თბილისის სახელმწიფო უნივერსიტეტი

ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერული მეცნიერებები

ბაკალავრიატი

ფუნქციონალური პროგრამირება Haskell-ზე

Functional Programming using Haskell

O'Sullivan, Stuart, Goerzen. *Real World Haskell*, O'Reilly, 2008. http://book.realworldhaskell.org/read/.

Душкин Р. В. Справочник по языку Haskell. — М.: ДМК-Пресс, 2008. Websites: haskell.org , haskell.ru

საფუძველზე

ლექციათა კურსი

ნათელა არჩვაძე

თბილისი
2012

1. შესავალი

ფუნქციონალური პროგრამირების მიზანია თითოეულ პროგრამას მისცეს მარტივი მათემატიკური ინტერპრეტაცია. ეს ინტერპრეტაცია უნდა იყოს შესრულების დეტალებისგან დამოუკიდებელი და ამავე დროს გასაგები მათთვის, ვისაც არ აქვს სამეცნიერო ხარისხი საგნობრივ არეში.

ლორენს პაულსონი

ფუნქციონალური პროგრამირების აღწერამდე გავიხსენოთ ზოგადად პროგრამირების განვითარების ისტორია.

XX საუკუნის 40–იან წლებში გაჩნდა პირველი ციფრული კომპიუტერები, რომლებიც, როგორც ცნობილია, პროგრამირდებოდა სხვადასხვა რიგის ტუმბლერების, გაყვანილობისა და ღილაკების საშუალებით. ასეთი გადამრთველობის რაოდენობა იყო რამდენიმე ასეული და მკეთრად იზრდებოდა პროგრამის სირთულესთან ერთად. ამიტომ პროგრამირების განვითარების შემდეგი ეტაპი გახდა ასემბლერის ენების შექმნა მარტივი მნემონიკით.

ვერც ასემბლერები გახდა ის ინსტრუმენტები, რომლებსაც ჩვეულებრივი ხალხი გამოიყენებდა. მნემოკოდები ჯერ კიდევ რჩებოდა ძალზე რთული, მით უმეტეს, რომ ყოველი ასემბლერი მყარად იყო მიბმული იმ არქიტექტურასთან, რომელზედაც ის სრულდებოდა.

მორიგი ნაბიჯი ასემბლერის შემდეგ გახდა ეგრეთ წოდებული მაღალი დონის იმპერატიული ენები (BASIC, Pascal, C, Ada და სხვა). ამ ენებს იმპერატიული ეწოდა იმ მარტივი მიზეზის გამო, რომ მათი მთავარი თვისება არის ორიენტირებულობა ინსტრუქციების თანმიმდევრულ შესრულებაზე, რომლებიც ოპერირებენ მეხსიერებაზე (ანუ მინიჭებები) და იტერაციული ციკლები. ფუნქციებისა და პროცედურების გამოძახებას ვერ გამოჰყავდა ასეთი ენები ცხადი იმპერატიულობიდან.

დავუბრუნდეთ ფუნქციონალურ პროგრამირებას... ფუნქციონალური პარადიგმის ქვაკუთხედს წარმოადგენს ფუნქცია. თუ ჩვენ გავიხსენებთ მათემატიკის ისტორიას, შეიძლება შევაფასოთ ცნება „ფუნქციის“ ასაკი. იგი უკვე ოთხასი წლისაა. მათემატიკამ ფუნქციებთან ოპერირებისთვის მოიგონა უამრავი რაოდენობა თეორიული და პრაქტიკული აპარატი დაწყებული ჩვეულებრივი დიფერენცირებიდან და ინტეგრირებიდან, დამთავრებული ფუნქციონალური ანალიზით, არამკაფიო სიმრავლეთა თეორიითა და კომპლექსური ცვლადის ფუნქციებით.

მათემატიკური ფუნქციები გამოხატავს კავშირს პროცესის პარამეტრებსა (შესასვლელი) და შედეგს (გამოსასვლელი) შორის. რადგან გამოთვლა ასევე

პროცესია, რომელსაც აქვს შესასვლელი და გამოსასვლელი, ფუნქცია სავსებით შესაფერისი და ადექვატური საშუალებაა გამოთვლების აღსაწერად. სწორედ ეს მარტივი პრინციპი არის ჩადებული ფუნქციონალური პარადიგმისა საფუძველში და ფუნქციონალური სტილით პროგრამირებაში. ფუნქციონალური პროგრამა წარმოადგენს ფუნქციების განსაზღვრებების ერთობლიობას. ფუნქცია განისაზღვრება სხვა ფუნქციებით, ან რეკურსიულად – თავისი თავით. პროგრამის შესრულების მომენტში ფუნქცია იღებს პარამეტრებს, ითვლის და აბრუნებს შედეგს, საჭიროების შემთხვევაში, ითვლის სხვა ფუნქციის მნიშვნელობებსაც. ფუნქციონალურ ენაზე პროგრამირებისას პროგრამისტი არ აღწერს გამოთვლების თანმიმდევრობას. მისთვის აუცილებელია მხოლოდ აღწეროს სასურველი შედეგი ფუნქციების სისტემის სახით.

ავღნიშნოთ, რონ ფუნქციონალური პროგრამირებამ, ისევე როგორც ლოგიკურმა პროგრამირებამ დიდი გამოყენება ჰპოვა ხელოვნურ ინტელექტსა და მის დანართებში. გავეცნოთ ფუნქციონალური პროგრამირების ისტორიას და ამოცანებს.

ფუნქციონალური პროგრამირების ისტორია

როგორც ცნობილია, იმპერატიულ პროგრამირებას თეორიული საფუძველი ჩაეყარა ჯერ კიდევ XX საუკუნის 30–იან წლებში ალან ტიურინგისა და ჯონ ფონ ნეიმანის მიერ. თეორია, რომელიც ფუნქციონალური მიდგომის საფუძველია, ასევე დაიბადა 20–იან – 30–იან წლებში. მათ შორის, ვინც შეიმუშავა ფუნქციონალური პროგრამირების მათემატიკური საფუძვლები, შეიძლება დავასახელოთ მოზეს შენფინკელი (გერმანია და რუსეთი) და ჰასკელ ლარი (ინგლისი), რომელმაც კომბინატორული ლოგიკა დაამუშავა, ასევე ალონზო ჩერჩი (აშშ), რომელმაც შექმნა λ აღრიცხვა.

თეორია რჩებოდა თეორიად, სანამ წინა საუკუნის 50–იანი წლების დასაწყისში ჯონ მაკარტმა არ შეიმუშავა ენა Lisp, რომელიც გახდა პირველი ფუნქციონალური ენა და წლების განმავლობაში რჩებოდა ერთადერთად. ენა Lisp დღემდე გამოიყენება (მაგალითად, FORTRAN–ის მსგავსად), თუმცა ველარ აკმაყოფილებს ზოგიერთ თანამედროვე მოთხოვნას, რაც აიძულებს პროგრამების შემქმნელებს დიდი ძალისხმევა გადაიტანონ კომპილერზე. ამის აუცილებლობას იწვევს სულ უფრო მზარდი სირთულის პროგრამული უზრუნველყოფა.

ამ გარემოების გამო დიდ როლს თამაშობს ტიპიზაცია. XX საუკუნის 70–იანი წლების ბოლოსა და 80–იანი წლების დასაწყისში ინტენსიურად მუშავდებოდა ფუნქციონალური პროგრამირების შესაბამისი ტიპიზაციის მოდელები. მათი უმრავლესობა შეიცავს ისეთ ძლერ მექანიზმებს, როგორიცაა მონაცემთა აბსტრაქცია და პოლომორფიზმი. გაჩნდა მთელი რიგი ტიპიზირებული ფუნქციონალური ენები:

ML, Scheme, Hope, Miranda, Clean და ბევრი სხვა. დამატებით, მუდმივად იზრდებოდა დიალექტების რაოდენობაც.

და დადგა სიტუაცია, რომ პრაქტიკულად ყველა ჯგუფი, ვინც ფუნქციონალურ პროგრამირებაში მუშაობდა, იყენებდა საკუთარ ენას. ამან ხელი შეუშალა ასეთი ენების შემდგომ გავრცელებას და გააჩინა მთელი რიგი პრობლემები. ფუნქციონალური პროგრამირების სფეროში წამყვანი მკვლევარების გაერთიანებულმა ჯგუფმა სიტუაციის გამოსწორების მიზნით გადაწყვიტა სხვადასხვა ენის ღირსებები გამოეყენებინათ ახალი უნივერსალური ფუნქციონალური ენის შექმნისას. ასეთი ენის პირველი რეალიზაცია, რომელსაც დაერქვა Haskell ჰასკელ კარის საპატივსაცემოდ, შეიქმნა 90–იანი წლების დასაწყისში. ამჟამად ფუნქციონირებს სტანდარტი Haskell-98.

ფუნქციონალური ენების უმრავლესობისთვის, Lisp–ის ტრადიციებიდან გამომდინარე, რეალიზებულია ინტერპრეტატორი. ინტერპრეტორები მოსახერხებელია პროგრამის სწრაფი გამართვისთვის. ამ დროს კომპილაციის გრძელი პროცესი გამოიტოვება, რითაც ჩქარდება დამუშავების ჩვეულებრივი ციკლი. თუმცა, მეორეს მხრივ, ინტერპრეტატორები, კომპილერებთან შედარებით რამდენჯერმე აგებენ კოდის შესრულების სიჩქარეში. ამიტომაც, ინტერპრეტატორის გვერდით არსებობს კომპილერები, რომლებიც გენერირებენ მანქანურ კოდს (მაგალითად, Objective Caml) ანდა C/C++ კოდს (მაგალითად, Glasgow Haskell Compiler). ნიშანდობლივია ის, რომ პრაქტიკულად ყველა კომპილერი რეალიზებულია თვით ამ ენაზე.

ავღნიშნოთ, რომ ქვემოთ მოყვანილ ფუნქციონალური პროგრამირების მაგალითებში გამოვიყენებთ ან აბსტრაქტული ფუნქციონალური ენას, რომელიც ახლოა მათემატიკურ ნოტაციასთან, ან Haskell-ს, რომლის უფასო კომპილერები შეიძლება გადმოწერილი იყოს გვერდიდან www.haskell.org.

ფუნქციონალური ენების თვისებები

შეიძლება მოკლედ ჩამოვთვალოთ ფუნქციონალური ენების ძირითადი თვისებები:

- მოკლე და მარტივი კოდი;
- მკაცრი ტიპიზაცია;
- მოდულირება;
- ფუნქცია – ეს მნიშვნელობაა;
- სისუფთავე (გვერდითი ეფექტების არარსებობა);
- გადატანილი (ზარმაცი) გამოთვლები.

მოკლე და მარტივი კოდი

პროგრამა ფუნქციონალურ ენაზე საზოგადოდ უფრო მოკლეა და მარტივი, ვიდრე იგივე პროგრამა იმპერატიულ ენაზე. შევადაროთ პროგრამები C–ზე და აბსტრაქტულ ფუნქციონალურ ენაზე ჰოარეს სწრაფი დახარისხების მაგალითზე. (ეს მაგალითი გახდა კლასიკური მაგალითი ფუნქციონალური ენების უპირატესობების საჩვენებლად).

მაგალითი 1. ჰოარეს სწრაფი დახარისხება C–ზე.

```
void quickSort (int a[], int l, int r)
{
      int i = l;
      int j = r;
      int x = a[(l + r) / 2];
      do
      {
            while (a[i] < x) i++;
            while (x < a[j]) j--;
            if (i <= j)
            {
                  int temp = a[i];
                  a[i++] = a[j];
                  a[j--] = temp;
            }
      }
      while (i <= j);
      if (l < j) quickSort (a, l, j);
      if (i < r) quickSort (a, i, r);
}
```

მაგალითი 2. ჰოარეს სწრაფი დახარისხება აბსტრაქტულ ფუნქციონალურ ენაზე.

```
quickSort ([]) = []
quickSort ([h : t]) = quickSort (n | n   t, n <= h) + [h] +
quickSort (n | n   t, n > h)
```

მაგალითი 2 შეიძლება წავიკითხოთ ასე:

1. თუ სია ცარიელია, მაშინ შედეგიც იქნება ცარიელი სია.

2. წინააღმდეგ შეთხვევაში (ანუ სია როცა არ არის ცარიელი) გამოიყოფა თავი (პირველი ელემენტი) და კუდი (დარჩენილი ელემენტების სია, რომელიც შეიძლება იყოს ცარიელი). ამ შემთხვევაში შედეგი იქნება კონკატენაცია კუდის ყველა ელემენტისა, რომელიც ნაკლებია ან ტოლი თავის, სიასთან, რომელიც შედგება თავისა და კუდის ყველა ელემენტისგან, რომელიც მეტია თავზე.

მაგალითი 3. ჰოარეს სწრაფი დახარისხება ენა Haskell–ზე.

```
quickSort [] = []
quickSort (h : t) = quickSort [y | y <- t, y < h] ++ [h] ++
quickSort [y | y <- t, y >= h]
```

ამ მარტივ მაგალითზეც ჩანს, თუ როგორ იგებს ფუნქციონალური პროგრამირების სტილი როგორც კოდის რაოდენობაში, ასევე მის ელეგანტურობაში.

ამას გარდა, ყველა ოპერაცია მეხსიერებასთან სრულდება ავტომატურად. ნებისმიერი ობიექტის შექმნისას მას ავტომატურად გამოეყოფა მეხსიერება. მას შემდეგ, რაც ობიექტი თავის დანიშნულებას შეასრულებს, ის ავტომატურადვე განადგურდება დამლაგებლის მიერ, რომელიც არის ნებისმიერი ფუნქციონალური ენის ნაწილი.

კიდევ ერთი სასარგებლო თვისება, რომელიც იძლევა პროგრამის შემცირების საშუალებას, არის ნიმუშთან შედარების მექანიზმი. ეს იძლევა საშუალებას აღიწეროს ფუნქცია, როგორც ინდუციური განსაზღვრება. მაგალითად:

მაგალითი 4. ფიბონაჩის N–ური რიცხვის განსაზღვრა.

```
fibb (0) = 1
fibb (1) = 1
fibb (N) = fibb (N – 2) + fibb (N – 1)
```

როგორც ჩანს, ფუნქციონალური ენები ადის უფრო მაღალ აბსტრაქტულ დონეზე, ვიდრე ტრადიციული იმპერატიული ენები. ნიმუშთან შედარების მექანიზმს განვიხილავთ შემდგომში.

მკაცრი ტიპიზაცია

პრაქტიკულად ყველა თანამედროვე პროგრამირების ენა წარმოადგენს მკაცრად ტიპიზირებულ ენას (შესაძლებელა JavaScript–ისა და მისი დიალექტების გამოკლებით. არ არსებობს იმპერატიული ენა, რომელშიც არ იყოს ცნება „ტიპი“). მკაცრი ტიპიზაცია უზრუნველყოფს უსაფრთხოებას. პროგრამა, რომელიც ამოწმებს

ტიპებს, არ შეწყდება ოპერციული სისტემის შეტყობინებით "access violation". ეს განსაკუთრებით ეხება ისეთ ენებს, როგორიცაა C/C++ და Object Pascal, სადაც მიმთითებლების გამოყენება ხშირად ხდება. ფუნქციონალურ ენებში შეცდომების დიდი ნაწილის გასწორება ხდება კომპილაციის ეტაპზე, ამიტომ გამართვის სტადია და პროგრამის დამუშავების მთლიანი დრო მცირდება. და კიდევ, მკაცრი ტიპიზაცია კომპილერს აძლევს უფრო ეფექტური კოდის გენერირების საშუალებას და ამით აჩქარებს პროგრამის შესრულების დროს.

თუ განვიხილავთ ჰოარეს სწრაფი დახარისხების მაგალითს, შეიძლება დავინახოთ, რომ უკვე ნახსენებ განსხვავებების გარდა C ენაზე ვარიანტსა და აბსტრაქტულ ფუნქციონალურ ენაზე ვარიანტს შორის, არის კიდევ ერთი ძირითადი განსხვავება: ფუნქცია C–ზე ახდენს int ტიპის (მთელი რიცხვების) დახარისხებას, ხოლო აბსტრაქტულ ფუნქციონალურ ენაზე – ნებისმიერი ტიპის მნიშვნელობების სიის, რომელიც ეკუთვნის დალაგებული სიდიდეების კლასს. ამიტომ, ბოლო ფუნქციას შეუძლია დაახარისხოს მთელი რიცხვების სია, ასევე ნამდვილი რიცხვების სია და სტრიქონების სია. შეიძლება ავღწეროთ ახალი ტიპი. მისთვის განვსაზღვროთ შედარების ოპერაცია და შემდეგ გამოვიყენოთ ხელახალი კომპილაციის გარეშე quickSort ამ ახალი ტიპის მნიშვნელობების სიისთვისაც. ამ სასარგებლო თვისებას ეწოდება პარამეტრული ანუ ჭეშმარიტი პოლიმორფიზმი და მას მხარს უჭერს ფუნქციონალური ენების უმრავლესობა.

პოლიმორფიზმის კიდევ ერთი სახეობაა ფუნქციების გადატვირთვა, რომელიც სხვადასხვა ფუნქციებს, მაგრამ რაღაცით მსგავსს, აძლევს ერთიდაიგივე სახელებს. გადატვირთული ოპერაციის ტიპიური მაგალითია შეკრების ოპერაცია. მთელი რიცხვებისა და ნამდვილი რიცხვების შეკრების ფუნქციები სხვადასხვაა, მაგრამ მოხერხებულობისთვის ისინი ატარებენ ერთიდაიგივე სახელს. ზოგიერთი ფუნქციონალური ენა, პოლიმორფიზმის გარდა მხარს უჭერს ოპერაციების გადატვირთვასაც.

ენა C++ არის ისეთი ცნება, როგორიცაა შაბლონი, რომელიც იძლევა საშუალებას განისაზღვროს პოლომორფული ფუნქციები, მსგავსი quickSort–ის. C++–ის სტანდარტულ ბიბლიოთეკაში STL შედის ასეთი ფუნქცია და კიდევ მრავალი სხვა პოლიმორფული ფუნქცია. მაგრამ C++–ის შაბლონები და Ada–ს გვაროვნული ფუნქციები, ბადებს გადატვირთული ფუნქციების სიმრავლეს, რომლებსაც კომპილერი ყოველ ჯერზე აკომპილირებს, რაც უარყოფითად მოქმედებს კომპილაციის დროსა და კოდის ზომაზე. ხოლო ფუნქციონალურ ენებში პოლიმორფული ფუნქცია quickSort – ეს ერთი, ერთადერთი ფუნქციაა.

ზოგიერთი ენაში, მაგალითად, Ada–ში მკაცრი ტიპიზაცია ითხოვს პროგრამისტისგან ცხადად აღწეროს ყველა მნიშვნელობის და ფუნქციის ტიპი. ამის თავიდან ასაცილებლად, მკაცრად ტიპიზირებულ ფუნქციონალურ ენებში ჩადგმულია სპეციალური მექანიზმი, რომელიც კომპილერს საშუალებას აძლევს განსაზღვროს კონსტანტის, გამოსახულების და ფუნქციის ტიპი კონტექსტიდან

გამომდინარე. ამ მექანიზმს უწოდებენ ტიპების გამოყვანის მექანიზმს. ცნობილია რამდენიმე ასეთი მექანიზმი, თუმცა მათი უმრავლესაბა წარმოადგენს სახესხვაობებს ჰინდლი–მილნერის ტიპიზაციის მოდელისა, რომელიც XX საუკუნის 80–იანი წლების დასაწყისში დამუშავდა. ამრიგად, უმრავლეს შემთხვევებში შეიძლება არ მივუთითოთ ფუნქციის ტიპი.

მოდულირება

მოდულირების მექანიზმი საშუალებას იძლევა პროგრამა დავყოთ რამდენიმე დამოუკიდებელ ნაწილად (მოდულად) მათ შორის მკვეთრად განსაზღვრული კავშირებით. ამით მარტივდება დიდი პროგრამული სისტემების პროექტირებისა და შემდგომი მხარდაჭერის პროცესები. მოდულირების მხარდაჭერა არ წარმოადგენს კონკრეტულად ფუნქციონალური ენების თვისებას, თუმცა მას მხარს უჭერს ფუნქციონალური ენების უმრავლესობა. არსებობს ძალზე განვითარებული მოდულური იმპერატიული ენები. ასეთი ენებია, მაგალითად, Modula-2 და Ada-95.

ფუნქცია – ეს მნიშვნელობაა

ფუნქციონალურ ენებში (ისევე, როგორც, საზოგადოდ, პროგრამირებასა და მათემატიკაში) ფუნქციები შეიძლება გადაეცეს სხვა ფუნქციებს არგუმენტად ან დაბრუნდეს როგორც შედეგი. ფუნქციებს, რომლებიც ფუნქციონალურ არგუმენტებს იღებს, უწოდებენ მაღალი რიგის ფუნქციებს ანუ ფუნქციონალებს. ყველაზე ცნობილი ფუნქციონალი არის ფუნქცია map. ეს ფუნქცია იყენებს მოცემულ ფუნქციას სიის ყველა ელემენტთან და აფორმირებს შედეგად სხვა სიას. მაგალითად, განვსაზღვროთ ფუნქცია, რომელსაც აჰყავს მთელი რიცხვი კვადრატში, ასე:

```
square (N) = N * N
```

შეიძლება გამოვიყენოთ ფუნქცია map ნებისმიერი სიის ყველა ელემენტის კვადრატში ასაყვანად:

```
squareList = map (square, [1, 2, 3, 4])
```

ამ ინსტრუქციის შესრულების შედეგი იქნება სია [1, 4, 9, 16].

```
squareList x = map square x
```

სისუფთავე (გვერდითი ეფექტების არ არსებობა)

იმპერატიულ ენებში ფუნქციამ მისი შესრულების პროცესში, შეიძლება წაიკითხოს ან შეცვალოს გლობალური ცვლადების მნიშვნელობები და შეასრულოს შეტანა–გამოტანის ოპერაციები. ამიტომ, თუ გამოვიძახებთ ერთიდაიგივე ფუნქციას ორჯერ ერთიდაიგივე არგუმენტებით, შეიძლება მოხდეს, რომ შედეგად გამოითვალოს ორი სხვადასხვა მნიშვნელობა. ასეთ ფუნქციას უწოდებენ ფუნქციას გვერდითი ეფექტებით.

ფუნქციის აღწერა გვერდითი ეფექტების გარეშე პრაქტიკულად შესაძლებელია ყველა ენაში, მაგრამ ზოგიერთი ენა მხარს უჭერს, ითხოვს გვერდით ეფექტებს. მაგალითად, მრავალ ობიექტ–ორიენტირებულ ენაში კლასის წევრ ფუნქციას გადაეცემა ფარული არგუმენტი (ხშირად მას უწოდებენ `this` ან `self`), რომელსაც ეს ფუნქცია არაცხადად მოდიფიცირებს.

წმინდა ფუნქციონალურ ენაში მინიჭების ოპერატორი არ არსებობს. ობიექტები არ შეიძლება შეიცვალოს და განადგურდეს, შესაძლოა მხოლოდ ახალი შეიქმნას არსებულების დეკომპოზიციითა და სინთეზით. არასაჭირო ობიექტებზე ზრუნავს ენაში ჩადგმული დამლაგებელი. ამის წყალობით წმინდა ფუნქციონალურ ენებში ყველა ფუნქცია თავისუფალია გვერდითი ეფექტებისგან. თუმცა, ეს არ უშლის ხელს მოხდეს ამ ენებში ზოგიერთი სასარგებლო იმპერატიული თვისების იმიტირება, ისეთის, როგორიცაა გამონაკლისი სიტუაცია და მასივები. ამისთვის არსებობს სპეციალური მეთოდები.

რა უპირატესობა აქვთ წმინდა ფუნქციონალურ ენებს? გარდა პროგრამების გამარტივებული ანალიზისა, არსებობს კიდევ ერთი ძლიერი უპირატესობა – პარალელიზმი. ვინაიდან ფუნქცია გამოთვლისას იყენებს მხოლოდ თავის პარამეტრებს, ჩვენ შეგვიძლია გამოვთვალოთ დამოუკიდებელი ფუნქციები ნებისმიერი რიგით ან პარალელურად, ეს შედეგზე ასახვას ვერ ჰპოვებს. ამასთან, პარალელიზმი შეიძლება განხორციელდეს არა მხოლოდ ენის კომპილატორის დონეზე, არამედ არქიტექტურის დონეზეც. ზოგიერთ ლაბორატორიაში უკვე შემუშავებულია და გამოიყენება ექსპერიმენტალური კომპიუტერები, რომლებიც მსგავს არქიტექტურას ეყრდნობა. მაგალითისთვის შეიძლება დავასახელოთ `Lisp`-მანქანა.

გადატანილი გამოთვლები

ტრადიციულ პროგრამიების ენებში (მაგალითად,`C++`–ში) ფუნქციის გამოძახება იწვევს ყველა არგუმენტის გამოთვლას. ფუნქციის გამოძახების ამ მეთოდს უწოდებენ გამოძახებას–მნიშვნელობით. თუ ფუნქციაში რომელიღაც არგუმენტი არ

გამოიყენება, მაშინ გამოთვლის შედეგი იკარგება. აქედან გამომდინარე, გამოთვლები ამაოდ ჩატარდა. რაღაც აზრით, მნიშვნელობით გამოძახების საწინააღმდეგოა გამოძახება საჭიროების მიხედვით. ამ შემთხვევაში არგუმენტი გამოიძახება, თუ საჭიროა შედეგის გამოთვლისათვის. ასეთი გამოთვლების მაგალითად შეიძლება დავასახელოთ კონიუქციის ოპერატორი (&&) C++–დან, რომელიც არ ითვლის მეორე არგუმენტს, თუ პირველ არგუმენტს აქვს მცდარი მნიშვნელობა.

თუ ფუნქციონალური ენა მხარს არ უჭერს გადატანილ გამოთვლებს, მას უწოდებენ მკაცრ ენას. მართლაც, ასეთ ენებში გამოთვლების რიგი მკაცრად არის განსაზღვრული. მკაცრი ენების მაგალითად შეიძლება დავასახელოთ Scheme, Standard ML და Caml.

ენებს, რომლებიც იყენებენ გადატანილ გამოთვლებს, უწოდებენ არამკაცრს. Haskell — არამკაცრი ენაა, ისევე, როგორც Gofer და Miranda. არამკაცრი ენები ამასთანავე არის სუფთა.

ძალზე ხშირად მკაცრი ენები შეიცავს ზოგიერთი ისეთი შესაძლებლობების მხარდაჭერას, რაც ახასიათებს არამკაცრ ენებს. მაგალითად, უსასრულო სიებს. Standard ML–ში სპეციალური მოდულია, რომელიც გადატანილ გამოთვლებს უჭერს მხარს. ხოლო Objective Caml, ამის გარდა, შეიცავს დარეზერვირებულ სიტყვა lazy და კონსტრუქციას მნიშვნელობათა სიისთვის, რომელიც გამოითვლება აუცილებლობის მიხედვით.

ამოსახსნელი ამოცანები

ფუნქციონალური პროგრამირების კურსებში, ტრადიციულად განხილული ამოცანებიდან, შეიძლება გამოვყოთ შემდეგი:

1. დარჩენილი პროცედურის მიღება

თუ მოცემულია შემდეგი ობიექტები:

P (x_1, x_2, ..., x_n) — რაღაც პროცედურა.

$x_1 = a_1$, $x_2 = a_2$ — პარამეტრების ცნობილი მნიშვნელობები.

x_3, ..., x_n — პარამეტრების უცნობი მნიშვნელობები.

მოითხოვება დარჩენილი პროცედურის მიღება P1 (x_3, ..., x_n). ეს ამოცანა იხსნება პროგრამების მხოლოდ ვიწრო კლასისთვის.

2. ფუნქციის მათემატიკური აღწერის მიღება

ვთქვათ, გვაქვს P პროგრამა. მისთვის განსაზღვრულია შესასვლელი მნიშვნელობები და გამოსასვლელი მნიშვნელობები. მოითხოვება აიგოს ფუნქციის მათემატიკური აღწერა

$f : Dx_1, \ldots, Dx_n \rightarrow Dy_1, \ldots, Dy_m.$

3. პროგრამირების ენის სემანტიკის ფორმალური აღწერა.
4. მონაცემთა დინამიური სტრუქტურების აღწერა.
5. პროგრამის „მნიშვნელოვანი“ ნაწილის აგება მონაცემთა სტრუქტურის აღწერით, რომლებსაც ამუშავებს ასაგები პროგრამა.
6. პროგრამის ზოგიერთი თვისების არსებობის დამტკიცება.
7. პროგრამების ექვივალენტური ტრანსფორმაცია.

ყველა ეს ამოცანა საკმაოდ მარტივად იხსნება ფუნქციონალური პროგრამირების საშუალებებით, მაგრამ პრაქტიკულად გადაუწყვეტია იმპერატიულ ენებზე.

საცნობარო მასალა

ფუნქციონალური პროგრამირების ენები

მოვიყვანოთ ზოგიერთი ფუნქციონალური ენის მოკლე აღწერა. დამატებითი ინფორმაცია იხილეთ ქვემოთ ჩამოთვლილ რესურსებში.

Lisp (List processor). ითვლება, რომ არის პირველი ფუნქციონალური ენა. არატიპიზირებულია. შეიცავს იმპერატიულ თვისებებსაც, თუმცა საზოგადოდ წაახალისებს ფუნქციონალური პროგრამირების სტილს. გამოთვლებისას გამოიყენება გამოძახება მნიშვნელობით. არსებობს ენის ობიექტ–ორიენტირებული დიალექტი CLOS.

ISWIM (If you See What I Mean). ფუნქციონალური ენა–პროტოტიპი. დამუშავებულია ლანდიმის მიერ XX საუკუნის 60-იან წლებში იმის სადემონსტრაციოდ, თუ როგორი შეიძლება იყოს ფუნქციონალური ენა. ენასთან ერთად ლანდინმა შექმნა სპეციალური ვირტუალური მანქანა, რომელიც ასრულებდა პროგრამებს ISWIM–ზე. ამ ვირტუალურმა მანქანამ, რომელიც მუშაობდა მნიშვნელობით გამოძახებაზე, მიიღო სახელწოდება SECD-მანქანისა. ISWIM ენის სინტაქსს იყენებს მრავალი ფუნქციონალური ენა. ISWIM სინტაქს ჰგავს ML–ის სინტაქსი, განსაკუთრებით Caml–ის სინტაქსი.

Scheme. Lisp–ის დიალექტი, რომელიც დანიშნულია სამეცნიერო კვლევებისთვის computer science დარგში. Scheme–ის შექმნისას ყურადღება გამახვილდა ელეგენტურობასა და სიმარტივეზე. ამის გამო ენა უფრო პატარა გამოვიდა, ვიდრე Common Lisp.

ML (Meta Language). მკაცრი ენების ოჯახი განვითარებული ტიპების პოლიმორფული სისტემით და პარამეტრიზირებული მოდულებით. ML ისწავლება დასავლეთის მრავალ უნივერსიტეტში (ზოგან, პროგრამირების პირველ ენადაც კი).

Standard ML. ერთ–ერთი პირველი ტიპიზირებული ფუნქციონალური პროგრამირების ენაა. შეიცავს ზოგიერთ იმპერატიულ თვისებას, ისეთს, როგორიცაა მიმთითებლები (გამოიყენება მნიშვნელობების შესაცვლელად) და ამიტომაც, არ არის სუფთა ენა. ძალზე საინტერესოდ არის რეალიზებული მოდულურობა. გამოთვლებისას იყენებს გამოძახებას მნიშვნელობით. აქვს ტიპების ძლიერი პოლიმორფული სისტემა. ენის ბოლო სტანდარტია Standard ML-97, რომლისთვისაც არსებობს სინტაქსის, ასევე ენის სტატიკური და დიმანიკური სემანტიკის ფორმალური მათემატიკური აღწერები.

Caml Light და **Objective Caml.** როგორც Standard ML მიეკუთვნება ML ოჯახს. Objective Caml განსხვავდება Caml Light–გან იმით, რომ მხარს უჭერს კლასიკურ ობიექტ–ორიენტირებულ პროგრამირებას. როგორც Standard ML, ისიც მკაცრია, მაგრამ აქვს ჩადგმული მექანიზმი გადატანილი გამოთვლებისთვის.

Miranda. შემუშავებულია დევიდ ტერნეტის მიერ, როგორც სტანდარტული ფუნქციონალური ენა, რომელიც იყენებს გადატანილ გამოთვლებს. აქვს მკაცრი ტიპების პოლიმორფული სისტემა. მსგავსად ML-ისა, ისწავლება მრავალ უნივერსიტეტში. დიდი ზეგავლენა იქონია Haskell ენის მკვლევარებზე.

Haskell. ერთ–ერთი ყველაზე გავრცელებული არამკაცრი ენა. აქვს ტიპიზაციის ძალზე განვითარებული სისტემა. უფრო ცუდად მუშაობს მოდულებთან. ენის ბოლო სტანდარტია — Haskell 98.

Gofer (GOod For Equational Reasoning). Haskell–ის გამარტივებული დიალექტი. გამიზნულია ფუნქციონალური პროგრამირების შესასწავლად.

Clean. სპეციალურად დანიშნულია პარალელური და დანაწილებული პროგრამირებისთვის. სინტაქსით მიაგავს Haskell–ს. სუფთაა. იყენებს გადატანილ გამოთვლებს. კომპილატორთან ერთად მოყვება ბიბლიოთეკები (I/O libraries), რომელიც იძლევა საშუალებას დაპროგრამდეს გრაფიკული ინტერფეისის Win32–თვის ან MacOS–თვის.

Internet-რესურსები ფუნქციონალურ პროგრამირებაში

www.haskell.org — საიტი, რომელიც ეხება ფუნქციონალურ პროგრამირებას ზოგადად და ენა Haskell – კონკრეტულად. შეიცავს სხვადასხვა საცნობარო

ინფორმაციას, ინტერპრეტატორების სიას და Haskell–ის კომპილერს (ამჟამად, ყველა ინტერპრეტატორი და კომპილერი არის უფასო). ამას გარდა, საიტზე ნახავთ უამრავი რესურსის მისამართს ფუნქციონალური პროგრამირების თეორიისა და სხვა ენების შესახებ (Standard ML, Clean).

cm.bell-labs.com/cm/cs/what/smlnj – Standard ML of New Jersey. ძალზე კარგი კომპილერია. უფასო დისტრიბუციაში კომპილერის გარდა ასევე შედის MLYacc და MLLex უტილიტები და ბიბლიოთეკა Standard ML Basis Library. შეიძლება გაეცნოთ კომპილერისა და ბიბლიოთეკის დოკუმენტაციასაც.

www.harlequin.com/products/ads/ml/–Harlequin MLWorks, კომერციული კომპილერი Standard ML–ის. თუმცა, არაკომერციული მიზნით შესაძლოა ისარგებლოთ უფასო, ცოტათი შეზღუდული ვერსიით.

caml.inria.fr – ინსტიტუტი INRIA. საშინაო საიტი ენების Caml Light–ის და Objective Caml–ის მკვლევართა ჯგუფის. შესაძლოა უფასოდ გადმოქაჩოთ Objective Caml, რომელიც შეიცავს ინტერპრეტატორს, კომპილერს ბაიტ–კოდში და მანქანურ კოდში, Yacc და Lex Caml–ისთვის, გამმართველს და პროფაილერს, დოკუმენტაციას, მაგალითებს. კომპილირებული კოდის ხარისხი ამ კომპილერს აქვს ძალზე კარგი, სიჩქარით სჯობნის Standard ML–აც კი New Jersey–დან.

www.cs.kun.nl/~clean/ – შეიცავს ენა Clean–ის კომპილატორის დისტრიბუციას. ეს კომპილერი არის კომერციული, მაგრამ უშვებს უფასო გამოყენებას არაკომერციული მიზნებისთვის. იქიდან გამომდინარე, რომ კომპილერი არის ფასიანი, გამომდინარეობს მისი ხარისხი (ძალზე სწრაფია), აქვს დამუშავების გარემო, მოყვება კარგი დოკუმენტაცია და სტანდარტული ბიბლიოთეკები.

ლიტერატურა

1. Хювёнен Э., Сеппенен И. Мир Lisp'a. В 2-х томах. М.: Мир, 1990.
2. Бердж В. Методы рекурсивного программирования. М.: Машиностроение, 1983.
3. Филд А., Харрисон П. Функциональное программирование. М.: Мир, 1993.
4. Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.
5. Джонс С., Лестер Д. Реализация функциональных языков. М.: Мир, 1991.
6. Henson M. Elements of functional languages. Dept. of CS. University of Sassex, 1990.

7. Fokker J. Functional programming. Dept. of CS. Utrecht University, 1995.
8. Thompson S. Haskell: The Craft of Functional Programming. 2-nd edition, Addison-Wesley, 1999.
9. Bird R. Introduction to Functional Programming using Haskell. 2-nd edition, Prentice Hall Press, 1998.

2. მონაცემთა სტრუქტურები და ბაზური ოპერაციები

როგორც უკვე ავღნიშნეთ, პროგრამირების ფუნქციონალური პარადიგმის საფუძველს წარმოადგენს მათემატიკური აზროვნების განვითარების ისეთი მიმართულებები, როგორიცაა კომბინატორული ლოგიკა და λ–აღრიცხვა. ეს უკანასკნელი უფრო მჭიდროდ არის დაკავშირებული ფუნქციონალურ პროგრამირებასთან. სწორედ λ–აღრიცხვა არის ფუნქციონალური პროგრამირების თეორიული საფუძველი.

იმისათვის, რომ განვიხილოთ ფუნქციონალური პროგრამირების თეორიული საფუძვლები, პირველ რიგში აუცილებელია შემოვიტანოთ ზოგიერთი შეთანხმება, ავღწეროთ აღნიშვნები და ავაგოთ ფორმალური სისტემა.

ვთქვათ, მოცემულია რომელიღაც `A` პირველადი ტიპის ობიექტები. ამჟამად არ აქვს მნიშვნელობა, თუ კონკრეტულად რას წარმოადგენს გამოყოფილი ობიექტები. საზოგადოდ, ითვლება, რომ ამ ობიექტებზე განისაზღვრება ბაზისური ოპერაციების და პრედიკატების ერთობლიობა. ტრადიციულად, ეს მოდის მაკარტიდან (`Lisp`–ის ავტორისგან), რომ ობიექტებს უწოდებენ ატომებს. თეორიულად, არანაირი მნიშვნელობა არ აქვს ბაზისური ოპერაციებისა და პრედიკატების რეალიზების საშუალებებს, უბრალოდ ხდება მათი პოსტილურება. თუმცა, ყოველი ფუნქციონალური ენა თავისებურად რეალიზებს ბაზისურ ნაკრებს.

ტრადიციულად (და, უპირველეს ყოვლისა, ეს აიხსნება თეორიული აუცილებლობით) ბაზისურ ოპერაციად გამოიყოფა შემდეგი სამი ოპერაცია:

1. წყვილის შექმნის ოპერაცია — **prefix** `(x, y) = x : y = [x | y]`. მას ხშირად უწოდებენ კონსტრუქტორს ანუ შემდგენელს.

2. თავის გამოყოფის ოპერაცია — **head** `(x) = h (x)`. ეს არის პირველი სელექტორული ოპერაცია.

3. კუდის გამოყოფის ოპერაცია — **tail** `(x) = t (x)`. ეს არის მეორე სელექტორული ოპერაცია.

თავისა და კუდის გამოყოფის სელექტორულ ოპერაციებს ხშირად უწოდებენ უბრალოდ სელექტორებს. ეს ოპერაციები დაკავშირებულია ერთმანეთთან შემდეგი სამი აქსიომით:

1. **head** `(x : y) = x`
2. **tail** `(x : y) = y`
3. **prefix** (**head** `(x : y)`, **tail** `(x : y)) = (x : y)`

ყველა ობიექტის სიმრავლე, რომელიც შეიძლება კონსტრუირდეს პირველადი ტიპის ობიექტებისგან ბაზისური ოპერაციების გამოყენების შედეგად, უწოდებენ `S` გამოსახულებას (აღნიშვნა — `Sexpr (A)`). მაგალითად:

`a1 : (a2 : a3) in Sexpr`

შემდგომი კვლევისთვის შემოდის ცნება ფიქსირებული ატომი, რომელიც აგრეთვე ეკუთვნის პირველად `A` ტიპს. ამ ატომს შემდგომში ვუწოდებთ „ცარიელ სიას“ და ავღნიშნავთ სიმბოლოებით `[]` (თუმცა, ფუნქციონალური პროგრამირების სხვადასხვა ენაში შეიძლება გამოყენებული იყოს ცარიელი სიის სხვა აღნიშვნებიც). ეხლა ავღწეროთ ის, რაზეც ოპერირებს ფუნქციონალური პროგრამირება – `List (A) Sexpr (A)`, საკუთრივი ქვესიმრავლე, რომელსაც ეწოდება „სია `A`–ზე“.

განმარტება:

`1°. ცარიელი სია [] in List (A)`

`2°. x in A & y in List (A) => x : y in List (A)`

სიის მთავარი თვისება არის: `x in List (A) & x =/= [] =>` **`head`** `(x) in A;` **`tail`** `(x) in List (A).`

`n` ელემენტიანი სიის აღსანიშნავად შეიძლება გამოყენებული იყოს სხვადასხვა ნოტაცია, თუმცა ჩვენ გამოვიყენებთ მხოლოდ ასეთს: `[a1, a2, ..., an].` სიასთან ოპერაციების `head`–ის და `tail`–ის მეშვეობით შეიძლება სიის თითოეულ ელემენტთან წვდომა, რადგანაც:

`head` `([a1, a2, ..., an]) = a1`

`tail` `([a1, a2, ..., an]) = [a2, ..., an] (როცა n > 0).`

სიების გარდა შემოდის მონაცემების კიდევ ერთი ტიპი, რომელსაც ეწოდება „სიური სტრუქტურა `A`–ზე“ (აღნიშვნა – `List_str (A)`), ამასთან, შესაძლოა აგებული იყოს დამოკიდებულების შემდეგი სტრუქტურა: `List (A) List_str (A) Sexpr (A).` სიური სტრუქტურის განმარტებას აქვს შემდეგი სახე:

განმარტება:

`1°. a in A => a in List_str (A)`

`2°. List (List_str (A)) in List_str (A)`

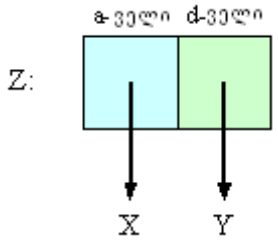
ანუ, ჩანს, რომ სიური სტრუქტურა – ეს არის სია, რომლის ელემენტები შეიძლება იყოს როგორც ატომები, ასევე სხვა სიური სტრუქტურები, მათ შორის ჩვეულებრივი სიები. სიური სტრუქტურის მაგალითი, რომელიც ამავე დროს არ არის მარტივი სია, არის შემდეგი გამოსახულება: `[a1, [a2, a3, [a4]], a5].` სიური სტრუქტურისთვის შემოდის ისეთი ცნება, როგორიცაა ჩადგმის დონე.

რამდენიმე სიტყვა პროგრამულ რეალიზაციაზე

განვიხილოთ სიებისა და სიური სტრუქტურების პროგრამული რეალიზაციები. ეს საჭიროა იმისთვის, რომ გავიგოთ, რა ხდება ფუნქციონალური პროგრამის

მუშაობისას როგორც რომელიმე კონკრეტულ ფუნქციონალურ ენაზე, ასევე აბსტრაქტულ ენაზე.

თითოეული ობიექტი მანქანის მეხსიერებაში იკავებს რაღაც ადგილს. ატომები წარმოადგენს მიმთითებლებს (მისამართებს) უჯრედებზე, რომლებშიც ობიექტი ინახება. ასეთ შემთხვევაში წყვილი `z = x : y` გრაფიკულად შეიძლება წარმოვადგინოთ, როგორც ნაჩვენებია შემდეგ სურათი 1-ზე:

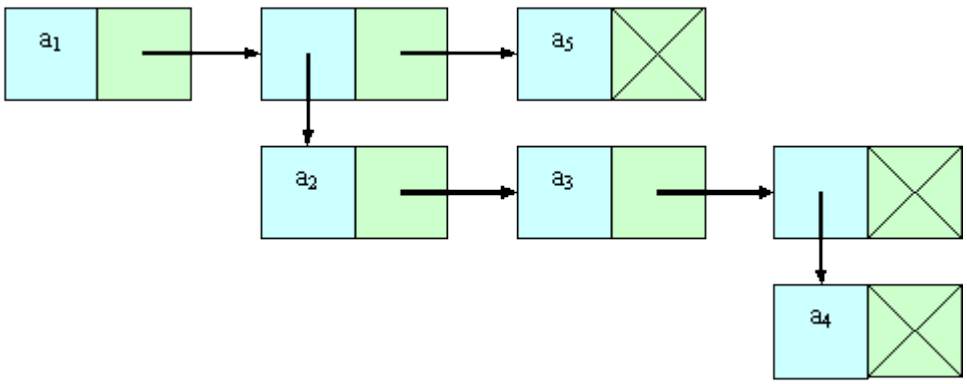


სურათი 1. წყვილის წარმოდგენა კომპიუტერის მეხსიერებაში

უჯრედის მისამართი, რომელიც შეიცავს მიმთითებლებს `x`–ზე და `y`–ზე, არის ობიექტი `z`. როგორც ნახატიდან ჩანს, წყვილი წარმოდგენილია ორი მისამართით – მიმთითებლით თავზე და კუდზე. ტრადიციულად პირველ მიმთითებელს უწოდებენ `a`-ველს, მეორე მიმთითებელს – `d`-ველს.

ობიექტები, რომელზეც `a`-ველი და `d`-ველი მიუთითებს, რომ უფრო მოსახერხებლად წარმოვადგინოთ, შემდგომში მათ ჩავწერთ უშუალოდ ველებში. ცარიელ სიას ავღნიშნავთ გადახაზული კვადრატით.

ამრიგად, სიური სტრუქტურა `[a1, [a2, a3, [a4]], a5]` შეიძლება წარმოდგეს შემდეგნაირად (სურათი 2):



სურათი 2. `[a1, [a2, a3, [a4]], a5]` გრაფიკული წარმოდგენა სიური სტრუქტურის

სურათზე კარგად ჩანს ჩადგმის დონეები – `a1` და `a5` ატომებს აქვთ ჩადგმის დონე 1, ხოლო ატომებს `a2` და `a3` – 2, ხოლო ატომს `a4` – 3 შესაბამისად.

ავღნიშნოთ, რომ ოპერაცია prefix ითხოვს მეხსიერებას, ვინაიდან წყვილის კონსტუირების დროს გამოიყოფა მეხსიერება მიმთითებლებისთვის. მეორეს მხრივ, ოპერაციებს head და tail არ სჭირდება მეხსიერება, ისინი უბრალოდ აბრუნებენ მისამართებს, რომლებიც შეიცავს შესაბამისად a-ველს და d-ველს.

მაგალითები

მაგალითი 5. ოპერაცია prefix.

თავდაპირველად უფრო დატალურად განვიხილოთ ოპერაცია prefix–ის მუშაობა. ოპერატორის მუშაობა განვიხილოთ სამ ზოგად მაგალითზე:

1°. prefix (a1, a2) = a1 : a2 (ამასთან, შედეგი არ არის List_str (A)–ის ელემენტი).

2°. prefix (a1, [b1, b2]) = [a1, b1, b2]

3°. prefix ([a1, a2], [b1, b2]) = [[a1, a2], b1, b2]

მაგალითი 6. სიის სიგრძის განსაზღვრის ფუნქცია Length.

მანამ, სანამ დავიწყებდებდეთ უშუალოდ ფუნქცია Length–ის რეალიზებას, ვნახოთ, თუ რას აბრუნებს იგი. ფუნქცია Length–ის შედეგი არის ელემენტების რაოდენობა იმ სიაში, რომელიც გადაეცემა მას პარამეტრად. აქ ორი შემთხვევაა– ფუნქციას გადაეცა ცარიელი სია და ფუნქციას გადაეცა არაცარიელი სია. პირველ შემთხვევაში ცხადია შედეგი უნდა იყოს 0. მეორე შემთხვევაში ამოცანა ორ ქვეამოცანად იყოფა, სია იყოფა თავად და კუდად ოპერაციების head–ის და tail–ის საშუალებით.

ვიცით, რომ head აბრუნებს სიის პირველ ელემენტს, ხოლო ოპერაცია tail აბრუნებს დანარჩენი ელემენტების სიას. თუ გვეცოდინება რისი ტოლია tail ოპერაციით მიღებული სიის სიგრძე, მაშინ თავდაპირველი სიის სიგრძე იქნება ეს სიგრძე ერთით გადიდებული. ეხლა უკვე ადვილად შეგვიძლია დავწეროთ ფუნქცია Length–ის განმარტება:

```
Length ([]) = 0
Length (L) = 1 + Length (tail (L))
```

მაგალითი7. ორი სიის შერწყმის ფუნქცია Append.

ორი სიის შერწყმა (ანუ გაერთიანება) შეიძლება რამდენიმე საშუალებით. პირველი –დესტრუქციული მინიჭება, ანუ შევცვალოთ [] სიაზე მინიჭება მეორე სიის თავზე მიმთითებლით და ამით მივიღებთ შედეგს პირველ სიაში. მაგრამ ამ დროს

იცვლება პირველი სია. ასეთი მიდგომები ფუნქციონალურ პროგრამირებაში დაუშვებელია (თუმცა ზოგიერთ ენაში ეს დასაშვებია).

მეორე მიდგომა მდგომარეობს შემდეგში: მოვახდინოთ პირველი სიის კოპირება ზედა დონეზე და მოვათავსოთ კოპიოს ბოლო მიმთითებლის ნაცვლად მიმთითებელი მეორე სიის პირველ ელემენტზე. ეს მიდგომა კარგია იმით, რომ არ ასრულებს დესტრუქციულ მოქმედებას და არ აქვს გვერდითი ეფექტები, თუმცა მოითხოვს დამატებით მეხსიერებასა და დროს.

```
Append ([], L2) = L2
Append (L1, L2) = prefix (head (L1), Append (tail (L1), L2))
```

ბოლო მაგალითი გვიჩვენებს, თუ როგორ შეიძლება თანდათანობითი კონსტრუირებით აიგოს ახალი სია, რომელიც იქნება ორი მოცემულის შერწყმა.

სავარჯიშოები

1. ააგეთ ფუნქცია, რომელიც გამოითვლის შემდეგი მიმდევრობების n-ურ წევრს:

a. $a_n = x_n$

b. $a_n = Summ\ i,\ (i = 1,n)$

c. $a_n = Summ\ (Summ\ i),\ (j = 1,n\ i = 1,j)$

d. $a_n = Summ\ n^{-i},\ (i = 1,p)$

e. $a_n = e^n = Summ\ (n^i / i!),\ (i = 0,infinity)$

2. ახსენით prefix ოპერაციის შედეგი, რომელიც მოყვანილია მაგალით 5–ში. ახსნისას შეგიძლიათ გამოიყენოთ გრაფიკული მეთოდი.

3. ახსენით ფუნქცია Append–ის შედეგი (მაგალითი 7). ახსენით, რატომ არ არის ფუნქცია დესტრუქციული.

4. ააგეთ ფუნქცია, რომელიც მუშაობს სიებთან:

a. GetN – ფუნქცია, რომელიც მოცემული სიიდან n-ურ ელემენტს გამოყოფს.

b. ListSumm – ორი სიის ელემენტების შეკრება. აბრუნებს სიას, რომელიც არის პარამეტრი სიების ელემენტების ჯამი. გაითვალისწინეთ, რომ სიების სიგრძე შეიძლება იყოს სხვადასხვა.

c. OddEven – ფუნქცია უცვლის ადგილებს მეზობელ ლუწ და კენტ ელემენტებს მოცემულ სიაში.

d. Reverse — ფუნქცია, რომელიც აბრუნებს სიას (სიის პირველი ელემენტი ხდება ბოლო, მეორე–ბოლოდან მეორე და ა.შ. ბოლო ელემენტამდე).

e. Map — ფუნქცია, რომელიც იყენებს მეორე ფუნქციას (რომელიც პარამეტრად გადაეცემა თავდაპირველ ფუნქციას) მოცემული სიის ყველა ელემენტთან.

პასუხები თვითშემოწმებისთვის

პასუხები ხშირად წარმოდგენილი გვაქვს რამდენიმე შესაძლო ვარიანტიდან ერთ–ერთი.

1. ფუნქციები, რომლებიც ითვლის მიმდევრობის N–ურ წევრს:

a. Power:

```
Power (X, 0) = 1
Power (X, N) = X * Power (X, N - 1)
```

b. Summ_T:

```
Summ_T (1) = 1
Summ_T (N) = N + Summ_T (N - 1)
```

c. Summ_P:

```
Summ_P (1) = 1
Summ_P (N) = Summ_T (N) + Summ_P (N - 1)
```

d. Summ_Power:

```
Summ_Power (N, 0) = 1
Summ_Power (N, P) = (1 / Power (N, P)) + Summ_Power (N, P - 1)
```

e. Exponent:

```
Exponent (N, 0) = 1
Exponent (N, P) = (Power (N, P) / Factorial (P)) + Exponent (N, P - 1)


Factorial (0) = 1
Factorial (N) = N * Factorial (N - 1)
```

2. ოპერაცია prefix–ის მუშაობის მაგალითი შეიძლება წარმოვადგინოთ სამი მიდგომით (ისევე, როგორც ეს განხილულია მაგალითში). შესაძლებელია ოპერაცია prefix–ის წარმოდგენა ინფიქსური ჩანაწერის, სიმბოლო :–ის გამოყენებით.

a. ოპერაციის მუშაობის პირველი მაგალითია თვით ოპერაციის განმარტება. მის განხილვას არ აქვს აზრი, რადგან prefix სწორედ ასე განიმარტება. b. prefix (a1, [b1, b2]) = prefix (a1, b1 : (b2 : [])) = a1 : (b1 : (b2 : [])) = [a1, b1, b2]

(ეს გარდაქმნები მოყვანილია სიის განმარტების მიხედვით).

c. prefix ([a1, a2], [b1, b2]) = prefix ([a1, a2], b1 : (b2 : [])) = ([a1, a2]) : (b1 : (b2 : [])) = [[a1, a2], b1, b2].

3. Append ფუნქციის მუშაობის მაგალითად განვიხილოთ ორი სიის შერწყმის მაგალითი, რომლიდანაც თითოეული შედგება ორი ელემენტისგან: [a, b] და [c, d]. რომ არ გადავტვირთოთ ახსნით, ოპერაცია prefix–სთვის გამოვიყენოთ ინფიქსური ფორმის ჩანაწერი:

Append ([a, b], [c, d]) = a : Append ([b], [c, d]) = a : (b : Append ([], [c, d])) = a : (b : ([c, d])) = a : (b : (c : (d : []))) = [a, b, c, d].

4. ფუნქციები, რომლებიც მუშაობენ სიებთან:

a. GetN:

```
GetN (N, []) = _
GetN (1, H:T) = H
GetN (N, H:T) = GetN (N - 1, T)
```

b. ListSumm:

```
ListSumm ([], L) = L
ListSumm (L, []) = L
ListSumm (H1:T1, H2:T2) = prefix ((H1 + H2), ListSumm (T1, T2))
```

c. OddEven:

```
OddEven ([]) = []
OddEven ([X]) = [X]
OddEven (H1:[H2:T]) = prefix (prefix (H2, H1), OddEven (T))
```

d. Reverse:

```
Reverse ([]) = []
```

```
Reverse (H:T) = Append (Reverse (T), [H])
```

e. Map:

```
Map (F, []) = []
Map (F, H:T) = prefix (F (H), Map (F, T))
```

მონაცემთა სტრუქტურა და ბაზისური ოპერაციები (გაგრძელება)

ტიპები ფუნქციონალურ ენებში

როგორც ცნობილია, ფუნქციის არგუმენტები შეიძლება იყოს არა მხოლოდ ბაზური ტიპის ცვლადები, არამედ სხვა ფუნქციებიც. ამ შემთხვევაში ჩნდება მაღალი რიგის ფუნქციის ცნება. შემოვიღოთ ფუნქციონალური ტიპის ცნება (ანუ ტიპის, რომელიც აბრუნებს ფუნქციას). ვთქვათ, რომელიღაც `f` – ფუნქცია არის ერთი ცვლადის ფუნქცია `A` სიმრავლიდან, რომელიც ღებულობს მნიშვნელობებს `B` სიმრავლიდან, მაშინ განსაზღვრების თანახმად:

```
#(f) : A -> B
```

აქ ნიშანი `#(f)` აღნიშნავს „ფუნქცია `f`-ის ტიპი“. ამრიგად, ტიპს, რომელსაც აქვს სიმბოლო ისარი `->`, ეწოდება ფუნქციონალური ტიპი. ზოგჯერ მისთვის გამოიყენება აღნიშვნა: B^A (შემდგომში გამოვიყენებთ მხოლოდ ისრიან ჩანაწერს, ვინაიდან ზოგიერთი ფუნქციის ტიპი ძალზე რთულად წარმოდგება ხარისხებით).

მაგალითად:

```
#(sin) : Real -> Real
#(Length) : List (A) -> Integer
```

მრავალარგუმენტიანი ფუნქციისთვის ტიპის განსაზღვრა შეიძლება გამოყვანილი იყოს ოპერაციით – დეკარტული ნამრავლით (მაგალითად, `#(add(x, y)) : Real x Real -> Real`). თუმცა ფუნქციონალურ პროგრამირებაში ასეთმა საშუალებამ გამოყენება ვერ ჰპოვა.

1924 წელს მ. შონფიკელმა მრავალარგუმენტიანი ფუნქცია წარმოადგინა როგორც ერთარგუმენტიანი ფუნქციების თანმიმდევრობა. ასეთ შემთხვევაში, ფუნქციის ტიპი, რომელიც შეკრებს ორ ნამდვილ რიცხვს, წარმოდგება ასე: `Real -> (Real -> Real)`. ანუ ასეთი ფუნქციის ტიპი მიიღება სიმბოლო ისრის `->` თანმიმდევრული გამოყენებით. განვსაზღვროთ ეს პროცესი შემდეგ მაგალითზე:

მაგალითი 8. ფუნქცია `add (x,y)`–ის ტიპი.

დავუშვათ, ფუნქცია `add`–ის თითოეული არგუმენტი უკვე აღნიშნულია, ვთქვათ `x = 5, y = 7`. ამ შემთხვევაში თუ ფუნქცია `add`–ს მოვაშორებთ პირველ არგუმენტს, მივიღებთ ახალ ფუნქციას – `add5`, რომელიც თავის ერთადერთ არგუმენტს უმატებს რიცხვს `5`–ს. ამ ფუნქციის ტიპი მიიღება ადვილად და

წარმოდგება ასე: `Real -> Real`. ეხლა, თუ უკან დავბრუნდებით, უკვე გვესმის, რატომ არის `add` ფუნქციის ტიპი `Real -> (Real -> Real)`.

რათა ავიცილოთ `add5` ტიპის ფუნქციების დაწერა (როგორც წინა მაგალითში), მოგონებული იყო სპეციალური აპლიკაციური ჩაწერის ფორმა სახით „ოპერატორი – ოპერანდი". ამის წინაპირობა გახდა ახალი ხედვა ფუნქციისა ფუნქციონალურ პროგრამირებაში. ტრადიციულად ითვლებოდა, რომ გამოსახულება `f (5)` აღნიშნავს „`f` ფუნქციის გამოყენება არგუმენტის მნიშვნელობასთან, რომელიც ტოლია `5`–ის" (ანუ, გამოითვლება მხოლოდ არგუმენტი). ფუნქციონალურ პროგრამირებაში კი ითვლება, რომ ფუნქციის მნიშვნელობაც ასევე ითვლება. ასე, რომ, დავუბრუნდეთ მაგალით 8–ს, ფუნქცია `add` შეიძლება ასე `(add (x)) y`, ხოლო როცა არგუმენტები იღებს კონკრეტულ მნიშვნელობებს (მაგალითად, `(add (5)) 7`), თავიდან ითვლება ყველა ფუნქცია, სანამ არ დარჩება ერთარგუმენტიანი ფუნქცია, რომელიც გამოიყენება უკანასკნელთან.

ამრიგად, თუ ფუნქცია `f`–ს აქვს ტიპი A_1 `->` (`(`A_2 `->` `( ... (`A_n `-> B) ... ))`, მაშინ, რათა სრულად გამოვთვალოთ მნიშვნელობა `f` (`(`a_1`,` a_2`, ...,` a_n`)`, აუცილებელია თანმიმდევრულად გამოვთვალოთ `( ... (f (`a_1`)` a_2`) ... )` a_n და გამოთვლის შედეგი იქნება `B` ტიპის ობიექტი.

შესაბამისად, გამოსახულება, რომელშიც ყველა ფუნქცია განიხილება როგორც ერთარგუმენტიანი ფუნქცია და ერთადერთ ოპერაციას წარმოადგენს აპლიკაცია (გამოყენება), ეწოდება გამოსახულება ფორმით „ოპერატორი–ოპერანდი". ასეთმა ფუნქციებმა მიიღეს სახელწოდება „კარირებული", ხოლო თვითონ ფუნქციის დაყვანის ზემოთ აღწერილმა პროცესმა – „კარირება" (კარი ჰასკელის სახელიდან გამომდინარე).

თუ გავიხსენებთ λ–აღრიცხვას, აღმოვაჩენთ, რომ მასში უკვე არის მათემატიკური აბსტრაქცია ჩანაწერების აპლიკაციური ფორმისთვის. მაგალითად:

`f (x) = x2 + 5`	`<=>`	`λx.(x2 + 5)`
`f (x, y) = x + y`	`<=>`	`λy. λx.(x + y)`
`f (x, y, z) = x2 + y2 + z2`	`<=>`	`λz. λy. λx.(x2 + y2 + z2)`

და ა.შ....

რამდენიმე სიტყვა აბსტრაქტული ენის ნოტაციის შესახებ

ნიმუშები და კლოზები

ავღნიშნოთ, რომ აბსტრაქტული ფუნქციონალური ენის ნოტაციაში, რომელსაც ვიყენებით ფუნქციის მაგალითების დაწერისას, შესაძლებელი იყო გამოგვეყენებინა

ისეთი კონსტრუქცია, როგორიცაა `if-then-else`. მაგალითად, ფუნქცია `Append`–ის აღწერისას (იხილეთ მაგალითი 7), მისი ტანი შეიძლება ჩაწერილიყო შემდეგნაირად:

```
Append (L1, L2) = if (L1 == []) then L2
                  else head (L1) : Append (tail (L1), L2)
```

თუმცა მოცემული ჩანაწერი ცუდად გასარჩევია, ამიტომაც მაგალითში 7 უკვე გამოვიყენეთ ნოტაცია, რომელიც მხარს უჭერს ე.წ. "ნიმუშებს".

განმარტება:

ნიმუში ეწოდება გამოსახულებას, რომელიც აგებულია მონაცემთა კონსტრუირების ოპერაციით და რომელიც გამოიყენება მონაცემებთან შესაბამისობისათვის. ცვლადები აღინიშნება დიდი ასოებით, კონსტანტები–პატარათი.

ნიმუშის მაგალითებია:

`5` – რიცხვითი კონსტანტა.

`X` –ცვლადი.

`X : (Y : Z)` – წყვილი.

`[X, Y]` – სია.

ნიმუშმა აუცილებლად უნდა დააკმაყოფილოს ერთი მოთხოვნა, წინააღმდეგ შემთხვევაში მასთან შედარება არასწორად შესრულდება. ეს მოთხოვნა ასე ჟღერს: ნიმუშთან მონაცემების შედარებისას ცვლადისთვის მნიშვნელობის მინიჭება უნდა მოხდეს მხოლოდ ერთადერთი გზით, ანუ, მაგალითად, გამოსახულება `(1 + X ==> 5)` შეიძლება გამოვიყენოთ როგორც ნიმუში, რადგანაც `X` ცვლადის აღნიშვნა ხდება ერთადერთი გზით `(X = 4)`, ხოლო შემდეგი გამოსახულების `(X + Y ==> 5)` გამოყენება ნიმუშად არ შეიძლება, ვინაიდან `X` და `Y` ცვლადების აღინიშნება სხვადასხვანაირად (არაცალსახად).

ფუნქციონალურ პროგრამირებაში ნიმუშის გარდა შემოდის ისეთი ცნება, როგორიცაა „კლოზი" (ინგლისურიდან „clause"). განმარტებით, კლოზი ეს არის:

def f p_1, ..., p_n **=** expr

სადაც:

def და **=** – აბსტრაქტული ენის კონსტანტებია.

`f` – განსაზღვრულიი ფუნქციის სახელია.

p_i – ნიმუშების თანმიმდევრობაა (ამასთან, `>= 0`).

`expr` – გამოსახულებაა.

ამრიგად, ფუნქციონალურ პროგრამირებაში ფუნქციების განსაზღვრება არის უბრლოდ კლოზების თანმიმდევრობა (შესაძლოა, მხოლოდ ერთი ელემენტისგან შემდგარი). რათა გავამარტივოთ ფუნქციის განსაზღვრების ჩაწერა, შემდგომში სიტყვა `def`–ს გამოვტოვებთ.

მაგალითი 9. ნიმუშები და კლოზები ფუნქციაში `Length`.

```
Length ([]) = 0
Length (H:T) = 1 + Length (T)
```

ვთქვათ ფუნქცია `Length`–ის გამოძახება ხდება პარამეტრით `[a, b, c]`. ამ დროს მუშაობას იწყებს ნიმუშთან შედარების მექანიზმი. სათითაოდ გადაისინჯება ყველა კლოზი და ხდება შედარებების მცდელობები. ამ შემთხვევაში წარმატებით მოხდება მხოლოდ მეორე კლოზთან შედარება (რადგანაც სია `[a, b, c]` არ არის ცარიელი).

ფუნქციის გამოძახების ინტერპრეტაცია მდგომარეობს შემდეგში: ხდება შედარება ზემოდან ქვემოთ ნიმუშებში და რიგით პირველი ნიშუშის პოვნა, რომელიც წარმატებით შედარდა ფაქტიურ პარამეტრებს. ნიმუშის ცვლადების მნიშვნელობები, რომლებიც მათ მიენიჭათ შედარების შედეგად, ჩაისმის კლოზის მარჯვენა მხარეს (გამოსახულებაში `expr`), რომლის მნიშვნელობის გამოთვლაც მოცემულ კონტექსტში წარმოადგენს ფუნქციის გამოძახების მნიშვნელობას.

დაცვა

აბსტრაქტულ ნოტაციაში ფუნქციის დაწერისას დაშვებულია ე.წ. დაცვის გამოყენება, ანუ ნიმუშის ცვლადებზე შეზღუდვების გამოყენება. მაგალითად, დაცვის გამოყენებით ფუნქცია `Length`–ის განსაზღვრა შეიძლება იყოს შემდეგი:

```
Length (L) = 0 when L == []
Length (L) = 1 + Length (tail (L)) otherwise
```

განხილულ კოდში სიტყვა `when` (მაშინ) და `otherwise` (წინააღმდეგ შემთხვევაში) წარმოადგენს ენის დარეზერვირებულ სიტყვებს. თუმცა, ამ სიტყვების გამოყენება არ არის დაცვის გამოყენებისთვის აუცილებელი პირობა. დაცვა შეიძლება განვახორციელოთ სხვადასხვა საშუალებით, მათ შორის λ–აღრიცხვით:

```
Append = λ[].(λL.L)
Append = λ(H:T).(λL.H : Append (T, L))
```

წარმოდგენილი ჩანაწერი ცუდი წასაკითხია, ამიტომ მას მხოლოდ უკიდურეს შემთხვევებში გამოვიყენებთ.

ლოკალური ცვლადები

როგორც უკვე ავღნიშნეთ, ლოკალური ცვლადების ცვლადების გამოყენება იწვევს გვერდით ეფექტს, ამიტომ იგი დაუშვებელია ფუნქციონალურ ენებში. თუმცა, ზოგიერთ შემთხვევაში ლოკალური ცვლადების გამოყენება არის ოპტიმალური, რაც იძლევა გამოთვლების დროის და რესურსების ეკონომიის საშუალებას.

დავუშვათ, `f` და `h` ფუნქციებია და აუცილებელია გამოითვალოს გამოსახულება `h (f(X), f(X))`. თუ ენაში არ არის ჩადებული ოპტიმიზაციის მეთოდები, მაშინ ხდება `f (X)` გამოსახულების ორჯერ გამოთვლა. ეს რომ არ მოხდეს, შეიძლება მივმართოთ ასეთ საშუალებას: `(λv.h (v, v))(f (X))`. ბუნებრივია, რომ ამ შემთხვევაში გამოსახულება `f (X)` გამოითვლება პირველად და ერთხელ. იმისათვის, რომ λ–აღრიცვის გამოყენება მინიმალურად მოხდეს, შემდგომში შემდეგი სახის ჩანაწერს გამოვიყენებთ:

```
let v = f (X) in h (v, v)
```

(სიტყვები **let, =** და **in** – ენაში დარეზერვირებულია). ამ შემთხვევაში `v` ვუწოდებთ ლოკალურ ცვლადს.

პროგრამირების ელემენტები

პარამეტრების დაგროვება – აკუმულატორი

ფუნქციის შესრულებისას შეიძლება დადგეს მეხსიერების ხარჯვის სერიოზული პრობლემა. ავხსნათ ეს პრობლემა ფუნქციის მაგალითზე, რომელიც ითვლის რიცხვის ფაქტორიალს:

```
Factorial (0) = 1
Factorial (N) = N * Factorial (N - 1)
```

თუ მოვიყვანთ ამ ფუნქციის გამოთვლის მაგალითს არგუმენტზე 3, მაშინ დავინახავთ შემდეგ თანმიმდევრობას:

```
Factorial (3)
3 * Factorial (2)
3 * 2 * Factorial (1)
3 * 2 * 1 * Factorial (0)
3 * 2 * 1 * 1
3 * 2 * 1
3 * 2
```

```
6
```

ამ გამოთვლის მაგალითზე ვხედავთ, რომ ფუნქციის რეკურსიული გამოყენება ძალიან ძლიერად იყენებს მეხსიერებას. ამ შემთხვევაში მეხსიერება არგუმენტის მნიშვნელობის პროპორციულია, მაგრამ არგუმენტებიც შეიძლება იყოს დიდი. ჩნდება კითხვა, შესაძლებელია თუ არა ისე დაიწეროს ფაქტორიალის გამოთვლის ფუნქცია (და მისი მსგავსი ფუნქციები), რომ მეხსიერება მინიმალურად იქნას გამოყენებული?

ამ შეკითვაზე დადებითი პასუხისთვის აუცილებელია განვიხილოთ აკუმულატორის (დამგროვებლის) ცნება. ამისთვის განვიხილოთ შემდეგი მაგალითი:

მაგალითი 10. ფაქტორიალის გამოთვლის ფუნქცია აკუმულატორის გამოყენებით.

```
Factorial_A (N) = F (N, 1)


F (0, A) = A
F (N, A) = F ((N - 1), (N * A))
```

ამ მაგალითში ფუნქცია F–ის მეორე პარამეტრი ასრულებს აკუმულატორი ცვლადის როლს. სწორედ იგი შეიცავს შედეგს, რომელიც ბრუნდება რეკურსიის დამთავრებისას. თვითონ რეკურსიას კი, ამ დროს აქვს „კუდური“ რეკურსიის სახე, ამასთან, მეხსიერება გამოიყენება მხოლოდ ფუნქციის მნიშვნელობების დასაბრუნებელი მისამართების შენახვისათვის.

კუდური რეკურსია წარმოადგენს რეკურსიის სპეციალურ სახეს, რომლის დროსაც მხოლოდ ერთხელ ხდება რეკურსიული ფუნქციის გამოძახება და ეს გამოძახებაც სრულდება ყველა გამოთვლის შემდეგ.

კუდური რეკურსიის რეალიზაცია შეიძლება შესრულდეს იტერაციის პროცესის საშუალებით. პრაქტიკაში ეს ნიშნავს, რომ ფუნქციონალური ენის „კარგ“ ტრანსლიატორს უნდა „შეეძლოს“ აღმოაჩინოს კუდური რეკურსია და მოახდინოს მისი რეალიზება ციკლის სახით. თავის მხრივ, პარამეტრის დაგროვების მეთოდს ყოველთვის არ მოვყავართ კუდურ რეკურსიამდე, თუმცა იგი ცალსახად გვეხმარება საერთო მეხსიერების მოცულობის შემცირებაში.

დამგროვებელი პარამეტრებით განსაზღვრებების აგების პრინციპები:

1. შემოდის ახალი ფუნქცია დამატებითი არგუმენტით (აკუმულატორით), რომელშიც გროვდება გამოთვლების შედეგები.

2. აკუმულატორი არგუმენტის საწყისი მნიშვნელობა მოიცემა ტოლობით, რომელიც აკავშირებს ძველ და ახალ ფუნქციებს.

3. საწყისი ფუნქციის ის ტოლობა, რომელიც შეესაბამება რეკურსიიდან გამოსავალს, იცვლება აკუმულატორით დაბრუნებით.

4. ტოლობა, რომელიც შეესაბამება რეკურსიულ განსაზღვრებას, გამოიხატება როგორც ახალ ფუნქციაზე მიმართვა, რომელშიც აკუმულატორი იღებს იმ მნიშვნელობას, რომელიც ბრუნდება საწყისი ფუნქციით.

ისმის შეკითვა: ნებისმიერი ფუნქცია შეიძლება გარდავქმნათ აკუმულატორის გამოთვლის ფორმით? ბუნებრივია, რომ ამ შეკითხვის პასუხი არის უარყოფითი. დამგროვებელი პარამეტრიანი ფუნქციის აგების ხერხი არ არის უნივერსალური და იგი არ არის კუდური რეკურსიის აგების გარანტია. მეორეს მხრივ, განსაზღვრების აგება დამგროვებელი პარამეტრით არის შემოქმედებითი საქმე. ამ პროცესში აუცილებელია ზოგიერთი ევრისტიკის გამოყენება.

განსაზღვრება:

რეკურსიული განსაზღვრების ზოგად სახეს, რომელიც საშუალებას იძლევა ტრანსლიაციისას უზრუნველყოს გამოთვლები მეხსიერების მუდმივ მოცულობაში იტერაციის საშუალებით, უწოდებენ იტერაციული სახის ტოლობებს.

$f_i\ (p_{ij}) = e_{ij}$

ამასთან, გამოსახულება e_{ij}-ს ედება შემდეგი შეზღუდვები:

1. e_{ij} — „მარტივი“ გამოსახუილებაა, ანუ ის შეიცავს მხოლოდ მონაცემებზე ოპერაციებს და არ შეიცავს რეკურიულ გამოძახებებს.

2. e_{ij}-ს აქვს სახე $f_k\ (v_k)$, ამასთან v_k — მარტივი გამოსახულებების თანმიმდევრობაა. ეს არის კუდური რეკურსია.

3. e_{ij} — პირობითი გამოსახულებაა პირობაში მარტივი გამოსახულებით, რომლის შტოები განისაზღვრება ამავე სამი პირობით.

სავარჯიშოები

1. ააგეთ ფუნქცია, რომლებიც მუშაობს სიებთან. საჭიროების შემთხვევაში გამოიყენეთ დამატებითი ფუნქციები და ზემოთ განსაზღვრული ფუნქციები.

a. Reverse_all — ფუნქცია, რომელიც შესასვლელზე იღებს სიურ სტრუქტურას და აბრუნებს მის ყველა ელემენტს და ასევე მას.

b. Position — ფუნქცია, რომელიც აბრუნებს მოცემული ატომის სიაში პირველად შესვლის ნომერს.

c. Set — ფუნქცია, რომელიც აბრუნებს სიას, რომელშიც მოცემული სიის ყველა ატომი მხოლოდ ერთხელ შედის.

d. Frequency — ფუნქცია, რომელიც აბრუნებს წყვილების სიას (სიმბოლო, სიხშირე). თითოეული წყვილი განისაზღვრება მოცემული სიის ატომით და ამ სიაში მისი შესვლის სიხშირით.

2. დაწერეთ ფუნქციები დამგროვებელი პარამეტრებით (თუ ეს შესაძლებელია) სავარჯიშო 1-ში მოყვანილი ფუნქციებისთვის.

პასუხები თვითშემოწმებისთვის

1. შემდეგი აღწერები რეალიზებს მოთხოვნილ ფუნქციებს. ზოგიერთ პუნქტებში რეალიზებულია დამატებითი ფუნქციები, რომლებზეც გვერდის ავლა ძნელად წარმოგვიდგენია. a. Reverse_all:

```
Reverse_all (L) = L when atom (L)
Reverse_all ([]) = []
Reverse_all (H:T) = Append (Reverse_all (T), Reverse_all (H)) otherwise
```

b. Position:

```
Position (A, L) = Position_N (A, L, 0)

Position_N (A, (A:L), N) = N + 1
Position_N (A, (B:L), N) = Position_N (A, L, N + 1)
Position_N (A, [], N) = 0
```

c. Set:

```
Set ([]) = []
Set (H:T) = Include (H, Set (T))

Include (A, []) = [A]
Include (A, A:L) = A:L
Include (A, B:L) = prefix (B, Include (A, L))
```

d. Frequency:

```
Frequency L = F ([], L)
```

```
F (L, []) = L
F (L, H:T) = F (Corrector (H, L), T)

Corrector (A, []) = [A:1]
Corrector (A, (A:N):T) = prefix ((A:N + 1), T)
Corrector (A, H:T) = prefix (H, Corrector (A, T))
```

2. სავარჯიშო 1–ის ყველა ფუნქციისთვის შეიძლება ავაგოთ განმარტებები დამგროვებელი პარამეტრებით. მეორეს მხრივ, შესაძლოა ზოგიერთი ახლადგანმარტებული ფუნქცია არ იყოს ოპტიმიზირებული.

a. Power_A:

```
Power_A (X, N) = P (X, N, 1)

P (X, 0, A) = A
P (X, N, A) = P (X, N - 1, X * A)
```

b. Summ_T_A:

```
Summ_T_A (N) = ST (N, 0)

ST (0, A) = A
ST (N, A) = ST (N - 1, N + A)
```

c. Summ_P_A:

```
Summ_P_A (N) = SP (N, 0)

SP (0, A) = A
SP (N, A) = SP (N - 1, Summ_T_A (N) + A)
```

d. Summ_Power_A:

```
Summ_Power_A (N, P) = SPA (N, P, 0)

SPA (N, 0, A) = A
SPA (N, P, A) = SPA (N, P - 1, (1 / Power_A (N, P)) + A)
```

e. Exponent_A:

```
Exponent_A (N, P) = E (N, P, 0)

E (N, 0, A) = A
E (N, P - 1, (Power_A (N, P) / Factorial_A (P)) + A)
```

4. Haskell ენის საფუძვლები

გავეცნოთ Haskell ენის სინტაქსს. განვიხილოთ ენის ყველა მნიშვნელოვანი ცნება, მათი შესაბამისობა აბსტრაქტულ ფუნქციონალური ენის ცნებებთან. ასევე, არსებული ტრადიციების შესაბამისად, მოვიყვანოთ მაგალითები Lisp-ზე.

მონაცემთა სტრუქტურები და მათი ტიპები

პროგრამირების ნებისმიერი ენის ძირითადი ბაზური ელემენტი არის სიმბოლო (ლექსემი). სიმბოლოს ტრადიციულად უწოდებენ ასოების, ციფრებისა და სპეციალური ნიშნების შეზღუდული ან შეუზღუდავი სიგრძის თანმიმდევრობას. ზოგიერთ ენაში დიდი და პატარა ასოები განსხვავდება, ზოგიერთში - არა. Lisp-ში - არ განსხვავდება, Haskell-ში - განსხვავება არის.

სიმბოლოები ყველაზე ხშირად გამოდის იდენტიფიკატორების როლში, როგორიცაა მუდმივების (კონსტანტების), ცვლადების, ფუნქციების სახელები. მუდმივების, ცვლადების და ფუნქციების მნიშვნელობები კი არის ნიშნაკების ტიპიზიტებული თანმიმდევრობა. ასე, რომ რიცხვითი კონსტანტის მნიშვნელობა შეიძლება იყოს ასოების სტრიქონი და ა.შ. ფუნქციონალურ ენებში არსებობს ბაზური განმარტება - ატომი. რეალიზაციებში ატომებს უწოდებენ სიმბოლოებს და ციფრებს, ამასთან, რიცხვი შეიძლება იყოს სამი სახის: მთელი, ფიქსირებული და მცოცავი მძიმით.

ფუნქციონალური პროგრამირების შემდეგ ცნებას წარმოადგენს სია. აბსტრაქტულ მათემატიკურ ნოტაციაში გამოიყენება სიმბოლოები [], რომლებიც გამოიყენება Haskell-შიც. Lisp-ში გამოიყენება მრგვალი ფრჩხილები — ().Lisp-ში სიის ელემენტები ერთმანეთისგან ხარვეზებით გამოიყოფა, რაც ნაკლებ თვალსაჩინოა, ამიტომაც Haskell-ში სიის ელემენტების გამოსაყოფად გადაწყდა მძიმის (,) გამოყენება. ასე, რომ სია [a, b, c] სწორი ჩანაწერია Haskell-ის სინტაქსის შესაბამისად. Lisp-ის ნოტაციით ის ჩაიწერება როგორც (a b c). თუმცა Lisp-ის შემქმნელებმა დაუშვეს წერტილოვანი ჩაწერა წყვილებისთვის. ასე, რომ ზემოთ მოყვანილი სია ასეც შეიძლება ჩაიწეროს: (a.(b.(c.NIL))).

სიური სტრუქტურები Lisp-შიც და Haskell-შიც აღიწერება ნოტაციის შესაბამისად - ერთი სია შეიცავს მეორეს.

როგორც შესავალში იყო აღნიშნული, ფუნქციონალურ ენებში მონაცემთა ტიპები განისაზღვრება ავტომატურად. ტიპების ავტომატურად განსაზღვრის მექანიზმი დევს Haskell-შიც. თუმცა, ზოგიერთ შემთხვევებში აუცილებელია ცხადად მიუთითოთ ტიპი, წინააღმდეგ შემთხვევაში შესაძლოა ინტერპრეტატორმა

ვერ განსაზღვროს ტიპი (ხშირ შემთხვევებში გამოდის შეტყობინება ან შეცდომა). Haskell-ში გამოიყენება სპეციალური სიმბოლო ::(ორი ორიწერტილი), რომელიც ასე იკითხება: „აქვს ტიპი“, ანუ, თუ დავწერთ:

```
5 :: Integer
```

ეს წაიკითხება ასე: „რიცხვით კონსტანტას 5-ს აქვს ტიპი Integer (მთელი რიცხვი)“.

თუმცა Haskell მხარს უჭერს ისეთ გამორჩეულ საშუალებას, როგორიცაა პოლიმორფული ტიპები, ანუ ტიპების შაბლონებს. თუ, მაგალითად, ჩავწერთ [a], ეს აღნიშნავს, რომ ტიპს „ნებისმიერი ტიპის ატომების სია“, ამასთან, ატომების ტიპი უნდა იყოს ერთიდაიგივე მთელი სიისთვის. მაგალითად, სიებს: [1, 2, 3] და ['a', 'b', 'c'] ექნება ტიპი [a], ხოლო სია [1, 'a'] იქნება სხვა ტიპის. ამ შემთხვევაში ჩანაწერში [a] სიმბოლო a-ს აქვს ტიპური ცვლადის მნიშვნელობა.

შეთანხმებები დასახელებებში

Haskell-ში მნიშვნელოვანია შეთანხმებები დასახელებებში, ვინაიდან ისინი ცხადად შედის ენის სინტაქსში (რაც, საზოგადოდ, არ არის იმპერატიულ ენებში). ყველაზე მთავარი შეთანხმებაა - იდენტიფიკატორის დასაწყისში დიდი ასოს გამოყენება. ტიპების სახელები, მათ შორის პროგრამისტის მიერ განსაზღვრული, უნდა იწყებოდეს დიდი ასოთი. ფუნქციების, ცვლადებისა და მუდმივების სახელები უნდა იწყებოდეს პატარა ასოთი. იდენტიფიკატორის პირველი სიმბოლო შეიძლება იყოს სპეციალური ნიშანი, რომელთაგან ზოგიერთი ცვლის მის სემანტიკას.

სიებისა და მათემატიკური თანმიმდევრობების განსაზღვრა

Haskell, სამწუხაროდ, ერთადერთი პროგრამირების ენაა, რომელიც შესაძლებლობას გაძლევს მარტივად და სწრაფად მოვახდინოთ სიების კონსტრუირება, რომლებიც განსაზღვრულია მარტივი ფორმულით. იგი ჩვენ უკვე გამოვიყენეთ სიის ჰუარეს სწრაფი დახარისხების მეთოდის დემონსტაციისას. (მაგალითი 3). სიების განსაზღვრის ყველაზე ზოგადი სახე ასეთია:

```
[ x | x <- xs ]
```

ეს ჩანაწერი შეიძლება ასე წავიკითხოთ: „ყველა ისეთი x-ის სია, რომელიც აღებულია xs-დან“. სტრუქტურას „x xs“ უწოდებენ გენერატორს. ასეთი გენერატორის შემდეგ (ის უნდა იყოს მხოლოდ ერთი და იდგეს პირველ ადგილას

სიის განსაზღვრის ჩანაწერში) შეიძლება იყოს დაცვის რამდენიმე გამოსახულება, ერთმანეთისგან მძიმეებით გამოყოფილი. ასეთ დროს, ამოირჩევა ყველა ისეთი x, რომლებისთვისაც დაცვის ყველა გამოსახულების მნიშვნელობებიც იქნება ჭეშმარიტი. ანუ, ჩანაწერი:

```
[ x | x <- xs, x > m, x < n ]
```

შეიძლება წავიკითხოთ ასე: „ყველა ისეთი x-ის სია, აღებული xs-დან, რომ (x მეტია m-ზე) და (x ნაკლებია n-ზე)“.

Haskell-ის შემდეგი მნიშვნელოვანი თავისებურებაა უსასრულოს სიებისა და მონაცემთა სტრუქტურების მარტივი ფორმირება. უსასრულო სიები შეიძლება ფორმულირდეს როგორც განსაზღვრული სიების საფუძველზე, ასევე სპეციალური ნოტაციის საშუალებით. მაგალითად, ქვემოთ მოყვანილია უსასრულო სია, რომელიც ნატურალური რიცხვებისგან შედგება. მეორე სია წარმოადგენს კენტი ნატურალური რიცხვების სიას:

```
[1, 2 ..]
[1, 3 ..]
```

ორი წერტილის საშუალებით ასევე შესაძლებელია განისაზღვროს ნებისმიერი არითმეტიკული პროგრესია როგორც სასრული, ისე უსასრულო. თუ პროგრესია სასრულია, მაშინ მოიცემა პირველი და ბოლო ელემენტები. პროგრესიის სხვაობა გამოითვლება მოცემული მეორე და პირველი ელემენტის სხვაობით. ზემოთ მოყვანილ მაგალითებში პირველი პროგრასიის სხვაობაა 1, მეორისა - 2. ასე, რომ, თუ საჭიროა განისაზღვროს კენტი რიცხვების პროგრასია 1-დან 10-მდე, მაშინ საჭიროა ჩაიწეროს ასე: [1, 3 .. 10]. შედეგი იქნება სია [1, 3, 5, 7, 9].

მონაცემთა უსასრულო სტრუქტურა შეიძლება განისაზღვროს უსასრულო სიების საფუძველზე, ასევე შესაძლოა რეკურსიული მექანიზმების გამოყენება. ამ შემთხვევაში რეკურსია გამოიყენება როგორც რეკურსიულ ფუნქციაზე მიმართვა. მონაცემთა უსასრულო სტრუქტურების შექმნის მესამე საშუალებაა უსასრულო ტიპების გამოყენება.

მაგალითი 11. ორობითი ხეების წარმოდგენის ტიპის განსაზღვრა.

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


Branch       :: Tree a -> Tree a -> Tree a
Leaf         :: a -> Tree a
```

ამ მაგალითში ნაჩვენებია უსასრულო ტიპის განსაზღვრის საშუალება. ჩანს, რომ რეკურსიის გარეშე ეს არ მოხერხდა. თუმცა, თუ არ არის აუცილებლობა შეიქმნას მონაცემთა ახალი ტიპი, უსასრულო სტრუქტურა შეიძლება მივიღოთ ფუნქციის საშუალებით:

```
ones                = 1 : ones
numbersFrom n       = n : numberFrom (n + 1)
squares             = map (^2) (numbersFrom 0)
```

პირველი ფუნქცია განსაზღვრავს უსასრულო თანმიმდევრობას, შედგენილს მხოლოდ ერთიანებისგან. მეორე ფუნქცია აბრუნებს მთელ რიცხვებს, დაწყებულს მოცემული რიცხვიდან. მესამე ფუნქცია აბრუნებს ნატურალური რიცხვების კვადრატებს დაწყებულს ნულიდან.

ფუნქციის გამოძახებები

ფუნქციის გამოძახების მათემატიკური ნოტაცია ტრადიციულად გულისხმობდა პარამეტრის ჩასმას ფრჩხილებში. ეს ტრადიცია პრაქტიკულად ყველა იმპერატიულმა ენამ გააგრძელა. ფუნქციონალურ ენებში მიღებულია სხვა ნოტაცია - ფუნქციის სახელი გამოიყოფა მისი პარამეტრებისგან უბრალოთ ხარვეზით. Lisp-ში ფუნქცია length-ის გამოძახება მოცემული L პარამეტრით, ჩაიწერება სიის სახით: (length L). ასეთი ნოტაცია აიხსნება იმით, რომ ფუნქციონალურ ენებში ფუნქციათა უმრავლესობა კარირებულია.

Haskell-ში საჭირო არ არის ფუნქციის გამოძახება მოვათავსოთ ფრჩხილებში. მაგალითად, თუ განსაზღვრავთ ორი რიცხვის შეკრების ფუნქციას ასე:

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

მაშინ მის გამოძახებას კონკრეტული პარამეტრებით (მაგალითად, 5 და 7) ექნება სახე:

```
add 5 7
```

აქ ჩანს, რომ Haskell-ის ნოტაცია ძლიერ არის მიახლოებული აბსტრაქტული მათემატიკური ენის ნოტაციასთან. თუმცა Haskell-ში Lisp-გან განსხვავებით, არის ნოტაცია არაკარირებული ფუნქციების აღსაწერადაც, ანუ ისეთი ფუნქციების, რომელთა ტიპი არ შეიძლება წარმოდგეს სახით A1 (A2 ... (An B) ...).ეს

ნოტაცია, როგორც იმპერატიული პროგრამირების ენები, იყენებს მრგვალ ფრჩხილებს:

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

შევნიშნოთ, რომ უკანასკნელი ჩანაწერი - ეს არის ერთი არგუმენტის ფუნქცია Haskell-ის მკაცრ ნოტაციაში. მეორეს მხრივ, კარირებული ფუნქციებისთვის შესაძლებელია ნაწილობრივი გამოყენება. ანუ, ორარგუმენტიანი ფუნქციის გამოძახებისას გადავცეთ მხოლოთ ერთი არგუმენტი. ასეთი გამოძახების შედეგი იქნება ასევე ფუნქცია. ამ პროცესის საილუსტრაციოდ განვიხილოთ ფუნქცია inc, რომელიც უმატებს ერთიანს მოცემულ არგუმენტს:

```
inc    :: Integer -> Integer
inc    = add 1
```

ანუ, ამ შემთხვევაში ფუნქცია inc ერთი არგუმენტით უბრალოდ იძახებს ფუნქცია add–ს ორი არგუმენტით, რომელთაგანაც პირველია – 1. ეს არის ნაწილობრივი გამოყენების ცნების ინტუიციური განმარტება. განვიხილოთ კლასიკური მაგალითიც – ფუნქცია map (მისი აღწერა აბსტრაქტულ ფუნქციონალურ ენაზე, უკვე განვიხილეთ). აი ფუნქცია map–ის აღწერა ენა Haskell–ზე:

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

როგორც ხედავთ, აქ გამოყენებულია ოპერაცია prefix–ის ინფიქსური ჩანაწერი – ორიწერტილი, მხოლოდ ასეთი ჩანაწერი გამოიყენება Haskell–ში წყვილის წარმოდგენისას. ზემოთ მოყვანილი განმარტების შემდეგ შეიძლება მოვახდინოთ შემდეგი გამოძახება:

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

რომლის შედეგიც იქნება [2, 3, 4, 5].

λ –აღრიცხვის გამოყენება

რადგანაც პროგრამირების ფუნქციონალური პარადიგმა დაფუძნებულია λ–აღრიცხვაზე, ამიტომ ბუნებრივია, რომ ყველა ფუნქციონალური ენა მხარს უჭერს ნოტაციას λ–აბსტრაქციის წარმოსადგენად. Haskell–ში შესაძლებელია ფუნქციის λ–აბსტრაქციის საშუალებით აღწერა. ამას გარდა, λ–აბსტრაქციის საშუალებით შესაძლებელია ანონიმური ფუნქციის აღწერა (მაგალითად, ერთეული

გამოძახებისათვის). ქვემოთ მოყვანილია მაგალითი, სადაც განსაზღვრულია ფუნქციები `add` და `inc` λ-აღრიცხვის საშუალებით.

მაგალითი 12. ფუნქციები `add` და `inc`, განსაზღვრული λ-აბსტრაქციით.

```
add   = \x y -> x + y
inc   = \x -> x + 1
```

მაგალითი 13. ანონიმური ფუნქციის გამოძახება.

```
cubes = map (\x -> x * x * x) [0 ..]
```

მაგალითი `13` გვიჩვენებს ანონიმური ფუნქციის გამოძახებას, რომელსაც გადაცემული პარამეტრი აჰყავს კუბში. ამ ინსტრუქციის შესრულების შედეგი იქნება მთელი რიცხვების კუბების უსასრულო სია, დაწყებული ნულიდან. აუცილებელია ავღნიშნოთ, რომ `Haskell`–ში გამოიყენება λ-გამოსახულების ჩაწერის გამარტივებული საშუალება, რადგანაც ფუნქცია `add` ზუსტ ნოტაციაში სწორი იყო დაგვეწერა ასე:

```
add   = \x -> \y -> x + y
```

შევნიშნოთ, რომ λ–აბსტრაქციის ტიპი განისაზღვრება აბსოლუტურად ისევე, როგორც ფუნქციის ტიპი. `λx.expr` სახის λ–გამოსახულების ტიპი ასე გამოიყენება `T1 -> T2`, სადაც `T1` – არის ცვლადი `x`–ის ტიპი, ხოლო `T2` – `expr` გამოსახულების ტიპი.

ფუნქციის ჩაწერის ინფიქსური ფორმა

ზოგიერთი ფუნქცია შესაძლოა ჩაიწეროს ინფიქსური ფორმით. ეს ოპერაციები, როგორც წესი, მარტივი ბინარული ოპერაციებია. მაგალითად, ასე წარმოდგება სიების კონკატენაციისა და ფუნქციების კომპოზიციის ოპერაციები:

მაგალითი 14. სიების კონკატენაციის ინფიქსური ოპერაცია.

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

მაგალითი 15. ფუნქციების კომპოზიციის ინფიქსური ოპერაცია

```
(.)    :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

რადგანაც ინფიქსური ოპერაციები წარმოადგენს Haskell–ის ფუნციებს, ანუ ისინი კარირებულია, ამიტომ შესაძლებელია ასეთი ფუნქციების ნაწილობრივი გამოყენება. ამ მიზნისთის Haskell–ში არსებობს სპეციალური ჩანაწერი, რომელსაც უწოდებენ „სექციას“.

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

ზემოთ მოყვანილია სამი სექცია. თითოეული მათგანი განსაზღვრავს სიების კონკატენაციის ინფიქსურ ოპერაციას მასზე გადაცემული არგუმენტების რაოდენობის შესაბამისად. სექციის ჩანაწერში მრგვალი ფრჩხილების გამოყენება არის სავალდებულო.

თუ რომელიმე ფუნქცია იღებს ორ პარამეტრს, ასევე შეიძლება მისი ჩაწერა ინფიქსური ფორმით. თუმცა პარამერტებს შორის ფუნქციის სახელი აუცილებელია ჩაიწეროს სიმბოლოთი ` (შებრუნებული აპოსტროფი).

ახლად განსაზღვრული ინფიქსური ოპერაციისთვის შესაძლოა მოცემული იყოს გამოთვლის რიგი. ამისთვის Haskell–ში არის დარეზერვირებული სიტყვა infixr, რომელიც განუსაზღვრავს ოპერაციას მის ნიშნადობას (შესრულების რიგს) ინტერვალში 0–დან 9–მდე. ამასთან, 9 არის ყველაზე მაღალნიშნადი ოპერაცია. (ამ ინტერვალში შედის რიცხვი 10, რომლითაც აღინიშნება ოპერაცია „გამოყენება“). მაგალითებში 14 და 15 განსაზღვრული ოპერაციების ხარისხები ასე განისაზღვრება:

```
infixr 5 ++
infixr 9 .
```

შევნიშნოთ, რომ Haskell–ში ყველა ფუნქცია არის არამკაცრი, ვინაიდან ყველა მათგანი მხარს უჭერს გადატანილ გამოთვლებს. მაგალითად, თუ ფუნქცია ასეა განსაზღვრული:

```
bot    = bot
```

ასეთი ფუნქციის გამოძახებისას ხდება შეცდომა და ჩვეულებრივ, ასეთი შეცდოემბი ძნელი აღმოსაჩენია. მაგრამ თუ არის რომელიღაც კონსტანტური ფუნქცია, რომელიც ასეა განსაზღვრული:

```
constant_1 x = 1
```

მაშინ კონსტრუქციის (constant_1 bot) გამოძახებისას არანაირი შეცდომა არ მოხდება, რადგანაც ამ შემთხვევაში ფუნქცია bot არ გამოითვლება (გამოთვლა

გადატანილია, გამოსახულება გამოითვლება მხოლოდ მაშინ, როცა ნამდვილად მოითხოვება). გამოთვლის შედეგი, რა თქმა უნდა, არის რიცხვი1.

სავარჯიშოები

1.შეადგინეთ შემდეგი სასრული სიები (N — სიებში ელემენტების რაოდენობაა). ამისთვის გამოიყენეთ ან სიების გენერატორი ან კონსტრუქტორი ფუნქციები:

a. ნატურალური რიცხვების სია. N = 20.

b. კენტი ნატურალური რიცხვების სია. N = 20.

c. ლუწი ნატურალური რიცხვების სია. N = 20.

d. ორის ხარისხების სია. N = 25.

e. სამის ხარისხების სია. N = 25.

f. ფერმას სამკუთხა რიცხვების სია. N = 50.

g. ფერმას პირამიდალური რიცხვების სია. N = 50.

2. შეადგინეთ შემდეგი უსასრულო სიები. ამისთვის გამოიყენეთ ან სიების გენერატორი ან კონსტრუქტორი ფუნქციები:

a. ფაქტორიალების სია.

b. ნატურალური რიცხვების კვადრატების სია.

c. ნატურალური რიცხვების კუბების სია.

d. ხუთიანის ხარისხების სია.

e. ნატურალური რიცხვების მეორე სუპერხარისხების სია.

პასუხები თვითშემოწმებისთვის

1. სასრული სიები აიგება ან შეზღუდვების საშუალებით, რომლებიც ჩაიდება სიების გენერატორში ან დამატებითი შემზღუდავი პარამეტრების გამოყენებით.

a. [1 .. 20]

b. [1, 3 .. 40] ან [1, 3 .. 39]

c. `[2, 4 .. 40]`

d. 2-ის ხარისხების სია ყველაზე მარტივად შეიძლება აიგოს ფუნქციის საშუალებით (აქ `reverse` — სიის შეტრიალების ფუნქციაა):

```
powerTwo 0 = []
powerTwo n = (2 ^ n) : powerTwo (n - 1)

reverse (powerTwo 25)
```

e. 3-ის ხარისხების სია ყველაზე მარტივად შეიძლება აიგოს ფუნქციის საშუალებით (აქ `reverse` — სიის შეტრიალების ფუნქციაა):

```
powerThree 0 = []
powerThree n = (3 ^ n) : powerThree (n - 1)

reverse (powerThree 25)
```

f. წინა ორი სავარჯიშოსგან განსხვავებით, აქ შეიძლება გამოყენებული იყოს ფუნქცია `map`, რომელიც მოცემულ ფუნქციას იყენებს სიის ყველა ელემენტზე:

```
t_Fermat 1 = 1
t_Fermat n = n + t_Fermat (n - 1)

map t_Fermat [1 .. 50]
```

g. ფერმას პირამიდალური `50` რიცხვის სიის აგებაც ასევე დაფუძნებულია `map` ფუნქციის გამოყენებაზე:

```
p_Fermat 1 = 1
p_Fermat n = t_Fermat n + p_Fermat (n - 1)

map p_Fermat [1 .. 50]
```

2. უსასრულო სიები აიგება ან შეუზღუდავი გენერატორების საშუალებით, ანდა კონსტრუქტორი ფუნქციების საშუალებით პარამეტრების შეზღუდვების გარეშე.

a. ფაქტორიალების უსასრულო სია:

```
numbersFrom n = n : numbersFrom (n + 1)
```

```
factorial n = f_a n 1

f_a 1 m = m
f_a n m = f_a (n - 1) (n * m)

map factorial (numbersFrom 1)
```

b. ნატურალური რიცხვის კვადრატების უსასრულო სია:

```
square n = n * n

map square (numbersFrom 1)
```

c. ნატურალური რიცხვის კუბების უსასრულო სია:

```
cube n = n ^ 3
map cube (numbersFrom 1)
```

d. ხუთის ხარისხების უსასრულო სია:

```
powerFive n = 5 ^ n

map powerFive (numbersFrom 1)
```

e. ნატურალური რიცხვების მეორე სუპერხარისხების უსასრულო სია:

```
superPower n 0 = n
superPower n p = (superPower n (p - 1)) ^ n

secondSuperPower n = superPower n 2
map secondSuperPower (numbersFrom 1)
```

5. Haskell–ის სინტაქსი და მოსამსახურე სიტყვები

ჩვენ უკვე ავღნიშნეთ, რომ Haskell ენის სინტაქსი ძალზე წააგავს აბსტრაქტული ფუნქციონალური ენის სინტაქსს. Haskell–ის მკვევარებმა მიზნად დაისახეს იმ დროისთვის არსებული ფუნქციონალური ენების საუკეთესო თვისებების თავმოყრა ერთ ენაში და ამავე დროს ცუდის უარყოფა, რაც მათ მოახერხეს კიდეც...

ქვემოთ განვიხილავთ მოსამსახურე სიტყვებს, რომლებიც აქამდე არ გამოგვიყენებია. ასევე განვიხილავთ იმ ახალ შესაძლებლობებს, რაც ფუნქციონალურ პარადიგმაში შევიდა იმის გამო, რომ Haskell–შია რეალიზებული.

დაცვა და ლოკალური ცვლადები

როგორც უკვე ვაჩვენეთ, დაცვა და ლოკალური ცვლადები გამოიყენება ფუნქციონალურ პროგრამირებაში მხოლოდ ჩანაწერის გასამარტივებლად და ტექსტის გასაგებად. Haskell–ის სინტაქსში არის სპეციალური საშუალება, რაც უზრუნველყოფს დაცვის ორგანიზებასა და ლოკალური ცვლადების გამოყენებას.

ფუნქციის განსაზღვრისას დაცვის მექანიზმი მიეთითება ვერტიკალური ხაზის სიმბოლოს " | "–ის გამოყენებით:

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

ამ მაგალითში ფუნქცია sign იყენებს სამ დამცავ კონსტრუქციას, რომლებიდანაც თითოეული გამოიყოფა წინა განსაზღვრებისგან ვერტიკალური ხაზით. ასეთი კონსტრუქცია შეიძლება იყოს ნებისმიერი რაოდენობის. მათი გარჩევა განხორციელდება, ბუნებრივია, თანმიმდევრობით ზემოდან ქვემოთ და თუ არსებობს არაცარიელი გადაკვეთა დაცვის განსაზღვრებაში, მაშინ იმუშავებს ის კონსტრუქცია, რომელიც არის უფრო ადრე (მაღლა) ფუნქციის განსაზღვრების ჩანაწერში.

რათა განმარტივდეს პროგრამის დაწერა და გახდეს იგი უფრო წაკითხვადი და გასაგებად მარტივი იმ შემთხვევაში, როცა ფუნქციის განმარტება დაწერილია დიდი რაოდენობით კლოზების გამოყენებით, Haskell–ში არსებობს გასაღები სიტყვა „case“. ამ სიტყვის გამოყენებით შესაძლოა ფუნქციის განსაზღვრისას კლოზები არ ჩავწეროთ ისე, როგორც ეს „წმინდა“ ფუნქციონალურ ენაშია მიღებული, არამედ

შევამციროთ ჩანაწერი. ფუნქციის განმარტების ზოგადი სახე გასაღები სიტყვა „case“–ის გამოყენებით შენდეგია:

Function X_1 X_2 ... X_k = **case** (X_1, X_2, ..., X_k) **of**

(P_{11}, P_{21}, ..., P_{k1}) -> $Expression_1$

...

(P_{1n}, P_{2n}, ..., P_{kn}) -> $Expression_n$

ზემოთ მოყვანილ აღწერაში გამუქებულია ენის გასაღები სიტყვები და სიმბოლოები.

ამრიგად, ფუნქცია, რომელიც აბრუნებს მოცემული სიის პირველ n ელემენტს, შეიძლება განისაზღვროს გასაღები სიტყვა „case“–ის გამოყენებით შემდეგნაირად:

```
takeFirst n l =    case (n, l) of
            (0, _) -> []
            (_, []) -> []
            (n, (x:xs)) -> (x) : (takeFirst (n - 1) xs)
```

ასეთი ჩანაწერი იქნება ფუნქციის ჩვეულებრივი განმარტების ექვივალენტური:

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

განვმარტოთ ცნება „ჩასმის ნიღაბი“. Haskell–ში ნიღაბს აღნიშნავენ ქვედა ტირე სიმბოლოთი „ _ “, ისევე, როგორც Prolog–ში. ეს სიმბოლო ცვლის ნებისმიერ ნიმუშს და წარმოადგენს თავის მხრივ ანონიმურ ცვლადს. თუ კლოზის გამოსახულებაში არ არის აუცილებელი გამოყენებული იყოს ნიმუშის ცვლადი, მაშინ შესაძლებელია იგი შეიცვალოს ჩასმის ნიღბით. ამასთან, ბუნებრივია ხდება გადატანილი გამოთვლები – ის გამოსახულება, რომელიც შეიძლება ჩაისვას ნიღბის ნაცვლად, არ გამოითვლება.

დამცავი კონსტრუქციების გამოყენების შემდეგი საშუალებაა კონსტრუქციის „if-then-else“–ის გამოყენება. Haskell–ში ეს შესაძლებლობა რეალიზებულია. ფორმალურად, ეს კონსტრუქცია შეიძლება მარტივად გადავიდეს გამოსახულებაში გასაღები სიტყვა „case“–ის გამოყენებით. შეიძლება ჩავთვალოთ, რომ გამოსახულება:

```
if Exp1 then Exp2 else Exp3
```

ჩაწმოადგენს შემდეგი გამოსახულების შემოკლებას:

```
case (Exp1) of
```

```
      (True) -> Exp2
      (False) -> Exp3
```

ბუნებრივია, რომ Exp1–ის ტიპი უნდა იყოს Boolean (Bool Haskell–ში, ხოლო გამოსახულებების Exp2–ის და Exp3–ის ტიპები უნდა ემთხვეოდეს ერთმანეთს (სწორედ, მათი მნიშვნელობები უნდა დაბრუნდეს „if-then-else“ კონსტრუქციით).

ლოკალური ცვლადების გამოსაყენებლად Haskell–ში არსებობს ჩანაწერის ორი სახე. პირველი სრულად შეესაბამებს უკვე განსაზღვრულ მათემატიკურ ნოტაციას:

```
let   y = a * b
      f x = (x + y) / y
in f c + f d
```

ლოკალური ცვლადის განსაზღვრის სხვა საშუალებაა მისი აღწერა გამოყენების შემდეგ. ამ შემთხვევაში გამოიყენება გასაღები სიტყვა „where“, რომელიც ჩაისმის გამოსახულების ბოლოს.

```
f x y | y > z      = y - z
      | y == z     = 0
      | y < z      = z - y
      where z = x * x
```

როგორც დავინახეთ, Haskell მხარს უჭერს ლოკალური ცვლადის განმარტების ორ საშუალებას – პრეფიქსულს (გასაღები სიტყვა „let“–ის საშუალებით) და პოსტფიქსულს (გასაღები სიტყვა „where“–ის საშუალებით). ორივე საშუალება თანაბარმნიშვნელოვანია, მათი გამოყენება მხოლოდ პროგრამისტის ჩვევაზეა დამოკიდებული. თუმცა, ჩვეულებრივ, პოსტფიქსური ჩანაწერი გამოიყენება გამოსახულებებში, სადაც არის დაცვა, მაშინ, როცა პრეფიქსული – ყველა დანარჩენ შემთხვევაში.

პოლიმორფიზმი

როგორც უკვე ავღნიშნეთ, პროგრამირების ფუნქციონალური პარადიგმა მხარს უჭერს წმინდა ან პარამეტრიზირებულ პოლიმორფიზმს. თუმცა თანამედროვე ენების უმრავლესობა მხარს უჭერს პოლიმორფიზმს ad hoc ანუ გადატვირთვას.

მაგალითად, გადატვირთვა პრატიკულად მუდმივად გამოიყენება შემდეგი მიზნებისთვის:

- ლიტერალები `1`, `2`, `3` და ა.შ. (ანუ ციფრები) გამოიყენება როგორც მთელი რიცხვების ჩასაწერად, ასევე ნებისმიერი სიზუსტის რიცხვების ჩასაწერად.
- არითმეტიკული ოპერაციები (მაგალითად, შეკრება – ნიშანი „+“) გამოიყენება სხვადასხვა ტიპის მონაცემებთან (მათ შორის – არარიცხვითთანაც, მაგალითად, სტრიქონების კონკატენაციისთვის).
- შედარების ოპერაციები (`Haskell`–ში ორმაგი ტოლობის ნიშანი ”==“) გამოიყენება სხვადასხვა ტიპის მონაცემების შესადარებლად.

ოპერაციის გადატვირთული მოქმედება განსხვავებულია სხვადასხვა ტიპისთვის, მაშინ როცა პარამეტრიზებული პოლიმორფიზმისას მონაცემთა ტიპი არ არის მნიშვნელოვანი. `Haskell`–ში არის მექანიზმი გადატვირთის გამოსაყენებლად.

`ad hoc` პოლიმორფიზმის გამოყენების შესაძლებლობის განხილვა ყველაზე ადვილია შედარების ოპერაციის მაგალითზე. არსებობს ტიპების დიდი სიმრავლე, რომელთათვისაც შესაძლებელია და მიზანშეწონილია პოლიმორფიზმის გამოყენება, მაგრამ ზოგიერთი ტიპისთვის ეს ოპერაცია უსარგებლოა. მაგალითად, ფუნქციის შედარებები უაზრობაა, ფუნქცია აუცილებლად უნდა გამოითვალოს და შედარდეს ამ გამოთვლების შედეგები. თუმცა, მაგალითად, შეიძლება გახდეს აუცილებლობა შედარდეს სიები. ცხადია, აქ საუბარია სიების ელემენტების მნიშვნელობების შედარებაზე და არა მათი მიმთითებლების შედარებაზე (როგორც ეს გაკეთებულია `Java`–ში).

განვიხილოთ ფუნქცია `element`, რომელიც აბრუნებს ჭეშმარიტ მნიშვნელობას იმის შესაბამისად, თუ არის მოცემული ელემენტი მოცემულ სიაში. მოცემული ფუნქციის აღწერა ინფიქსური ფორმითაა.

```
x `element` []     = False
x `element` (y:ys) = (x == y) || (x `element` ys)
```

აქ ჩანს, რომ ფუნქცია `element`–ს აქვს ტიპი `(a -> [a] -> Bool)`, მაგრამ ოპერაცია „==“–ის ტიპი უნდა იყოს `(a -> a -> Bool)`. ვინაიდან ტიპის ცვლადმა შეიძლება აღნიშნოს ნებისმიერი ტიპი (მათ შორის სიაც), მიზანშეწონილია გადავსაზღვროთ ოპერაცია „==“ ნებისმიერ ტიპთან სამუშაოდ.

ტიპების კლასები ამ პრობლემის გადაწყვეტაა `Haskell`–ში. რათა განვიხილოთ ეს მექანიზმი მოქმედებაში, განვსაზღვროთ კლასის ტიპი, რომელიც შეიცავს ტოლობის ოპერატორს.

```
class Eq a where
      (==)   :: a -> a -> Bool
```

ამ კონსტრუქციაში გამოიყენება მოსამსახურე სიტყვები „class“ და „where“, ასევე ტიპის ცვლადი a. სიმბოლო „Eq“ წარმოადგენს განსაზღვრული კლასის სახელს. ეს ჩანაწერი შეიძლება ასე წავიკითხოთ: „ტიპი a არის Eq კლასის ეგზემპლარი, თუ ამ ტიპისთვის გადატვირთულია შესაბამისი ტიპის შედარების ოპერაცია „==“. აუცილებელია შევნიშნოთ, რომ შედარების ოპერაცია უნდა იყოს განსაზღვრული ერთიდაიგივე ტიპის ობიექტების წყვილზე.

იმ ფაქტის აღნიშვნა, რომ ტიპი a უნდა იყოს Eq კლასის ელემენტი, ჩაიწერება ასე (Eq a). ამიტომაც ჩანაწერი (Eq a) არ წარმოადგენს ტიპის აღწერას, ის აღნიშნავს შეზღუდვას, რომელიც ედება a ტიპზე, და ამ შეზღუდვას Haskell–ში უწოდებენ კონტექსტს. კონტექსტები იწერება ტიპების განმარტების წინ და გამოიყოფა მისგან სიმბოლოებით „ => “:

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

ეს ჩანაწერი შეიძლება ასე წავიკითხოთ: „ყოველი a ტიპისთვის, რომელიც არის Eq კლასის ეგზემპლიარი, განსაზღვრულია ოპერაცია „==“, რომელსაც აქვს ტიპი (a -> a -> Bool)”. ეს ოპერაცია უნდა იყოს გამოყენებული ფუნქციაში element, ამიტომ შეზღუდვა ვრცელდება მასზეც. ამ შემთხვევაში აუცილებელია ცხადად მივუთითოთ ფუნქციის ტიპი:

```
element      :: (Eq a) => a -> [a] -> Bool
```

ამ ჩანაწერით ხდება იმის განაცხადი, რომ ფუნქცია element განსაზღვრულია არა მონაცემთა ყველა ტიპისთვის, არამედ მხოლოდ მათთვის, რომელთათვისაც განსაზღვრულია შესაბამისი შედარების ოპერაცია.

თუმცა, ეხლა ჩნდება იმის განსაზღვრის პრობლემა, თუ რომელი ტიპებია Eq კლასის ეგზემპლარები. ამისთვის არსებობს გასაღები სიტყვა „instance“. მაგალითად, იმის მისაწერად, რომ ტიპი Integer წარმოადგენს Eq კლასის ეგზემპლიარს, აუცილებელია დავწეროთ:

```
instance Eq Integer where
      x == y = x `integerEq` y
```

ამ გამოსახულებაში ოპერაცია „==“ –ის განსაზღვრას უწოდებენ მეთოდის განსაზღვრას (როგორც ეს ობიექ–ორიენტირებულ პარადიგმაშია). ფუნქცია integerEq შეიძლება იყოს ნებისმიერი (და არა მხოლოდ ინფიქსური), მთავარია, რომ მას ჰქონდეს ტიპი (a a Bool). ამ შემთხვევას ყველაზე მეტად მიესადაგება ორი ნატურალური რიცხვის შედარების პრიმიტიული ფუნქცია. თავის მხრივ, დაწერილი გამოსახულება შეიძლება ასე წავიკითხოთ: „ ტიპი Integer წარმოადგენს Eq კლასის ეგზემპლიარს, ხოლო შემდეგ მოდის მეთოდის აღწერა, რომელიც ადარებს ორ Integer ტიპის ობიექტს“.

ამრიგად, შესაძლოა შედარების ოპერაცია განისაზღვროს უსასრულო რეკურსიული ტიპებისთვისაც. მაგალითად, უკვე განხილული Tree ტიპისთვის განსაზღვრებას ექნება შემდეგი სახე:

```
instance (Eq a) => Eq (Tree a) where
      Leaf a == Leaf b                  = a == b
      (Branch l1 r1) == (Branch l2 r2)= (l1 == l2) && (r1 == r2)
      _ == _                             = False
```

ბუნებრივია, თუ ენაში განსაზღვრულია კლასის ცნება, მაშინ უნდა იყოს განსაზღვრული მემკვიდრეობითობის კონცეფცია. თუმცა Haskell–ში კლასის ქვეშ გაიგება უფრო აბსტრაქტული რამ, ვიდრე ჩვეულებრივად ობიექტ–ორიენტირებულ ენებში, მაგრამ Haskell–ში ასევე არის მემკვიდრეობითობა. ამავე დროს მემკვიდრეობითობის კონცეფცია ისევე დახვეწილად არის განმარტებული, როგორც კლასი. მაგალითად, ზემოთ განმარტებული Eq კლასიდან მემკვიდრეობით შეიძლება მივიღოთ კლასი Ord, რომელიც წარმოადგენს მონაცემთა შედარებით ტიპებს. მისი განმარტება გამოიყურება შემდეგნაირად:

```
class (Eq a) => Ord a where
      (<), (>), (<=), (>=)      :: a -> a -> Bool
      min, max                  :: a -> a -> a
```

Ord კლასის ყველა ეგზემპლიარი განსაზღვრავს გარდა ოპერაციებისა „ნაკლებია“, „მეტია“, „ნაკლების და ტოლია“, „მეტია და ტოლია“, „მინიმუმ“ და „მაქსიმუმ“, კიდევ შედარების ოპერაციას „==“, რადგანაც მისი კლასი Ord მემკვიდრეობითაა მიღებული Eq კლასიდან.

ყველაზე გასაოცარია ის, რომ Haskell მხარს უჭერს მრავლობით მემკვიდრეობას. თუ ვიყენებთ რამდენიმე ბაზურ კლასს, მათ უბრალოდ შესაბამის სექციაში ჩამოვთლით (მძიმეებით გამოვყოფთ). ამასთან, კლასის ეგზემპლარები, რომლებიც რამდენიმე ბაზური კლასიდან მემკვიდრეობით არის მიღებული, ცხადია მხარს უჭერს ბაზური კლასების ყველა მეთოდს.

შედარება სხვა ენებთან

თუმცა კლასები არსებობს პროგრამირების მრავალ ენაში, Haskell–ში კლასის ცნრბა რამდენადმე განსხვავებულია.

- Haskell–ში გაყოფილია კლასის განსაზღვრა მისი მეთოდების განსაზღვრისაგან, მაშინ როცა, ისეთი ენები, როგორცაა C++ და Java

ერთად განსაზღვრავს მონაცემთა სტრუქტურას და მისი დამუშავების მეთოდებს.

- მეთოდების განსაზღვრა Haskell–ში შეესაბამება ვირტუალურ ფუნციებს C++–ში. კლასის თითოეულმა ეგზემპლიარმა უნდა გადასაზღვროს კლასის მეთოდები.
- ყველაზე მეტად Haskell–ის კლასები წააგავს Java–ს ინტერფეისებს. როგორც ინტერფეისის განსაზღვრება, კლასებიც Haskell–ში წარმოადგენს ობიექტების გამოყენების ოქმებს თვითონ ობიექტების განსაზღვრის ნაცვლად.
- Haskell მხარ არ უჭერს ფუნქციების გადატვირთვას, რომელიც C++–შია გამოყენებული, როცა ერთიდაიგივე სახელის ფუნქციები დასამუშავებლად ღებულობს სხვადასხვა ტიპის მონაცემებს.
- ობიექტების ტიპი Haskell–ში არ შეიძლება არაცხადად იყოს გამოყვანილი. Haskell–ში არ არსებობს ბაზური კლასი ყველა კლასისთვის (ისეთი, როგორიცაა, მაგალითად, TObject კლასი ენა Object Pascal–ში).
- C++ და Java კომპილირებულ კოდს უმატებს იდენტიფიცირებულ ინფორმაციას (მაგალითად, ვირტუალური ფუნქციების განლაგების ცხრილებს). Haskell–ში ასე არ არის. ინტერპრეტაციის (კომპილაციის) დროს ყველა ინფორმაცია გამოდის ლოგიკურად.
- არ არსებობს შეღწევადობაზე კონტროლის ცნება – არ არის ღია და დახურული მეთოდები. ამის ნაცვლად Haskell გვთავაზობს პროგრამების მოდულარიზაციის მექანიზმს.

სავარჯიშოები

1. ჩაწერეთ Haskell–ის ნოტაციით ფუნქციები, რომლებიც მუშაობს სიებთან. შესაძლებლობისამებრ გამოიყენეთ დაცვისა და ლოკალური ცვლადების ფორმალიზმები.

a. getN – ფუნქცია მოცემული სიიდან N-ური ელემენტის გამოსაყოფად.

b. listSumm – ორი სიის შეკრების ფუნქცია. აბრუნებს სიას, რომელიც შედგება სია–პარამეტრების ელემენტების ჯამისგან. გაითვალისწინეთ, რომ გადაცემული სიები შეიძლება იყოს სხვადასხვა სიგრძის.

c. oddEven – მოცემულ სიაში მეზობელი კენტი და ლუწი ელემენტების გადაადგილების ფუნქცია.

d. reverse – ფუნქცია, რომელიც აბრუნებს სიას (სიის პირველი ელემენტი ხდება ბოლო ელემენტი, მეორე–ბოლოდან მეორე და ა.შ. ბოლო ელემენტამდე).

e. map — ფუნქცია მასზე პარამეტრად გადაცემულ მეორე ფუნქციას იყენებს მოცემული სიის ყველა ელემენტთან.

2. ჩაწერეთ Haskell-ის ნოტაციაში ფუნქციები, რომლებიც მუშაობს სიებთან. აუცილებლობის შემთხვევაში ისარგებლეთ დამატებითი, ასევე წინა სავარჯიშოში განსაზღვრული ფუნქციებით. შესაძლებლობების მიხედვით გამოიყენეთ დაცვისა და ლოკალური ცვლადების ფორმალიზმები.

a. reverseAll — ფუნქცია, რომელიც შესასვლელზე იღებს სიურ სტრუქტურას და შეაბრუნებს მის ყველა ელემენტს, ასევე თავის თავსაც.

b. position — ფუნქცია, რომელიც აბრუნებს ნომერს, თუ მოცემული ატომი სიაში პირველად როდის შევა.

c. set — ფუნქცია, რომელიც აბრუნებს სიას, რომელშიც მოცემული სიის ყველა ატომი მხოლოდ ერთხელ შედის.

d. frequency — ფუნქცია, რომელიც აბრუნებს წყვილების სიას (სიმბოლო, სიხშირე). თითოეული წყვილი განისაზღვრება მოცემული სიის ატომით და ამ სიაში მისი შესვლის სიხშირით.

3. აღწერეთ ტიპების შემდეგი კლასები. აუცილებლობის შემთხვევაში ისარგებლეთ კლასების მემკვიდრეობითობის მექანიზმოთ.

a. Show — კლასი, რომლის ობიექტების ეგზემპლიარები შეიძლება შეტანილი იყოს ეკრანიდან.

b. Number — კლასი, რომელიც აღწერს სხვადასხვა ბუნების რიცხვებს.

c. String — კლასი, რომელიც აღწერს სტრიქონებს (სიმბოლოების სიებს).

4. განსაზღვრეთ ტიპები - წინა დავალებაში აღწერილი კლასების ეგზემპლიარები. შესაძლებლობების მიხედვით კლასის თითოეული ეგზემპლიარისთვის განსაზღვრეთ მეთოდები, რომლებიც მუშაობენ ამ კლასის ობიექტებთან.

a. Integer — მთელი რიცხვების ტიპი.

b. Real — ნამდვილი რიცხვების ტიპი.

c. Complex — კომპლექსური რიცხვები ტიპი.

d. WideString — სტრიქონების ტიპი, როგორც ორბაიტიანი სიმბოლოების თანმიმდევრობა UNICODE-ის კოდირებაში.

პასუხები თვითშემოწმებისთვის

1. ყველა ქვემოთ მოყვანილი აღწერა Haskell-ზე არის ერთ-ერთი მრავალი შესაძლებლობიდან:

a. getN:

```
getN        :: [a] -> a
getN n []         = _
getN 1 (h:t)      = h
getN n (h:t)      = getN (n - 1) t
```

b. listSumm:

```
listSumm                  :: Ord (a) => [a] -> [a]
listSumm [] l             = l
listSumm l []             = l
listSumm (h1:t1) (h2:t2)  = (h1 + h2) : (listSumm t1 t2)
```

c. oddEven:

```
oddEven                   :: [a] -> [a]
oddEven []                = []
oddEven [x]         = [x]
oddEven (h1:(h2:t))= (h2:h1) : (oddEven t)
```

d. reverse:

```
append       :: [a] -> [a] -> [a]
append [] l  = l
append (h:t) l2    = h : (append t l2)

reverse            :: [a] -> [a]
reverse []         = []
reverse (h:t)      = append (reverse t [h])
```

e. map:

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

```
map f (h:t)  = (f h) : (map f t)
```

2. ყველა ქვემოთ მოყვანილი აღწერა Haskell-ზე არის ერთ-ერთი მრავალი შესაძლებლობიდან:

a. reverseAll:

```
atom         :: ListStr (a) -> Bool
atom a       = True
atom _       = False

reverseAll         :: ListStr (a) -> ListStr (a)
reverseAll l = l
reverseAll []      = []
reverseAll (h:t)   = append (reverseAll t) (reverseAll h)
```

b. position:

```
position           :: a -> [a] -> Integer
position a l = positionN a l 0

positionN          :: a -> [a] -> Integer -> Integer
positionN a (a:t) n= (n + 1)
positionN a (h:t) n= positionN a t (n + 1)
positionN a [] n   = 0
```

c. set:

```
set          :: [a] -> [a]
set []       = []
set (h:t)          = include h (set t)

include            :: a -> [a] -> [a]
include a [] = [a]
include a (a:t)    = a : t
include a (b:t)    = b : (include a t)
```

d. frequency:

```
frequency           :: [a] -> [(a : Integer)]
frequency l  = f [] l

f            :: [a] -> [a] -> [(a : Integer)]
f l []       = l
f l (h:t)           = f (corrector h l) t

corrector           :: a -> [a] -> [(a : Integer)]
corrector a []      = [(a : 1)]
corrector a (a:n):t= (a : (n + 1)) : t
corrector a h:t     = h : (corrector a t)
```

3. ყველა ქვემოთ მოყვანილი აღწერა Haskell-ზე არის ერთ-ერთი მრავალი შესაძლებლობიდან:

a. Show:

```
class Show a where
      show   :: a -> a
```

b. Number:

```
class Number a where
      (+)    :: a -> a -> a
      (-)    :: a -> a -> a
      (*)    :: a -> a -> a
      (/)    :: a -> a -> a
```

c. String:

```
class String a where
      (++)          :: a -> a -> a
      length :: a -> Integer
```

4. ყველა ქვემოთ მოყვანილი აღწერა Haskell-ზე არის ერთ-ერთი მრავალი შესაძლებლობიდან:

a. Integer:

```
instance Number Integer where
      x + y  = plusInteger x y
      x - y  = minusInteger x y
      x * y  = multInteger x y
      x / y  = divInteger x y

plusInteger         :: Integer -> Integer -> Integer
plusInteger x y     = x + y

...
```

b. Real:

```
instance Number Real where
      x + y  = plusReal x y
...
```

c. Complex:

```
instance Number Complex where
      x + y  = plusComplex x y
...
```

d. WideString:

```
instance String WideString where
      x ++ y= wideConcat x y
      length x     = wideLength x

wideConcat x y     = append x y
wideLength x = length x
```

6. მოდულები და მონადები Haskell-ში

არც ერთი ენა არ არის მოდულის ცნების გარეშე, თუ არ ჩავთვლით ყველაზე დაბალი დონის ენებს და მათთაც კი, ამ ბოლო დროს შეიძინეს მაღალი დონის ენის თვისებები. მოდულის ცნება შემოვიდა იმ დროდან, როცა პროგრამირება როგორც ხელოვნება (ან ხელობა) ჯერ კიდევ ვითარდებოდა. იგი გაჩნდა პროგრამის ლოგიკურ ნაწილებად დაყოფის აუცილებლობიდან, რომელთაგანაც თითოეულს ამუშავებდა ცალკეული მომხმარებელი ან მკვლევართა კოლექტივი.

Haskell-ში ასევე არსებობს მოდულის ცნება, თუმცა ამ ენაში უფრო მეტად მნიშვნელოვანია „მონადის“ ცნება. განვიხილოთ ორივე ცნება - „მოდული“ და „მონადა“.

მოდულები

Haskell-ში მოდულებს ორმაგი დანიშნულება აქვთ. ერთის მხრივ, მოდულები აუცილებელია სახელთა არის კონტროლისთვის (ისევე, როგორც ყველა სხვა ენაში), მეორეს მხრივ, მოდულების საშუალებით შეიძლება შეიქმნას მონაცემთა აბსტრაქტული ტიპები.

მოდულის განმარტება Haskell-ში საკმაოდ მარტივია. მოდულის სახელი შეიძლება იყოს ნებისმიერი სიმბოლო, მხოლოდ სახელი იწყება დიდი ასოთი. მოდულის სახელი დამატებით არ არის დაკავშირებული ფაილურ სისტემასთან (ისე, როგორც ეს Pascal-ში და Java-შია), ანუ მოდულის შემცველი ფაილის სახელი შეიძლება იყოს არა ისეთივე, როგორც მოდულის სახელია. უფრო ზუსტად, ფაილში შეიძლება იყოს რამდენიმე მოდული, რადგანაც მოდული - ეს ყველაზე მაღალი დონის დეკლარაციაა.

როგორც ცნობილია, ზედა დონეზე Haskell-ის მოდულები შეიძლება შეიცავდეს დეკლარაციების სიმრავლეს (აღწერებსა და განსაზღვრებებს), როგორიცაა ტიპები, კლასები, მონაცემები, ფუნქციები. თუმცა დეკლარაციის ერთი სახე მოდულში უნდა იდგეს აუცილებლად პირველ ადგილას (თუ, რა თქმა უნდა, დეკლარაციის ეს სახე საერთოდ გამოიყენება). საქმე ეხება მოდულში სხვა მოდულის ჩართვას - ამისთვის გამოიყენება მოსამსახურე სიტყვა import. დანარჩენი განსაზღვრებები შეიძლება იყოს ნებისმიერი თანმიმდევრობით.

მოდულის განმარტება იწყება მოსამსახურე სიტყვით module. მაგალითად, მოდული Tree შემდეგნაირად განიმარტება:

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

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

fringe              :: Tree a -> [a]
fringe (Leaf x)             = [x]
fringe (Branch left right)      = fringe left ++ fringe right
```

ამ მოდულში აღწერილია ერთი ტიპი (`Tree` - არ არის სახიფათო, ტიპის სახელი რომ ემთხვევა მოდულის სახელს, ამ შემთხვევაში ისინი სახელთა სხვადასხვა არეში არიან) და ერთი ფუნქცია `(fringe)`. მოცემულ შემთხვევაში მოდული `Tree` ცხადად ექსპორტირებს ტიპს `Tree` (თავის ქვეტიპებთან `Leaf`-თან და `Branch`-თან ერთად) და ფუნქცია `fringe` - ამისთვის ტიპისა და ფუნქციის სახელები მითითებულია მოდულის სახელის შემდეგ ფრჩხილებში. თუ რომელიმე ობიექტის დასახელებას ფრჩხილებში არ მოვუთითებთ, მაშინ მისი ექსპორტირება არ მოხდება, ანუ ეს ობიექტი არ გამოჩნდება მიმდინარე მოდულის გარედან.

ერთი მოდულის გამოყენება მეორედან გამოიყურება კიდევ უფრო მარტივად:

```
module Main where

import Tree (Tree(Leaf, Branch), fringe)

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

მოცემულ მაგალითში ჩანს, რომ მოდული `Main` იმპორტილებს მოდულს `Tree`-ის, ამასთან `import` დეკლარაციაში ცხადად არის აღწერილი, თუ რომელი ობიექტები იმპორტირდება მოდული`Tree`–დან. თუ ამ აღწერას გამოვტოვებთ, მაშინ იმპორტირდება ყველა ობიექტი, რომელსაც მოდული ექსპორტირებს, ანუ, ამ შემთხვევაში შეგვეძლო უბრალოდ დაგვეწერა : `import Tree`.

ზოგჯერ ისე ხდება, რომ ერთი მოდული იმპორტირებს რამდენიმე სხვას (შევნიშნოთ, რომ ეს ჩვეულებრივი სიტუაციაა), მაგრამ ამასთან იმპორტირებულ მოდულში არსებობს ობიექტები ერთი და იგივე სახელებით. ბუნებრივია, რომ ამ შემთხვევაში ჩნდება სახელთა კონფლიქტი. ამის თავიდან ასაცილებლად, `Haskell`-ში არსებობს სპეციალური მოსამსახურე სიტყვა `qualified`, რომლის საშუალებითაც განისაზღვრება ის იმპორტირებული მოდულები, რომლის ობიექტის სახელები იღებს სახეს: <მოდულის სახელი>.<ობიექტის სახელი>, ანუ იმისათვის, რომ მივმართოთ მოდულიდან ობიექტს, მისი სახელის წინ აუცილებელია დავწეროთ მოდულის სახელი:

```
module Main where

import qualified Tree

main = print (Tree.fringe (Tree.Leaf 'a'))
```

ასეთი სინტაქსის გამოყენება პროგრამისტის გემოვნებაზეა, თუმცა ამით პროგრამის სიდიდე იზრდება. ზოგს მოსწონს მოკლე მნემონიმური სახელების გამოყენება და ისინი იყენებენ კვალიფიკატორებს (მოდულების სახელებს) მხოლოდ აუცილებლობის შემთხვევაში.

მონაცემთა აბსრტაქტული ტიპები

Haskell-ში მოდული ერთადერთი საშუალებაა ე.წ. მონაცემთა აბსტრაქტული ტიპების შესაქმნელად. აბსტრაქტულ ტიპებში დაფარულია ტიპის წარმოდგენა, ღიაა მხოლოდ სპეციფიური ოპერაციები შექმნილ ტიპებზე, რომელთა სიმრავლე სრულად არის საკმარისი ამ ტიპთან სამუშაოდ. მაგალითად, თუმცა ტიპი Tree საკმაოდ მარტივ ტიპს წარმოადგენს, მაინც უმჯობესია გავხადოთ იგი აბსტრაქტულ ტიპად, ანუ, დავმალოთ, რომ ის შედგება Leaf-გან და Branch-გან. ეს ასე ხორციელდება:

```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf)
where

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

leaf                = Leaf
branch              = Branch
cell (Leaf a)             = a
left (Branch l r)         = l
right (Branch l r) = r
isLeaf (Leaf _)           = True
isLeaf _                  = False
```

ჩანს, რომ შიდა ტიპ Tree-სთან პროგრამისტს შეუძლია მიაღწიოს მხოლოდ სპეციალური ფუნქციების გამოყენებით. აქედან გამომდინარე, თუ ამ მოდულის

შემქმნელი შეეცდება შეცვალოს ტიპის წარმოდგენა (მაგალითად, მოახდინოს მისი ოპტიმიზაცია), მან უნდა შეცვალოს ის ფუნქციებიც, რომლებიც მოქმედებს `Tree` ტიპის ველებზე. თავის მხრივ, თუ პროგრამისტი გამოიყენებს თავის პროგრამაში ტიპს `Tree`, მას არაფრის შეცვლა არ მოუწევს, რადგანაც პროგრამა რჩება მომუშავე.

მოდულების გამოყენების სხვა ასპექტები

შემდგომში მოყვანილი გვაქვს `Haskell`–ში მოდულების დამატებითი ასპექტები:

- იმპორტის დეკლარაციაში (`import`) შეიძლება ამორჩევით დავმალოთ ზოგიერთი ექსპორტირებული ობიექტიდან (მოსამსახურე სიტყვა `hiding`–ის საშუალებით). ეს სასარგებლოა იმპორტირებული მოდულიდან ზოგიერთი ობიექტის ცხადად ამოსაგდებად.
- იმპორტისას შეიძლება განსაზღვროთ მოდულის ფსევდონიმი მისგან ექსპორტირებული ობიექტების კვალიფიკაციისათვის. ამისთვის გამოიყენება გასაღები სიტყვა `as`. იგი გამოიყენება მოდულების სახელის შესამოკლებლად.
- ყველა პროგრამა არაცალსახად იმპორტირებს მოდულს `Prelude`. თუ გავაკეთებთ ამ მოდულის ცხადად იმპორტს, მაშინ მის დეკლარაციაში შესაძლებელია დაიფაროს ზოგიერთი ობიექტი, რათა მოხდეს შემდგომში მათი გადასაზღვრა.
- ყველა `instance` დეკლარაციები არაცხადად ექსპორტირდება და იმპორტირდება ყველა მოდულის მიერ.
- კლასის მეთოდები შეიძლება, ისევე როგორც კლასების ქვეტიპები, ჩამოითვალოს ფრჩხილებში შესაბამისი კლასის სახელის შემდეგ ექსპორტის/იმპორტის დეკლარაციის დროს.

მონადები

ფუნქციონალურ პროგრამირების ბევრი დამწყები პროგრამისტი შეცბუნებულია `Haskell`–სი მონადის ცნებით. მაგრამ მონადები ენაში ძალზე ხშირად გვხვდება, მაგალითად, შეტანა–გამოტანის სისტემა მონადის ცნებაზეა დაფუძნებული, სტანდარტული ბიბლიოთეკა შეიცავს მთელ რიგ მოდულებს, რომელებიც ეძღვნება მონადებს. აუცილებელია ავღნიშნოთ, რომ `Haskell`–ში მონადის ცნება ეფუძვნება კატეგორიების თეორიას, თუმცა, რათა არ ჩავღრმავდეთ აბსტრაქტულ მათემატიკაში, შემდგომში წარმოდგენილი იქნება მონადის ინტუიციური გაგება.

მონადები წარმოადგენენ ტიპებს, რომლებიც განსაზღვრავს შემდეგი მონადური კლასებიდან ერთ–ერთის ეგზემპლიარებს. ეს კლასებია: `Functor`, `Monad` და `MonadPlus`. არცერთი მათგანი არ შეიძლება იყოს სხვა კლასის წინაპარი, ანუ მონადური კლასიდან მემკვიდრეობითობა არ ხდება. მოდულში `Prelude` სამი მონადაა განსაზღვრული: `IO`, `[]` и `Maybe`,ანუ სია ასევე არის მონადა.

მათემატიკურად მონადა განისაზღვრება წესების ერთობლიობით, რომლებიც აკავშირებს მონადებზე მოქმედ ოპერაციებს. ეს წესები ინტუიციურად გვაგებინებს, როგორ უნდა იყოს გამოყენებული მონადა და როგორია მისი შინაგანი სტრუქტურა. დასაკონკრეტებლად განვიხილოთ კლასი `Monad`, რომელშიც განსაზღვრულია ორი ბაზური ოპერაცია და ერთი ფუნქცია:

```
class Monad m where
      (>>=)  :: m a -> (a -> m b) -> m b
      (>>)   :: m a -> m b -> m b
      return :: a -> m a
      fail   :: String -> m a

      m >> k = m >>= \_ -> k
```

ორი ოპერაცია `(>>=)` და `(>>)` — ეს არის დაკავშირების ოპერაციები. ისინი ახდენენ ორი მონადური მნიშვნელობის კომბინაციას მაშინ, როცა ფუნქცია `return` გარდაქმნის გადაცემულ რომელიღაც `a` ტიპის მნიშვნელობას `m a` ტიპის მონადურ მნიშვნელობაში. ოპერაცია `(>>)`–ის სიგნატურა გვეხმარება გავიგოთ დაკავშირების ოპერაცია: გამოსახულება `(m a >>= \v m b)` კომბინირებს მონადურ მნიშვნელობას `m a`, რომელიც შეიცავს `a` ტიპის ობიექტს ფუნქციასთან, რომელიც ოპერირებს `v` ტიპის მნიშვნელობებით და აბრუნებს `m b` ტიპის შედეგს. კომბინაციის შედეგი კი იქნება `m b` ტიპის მონადური მნიშვნელობა. ოპერაცია `(>>)` გამოიყენება მაშინ, როცა ფუნქცია არ იყენებს მნიშვნელობებს, მიღებულს პირველი მონადური ოპერანდით.

დაკავშირების ოპერაციის ზუსტი მნიშვნელობა, რა თქმა უნდა, დამოკიდებულია მონადის კონკრეტულ რეალიზაციასთან. ასე, მაგალითად, ტიპი `IO` განსაზღვრავს ოპერაციას `(>>=)` როგორც მისი ორი ოპერანდის თანმიმდევრული შესრულება, ხოლო პირველი ოპერანდის შესრულების შედეგი თანმიმდევრულად გადაეცემა მეორეს. დანარჩენი ორი ჩადგმული მონადური ტიპისთვის (სიები და `Maybe`) ეს ოპერაცია განსაზღვრულია როგორც ნული ან მეტი პარამეტრის გადაცემა ერთი გამოთვლითი პროცესიდან შემდეგზე.

`Haskell`–ში განსაზღვრულია სპეციალური მოსამსახურე სიტყვა, რომელიც ენის დონეზე მხარს უჭერს მონადების გამოყენებას. ეს არის სიტყვა `do`, რომლის აზრიც შეიძლება გავიგოთ მისი გამოყენების შემდეგი წესების მიხედვით:

```
do e1 ; e2          = e1 >> e2
do p <- e1 ; e2     = e1 >>= \p -> e2
```

პირველი გამოსახულება სრულდება ყოველთვის (მნიშვნელობების გადატანა პირველი ოპერანდიდან მეორეში არ ხდება). მეორე გამოსახულებაში შეიძლება იყოს შეცდომა, ამ დროს ხდება `fail` ფუნქციის გამოძახება, რომელიც ასევე განსაზღვრულია კლასში `Monad`. ამიტომაც მოსამსახურე სიტყვა `do`–უფრო ზუსტი განსაზღვრება მეორე შემთხვევისთვის ასე გამოიყურება:

```
do p <- e1 ; e2     = e1 >>= (\v -> case v of
                    p -> e2
                    _ -> fail "s")
```

სადაც `s` — ეს არის სტრიქონი, რომელსაც შეუძლია განსაზღვროს ოპერატორი `do`-ს ადგილმდებარეობა პროგრამაში, `a`-ს შეიძლება ჰქონდეს რაიმე სემანტიკური დატვირთვა. მაგალითად, `IO` მონადაში მოქმედება `('a' getChar)` იძახებს ფუნქციას `fail` იმ შემთხვევაში, თუ წაკითხული სიმბოლო არ არის სიმბოლო `'a'`. ეს მოქმედება წყვეტს პროგრამის შესრულებას, რადგან `IO` მონადაში განსაზღვრული ფუნქცია `fail` თავის მხრივ იძახებს სისტემურ ფუნქციას `error`.

კლასი `MonadPlus` გამოიყენება ისეთი მონადისათვის, რომელშიც არის ნულოვანი ელემენტი და ოპერაცია `"+"`. ამ კლასის აღწერა შემდეგნაირად გამოიყურება:

```
class (Monad m) => MonadPlus m where
      mzero :: m a
      mplus :: m a -> m a -> m a
```

მონადის ამ კლასში ნულოვანი ელემენტი ექვემდებარება შემდეგ წესებს:

```
m >>= \x -> mzero  = mzero
mzero >>= m  = mzero
```

მაგალითად, სიებისთვის ნულოვანი ელემენტი არის ცარიელი სია `[]`, ხოლო ოპერაცია „+“ - სიების კონკატენაცია. ამიტომაც, სიების მონადა წარმოადგენს კლასი `MonadPlus`-ის ეგზემპლიარს. მეორეს მხრივ, მონადა `IO`-ს არ აქვს ნულოვანი ელემენტი, ამიტომ იგი არის მხოლოდ კლასი `Monad`-ის ეგზემპლიარი.

ჩადგმული მონადები

მიღებული ცნობების კონკრეტიზაციისათვის აუცილებელია განვიხილოთ უფრო კონკრეტული მაგალითი. რადგანაც სიები არის მონადები და ამავე დროს შესწავლილია საკმაოდ დეტალურად, ამიტომ ისინი წარმოადგენს საუკეთესო მასალას მონადის მექანიზმის პრაქტიკული გამოყენების განხილვისათვის.

სიებისთვის დაკავშირების ოპერაციას აქვს ის აზრი, რომ გააერთიანოს იმ ოპერაციების ერთობლიობა, რომლებიც სრულდება სიის თითოეულ წევრთან. სიებთან გამოყენებისას ოპერაცია `(>>=)`-ის სიგნატურა იღებს შემდეგ სახეს:

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

ეს აღნიშნავს, რომ მოცემულია ტიპის მნიშვნელობისა და ფუნქციისგან შემდგარი სია, რომელიც დაიყვანს `a` ტიპის მნიშვნელობას `b` ტიპის მნიშვნელობების სიაზე. დაკავშირება იყენებს ფუნქციას მოცემული `a` ტიპის მნიშვნელობების სიის თითოეულ ელემენტზე და აბრუნებს მიღებულ სიას `b` ტიპის მნიშვნელობებით. ეს ოპერაცია უკვე ცნობილია - სიების განსაზღვრა სწორედ ასე მუშაობს. ამრიგად, შემდეგი სამი გამოსახულება აბსოლუტურად ერთნაირია:

```
-- გამოსახულება 1 ---------------------------------------------
[(x, y) | x <- [1, 2, 3], y <- [1, 2, 3], x /= y]


-- გამოსახულება 2----------------------------------------------
do     x <- [1, 2, 3]
       y <- [1, 2, 3]
       True <- return (x /= y)
       return (x, y)


-- გამოსახულება 3----------------------------------------------
[1, 2, 3] >>= (\x -> [1, 2, 3] >>= (\y -> return (x /= y) >>=
       (\r -> case r of
              True   -> return (x, y)
              _      -> fail "")))
```

რომელ გამოსახულებას გამოიყენებს პროგრამის დაწერისას, ეს პროგრამისტის გადასაწყვეტია.

სავარჯიშოები

1. რომელი ინტუიციური ცნებები შეიძლება ჩაიდოს ცნება „მონადაში“?

2. რა პრაქტიკული აზრი აქვს მონადის გამოყენებას ფუნქციონალურ პროგრამირებაში?

პასუხები თვითშემოწმებისთვის

1. პირველი, რაც გვინდა ვთქვათ, რომ მონადა არის კონტეინერული ტიპი. მართლაც, სია- ეს კონტეინერული ტიპია, რადგანაც სიის შიგნით არის სხვა ტიპის ელემენტები. სწორედ ეს არის ნაჩვენები მონადური ტიპის განმარტებაში:

```
class Monad m where
      (>>=)  :: m a -> (a -> m b) -> m b
      (>>)   :: m a -> m b -> m b
      return :: a -> m a
      fail   :: String -> m a
```

ჩანაწერი „`m a`“ თითქოს და გვიჩვენებს, რომ ტიპი `a` (აუცილებელია, რომ გვახსოვდეს, რომ კლასებისა და მონაცემთა სხვა ტიპების აღწერისას სიმბოლოები `a`, `b` და ა.შ. აღნიშნავს ტიპების ცვლადებს) შემოსაზღვრულია მონადური ტიპით `m`. თუმცა რეალური ფიზიკური შემოსაზღვრა შესაძლებელია მხოლოდ ტიპისთვის „სია“, რადგანაც მისი აღნიშვნა კვადრატული ფრჩხილებით უკვე ტრადიციაა. `Haskell`-ის მკაცრ ნოტაციაში შეიძლება დაგვეწერა ასეთი რამ `List (1 2 3 4 5)` - ეს არის სია `[1, 2, 3, 4, 5]`.

2. ფუნქციონალურ ენებში მონადის გამოყენება - ფაქტიურად ეს არის უკან დაბრუნება იმპერატიულისკენ. დაკავშირების ოპერაციები `(>>=)` და `(>>)` ხომ გულისხმობს დაკავშირებული გამოსახულებების თანმიმდევრულ შესრულებას გადაცემით ან გამოთვლების შედეგების გარეშე. ანუ მონადები - ეს იმპერატიული ბირთვია ფუნქციონალური ენების შიგნით. ერთის მხრივ, ეს ეწინააღმდეგება ფუნქციონალური პროგრამირების თეორიას, მაგრამ, მეორეს მხრივ, ზოგიერთი ამოცანა იხსნება მხოლოდ იმპერატიული ენების პრინციპებით. და კიდევ, `Haskell` იძლევა სიების შექმნის უნიკალურ შესაძლებლობას, მაგრამ ეს იმის ხარჯზე, რომ ტიპი „სია“ შესრულებულია მონადის სახით.

7. შეტანა–გამოტანის ოპერაციები Haskell–ში

Haskell–ში, როგორც სხვა დანარჩენ ენებში, არსებობს შეტანა–გამოტანის ოპერაციების ჩადგმული სისტემა. თუმცა, საზოგადოდ შეტანა–გამოტანის ოპერაციები გულისხმობს თავის შესრულებაში რაღაც თანმიმდევრობას, ანუ თავისი არსით იმპერატიულობას, Haskell–ში შეტანა–გამოტანის სისტემა სრულად უჭერს მხარს პროგრამირების ფუნქციონალურ პარადიგმას.

ჩვენ უკვე ვისაუბრეთ, რომ შეტანა–გამოტანის ოპერაციები აგებულია ენა Haskell–ის ისეთი ცნებისგან, როგორიცაა მონადა. ამავე დროს Haskell–ში შეტანა–გამოტანის სისტემის გასაგებად, არ არის აუცილებელი გესმოდეთ ცნება მონადის თეორიული საფუძვლები. მონადები შეიძლება განვიხილოთ როგორც კონცეპტუალური ჩარჩოები, რომელშიც მოთავსებული შეტანა–გამოტანის სისტემა. შეიძლება ითქვას, რომ კატეგორიების თეორიის გაგება შეტანა–გამოტანის სისტემის ოპერაციების გამოყენებისათვის ისევე აუცილებელია, როგორც ჯგუფების თეორიის გაგება არითმეტიკული ოპერაციების შესასრულებლად.

შეტანა–გამოტანის ოპერაცია ნებისმიერ ენაში ეფუძნება მოქმედების ცნებას. მოქმედების აღგრძნებისას ის სრულდება. თუმცა Haskell–ში მოქმედება არ აღიგრძნება, არამედ დეკლარირდება. თავის მხრივ, მოქმედება შეიძლება იყოს ატომარული ან შედგენილი სხვა მოქმედებების თანმიმდევრობისაგან. მონადა IO შეიცავს ოპერაციებს, რომლებიც იძლევა საშუალებას შეიქმნას რთული მოქმედებები ატომარულებისაგან, ანუ მონადა ამ შემთხვევაში შეიძლება განვიხილოთ როგორც წებო, რომელიც შეკრავს მოქმედებებს პროგრამაში.

შეტანა–გამოტანის ბაზური ოპერაციები

შეტანა–გამოტანის ყოველი მოქმედება აბრუნებს რაღაც მნიშვნელობებს. იმისათვის, რომ ეს მნიშვნელობები ბაზურისაგან განსხვავდებოდეს, ამ მნიშვნელობების ტიპები თითქოს და შეხვეულია IO ტიპით (აუცილებელია გვახსოვდეს, რომ მონადა წარმოადგენს კონტეინერულ ტიპს). მაგალითად, ფუნქცია getChar–ის ტიპი შემდეგია:

```
getChar      :: IO Char
```

ამ მაგალითში ნაჩვენებია, რომ ფუნქცია getChar ასრულებს რაღაც მოქმედებებს, რომელიც აბრუნებს Char ტიპის მნიშვნელობას. მოქმედებებს, რომლებიც არაფერ საინტერესოს არ აბრუნებენ, აქვთ ტიპი IO (). ანუ სიმბოლო () აღნიშნავს ცარიელ ტიპს (სხვა ენებში void). ასე, რომ ფუნქცია putChar აქვს ტიპი:

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

ერთმანეთთან მოქმედებები დაკავშირებულია დაკავშირების ოპერატორის საშუალებით. ანუ, სიმბოლოები >>= აგებენ მოქმედებების თანმიმდევრობას. როგორც ცნობილია, ამ ფუნქციის ნაცვლად შეიძლება გამოვიყენოთ მოსამსახურე სიტყვა do. იგი იყენებს ისეთივე ორმაგ სინტაქსს, როგორიცაა სიტყვები let და where, ამიტომაც შეიძლება არ იყოს გამოყენებული სიმბოლო „ ; “ ფუნქციების გამოძახების გამყოფად. სიტყვა do–ს საშუალებით შეიძლება დააკავშიროთ ფუნქციის გამოძახებები, მონაცემების განსაზღვრა (სიმბოლო „ <- “ –ის საშუალებით) და ლოკალური ცვლადების განმარტებების სიმრავლე (მოსამსახურე სიტყვა let).

მაგალითად, პროგრამა, რომელიც კითხულობს სიმბოლოს კლავიატურიდან და გამოჰყავს ეკრანზე, შეიძლება ასე განვსაზღვროთ:

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

ამ მაგალითში ფუნქციის სახელად სიტყვა main შემთხვევით არ არის ამორჩეული. Haskell–ში, ისევე როგორც C/C++ ენაში ფუნქციის სახელი main გამოიყენება პროგრამაში შესასვლელი წერტილის აღსანიშნავად. ამასთან, Haskell–ში main ფუნქციის ტიპი უნდა იყოს მონადა IO–ს ტიპი (ჩვეულებრივ გამოიყენება IO ()). გარდა ამ ყველაფრისა, შესასვლელი წერტილი ფუნქცია main–ის სახით უნდა იყოს განსაზღვრული მოდულში სახელით Main.

ვთქვათ, არის ფუნქცია ready, რომელმაც უნდა დააბრუნოს True, თუ დაჭერილია კლავიშა „y“ და False – დანარჩენ შემთხვევებში. არ შეიძლება უბრალოდ დაწეროთ:

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

ვინაიდან ამ შემთხვევაში შედარების ოპერაციის შედეგი იქნება Bool ტიპის მნიშვნელობა და არა IO Bool. იმისათვის, რომ მონადური მნიშვნელობა დავაბრუნოთ, არსებობს სპეციალური ფუნქცია return, რომელიც მონაცემთა მარტივი ტიპიდან შექმნის მონადურს. ანუ, წინა მაგალითში ბოლო სტრიქონი, სადაც განმარტებულია ფუნქცია ready, ასე უნდა გამოიყურებოდეს „return (c == 'y')“.

შემდეგ მაგალითში განმარტებულია უფრო რთული ფუნქცია, რომელიც კითხულობს სიმბოლოების სტრიქონს კლავიატურიდან:

მაგალითი 16. ფუნქცია getLine.

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

აუცილებელია გვახსოვდეს, რომ იმ მომენტში, როცა პროგრამისტი გადავიდა მოქმედებების სამყაროში (შეტანა–გამოტანის ოპერაციის გამოყენებით), უკან დასაბრუნებელი გზა არ არის. ანუ, თუ ფუნქცია არ იყენებს მონადურ ტიპს, მას არ შეუძლია შეტანა–გამოტანის განხორციელება და პირიქით, თუ ფუნქცია აბრუნებს მონადურ ტიპს IO–ს, მაშინ ის უნდა დაექვემდებაროს მოქმედებების პარადიგმას Haskell–ში.

პროგრამირება მოქმედებების საშუალებით

Haskell–ის ტერმინებში შეტანა–გამოტანის მოქმედებები ჩვეულებრივ მნიშვნელობებს წარმოადგენს. ანუ, მოქმედება შეიძლება გადავცეთ ფუნქციას პარამეტრად, ჩავრთოთ მონაცემთა სტრუქტურაში და საერთოდ, გამოვიყენოთ იქ, სადაც შეიძლება Haskell–ის მონაცემების გამოყენება. ამ აზრით შეტანა–გამოტანის ოპერაციების სისტემა წარმოადგენს სრულად ფუნქციონალურს. მაგალითად, შეიძლება ჩავთვალოთ მოქმედებების სია:

```
todoList     :: [IO ()]
todoList     = [putChar 'a',
                do putChar 'b'
                   putChar 'c',
                do c <- getChar
                   putChar c]
```

ეს სია არ აღაგრძნებს არანაირ მოქმედებებს, ის უბრალოდ შეიცავს მათ აღწერას. მაგრამ რომ შევასრულოთ ეს სტრუქტურა, ანუ აღვაგრძნოთ ყველა მისი მოქმედება, აუცილებელია რომელიღაც ფუნქცია (მაგალითად, sequence_) :

```
sequence_    :: [IO ()] -> IO ()
sequence_ [] = return ()
```

```
sequence_ (a:as)    = do   a
                           sequence as
```

ეს ფუნქცია იქნება სასარგებლო ფუნქცია putStr–ის დაწერისას, რომელსაც გამოჰყავს სტრიქონი ეკრანზე:

```
putStr :: String -> IO ()
putStr s      = sequence_ (map putChar s)
```

ამ მაგალითზე ჩანს ცხადი განსხვავება ენა Haskell–ის შეტანა–გამოტანის სისტემისა იმპერატიული ენების სისტემისაგან. თუ რომელიმე იმპერატიულ ენაში იქნებოდა ფუნქცია map, ის შეასრულებდა უამრავ მოქმედებებს. ამის ნაცვლად, Haskell–ში უბრალოდ იქმნება მოქმედებების სია (სტრიქონის თითოეული სიმბოლოსთვის ერთი), რომელიც შემდეგ შესასრულებლად მუშავდება ფუნქციით sequence_ .

გამონაკლისი სიტუაციების დამუშავება

რა გაკეთდეს, თუ შეტანა–გამოტანის ოპერაციის შესრულების პროცესში წარმოიშვა არაორდინალური სიტუაცია? მაგალითად, ფუნქცია getChar–მა აღმოაჩინა ფაილის ბოლო. ამ დროს ხდება შეცდომა. როგორც ყველა განვითარებული პროგრამირების ენა, Haskell–იც ასეთი მიზნებისთვის გვთავაზობს გამონაკლისი სიტუაციების დამუშავების მექანიზმს. ამისთვის არ გამოიყენება სპეციალური სინტაქსი, მაგრამ არის სპეციალური ტიპი IOError, რომელიც შეიცავს შეტანა–გამოტანის პროცესში წარმოშობილ ყველა შეცდომას.

გამონაკლისი სიტუაციების დამუშავებას აქვს სახე (IOError -> IO a), ამასთან ფუნქცია catch ასოცირებს (აკავშირებს) გამონაკლის სიტუაციას მოქმედებათა ერთობლიობასთან.

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

ამ ფუნქციის არგუმენტს წარმოადგენს მოქმედება (პირველი არგუმენტი) და გამონაკლისი სიტუაციის დამუშავება (მეორე არგუმენტი). თუ მოქმედება წარმატებით შესრულდა, მაშინ უბრალოდ ბრუნდება შედეგი გამონაკლისი სიტუაციების დამუშავების აღგზნების გარეშე. თუ მოქმედების შესრულების პროცესში აღმოჩნდა შეცდომა, მაშინ იგი გადაეცემა დამმუშავებელს IOError ტიპის ოპერანდის სახით, რის შემდეგაც მუშაობს თვითონ დამამუშავებელი.

ამრიგად, შეიძლება დავწეროთ უფრო რთული ფუნქციები, რომლების უფრო წიგნიერად მოიქცევა შეცდომითი სიტუაციის გამოვარდნის შემთხვევაში.

```
getChar'     :: IO Char
getChar'     = getChar `catch` eofHandler
              where eofHandler e = if isEofError e then return
\'n\ else ioError e

getLine'     :: IO String
getLine'     = catch getLine'' (\err -> return ("Error: " ++ show
err))
              where getLine'' = do c <- getChar'
                                   id c == '\n' then return ""
                                         else do l <-  getLine'
                                                 return (c : l)
```

ამ პროგრამიდან ჩანს, რომ შეიძლება ერთმანეთში ჩაიდოს შეცდომების დამამუშავებლები. ფუნქცია `getChar'`–ში ხდება შეცდომის გამოჭერა, რომელიც გაჩნდა სიმბოლოს „ფაილის ბოლო“–ს აღმოჩენისას. თუ შეცდომაა სხვაა, მაშინ ფუნქცია `ioError`–ის დახმარებით ის მიემართება უფრო შორს და მას დაიჭერს დამმუშავებელი, რომელიც „ზის“ `getLin` ფუნქციაში.

`Haskell`-ში გათვალისწინებულია აგრეთვე გამონაკლისი სიტუაციების დამუშავება გაჩუმებით, რომელიც მოთავსებულია ჩადგმის ყველაზე ზედა დონეზე. თუ შეცდომა არ იქნა გამოჭერილი არცერთი დამამუშავებელის მიერ, რომლებიც პროგრამაშია დაწერილი, მაშინ მას გამოიჭერს დამამუშავებელი გაჩუმებით, რომელიც ეკრანზე გამოიყვანს შეტყობინებას შეცდომის შესახებ და გააჩერებს პროგრამას.

ფაილები, არხები და დამამუშავებლები

ფაილებთან სამუშაოს `Haskell` გვთავაზობს ყველა იმ შესაძლებლობებს, როგორც პროგრამირების სხვა ენები. თუმცა ამ შესაძლებლობების უმრავლესობა განსაზღვრულია მოდულში `IO` და არა `Prelude`–ში, ამიტომ ფაილებთან სამუშაოდ აუცილებელია ცხადად გაუკეთოთ იმპორტი მოდულს `IO`.

ფაილის გახსნას ახდენს დამამუშავებელი (მას აქვს ტიპი `Handle`). დამამუშავებელის დახურვა ინიცირებს ფაილის დახურვას. დამამუშავებელი შეიძლება ასოცირდეს არხთან, ამუ ურთიერთქმედების პორტებთან, რომლებიც აკავშირებს პირდაპირ ფაილთან. ჰასკელში არის სამი ასეთი არხი – `stdin` (შეტანის

სტანდარტული არხი, stdout (გამოტანის სტანდარტული არხი) და stderr (სტანდარტული არხი შეცდომების შესახებ შეტყობინებების გამოსატანად).

ამრიგად, ფაილების გამოსაყენებლად შეიძლება ვისარგებლოთ შემდეგი საშუალებებით:

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

შემდგომში მოყვანილია პროგრამის მაგალითი, რომელიც აკოპირებს ერთ ფაილს მეორეში:

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

getAndOpenFile     :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
    do putStr prompt
       name <- getLine
       catch (openFile name mode)
             (\_ -> do putStrLn ("Cannot open "++ name ++ "\n")
                       getAndOpenFile prompt mode)
```

აქ გამოიყენება ერთი საინტერესო და სასარგებლო ფუნქცია – hGetContents, რომელიც იღებს მასზე არგუმენტად გადაცემული ფაილის შინაარს და აბრუნებს მას ერთი გრძელი სტრიქონის სახით.

შენიშვნები

გამოდის, რომ Haskell–ში თავიდან აღმოაჩინეს იმპერატიული პროგრამირება...

რაღაც აზრით, დიახ ასეა. Haskell–ში მონადა IO არის ჩადგმული პატარა იმპერატიული ქვეენა, რომლის საშუალებით შესაძლებელია შეტანა–გამოტანის ოპერაციების შესრულება. ამ ქვეენაზე პროგრამის დაწერა წააგავს ჩვეულებრივი აზრით იმპერატიული ენებზე დაწერას. მაგრამ არსებობს არსებითი განსხვავება: Haskell–ში არ არის სპეციალური სინტაქსი პროგრამულ კოდში იმპერატიული ფუნქციის შეტანისა, ყველაფერი სრულდება ფუნქციონალური პარადიგმის დონეზე. ამავე დროს გამოცდილი პროგრამისტები მინიმიზირებენ იმპერატიულ კოდს, იყენებენ რა IO მონადას მხოლოდ თავისი პროგრამების ზედა დონეზე, რადგანაც Haskell–ში იმპერატიული და ფუნქციონალური სამყაროები მკვეთრად განსხვავდება ერთმანეთისგან. Haskell–გან განსხვავებით, იმ იმპერატიულ ენებში, რომლებშიც არის ფუნქციონალური ქვეენები, არ არის მკვეთრი განსაზღვრა ამ სამყაროებს შორის.

8. ფუნქციების კონსტრუირება

ფუნქციების კონსტრუირებისთვის გამოიყენება სხვადასხვა ფორმალიზმები. ერთ-ერთია სინტაქსურად ორიენტირებული კონსტრუქცია. ამ უკანასკნელი მეთოდიკის გამოსაყენებლად, გავეცნოთ მეთოდს, რომელიც თავის დროზე შემოგვთავაზა ჰოარმა.

ქვემოთ მოყვანილია მეტაენის აღწერა, რომელიც გამოიყენება მონაცემთა სტრუქტურის აღსაწერად (აბსტრაქტულ სინტაქსში):

1. **დეკარტული ნამრავლი:** თუ $C_1, \ldots, C_n$ - ტიპებია, ხოლო C – ტიპია, რომელიც შედგება შემდეგი სახის n-ურები სიმრავლისგან: $\langle c_1, \ldots, c_n \rangle$, $c_i \to C_i$, $i = 1,n$, მაშინ ამბობენ, რომ C - დეკარტული ნამრავლია $C_1, \ldots, C_n$ ტიპების და აღნიშნავენ ასე: $C = C_1 \times \ldots \times C_n$. ამასთან იგულისხმენება, რომ განსაზღვრულია C ტიპისთვის სელექტორები $s_1, \ldots, s_n$, რაც ჩაიწერება როგორც $s_1, \ldots, s_n$ = selectors C.

ასეთივე სახით ჩაიწერება კონსტრუქტორი g: g = constructor C. კონსტრუქტორი - ეს ფუნქციაა, რომელსაც აქვს ტიპი $(C_1 \to \ldots (C_n \to C) \ldots)$, ანუ $c_i \to Ci$-თვის , i = 1,n : g c1 ... cn = <c1, ..., cn>.

ჩავთვალოთ, რომ სამართლიანია შემდეგი ტოლობა:

$$Ax \to C : \text{constructor } C\ (s_1, x) \ldots (s_n, x) = x$$

ამ ტოლობას უწოდებენ ტეკტონიკურობის აქსიომას. ზოგჯერ ამ აქსიომას შემდეგნაირად წერენ:

$$s_i\ (\text{constructor } C\ c_1 \ldots c_n) = c_i$$

2. **სპეციალური გაერთიანება:** თუ $C_1, \ldots, C_n$ – ტიპებია, ხოლო C – ტიპია, რომელიც შედგება $C_1, \ldots, C_n$, ტიპების გაერთიანებისგან „სპეციალურობის“ პირობის შესრულების პირობებში, მაშინ C-ს $C_1, \ldots, C_n$ ტიპებს უწოდებენ სპეციალურ გაერთიანებას. ეს ფაქტი აღინიშნება ასე: $C = C_1 + \ldots + C_n$. სპეციალურობის პირობა აღნიშნავს, რომ თუ C-დან ავიღებთ რომელიღაც ელემენტს c_i, მაშინ ცალსახად განისაზღვრება ამ ელემენტის ტიპი C_i. სპეციალურობა შეიძლება განვსაზღვროთ პრედიკატებით $P_1, \ldots, P_n$ ისეთებით, რომ:

$$(x \to C)\ \&\ (x \to C_i) \Longrightarrow (P_i\ x = 1)\ \&\ (Aj \neq i : P_j\ x = 0)$$

სპეციალური გაერთიანება უზრუნველყოფს ასეთი ელემენტების არსებობას. ეს ფაქტი მიეთითება შემდეგი ჩანაწერით: $P_1, \ldots, P_n$ = predicates C. არსებობს ასევე ტიპის ნაწილები, რომლებიც ასე აღინიშნება: $N_1, \ldots, N_n$ = parts C.

როგორც ვხედავთ, მეტაენაში გამოიყენება ტიპების ორი კონსტრუქტორი - : და +. შემდეგ განვიხილავთ ახალი ტიპის განსაზღვრის რამდენიმე მაგალითს.

მაგალითი 17. ტიპი List (A)-ს ფორმალური აღწერა.

```
List (A) = NIL + (A x List (A))
null, nonnull = predicates List (A)
NIL, nonNIL = parts List (A)
head, tail = selectors List (A)
prefix = constructor List (A)
```

თუ შევხედავთ ტიპის ამ აღწერას (უფრო განსაზღვრებას), შეგვიძლია აღვწეროთ ფუნქციის გარეგნული სახე, რომელიც დაამუშავებს `List (A)` ტიპის სტრუქტურას:

თითოეული ფუნქცია უნდა შეიცავდეს მინიმუმ ორ კლოზს, პირველი ამუშავებს `NIL`-ს, მეორე - `nonNIL`-ს შესაბამისად. `List (A)` ტიპის ამ ორ ნაწილს აბსტრაქტულ ჩანაწერში შეესაბამება სელექტორები `[]` და `(H : T)`. ორი კლოზი შეიძლება გავაერთიანოთ დაცვის გამოყენებით. მეორე კლოზის ტანში (ანუ დაცვის მეორე გამოსახულება ელემენტი `T`–ს დამუშავება (ან `tail (L)`) სრულდება იგივე ფუნქციის მიერ.

მაგალითი 18. List_str (A) ტიპის ფორმალური აღწერა.

```
List_str (A) = A + List (List_str (A))
atom, nonAtom = predicates List_str (A)
```

ფუნქციებს `List_str (A)` უნდა ჰქონდეთ, უკიდურეს შემთხვევაში, შემდეგი კლოზები მაინც:

```
1°. A -> when (atom (A))
2°. [] -> when (null (L))
3°. (H : T) -> head (L), tail (L)
3.1°. atom (head (L))
3.2°. nonAtom (head (L))
```

მაგალითი 19. მონიშნული წვეროებით ხეებისა და ტყეების ფორმალური აღწერა.

```
Tree (A) = A x Forest (A)
Forest (A) = List (Tree (A))
root, listing = selectors Tree (A)
ctree = constructor Tree (A)
```

მაგალითი 20. მონიშნული წვეროებითა და გვერდებით ხეების ფორმალური აღწერა.

```
MTree (A, B) = A x MForest (A, B)
MForest (A, B) = List (Element (A, B))
Element (A, B) = B x MTree (A, B)
mroot, mlist = selectors MTree (A, B)
null, nonNull = predicates MForest (A, B)
arc, mtree = selectors Element (A, B)
```

მტკიცდება, რომ ნებისმიერი ფუნქცია, რომელიც მუშაობს ტიპთან `MTree (A, B)` შეიძლება წარმოდგეს მხოლოდ ნახსენები ექვსი ოპერაციით იმის და მიუხედავად, როგორ არის ის რეალიზებული. ეს დებულება შეიძლება შემოწმდეს დიაგრამის საშუალებით (უფრო სწორედ, ეს ჰიპერგრაფია), რომელზეც ნათლად ჩანს, რომ ტიპი `MTree (A, B)`–ის ნებისმიერ ნაწილთან შეიძლება „მიღწევა“ მხოლოდ ამ ექვსი ოპერაციის გამოყენებით.

ფუნქციის კონსტრუირებისთვის, რომელიც ამუშავებს `MTree` მონაცემთა სტრუქტურას, აუცილებელია შემოვიტანოთ რამდენიმე დამატებითი ცნება და აღნიშვნები. საწყისი წვერო, წვერო `MForest` და წვერო `MTree` (რომელიც გამოდის `Element`–დან) აღინიშნება, შესაბამისად, როგორც `S0`, `S1` და `S2`. ამ წვეროების დასამუშავებლად აუცილებელია სამი ფუნქცია – `f0`, `f1` და `f2`, ამასთან, `f0` – საწყისი ფუნქციაა, დანარჩენი ორი კი – რეკურსიული.

`f0`–ის კონსტრუირება მარტივია _ ამ ფუნქციას ერთი პარამეტრი აქვს `T`, რომელიც შეესაბამება საწყის წვეროს `S0`–ს. შემდეგი ორი ფუნქცია კი რთულად კონსტუირდება.

ფუნქცია `f1` იღებს შემდეგ პარამეტრებს:

`A` - წვეროს ჭდე;

`K` - პარამეტრი, რომელიც შეიცავს ხის გადახედილი ნაწილის დამუშავების შედეგს.

`L` - ტყე, რომლის დამუშავებაც აუცილებელია.

```
f1 A K L = g1 A K when null L
f1 A K L = f1 A (g2 (f2 A (arc (head L)) (mtree (tail L)) K) A
(arc L) K) (tail L) otherwise
```

ეს ფუნქცია რეალიზებს ხის გადახედვის რეჟიმს „თავიდან სიღრმისკენ“.

ფუნქცია `f2` იღებს შემდეგ პარამეტრებს (და ეს უკვე ცხადია მისი გამოძახებიდან `f1` ფუნქციის მეორე კლოზში):

A - წვეროს ჭდე;

B - ფერდის ჭდე;

T - ქვეხე დამუშავებისთვის;

K - ხის გადახედილი ნაწილის დამუშავების შედეგი.

```
f2 A B T K = f1 (mroot T) (g3 A B K) (mlist T)
```

აუცილებელია ავღნიშნოთ, რომ ეს არის ფუნქციის ზოგადი სახე MTree მონაცემთა სტრუქტურის დასამუშავებლად. დამატებითი g1, g2 და g3 ფუნქციების რეალიზება დამოკიდებულია კონტრეტულ ამოცანაზე. ეხლა შეიძლება ავაგოთ f0 ფუნქციის ზოგადი სახე:

```
f0 T = f1 (root T) k (mlist T)
```

სადაც k - K პარამეტრის საწყისი მნიშვნელობაა.

ფუნქციის კონსტრუირების მეთოდიკის უფრო ღრმა გაგებისთვის განვიხილოთ B–ხეებთან მუშაობის ფუნქციის კონკრეტული რეალიზაცია. დავუშვათ BTree მონაცემთა სტრუქტურისთვის არსებობს საბაზისო ოპერაციების ერთობლიობა, ხოლო თვითონ ხე წარმოადგენილია სიების სახით (წარმოდგენას არ აქვს განსაკუთრებული მნიშვნელობა). ბაზისური ოპერაციები შემდეგია:

1°. cbtree A Left Right = [A, Left, Right]

2°. ctree = []

3°. root T = head T

4°. left T = head (tail T)

5°. right T = head (tail (tail T))

6°. empty T = (T == [])

მაგალითი 21. ხეში ელემენტის ჩასმის ფუნქცია insert.

```
insert (A:L) T = cbtree (A:L) ctree ctree when (empty T)
insert (A:L) T = cbtree (root T) (insert (A:L) (left T)) (right
T) when (A < head (root T))
insert (A:L) T = cbtree (A:(L:tail (root T))) (left T) (right T)
when (A == head (root T))
insert (A:L) T = cbtree (root T) (left T) (insert (A:L) (right
T)) otherwise
```

ეს არის რეალიზაცია აბსტრაქტულ დონეზე.

მაგალითი 22. B-ხეებში ელემენტის ძებნის ფუნქცია access.

```
access A Emptic = []
access A ((A1:L) Left Right) = access A Left when (A < A1)
access A ((A1:L) Left Right) = access A Right when (A > A1)
access A ((A:L) Left Right) = L
access A (Root Left Right) = access A Right otherwise
```

ამ მაგალითში მოყვანილია ორი კონსტრუქცია – აბსტრაქტული ელემენტი `Emptic`, რომელიც წარმოადგენს ცარიელ ხეს, ასევე ნიშანი : , რომლის საშუალებით აბსტრაგირდება დეკარტეს ნამრავლი, რომელიც აქ გამოიყენება სიური წარმოდგენის ნაცვლად. მაგრამ უნდა გვახსოვდეს, რომ ეს მხოლოდ აბსტრაქტული ფუნქციონალური ენაა.

წარმოდგენილ ორივე მაგალითში არსებობს ერთი პრობლემა. დაწერილი ფუნქციების გამოყენებისას ხდება ძალზე დიდი რაოდენობის ზედმეტი კოპირებები მეხსიერების ერთი ადგილიდან მეორეში. არსებითად, ეს არის ახალი ხის აგება ახალი ელემენტებით (საუბარი ეხება ფუნქციას `insert`). ეს შეიძლება თავიდან ავიცილოთ დესტრუქტიული მინიჭებით.

სავარჯიშოები

1. მოახდინეთ ფუნქციის კონსტრუირება ფუნქცია `insert`–ის, რომელიც ჩასვამს ელემენტს `B`–ხეში, გამოიყენებს რა დესტრუქციულ მინიჭებას.

პასუხები თვითშემოწმებისთვის

1. ფუნქცია `insert`–ის ერთ–ერთი შესაძლო ვარიანტი დესტრუქციული მინიჭებით:

```
-- "ფსევდო ფუნქციები" დესტრუქციული მინიჭებისთვის. მკაცრ
-- ფუნქციონალურ ენაზე (Haskell)
-- ასე გაკეთება არ შეიძლება. Lisp–ში არის დესტრუქციული მინიჭების
-- გამოყენების შესაძლებლობა
```

```
replace_root A T - ფუნქცია ამატებს ელემენტს ხის ფესვში replace_left K
(Root x Emptic x Right) => (Root x (K x Emptic x Emptic) x
Right)

replace_right K (Root x Left x Emptic) => (Root x Left (K x
Emptic x Emptic))


-- ფუნქცია insert


insert K Emptic = cbtree K ctree ctree

insert (A:L) ((A1:L1) x Left x Right) = insert (A:L) Left when
((A < A1) & nonEmpty Left)

insert (A:L) ((A1:L1) x Emptic x Right) = replace_left (A:L)
((A1:L1) x Emptic x Right) when (A < A1)

insert (A:L) ((A1:L1) x Left x Right) = insert (A:L) Right when
((A > A1) & nonEmpty Right)

insert (A:L) ((A1:L1) x Left x Emptic) = replace_right (A:L)
((A1:L1) x Left x Emptic) when (A > A1)

insert A T = replace_root A T otherwise
```

9. ფუნქციების თვისებების დამტკიცება

ფორმალური ამოცანა: ვთქვათ გვაქვს ფუნქციების ერთობლიობა $f = \langle f_1, \ldots, f_n \rangle$, რომლებიც განსაზღვრულია არეებზე $D = \langle D_1, \ldots, D_n \rangle$. საჭიროა დამტკიცდეს, რომ მნიშვნელობების ნებისმიერი d ერთობლიობისთვის ადგილი აქვს შემდეგ თვისებას, ანუ:

$$\forall d \in D : P(f(d)),$$

სადაც P — განხილული თვისებაა. მაგალითად:

1. $\forall d \in D: f(d) \geq 0$
2. $\forall d \in D: f(d) = f(f(d))$
3. $\forall d \in D: f(d) = f_1(d)$

შემოდის პრინციპური შეზღუდვა განსახილველ თვისებებზე – თვისებები არის მხოლოდ ტოტალური, ანუ მართებულია მთელი D არესთვის.

შემდომში განვიხილავთ D განსაზღვრის არის ზოგიერ სახეებს...

1. D — წრფივად დალაგებული სიმრავლეა.

ნახევრად ფორმალურად წრფივად დალაგებული სიმრავლე შეიძლება განვსაზღვროთ როგორც სიმრავლე, რომლის თითოეული ელემენტისათვის შეიძლება ითქვას, რომელია ნაკლები (ან მეტი), ანდა ისინი ტოლია, ანუ:

$$\forall d_1, d_2 \in D: (d_1 \geq d_2) \vee (d_1 \leq d_2).$$

მაგალითისთვის მოვიყვანოთ მთელი რიცხვების სიმრავლე. წრფივად დალაგებული სიმრავლე ფუნქციონალურ პროგრამირებაში გვხვდება ძალზე იშვიათად. ავიღოთ, თუნდაც უმარტივესი სტრუქტურა, რომელიც ყველაზე ხშირად მუშავდება ფუნქციონალურ პროგრამირებაში – სია. სიებისთვის უკვე ძალზე ძნელია განსაზღვრო რიგის მიმართება.

ფუნქციის თვისების დასამტკიცებლად წრფივად დალაგებული სიმრავლისთვის საკმარისია განახორციელო ინდუქცია მონაცემებზე, ანუ საკმარისია დაამტკიცო ორი პუნქტი:

1. $P(f(\varnothing))$ — ინდუქციის ბაზისი;
2. $\forall d_1, d_2 \in D, d_1 \leq d_2 : P(f(d_1))$ — ინდუქციის ბიჯი.

იმის გამო, რომ სტრუქტურები იშვიათად ქმნის წრფივად დალაგებულ სიმრავლეს, უფრო ეფექტური საშუალებაა ინდუქციის მეთოდის გამოყენება D ტიპის აგებისას.

3. D – განისაზღვრება როგორც ინდუქციური კლასი

ჩვენთვის უკვე ცნობილია, რომ ინდუქციური კლასი განისაზღვრება ბაზისური კლასის შემოტანით (ეს არის რომელიღაც d_i = 0,n in D კონსტანტების ერთობლიობა ან საწყისი A_i = 0,n : d in A_i => d in D ტიპების ერთობლიობა). ინდუქციური კლასი განისაზღვრება ასევე ინდუქციის ბიჯით – მოცემულია კონსტრუქტორები g_1, ..., g_m, რომლებიც განსაზღვრულია A_i და D–ზე და სამართლიანია, რომ:

$$(a_i \in A_i) \wedge (x_j \in D) \Rightarrow g_k(a_i, x_j) \in D,\ k = \overline{1,m}\ .$$

1. P (f (d)) აუცილებელია დავამტკიცოთ კლასის ბაზისისათვის;
2. ინდუქციის ნაბიჯი: P (f (d)) = P (f (g_i (d))).

მაგალითად, სიებისთვის (ტიპი List (A)) ფუნქციების თვისებების დასამტკიცებლად, საკმარისია დამტკიცდეს თვისება ორი შემდეგი შემთხვევისთვის:

1. P (f ([])).
2. $\forall a \in A, \forall L \in List(A) : P(f(L)) \Rightarrow P(f(a : L))$.

ფუნქციებისთვისებების დამტკიცება S-გამოსახულებებზე (ტიპი S-expr (A)) შეიძლება ჩავატაროთ შემდეგი ინდუქციის მაგალითზე:

1. $\forall a \in A : P(f(a))$.
2. $\forall x, y \in S - expr(A) : P(f(x)) \wedge P(f(y)) \Rightarrow P(f(x : y))$.

მაგალითი 23. დავამტკიცოთ, რომ $\forall L \in List(A) : L * [] = L$.

ამ თვისების დასამტკიცებლად შეიძლება განვიხილოთ მხოლოდ List (A) ტიპის განსაზღვრებები და თვითონ ფუნქცია append (ინფიქსურ ჩანაწერში გამოვიყენოთ სიმბოლო *).

1. L = [] : [] * [] = [] = L. ინდუქციის ბაზისი დამტკიცდა.
2. $\forall L \in List(A) : L * [] = L$. ეხლა გამოვიყენოთ კონსტრუქტორი: a : L. აუცილებელია დავამტკიცოთ, რომ (a : L) * [] = a : L. ეს კეთდება ფუნქცია append–ის განსაზღვრების მეორე კლოზის გამოყენების საშუალებით:

$$(a : L) * [] = a : (L * []) = a : (L) = a : L\ .$$

მაგალითი 24. დავამტკიცოთ ფუნქცია append–ის ასოციურობა.

ანუ საჭიროა დავამტკიცოთ, რომ ნებისმიერი სამი სიისათვის L1, L2 და L3 ადგილი აქვს ტოლობას (L1 * L2) * L3 = L1 * (L2 * L3). დამტკიცებისას ინდუქციას გამოვიყენებთ პირველ ოპერანდთან, ანუ სია L1–თან:

1°. L_1 = []:

```
([] * L2) * L3 = (L2) * L3 = L2 * L3.
[] * (L2 * L3) = (L2 * L3) = L2 * L3.
```

2°. ვთქვათ სიებისთვის L_1, L_2 და L_3 ფუნქცია append–ის ასოციურობა დამტკიცებულია. აუცილებელია დავამტკიცოთ სიებისთვის (a : L_1), L_2 და L_3:

```
((a : L1) * L2) * L3 = (a : (L1 * L2)) * L3 = a : ((L1 * L2) *
L3).
(a : L1) * (L2 * L3) = a : (L1 * (L2 * L3)).
```

როგორც ჩანს, უკანასკნელი ორი გამოსახულება არის ტოლი, რაც გულისხმობს, რომ სიებისთვის L1, L2 და L3 ასოციურობა დამტკიცებულია.

მაგალითი 25. ფუნქცია reverse–ის ორი განსაზღვრების იგივეობის დამტკიცება.

განმარტება 1:

```
reverse [] = []
reverse (H : T) = (reverse T) * [H]
```

განმარტება 2:

```
reverse' L = rev L []

rev [] L = L
rev (H : T) L = rev T (H : L)
```

ჩანს, რომ სიების შებრუნების ფუნქციის პირველი განმარტება – ეს ჩვეულებრივი რეკურსიული განმარტებაა. მეორე განმარტება იყენებს აკიუმულატორს. საჭიროა დავამტკიცოთ, რომ:

$$\forall L \in List(A):\ reverse\, L = reverse'\, L\ .$$

1. ბაზისი – L = []:

```
reverse [] = [].
reverse' [] = rev [] [] = [].
```

2. ბიჯი – ვთქვათ L სიისთვის ფუნქციების reverse და reverse' იგივეობა დამტკიცებულია. აუცილებელია დამტკიცდეს იგი სიისთვის (H : L):

```
    reverse (H : L) = (reverse L) * [H] = (reverse' L) * [H].
    reverse' (H : L) = rev (H : L) [] = rev L (H : []) = rev L
[H].
```

ეხლა აუცილებელია დავამტკიცოთ ბოლო ორი გამოყვანილი გამოსახულების ტოლობა ნებისმიერი სიისთვის A ტიპის. ესეც ინდუქციით ხდება:

2.1. ბაზისი – L = []:

```
    (reverse' []) * [H] = (rev [] []) * [H] = [] * [H] = [H].
    rev [] [H] = [H].
```

2.2. ბიჯი – L = (A : T):

```
    (reverse' (A : T)) * [H] = (rev (A : T) []) * [H] = (rev T (A
: [])) * [H] = (rev T [A]) * [H].
    rev (A : T) [H] = rev L (A : H).
```

აქ მოხდა ცუდ უსასრულობაზე გადავარდნა. თუ გავაგრძელებთ დამტკიცებას ინდუქციის გამოყენებით ახალ გამოყვანილ გამოსახულებებზე, მაშინ ეს გამოსახულებები სულ უფრო გართულდება და გართულდება. მაგრამ ეს არ არის სასოწარკვეთის მიზეზი, მაინც შეიძლება დავამტკიცოთ. საჭიროა უბრალოთ მოვიფიქროთ „ინდუქციური ჰიპიტეზა“, როგორც ეს იყო წინა მაგალითში.

ინდუქციური ჰიპოტეზა: (reverse' L1) * L2 = rev L1 L2. ეს ინდუქციური ჰიპოტეზა წარმოადგენს განზოგადოებულ გამოსახულებას (reverse' L) * [H] = rev L [H].

ინდუქციის ბაზისი ამ ჰიპოტეზისთვის ცხადია. ინდუქციის ბიჯის გამოყენება 2.2 პუნქტში მყოფ გამოსახულებასთან ასე გამოიყურება:

```
(reverse' (A : T)) * L2 = (rev (A : T) []) * L2 = (rev T [A]) *
      L2 = ((reverse' T) * [A]) * L2 =
         = (reverse' T) * ([A] * L2) = (reverse' T) * (A : L2).


rev (A : T) L2 = rev T (A : L2) = (reverse' T) * (A : L2).
```

რისი დამტკიცებაც მოითხოვებოდა.

დასკვნა: ზოგად შემთხვევებში ფუნქციის თვისებების დასამტკიცებლად ინდუქციის მეთოდით შეიძლება საჭირო გახდეს ზოგიერთი ევრისტიკის გამოყენება,

უფრო ზუსტად, ინდუქციური ჰიპოტეზების შემოტანა. ევრისტიკული ბიჯი არის დებულების ფორმულირება, რომელიც არსაიდან არ გამომდინარეობს. ასე, რომ ფუნქციის თვისებების დამტკიცება არის ხელოვნება.

10. ფუნქციონალური პროგრამირების ფორმალიზაცია λ–აღრიცხვის საფუძველზე

შესწავლის ობიექტია: ფუნქციების განსაზღვრებების სიმრავლე.

დაშვებები: ჩავთვლით, რომ ნებისმიერი ფუნქცია შეიძლება განსაზღვრული იყოს რომელიღაც λ–გამოსახულებით.

კვლევის მიზანი: ორი ფუნქციის <f_1>–ის და <f_2>–ის აღწერის მიხედვით განვსაზღვროთ მათი იგივეობა განსაზღვრის მთელ არეზე – $\forall x : f_1(x) = f_2(x)$. (აქ გამოყენებულია ასეთი ნოტაცია: `f` – განსაზღვრული ფუნქცია – ამ ფუნქციის აღწერა λ–აღრიცხვის ტერმინებში.)

პრობლემა მდგომარეობს იმაში, რომ ჩვეულებრივ ფუნქციის აღწერისას მოიცემა ამ ფუნქციის ინტენსიონალი, ხოლო შემდეგ მოითხოვება დამყარდეს ექსტენსიონალური ტოლობა. ექსტენსიონალური ფუნქციის ქვეშ გაიგება მისი გრაფიკი (ან ცხრილი <არგუმენტი, მნიშვნელობა> წყვილების სიმრავლის სახით). ფუნქციის ინტენსიონალის ქვეშ გაიგება მოცემულ არგუმენტზე ფუნქციის მნიშვნელობის გამოთვლის წესი.

ისმის შეკითხვა: როგორ გავითვალისწინოთ ჩადგმული ფუნქციების სემანტიკა მათი ექსტენსიონალების შედარებისას (რადგანაც ან ფუნქციების ცხადი აღწერა არ არის ცნობილი)? პასუხების ვარიანტებია:

1. შეიძლება შევეცადოთ გამოვსახოთ ჩადგმული ფუნქციების სემანტიკა λ–აღრიცხვის მექანიზმის გამოყენებით. ეს პროცესი შეიძლება დავიყვანოთ იმ შემთხვევამდე, როცა ყველა ჩადგმული ფუნქცია არ შეიცავს არაინტერპრეტირებულ ოპერაციებს.
2. ამბობენ, რომ <f1> და <f2> სემანტიკურად ტოლია (ეს ფაქტი აღინიშნება როგორც `|= f1 = f2`), თუ `f1 (x) = f2 (x)` არაინტერპრეტირებული იდენტიფიკატორების ნებისმიერი ინტერპრეტაციისას.

ფორმალური სისტემის ცნება

ფორმალური სისტემა წარმოადგენს ოთხეულს:

`P = <V, Φ, A, R>`, სადაც

`V` – ანბანი.

Φ — სწორად აგებული ფორმულების სიმრავლე.

A — აქსიომები (ამასთან **A in Φ**).

R — გამოყვანის წესი.

განხილულ ამოცანაში ფორმულებს აქვთ სახე **(t1 = t2)**, სადაც **t1** და **t2** **&mdashl** აქვთ λ–გამოსახულების სახე. თუ რომელიმე ფორმული გამოყვანადია ფორმალურ სისტემაში, მაშინ ეს ფაქტი ასე ჩაიწერება **(|- t1 = t2)**.

ამბობენ, რომ ფორმალური სისტემა კორექტულია, თუ **(|- t1 = t2) => (|= t1 = t2)**.

ამბობენ, რომ ფორმალური სისტემა სრულია, თუ **(|= t1 = t2) => (|- t1 = t2)**.

ცნება „კონსტრუქციის“ სემანტიკური განმარტება (აღნიშვნა – **Exp**):

1. $v \in Id \Rightarrow v \in Exp$
2. $v \in Id, E \in Exp \Rightarrow \lambda v.E \in Exp$
3. $E, E' \in Exp \Rightarrow (EE') \in Exp$
4. $E \in Exp \Rightarrow (E) \in Exp$
5. სხვა არანაირი **Exp** არ არის.

შენიშვნა: **Id** — იდენტიფიკატორების სიმრავლე.

ამბობენ, რომ **v** თავისუფალია **M Exp**–ში, თუ:

1. **M = v.**
2. **M = (M1M2)**, და **v** თავისუფალია **M1**–ში ან **M2**–ში.
3. **M = λv'.M'**, და **и v != v'**, და **v** თავისუფალია **M'**–ში.
4. **M = (M')**, და **v** თავისუფალია **в M'**.

იდენტიფიკატორების სიმრავლე **v**–ს, რომლებიც თავისუფალია **M**–ში, აღნიშნავენ როგორც **FV (M)**.

ამბობენ, რომ **v** დაკავშირებულია **M in Exp**–ში, თუ:

1. **M = λv'.M'**, და **v = v'**.
2. **M = (M1M2)**, და **v** დაკავშირებულია **M1**–ში ან **M2**–ში (ანუ ერთი და იგივე იდენტიფიკატორი შეიძლება იყოს თავისუფალი და დაკავშირებული **Exp**–ში).
3. **M = (M')**, და **v** დაკავშირებულია **M'**–ში.

მაგალითი 26. თავისუფალი და დაკავშირებული იდენტიფიკატორები.

M = v. v — თავისუფალია. **M = λx.xy. x** — დაკავშირებულია, **y** — თავისუფალია. **M = (λv.v)v. v** შედის ამ გამოსახულებაში როგორც თავისუფალი, ისევე, როგორც დაკავშირებული. **M = VW. V** და **W** — თავისუფალია.

ჩასმის წესები: **E** გამოსახულებაში **E'**გამოსახულების ჩასმა ცვლადი **x**–ის ყველა თავისუფალი შესვლასთან ერთად აღინეშნება ასე **E[x <- E']**. ჩასმის დროს ზოგჯერ ხდებზ ისე, რომ ხდება სახელების კონფლიქტი, ანუ ცვლადების კოლოზია. კოლოზიების მაგალითებია:

(λx.yx)[y <- λz.z] = λx.(λz.z)x = λx.x — კორექტული ჩასმა.

(λx.yx)[y <- xx] = λx.(xx)x — ცვლადების სახელების კოლიზია.

(λz.yz)[y <- xx] = λz.(xx)z — კორექტული ჩასმა.

ბაზისური ჩასმების ზუსტი განსაზღვრებები: **x[x <- E'] = E'**

1. **y[x <- E'] = y**
2. **(λx.E)[x <- E'] = λx.E**
3. **(λy.E)[x <- E'] = λy.E[x <- E']**, იმ პირობით, რომ **y in FV (E')**
4. **(λy.E)[x <- E'] = (λz.E[y <- z])[x <- E']**, იმ პირობით, რომ **y !in FV (E')**
5. **(E_1E_2)[x <- E'] = (E_1[x <- E']E_2[x <- E'])**

ფორმალური სისტემის აგება

ამრიგად, უკვე მზად ვართ, რათა გადავიდეთ ფორმალური სისტემის აგებამდე, რომელიც აღწერს ფუნციონალურ პროგრამირებას λ–აღრიცხვის ტერმინებში.

ფორმულების აგების წესები ასე გამოიყურება: **Exp = Exp.**

აქსიომები:

(α): |- λx.E = λy.E[x <- y]

(β): |- (λx.E)E' = E[x <- E']

(ρ): |- t = t, იმ შემთხვევაში,თუ **t** — იდენტიფიკატორებია

გამოყვანის წესები:

(μ): $t_1 = t_2$ => $t_1t_3 = t_2t_3$

(ν): $t_1 = t_2 \Rightarrow t_3t_1 = t_3t_2$

(σ): $t_1 = t_2 \Rightarrow t_2 = t_1$

(τ): $t_1 = t_2,\ t_2 = t_3 \Rightarrow t_1 = t_3$

(ξ): $t_1 = t_2 \Rightarrow \lambda x.t_1 = \lambda x.t_2$

მაგალითი 27. დავამტკიცოთ ფორმულის გამოყვანადობა: (λx.xy)(λz.(λu.zu))v = (λv.yv)v

(λx.xy)(λz.(λu.zu))v = (λv.yv)v

(μ): (λx.xy)(λz.(λu.zu)) = (λv.yv)

(β): z.(λu.zu))y = (λv.yv)

(α): u.yu = λv.yv — გამოყვანადია.

ფუნქციონალური პროგრამირების ფორმალიზაციის მეორე ვარიანტში შეიძლება ვისარგებლოთ არა სიმეტრიული თვისების დამოკიდებულებით „=“, არამედ არასიმეტრიული დამოკიდებულებით „->“.

მეორე ფორმალურ სისტემაში ფორმულის გამოყვანის წესი აბსოლუტურად იგივეა, რაც პირველ ვარიანტში. თუმცა აქსიომები ღებულობენ სხვა სახეს:

(α'): |- λx.M -> λy.M[x <- y]

(β'): |- (λx.M)N -> M[x <- N]

(ρ'): |- M -> M

გამოყვანის წესი ფორმალური სისტემის მეორე ვარიანტში ერთია:

(π): t_1 -> t_1', t_2 -> t_2' => t_1t_2 -> t_1't_2'

არსებითად, გამოყვანის ეს წესი ამბობს, რომ ნებისმიერ გამოსახულებაში შეიძლება გამოიყოს შემავალი ქვეგამოსახულება და შეიცვალოს იგი.

განსაზღვრება:

• გამოსახულებას λx.M ტიპისას ეწოდება α–რედექსი.

• გამოსახულებას (λx.M)N ტიპისას ეწოდება β-რედექსი.

• გამოსახულება, რომელიც არ შეიცავს β–რედექსებს, ეწოდება გამოსახულება ნორმალური ფორმით.

რამდენიმე თეორემა (დამტკიცების გარეშე):

- |- E1 = E2 => E1 -> E2 | E2 -> E1.
- E -> E1 & E -> E2 => არსებობს F : E1 -> F & E2 > F. (ჩერჩ–როსელის თეორემა).

- თუ **E** აქვს ნორმალური ფორმები E_1 და E_2, მაშინ ისინი ექვივალენტურია სიზუსტით α–კონვენსიით, ანუ განსხვავდებიან მხოლოდ დაკავშირებული ცვლადების აღნიშვნებით.

რედუქციის სტრატეგია

1. ნორმალური რედუქციური სტრატეგია. რედუქციის თითოეულ ბიჯზე ირჩევა ტექსტურად ყველაზე მარცხენა β–რედექსი. დამტკიცებულია, რომ ნორმალური რედუქციული სტრატეგია გარანტირებს გამოსახულების ნორმალური ფორმის მიღებას, თუ ის არსებობს.
2. აპლიკაციური რედუქციული სტრატეგია. რედუქციის თითოეულ ბიჯზე ამოირჩევა β–რედექსი, რომელიც არ შეიცავს თავის შიგნით სხვა β–რედექსებს. შემდეგში ვაჩვენებთ, რომ აპლიკაციურ რედუქციულ სტრატეგიას ყოველთვის არ იძლევა გამოსახულების ნორმალური ფორმის მიღების საშუალებას.

მაგალითი 28. გამოსახულების **M = (λy.x)(EE)** რედუქცია, სადაც **E = λ x.xx**

1. **HPC: (λy.x)(EE) = (λy.x)[y <- EE] = x.**

2. **APC: (λy.x)(EE) = (λy.x)((λx.xx)(λx.xx)) = (λy.x)((λx.xx)(λx.xx)) = ...**

ამ მაგალითში ჩანს, როგორ შეიძლება აპლიკაციურმა რედუქციულმა სტრატეგიამ მიგვიყვანოს ცუდ უსასრულობამდე. **M** გამოსახულების ნორმალური ფორმის მიღება აპლიკაციური რედუქციის სტრატეგიის გამოყენების დროს შეუძლებელია.

შენიშვნა: წითელი ფერით გამოყოფილია β რედექსი, რომელიც შემდეგი ბიჯით რედუცირდება.

მაგალითი 29. გამოსახულების **M = (λx.xyxx)((λz.z)w)** რედუქცია

1. **HPC: (λx.xyxx)((λz.z)w) = ((λz.z)w)y((λz.z)w)((λz.z)w) = wy((λz.z)w)((λz.z)w) = wyw((λz.z)w) = wyww.**

2. **APC: (λx.xyxx)((λz.z)w) = (λx.xyxx)w = wyww.**

პროგრამირებაში ნორმალური რედუქციული სტრატეგია შეესაბამება გამოძახებას სახელით, ანუ გამოსახულების არგუმენტი არ გამოითვლება მანამ, სანამ გამოსახულების ტანში არ გაჩნდება მასზე მიმართვა. აპლიკაციურ რედუქციულ სტრატეგიას შეესაბამება მნიშვნელობით გამოძახება.

შესაბამისობა ფუნქციონალური პროგრამის გამოთვლებსა და რედუქციას შორის

ინტერპრეტატორის მუშაობა აღიწერება რამდემინე ნაბიჯით:

1. გამოსახულებაში აუცილებელია გამოიყოს რაღაც მიმართვა რეკურსიულ ან ჩადგმულ ფუნქციებს შორის სრულად განსაზღვრული არგუმენტებით. თუ ჩადგმულ ფუნქციაზე გამოყოფილი მიმართვა არსებობს, მაშინ ხდება მისი შესრულება და დაბრუნება პირველ ნაბიჯზე.
2. თუ პირველ ნაბიჯზე გამოყოფილია რეკურსიულ ფუნქციაზე მიმართვა, მაშინ მის ნაცვლად ჩაისმევა ფუნქციის ტანი ფაქტიური პარამეტრებით (რადგანაც ისინი უკვე აღნიშნულია) . შემდეგ ხდება გადასვლა პირველი ბიჯის დასაწყისზე.
3. თუ მეტი მიმართვა არ არის, მოხდება გაჩერება.

ფუნქციონალურ პროგრამირებში გამოთვლები, პრინციპში, იმეორებს რედუქციის ნაბიჯებს, მაგრამ დამატებით შეიცავენ ჩადგმული ფუნქციების გამოთვლას.

განსაზღვრული ფუნქციის წარმოდგენა λ-გამოსახულების სახით

ნაჩვენებია, რომ ფუნქციის ნებისმიერი განსაზღვრება შეიძლება წარმოდგეს λ–გამოსახულების სახით, რომელიც არ შეიცავს რეკურსიას. მაგალითად:

```
fact = λn.if n == 0 then 1 else n * fact (n - 1)
```

იგივე განსაზღვრება შეიძლება აღიწეროდ რომელიღაც ფუნქციონალის გამოყენებით:

```
fact = (λf.λn.if n == 0 then 1 else n * f (n - 1)) fact
```

წარმოდგენილ გამოსახულებაში მსხვილი შრიფტით გამოყოფილია ფუნქციონალი **F**. ამრიგად, ფაქტორიალის გამოთვლის ფუნქცია შეიძლება ჩავწეროთ ასე: **fact = F fact**. ნებისმიერი რეკურსიული განსაზღვრება **f** ფუნქციისა შეიძლება წარმოდგეს ასეთი სახით:

```
f = F f
```

ეს გამოსახულება შეიძლება განვიხილოთ როგორც განტოლება, რომელშიც რეკურსიული ფუნქცია **f** წარმოადგენს **F** ფუნქციონალის უძრავ წერტილს. შესაბამისად, ფუნქციონალური ენის ინტერპრეტატორი შეიძლება განვიხილოთ როგორც რიცხვითი მეთოდი ამ განტოლების ამოსახსნელად.

შეიძლება გამოვთქვათ ვარაუდი, რომ ეს რიცხვითი მეთოდი (ანუ ინტერპრეტატორი) თავის მხრივ შეიძლება რეალიზებული იყოს რომელიღაც **Y** ფუნქციის საშუალებით, რომელიც **F** ფუნქციონალისთვის პოულობს მის უძრავ წერტილს. შესაბამისად განსაზღვრავს საძებნ ფუნქციას **— f = Y F.**

Y ფუნქციის თვისებები განისაზღვრება ტოლობით:

```
Y F = F (Y F)
```

თეორემა (დამტკიცების გარეშე) :

ნებისმიერ λ-ტერმს აქვს უძრავი წერტილი.

λ-აღრიცხვა საშუალებას იძლევა ნებისმიერი ფუნქცია გამოხატული იყოს წმინდა λ- გამოსახულებით ჩადგმული ფუნქციების გამოყენების გარეშე. მაგალითად:

1.

```
prefix = λxyz.zxy
head = λp.p(λxy.x)
tail = λp.p(λxy.y)
```

2. პირობითი გამოსახულების მოდელირება:

```
True = λxy.x
False = λxy.y
if B then M else N = BNM, где B in {True, False}.
```

11. პროგრამების ტრანსფორმაცია

P პროგრამის ქვეშ რომელიღაც L ენაზე გაიგება რაღაც ტექსტი L–ზე. ფუნქციონალური პროგრამის ქვეშ გაიგება კლოზების ნაკრები. L ენის სემანატიკა განისაზღვრება, თუ მოცემულია ამ ენის ინტერპრეტატორი. ინტერპრეტატორი განისაზღვრება ფორმულით:

Int (P, d) = d'

სადაც:

P — პროგრამაა;

d — საწყისი მონაცემები;

d' — გამოსასვლელი მონაცემები.

თუ ინტერპრეტატორი Int წარმოდგენილია კარირებული f ფუნციის სახით ისე, რომ f P d = d', მაშინ განსაზღვრება f = M f, უფრო ზუსტად M ფუნქციონალს ეწოდებენ L ენის დენოტაციურ სემენტიკას. ამ შემთხვევაში აზრი აქვს λ–გამოსახულებას: f P = λd.M' : D -> D'. ამასთან ნაწილობრივი ფუნქცია f P შეიძლება განვიხილოთ, როგორც ერთარგუმენტიანი ფუნქცია f_P. ეს არის ფუნქცია, რომელიც ახდებს პროგრამა P–ს რეალიზებას. როგორც უკვე ვაჩვენეთ, შეიძლება აიგოს რეკურსიული განმარტება შემდეგი სახის: $f_P = M_P\ f_P$. ასეთი სახე აქვს თავდაპირველად ყველა ფუნქციას ფუნქციონალურ ენაზე, მაგრამ ეს ჩანაწერი შეიძლება გავიგოთ ორნაირად:

- ეს განმარტება შეიძლება გავიგოთ როგორც სიმბოლოების სტრიქონი, რომელიც ეძლევა შესასვლელზე ინტერპრეტატორს. ფუნქცია, რომელსაც ინტერპრეტატორი ითვლის ტექსტის მიხედვით f = M f, აღინიშნება როგორც f_{int}.
- f = M f — არის f ფუნქციის წმინდა მათემატიკური განსაზღვრება. ამ განტოლების ამოხსნა აღინიშნება როგორც f_{mat}.

ისმის კითხვა: რა კავშირშია ეს ორი ფუნქცია— f_{int} და f_{mat}? უნდა ვიმედოვნოთ, რომ კორექტული ინტერპრეტატორი სწორედ ითვლის f_{mat}–ს.

განმარტება:

ამბობენ, რომ ფუნქცია f_1 ნაკლებად განსაზღვრულია, ვიდრე ფუნქცია f_2 (აღინიშნება როგორც $f_1 \subseteq f_2$), თუ $\forall x: f_1 x = y \Rightarrow f_2 x = y$. იმ შემთხვევაში, როცა ორი ფუნქციისთვის ერთდროულად სრულდება $f_1 \subseteq f_2$ და $f_2 \subseteq f_1$, მაშინ ადგილი აქვს ფუნქციების იგივეობას.

როგორც წესი, $f_{int} \subseteq f_{mat}$ — ეს ხდება იმიტომ, რომ ჩვეულებრივი ინტერპრეტატორი რეალიზებს რედუქციის აპლიკაციურ სტრატეგიას. თუმცა

შემდგომში ჩვენ ვიგულისხმებთ ფუნქციების f_{int} და f_{mat} იგივეობას, ამიტომ პროგრამების ტექსტს განვიხილავთ როგორც ფუნქციის მათემატიკურ განმარტებას. მაშინ ფუნქციონალური პროგრამების ექვივალენტური გარდაქმნა არის მათემატიკურად განსაზღვრული ფუნქციის უბრალოდ სპეციალური სახის ექვივალენტური გარდაქმნა.

პროგრამების ტრანსფორმაცია – არის არის უბრალოდ სინტაქსური გარდაქმნა, რომლის დროსაც საერთოდ არ არის პროგრამის სემანტიკასთან შეხება, ვინაიდან პროგრამა გაიგება როგორც სიმბოლოების ნაკრები. იმ ფაქტის აღნიშვნა, რომ ერთი ტექსტი f_1 მიიღება სინტაქსური ტრანსფორმაციის შედეგად მეორე f_2 ტექსტიდან, წარმოდგება ასე: `f1 |- f2.`

ამბობენ, რომ გარდაქმნა კორექტულია, თუ $f_1 \subseteq f_2$.

ამბობენ, რომ გარდაქმნა ექვივალენტურია, თუ $f_1 \equiv f_2$.

შემოდის კიდევ რამდენიმე სპეციალური აღნიშვნა:

გვაქვს კლოზების საწყისი ნაკრები (ანუ გარდაქმნის ობიექტების) და ეს ნაკრები აღნიშნავს ან `DEF`, ან `SPEC`.

1. კლოზები, რომლებიც აღწერენ ფუნქციას, რომლებიც შეიცავს საწყისი კლოზებიდან ასახვას, ავღნიშნოთ, როგორც `INV`.

2. ზოგიერთი ტოლობა, რომლებიც გამოხატავენ ფუნქციის შიდა (დარეზერვირებულ) თვისებას, აღინიშნება როგორც `LOW`.

განმარტება:

ვთქვათ `F (X)` — რაღაც გამოსახულებაა (ტოლობაა), რომელშიც გამოყოფილია `X` ტერმის ყველა თავისუფალი შესვლა. მაშინ `F [X <- M]`–ს უწოდებენ `F`–ის მაგალითს. ამასთან, ითვლება, რომ `M` — ეს რაღაც გამოსახულებაა.

გარდაქმნის სახეები

1°. კონკრეტიზაცია (instantiation) — `INS`.

$$\frac{E_1(X) = E_2(X)}{E_1[X \leftarrow N] = E_2[X \leftarrow N]}$$

2°. გარდაქმნა სახელის გარეშე :)

$$\frac{M(Y) = N(Y), E_1 = E_2[X \leftarrow M'], M' = M[Y \leftarrow G]}{E_1 = E_2[X \leftarrow N'] \quad \text{სადაც} \quad N' = N[Y \leftarrow G]}$$

3°. გაშლა(unfolding) — `UNF`.

$$\frac{M(Y) = N(Y), E_1 = E_2(M'), M' = M[Y \leftarrow G]}{E_1 = E_2(N'), \text{ სადაც } N' = N[Y \leftarrow G]}$$

4°. შეფუთვა (folding) — FLD.

$$\frac{M(Y)=N(Y), E_1=E_2(N'), N'=N[Y\leftarrow G]}{E_1=E_2(M'),\ \text{სადაც}\quad M'=M[Y\leftarrow G]}$$

5°. კანონი (law) — LAW.

$$\frac{M(Y)=N(Y), E_1=E_2(M'), M'=M[Y\leftarrow G]}{E_1=E_2(N'), \text{სადაც } N'=N[Y\leftarrow G]}$$

6°. აბსტრაქცია და აპლიკაცია (abstraction & application) — ABS.

$$\frac{M[X\leftarrow G]=(\lambda X.M)G, E_1=E_2(M[X\leftarrow G]}{E_1=E_2((\lambda X.M)G)}$$

მაგალითი 30. ფუნქცია `length`–ის გარდაქმნა.

```
length [] = 0                                  1 (DEF)
length (H:T) = 1 + length T                          2 (DEF)
length_2 L1 L2 = length L1 + length L2         3 (SPEC)

length_2 [] L      = length [] + length L            4 (INS 3)
             = 0 + length L                    5 (UNF 1)
             = length L                        6 (LAW +) (*)

length_2 (H:T) L= length (H:T) + length L      7 (INS 3)
             = (1 + length T) + length L             8 (UNF 2)
             = 1 + (length T + length L)             9 (LAW +)
             = 1 + length_2 T L                10 (FLD 3) (**)
```

ეხლა დაგვრჩა, ავიღოთ ორი მიღებული კლოზი (*) და (**) შევადგინოთ მათგან ახალი რეკურსიული განმარტება ახალი ფუნციისა, რომელიც არ იყენებს ძველი ფუნქციის გამოძახებას:

```
length_2 [] L = length L
length_2 (H:T) L = 1 + length_2 T L
```

უნდა ავღნიშნოთ, რომ ახალი კლოზების არჩევა ახალი განმარტებისთვის ითხოვს დამატებით კვლევას და არ სრულდება ავტომატურად.

ფუნქციის განსაზღვრის ასეთი ტრანსფორმაცია ხშირად მიგვიყვანს ფუნქციის სირთულის შემცირებამდე. მაგალითად, ფუნქციისთვის, რომელიც ითვლის ფიბონაჩის N–ურ რიცხვს, შეიძლება ავაგოთ განსაზღვრება, რომლის გამოთვლის

სიზუსტე წრფივად არის N–ზე დამოკიდებული, და არა ფიბონაჩის წესების მიხედვით, რომეორც ეს არის ჩვეულებრივ განსაზღვრებაში.

მაგრამ, ტრანსფორმაცია უნდა შესრულდეს მოფიქრებულად, რადგანაც შეიძლება მივიდეთ FLD და UNF ბიჯების უსასრულო ციკლამდე. რათა ტრანსფორმაციისას არ მივიდეთ აბსურდამდე, უნდა ყურადღება მივაქციოთ იმას, რომ გარდაქმნის პროცესში მიღებული გამოსახულებების ზოგადობა არ გაიზარდოს, ანუ ტრანსფორმაცია უნდა განხორციელდეს ზოგადიდან კერძოსკენ.

ინფორმატიკის მეორე კანონი

- არსებობს ამოუხსნადი ამოცანები.
- არ არსებობს ეფექტური რეალიზაცია დეკლარატიული ენებისთვის, თუ ისინი არის უნივერსლური.

ტრანსფორმაციული სინთეზის კონცეფცია: საშუალებას აძლევს პროგრამისტს დაწეროს ფუნქციის განსაზღვრება ისე, რომ არ იზრუნოს მის ეფექტურობაზე.

თუმცა, დამტკიცდა, რომ სპეციფიკაციების ენის მიხედვით არ შეიძლება გამომუშავდეს (ანუ ტრანსფორმირდეს საწყისი ტექსტი) ფუნქციის ვარიანტი, რომელიც მუშაობს ეფექტურად. თუ სპეციფიკაციის ენად განვიხილავთ ფუნქციონალურ ენას, მაშინ ცხადი ხდება, რომ პროგრამისტმა თავად უნდა იზრუნოს თავისი პროგრამის ეფექტურობაზე – ტრანსფორმაციული სინთეზის კონცეფცია აქ არ გამოდგება.

ნაწილობრივი გამოთვლები

ვთქვათ P და S — ორი ენაა, რომელიც მუშაობს სიმბოლოების სტრიქონებთან (ეს არ არღვევს განხილვის ზოგადობას), ხოლოa **P** და **S** — სინტაქსურად სორი პროგრამების ერთობლიობაა. D —ყველა შესაძლო სიმბოლოების თანმიმდევრობის დომენია.

```
P :: D -> (D* -> D)
```

თუ p — პროგრამაა P–დან, მაშინ:

```
P (p) :: D* -> D
```

```
P (p) (<d1, ..., dn>) = d, და d in D
```

```
P (r) (<y1, ..., yn>) = P (p) (<d1, ..., dm, y1, ..., yn>)
```

ბოლო ტოლობაში ცვლადებით yi აღნიშნულია უცნობი მონაცემები. პროგრამა p–თვის ეს n ცვლადი წარმოადგენს დარჩენილ კოდს.

ნაწილობრივი გამომთველი MIX ეწოდება პროგრამას **P**–დან (თუმცა ნაწილობრივი გამომთვლელი შეიძლება რეალიზებული იყოს ნებისმიერ ენაზე), ისეთს, რომ:

$$\forall p \in \mathbf{P}, p(x_1,\ldots,x_m,x_{m+1},\ldots,x_n) : P(MIX)(< p, d_1,\ldots,d_m >) = r$$

ანუ MIX – ეს არის პროგრამა, რომელიც ღებულობს რა საწყის პროგრამას და მონაცემებს მისი ცნობილი პარამეტრებისთვის, გვაძლევს საწყისისთვის დარჩენილ პროგრამას.

ენა S–ის ინტერპრეტატორი ეწოდება პროგრამას INT in **P**, ისეთს, რომ:

$$\forall s \in \mathbf{S}, \langle d_1,\ldots,d_n \rangle \in D^* : P(INT)(< s, d_1,\ldots,d_n >) = S(s)(< d_1,\ldots,d_n >)$$

S ენის კომპილერი ეწოდება პროგრამას COMP in **P**, ისეთს, რომ:

$$P(COMP)(< s >) = TARGET$$
$$P(TARGET)(< d_1,\ldots,d_n >) = S(s)(< d_1,\ldots,d_n >)$$

ანუ იგივეა:

$$P(P(COMP)(< s >)(< d_1,\ldots,d_n >) = S(s)(< d_1,\ldots,d_n >)$$

S ენის კომპილატორების კომპილატორი ეწოდება პროგრამას COCOM in **P**, ისეთს, რომ:

$$P(COCOM)(< INT >) = COMP$$
$$P(P(P(COCOM)(< INT >))(< s >))(< d_1,\ldots,d_n >) = S(s)(< d_1,\ldots,d_n >)$$

ფუტამორების პროექციები:

- TARGET = P (MIX) (<INT, s>)
- COMP = P (MIX) (<MIX, INT>)
- COCOM = P (MIX) (<MIX, MIX>)

ეს სამი დებულება არის თეორემები, რომლებიც საკმაოდ ადვილად შეიძლება დამტკიცდეს ზემოთ მოყვანილი განმარტებებით.