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

b. listSumm:

```
listSumm                  :: Ord (a) => [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
```