც ყველა ჩატვირთულ მოდულში გვხვდება. კონსტრუქტორებს მიეკუთვნება: მონაცემთა კონსტრუქტორი (`data` დამხმარე სიტყვა), ჩაშენებული ტიპის აღწერა (`type` დამხმარე სიტყვა) და ახალი ტიპის კონსტრუქტორი (`newtype` დამხმარე სიტყვა). ეს ინსტრუმენტი გამოსახულია მე-4 სურათზე.

ზედა ველში მოცემულია ტიპების კონსტრუქტორთა სახელების სია შესაბამისი პიქტოგრამით, რომელიც კონსტრუქტორის ბუნებას აღნიშნავს.

ძებნის სტრიქონის საშუალებით შეიძლება ინკრემენტული ძებნა მთელ სიაში: მორიგი ასოს შეყვანისას კურსორი სიაში გადადის პირველივე სახელზე, რომელიც სიმბოლოთა შეყვანილი თანამიმდევრობით იწყება.

«Type» ველში მოიცემა შესაბამისი ტიპის განსაზღვრება.

ქვედა ორ ველში აისახება ინფორმაცია მონიშნული ტიპის კონსტრუქტორებსა და სელექტორებზე, ასევე ტიპის ეგზემპლარებზეც (ცხადია, თუ ასეთი არსებობს).



სურ. 4. ტიპების კონსტრუქტორთა დათვალიერების ფანჯარა

ამ დიალოგური ფანჯრის საშუალებით დამპროექტებელს შეუძლია გადასვლა მონიშნული კონსტრუქტორის რედაქტირებასა («Edit type» ღილაკზე დაჭერით) ან ტიპის მონიშნულ ეგზემპლართან მუშაობაზე («Edit instance» ღილაკზე დაჭერით).

3.4. კლასთა იერარქიის დათვალიერება

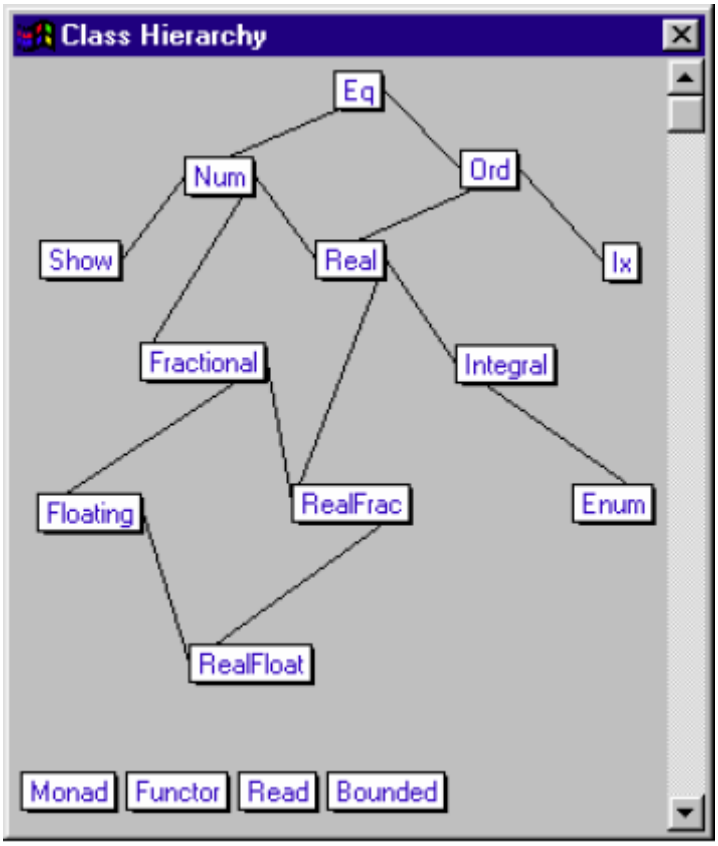
კლასთა იერარქიის დათვალიერებისას პროგრამისტს შეუძლია მემკვიდრეობითობის მიმართებათა დანახვა შექმნილ კლასებს შორის.

უნდა აღინიშნოს, რომ კლასებისა და მიმართებების გამოხატვის ალგორითმი HUGS 98 ინტეგრირებულ გარემოში რამდენადმე არაადეკვატურია, ამიტომ სრული გაგებისათვის პროგრამისტს მოეთხოვება ან ალღო, ან უნარი სწრაფად გააბნიოს ყველა კლასი დიალოგურ ფანჯარაში და შექმნას პლანარული გრაფი.

ამ დიალოგური ფანჯრის გამოძახება შეიძლება არა მარტო მთავარი მენიუდან, არამედ კლასების სიათა დათვალიერების დიალოგური ფანჯრიდანაც.

მე-5 სურათზე ნაჩვენებია კლასების იერარქია `Prelude.hs` ფაილიდან. როგორც სურათიდან ჩანს, ამ ფაილში განსაზღვრულია:

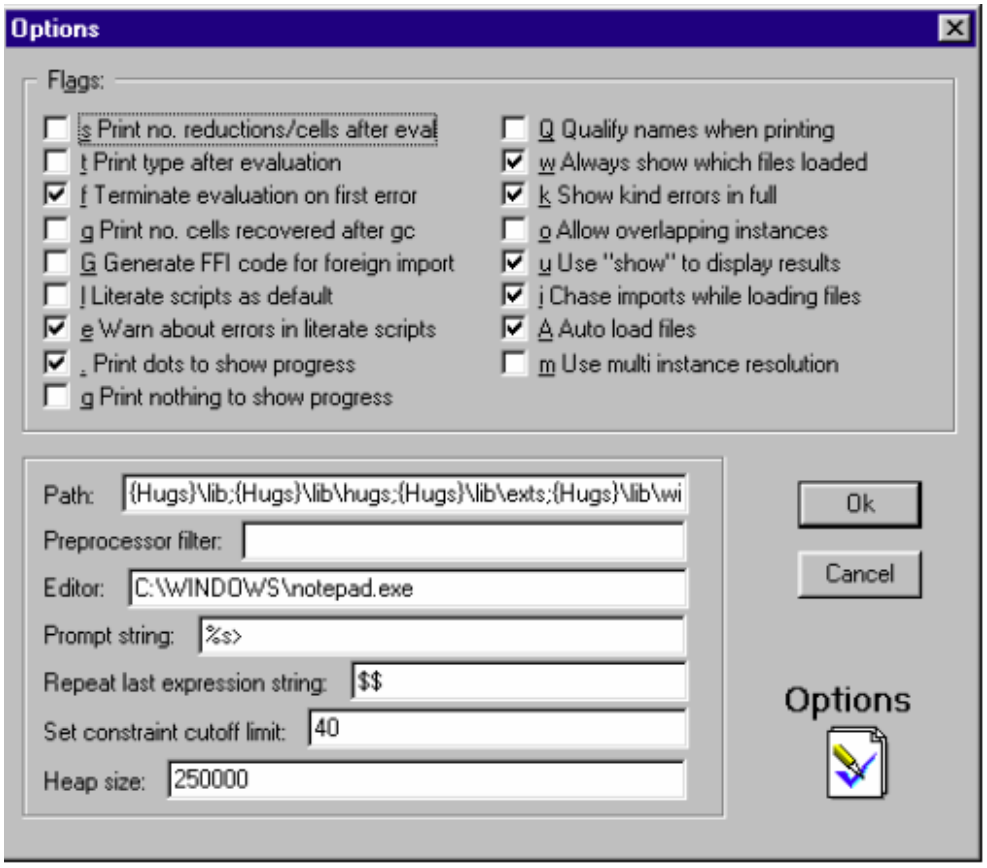
- ეკვივალენტობის კლასები (ყველა, ვინც მემკვიდრეობით იძენს `Eq` კლასის თვისებებს),
- კლასები-მონადები (`Monad`, `Functor`, `Read` და `Bounded`).



სურ. 5. კლასების იერარქია `Prelude.hs` ფაილიდან

4. HUGS-98 ინტეგრირებული გარემოს პარამეტრები

HUGS 98 ინტეგრირებული გარემო საშუალებას აძლევს პროგრამისტს ზუსტად ააწყოს ინტერპრეტატორი და საკუთრივ ინტეგრირებული გარემო ამა თუ იმ ამოცანაზე. ეს შესაძლებელია ინტეგრირებული გარემოს პარამეტრების ცვლილებით. მე-6 სურათზე ნაჩვენებია პარამეტრები, რომლებიც იტვირთება მინიშნების გარეშე (პარამეტრთა ასეთი ნაკრები მოქმედებს HUGS 98 ინტეგრირებული გარემოს საწყისი დაყენებისას).



სურ. 6. ინტეგრირებული გარემოს პარამეტრების აწყობის დიალოგური ფანჯარა

წარმოდგენილი დიალოგური ფანჯრის ზედა ნაწილში განთავსებულია ე.წ. ალმები.

მათი მნიშვნელობებია ან «ჭეშმარიტი» (ალამი აღმართულია), ან «მცდარი» (ალამი დაშვებულია).

ყოველი ალამი პასუხს აგებს ინტერპრეტატორსა ან თვით გარსის ამა თუ იმ პარამეტრზე.

პარამეტრთა აწყობის დიალოგური ფანჯრის ქვედა ნაწილში მოთავსებულია HUGS 98 ინტეგრირებული გარემოს შიგა და გარე ცვლადების შეყვანის ველები.

ყოველი ალამი აღნიშნულია ლათინური ანბანის რომელიმე ასოთი ზედა ან ქვედა რეგისტრში.

ქვემოთ აღწერილია ყველა არსებული ალამი:

s – ბეჭდავს რედუქციებისა და მეხსიერების უჯრედების რაოდენობას გამოთვლების შესრულების შემდეგ;

t – ბეჭდავს გამოსახულების ტიპს მისი გამოთვლის შემდეგ;

f – გამოთვლის წყვეტა პირველი შეცდომის შემდეგ;

g – ბეჭდავს მეხსიერების უჯრედების რაოდენობას, რომლებიც «ნაგვის», «ნარჩენების» (ინგლ. junk) აღებისას დაგროვდა;

G – FFI[1] კოდის გენერაცია იმპორტირებული ფაილებისათვის;

l – სკრიპტების ოპტიმიზაცია გაუცხადებელი შეთანხმებით;

e – გაფრთხილება შეცდომების შესახებ ოპტიმიზებულ სკრიპტებში;

. – წერტილების ბეჭდვა გამოთვლის პროცესის სავიზუალიზაციოდ;

q – უარი რაიმეს ბეჭდვაზე გამოთვლის პროცესის სავიზუალიზაციოდ;

---

[1] FFI (ინგლ. Foreign Function Interface – უცხო ფუნქციათა ინტერფეისი) – ინტერფეისი სხვა ენებზე დაწერილი ფუნქციების გამოსაძახებლად.

Q – სახელების კვალიფიცირება ამობეჭდვისას;

w – ჩატვირთული ფაილების დასახელებათა ყოველთვის ჩვენება;

k – შეცდომათა ტიპისა და აღწერის ჩვენება ყოველთვის;

o – კლასების ეგზემპლართა გადაკვეთის ნების დართვა;

u – «show» ფუნქციის გამოყენება შედეგების წარმოსადგენად;

i – იმპორტირებული ფაილის განადგურება ახლის ჩატვირთვისას;

A – ფაილების ავტომატური ჩატვირთვა;

m – კლასების ეგზემპლართა მრავალფორმიანი რეზოლუციის გამოყენება.

II ნაწილი: ლაბორატორიული სამუშაოები

ლაბორატორიული სამუშაო 1

1. სამუშაოს მიზანი

Haskell ენის ინტერპრეტატორთან მუშაობის ჩვევათა შეძენა. Haskell ენის ძირითად ტიპებზე წარმოდგენის შექმნა. უმარტივეს ფუნქციათა განსაზღვრის შესწავლა.

2. Hugs ინტერპრეტატორთან მუშაობის საფუძვლები

ლაბორატორიული სამუშაოების შესასრულებლად გამოვიყენებთ Haskell ენის ინტერპრეტატორს. არსებობს ინტერპრეტატორის რეალიზაციის რამდენიმე ვერსია; მოცემულ კურსში გამოვიყენებთ Hugs ინტერპრეტატორს (ეს არის შემდეგი სიტყვების აბრევიატურა: «Haskell user's Gofer system», სადაც Gofer – დაპროგრამების ენაა, რომელიც Haskell-ის ერთ-ერთი წინამორბედი იყო).

Hugs ინტერპრეტატორის გაშვების შემდეგ ეკრანზე ჩნდება პროგრამის დამპროექტებლის გარემოს დიალოგური ფანჯარა, ავტომატურად იტვირთება Haskell ენის ტიპების წინასწარ განსაზღვრებათა და სტანდარტული ფუნქციების განსაზღვრებათა სპეციალური ფაილი (Prelude.hs) და სტანდარტულად გამოდის სამუშაოდ მიწვევა. ამ მიწვევას Prelude> სახე აქვს; საზოგადოდ, > სიმბოლოს წინ ბოლოს ჩატვირთული მოდულის სახელი გამოდის.

მიწვევის გამოსვლის შემდეგ შესაძლებელია Haskell ენის გამოსახულებათა ან ინტერპრეტატორის ბრძანებათა შეტანა. ინტერპრეტატორების ბრძანებები განსხვავდება Haskell ენის გამოსახულებებისაგან იმით, რომ იწყება სიმბოლო ორწერტილით (:). ინტერპრეტატორის ბრძანების მაგალითია :quit ჩანაწერი, რომლითაც სრულდება ინტერპრეტატორის მუშაობა. დასაშვებია ინტერპრეტატორის ბრძანებათა შეკვეცა ერთ ასომდე; ამრიგად, :quit და :q ბრძანებები ეკვივალენტურია. :set ბრძანება გამოიყენება იმისათვის, რომ ინტერპრეტატორის სხვადასხვა ოპციის (შესაძლო ვარიანტის) არჩევა შეიძლებოდეს. :? ბრძანებას გამოაქვს ინტერპრეტატორის ხელმისაწვდომ ბრძანებათა ნუსხა. მოგვიანებით სხვა ბრძანებებსაც განვიხილავთ.

3. ტიპები

პროგრამა Haskell ენაზე წარმოადგენს გამოსახულებას, რომლის გამოთვლა *მნიშვნელობას* იძლევა. ყოველ მნიშვნელობას აქვს ტიპი. ინტუიციით ტიპი შეიძლება გავიგოთ როგორც გამოსახულების დასაშვებ მნიშვნელობათა სიმრავლე. გარკვეული გამოსახულების ტიპის გამოსაცნობად შეიძლება ინტერპრეტატორის :type (ან :t) ბრძანების გამოყენება. გარდა ამისა, დასაშვებია :set +t ბრძანების შესრულება იმისათვის, რომ ინტერპრეტატორი ავტომატურად ბეჭდავდეს ყოველი გამოთვლილი შედეგის ტიპს.

Haskell ენის ძირითადი ტიპების მოკლე დახასიათება:

• `Integer` და `Int` ტიპები გამოიყენება მთელი რიცხვის წარმოსადგენად, მაგრამ, ამასთან ერთად `Integer` ტიპის მნიშვნელობები არ იზღუდება სიგრძით.

• `Float` და `Double` ტიპები გამოიყენება ნამდვილი რიცხვის წარმოსადგენად.

• `Bool` ტიპი ორ მნიშვნელობას შეიცავს. ესაა: `True` და `False`. იგი განკუთვნილია ლოგიკურ გამოსახულებათა წარმოსადგენად.

• `Char` ტიპი გამოიყენება სიმბოლოების წარმოსადგენად.

`Haskell` ენაში ტიპის სახელი ყოველთვის მთავრული ასოთი იწყება.

`Haskell` ენა დაპროგრამების ძლიერ ტიპიზებული ენაა. მიუხედავად ამისა, უმეტეს შემთხვევაში, პროგრამისტი ვალდებული არ არის გამოაცხადოს თუ რა ტიპს მიეკუთვნება მის მიერ შემოტანილი ცვლადი. ინტერპრეტატორს თვითონ შეუძლია გამოიტანოს მომხმარებლის მიერ გამოყენებულ ცვლადთა ტიპები. მაგრამ, თუ რაიმე მიზნისათვის მაინც აუცილებელია გარკვეული მნიშვნელობისათვის ტიპის გამოცხადება, მაშინ გამოიყენება შემდეგი სახის კონსტრუქცია: *ცვლადი* `::` *ტიპი.*

თუ ჩართულია ინტერპრეტატორის `+t` ოპცია, ეს ინტერპრეტატორი ბეჭდავს მნიშვნელობებს ამავე ფორმატით.

ქვემოთ მაგალითის სახით მოყვანილია ინტერპრეტატორთან ჩატარებული სამუშაო სეანსის ოქმი. იგულისხმება, რომ `Prelude>` მიწვევის შემდეგ

მოთავსებული ტექსტი შეაქვს მომხმარებელს, ხოლო მომდევნო ტექსტი ასახავს სისტემის პასუხს.

```
Prelude>:set +t

Prelude>1

1 :: Integer

Prelude>1.2

1.2 :: Double

Prelude>'a'

'a' :: Char

Prelude>True

True :: Bool
```

ამ ოქმიდან შეიძლება დავასკვნათ, რომ Integer, Double და Char ტიპის მნიშვნელობები მოიცემა დაპროგრამების C ენაში მიღებული წესების მსგავსად.

ტიპების განვითარებული სისტემა და მკაცრი ტიპიზაცია უზრუნველყოფს პროგრამათა უსაფრთხოებას Haskell ენაში ტიპების მიხედვით. გარანტირებულია, რომ Haskell ენაზე დაწერილ სწორ პროგრამაში ყველა ტიპის გამოყენებაც სწორი იქნება. პრაქტიკული თვალსაზრისით ეს ნიშნავს, რომ Haskell ენაზე დაწერილ პროგრამას შესრულებისას არ შეუძლია მეხსიერებაში შეღწევის (Access violation) შეცდომის გა-

მოწვევა. ასევე გარანტირებულია, რომ პროგრამაში არ შეიძლება მოხდეს არაინიციალიზებული ცვლადების გამოყენება. ერთი სიტყვით, მრავალი შეცდომა პროგრამაში მჟღავნდება უკვე მისი კომპილაციის და არა შესრულების ეტაპზე.

4. არითმეტიკა

Hugs ინტერპრეტატორი შეიძლება გამოვიყენოთ არითმეტიკული გამოსახულების გამოსათვლელად. ამ დროს შესაძლებელია +, -, *, / (შეკრების, გამოლების, გამრავლებისა და გაყოფის) ოპერატორების გამოყენება პრიორიტეტების ჩვეულებრივი წესებით. გარდა ამისა, შეიძლება ^ (ხარისხში აყვანის) ოპერატორის გამოყენებაც. ამრიგად, მუშაობის სეანსი შეიძლება ასეთი იყოს:

```
Prelude>2*2

4 :: Integer

Prelude>4*5 + 1

21 :: Integer

Prelude>2^3

8 :: Integer
```

გარდა ამისა, შეიძლება სტანდარტული მათემატიკური ფუნქციების გამოყენებაც, როგორიცაა: sqrt (კვადრატული ფესვი), sin, cos, exp და მისთანანი. სხვა ენებისგან განსხვავებით Haskell-ში ფუნქციის გამოძახებისას სავალდებულო არ არის არგუმენტების ფრჩხილებში მოთავსება.

მაშასადამე, დასაშვებია `sqrt 2` ჩაწერა ტრადიციული `sqrt(2)` ფორმის გამოყენების ნაცვლად. მაგალითი:

```
Prelude>sqrt 2

1.4142135623731 :: Double

Prelude>1 + sqrt 2

2.4142135623731 :: Double

Prelude>sqrt 2 + 1

2.4142135623731 :: Double

Prelude>sqrt (2 + 1)

1.73205080756888 :: Double
```

მოცემული მაგალითიდან იმ დასკვნის გამოტანა შეიძლება, რომ ფუნქციის გამოძახებას უფრო დიდი პრიორიტეტი აქვს, ვიდრე არითმეტიკულ ოპერაციას, ასე რომ `sqrt 2 + 1` გამოსახულება ინტერპრეტირებულია `(sqrt 2) + 1` სახით და არა როგორც `sqrt (2 + 1)`. გამოთვლის ზუსტი თანამიმდევრობის მისათითებლად საჭიროა ფრჩხილების გამოყენება, უკანასკნელი მაგალითის მსგავსად. სინამდვილეში ფუნქციის გამოძახებას უფრო დიდი პრიორიტეტი აქვს, ვიდრე ნებისმიერ ბინარულ ოპერატორს.

ასევე უნდა აღინიშნოს, რომ დაპროგრამების მრავალი სხვა ენისგან განსხვავებით, მთელრიცხვა გამოსახულებები `Haskell` ენაში გამოითვლება თანრიგების შეუზღუდავი რაოდენობით. გამოთვალეთ, მაგალითად,

2^5000 გამოსახულება. C ენისგან განსხვავებით, სადაც int ტიპის მაქსიმალურად შესაძლებელი მნიშვნელობა შემოფარგლულია მანქანების თანრიგებით (თანამედროვე პერსონალურ კომპიუტერებში იგი, ჩვეულებრივ, 2^31-1 = 2147483647 სიდიდეა), Haskell ენაში Integer ტიპს შეუძლია ნებისმიერი სიგრძის მთელი რიცხვების შენახვა.

5. კორტეჟი

ზემოთ ჩამოთვლილი მარტივი ტიპების გარდა, Haskell ენაში შეიძლება შედგენილი ტიპების მნიშვნელობათა განსაზღვრაც.

მაგალითად, სიბრტყეზე წერტილი მოიცემა მისი კოორდინატების შესაბამისი ორი რიცხვით. Haskell ენაში წყვილის მოცემა შესაძლებელია მძიმით განცალკევებული და ფრჩხილებში მოთავსებული ორი კომპონენტის ჩამოთვლით: (5,3).

აუცილებელი არ არის, რომ წყვილის კომპონენტები ერთსა და იმავე ტიპს მიეკუთვნებოდეს: დასაშვებია წყვილის შედგენა, სადაც პირველი ელემენტი იქნება სტრიქონი, ხოლო მეორე ელემენტი – რიცხვი და ა.შ.

საზოგადოდ, თუ a და b Haskell ენის რაღაც ნებისმიერი ტიპებია, მაშინ იმ წყვილის ტიპი, რომელშიც პირველი ელემენტი a ტიპს მიეკუთვნება, ხოლო მეორე – b ტიპს, (a,b) ჩანაწერით აღინიშნება. მაგალითად, (5,3) წყვილს (Integer, Integer) ტიპი აქვს; (1, 'a') წყვილი მიეკუთვნება (Integer, Char) ტიპს. შეიძლება უფრო რთული მაგალითის განხილვაც: ((1,'a'),1.2) წყვილი ((In-

teger,Char),Double) ტიპს მიეკუთვნება. შეამოწმეთ ეს ინტერპრეტატორის საშუალებით.

ყურადღება უნდა მიექცეს იმ გარემოებას, რომ, მართალია, (1,2) და (Integer,Integer) სახის კონსტრუქციები ვიზუალურად ერთმანეთის მსგავსია, Haskell ენაში მათი არსი სრულიად განსხვავებულია. პირველი ჩანაწერი არის მნიშვნელობა, მეორე გამოსახულება კი – ტიპი.

წყვილებთან სამუშაოდ Haskell ენაში არსებობს სტანდარტული fst და snd ფუნქციები, რომლებიც აბრუნებს, შესაბამისად, წყვილის პირველ და მეორე ელემენტებს (ამ ფუნქციათა სახელწოდებები ნაწარმოებია ინგლისური სიტყვებისგან: «first» – პირველი და «second» - მეორე). ამრიგად, შესაძლებელია მათი გამოყენება, მაგალითად, ასეთი ფორმით:

```
Prelude>fst (5, True)

5 :: Integer

Prelude>snd (5, True)

True :: Bool
```

წყვილების გარდა, შესაძლებელია სამეულების, ოთხეულების და ა.შ. განსაზღვრაც. მათი ტიპები ანალოგიურად ჩაიწერება:

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

(1,2,3) :: (Integer,Integer,Integer)
```

```
Prelude>(1,2,3,4)

(1,2,3,4) :: (Integer,Integer,Integer,Integer)
```

მონაცემთა ასეთ სტრუქტურას კორტეჟი (ამალა) ეწოდება. კორტეჟში შეიძლება ინახებოდეს სხვადასხვა ჯურის მონაცემები. `fst` და `snd` ფუნქციები განსაზღვრულია მხოლოდ წყვილებისათვის და არ მუშაობს სხვა სახის კორტეჟთან. მათი გამოყენების მცდელობისას, ვთქვათ, სამეულებისათვის, ინტერპრეტატორს გამოაქვს შეტყობინება შეცდომის შესახებ.

კორტეჟის ელემენტად შეიძლება ნებისმიერი ტიპის მნიშვნელობის გამოყენება სხვა კორტეჟის ჩათვლით. წყვილებით შედგენილი კორტეჟის ელემენტებში შესაღწევად შეიძლება `fst` და `snd` ფუნქციათა კომბინაციის გამოყენება. შემდეგი მაგალითი გვიჩვენებს `'a'` ელემენტის ამოღებას `(1, ('a', 23.12))` კორტეჟიდან:

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

'a' :: Char
```

6. სია

კორტეჟისგან განსხვავებით, სიას შეუძლია ნებისმიერი რაოდენობის ელემენტის შენახვა. სიის მოსაცემად `Haskell`-ში აუცილებელია კვადრატულ ფრჩხილებში მძიმით განცალკევებული ყველა მისი ელემენტის ჩამოთვლა. ყველა ეს ელემენტი ერთსა და იმავე ტიპს უნდა მიეკუთვნებოდეს. იმ სიის ტიპი, რომლის ელემენტები `a` ტიპს მიეკუთვნება, `[a]` ფორმით აღინიშნება.

```
Prelude>[1,2]

[1,2] :: [Integer]

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

['1','2','3'] :: [Char]
```

სია შეიძლება არ შეიცავდეს ელემენტებს და ცარიელი იყოს. ასეთ შემთხვევაში იგი [] ფორმით ჩაიწერება.

ოპერატორი : (ორწერტილი) გამოიყენება ელემენტის დასამატებლად სიის დასაწყისში. მისი მარცხენა არგუმენტი ელემენტი უნდა იყოს, ხოლო მარჯვენა – სია (თუნდაც ცარიელი):

```
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]
```

(:) ოპერატორისა და ცარიელი სიის საშუალებით შეიძლება ნებისმიერი სიის აგება:

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

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

(:) ოპერატორი ასოციაციურია მარჯვნივ, ამიტომ ზემოთ მოცემულ გამოსახულებაში ფრჩხილები შეიძლება არც ვიხმაროთ:

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

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

სიის ელემენტებად შეიძლება გამოვიყენოთ — რიცხვები, სიმბოლოები, კორტეჟები, სხვა სიები და ასე შემდეგ:

```
Prelude>[(1,'a'),(2,'b')]

[(1,'a'),(2,'b')] :: [(Integer,Char)]

Prelude>[[1,2],[3,4,5]]

[[1,2],[3,4,5]] :: [[Integer]]
```

სიებთან სამუშაოდ, Haskell ენაში, ფუნქციათა დიდი რაოდენობა არსებობს. მოცემულ ლაბორატორიულ სამუშაოში მხოლოდ ზოგიერთ მათგანზე შევაჩერებთ ყურადღებას:

- head ფუნქცია, რომელიც სიის პირველ ელემენტს გვიბრუნებს.
- tail ფუნქცია, რომელიც გვიბრუნებს სიას პირველი ელემენტის გარეშე.
- length ფუნქცია, რომელიც გვიბრუნებს სიის სიგრძეს.

head და tail ფუნქციები განსაზღვრულია არაცარიელი სიებისათვის. ცარიელი სიისადმი მათი გამოყენების მცდელობისას ინტერპრეტატორს

გამოაქვს შეტყობინება შეცდომის შესახებ. მოცემულ 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
```

ზაზგასმით უნდა აღინიშნოს ის გარემოება, რომ `length` ფუნქცია მიეკუთვნება `Int` და არა `Integer` ტიპს.

სიათა შესაერთებლად (კონკატენაციისათვის) `Haskell`-ში განსაზღვრულია `++` ოპერატორი:

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

[1,2,3,4] :: Integer
```

7. სტრიქონი

Haskell ენაში, ისევე, როგორც C-ში, სტრიქონული მნიშვნელობები ორმაგ ("...")ბრჭყალებში მოიცემა. ისინი String ტიპს მიეკუთვნება:

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

სინამდვილეში სტრიქონები სიმბოლოთა სიებია. მაშასადამე:

"hello",

['h','e','l','l','o'] და 'h':'e':'l':'l':'o':[]

გამოსახულებები ერთსა და იმავეს ნიშნავს, ხოლო String ტიპი [Char]-ის სინონიმია. სიებთან სამუშაოდ განკუთვნილი ყველა ფუნქცია შეიძლება სტრიქონებთან სამუშაოდაც გამოვიყენოთ:

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

```
"hello, world" :: [Char]
```

რიცხვითი მნიშვნელობების სტრიქონების სახით წარმოსადგენად და პირიქით, არსებობს `show` და `read` ფუნქციები:

```
Prelude>show 1

"1" :: [Char]

Prelude>"Formula " ++ show 1

"Formula 1" :: [Char]

Prelude>1 + read "12"

13 :: Integer
```

თუ გარდაქმნისას `read` ფუნქცია ვერ აქცევს სტრიქონს რიცხვად, მაშინ იგი გამოიტანს შეტყობინებას შეცდომის შესახებ.

8. ფუნქცია

აქამდე ჩვენ ვიყენებდით `Haskell` ენის ჩაშენებულ ფუნქციებს. ახლა დადგა დრო ვისწავლოთ საკუთარი ფუნქციების განსაზღვრა. ამისათვის აუცილებელია ინტერპრეტატორის კიდევ რამდენიმე ბრძანების გაცნობა (შეგახსენებთ, რომ ეს ბრძანებები შეიძლება დავიდეს ერთ ასომდე):

- `:load` ბრძანება საშუალებას იძლევა ჩავტვირთოთ ინტერპრეტატორში `Haskell` ენაზე დაწერილი პროგრამა, რომელიც არის მითითებულ ფაილში.

• :edit ბრძანება იწყებს ბოლოს ჩატვირთული ფაილის რედაქტირების პროცესს.

• :reload ბრძანება ახდენს ბოლოს ჩატვირთული ფაილის ხელახლა წაკითხვას.

მომხმარებლის ფუნქციათა განსაზღვრებანი უნდა იმყოფებოდეს ფაილში, რომელიც იტვირთება Hugs ინტერპრეტატორში :load ბრძანების საშუალებით. ჩატვირთული პროგრამის რედაქტირებისათვის შეიძლება :edit ბრძანების გამოყენება, რომლითაც ფაილის რედაქტირებისათვის ამოქმედდება გარეშე რედაქტორი (გაუცხადებელი შეთანხმებით – Notepad პროგრამა). რედაქტირების სეანსის დასრულების შემდეგ აუცილებელია რედაქტორის დახურვა; ამასთან Hugs ინტერპრეტატორი ხელახლა წაიკითხავს შეცვლილი ფაილის შიგთავსს. მაგრამ ფაილის რედაქტირება უშუალოდ Windows-ის გარსიდანაც შეიძლება. ამ შემთხვევაში, იმისათვის რომ ინტერპრეტატორმა შეძლოს ფაილის კვლავ წაკითხვა, აუცილებელია :reload ბრძანების ცხადად გამოძახება.

განვიხილოთ მაგალითი. შექმენით რომელიმე კატალოგში lab1.hs ფაილი. დავუშვათ, რომ ამ ფაილისაკენ მიმავალი გზა მოცემულია c:\labs\lab1.hs ჩანაწერით (უნდა ხვდებოდეთ, რომ ეს მხოლოდ

მაგალითია და თქვენ ფაილებს შეიძლება სხვა სახელები ჰქონდეს). Hugs ინტერპრეტატორში შეასრულეთ შემდეგი ბრძანებები:

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

თუ ჩატვირთვა წარმატებით განხორციელდა, ინტერპრეტატორის მიწვევა `Main>` სახეს იძენს. საქმე ისაა, რომ, თუ მოდულის სახელი მითითებული არ არის, იგი `Main`-თან იგივდება.

```
Main>:edit
```

ამ დროს გაიხსნება რედაქტორის ფანჯარა, რომელშიც შესაძლებელია პროგრამის ტექსტის შეყვანა. მაგალითად:

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

შეინახეთ ფაილი და დახურეთ რედაქტორი. ახლა, თუ `Hugs` ინტერპრეტატორი ჩატვირთავს c:\labs\lab1.hs ფაილს, `x` ცვლადი განსაზღვრული აღმოჩნდება:

```
Main>x

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

ყურადღება მიაქციეთ იმ გარემოებას, რომ ფაილის სახელის ჩაწერისას `:load` ბრძანების არგუმენტში \ სიმბოლოები დუბლირებულია. `C` ენის მსგავსად, `Haskell`-შიც \ სიმბოლო დამხმარე (მაგალითად, `'\n'`) სიმბოლოს დასაწყისის ინდიკატორია. ამიტომ უშუალოდ \ სიმბოლოს შესაყვანად აუცილებელია (როგორც ეს ზოგიერთ სხვა ენაშიცაა მიღებული) მისი ეკრანირება კიდევ ერთი \ სიმბოლოთი.

ახლა უკვე შეიძლება შევუდგეთ ფუნქციათა განსაზღვრას. ამისათვის ზემოთ აღწერილი პროცესის შესაბამისად შექმენით რაღაც ფაილი და ჩაწერეთ მასში შემდეგი ტექსტი:

```
square :: Integer -> Integer

square x = x * x
```

პირველი (square :: Integer -> Integer) სტრიქონი აცხადებს, რომ ჩვენ განვსაზღვრავთ square ფუნქციას. იგი იღებს Integer ტიპის პარამეტრს და გვიბრუნებს კვლავ Integer ტიპის შედეგს. მეორე (square x = x * x) სტრიქონი ფუნქციის უშუალო განსაზღვრას იძლევა. სახელდობრ square ფუნქცია იღებს ერთ არგუმენტს და გვიბრუნებს მის კვადრატს.

Haskell ენაში ფუნქციები «პირველი კლასის» მნიშვნელობებია. ეს ნიშნავს, რომ ფუნქციები, მთელი და ნამდვილი რიცხვები, სიმბოლოები, სტრიქონები, სიები და ა.შ. ენის «თანასწორუფლებიანი» შემდგენებია. ფუნქცია შეიძლება არგუმენტად გადაიცეს სხვა ფუნქციაში, დაბრუნდეს ფუნქციიდან და სხვ. როგორც ყველა მნიშვნელობას Haskell ენაში, ფუნქციასაც აქვს ტიპი. იმ ფუნქციის ტიპი, რომელიც იღებს a ტიპის მნიშვნელობებს და გვიბრუნებს b ტიპის მნიშვნელობებს, აღინიშნება a->b ჩანაწერით.

ჩატვირთეთ შექმნილი ფაილი ინტერპრეტატორში და შეასრულეთ შემდეგი ბრძანებები:

```
Main>:type square

square :: Integer -> Integer

Main>square 2
```

```
4 :: Integer
```

როგორც ვხედავთ, `square` ფუნქციის ტიპის გამოცხადება აუცილებელი არ აღმოჩნდა: ინტერპრეტატორს თავად შეეძლო დაედგინა საჭირო ინფორმაცია ფუნქციის ტიპის შესახებ ამ ფუნქციის განსაზღვრებიდან. მაგრამ, ჯერ ერთი, დადგენილი ტიპი უფრო ზოგადი იქნებოდა, ვიდრე `Integer -> Integer` ჩანაწერია, მეორე მხრივ, ფუნქციის ტიპის ცხადი მითითება «კარგ ტონად» ითვლება `Haskell` ენაზე დაპროგრამებისას, ვინაიდან ტიპის გამოცხადება ფუნქციის თავისებური «დოკუმენტაციის» როლს ასრულებს და ხელს უწყობს დაპროგრამების შეცდომათა გამომჟღავნებას.

მომხმარებლის მიერ განსაზღვრული ფუნქციებისა და ცვლადების სახელები უნდა იწყებოდეს ლათინური ასოთი ქვედა რეგისტრში. ნებისმიერი სხვა პოზიცია სახელში შეიძლება წარმოდგენილი იყოს მთავრული ან ნუსხური ლათინური ასოთი, ციფრით ან `( _ )`ქვედა ხაზგასმისა და `( ' )` აპოსტროფის სიმბოლოთი. ქვემოთ ჩამოთვლილია ცვლადთა სწორი სახელების მაგალითები:

```
var

var1

variableName

variable_name

var'
```

9. პირობითი გამოსახულება

Haskell ენაში ფუნქციის განსაზღვრება შეიძლება იყენებდეს პირობით გამოსახულებებს. მაგალითის სახით ჩავწეროთ signum ფუქნცია, რომელსაც მისთვის გადაცემული არგუმენტის ნიშნის გარკვევა ესაჭიროება სათანადო შედეგის დასაბრუნებლად:

```
signum :: Integer -> Integer

signum x = if x > 0 then 1

           else if x < 0 then -1

                else 0
```

პირობითი გამოსახულება შემდეგი სახით ჩაიწერება:

if პირობა then გამოსახულება else გამოსახულება.

მიაქციეთ ყურადღება, რომ, თუმცა ეს ჩანაწერი გვაგონებს შესაბამის ოპერატორს C ენაში, Haskell ენის პირობით გამოსახულებაში წარმოდგენილი უნდა იყოს როგორც then-ნაწილი, ასევე else-ნაწილიც. გამოსახულებები პირობითი ოპერატორის then-ნაწილსა და else-ნაწილში ერთსა და იმავე ტიპს უნდა მიეკუთვნებოდეს.

პირობითი ოპერატორის განსაზღვრებაში პირობა არის Bool ტიპის ნებისმიერი გამოსახულება. ასეთი გამოსახულებების მაგალითად შეიძლება დავასახელოთ შედარებები. შედარებისას შეიძლება შემდეგი ოპერატორების გამოყენება:

• `<`, `>`, `<=`, `>=` – ამ ოპერატორებს იგივე აზრი აქვს, რაც `C` ენაში, სახელდობრ, *ნაკლებია*, *მეტია*, *ნაკლებია ან ტოლი*, *მეტია ან ტოლი*.

• `==` – ტოლობის შემოწმების ოპერატორი.

• `/=` – უტოლობის შემოწმების ოპერატორი.

შესაძლებელია `Bool` ტიპის გამოსახულებათა კომბინირება საყოველთაოდ მიღებული ლოგიკური `&&` და `||` (*და* და *ან*) ოპერატორებისა და უარყოფის `not` ფუნქციის საშუალებით. დასაშვებ პირობათა მაგალითებია:

```
x >= 0 && x <= 10
```

```
x > 3 && x /= 10
```

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

რა თქმა უნდა, შეიძლება საკუთარი ფუნქციების განსაზღვრაც, რომლებიც გვიბრუნებს `Bool` ტიპის მნიშვნელობებს, და მათი გამოყენებაც პირობების როლში. მაგალითად, შეიძლება განვსაზღვროთ `isPositive` ფუნქცია, რომელიც გვიბრუნებს `True`-ს, თუ მისი არგუმენტი არაუარყოფითია, და `False`-ს – წინააღმდეგ შემთხვევაში:

```
isPositive :: Integer -> Bool
```

```
isPositive x = if x > 0 then True else False
```

ახლა `signum` ფუნქცია შემდეგი სახით შეიძლება განისაზღვროს:

```
signum :: Integer -> Integer
```

```
signum x = if isPositive x then 1

else if x < 0 then -1

else 0
```

აღსანიშნავია, რომ `isPositive` ფუნქცია უფრო მარტივადაც შეიძლება განვსაზღვროთ:

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

10. მრავალი ცვლადის ფუნქცია და ფუნქციის განსაზღვრის თანამიმდევრობა

აქამდე ჩვენ ვსაზღვრავდით ფუნქციას, რომელიც ერთ არგუმენტს იღებდა. რა თქმა უნდა, `Haskell` ენაში შესაძლებელია ისეთი ფუნქციის განსაზღვრაც, რომელიც არგუმენტების ნებისმიერ რაოდენობას იღებს. მაგალითად, `add` ფუნქციის განსაზღვრება, რომელიც ორ მთელ რიცხვს იღებს და მათ ჯამს გვიბრუნებს, ასე გამოიყურება:

```
add :: Integer -> Integer -> Integer

add x y = x + y
```

`add` ფუნქციის ტიპი აქ შეიძლება რამდენადმე უცნაურადაც კი ჩანდეს. `Haskell` ენაში ითვლება, რომ `->` ოპერაცია[2] ასოციაციურია მარჯვნივ.

[2] `a -> b` კონსტრუქცია შეესაბამება იმ ფუნქციათა ტიპს, რომლებიც იღებს შესასვლელზე `a` ტიპის ელემენტს და გვიბრუნებს `b` ტიპის ელემენტს. მაშასადამე, `Integer -> Integer -> Integer` არის ფუნქცია, რომელიც იღებს შესასვლელზე ორ მთელ რიცხვს და გვიბრუნებს მთელ რიცხვს. `Float -> Float -> Float` არის ფუნქცია, რომელიც იღებს შესასვლელზე ორ ნამდვილ რიცხვს და გვიბრუნებს ნამდვილ

ამრიგად, `add` ფუნქციის ტიპი შეიძლება იყოს წაკითხული როგორც `Integer -> (Integer -> Integer)`, ესე იგი *კარირების* წესის შესაბამისად, `add` ფუნქციის გამოყენების შედეგი ერთი არგუმენტისადმი იქნება ფუნქცია, რომელიც იღებს `Integer` ტიპის ერთ პარამეტრს. საზოგადოდ, იმ ფუნქციის ტიპი, რომელიც იღებს `t1, t2, . . . , tn` ტიპის `n` არგუმენტს და გვიბრუნებს `a` ტიპის შედეგს, ჩაიწერება `t1->t2->...->tn->a` სახით.

საჭიროა კიდევ ერთი შენიშვნის გაკეთება, რომელიც ფუნქციათა განსაზღვრის თანამიმდევრობას ეხება. წინა პუნქტში ჩვენ ორი — `signum` და `isPositive` — ფუნქცია განვსაზღვრეთ და მათ შორის ერთ-ერთი საკუთარი თავის განსაზღვრისათვის იყენებდა მეორეს. ჩნდება კითხვა: ამ ორი ფუნქციიდან რომელი უნდა განისაზღვროს უფრო ადრე? ძალაუნებურად ჩნდება პასუხი, რომ `isPositive` ფუნქციის განსაზღვრა უნდა უსწრებდეს `signum` ფუნქციის განსაზღვრას; მაგრამ სინამდვილეში Haskell ენისათვის ფუნქციათა განსაზღვრის თანამიმდევრობას მნიშვნელობა არა აქვს! ამრიგად, `isPositive` ფუნქცია შეიძლება განისაზღვროს როგორც `signum` ფუნქციამდე, ასევე მის შემდეგაც.

რიცხვს. დაბოლოს, `(Float, Integer) -> [(Float, Float)]` არის ფუნქცია, რომელიც იღებს შესასვლელზე `Float` და `Integer` ტიპის რიცხვთა წყვილს და გვიბრუნებს `Float` ტიპის რიცხვთა წყვილების სიას.

11. დავალება ლაბორატორიული სამუშაოსათვის

1. დაასახელეთ შემდეგი ტიპის არატრივიალურ გამოსახულებათა მაგალითი:

1) `((Char,Integer), String, [Double])`

2) `[(Double,Bool,(String,Integer))]`

3) `([Integer],[Double],[(Bool,Char)])`

4) `[[[(Integer,Bool)]]]`

5) `(((Char,Char),Char),[String])`

6) `(([Double],[Bool]),[Integer])`

7) `[Integer, (Integer,[Bool])]`

8) `(Bool,([Bool],[Integer]))`

9) `[([Bool],[Double])]`

10) `[([Integer],[Char])]`

არატრივიალურობა მოცემულ შემთხვევაში ნიშნავს, რომ გამოსახულებებში გამოყენებული სიები უნდა შეიცავდეს ერთზე მეტ ელემენტს.

2. განსაზღვრეთ შემდეგი ფუნქციები:

1) `max3` ფუნქცია, რომელიც სამი მთელი რიცხვის მიღებისას გვიბრუნებს მათ შორის უდიდესს.

2) `min3` ფუნქცია, რომელიც სამი მთელი რიცხვის მიღებისას გვიბრუნებს მათ შორის უმცირესს.

3) `sort2` ფუნქცია, რომელიც ორი მთელი რიცხვის მიღებისას გვიბრუნებს წყვილს, სადაც ორ შეტანილ რიცხვს შორის უმცირესი დგას პირველ ადგილზე, ხოლო უდიდესი – მეორეზე.

4) `bothTrue :: Bool -> Bool -> Bool` ფუნქცია, რომელიც გვიბრუნებს `True`-ს მაშინ და მხოლოდ მაშინ, როდესაც მისი ორივე არგუმენტი `True`-ს ტოლი იქნება. ფუნქციის განსაზღვრისას არ გამოიყენოთ სტანდარტული ლოგიკური ოპერაციები, როგორიცაა `&&, ||` და მისთანანი.

5) `solve2::Double->Double->(Bool,Double)` ფუნქცია, რომელიც ორი რიცხვის (სახელდობრ, `ax + b = 0` წრფივი განტოლების კოეფიციენტების) საფუძველზე გვიბრუნებს წყვილს, სადაც პირველი ელემენტი `True`-ს ტოლია, თუ ამონახსნი არსებობს, და `False`-ს უდრის წინააღმდეგ შემთხვევაში; ამასთან ერთად, პასუხის მეორე ელემენტი ან ფესვის მნიშვნელობით არის წარმოდგენილი, ან `0.0` სიდიდით, წრფივი განტოლების ამონახსნის არსებობის თუ არარსებობის შესაბამისად.

6) `isParallel` ფუნქცია, რომელიც გვიბრუნებს `True`-ს, თუ ორი მონაკვეთი, რომელთა ბოლოები ფუნქციის არგუმენტებითაა მოცემული, პარალელურია (ან ერთ წრფეზეა). მაგალითად, `isParallel (1,1) (2,2) (2,0) (4,2)` გამოსახულების

მნიშვნელობა უნდა იყოს `True`-ს ტოლი, ვინაიდან `(1, 1) - (2, 2)` და `(2, 0) - (4, 2)` მონაკვეთები პარალელურია.

7) `isIncluded` ფუნქცია, რომლის არგუმენტები არის ერთ სიბრტყეზე განთავსებული ორი წრეწირის პარამეტრები (ცენტრების კოორდინატები და რადიუსები); ფუნქცია გვიბრუნებს `True`-ს, თუ მეორე წრეწირი მთლიანად განლაგებულია პირველის შიგნით.

8) `isRectangular` ფუნქცია, რომელიც პარამეტრებად იღებს სიბრტყეზე განთავსებული სამი წერტილის კოორდინატებს და გვიბრუნებს `True`-ს, თუ მათ მიერ შექმნილი სამკუთხედი მართკუთხაა.

9) `isTriangle` ფუნქცია, რომელიც არკვევს მოცემული `x`, `y` და `z` სიგრძის მონაკვეთებით სამკუთხედის აგების შესაძლებლობას.

10) `isSorted` ფუნქცია, რომელიც იღებს შესასვლელზე სამ რიცხვს და გვიბრუნებს `True`-ს, თუ ისინი მოწესრიგებულია ზრდადობის ან კლებადობის მიხედვით.

12. ლაბორატორიული სამუშაოს შესრულების წესი

ლაბორატორიული სამუშაო შედგება ორი დავალებისგან. მათი ნომრები შეესაბამება სტუდენტის ვარიანტის ნომერს. ვარიანტის ნომერი უდრის სტუდენტის რიგით ნომერს სიაში (შესაძლებელია ათის მოდულით, თუ სტუდენტთა რაოდენობა ლაბორატორიაში ათზე მეტია). ლაბორატორიული სამუშაოს ჩასაბარებლად ყოველი სტუდენტისათვის აუცილებელია:

1. ვარიანტის შესაბამისი დავალების მიღება;

2. მისი დამოუკიდებლად შესრულება;

3. კომპიუტერზე მასწავლებლის თანდასწრებით პროგრამის მუშაობის დემონსტრირება;

4. ანგარიშის გაფორმება და ჩაბარება;

5. ლაბორატორიული სამუშაოს დაცვა.

ანგარიშში *აუცილებლად* უნდა იყოს:

1. ვარიანტის ნომერი;

2. დავალებათა ტექსტი;

3. აუცილებლობის შემთხვევაში – ახსნა-განმარტება რეალიზაციასთან დაკავშირებით;

4. პროგრამის ტექსტი;

5. ტესტების შედეგები.

ლაბორატორიული სამუშაოს დაცვა გულისხმობს: ა) მასწავლებელთან გასაუბრებას (წარმოდგენილი ანგარიშის მასალის ირგვლივ) და ბ) სტუდენტის პასუხებს საკონტროლო შეკითხვებზე.

13. საკონტროლო შეკითხვები

1. რით განსხვავდება ინტერპრეტატორის ბრძანებები `Haskell` ენის გამოსახულებებისგან?

2. `Haskell` ენის ძირითადი ტიპები.

3. ფუნქციები კორტეჟებთან სამუშაოდ.

4. ფუნქციები სიებთან სამუშაოდ.

5. ცვლადებისა და ფუნქციების დასაშვები სახელები.

6. ინტერპრეტატორის ბრძანებები პროგრამების ფაილებთან სამუშაოდ.

7. პირობითი გამოსახულებები `Haskell` ენაში.

8. ფუნქციათა განსაზღვრა `Haskell` ენაში.

ლაბორატორიული სამუშაო 2

1. სამუშაოს მიზანი

რეკურსიულ ფუნქციათა განსაზღვრის ათვისება. ნიმუშთან შედარების მექანიზმზე წარმოდგენის შექმნა. სიების დასამუშავებლად განკუთვნილ ფუნქციათა განსაზღვრის ჩვევათა შეძენა.

2. კომენტარები

პროგრამაში კომენტარების არსებობის აუცილებლობა ცხადია. სამწუხაროდ, დაპროგრამების სხვადასხვა ენის ავტორთა შეხედულებები, კოდში კომენტარების აღნიშვნის შესახებ, დაშორდა ერთმანეთს. დაპროგრამების `Haskell` ენაც არ გამხდარა გამონაკლისი ამ თვალსაზრისით.

`Haskell` ენაში, ისევე, როგორც `C++` ენაში, ორი სახის კომენტარია განსაზღვრული: სტრიქონული და ბლოკური. სტრიქონული კომენტარი იწყება `--` სიმბოლოებით და გრძელდება სტრიქონის ბოლომდე (`C++` ენაში ამის ანალოგია `//` სომბოლოებით დაწყებული კომენტარი). ბლოკური კომენტარი იწყება `{-` სიმბოლოებით და გრძელდება `-}` სიმბოლოებამდე (`C++` ენაში მისი ანალოგია `/*` და `*/` სიმბოლოებით შეზღუდული კომენტარები). რასაკვირველია, ყველაფერი, რაც კომენტარს წარმოადგენს, იგნორირებული იქნება `Haskell` ენის ინტერპრეტატორით ან კომპილატორით. მაგალითად:

```
f x = x -- ეს კომენტარია.

g x y =
```

```
{- ეს ასევე კომენტარია,

მაგრამ უფრო გრძელი. -}

x + y
```

3. რეკურსია

დაპროგრამების იმპერატიულ ენებში ძირითადი კონსტრუქცია არის ციკლი. Haskell ენაში ციკლების ნაცვლად რეკურსია გამოიყენება. ფუნქციას რეკურსიული ეწოდება, თუ იგი საკუთარი თავის გამოძახებას ახდენს (ანუ, უფრო ზუსტად რომ ვთქვათ, განსაზღვრულია საკუთარი თავის ტერმინებით). რეკურსიული ფუნქციები არსებობს იმპერატიულ ენებშიც, მაგრამ აქ ეს ასე ფართოდ არ გამოიყენება. ერთ-ერთი უმარტივესი რეკურსიული ფუნქცია არის ფაქტორიალი:

```
factorial :: Integer -> Integer

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

(აღსანიშნავია, რომ ჩვენ ვწერთ `factorial (n - 1)`, და არა `factorial n - 1` გამოსახულებას — გაიხსენეთ ოპერაციათა პრიორიტეტები).

რეკურსიის გამოყენებამ შეიძლება სიძნელეები გამოიწვიოს. რეკურსიის კონცეფცია მოგვაგონებს ინდუქციით მტკიცების მიდგომას, რომელიც მათემატიკაში გამოიყენება. ფაქტორიალის აქ მოცემულ განსაზღვრებაში ჩვენ გამოვყოფთ «ინდუქციის ბაზას» (`n == 0` შემთხვევას) და «ინდუქციის ბიჯს» — გადასვლას (`factorial n`)-დან (`factorial (n -`

1))-ზე). ასეთი კომპონენტების გამოყოფა მნიშვნელოვანი ნაბიჯია რეკურსიული ფუნქციის განსაზღვრის პროცესში.

4. არჩევის ოპერაცია და შეღწევათა გასწორების წესები

წინა სამუშაოში განხილული იყო პირობითი ოპერატორი. მისი ბუნებრივი გაგრძელება არის არჩევის `case` ოპერატორი, რომელიც `C` ენის `switch` კონსტრუქციის მსგავსია.

დავუშვათ, რომ გვჭირდება რაღაც (საკმაოდ უცნაური) ფუნქციის განსაზღვრა, რომელიც გვიბრუნებს 1-ს, თუ მას გადაეცა არგუმენტად 0; 5-ს, თუ არგუმენტი იყო 1; 2-ს, თუ არგუმენტია 2 და (-1)-ს ყველა დანარჩენ შემთხვევაში. ძირითადად, ამ ფუნქციის ჩაწერა `if` ოპერატორების საშუალებით შეიძლება, მაგრამ შედეგი იქნება გრძელი და ნაკლებად გასაგები. ასეთ შემთხვევებში გვეხმარება `case` კონსტრუქციის გამოყენება:

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

`case` ოპერატორის სინტაქსი სავსებით ცხადია ამ მაგალითიდან; საჭიროა მხოლოდ შენიშვნის გაკეთება იმის შესახებ, რომ (_) სიმბოლო `C` ენის `default` კონსტრუქციის ანალოგიურია. მაგრამ დაკვირვებულ მკითხველს შეიძლება კანონზომიერი შეკითხვა გაუჩნდეს: როგორ ახერ-

ხებს Haskell ენის ინტერპრეტატორი იმის გამოცნობას, თუ სად დასრულდა ერთი შემთხვევის განსაზღვრა და დაიწყო მეორის განსაზღვრა?

პასუხი ისაა, რომ Haskell ენაში გამოიყენება ტექსტის სტრუქტურირების ორგანზომილებიანი სისტემა (ანალოგიურ სისტემას იყენებენ უფრო ფართოდ ცნობილ Python ენაშიც). ეს სისტემა საშუალებას იძლევა არ გამოვიყენოთ ოპერატორთა დაჯგუფებისა და განცალკევების სპეციალური სიმბოლოები, რომლებიც C ენის {, } და ; სიმბოლოების მსგავსია.

თუმცა პრაქტიკულად Haskell ენაში ასევე შეიძლება ამ სიმბოლოების გამოყენება იმავე აზრით[3]. მაგალითად, ზემოთ მოცემული ფუნქცია ასეთი სახითაც შეიძლება ჩაიწეროს:

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

ეს ფრაგმენტი თვალსაჩინო ნიმუშია იმისა, თუ როგორ *არ უნდა* ხდებოდეს პროგრამის ტექსტის გაფორმება! ასეთი ხერხი ცხადი ფორმით ასა-

[3] იმ გამონაკლისით, რომ Haskell-ში ';' სიმბოლო, პასკალის მსგავსად, გამოიყენება როგორც ოპერატორების მაცალკევებელი და არა როგორც ოპერატორის დამთავრების ნიშანი.

ხავს ენის კონსტრუქციათა დავჯგუფებასა და დაყოფას. მაგრამ შეიძლება უარი ითქვას ამ მიდგომაზე.

ზოგადი წესი ასეთია. `where`, `let`, `do` და `of` საკვანძო სიტყვების შემდეგ ინტერპრეტატორი ათავსებს გამხსნელ (`{`) ფრჩხილს და იმახსოვრებს სვეტს, რომელშიც მომდევნო ბრძანებაა ჩაწერილი. შემდეგ ყოველი ახალი სტრიქონის წინ, რომელიც გასწორებულია დამახსოვრებული სიდიდით, მოთავსდება მაცალკევებელი `';'` სიმბოლო. თუ შემდეგი სტრიქონი ნაკლებადაა გასწორებული (ესე იგი თუ მისი პირველი სიმბოლო დამახსოვრებულ პოზიციაზე უფრო მარცხნივ იმყოფება), ხდება ჩამკეტი ფრჩხილის ჩასმა. ეს შეიძლება რამდენადმე რთულადაც კი გამოიყურებოდეს, მაგრამ სინამდვილეში ყველაფერი საკმარისად მარტივია.

აღწერილი წესის გამოყენებისას `f` ფუნქციის განსაზღვრის დროს, მივიღებთ, რომ მისი აღქმა ინტერპრეტატორის მიერ შემდეგნაირად ხდება:

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

ნეისმიერ შემთხვევაში შეიძლება არ გამოვიყენოთ ეს მექანიზმი და ყოველთვის ცხადად მივუთითოთ `{`, `}` და `;` სიმბოლოები. მაგრამ, კლავიშ-

ზე დაჭერის რაოდენობის შემცირებასთან ერთად, აღწერილი წესის გამოყენება მიღებულ პროგრამებს აადვილებს საკითხავად. მაშასადამე, ლაბორატორიული სამუშაოებისათვის ასეთი გაფორმების გამოყენება სავალდებულო შეიძლება გავხადოთ.

აუცილებელია კიდევ ერთი შენიშვნის გაკეთება. ვინაიდან Haskell ენაზე დაწერილ პროგრამაში ცარიელი პოზიციების დატოვება ძალიან მნიშვნელოვანია, განსაკუთრებული ყურადღებით მოვექცეთ ტაბულაციის სიმბოლოთა გამოყენებას. ინტერპრეტატორი თვლის, რომ ტაბულაციის სიმბოლო რვა ცარიელ პოზიციას უდრის. მაგრამ ზოგიერთი ტექსტური რედაქტორი ტაბულაციის ასახვის გადაწყობისა და ცარიელ პოზიციათა სხვა რაოდენობის ეკვივალენტად გარდასახვის საშუალებას იძლევა (მაგალითად, მინიშნების გარეშე Visual Studio რედაქტორში ტაბულაცია ოთხი ცარიელი პოზიციით აისახება). ამან შეიძლება შეცდომები გამოიწვიოს, თუ ერთ პროგრამაში ერთდროულად გამოვიყენებთ ცარიელ პოზიციებსა და ტაბულაციას. Haskell-ზე დაპროგრამებისას უკეთესია საერთოდ არ გამოვიყენოთ ტაბულაცია (მრავალი რედაქტორი ტაბულაციის კლავიშზე თითის დაჭერისას ცარიელი პოზიციების სასურველი რაოდენობის შეყვანის საშუალებას იძლევა).

5. ფუნქციის უბან-უბან მოცემა

ფუნქცია შეიძლება ცალკეულ უბნებზე იყოს განსაზღვრული. გაიხსენეთ, მაგალითად, ლოკალურად ანუ უბან-უბან მუდმივი ან წრფივი ფუნქციები მათემატიკაში. ეს ნიშნავს, რომ შესაძლებელია ფუნქციის ერთი ვერსიის განსაზღვრა გარკვეული პარამეტრებისათვის და მეორე ვერსიის

განსაზღვრა სხვა პარამეტრებისათვის. მაგალითად, `f` ფუნქცია წინა პუნქტიდან შეიძლება ასე განისაზღვროს:

```
f 0 = 1

f 1 = 5

f 2 = 2

f _ = -1
```

განსაზღვრის თანამიმდევრობა ამ შემთხვევაში მნიშვნელოვანია. ჩვენ რომ დაგვეწერა ჯერ `f _ = -1` ფუნქციის განსაზღვრება, მაშინ `f` დაგვიბრუნებდა (`-1`) სიდიდეს ნებისმიერი არგუმენტისათვის. ჩვენ რომ საერთოდ ამოგველო ეს სტრიქონი, მივიღებდით შეცდომას, თუ შევეცდებოდით მისი მნიშვნელობის გამოთვლას არგუმენტისათვის, რომელიც არ არის `0`, `1` ან `2`.

ფუნქციათა ასეთი განსაზღვრა საკმაოდ ფართოდ გამოიყენება `Haskell` ენაში. ამ მიდგომით ხშირად შესაძლებელია `if` და `case` ოპერატორებზეც უარის თქმა. მაგალითად, ფაქტორიალის ფუნქცია შეიძლება ასეთი ფორმითაც განისაზღვროს:

```
factorial 0 = 1

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

6. ნიმუშთან შედარება

მთელ რიცხვებზე მოცემული რეკურსიული ფუნქციების გარდა, შესაძლებელია რეკურსიული ფუნქციების განსაზღვრა სიებზე. ასეთ შემთხვევაში «რეკურსიის ბაზა» ცარიელი (`[]`) სია იქნება. განვსაზღვროთ ფუნქცია, რომელიც სიის სიგრძეს ანგარიშობს. ვინაიდან `length` სახელი უკვე დაკავებულია სტანდარტული ბიბლიოთეკით, გამოვიყენოთ `len` იდენტიფიკატორი:

```
len [] = 0

len s = 1 + len (tail s)
```

გავიხსენოთ, რომ სია, რომლის პირველი ელემენტი («თავი») არის `x`, ხოლო ყველა დანარჩენი ელემენტი («კუდი») მოცემულია `xs` სიით, ჩაიწერება `x:xs` ფორმით. თურმე ასეთი კონსტრუქციის გამოყენება შესაძლებელია ფუნქციის აღწერის დროსაც:

```
len [] = 0

len (x:xs) = 1 + len xs
```

(`xs` სტრიქონი უნდა მივიჩნიოთ `x`-ის მრავლობით რიცხვად, რომელიც ინგლისური ენის წესებით არის ნაწარმოები).

შევჩერდეთ კიდევ ერთ მაგალითზე. ფუნქცია, რომელიც იღებს შესასვლელზე რიცხვთა წყვილს და მათ ჯამს გვიბრუნებს, შეიძლება ასეთი სახით განვსაზღვროთ:

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

მაგრამ როგორ მოვიქცეთ, თუ აუცილებელია ისეთი ფუნქციის განსაზღვრა, რომელიც შესასვლელზე იღებს რიცხვთა *სამეულს* და მათ ჯამს გვიბრუნებს? ჩვენს განკარგულებაში არ არის `fst` და `snd` ფუნქციათა მსგავსი საშუალება სამეულის ელემენტთა ამოსაღებად. თურმე შეიძლება ასეთი ფუნქციების ჩაწერა შემდეგი სახით:

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

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

ამ ხერხს *ნიმუშთან შედარება* ეწოდება[4]. იგი ენის მეტად მძლავრ კონსტრუქციას წარმოადგენს, რომელიც გამოიყენება მის მრავალ ნაწილში, სახელდობრ, ფუნქციათა არგუმენტებსა და `case` ოპერატორის ვარიანტებში. ფუნქციის არგუმენტებში ჩაწერილი «ნიმუშების» «შედარება» ხდება მასში გადაცემულ ფაქტობრივ პარამეტრებთან.

თუ ნიმუშთან შედარება ხორციელდება, მასში ჩამოთვლილი ცვლადები შესაბამის მნიშვნელობებს იძენს. თუ ეს მნიშვნელობები ფუნქციის გამოთვლისას საჭირო არ არის (როგორც `my_tail` ფუნქციაში მომდევნო მაგალითიდან), მაშინ ზედმეტ სახელთა შემოტანის თავიდან ასაცილებლად, შეიძლება `( _ )` სიმბოლოს გამოყენება. იგი აღნიშნავს ნიმუშს, რომელთანაც ნებისმიერი მნიშვნელობის შედარება შეიძლება, მაგრამ თვით ეს მნიშვნელობა არავითარ ცვალდს არ მიებმება.

შემდეგი მაგალითები ნიმუშთან შედარების გამოყენებათა სხვადასხვა ვარიანტს იძლევა.

---

[4]ინგლისურად «`pattern matching`»

– სიის პირველი ორი ელემენტის შემკრები ფუნქცია:

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

– head ფუნქციის მსგავსი ფუნქციის განსაზღვრა:

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

– tail ფუნქციის მსგავსი ფუნქციის განსაზღვრა (ვიყენებთ _ ნიშანს, რადგან არ გვჭირდება სიის პირველი ელემენტი):

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

– სამეულის პირველი ელემენტის ამოღების ფუნქცია:

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

ნიმუშთან შედარების გამოყენება case ოპერატორშიც შეიძლება.

– სიის სიგრძის გამომთვლელი ფუნქციის კიდევ ერთი განსაზღვრა:

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

შესაძლებელია საკმაოდ რთული ნიმუშების მოცემა. განვსაზღვროთ ფუნქცია, რომელიც შესასვლელზე იღებს რიცხვთა წყვილების სიას და გვიბრუნებს მათი სხვაობების ჯამს:

```
f [(x1,y1),(x2,y2),...,(xn,yn)] = (x1-y1)+(x2- y2)+...+(xn– yn)
```

იგი ასე ჩაიწერება:

```
f [] = 0

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

7. სიის აგება

იმ ფუნქციის განსაზღვრისას, რომელიც სიას გვიბრუნებს, ხშირად გამოიყენება ოპერატორი (:) ორწერტილი. მაგალითად, ფუნქცია, რომელიც შესასვლელზე იღებს რიცხვთა სიას და გვიბრუნებს მათი კვადრატების სიას, შეიძლება შემდეგი სახით განისაზღვროს:

```
square [] = []

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

8. ზოგიერთი სასარგებლო ფუნქცია

ლაბორატორიული სამუშაოს შესრულებისას შესაძლებელია Haskell ენის შემდეგ სტანდარტულ ფუნქციათა გამოყენების საჭიროება გაჩნდეს:

• even-ი გვიბრუნებს True-ს ლუწი არგუმენტისათვის და False-ს – კენტისათვის.

• odd-ი წინა ფუნქციის მსგავსია, მაგრამ არგუმენტის კენტობაზე მოწმდება.

9. დავალება ლაბორატორიული სამუშაოსათვის

1. განსაზღვრეთ ფუნქცია, რომელიც შესასვლელზე იღებს მთელი ტიპის რომელიმე n რიცხვს და გვიბრუნებს ზრდადობის მიხედვით მოწესრიგებული n ელემენტის სიას.

1) ნატურალურ რიცხვთა სია.

2) კენტ ნატურალურ რიცხვთა სია.

3) ლუწ ნატურალურ რიცხვთა სია.

4) ნატურალურ რიცხვთა კვადრატების სია.

5) ფაქტორიალთა სია.

6) ორიანის ხარისხთა სია.

7) სამკუთხა რიცხვთა[5] სია.

8) პირამიდულ რიცხვთა[6] სია.

2. განსაზღვრეთ შემდეგი ფუნქციები:

1) ფუნქცია, რომელიც შესასვლელზე იღებს ნამდვილი რიცხვების სიას და ითვლის მათი მნიშვნელობების საშუალო არითმეტიკულს.

---

[5] n-ური სამკუთხა t_n რიცხვი უდრის ერთნაირი მონეტების რაოდენობას: მათი საშუალებით შეიძლება ტოლგვერდა სამკუთხედის აგება, რომლის თითოეულ გვერდზე n მონეტა თავსდება. ადვილად შეიძლება დავრწმუნდეთ, რომ $t_1=1$ და $t_n = n+t_{n-1}$.

[6] n-ური პირამიდული p_n რიცხვი უდრის ერთნაირი სფეროების რაოდენობას: მათი საშუალებით შეიძლება სამკუთხა ფუძის მქონე წესიერი პირამიდის აგება, რომლის თითოეულ გვერდზე n სფერო თავსდება. ადვილად შეიძლება დავრწმუნდეთ, რომ $p_1=1$ და $p_n= t_n+p_{n-1}$.

შეეცადეთ, რომ ფუქნცია ახორციელებდეს სიის გავლას მხოლოდ ერთხელ.

2) მოცემული სიის `n`-ური ელემენტის გამოცალკევების ფუნქცია.

3) ორი სიის ელემენტთა შეკრების ფუნქცია. გვიბრუნებს სიას, რომელიც შედგენილია პარამეტრებად მოცემულ სიათა ელემენტების ჯამებით. გაითვალისწინეთ, რომ გადაცემულ სიებს შეიძლება სხვადასხვა სიგრძე ჰქონდეს.

4) მოცემულ სიაში მეზობელ ლუწ და კენტ ელემენტთა გადაადგილების ფუნქცია.

5) `twopow n` ფუნქცია, რომელიც ითვლის 2^n სიდიდეს შემდეგი მოსაზრებებიდან გამომდინარე: დავუშვათ, რომ აუცილებელია `2`-ის აყვანა `n` ხარისხში. თუ `n` ლუწია, ე.ი `n = 2k`, მაშინ $2^n = 2^{2k} = (2^k)^2$. თუ `n` კენტია, ე.ი. `n = 2k + 1`, მაშინ $2^n = 2^{2k+1} = 2 \cdot (2^k)^2$. `twopow` ფუნქცია არ უნდა იყენებდეს (^) ოპერატორს ან ახარისხების ნებისმიერ ფუნქციას სტანდარტული ბიბლიოთეკიდან. ფუნქციის რეკურსიულ გამოძახებათა რაოდენობა `log n` სიდიდის პროპორციული უნდა იყოს.

6) `removeOdd` ფუნქცია, რომელიც მთელი რიცხვების მოცემულ სიაში სპობს ყველა კენტ რიცხვს. მაგალითად: `removeOdd [1,4,5,6,10]` ფუნქცია უნდა გვიბრუნებდეს `[4,6,10]` სიას.

7) `removeEmpty` ფუნქცია, რომელიც სტრიქონთა მოცემულ სიაში სპობს ცარიელ სტრიქონებს. მაგალითად:

```
removeEmpty ["", "Hello", "", "", "World!"]
```

ფუნქცია გვიბრუნებს სტრიქონთა `["Hello","World!"]` სიას.

8) `countTrue :: [Bool] -> Integer` ფუნქცია, რომელიც გვიბრუნებს სიის `True`-ს ტოლი ელემენტების რაოდენობას.

9) `makePositive` ფუნქცია, რომელიც რიცხვთა სიის ყველა უარყოფით ელემენტს უცვლის ნიშანს. მაგალითად:

```
makePositive [-1, 0, 5, -10, -20]
```

ფუნქცია `[1, 0, 5, 10, 20]` სიას იძლევა.

10) `delete :: Char -> String -> String` ფუნქცია, რომელიც შესასვლელზე იღებს სტრიქონსა და სიმბოლოს, ხოლო გვიბრუნებს სტრიქონს, საიდანაც ამოღებულია ამ სიმბოლოს ყველა ჩართვა. მაგალითად:

```
delete 'l' "Hello world!"
```

ფუნქცია უნდა გვიბრუნებდეს `"Heo word!"` სტრიქონს.

11) ფუნქცია:

`substitute :: Char -> Char -> String -> String`,

რომელმაც უნდა ჩაანაცვლოს მითითებული სიმბოლო (პირველი არგუმენტი) მოცემულით (მეორე არგუმენტი) გარკვეულ სტრიქონში (მესამე არგუმენტი) და დაგვიბრუნოს ახალი სტრიქონი. მაგალითად:

```
substitute 'e' 'i' "eigenvalue"
```

ფუნქცია გვიბრუნებს `"iiginvalui"` სტრიქონს.

10. ლაბორატორიული სამუშაოს შესრულების წესი

ლაბორატორიული სამუშაო ორი ნაწილისაგან შედგება. პირველ ნაწილში აუცილებელია პირველი პუნქტის *ყველა* დავალების შესრულება. მეორე ნაწილი კი შეიცავს მეორე პუნქტის ორ დავალებას. ეს დავალებები განაწილებულია ვარიანტების მიხედვით შემდეგი ცხრილის შესაბამისად:

ვარიანტი	1	2	3	4	5	6	7	8	9	10	11
დავალება	1,6	2,8	3,7	4,10	5,9	6,11	7,5	8,11	9 ,2	10,3	11,4

ანგარიშის გაფორმების წესები ისეთივეა, როგორც წინა სამუშაოში.

11. საკონტროლო შეკითხვები

1. შეწევათა გასწორების წესები.

2. ნიმუშთან შედარება.

3. არჩევის ოპერაცია.

4. ფუნქციათა უბან-უბან მოცემა.

ლაბორატორიული სამუშაო 3

1. let-დაკავშირება

ფუნქციის განსაზღვრისას ხშირად აუცილებელია გარკვეული დროებითი ცვლადის გამოყენება შუალედური შედეგების შესანახავად. გავიხსენოთ, როგორ გამოითვლება $ax^2 + bx + c = 0$ სახის კვადრატული განტოლების ფესვები:

$$x_{1,2} = \left(-b \pm \sqrt{b^2 - 4ac}\right) / 2a.$$

განტოლების ფესვთა წყვილის გამოსათვლელად შემდეგი ფუნქციის ჩაწერა შეიძლება:

```
roots a b c =

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

(-b - sqrt (b*b - 4*a*c)) / (2*a))
```

ფუნქციის ასეთ სტილში ჩაწერას შეიძლება ბევრი პრობლემა მოჰყვეს. ჯერ ერთი, ადვილად შეიძლება შეცდომის დაშვება ერთი და იმავე გამოსახულების მეორეჯერ გადაწერისას. მეორე, ამ პროგრამის წაკითხვისას გვიწევს ორი გამოსახულების შედარება იმის გასაგებად, რომ ისინი ერთსა და იმავეს წარმოადგენს. მესამეც, პროგრამა უფრო გრძელი ხდება. დასასრულ, იგი ნაკლებად ეფექტურია, ვიდრე შეიძლება ყოფილიყო, ვინაიდან კომპიუტერს უწევს ერთი და იმავე გამოთვლების ორჯერ ჩატარება.

ამ პრობლემათა თავიდან ასაცილებლად ენაში შეიძლება ლოკალური ცვლადების შემოღება. ფუნქცია შეიძლება ასე ჩაიწეროს:

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

`det` ლოკალური ფუნქცია მისაწვდომია მხოლოდ `roots` ფუნქციის განსაზღვრებაში.

შესაძლებელია რამდენიმე ლოკალური ცვლადის განსაზღვრა:

```
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)
```

ყურადღება მიაქციეთ, რომ `let ... in ...` კონსტრუქციაში შეწევათა გასწორების წესი გამოიყენება: ცარიელი პოზიციისგან განსხვავებული პირველი სიმბოლო `let` საკვანძო სიტყვის შემდეგ იძლევა სვეტს, რომლის მიმართაც უნდა გასწორდეს მომდევნო განსაზღვრებანი. `'{'`, `'}'` და `';'` სიმბოლოთა გამოყენებით შეწევათა გასწორების წესი უკვე სავალდებულო არ არის და `roots` ფუნქციის ჩაწერა ასეც შეიძლება:

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

`let ... in ...` კონსტრუქციის გარდა ზოგჯერ უფრო მოხერხებულია `... where ...` კონსტრუქციის გამოყენება, რომელშიც ლოკალური ცვლადების განსაზღვრებანი ძირითადი ფუნქციის შემდეგ თავსდება:

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

აღსანიშნავია, რომ შეიძლება არ შემოვიღოთ ლოკალური ცვლადები და მათ მაგივრად ლოკალური ფუნქციები გამოვიყენოთ:

```
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)
```

მაგრამ ასეთი მიდგომის ნაკლი სრულიად ნათელია: გარდა იმ არასასიამოვნო ფაქტისა, რომ ჩვენ შემოვიღეთ ორი დამხმარე ფუნქცია სახელების გლობალურ სივრცეში (რის გამოც ახლა ჩვენ ვერ გამოვიყენებთ, მაგალითად, `det` სახელს რომელიმე სხვა სასარგებლო ფუნქციისათვის), $\sqrt{b^2-4ac}$ და $2a$ მნიშვნელობათა გამოსათვლელად საჭირო ხდება შესაბამისი პარამეტრების გადაცემა ფუნქციისთვის მაშინ, როდესაც ლოკალურ განსაზღვრებებს შეუძლია იმ ფუნქციის პარამეტრების თავისუფლად გამოყენება, რომლის ფარგლებშიც მოცემულია ეს განსაზღვრებანი.

`let` და `where` კონსტრუქციებში შეიძლება არა მხოლოდ ცვლადების, არამედ ფუნქციების განსაზღვრაც. განვიხილოთ, მაგალითად, ფუნქცია, რომელიც მოცემული `n` რიცხვის საფუძველზე გვიბრუნებს ნატურალური რიცხვების `[1,2,...,n]` სიას. შემოვიღოთ `numsFrom` დამხმარე ფუნქცია, რომელიც მოცემული `m` რიცხვით გვიბრუნებს შესაბამის `[m,m+1,m+2,...,n]` სიას, და მისი განსაზღვრება ლოკალურ განსაზღვრებად ვაქციოთ:

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

ყურადღება მიაქციეთ იმ გარემოებას, რომ `numsFrom` ფუნქცია განსაზღვრებაში იყენებს `n` ცვლადს, თუმცა იგი არ გადაეცემა მას პარამეტრის სახით.

2. შეტყობინება შეცდომის შესახებ

ჩვენ მიერ განსაზღვრული ფუნქციები შეიძლება ვერ გამოითვლებოდეს არგუმენტის ზოგიერთი მნიშვნელობისათვის. გავიხსენოთ ფაქტორიალის ფუნქციის განსაზღვრება:

```
factorial 0 = 1

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

ეს ფუნქცია შესანიშნავად მუშაობს მანამ, სანამ ჩვენ არ შევეცდებით უარყოფითი რიცხვის ფაქტორიალის გამოთვლას. ნათლად ჩანს, რომ ასეთ შემთხვევაში გამოთვლა გადადის უსასრულო რეკურსიაში, ვინაიდან საბაზო შემთხვევა არასოდეს მიიღწევა.

ასეთ შეცდომებზე შეტყობინების მიღების უმარტივესი ხერხი არის `error` სტანდარტული ფუნქციის გამოყენება. იგი არგუმენტად იღებს სტრიქონს, მისი გამოთვლა კი იწვევს პროგრამის გაჩერებას და ეკრანზე ამ სტრიქონის გამოტანას. მაშასადამე, ფაქტორიალის ფუნქცია შეიძლება ასე ჩავწეროთ:

```
factorial 0 = 1

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

3. დამცველი პირობები

ნიმუშთან შედარება ფართო შესაძლებლობებს იძლევა ფუნქციის განსაზღვრისას. მაგრამ მისი საშუალებით, არსებითად, შეიძლება მხოლოდ ფუნქციაში გადაცემულ პარამეტრთა სტრუქტურისა და ამ პარამეტრების ელემენტთა კონსტანტურ მნიშვნელობებთან ტოლობის გამოყოფა. მაგრამ ხშირად ეს საკმარისი არ არის: აუცილებელია, რომ შემავალი პარამეტრები უფრო რთულ პირობებს აკმაყოფილებდეს.

მაგალითად, `factorial` ფუნქციის ზემოთ მოცემულ განსაზღვრებაში ჩვენ ვიყენებდით ნიმუშთან შედარებისა და პირობითი ოპერატორის შეხამებას. ნიმუშთან შედარება უფრო ეკონომიური და დამაჯერებელი ჩანს. შეიძლება თუ არა მსგავსი სინტაქსის გამოყენება პირობებისათვის? შეიძლება, დამცველი პირობების გამოყენებით. ამ პირობებით ფაქტორიალის ფუნქცია შემდეგნაირად ჩაიწერება:

```
factorial 0 = 1

factorial n | n < 0 = error "factorial: negative
argument"

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

ამ მაგალითიდან თვალნათლივ ჩანს სინტაქსი. შევნიშნავთ ასევე, რომ ბოლო პირობის მაგივრად შეიძლება `otherwise` (ინგლ. წინააღმდეგ შემთხვევაში) სიტყვის გამოყენება. მაგალითად, რიცხვის ნიშნის განმსაზღვრელ ფუნქციას შემდეგი სახე აქვს:

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

```
        | x == 0 = 0
        | otherwise = 1
```

ფუნქციათა ასეთ სტილში განსაზღვრა, ჩვეულებრივ, უფრო თვალსაჩინოა და Haskell ენაზე დაწერილ პროგრამაში დამცველი პირობები ფართოდ გამოიყენება (შესაბამისად, პირობითი ოპერატორი იშვიათად გამოიყენება). ამ თვალსაჩინოების საილუსტრაციოდ იგივე signum ფუნქცია განვსაზღვროთ პირობითი ოპერატორების გამოყენებით:

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

4. პოლიმორფული ტიპი

Haskell ენაში გამოიყენება პოლიმორფული ტიპების სისტემა. არსებითად ეს ნიშნავს იმას, რომ ენაში არის ტიპის მქონე ცვლადები. განვიხილოთ ჩვენთვის უკვე ცნობილი tail ფუნქცია, რომელიც გვიბრუნებს სიის კუდს (პირველი ელემენტის ამოღების შედეგად დარჩენილ სი-

ას). როგორია ამ ფუნქციის ტიპი? იგი ერთნაირად გამოსადეგია როგორც მთელი რიცხვების, ისე სიმბოლოებისა და სტრიქონების სიებისათვის:

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

[2,3]

Prelude>tail ['a','b','c']

['b','c']

Prelude>tail ["list", "of", "lists"]

["of", "lists"]
```

`tail` ფუნქციას პოლიმორფული ტიპი აქვს: `[a] -> [a]`. ეს იმას ნიშნავს, რომ არგუმენტის სახით იგი იღებს ნებისმიერ სიას და გვიბრუნებს იმავე ტიპის სიას. აქ `a` ნიშნავს ტიპის მქონე ცვლადს, ესე იგი იგულისხმება, რომ მის ნაცვლად შეიძლება ჩავსვათ ნებისმიერი კონკრეტული ტიპი. ამრიგად, `[a] -> [a]` ჩანაწერი ტიპების მთელ ოჯახს იძლევა, რომლის წარმომადგენლებია, მაგალითად:

```
[Integer] -> [Integer], [Char] -> [Char],

[[Char]] -> [[Char]].
```

ამის მსგავსად, `tail` ფუნქცია, რომელიც სიის კუდს გვიბრუნებს, ხასიათდება `[a] -> a` სტრუქტურის მქონე ტიპით. ამ ოჯახის წარმომადგენლებია შემდეგი ტიპები :

`[Integer] -> Integer, [Char] -> Char` და ა.შ.

სიებთან, წყვილებთან და კორტეჟებთან მომუშავე მრავალი ფუნქცია პოლიმორფული ტიპისაა. მაგალითად, `fst` ფუნქცია `(a,b) -> a` ტიპისაა (ყურადღება მიაქციეთ, რომ ამ ტიპის განსაზღვრებაში ტიპის მქონე ორი ცვლადი გამოიყენება).

5. სამომხმარებლო ტიპი

სტანდარტული ტიპების გარდა, პროგრამისტს საშუალება ეძლევა განსაზღვროს მონაცემთა საკუთარი, სპეციფიკური ტიპები. ამისათვის გამოიყენება `data` საკვანძო სიტყვა.

5.1. წყვილი

მაგალითისათვის განვიხილოთ შემდეგი წყვილის განსაზღვრება:

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

იგი ძალიან ჰგავს სტანდარტულ წყვილს. შევისწავლოთ ეს კოდი უფრო დაწვრილებით. `data` საკვანძო სიტყვა ნიშნავს, რომ ჩვენ ვაპირებთ მონაცემთა ტიპის განსაზღვრას. შემდეგ მოდის ამ ტიპის დასახელება, მოცემულ შემთხვევაში `Pair` (გავიხსენოთ, რომ ტიპის სახელი იწყება მთავრული ასოთი). `a` და `b` სიმბოლოები, რომლებიც ამის შემდეგ გვხვდება, ტიპის მქონე ცვლადებია, რომლებიც ტიპის პარამეტრებს აღნიშნავს. მაშასადამე, ჩვენ განვსაზღვრავთ მონაცემთა სტრუქტურას, რომელიც პარამეტრიზებულია ორი — `a` და `b` — ტიპით[7].

[7] ეს `C++` ენის კლასთა შაბლონებს მოგვაგონებს.

ტოლობის ნიშნის შემდეგ ჩვენ ამ ტიპის მონაცემთა კონსტრუქტორებს ვუთითებთ. მოცემულ შემთხვევაში ჩვენ ერთადერთი `Pair` კონსტრუქტორი გვაქვს (კონსტრუქტორის ტიპი სავალდებულო არ არის ემთხვეოდეს ტიპის სახელს, მაგრამ ჩვენს მაგალითში ეს სავსებით ბუნებრივად ჩანს). კონსტრუქტორის სახელის შემდეგ ჩვენ კვლავ ვათავსებთ `a b` ჩანაწერს, რაც იმას ნიშნავს, რომ წყვილის ასაგებად ჩვენ ორი მნიშვნელობა გვჭირდება: ერთი, რომელიც მიეკუთვნება `a` ტიპს, და მეორე – `b` ტიპს.

ამ განსაზღვრებას შემოაქვს `Pair :: a -> b -> Pair a b` ფუნქცია, რომელიც გამოიყენება `Pair` ტიპის წყვილების კონსტრუირებისათვის.

როდესაც ამ კოდს ინტერპრეტატორში ჩავტვირთავთ, შეიძლება ვნახოთ თუ როგორ ხდება წყვილების აგება:

```
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]
```

მონაცემთა კონსტრუქტორის შესაბამის ფუნქციას ის თვისება აქვს, რომ შესაძლებელია მისი გამოყენება ნიმუშთან შედარებისას. მაგალითად,

ფუნქციები ჩვენი წყვილის პირველი და მეორე ელემენტის მისაღებად შემდეგნაირად შეიძლება განისაზღვროს:

```
pairFst (Pair x y) = x

pairSnd (Pair x y) = y
```

გამორიცხული არ არის, რომ მოცემულმა მაგალითმა დაბადოს შეკითხვა: რა საჭიროა საკუთარი `Pair` ტიპის განსაზღვრა, თუ არსებობს წყვილის განსაზღვრის სტანდარტული შესაძლებლობა? ჯერ ერთი, `Pair` ტიპის გამოყენებით შეიძლება განისაზღვროს მხოლოდ ამ ტიპთან მომუშავე ფუნქციების ნაკრები და მოხდეს მათი გამოყოფა «საერთოდ» წყვილებთან მომუშავე ფუნქციებისაგან. მეორე, თუ ერთ ნაბიჯს წინ გადავდგამთ, შეიძლება დადგინდეს გარკვეული შეზღუდვები მოღებულ წყვილებზე, რომელთა მიღწევა შეუძლებელია სტანდარტული ტიპის საშუალებით. მაგალითად, წარმოიდგინეთ, რომ გვესაჭიროება ტიპი, რომელიც ერთი და იმავე ტიპის ელემენტთა წყვილს ინახავს. მისი განსაზღვრა შემდეგნაირად შეიძლება:

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

აქ ტიპს ერთი პარამეტრი აქვს, მაგრამ მონაცემთა კონსტრუქტორი ერთი და იმავე ტიპის ორ პარამეტრს იღებს.

5.2. მრავალფორმიანი კონსტრუქტორი

წინა მაგალითში განიხილებოდა მონაცემთა ტიპები ერთადერთი კონსტრუქტორით. ასევე შესაძლებელია (და ხშირად სასარგებლოცაა) ტიპის

განსაზღვრა რამდენიმე კონსტრუქტორით. კონსტრუქტორთა ერთმანეთისაგან განსაცალკევებლად `'|'` სიმბოლო გამოიყენება.

განვიხილოთ `Color` ტიპი, რომელიც წარმოადგენს ფერს `Red`, `Green` და `Blue` შესაძლო მნიშვნელობებით. მისი განსაზღვრა ასე შეიძლება:

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

აქ `Color` ტიპის დასახელებაა, ხოლო `Red`, `Green` და `Blue` – მონაცემთა კონსტრუქტორები. ყურადღება მიაქციეთ, რომ ეს ტიპი არ იღებს პარამეტრებს. ასეთ ტიპებს *თვლადი* (ე.ი. ჩამოთვლისუნარიანი, გადანომვრისუნარიანი) ეწოდება და `C` ენაში `enum` (ინგლ. ჩამოთვლა, ჩამონათვალი) კონსტრუქციას შეესაბამება. ასეთი ტიპები ძალიან სასარგებლოა. მაგალითად, სტანდარტული `Bool` ტიპი შემდეგნაირადაა განსაზღვრული:

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

მაგრამ ე.წ. *მრავალფორმიან კონსტრუქტორებს* (*multiple constructors*) შეუძლია პარამეტრების მიღებაც. მაგალითად, შეიძლება შევნიშნოთ, რომ ჩვენი `Color` ტიპი მხოლოდ სამი ფიქსირებული ფერის განსაზღვრის საშუალებას იძლევა. გავაფართოოთ იგი ისე, რომ ნებისმიერი ფერის განსაზღვრის საშუალებას იძლეოდეს. ყოველი ფერი მოიცემა სამი მთელი რიცხვით, რომლებიც შეესაბამება წითლის, მწვანისა და ლურჯის დონეებს (სტანდარტული `rgb`-წარმოდგენა):

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

აქ `Red`, `Green` და `Blue` სტანდარტული ფერების გარდა (მათი სია, რა თქმა უნდა, შეიძლება გაფართოვდეს), `Color` ტიპი ნებისმიერი ფერის სინთეზის საშუალებას იძლევა `RGB` კონსტრუქტორით, რომელიც ფერის `rgb` კომპონენტების განმსაზღვრელ სამ მთელ რიცხვს იღებს. მაშინ, მაგალითად, ფუნქცია ფერის `red` კომპონენტის გამოსაყოფად შემდეგი სახით ჩაიწერება:

```
redComponent :: Color -> Int

redComponent Red = 255

redComponent (RGB r _ _) = r

redComponent _ = 0
```

ტიპები მრავალფორმიანი კონსტრუქტორებით ასევე შეიძლება პოლიმორფული იყოს. განვიხილოთ შემდეგი პრობლემა — დავუშვათ, რომ ფუნქციამ უნდა დაგვიბრუნოს რაღაც შედეგი ან გამოიტანოს შეტყობინება შეცდომის შესახებ. მაგალითად, ფუნქცია წრფივი განტოლების ამოსახსნელად გვიბრუნებს ნაპოვნ ფესვს; ფუნქცია პირველი არაუარყოფითი რიცხვის მოსაძებნად სიაში გვიბრუნებს ამ რიცხვს და ა.შ. ამასთან ერთად განტოლების ამონახსნი შეიძლება არ არსებობდეს, სიაში არაუარყოფითი რიცხვები არ აღმოჩნდეს და ა.შ. როგორ შევატყობინოთ ამის შესახებ ფუნქციის გამომძახებელს? ზოგჯერ (როგორც არაუარყოფითი ელემენტის მიღების შემთხვევაში) შეიძლება წინასწარ შევთანხმდეთ, რომ რომელიმე სპეციალური მნიშვნელობის — მაგალითად, `(-1)`-ის — დაბრუნება

ნიშნავს: «შედეგი არ არის»[8]. მაგრამ ეს ყოველთვის არ არის შესაძლებელი: წრფივი განტოლების ამონახსნის შემთხვევაში ასეთი მონიშნული, გამორჩეული მნიშვნელობა არ არსებობს. პრობლემა კოხტად წყდება სტანდარტული `Maybe` ტიპის საშუალებით, რომელიც შემდეგი სახითაა განსაზღვრული:

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

`Maybe` (ინგლ. `maybe`-დან — შესაძლოა, შესაძლებელია) ტიპი პარამეტრიზებულია ტიპის მქონე `a` ცვლადით და მომხმარებლისთვის ორ კონსტრუქტორს იძლევა: `Nothing` (ინგლ. არაფერი) — შედეგის არარსებობის წარმოსადგენად და `Just` (ინგლ. მხოლოდ, ზუსტად) — აზრიანი, გონივრული შედეგისათვის. მაშინ ჩვენი ფუნქციების ჩაწერა ასეც შეიძლება:

1) ფუნქცია გვიბრუნებს `ax + b = 0` განტოლების ფესვს

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

2) ფუნქცია გვიბრუნებს სიის პირველ არაუარყოფით ელემენტს

```
findPositive :: [Integer] -> Maybe Integer
findPositive [] = Nothing
```

[8] `C` ენის სტანდარტული ბიბლიოთეკის მრავალი ფუნქცია ასეც იქცევა.

```
findPositive (x:xs) | x > 0 = Just x
                    | otherwise = findPositive xs
```

`Maybe` ტიპის გამოყენებას მრავალი უპირატესობა აქვს. მისი ცხადი ფორმით გამოყენება გვიჩვენებს, რომ ფუნქციას შეუძლია დაგვიბრუნოს «შედეგის არარსებობა». მონიშნული, გამორჩეული მნიშვნელობის გამოყენებისას იმის გასაგებად, თუ როგორ გვატყობინებს ფუნქცია ამ მნიშვნელობას, აუცილებელია ფუნქციის დოკუმენტაციის შესწავლა (რომელიც შეიძლება არც არსებობდეს ან არასწორი იყოს). `Maybe`-ს შემთხვევაში ეს ინფორმაცია არის ფუნქციის ტიპში და მისი მოცემა თვით ინტერპრეტატორს შეუძლია. უფრო მეტიც, ფუნქციის მიერ დასაბრუნებელი მნიშვნელობის დამუშავებისას აუცილებელია მისი ცხადი ფორმით ნიმუშთან შედარება და, თუ დაგვავიწყდება `Nothing`-ით შემთხვევის დამუშავება, კომპილატორს შეუძლია გაფრთხილების გამოტანა.

5.3. ტიპების კლასი

ტიპების კლასს მოგვიანებით დაწვრილებით შევისწავლით. აქ კი მის შესახებ მხოლოდ საბაზო წარმოდგენებს განვიხილავთ, რადგან ეს არსებითად აადვილებს მომხმარებლის ტიპთან მუშაობას.

ტიპების კლასი არის მრავალი საერთო თვისების მქონე ტიპის გარკვეული სიმრავლე. მაგალითად, ტიპების `Eq` კლასში შედის ის ტიპები, რომელთა ობიექტებისათვის განსაზღვრულია ტოლობის მიმართება, ე.ი., თუ `x` და `y` ცვლადები `Eq` კლასში შემავალ ერთსა და იმავე ტიპს მიეკუთვნება, შეგვიძლია `x == y` და `x /= y` გამოსახულებათა გამო-

თვლა. ყველა მარტივი ტიპი, ასევე სიები და კორტეჟები წარმოდგენილია ამ კლასში, მაგრამ, მაგალითად, ფუნქციისათვის ტოლობის მიმართება განსაზღვრული არ არის და ფუნქციათა ტიპები არ მიეკუთვნება Eq კლასს.

ჩვენთვის მნიშვნელოვანი კიდევ ერთი კლასი Show კლასია. მასში შედის ის ტიპები, რომელთა ობიექტები შეიძლება სტრიქონად ვაქციოთ, რათა ეს უკანასკნელი ეკრანზე ავსახოთ. ამ კლასში შედის მარტივი ტიპები, კორტეჟები და სიები. ამიტომ ინტერპრეტატორს შეუძლია დაბეჭდოს, მაგალითად, სტრიქონი. ფუნქციები ამ კლასში არ შედის.

მინიშნების გარეშე მომხმარებლის ტიპები არც ერთ კლასში არ შედის, ამიტომ ამ ტიპების მნიშვნელობათა შედარება და ინტერპრეტატორის მიერ მათი დაბეჭდვა შეუძლებელია. ეს, ცხადია, მოუხერხებელია. ამიტომ, ტიპების განსაზღვრისას შეიძლება მათი მიკუთვნების მოცემა სასურველი კლასებისადმი. ამისათვის ტიპის განსაზღვრის შემდეგ აუცილებელია საკვანძო deriving სიტყვის დამატება და ფრჩხილებში იმ კლასების ჩამოთვლა, რომლებსაც უნდა მიეკუთვნებოდეს ტიპი. მაგალითი:

ტიპი, რომელიც დღის გარკვეულ შუალედს წარმოაჩენს

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

აქ გამოყენებულია ინგლისური სიტყვები: `Day` – დღე (დღე-ღამე), `Time` – დრო (დროის შუალედი), `Morning` – დილა (დღის პირველი ნახევარი), `Afternoon` – ნაშუადღევი, `Evening` – საღამო, `Night` – ღამე. ტიპის განსაზღვრისას დავალებაში მიაკუთვნეთ ის ან `Eq`, ან `Show` კლასს. ეს არსებითად გააადვილებს სამუშაოს.

6. დავალება

ლაბორატორიული სამუშაოს შესასრულებლად საჭიროა ქვემოთ მოცემულ დავალებათაგან ერთ-ერთის გაკეთება. დავალების ნომერი შეესაბამება სტუდენტის ვარიანტის ნომერს.

1. თანამედროვე ვებმაღაზიაში ხშირად ყიდიან წიგნებს, ვიდეოკასეტებს და კომპაქტ-დისკებს. ასეთი მაღაზიის მონაცემთა ბაზა უნდა შეიცავდეს საქონლის თითოეული ტიპის შემდეგ მახასიათებლებს:

- დასახელება და ავტორი;
- ვიდეოკასეტები: დასახელება;
- კომპაქტ-დისკები: დასახელება, შემსრულებელი და კომპოზიციათა რაოდენობა.

    1) დაამუშავეთ მონაცემთა `Product` ტიპი, რომელსაც შეუძლია საქონლის ამ სახეობათა წარმოდგენა.

    2) განსაზღვრეთ `getTitle` ფუნქცია, რომელიც საქონლის სახელს გვიბრუნებს.

3) მის საფუძველზე განსაზღვრეთ `getTitles` ფუნქცია, რომელიც საქონლის სიის მიხედვით მათ დასახელებათა სიას გვიბრუნებს.

4) განსაზღვრეთ `bookAuthors` ფუნქცია, რომელიც საქონლის სიის მიხედვით წიგნების ავტორთა სიას გვიბრუნებს.

5) განსაზღვრეთ ფუქნცია

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

რომელიც გვიბრუნებს მოცემული დასახელების საქონელს (ყურადღება მიაქციეთ ფუნქციის შედეგის ტიპს).

6) განსაზღვრეთ ფუნქცია

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

იგი პარამეტრების სახით იღებს დასახელებათა სიას და საქონლის სიის ყოველი დასახელებისათვის მეორე სიიდან ამოაქვს შესაბამისი საქონელი. სახელები, რომლებსაც არავითარი საქონელი არ შეესაბამება, იგნორირებული რჩება. ფუნქციის განსაზღვრისას აუცილებლად გამოიყენეთ `lookupTitle` ფუნქცია.

2. განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს ინფორმაციას ბანქოში კარტის შესახებ. ყოველი კარტი ხასიათდება ერთ-ერთი ფერით ოთხიდან (აგური ანუ წკენტა; გული, თხილი ანუ ნაზუქი; ჯვარი; ყვავი ანუ პიკი). კარტი შეიძლება იყოს ან უმცროსი (ორიანიდან ათიანამდე), ან სურათი (ვალეტი; ქალი; მეფე ანუ პაპა; კიკო ანუ ტუზი). განსაზღვრეთ ფუნქციები:

1) `isMinor` ფუნქცია, რომელიც ამოწმებს, რომ მისი არგუმენტი უმცროსი კარტია.

2) `sameSuit` ფუნქცია, რომელიც ამოწმებს, რომ მისთვის გადაცემული კარტები ერთი ფერისაა.

3) `beats :: Card -> Card -> Bool` ფუნქცია, რომელიც ამოწმებს, რომ მისთვის პირველ არგუმენტად გადაცემული კარტი ჭრის მეორე არგუმენტად გადაცემულ კარტს.

4) `beats2` ფუნქცია, რომელიც `beats` ფუნქციის ანალოგიურია, მაგრამ დამატებით არგუმენტად იღებს კოზირის (თულფის) ფერს.

5) `beatsList` ფუნქცია, რომელიც არგუმენტებად იღებს კარტების სიას, ერთ ცალკე კარტს და კოზირის (თულფის) ფერს, ხოლო გვიბრუნებს პირველი არგუმენტიდან იმ კარტებს, რომლებიც ჭრის მოცემულ კარტს კოზირის ფერის გათვალისწინებით.

6) ფუნქცია, რომელიც კარტების მოცემული სიის საფუძველზე გვიბრუნებს იმ რიცხვთა სიას, რომელთაგან თითოეული არის ხსენებული კარტების ქულათა შესაძლო ჯამი. ამ ქულების გამოთვლა ხდება იმ თამაშის წესებით, რომელსაც «ოცდაერთი» ეწოდება: უმცროსი კარტების ქულა განისაზღვრება მათი ნომინალებით, ვალეტის, ქალისა და პაპის – ათი ქულით, ხოლო ტუზს შეიძლება მივაწეროთ როგორც ერთი, ისე თერთმეტი ქულა. ფუნქციამ უნდა დაგვიბრუნოს ყველა შესაძლო ვარიანტი.

3. განსაზღვრეთ ტიპი, რომელიც წარმოადგენს გეომეტრიულ ფიგურებს სიბრტყეზე. ფიგურა შეიძლება იყოს წრეწირი (რომელიც ხასიათდება ცენტრის კოორდინატებითა და რადიუსით), მართკუთხედი (რომელიც ხასიათდება ზედა მარცხენა და ქვედა მარჯვენა კუთხეთა კოორდინატებით), სამკუთხედი (რომელიც ხასიათდება წვეროთა კოორდინატებით) და ტექსტური ველი ანუ არე (რომლისთვისაც აუცილებელია მარცხენა ქვედა კუთხის მდგომარეობის, შრიფტისა და წარწერის ამსახველი სტრიქონის შენახვა). შრიფტი მოიცემა სამი შესაძლო შრიფტის სიმრავლიდან. ესენია: `Courier`, `Lucida` და `Fixedsys`. განსაზღვრეთ შემდეგი ფუნქციები:

1) `area` ფუნქცია, რომელიც გვიბრუნებს ფიგურის ფართობს. ტექსტური ველის ფართობი დამოკიდებულია შრიფტში ასოთა სიმაღლესა და სიგანეზე. ვინაიდან ჩვენ მიერ არჩეულ შრიფტებში ყველა ასოს სიგანე მუდმივია, თქვენ აუცილებლად უნდა განსაზღვროთ ასევე დამხმარე ფუნქცია, რომელიც ყოველი შრიფტისათვის მის გაბარიტებს გვიბრუნებს.

2) `getRectangles` ფუნქცია, რომელიც ირჩევს ფიგურათა სიიდან მხოლოდ მართკუთხედებს.

3) `getBound` ფუნქცია, რომელიც მოცემული ფიგურის საფუძველზე გვიბრუნებს მის შემომსაზღვრელ მართკუთხედს.

4) `getBounds` ფუნქცია, რომელიც ფიგურათა სიის საფუძველზე გვიბრუნებს მათი შემომსაზღვრელი მართკუთხედების სიას.

5) `getFigure` ფუნქცია. იგი ფიგურათა მოცემული სიისა და წერტილის კოორდინატთა საფუძველზე გვიბრუნებს პირველ ფიგურას, რომლისთვისაც წერტილი ხვდება მის შემომსაზღვრელ მართკუთხედში. გამოიყენეთ `Maybe` ტიპი დასაბრუნებელი მნიშვნელობისათვის.

6) `move` ფუნქცია. იგი მოცემული ფიგურისა და წანაცვლების ვექტორის საფუძველზე გვიბრუნებს ახალ ფუგურას, რომელიც წანაცვლებულია მოცემულის მიმართ ხსენებული ვექტორით.

4. უძრავი ქონების სააგენტოში იყიდება ბინები, ოთახები და კერძო სახლები. ბინა ხასიათდება სართულით, ფართობით და სახლის სართულიანობით. ოთახი ხასიათდება, გარდა ამისა, ოთახის ფართობით (მთელი ბინის ფართობთან დამატებით). კერძო სახლი ხასიათდება მხოლოდ ფართობით. მონაცემთა ბაზაში ინახება მნიშვნელობათა წყვილები. მათ შორის პირველი წარმოადგენს უძრავი ქონების ობიექტს, მეორე კი — მის ფასს. განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს ინფორმაციას უძრავი ქონების ასეთი ობიექტების შესახებ. განსაზღვრეთ შემდეგი ფუნქციები:

1) `getHouses` ფუნქცია, რომელიც მონაცემთა ბაზიდან ირჩევს მხოლოდ კერძო სახლებს.

2) `getByPrice` ფუნქცია, რომელიც მონაცემთა ბაზიდან ირჩევს უძრავი ქონების მხოლოდ იმ ობიექტებს, რომელთა ფასი მითითებულზე ნაკლებია.

3) `getByLevel` ფუნქცია, რომელიც მონაცემთა ბაზიდან ირჩევს მითითებულ სართულზე განლაგებულ ბინებს

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

დაამუშავეთ მონაცემთა ტიპი, რომელიც ასახავს სხვადასხვა მოთხოვნას უძრავი ქონების ობიექტების მიმართ: მინიმალური ფართობი, მაქსიმალური ფასი, შეზღუდვა სართულის მიმართ.

შექმენით `query` ფუნქცია: მოთხოვნათა სიის საფუძველზე იგი ირჩევს მონაცემთა ბაზიდან უძრავი ქონების იმ ობიექტებს, რომლებიც ყველა მოთხოვნას აკმაყოფილებს.

5. ბიბლიოთეკაში ინახება წიგნები, გაზეთები და ჟურნალები. წიგნი ხასიათდება ავტორის სახელით და სათაურით; ჟურნალი – დასახელებით, გამოშვების თვით და წლით; გაზეთი – დასახელებით და გამოსვლის თარიღით. მონაცემთა ბაზა წარმოადგენს ამ ობიექტების სიას. დაამუშავეთ მონაცემთა ტიპი, რომელიც ასახავს საბიბლიოთეკო დაცვის ობიექტებს. განსაზღვრეთ შემდეგი ფუნქციები:

1) `isPeriodic` ფუნქცია, რომელიც ამოწმებს, რომ მისი არგუმენტი პერიოდული გამოცემაა.

2) `getByTytle` ფუნქცია, რომელიც დაცვის ობიექტთა სიიდან (მონაცემთა ბაზიდან) ირჩევს ობიექტებს მითითებული დასახელებით.

3) `getByMonth` ფუნქცია, რომელიც მონაცემთა ბაზიდან ირჩევს მითითებულ თვესა და მითითებულ წელს გამოშვებულ პერიოდულ გამოცემებს (აღსანიშნავია, რომ გაზეთები თვეში რამდენჯერმე გამოდის).

4) `getByMonths` ფუნქცია, რომელიც წინა ფუნქციის მსგავსად მოქმედებს, მაგრამ შესასვლელზე იგი იღებს თვეების სიას.

5) `getAuthors` ფუნქცია, რომელიც გვიბრუნებს მონაცემთა ბაზაში დაცულ გამოცემათა ავტორების სიას.

6. დაპროგრამების გარკვეულ ენაში არსებობს მონაცემთა შემდეგი ტიპები:

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

პროგრამის იდენტიფიკატორთა მონაცემების ბაზა არის ისეთი წყვილების სია, რომლებიც შედგება იდენტიფიკატორის სახელისა და მისი ტიპისაგან. დაამუშავეთ მონაცემთა ტიპი, რომელიც წარმოადგენს აღწერილ ინფორმაციას. განსაზღვრეთ შემდეგი ფუნქციები:

1) `isStructured` ფუნქცია – ამოწმებს, რომ მისი არგუმენტი რთული ტიპისაა.

2) `getType` ფუნქცია, რომელიც მოცემული სახელისა და იდენტიფიკატორების სიის (მონაცემთა ბაზის) საფუძველზე გვიბრუნებს იდენტიფიკატორის ტიპს მითითებული სახელით (გახსოვდეთ, რომ ასეთი იდენტიფიკატორი ბაზაში შეიძლება საერთოდ არ აღმოჩნდეს).

3) `getFields` ფუნქცია, რომელიც მოცემული სახელის საფუძველზე გვიბრუნებს იდენტიფიკატორთა ველების სიას, თუ ეს იდენტიფიკატორი სტრუქტურის ტიპისაა.

4) `getByType` ფუნქცია, რომელიც მითითებული ტიპის იდენტიფიკატორთა სახელების სიას გვიბრუნებს მონაცემთა ბაზიდან.

5) `getByTypes` ფუნქცია, რომელიც წინა ფუნქციის მსგავსია, მაგრამ ერთი ტიპის ნაცვლად იგი ტიპების სიას იღებს (ამ ფუნქციის საშუალებით შეიძლება მივიღოთ, მაგალითად, რიცხობრივი ტიპის მქონე ყველა იდენტიფიკატორის სია).

7. განვსაზღვროთ ოპერაციათა შემდეგი ნაკრები სტრიქონისათვის:

- გაწმენდა: ყველა სიმბოლოს ამოღება სტრიქონიდან;
- ამოღება: მითითებული სიმბოლოს ყველა ჩართვის ამოღება;
- შეცვლა: ერთი სიმბოლოს ყველა ჩართვის შეცვლა მეორე სიმბოლოთი;
- დამატება: სტრიქონის დასაწყისში მითითებული სიმბოლოს დამატება.

დაამუშავეთ მონაცემთა ტიპი, რომელიც ახასიათებს ოპერაციებს სტრიქონებზე. განსაზღვრეთ შემდეგი ფუნქციები:

1) `process` ფუნქცია, რომელიც არგუმენტად იღებს მოქმედებასა და სტრიქონს, ხოლო გვიბრუნებს მითითებული მოქმედების შესაბამისად მოდიფიცირებულ სტრიქონს.

2) `processAll` ფუნქცია, რომელიც წინა ფუნქციის მსგავსია, მაგრამ შესასვლელზე იღებს მოქმედებათა სიას და თანამიმდევრობით ასრულებს ამ მოქმედებებს.

3) `deleteAll` ფუნქცია, რომელიც შესასვლელზე იღებს ორ სტრიქონს და მეორე სტრიქონიდან აგდებს პირველის ყველა სიმბოლოს. რეალიზაციისას აუცილებელია `processAll` ფუნქციის გამოყენება.

8. ელექტრონულ უბის წიგნაკში ინახება შემდეგი სახის ჩანაწერები: ნაცნობის დაბადების დღის შეხსენება, ნაცნობის ტელეფონი და დანიშნული შეხვედრა.

შეხსენება შედგება ნაცნობის სახელისა და თარიღისაგან (დღე და თვე). ჩანაწერი ტელეფონის შესახებ უნდა შეიცავდეს ადამიანის სახელსა და მის ტელეფონს. ინფორმაცია დანიშნული შეხვედრის შესახებ შეიცავს შეხვედრის თარიღს (დღე, თვე, წელი) და მოკლე აღწერას (შესაძლებელია სტრიქონის სახით).

დაამუშავეთ მონაცემთა ტიპი, რომელიც ასეთ ჩანაწერს ასახავს. უბის წიგნაკი განიხილება ჩანაწერების სიად. განსაზღვრეთ შემდეგი ფუნქციები:

1) `getByName` ფუნქცია, რომელიც გვიბრუნებს ინფორმაციას მითითებული სახელის მქონე ადამიანის შესახებ (მის ტელეფონს და დაბადების თარიღს).

2) `getByLetter` ფუნქცია – გვიბრუნებს იმ ადამიანთა სიას, რომლებზეც ინფორმაცია არის უბის წიგნაკში, ხოლო ასეთი ადამიანის სახელი იწყება მითითებული ასოთი.

3) `getAssignment` ფუნქცია, რომელიც მითითებული თარიღის საფუძველზე გვიბრუნებს საქმეთა სიას (ინფორმაციას დანიშნული შეხვედრების შესახებ და იმ მეგობრების ტელეფონებს, რომლებსაც უნდა მივულოცოთ ეს დღე).

9. კლავიშები, კლავიატურაზე შეიძლება იყოს ან მმართველი, ან ანბანურ-ციფრული. ანბანურ-ციფრულ კლავიშზე დაჭერას შეიძლება თან სდევდეს `Shift` კლავიშზე დაჭერა. მმართველი კლავიშებიდან ჩვენ ახლა მხოლოდ `CapsLock` კლავიში გვაინტერესებს, ხოლო დანარჩენები შეიძლება დროებით დავივიწყოთ.

ანბანურ-ციფრულ კლავიშზე ყოველ დაჭერას თან მოაქვს ინფორმაცია სიმბოლოს სახით. `CapsLock` კლავიშზე დაჭერის შემდეგ, მომდევნო სიმბოლოები გადადის ზედა რეგისტრში (თუ მათთან ერთად `Shift` კლავიშზე არ იყო დაჭერილი), `CapsLock` კლავიშზე მორიგ დაჭერამდე.

თუ `CapsLock`-ის რეჟიმი გააქტიურებული არ არის, `Shift` კლავიშთან ერთად დაჭერილი სიმბოლოები ზედა რეგისტრში გადავა.

დაამუშავეთ მონაცემთა ტიპი, რომელიც აღნიშნულ ინფორმაციას წარმოგვიდგენს. კლავიშების დაჭერის მიმდევრობა აისახება სიით. ძირითადი ამოცანა ისაა, რომ შეიქმნას ფუნქცია, რომელიც ამ მიმდევრობას სიმბოლოების სტრიქონად აქცევს. მაგალითად, დაჭერათა

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

მიმდევრობამ შედეგად უნდა მოგვცეს `HeLLo` სტრიქონი.

განსაზღვრეთ შემდეგი ფუნქციები:

1) `getAlNum` ფუნქცია, რომელიც დაჭერათა სიიდან გვიბრუნებს მხოლოდ ანბანურ-ციფრული კლავიშის დაჭერას.

2) `getRaw` ფუნქცია, რომელიც გვიბრუნებს დაჭერილი სიმბოლოებით შედგენილ სტრიქონს `Shift` და `CapsLock` კლავიშათა შესახებ ინფორმაციის გაუთვალისწინებლად.

3) `isCapsLocked` ფუნქცია, რომელიც დაჭერათა მიმდევრობის საფუძველზე არკვევს, დარჩა თუ არა მის შემდეგ `CapsLock` რეჟიმი გააქტიურებული.

4) `getString` ფუნქცია, რომელიც დაჭერათა მიმდევრობას სტრიქონად აქცევს.

ფუნქციათა რეალიზაციისას შეიძლება `toUpper` და `toLower` სტანდარტული ფუნქციების გამოყენება, რომლებსაც გადაჰყავს სიმბოლო შესაბამისად ზედა და ქვედა რეგისტრებში. ისინი განსაზღვრულია `Char`

მოდულში და მათი გამოყენებისათვის პროგრამის დასაწყისში უნდა დაე-მატოს `import Char` სტრიქონი.

10. სასწავლო სემესტრის განმავლობაში სტუდენტმა უნდა ჩააბაროს ლაბორატორიული სამუშაოების, საანგარიშო-გრაფიკული დავალებისა და რეფერატების გარკვეული რაოდენობა.

ლაბორატორიული სამუშაო ხასიათდება საგნის დასახელებით და ნომრით, საანგარიშო-გრაფიკული დავალება — საგნის დასახელებით, რეფერატი — საგნის დასახელებით და რეფერატის თემის დასახელებით.

დაამუშავეთ მონაცემთა ტიპი, რომელიც ასახავს დავალებებთან დაკავში-რებულ ამ ინფორმაციას.

სტუდენტის სასწავლო გეგმა განიხილება სიად, რომელიც შედგება წყვილებისაგან. მათი პირველი ელემენტი არის დავალება, ხოლო მეორე — იმ სასწავლო კვირის რიგითი ნომერი, როდესაც ეს დავალება ჩაბარდა.

თუ დავალება ჯერ ჩაბარებული არ არის, წყვილის მეორე ელემენტი ცარიელი უნდა რჩებოდეს (გამოიყენეთ `Maybe` ტიპი).

განსაზღვრეთ შემდეგი ფუნქციები:

1) `getByTitle` — გვიბრუნებს დავალებას, რომლის ჩაბარება აუ-ცილებელია მითითებულ საგანში.

2) `getReferats` — გვიბრუნებს რეფერატების თემათა სიას.

3) getRest – გვიბრუნებს ჩაუბარებლად დარჩენილი დავალებების სიას.

4) `getRestForWeek` – გვიბრუნებს მითითებულ კვირაში ჩაუბარებლად დარჩენილი დავალებების სია.

5) `getPlot` – ქმნის წყვილებით შედგენილ სიას. ყოველი სიის პირველი ელემენტი სასწავლო კვირის რიგით ნომერს უდრის, მეორე კი – ამ კვირისათვის ჩაბარებულ დავალებათა რაოდენობას.

7. საკონტროლო შეკითხვები

1. ლოკალური ცვლადების განსაზღვრა.

2. დამცველი პირობები.

3. პოლიმორფიზმი.

4. მონაცემთა სამომხმარებლო ტიპების განსაზღვრა.

ლაბორატორიული სამუშაო 4

1. ოპერატორის განსაზღვრა

ბინარული ოპერატორები, როგორიცაა +, - და მისთანანი Haskell ენაში, ისეთივე ფუნქციებს წარმოადგენს, როგორც ყველა დანარჩენი, იმის გამოკლებით, რომ მათი გამოძახებისათვის შესაძლებელია ინფიქსური ნოტაციის გამოყენება. თუ ბინარულ ოპერატორს ფრჩხილებში მოვათავსებთ, მაშინ მის გამოსაძახებლად შეიძლება პრეფიქსული ნოტაციის გამოყენება და მუშაობა როგორც ჩვეულებრივ ფუნქციასთან. ამრიგად, შემდეგი ჩანაწერები ეკვივალენტურია:

```
2 + 2

(+) 2 2

x < y

(<) x y

x /= y

(/=) x y
```

პირიქითაც, ნებისმიერი ფუნქცია, რომელიც ორ არგუმენტს იღებს, შეიძლება გამოვიყენოთ ინფიქსურ სტილში. ამისათვის მისი სახელი უნდა ჩავსვათ შექცეულ ბრჭყალებში, რომლისთვისაც გამოიყენება (`) სიმბოლო (კლავიატურათა უმრავლესობაზე ეს სიმბოლო განთავსებულია მარცხენა ზედა კუთხეში Esc კლავიშის ქვეშ, ქვედა რეგისტრში). მაგალითად, რომ განისაზღვროს

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

ფუნქცია, მისი გამოძახება შემდეგი ორი სახით შეიძლება:

```
func 5 2
```

```
5 `func` 2
```

კიდევ ერთი ინფორმაცია. თუ ფუნქციის სახელში გვხვდება მხოლოდ «სიმბოლოები» (არც ასოები და არც ციფრები), მაშინ იგი ავტომატურად ჩაითვლება ინფიქსურ ოპერატორად. განსაზღვრისას მისი სახელი უნდა მოთავსდეს ფრჩხილებში. მაგალითად, განვსაზღვროთ ოპერატორი «მიახლოებითი ტოლობა»: იგი ამოწმებს, რომ ორი რიცხვი განსხვავდება ერთმანეთისაგან `0.001`-ზე ნაკლები სიდიდით:

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

ახლა ეს ოპერატორი ყველა დანარჩენი ოპერატორის მსგავსად შეიძლება გამოვიყენოთ:

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

2. რეკურსიული ტიპი

მონაცემთა ტიპის განსაზვრისას განმარტების მარჯვენა ნაწილში შეიძლება ამ კონსტრუქციით დადგენილი ტიპის გამოყენება. ეს მონაცემთა რეკურსიული სტრუქტურების განსაზღვრის საშუალებას გვაძლევს. ასეთი ერთ-ერთი ძირითადი სტრუქტურა არის ხე.

განვსაზღვროთ ბინარული ხე, რომლის ფოთლებში a ტიპის ელემენტები განლაგებულია შემდეგი სახით:

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

ეს განსაზღვრება გვეუბნება, რომ ხე (`Tree`) არის ან ფოთოლი (`Leaf`), ე.ი. კვანძი, რომელსაც შთამომავლები არ ჰყავს, ან შტო (`Branch`), ე.ი. კვანძი, რომელსაც აქვს მარცხენა და მარჯვენა ქვეხე. უნდა შევნიშნოთ, რომ ამ განმარტებაში `Leaf` და `Branch` მონაცემთა კონსტრუქტორებია, ხოლო განსაზღვრების მარცხენა და მარჯვენა ნაწილებში წარმოდგენილი `Tree a` კონსტრუქცია — ტიპის სახელი.

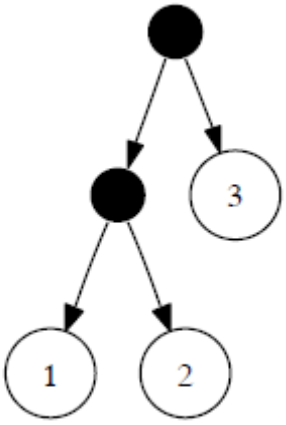
რეკურსიულ ტიპებთან მუშაობა პრაქტიკულად არ განსხვავდება ჩვეულებრივ ტიპებთან მუშაობისაგან, იმ გამონაკლისით, რომ რეკურსიულ ტიპებთან მომუშავე პრაქტიკულად ყველა ფუნქცია თვითონ არის რეკურსიული. მაგალითად, განვსაზღვროთ `treeSize` ფუნქცია, რომელიც გვიბრუნებს ხის ფოთლების რიცხვს. იგი შემდეგნაირად ჩაიწერება:

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

ამ ფუნქციის გამოყენება კი მოგვცემს:

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

აქ იგი ნახმარია ხისათვის, რომელიც მეშვიდე სურათზეა ნაჩვენები.



სურ. 7. `treeSize` ფუნქციის შესაბამისი ხე

ხესთან სამუშაოდ განკუთვნილი ფუნქციის კიდევ ერთი მაგალითია ფუნქცია ხის ყველა ფოთლის სიის მისაღებად:

```
leafList (Leaf x)            = [x]

leafList (Branch left right) = leafList left ++
                               leafList right
```

3. სია როგორც რეკურსიული ტიპი

სია რეკურსიული ტიპიც არის. განვიხილოთ შემდეგი პოლიმორფული ტიპი:

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

`List a` ტიპის მნიშვნელობა ან ცარიელია (`Nil`), ან შეიცავს `a` ტიპის ელემენტს და `List a` ტიპის მნიშვნელობას. ადვილი შესამჩნევია პირდაპირი ანალოგია სიებთან, რომლებიც ან ცარიელია (`[]`), ან შეიცავს `a` ტიპის თავს და კუდს — ასევე სიას. მსგავსება კიდევ უფრო ცხადი გახდება, თუ `Cons` კონსტრუქტორს ინფიქსური სახით ჩავწერთ:

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

ამრიგად, სიის ტიპი შეიძლება შემდეგი ფორმით იყოს განსაზღვრული:

— ეს არ არის `Haskell` ენის ნამდვილი კოდი

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

`List` ტიპის მნიშვნელობებისათვის შეიძლება სიებთან სამუშაოდ განკუთვნილი ყველა ფუნქციის განსაზღვრა. განვიხილოთ `head, tail` და `map` ფუნქციათა მაგალითები:

```
headList (Cons x _) = x

headList Nil        = error "headList: empty list"

tailList (Cons _ y) = y

tailList Nil        = error "tailList: empty list"
```

ამ ფუნქციათა მუშაობის ილუსტრაციას შემდეგი ჩანაწერები იძლევა:

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

1
```

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

Cons 2 Nil
```

4. სინტაქსური ხე

ხისებრი სტრუქტურები ფართოდ გამოიყენება დაპროგრამებაში. მაგალითად, პროგრამის გრამატიკული გარჩევის შედეგი ნებისმიერ კომპილატორში არის სინტაქსური ხე. განვიხილოთ ასეთი ხის მაგალითი გამოსახულებებისათვის, რომლებიც შეიცავს კონსტანტებს, შეკრებისა და გამრავლების სიმბოლოებს:

```
data Expr = Const Integer

          | Add Expr Expr

          | Mult Expr Expr
```

ამ განმარტებიდან ჩანს, რომ გამოსახულება `(Expression)` არის ან მთელრიცხვა კონსტანტა `(Constant)`, ან ორი გამოსახულების ჯამი თუ ნამრავლი. მაგალითად, `1 + 2 * (3 + 4)` გამოსახულებისათვის `Expr` ტიპის შესაბამისი მნიშვნელობა ასეთი სახისაა:

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

გამოსახულების მნიშვნელობის გამოთვლის ფუნქცია შეიძლება შემდეგნაირად განვსაზღვროთ:

```
eval :: Expr -> Integer

eval (Const x) = x
```

```
eval (Add x y) = eval x + eval y

eval (Mult x y) = eval x * eval y
```

`Expr` ტიპი შეიძლება გავაფართოოთ, თუ შემოვიღებთ ცვლადების გამოყენების შესაძლებლობას გამოსახულებებში:

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

`Var` კონსტრუქტორი განსაზღვრავს მითითებულსახელიან ცვლადს. ასეთი `Expr` ტიპი საშუალებას იძლევა განვსაზღვროთ, მაგალითად, ფუნქცია გამოსახულების დიფერენცირებისათვის:

```
diff :: Expr -> Expr

diff (Const _) = Const 0

diff (Var x) = Const 1

diff (Add x y) = Add (diff x) (diff y)

diff (Mult x y) = Add (Mult (diff x) y) (Mult x
(diff y))
```

შევამოწმოთ ამ ფუნქციის მუშაობა $x + x^2$ გამოსახულების დიფერენცირების მაგალითზე (არ დაივიწყოთ `deriving(Show)`-ს დამატება `Expr` ტიპის განსაზღვრის შემდეგ):

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

ამრიგად, დიფერენცირების შედეგად ჩვენ მივიღეთ $1+(1\cdot x+x\cdot 1)$ გამოსახულება, რომელიც სწორია, მაგრამ, ცხადია, გამარტივებას მოითხოვს.

`diff` ფუნქციის მეორე შეზღუდვა არის ის, რომ იგი ვერ არჩევს, თუ რომელი ცვლადით ხორციელდება დიფერენცირება. შესაბამისად, სინამდვილეში იგი უნდა იღებდეს დამატებით პარამეტრს — დიფერენცირების ცვლადის სახელს.

`Expr` ტიპის მნიშვნელობათა პირდაპირი მოცემა საკმაოდ მოუხერხებელია. უმთავრესად, შესაძლებელია ფუნქციის ჩაწერა, რომელიც `"1+x*y"` სახის სტრიქონს გარდაქმნის `Expr` ტიპის შესაბამის მნიშვნელობად. მაგრამ ასეთი ფუნქციის დაწერა საკმარისად შრომატევადია, ამიტომ სტუდენტს შეუძლია გამზადებული ფუნქციის გამოყენება. იგი განსაზღვრულია `expr.hs` ფაილში და მას `parseExpr` ეწოდება. ამავე ფაილშია განსაზღვრული `Expr` ტიპი. ამ ფაილის მისაერთებლად მოათავსეთ მისი პირი კატალოგში, სადაც თქვენი პროგრამა არის, და ამ პროგრამის თავში დაამატეთ სტრიქონი:

```
import Expr
```

`parseExpr` ფუნქციის ტიპი არის:

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

მოცემული სტრიქონით იგი გვიბრუნებს მის წარმოდგენას `Expr` ტიპის მნიშვნელობის სახით:

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

5. დავალება

1. `Expr` ტიპთან მუშაობა. ზემოთ განსაზღვრული `Expr` ტიპის გამოყენებით მიიღეთ შემდეგი ფუნქციები (ტესტირებისათვის გამოიყენეთ `parseExpr` ფუნქცია):

   1) განსაზღვრეთ `diff` კორექტული ფუნქცია; იგი იღებს დამატებით არგუმენტად იმ ცვლადის სახელს, რომლითაც აუცილებელია დიფერენცირების განხორციელება.

   2) განსაზღვრეთ `simplify` ფუნქცია, რომელიც ამარტივებს `Expr` ტიპის გამოსახულებებს, რისთვისაც გამოიყენეთ ცხადი სახის შემდეგი წესები:

      - `x + 0 = 0 + x = x`
      - `x · 1 = 1 · x = x`
      - `x · 0 = 0 · x = 0`

- და ასე შემდეგ

3) განსაზღვრეთ toString ფუნქცია, რომელიც `Expr` ტიპის გამოსახულებას სტრიქონად გარდაქმნის. მაგალითად, ამ ფუნქციის გამოყენებამ `Add (Mult (Const 2) (Var "x")) (Var "y")` გამოსახულების მიმართ უნდა მოგვცეს `"2*x+y"` სტრიქონი. გაითვალისწინეთ ფრჩხილების გამოყენების შესაძლებლობა, მაგალითად, `Mult (Const 2) (Add (Var "x") (Var "y"))` გამოსახულება უნდა გარდაიქმნებოდეს `"2*(x+y)"` სტრიქონად.

4) განსაზღვრეთ `eval` ფუნქცია, რომელიც იღებს შესასვლელზე ორ პარამეტრს: `Expr` ტიპის გამოსახულებასა და `(String, Integer)` ტიპის წყვილთა სიას. ხსენებული სიით ცვლადთა სახელებისა და მათი მნიშვნელობების შესაბამისობა მოიცემა. ფუნქცია უნდა ანგარიშობდეს გამოსახულების მნიშვნელობას გამოსახულებათა მოცემული მნიშვნელობების გათვალისწინებით. მაგალითად, `eval (Add (Var "x") (Var "y")) [("x",1),("y",2)]` გამოსახულებამ უნდა გამოიტანოს რიცხვი `3`.

2. ფუნქციები `List` ტიპტთან სამუშაოდ. ადრე შემოტანილი `List` ტიპისათვის განსაზღვრეთ შემდეგი ფუნქციები:

1) `lengthList`, რომელიც List ტიპის სიის სიგრძეს გვიბრუნებს;

2) `nthList`, რომელიც სიის `n`-ურ ელემენტს გვიბრუნებს;

3) `removeNegative`, რომელიც მთელი რიცხვების სიაში (`List Integer` ტიპი) ანადგურებს ყველა უარყოფით ელემენტს;

4) `fromList`, რომელიც List ტიპის სიას ჩვეულებრივ სიად გარდაქმნის;

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

3. ფუნქციები ძებნის ბინარულ ხეებთან სამუშაოდ. განსაზღვრეთ მონაცემთა ტიპი, რომელიც ძებნის ბინარულ ხეებს წარმოადგენს.

ჩვეულებრივი ხეებისაგან განსხვავებით, ძებნის ხეებში მონაცემები შეიძლება არა მხოლოდ ფოთლებში იყოს, არამედ – ხის შუალედურ კვანძებშიც.

გამოვიყენოთ ხეები ასოციაციური მასივის წარმოსადგენად. ეს ხეები ყოველი კვანძისათვის სტრიქონებით ასახულ *გასაღებთა* მნიშვნელობებს ადარებს მთელ რიცხვებს.

რაღაც გასაღების მქონე ყოველი კვანძისათვის მარცხენა ქვეხე უნდა შეიცავდეს ელემენტებს გასაღების ნაკლები მნიშვნელობით, ხოლო მარჯვენა ქვეხე – ელემენტებს გასაღების მეტი მნიშვნელობებით.

სტრიქონსა და რიცხვს შორის შესაბამისობის ძებნისას აუცილებელია ამ ინფორმაციის გათვალისწინება, ვინაიდან იგი ხიდან ინფორმაციის უფრო ეფექტურად ამოღების საშუალებას იძლევა.

განსაზღვრეთ მონაცემთა აღწერილი ტიპი და შემდეგი ფუნქციები:

1) `add`, რომელიც ამატებს ხეში გასაღებისა და მნიშვნელობის მოცემულ წყვილს.

2) `find`, რომელიც გვიბრუნებს მოცემული სტრიქონის შესაბამის რიცხვს.

3) `exists`, რომელიც ამოწმებს, შეიცავს თუ არა ხე ელემენტს მოცემული გასაღებით.

4) `toList`, რომელიც გარდაქმნის ძებნის მოცემულ ხეს გასაღებთა მნიშვნელობების შესაბამისად მოწესრიგებულ სიად.

4. დაამუშავეთ მონაცემთა ტიპი, რომელიც საფაილო სისტემის კატალოგის შიგთავსს წარმოადგენს.

ვთვლით, რომ თითოეული ფაილი ან შეიცავს გარკვეულ მონაცემებს, ან კატალოგია. კატალოგში მოცემულია სხვა ფაილები (რომლებიც, თავის მხრივ, შეიძლება კატალოგებს წარმოადგენდეს) მათ სახელებთან და ზომებთან ერთად.

იგულისხმება, რომ ფაილის ზომის ერთეულია ბაიტი. მოცემულ სამუშაოში ფაილების შიგთავსის იგნორირება შეიძლება: მონაცემთა ტიპი უნდა წარმოადგენდეს მხოლოდ მათ სახელებსა და კატალოგების სტრუქტურას.

განსაზღვრეთ შემდეგი ფუნქციები:

1) `dirAll`, რომელიც გვიბრუნებს კატალოგის (და ქვეკატალოგთა) ყველა ფაილის სრული სახელების სიას.

2) `find`, რომელიც გვიბრუნებს მოცემული სახელის მქონე ფაილისადმი მიმავალ გზას.

   მაგალითად, თუ კატალოგი შეიცავს `a`, `b` და `c` ფაილებს, ხოლო `b` არის `x`-ის და `y`-ის შემცველი კატალოგი, მაშინ ძებნის ფუნქცია `x`-სათვის უნდა გვიბრუნებდეს `"b/x"` სტრიქონს.

3) `du`, რომელიც მოცემული კატალოგისათვის გვიბრუნებს მისი (ასევე ქვეკატალოგთა) ფაილების მიერ დაკავებული ბაიტების რაოდენობას.

5. მტკიცება ვუწოდოთ ლოგიკურ ფორმულას, რომელსაც ერთ-ერთი შემდეგი ფორმა აქვს:

- ცვლადის სახელი (სტრიქონი)
- `p & q`
- `p | q`
- `~p`

აქ `p` და `q` მტკიცებებია. მაგალითად, მტკიცებებია შემდეგი ფორმულები:

- `x`
- `x | y`
- `x & (x | ~y)`

დაამუშავეთ მონაცემთა `Prop` ტიპი, რომელიც ასეთი სახის მტკიცებებს წარმოადგენს.

განსაზღვრეთ შემდეგი ფუნქციები:

1) `vars :: Prop -> [String]`, რომელიც (გამეორებათა გარეშე) გვიბრუნებს მტკიცებებში არსებული ცვლადების სახელთა სიას.

2) დავუშვათ, რომ მოცემულია ცვლადთა სახელებისა და `Bool` ტიპის მათი მნიშვნელობების სია, მაგალითად,

   `[("x",True),("y",False)]`.

   განსაზღვრეთ

   ```
   truthValue :: Prop -> [(String,Bool)] -> Bool
   ```

   ფუნქცია. იგი არკვევს საკითხს: სწორია მტკიცება, რომ ცვლადებს სიით მოცემული მნიშვნელიბები აქვს?

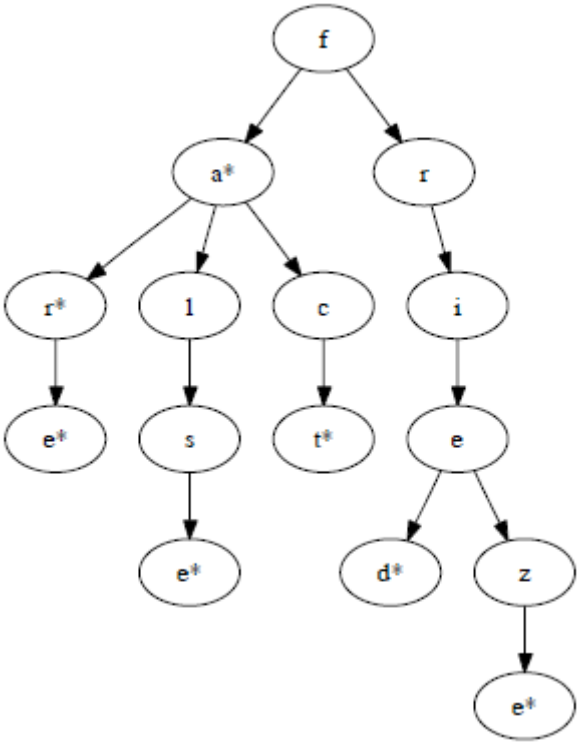
3) განსაზღვრეთ `tautology :: Prop -> Bool` ფუნქცია.

   იგი გვიბრუნებს `True`-ს, თუ მტკიცება სწორია ცვლადთა ყველა მნიშვნელობისათვის, რომლებიც მასში გვხვდება (მაგალითად, ეს სრულდება `(x | ~x)` მტკიცებისათვის).

6. ლექსიკური ხეები (`trie`-ხეები) გამოიყენება ლექსიკონების წარმოსადგენად.

ხის ყოველი კვანძი შეიცავს შემდეგ ინფორმაციას:

- სიმბოლოს,
- ბულის მნიშვნელობას,
- ქვეხეების სიას (ყოველ კვანძს შეიძლება ჰქონდეს შვილობილი ხეების ნებისმიერი რაოდენობა).



სურ. 8. ლექსიკური ხე (`trie`-ხე)

`trie`-ხის მაგალითი ნაჩვენებია მე-8 სურათზე. `True`-ს ტოლი ბულის მნიშვნელობა აღნიშნავს სიტყვის დასასრულს.

ყოველი სიტყვის წაკითხვა ხის ფესვიდან იწყება. სურათზე ამ მნიშვნელობების მქონე კვანძები მონიშნულია `( * )` სიმბოლოთი.

ამრიგად ხეში წარმოდგენილია `fa`, `false`, `far`, `fare`, `fact`, `fried`, `frieze` სიტყვები. განსაზღვრეთ შემდეგი ფუნქციები:

- `exists`, რომელიც ამოწმებს, რომ მოცემული სიტყვა მოიპოვება `trie`-ხეში.
- `insert`, რომელიც ხისა და სიტყვის საფუძველზე გვიბრუნებს ახალ ხეს, მასში ჩართული ამ სიტყვით. თუ სიტყვა ხეში არ მოიპოვება, ეს ხე უცვლელად უნდა გვიბრუნდებოდეს.
- `completions`, რომელიც მოცემული სტრიქონის საფუძველზე გვიბრუნებს ხიდან ყველა იმ სიტყვის სიას, რომელთა დასაწყისი მითითებული სტრიქონია. (მაგალითად, მე-2 სურათზე მოცემული ხისათვის `"fri"` სტრიქონის საფუძველზე უნდა გვიბრუნდებოდეს `["fried","frieze"]` სია).

7. თეორიულად შესაძლებელია, თუმცა ეფექტური არ არის, მთელი რიცხვების განსაზღვრა მონაცემთა რეკურსიული ტიპებით, შემდეგნაირად:

```
data Number = Zero | Next Number
```

ე.ი. რიცხვი არის ან ნული (`Zero`), ან განისაზღვრება როგორც რიცხვი, რომელიც წინა რიცხვს მოსდევს.

მაგალითად, რიცხვი `3` ჩაიწერება `Next (Next (Next Zero))` ფორმით.

ასეთი წარმოდგენისათვის განსაზღვრეთ შემდეგი ფუნქციები:

1) `fromInt`, რომელიც Integer ტიპის მოცემული მთელი რიცხვისათვის გვიბრუნებს Number ტიპის მის შესაბამის მნიშვნელობას.

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

3) `plus :: Number -> Number -> Number`, რომელიც თავისი არგუმენტების ჯამს პოულობს.

4) `mult :: Number -> Number -> Number`, რომელიც თავისი არგუმენტების ნამრავლს პოულობს.

5) `dec`, რომელიც თავის არგუმენტს ერთიანს აკლებს. `Zero`-სათვის ფუნქცია `Zero`-ს უნდა გვიბრუნებდეს.

6) `fact`, რომელიც ფაქტორიალს ანგარიშობს.

8. თანამდებობათა იერარქია ერთ-ერთ ორგანიზაციაში ხისებრ სტრუქტურას ქმნის. ყოველ მუშაკს, რომელიც ცალსახად ხასიათდება უნიკალური სახელით, რამდენიმე ხელქვეითი ჰყავს.

განსაზღვრეთ მონაცემთა ტიპი, რომელიც წარმოადგენს ასეთ იერარქიას, და აღწერეთ შემდეგი ფუნქციები:

1) `getSubordinate`, რომელიც მითითებული მუშაკის ხელქვეითთა სიას გვიბრუნებს.

2) `getAllSubordinate`, რომელიც მოცემული მუშაკის ყველა ხელქვეითის სიას გვიბრუნებს, არაუშუალო ხელქვეითების ჩათვლით.

3) `getBoss`, რომელიც მითითებული მუშაკის უფროსს გვიბრუნებს.

4) `getList`, რომელიც წყვილების სიას გვიბრუნებს. წყვილის პირველი ელემენტია მუშაკის სახელი, ხოლო მეორე — მისი ხელქვეითების რაოდენობა (არაუშუალო ხელქვეითების ჩათვლით).

9. არე სიბრტყეზე ან მართკუთხედია, ან წრე, ან არეთა გაერთიანება, ან მათი გადაკვეთა. მართკუთხედი ხასიათდება მარცხენა ქვედა და მარჯვენა ზედა კუთხეების კოორდინატებით, წრე — ცენტრის კოორდინატებით და რადიუსით. დაამუშავეთ მონაცემთა სტრუქტურა, რომელიც არის აღწერილი სახის არე. განსაზღვრეთ შემდეგი ფუნქციები:

1) `contains`, რომელიც ამოწმებს, რომ მოცემული წერტილი არეში ხვდება.

2) `isRectangular`, რომელიც ამოწმებს, რომ არე მოცემულია მხოლოდ მართკუთხედის სახით.

3) `isEmpty`, რომელიც ამოწმებს, რომ არე ცარიელია, ესე იგი სიბრტყის არც ერთი წერტილი არ ხვდება მასში.

10. ობიექტზე ორიენტირებულ ენაში, კლასი შეიცავს მეთოდების ნაკრებს (ამ სამუშაოში კლასის ველები მონაცემების სახით იგნორირებული იქნება).

გარდა ამისა, მას შეიძლება ჰქონდეს მშობელთა ერთადერთი კლასი (არ ვიხილავთ მრავალფორმიანი მემკვიდრეობითობის შემთხვევას).

მაგრამ არსებობს კლასები მშობლების გარეშეც. კლასის მემკვიდრეობითობის რეალიზაციისას, წინაპრის მეთოდები შთამომავლის მეთოდებს ემატება.

განსაზღვრეთ მონაცემთა ტიპი, რომელიც ასახავს ინფორმაციას კლასების იერარქიის შესახებ.

აღწერეთ შემდეგი ფუნქციები:

1) `getParent`, რომელიც გვიბრუნებს მითითებული სახელის მქონე კლასის უშუალო წინაპარს.

2) `getPath`, რომელიც გვიბრუნებს მოცემული კლასის ყველა წინაპრის სიას (უშუალო წინაპარი, ამ წინაპრის წინაპარი და ასე შემდეგ).

3) `getMethods`, რომელიც გვიბრუნებს მითითებული კლასის მეთოდების სიას მემკვიდრეობითობის გათვალისწინებით.

4) `inherit`, რომელიც კლასების იერარქიაში უმატებს მოცემული სახელის კლასს. ეს უკანასკნელი მიღებულია მემკვიდრეობით მითითებული წინაპრისაგან.

ლაბორატორიული სამუშაო 5

1. უმაღლესი რიგის ფუნქციები

განვიხილით ორი ამოცანა. დავუშვათ, რომ მოცემულია რიცხვების სია. აუცილებელია ორი ფუნქციის დაწერა, რომელთა შორის პირველი გვიბრუნებს ამ რიცხვთა კვადრატული ფესვების სიას, ხოლო მეორე — მათ ლოგარითმებს. ეს ფუნქციები შეიძლება ასე განვსაზღვროთ:

```
sqrtList [] = []

sqrtList (x:xs) = sqrt x : sqrtList xs

logList [] = []

logList (x:xs) = log x : logList xs
```

ადვილი შესამჩნევია, რომ ეს ფუნქციები იყენებს ერთსა და იმავე მიდგომას, ხოლო განსხვავება მათ შორის ისაა, რომ ახალი სიის ელემენტის გამოსათვლელად ერთ-ერთი იყენებს კვადრატული ფესვის ფუნქციას, ხოლო მეორე — ლოგარითმს. შესაძლებელია თუ არა ელემენტის გარდასახვის კონკრეტული ფუნქციისაგან აბსტრაჰირება? თურმე, შესაძლებელია. გავიხსენოთ, რომ `Haskell` ენაში ფუნქციები «პირველი კლასის» ელემენტებია, მათი გადაცემა სხვა ფუნქციებშიც დასაშვებია პარამეტრების სახით. განვსაზღვროთ `transformList` ფუნქცია, რომელიც იღებს ორ პარამეტრს: გარდაქმნის ფუნქციასა და გარდასაქმნელ სიას.

```
transformList f [] = []
```

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

ახლა `sqrtList` და `logList` ფუნქციები შეიძლება ასე განისაზღვროს:

```
sqrtList l = transformList sqrt l

logList l = transformList log l
```

ანუ კარირების გათვალისწინებით:

```
sqrtList = transformList sqrt

logList = transformList log
```

1.1. `map` ფუნქცია

სინამდვილეში ფუნქცია, რომელიც `transformList`-ის მსგავსია, უკვე მთლიანადაა განსაზღვრული ენის სტანდარტულ ბიბლიოთეკაში და მას `map`-ი ეწოდება (ინგლ. `map` — ასახვა). იგი შემდეგი სახისაა:

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

ეს ნიშნავს, რომ მისი პირველი არგუმენტია ტიპის ფუნქცია `a -> b`, რომელიც ნებისმიერი `a` ტიპის მნიშვნელობებს ასახავს `b` ტიპის მნიშვნელობებად (ზოგადად, ეს ტიპები შეიძლება ემთხვეოდეს ერთმანეთს). ფუნქციის მეორე არგუმენტია `a` ტიპის მნიშვნელობათა სია. მაშინ ფუნქციის შედეგი `b` ტიპის მნიშვნელობათა სია იქნება.

`map`-ის მსგავს ფუნქციას, რომელიც არგუმენტად სხვა ფუნქციას იღებს, უმაღლესი რიგის ფუნქცია ეწოდება. ის ფართოდ გამოიყენება ფუნქციონალური პროგრამების დაწერისას. მათი საშუალებით შე-

საძლებელია ალგორითმის რეალიზაციის კერძო წვრილმანები (მაგალითად, `map`-ად გარდასახვის კონკრეტული ფუნქცია) ცხადად გამოვყოთ ამ ალგორითმის მაღალი დონის სტრუქტურიდან (როგორიცაა, ვთქვათ, სიის ყოველი ელემენტის გარდასახვა). როგორც წესი, მაღალი რიგის ფუნქციათა გამოყენებით წარმოდგენილი ალგორითმები გაცილებით უფრო კომპაქტური და დამაჯერებელია, ვიდრე კონკრეტულ უნმიშვნელო წვრილმანზე ორიენტირებული რეალიზაციები.

1.2. `filter` ფუნქცია

ფართოდ გამოყენებადი მაღალი რიგის ფუნქციის კიდევ ერთ მაგალითია `filter` ფუნქცია. მოცემული პრედიკატით (ე.ი. ბულის მნიშვნელობის დამაბრუნებელი ფუნქციით) და სიით იგი გვიბრუნებს იმ ელემენტების სიას, რომლებიც აკმაყოფილებს ხსენებულ პრედიკატს:

```
filter :: (a -> Bool) -> [a] -> [a]

filter p [] = []

filter p (x:xs) | p x = x : filter xs

                | otherwise = filter xs
```

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

```
getPositive = filter isPositive

isPositive x = x > 0
```

1.3. `foldr` და `foldl` ფუნქციები

უფრო რთული მაგალითებია `foldr` და `foldl` ფუნქციები. განვიხილოთ ფუნქციები, რომლებიც გვიბრუნებს სიის ელემენტების ჯამსა და ნამრავლს.

```
sumList [] = 0

sumList (x:xs) = x + sumList xs

multList [] = 1

multList (x:xs) = x * multList xs
```

აქ, ასევე შეიძლება დავინახოთ საერთო ელემენტები: საწყისი მნიშვნელობა (`0` შეკრებისათვის, `1` გამრავლებისათვის) და ფუნქცია, რომელიც მნიშვნელობათა კომბინირებას, შერჩევას ახორციელებს. `foldr` ფუნქცია ამ სქემის თვალნათლივი განზოგადებაა:

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

foldr f z [] = z

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

`foldr` ფუნქცია პირველ არგუმენტად შემრჩევ ანუ მაკომბინირებელ ფუნქციას იღებს (გავიხსენოთ, რომ მას შეუძლია სხვადასხვა ტიპის არგუმენტის მიღება, მაგრამ შედეგის ტიპი უნდა ემთხვეოდეს მეორე არგუმენტის ტიპს). `foldr` ფუნქციის მეორე არგუმენტი არის საწყისი მნიშვნელობა კომბინირებისათვის. მესამე არგუმენტით კი გადაიცემა სია.

ფუნქცია ახორციელებს სიის «შეკვეცას», ქმნის მისგან «კომპოზიციას» გადაცემული პარამეტრების შესაბამისად.

იმისათვის, რომ უკეთ გავერკვეთ `foldr` ფუნქციის მუშაობაში, მისი განსაზღვრება ჩავწეროთ ინფიქსური ნოტაციის გამოყენებით:

```
foldr f z [] = z

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

ელემენტების `[a,b,c,...,z]` სია გამოვსახოთ `( : )` ოპერატორის გამოყენებით. `foldr` ფუნქციის გამოყენების წესი ასეთია: ყველა `(:)` ოპერატორი იცვლება `f` ფუნქციის გამოყენებით (`` `f` ``)ინფიქსური სახით, ხოლო ცარიელი სიის `[]` სიმბოლო იცვლება კომბინირების საწყისი მნიშნელობით. გარდაქმნის ნაბიჯები შეიძლება ასე გამოვსახოთ (ვუშვებთ, რომ საწყისი მნიშვნელობა `init`-ის ტოლია):

```
[a,b,c,...,z]

a : b : c : ... : []

a : (b : (c : (... (z : [])...)))

a `f` (b `f` (c `f` (... (z `f` init))))
```

`foldr` ფუნქციის საშუალებით სიის ელემენტთა შეკრებისა და გამრავლების ფუნქციები ასე განისაზღვრება:

```
sumList = foldr (+) 0

multList = foldr (*) 1
```

ვნახოთ, როგორ გამოითვლება ამ ფუნქციათა მნიშვნელობები `[1,2,3]` სიის მაგალითზე:

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

ამის მსგავსად გამრავლებისათვის:

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

`foldr` და `foldl` ფუნქციების სახელები ნაწარმოებია ინგლისური სიტყვა `fold`-ისგან, რაც ქართულად გაღუნვას, გაკეცვას (მაგალითად, ქაღალდის ფურცლის) ნიშნავს. `r` ასო ფუნქციის დასახელებაში გაჩნდა სიტყვა `right`-ისგან (მარჯვენა) და გვიჩვენებს შესაკვეცად გამოსაყენებელი ფუნქციის ასოციაციურობას (ჯუფთებადობას). ამრიგად, განხილული მაგალითებიდან ჩანს, რომ `foldr` ფუნქციის გამოყენება ჯგუფდება მარჯვნივ. `foldl` ფუნქციის განსაზღვრება, სადაც `l` ასო (სიტყვისგან `left` - მარცხენა) გვიჩვენებს, რომ ოპერაციის გამოყენება ჯგუფდება მარცხნივ, მოცემულია ქვევით:

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

```
foldl f z []     = z

foldl f z (x:xs) = foldl f (f z x) xs
```

ასოციაციური ოპერაციებისათვის, როგორიცაა შეკრება და გამრავლება, `foldr` და `foldl` ფუნქციები ეკვივალენტურია, მაგრამ, თუ ოპერაცია ასოციაციური არ არის, მათი შედეგები განსხვავებული იქნება:

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

2

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

-6
```

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

1.4. უმაღლესი რიგის სხვა ფუნქციები

სტანდარტულ ბიბლიოთეკაში განსაზღვრულია `zip` ფუნქცია. იგი ორ სიას წყვილთა სიად გარდაქმნის:

```
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')]
```

ყურადღება მიაქციეთ იმ გარემოებას, რომ დაბრუნებული სიის სიგრძე ყველაზე მოკლე საწყისი სიის სიგრძის ტოლია.

ამ ფუნქციის განზოგადება უმაღლესი რიგის `zipWith` ფუნქციაა, რომელიც «აერთიანებს» ორ სიას, მომხმარებლის მიერ მითითებული რომელიღაც ფუნქციის საშუალებით:

```
zipWith                 :: (a->b->c) -> [a]->[b]->[c]

zipWith z (a:as) (b:bs) = z a b : zipWith z as bs

zipWith _ _ _           = []
```

`zipWith` ფუნქციის გამოყენებით ადვილად განისაზღვრება, მაგალითად, ორი სიის ელემენტთა შეკრების უკვე ნახსენები `sumList` ფუნქცია:

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

ანუ, კარირების გათვალისწინებით:

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

2. ლამბდა-აბსტრაქცია

უმაღლესი რიგის ფუნქციათა გამოყენებისას ხშირად აუცილებელი ხდება მრავალი მცირე ფუნქციის განსაზღვრა. მაგალითად, `getPositive`

ფუნქციის განსაზღვრისას ჩვენ მოგვიწია `isPositive` დამატებითი ფუნქციის განსაზღვრა, რომელიც მხოლოდ იმისთვის არის საჭირო, რომ შეამოწმოს დადებითია თუ არა არგუმენტი. პროგრამის მოცულობის ზრდისას დამხმარე ფუნქციისათვის სახელის დარქმევის აუცილებლობა სულ უფრო მეტად უშლის ხელს პროგრამისტს. მაგრამ `Haskell` ენაში, ისევე როგორც მის საფუძვლად გამოყენებულ ლამბდა-აღრიცხვაში, უსახელო ფუნქციის განსაზღვრა შეიძლება ლამბდა-აბსტრაქციის კონსტრუქციის საშუალებით.

მაგალითად, განვიხილოთ სამი უსახელო ფუნქცია: პირველი პოულობს თავისი არგუმენტის კვადრატს, მეორე უმატებს თავის არგუმენტს ერთიანს, მესამე კი არგუმენტს ორზე ამრავლებს. ეს ფუნქციები შემდეგი ფორმით მოიცემა:

```
\x -> x * x
```

```
\x -> x + 1
```

```
\x -> 2 * x
```

ახლა შესაძლებელია მათი გამოყენება უმაღლესი რიგის ფუნქციათა არგუმენტებში. მაგალითად, სიის ელემენტთა კვადრატში ახარისხების ფუნქცია ასე ჩაიწერება:

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

`getPositive` ფუნქცია შეიძლება შემდეგნაირად განისაზღვროს:

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

ლამბდა-აბსტრაქციის განსაზღვრა რამდენიმე ცვლადისთვისაც არის შესაძლებელი:

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

ლამბდა-აბსტრაქციისა და ჩვეულებრივი ფუნქციის გამოყენება ერთნაირად დასაშვებია.

ლამბდა-აბსტრაქციის გამოყენება შეიძლება არგუმენტად. მაგალითად:

```
Main>(\x -> x + 1) 2

3
```

ან კიდევ:

```
Main>(\x -> x * x) 5

25

Main>(\x -> 2 * x + y) 1 2

4
```

ლამბდა-აბსტრაქციის საშუალებით შეიძლება ფუნქციათა განსაზღვრა. მაგალითად,

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

ეს ჩანაწერი შემდეგი გამოსახელების ეკვივალენტურია:

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

3. სექცია

ფუნქციის გამოყენება ნაწილობრივაც შეიძლება. სხვანაირად რომ ვთქვათ, ყველა არგუმენტის მნიშვნელობის მოცემა სავალდებულო არ არის. მაგალითად, თუ `add` ფუნქცია განსაზღვრულია როგორც

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

მაშინ შესაძლებელია `inc` ფუნქციის განსაზღვრა, რომელიც შემდეგნაირად ზრდის თავის არგუმენტს `1`-ით:

```
inc = add 1
```

თურმე, როგორც ენაში ჩაშენებული, ისე მომხმარებლის მიერ განსაზღვრული ბინარული ოპერატორი, შეიძლება გამოვიყენოთ თავისი არგუმენტების მხოლოდ ნაწილის მიმართ (ვინაიდან არგუმენტების რაოდენობა ბინარულ ოპერატორში ორს უდრის, ეს ნაწილი ერთი არგუმენტისაგან შედგება). ბინარულ ოპერაციას, რომელსაც ერთი არგუმენტის მიმართ იყენებენ, სექცია ეწოდება.

მაგალითად:

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

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

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

ფრჩხილების ხმარება აქ სავალდებულოა. ამრიგად, `add` და `inc` ფუნქციები შეიძლება ასე განვსაზღვროთ:

```
add = (+)

inc = (+1)
```

სექცია განსაკუთრებით სასარგებლოა მისი გამოყენებისას უმაღლესი რიგის ფუნქციის არგუმენტად. გავიხსენოთ სიის დადებითი ელემენტების მისაღებად განკუთვნილი ფუნქციის განსაზღვრება:

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

სექციების გამოყენებით იგი გაცილებით უფრო კომპაქტურად ჩაიწერება:

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

ფუნქცია სიის ელემენტთა გასაორკეცებლად:

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

4. დავალება

1. განსაზღვრეთ შემდეგი ფუნქციები უმაღლესი რიგის ფუნქციათა გამოყენებით:

1) ნამდვილ რიცხვთა სიის ელემენტების საშუალო არითმეტიკულის გამოთვლის ფუნქცია `foldr` ფუნქციის გამოყენებით. ფუნქცია უნდა ახორციელებდეს სიის მხოლოდ ერთ გავლას.

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

3) `countEven` ფუნქცია, რომელიც გვიბრუნებს სიის ლუწი ელემენტების რაოდენობას.

4) `quicksort` ფუნქცია, რომელიც ახორციელებს სიის სწრაფ დახარისხებას შემდეგი რეკურსიული ალგორითმით. იმისათვის, რომ დახარისხდეს `xs` სია, ამ სიიდან ამოიღება პირველი ელემენტი (აღვნიშნოთ იგი `x`-ით). დანარჩენი სია იყოფა ორ ნაწილად: ერთია სია, რომელიც შედგენილია `x`-ზე მცირე `xs` ელემენტებისაგან და მეორე — სია, რომელიც `x`-ზე დიდი ელემენტებისაგან შედგება. ხდება ამ სიათა დახარისხება (აქ ვლინდება რეკურსია, რადგან მათი დახარისხება ამავე ალგორითმით ხორციელდება), ხოლო შემდეგ, ხსენებული სიების გამოყენებით, `as++[x]++bs` სახის საბოლოო სია იწერება, სადაც `as` და `bs`, შესაბამისად, მცირე და დიდი ელემენტების შემცველი დახარისხებული სიებია.

5) წინა პუნქტში განსაზღვრული `quicksort` ფუნქცია სიას ზრდადობით ახარისხებს. განაზოგადეთ ეს ფუნქცია: დავუშვათ, რომ იგი კიდევ ერთ არგუმენტს იღებს შესასვლელზე — `a -> a -> Bool` ტიპის შედარების ფუნქციას და სიის დახარისხებას ახორციელებს სწორედ ამ ფუნქციის შესაბამისად.

2. დაუბრუნდით მესამე ლაბორატორიული სამუშაოს დავალებებს და შეასრულეთ ისინი უმაღლესი რიგის ფუნქციათა გამოყენებით. შეეცადეთ მთლიანად გამორიცხოთ სიის ცხადი გავლა ფუნქციათა განსაზღვრებიდან.

ლაბორატორიული სამუშაო 6

1. მოდული

პროგრამა Haskell ენაზე სხვადასხვა მოდულის ნაკრებია. ყოველი მოდული ორ მიზანს ემსახურება – სახელების სივრცის მართვასა და მონაცემთა აბსტრაქტული ტიპების შექმნას.

მოდულის სახელი მთავრული ასოთი იწყება; Hugs ინტერპრეტატორში მოდულის ტექსტი უნდა იყოს მოთავსებული ცალკე ფაილში, რომლის სახელი ემთხვევა მოდულის სახელს; ამ ფაილის გაფართოება კი (.hs) ფორმით უნდა იყოს მოცემული.

პრაქტიკული თვალსაზრისით მოდული module საკვანძო სიტყვით დაწყებული მხოლოდ ერთი დიდი გამოცხადებაა და მეტი არაფერი. განვიხილოთ მოდული, რომლის სახელია 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
```

Tree ტიპი და leafList ფუნქცია შემოვიღეთ მეოთხე ლაბორატორიულ სამუშაოში.

მოდული ცხადად ახდენს `Tree`, `Leaf`, `Branch` და `leafList` სახელების ექსპორტს. მოდულიდან ექსპორტირებადი სახელების ჩამოთვლა ხდება ფრჩხილებში `module` საკვანძო სიტყვის შემდეგ. თუ ეს ჩამოთვლა მითითებული არ არის, მაშინ მინიშნების გარეშე მოდულიდან ყველა სახელის ექსპორტირება ხორციელდება. მიაქციეთ ყურადღება, რომ ტიპისა და მისი კონსტრუქტორების სახელები უნდა იყოს დაჯგუფებული, როგორც ეს `Tree(Leaf,Branch)` კონსტრუქციაშია. შემოკლებად შეიძლება `Tree(..)` ჩანაწერის გამოყენება. ასევე შესაძლებელია მონაცემთა კონსტრუქტორების მხოლოდ ერთი ნაწილის ექსპორტი.

`Tree` მოდული ახლა შეიძლება იყოს იმპორტირებული რომელიმე სხვა მოდულში:

```
module Main where

import Tree (Tree(Leaf,Branch), leafList)

...
```

აქ ჩვენ ცხადად მივუთითეთ იმპორტირებად რაობათა (არსთა) სია; თუ ამ სიას არ მივუთითებთ, მაშინ მოდულიდან ექსპორტირებადი ყველა რაობის (არსის) იმპორტი ხდება.

ცხადია, რომ თუ ორ იმპორტირებად მოდულში არის ორი სხვადასხვა რაობა (არსი) ერთი და იმავე სახელით, ჩნდება პრობლემა. ამის თავიდან ასაცილებლად ენაში არსებობს `qualified` საკვანძო სიტყვა. ამ სიტყვის საშუალებით განისაზღვრება ის იმპორტირებადი მოდულები,

რომელთა ობიექტების სახელები იძენს სახეს: «მოდული. ობიექტი». მაგალითად, Tree მოდულისათვის:

```
module Main where

import qualified Tree

leafList = Tree.leafList
```

2. მონაცემის აბსტრაქტული ტიპი

მოდულის გამოყენება, მონაცემის აბსტრაქტული ტიპის განსაზღვრის საშუალებას იძლევა. აბსტრაქტული ტიპის შინაგანი სტრუქტურა დამალულია მისი მომხმარებლისაგან. მაგალითად, განვიხილოთ უმარტივესი ლექსიკონი, რომელიც მოცემული სიტყვით გვიბრუნებს 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
```

getMeaning ფუნქცია მოცემული ლექსიკონითა და სიტყვით გვიბრუნებს ნაპოვნ მნიშვნელობას (Maybe ტიპის გამოყენებით). თვით ლექსიკონი წარმოდგენილია წყვილთა სიით.

როგორ უნდა შეიქმნას ლექსიკონი? ამ მოდულის მომხმარებელს შეუძლია განსაზღვროს `addWord` ფუნქცია, რომელიც ამატებს წყვილს «სიტყვა — მნიშვნელობა» ლექსიკონში და გვიბრუნებს მოდიფიცირებულ ლექსიკონს:

```
import Dictionary

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

აქ მომხმარებელი ხედავს, რომ ლექსიკონი წარმოდგენილია სიის სახით და შეუძლია ისარგებლოს ამით. მაგრამ შემდეგ ჩვენ შეიძლება მოგვინდეს ლექსიკონის სხვაგვარად წარმოდგენა. სია მონაცემთა საკმაოდ არაეფექტური სტრუქტურაა ძებნისათვის, თუ იგი დიდი ხდება. უმჯობესია ძებნის ჰეშ-ცხრილის (`hash` ან `hashed table`) ან ხის გამოყენება. მაგრამ, თუ `Dictionary` ტიპის წარმოდგენა გახსნილია, ჩვენ არ შეგვიძლია მისი შეცვლა, რადგან ჩნდება სამომხმარებლო პროგრამის ფუნქციონირების დარღვევის რისკი.

ვაქციოთ `Dictionary` ტიპი აბსტრაქტულად, რათა დავმალოთ მოდულის მომხმარებლისგან მისი შინაგანი წარმოდგენა. განვსაზღვროთ მოდულში `emptyDict` მნიშვნელობა, რომელიც ასახავს ცარიელ ლექსიკონს, და `addWord` ფუნქცია. მაშინ მომხმარებელი `Dictionary` ტიპის მნიშვნელობებთან ურთიერთობას მხოლოდ ნებადართული ფუნქციების მეშვეობით შეძლებს:

```
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):dict)

emptyDict = Dictionary []
```

მონაცემის აბსტრაქტული ტიპი გვაძლევს ამ მონაცემის დაფარვის (დამალვის) მექანიზმს, რომელსაც ობიექტზე ორიენტირებული დაპროგრამების ენაზე ინკაფსულაცია (ინგლ. encapsulation) ეწოდება.

3. ტიპის სინონიმი

ენაში არის ტიპის სინონიმის (ე.ი. ხშირად გამოყენებადი ტიპისათვის სახელის) განსაზღვრის საშუალება. სინონიმი იქმნება type საკვანძო სიტყვის მეშვეობით. აი რამდენიმე მაგალითი:

```
type String = [Char]

type Person = (Name, Address)

type Name = String

type Address = Maybe String
```

ტიპის სინონიმი არ განსაზღვრავს ახალ ტიპს. იგი მხოლოდ და მხოლოდ ახალ სახელს იძლევა უკვე არსებული ტიპისათვის. მაგალითად, `Person -> Name` ტიპი მთლიანად `(String,Maybe String) -> String` ტიპის ეკვივალენტურია, მაგრამ მას იყენებენ, რადგან, ჯერ ერთი, იგი გვეხმარება უფრო მოკლე სახელი მივცეთ ტიპს და მეორე, ზრდის კოდის გაგების დონეს.

4. შეტანის/გამოტანის ოპერაცია

შეტანის/გამოტანის სისტემა `Haskell` ენაში მთლიანად ფუნქციონალურია, მაგრამ არ ჩამოუვარდება შესაძლებლობებში იმპერატიული ენის შეტანის/გამოტანის სისტემას. იმპერატიულ ენაში პროგრამა არის მოქმედებათა მიმდევრობა, რომლებიც ახორციელებს წაკითხვას და ცვლის გარე სამყაროს შიგთავსს. ტიპური მოქმედებებია გლობალური ცვლადების წაკითხვა და დაყენება, ფაილში ჩაწერა, კლავიატურიდან წაკითხვა და ა.შ. ასეთი მოქმედებები `Haskell` ენის შემადგენელი ნაწილებიცაა, მაგრამ ისინი მკაფიოდაა განცალკევებული ენის ფუნქციონალური ბირთვისგან.

შეტანის/გამოტანის სისტემა `Haskell` ენაში აგებულია მონადათა კონცეფციის გარშემო. მაგრამ შეტანის/გამოტანის დასაპროგრამებლად მონადათა გაგება იმაზე მეტად საჭირო არ არის, ვიდრე ზოგადი ალგებრის გაგება მარტივი არითმეტიკული მოქმედებების შესასრულებლად. ამიტომ შეტანის/გამოტანის სისტემის განხილვა მიბმული არ იქნება მონადაზე, რომელიც დაწვრილებით სალექციო კურსში შეიძლება იყოს განხილული და არა ლაბორატორიული პრაქტიკუმის წიგნში.

`Haskell` ენის საშუალებით მოქმედება კი არ სრულდება, არამედ განისაზღვრება. მოქმედების განსაზღვრა არ ნიშნავს, რომ იგი სრულდება. მოქმედების შესრულება გამოსახულებათა გამოთვლების ფარგლებს გარეთ ხდება.

მოქმედებები ან ატომარულია და განსაზღვრულია სისტემების პრიმიტივებით, ან სხვა მოქმედებათა თანამიმდევრული კომპოზიციებია. შეტანის/გამოტანის მონადა შეიცავს პრიმიტივებს, რომლებიც შედგენილ მოქმედებათა შექმნის საშუალებას იძლევა, როგორც ეს იმპერატიულ ენებში `(;)` წერტილ-მძიმის გამოყენებით ხდება. მონადა თავისებური «წებოს» როლს ასრულებს, რომლითაც მოქმედებები დაკავშირებულია პროგრამაში.

4.1. შეტანის/გამოტანის საბაზო ოპერაცია

ყოველი მოქმედება გვიბრუნებს მნიშვნელობას. ტიპების სისტემაში ეს მნიშვნელობა «მონიშნულია» `IO` ტიპით, რომელიც განასხვავებს მოქმედებას სხვა მნიშვნელობისგან. მაგალითად, განვიხილოთ `getChar` ფუნქცია:

```
getChar :: IO Char
```

`IO Char` ტიპი გვიჩვენებს, რომ `getChar` ფუნქცია გამოძახებისას ასრულებს გარკვეულ მოქმედებას, რომელიც გვიბრუნებს სიმბოლოს. მოქმედებები, რომლებიც არ გვიბრუნებს შედეგს, **`IO ()`** ტიპს იყენებს. **`()`** სიმბოლო ნიშნავს ცარიელ ტიპს (რაც `C` ენის `void` ტიპის მსგავსია). მაგალითად, `putChar` ფუნქცია:

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

იგი იღებს სიმბოლოს და რაიმე საინტერესოს არ გვიბრუნებს.

მოქმედებები ერთმანეთს `>>=` ოპერატორის საშუალებით უკავშირდება. მაგრამ ჩვენ გამოვიყენებთ ე.წ. `do`-ნოტაციას. საკვანძო `do` სიტყვა იწყებს ოპერატორების თანამიმდევრობას, რომლებიც რიგრიგობით სრულდება. ოპერატორი შეიძლება იყოს ან მოქმედება, ან ნიმუში, რომელიც მოქმედების შედეგს `<-` ნიშნით უკავშირდება. `do`-ნოტაცია იყენებს შეწევის გასწორების იმავე წესებს, რასაც `let` ან `where` საკვანძო სიტყვები. აი მარტივი პროგრამა, რომელიც კითხულობს სიმბოლოს და ბეჭდავს მას:

```
main :: IO ()

main = do c <- getChar

          putChar c
```

`main` სახელის გამოყენება აქ შემთხვევითი არ არის: `Main` მოდულის `main` ფუნქცია წარმოადგენს შესვლის წერტილს `Haskell`-ზე დაწერილ პროგრამაში, `C` ენის `main` ფუნქციის მსგავსად. მისი ტიპი `IO ()` უნდა იყოს. წარმოდგენილი პროგრამა ორ მოქმედებას ასრულებს თანამიმდევრობით: კითხულობს სიმბოლოს, აკავშირებს შედეგს `c` ცვლადთან და შემდეგ ბეჭდავს სიმბოლოს.

როგორ დავიბრუნოთ მნიშვნელობა მოქმედებათა თანამიმდევრობიდან? მაგალითად, თქვენთვის აუცილებელია `ready` ფუნქციის განსაზღვრა, რომელიც ახორციელებს სიმბოლოს წაკითხვას და გვიბრუნებს `True`-ს, თუ იგი `'y'`-ს უდრის:

```
ready :: IO Bool

ready = do c <- getChar

           c == 'y' -- შეცდომა!!!
```

ეს პროგრამა არ მუშაობს, ვინაიდან მეორე ოპერატორი do-ში მხოლოდ ბულის მნიშვნელობაა და არა მოქმედების. ჩვენ უნდა ავიღოთ ბულის მნიშვნელობა და შევქმნათ მოქმედება, რომელიც არაფერს აკეთებს, მაგრამ გვიბრუნებს ბულის ამ მნიშვნელობას არგუმენტის სახით. ამას return ფუნქცია ემსახურება:

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

return ფუნქცია ამთავრებს მოქმედებათა თანამიმდევრობას. ამრიგად, ready შემდეგნაირად განისაზღვრება:

```
ready :: IO Bool

ready = do c <- getChar

           return (c == 'y')
```

ახლა შეიძლება შეტანის/გამოტანის უფრო რთული ფუნქციების განსაზღვრა.

getLine ფუნქცია გვიბრუნებს კლავიატურიდან წაკითხულ სტრიქონს, რომლის ბოლოში სტრიქონის დასასრულის სიმბოლოა:

```
getLine :: IO String

getLine = do c <- getChar
```

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

return ფუნქციას შემოაქვს ჩვეულებრივი მნიშვნელობა შეტანის/გამოტანის მოქმედებათა გარემოში. რა ხდება შექცეული მიმართულებით? შესაძლებელია შეტანის/გამოტანის ჩვეულებრივ გამოსახულებაში მოქმედების შესრულება? თურმე, არა! ისეთ ფუნქციას, როგორიც არის f :: Int -> Int, არ შეუძლია შეტანის/გამოტანის ოპერაციათა შესრულება, ვინაიდან IO არ ჩნდება მისი დაბრუნებული მნიშვნელობის ტიპში.

4.2. შეტანის/გამოტანის სტანდარტული ოპერაცია

განვიხილოთ შემდეგი მოქმედებები და ტიპები ფაილების შეტანასთან/გამოტანასთან სამუშაოდ (ისინი განსაზღვრულია IO მოდულში):

```
type FilePath = String –- ფაილების სახელები სისტემაში

openFile :: FilePath -> IOMode -> IO Handle

hClose :: Handle -> IO ()

data IOMode = ReadMode | WriteMode | AppendMode |
ReadWriteMode
```

ფაილის გასახსნელად გამოიყენება openFile ფუნქცია, რომელსაც გადაეცემა ფაილის სახელი და ამ ფაილის გახსნის აუცილებელი რეჟიმი.

ამ დროს იქმნება (`Handle` ტიპის) ფაილის დესკრიპტორი, რომელიც შემდეგ უნდა დაიხუროს `hClose` ფუნქციის საშუალებით.

სიმბოლოსა და სტრიქონის წასაკითხად ფაილიდან შემდეგი ფუნქციები გამოიყენება:

```
hGetChar :: Handle -> IO Char

hGetLine :: Handle -> IO String
```

ფაილში ჩასაწერად მიმართავენ ფუნქციებს:

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

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

კლავიატურიდან წასაკითხად და ეკრანზე გამოსატანად შემდეგი ფუნქციები გამოიყენება:

```
getChar :: IO Char

getLine :: IO String

putChar :: Char -> IO ()

putStr :: String -> IO ()
```

გარდა ამისა, ძალიან სასარგებლოა ასეთი ფუნქცია:

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

იგი მთელ ფაილს ერთ დიდ სტრიქონად კითხულობს. ერთი მხრივ, ეს ფუნქცია ძალიან არაეფექტურია, მაგრამ სინამდვილეში, გადადებული

გამოთვლების გამოყენების გამო, ფაილიდან ხდება სიმბოლოთა მხოლოდ აუცილებელი რაოდენობის წაკითხვა და არც ერთი ზედმეტის.

4.3. მაგალითი

ჩავწეროთ პროგრამა ფაილების პირის გადასაღებად. იგი კითხულობს კლავიატურიდან ორი ფაილის — საწყისისა და მიზნობრივის — სახელებს და ერთი ფაილის პირი გადააქვს მეორეში.

— — ფუნქცია ბეჭდავს მოწვევას, კითხულობს ფაილის სახელს

— — და ხსნის მას მითითებულ რეჟიმში

```
getAndOpenFile prompt mode = do putStr prompt
                                name <- getLine
                                openFile name mode

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

მიუხედავად იმისა, რომ ჩვენ ვიყენებთ `hGetContents` ფუნქციას, ფაილის მთელი შიგთავსი არ იქნება მეხსიერებაში, ვინაიდან ამ შიგთავსის წაკითხვა მოხდება საჭიროების მიხედვით და ჩაიწერება დისკოზე. ეს დიდი ფაილების პირის გადაღების საშუალებასაც კი იძლევა, თუნდაც მათი მოცულობა კომპიუტერის ოპერატიული მეხსიერების მოცულობას აჭარბებდეს. საწყისი ფაილი არაცხადად დაიხურება, როდესაც ამ ფაილიდან უკანასკნელი სიმბოლო აღმოჩნდება წაკითხული.

პროგრამის ბრძანებათა სტრიქონის (`command line`) პარამეტრებში შესაღწევად შეიძლება `System` მოდულში განსაზღვრული შემდეგი ფუნქციის გამოყენება:

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

ეს ფუნქცია გვიბრუნებს იმ სტრიქონების სიას, რომლებიც ბრძანებათა სტრიქონის პარამეტრებია (`argv` მასივის მსგავსად `C`-ზე დაწერილ პროგრამებში). მაშინ პირის გადაღების პროგრამა ასე განისაზღვრება:

```
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 ბრძანებათა სტრიქონიდან. `copyFile` ფუნქცია ბეჭდავს შეტყობინებას შეცდომის შესახებ, თუ პროგრამაში გადაცემულია არგუმენტთა არასწორი რაოდენობა.

5. შესრულებადი პროგრამის შექმნა

აქამდე პროგრამას `Haskell` ენაზე ჩვენ ვქმნიდით ინტერპრეტატორის გამოყენებით. მაგრამ არსებობს შესაძლებლობა შევქმნათ შესრულებადი პროგრამა, რომლისთვისაც ინტერპრეტატორის გარემო საჭირო არ არის. ამისათვის გამოიყენება `Glasgow Haskell Compiler` კომპილატორი, რომელიც `ghc` ბრძანების საშუალებით გამოიძახება.

იმისათვის, რომ მოდულების ნაკრების კომპილირება მოხდეს შესრულებად პროგრამად, უნდა განისაზღვროს მოდული, რომლის სახელია `Main` და რომელშიც აუცილებელია `main :: IO ()` ფუნქციის განსაზღვრა. ეს მოდული უნდა მოთავსდეს `Main.hs` ფაილში. კომპილაციისათვის საჭიროა ბრძანებათა სტრიქონში შემდეგი ინსტრუქციის ჩაწერა:

```
ghc --make Main.hs
```

იმ შემთხვევაში, თუ პროგრამაში არის შეცდომები, ინფორმაცია მათ შესახებ ეკრანზე გამოვა. თუ შეცდომები არ არის, კომპილატორი შექმნის შესრულებად ფაილს და შესაძლებელია მისი ამოქმედება.

6. დავალება

1. დაწერეთ შემდეგი პროგრამები:

    1) პროგრამა, რომელიც კითხულობს ორ რიცხვს და გვიბრუნებს მათ ჯამს.

    2) პროგრამა, რომელიც ბეჭდავს ბრძანებათა სტრიქონის მისთვის გადაცემულ არგუმენტებს.

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

    4) პროგრამა, რომელიც ბრძანებათა სტრიქონში იღებს რიცხვს (ვთქვათ, `m`-ს) და ფაილის სახელს, ხოლო ეკრანზე გამოაქვს ფაილის პირველი `m` სტრიქონი. გამოიყენეთ `lines` ფუნქცია, რომელიც ახდენს სტრიქონის დაყოფას სტრიქონების სიად (სტრიქონის დასასრულის `\n` სიმბოლოს რაოდენობის შესაბამისად). მაგალითად, `lines "line1\nline2\nline3\n"` დაგვიბრუნებს

        `["line1","line2","line3"]`-ს.

ასევე სასარგებლოა `unlines` ფუნქცია, რომელიც შებრუნებულ ოპერაციას ახორციელებს.

სახელდობრ:

```
unlines ["line1","line2","line3"]
```

შემდეგ სტრიქონს გვიბრუნებს:

```
"line1\nline2\nline3\n"
```

2. შექმენით პროგრამები, რომლებითაც წყდება თქვენი ვარიანტის ყველა დავალება პირველი ლაბორატორიული სამუშაოდან. ფუნქციათა პარამეტრების წაკითხვა უნდა ხდებოდეს კლავიატურიდან.

ლიტერატურა

1. Хювёнен Э., Сеппенен И. Мир Lisp’а. В 2-х томах. М.: Мир, 1990.

2. Bird R. Introduction to Functional Programming using Haskell. 2-nd edition, Prentice Hall Press, 1998.

3. Филд А., Харрисон П. Функциональное программирование. М.: Мир, 1993.

4. Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.

5. Fokker J. Functional programming. Dept. of CS. Utrecht University, 1995.

6. Thompson S. Haskell: The Craft of Functional Programming. 2-nd edition, Addison-Wesley, 1999.

7. Norvig P. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Los Altos, CA: Morgan Kaufmann, 1992.

8. Russel S. J., Norvig P. Artificial Intelligence: A Modern Approach. Englewood Cliffs, NJ: Prentice-Hall, 1995.

9. Henson M. Elements of functional languages. Dept. of CS. University of Sassex, 1990.

10. Winston P. H., Horn K. P. LISP, 3-rd edition. Reading, MA: Addison-Wesley, 1988.

11. Graham P. On Lisp: Advanced Techniques for Common Lisp. Englewood Cliffs, NJ: Prentice-Hall, 1994.

12. Charniak E., Reisbeck C., McDermott D. Artificial Intelligence Programming, 2-nd edition. Hillsdale, NJ: Lawrence Eribaum, 1987.

13. Бердж В. Методы рекурсивного программирования. М.: Машиностроение, 1983.

14. Джонс С., Лестер Д. Реализация функциональных языков. М.: Мир, 1991.

15. არჩილ ფრანგიშვილი, ზურაბ წვერაიძე, ოლეგ ნამიჩეიშვილი. დაპროგრამება ჰასკელზე, სახელმძღვანელო.-თბილისი: საგამომცემლო სახლი «ტექნიკური უნივერსიტეტი», 2012.

16. არჩილ ფრანგიშვილი, ზურაბ წვერაიძე, ოლეგ ნამიჩეიშვილი. დაპროგრამება Haskell ენაზე (მულტიმედიური პრეზენტაცია), ელექტრონული სახელმძღვანელო, ინფორმატიკისა და მართვის სისტემების ფაკულტეტი, http://gtu.ge/learningStu/elBooks.php, 278 სლაიდი.

რედაქტორი მ. ბაზაძე

გადაეცა წარმოებას 28.03.2012. ხელმოწერილია დასაბეჭდად 11.05.2012. ქაღალდის ზომა 70x100 1/16. პირობითი ნაბეჭდი თაბახი 9. ტირაჟი 100 ეგზ.

საგამომცემლო სახლი ,,ტექნიკური უნივერსიტეტი'', თბილისი, კოსტავას 77