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)
```