ლაბორატორიული სამუშაო
თემა №1

1 მიზნები

ენა Haskell-ის ინტერპრეტატორთან მუშაობის უნარ-ჩვევების შეძენა. წარმოდგენის შექმნა ენა Haskell-ის ძირითად ტიპებზე. მარტივი ფუნქციების განსაზღვრებების შესწავლა.

2 Hugs ინტერპრეტატორთან მუშაობის საფუძვლები

ენა Haskell-ზე პროგრამირებისთვის საჭიროა გამოვიყენოთ ინტერპრეტატორი. არსებობს ინტერპრეტატორის რამდენიმე რეალიზაცია. ამ კურსში გამოვიყენებთ ინტერპრეტატორს **Hugs** (მისი დასახელება არის შემოკლება სიტყვებიდან "Haskell users' Gofer system", Gofer - პროგრამირების ენის სახელწოდებაა, რომელიც იყო Haskell-ის ერთ-ერთი წინამორბედი.)

Hugs ინტერპრეტატორის გაშვების შემდეგ ეკრანზე გამოჩნდება დამუშავების გარემოს დიალოგური ფანჯარა, ავტომატურად ჩაიტვირთება სპეციალური ფაილი (`Prelude.hs`), რომელიც შეიცავს ტიპების წინასწარ განსაზღვრებებს და Haskell ენის სტანდარტულ ფუნქციებს, და ეკრანზე გამოდის სტანდარტული მოსაწვევი მუშაობის დასაწყებად. ამ მოსაწვევს აქვს სახე `Prelude>`. საზოგადოდ, სიმბოლო >-ის წინ გამოდის იმ მოდულის სახელი, რომელიც ბოლოს ჩაიტვირთა.

მოსაწვევის გამოტანის შემდეგ შეიძლება ავკრიფოთ ენის გამოსახულება ან ინტერპრეტატორის ბრძანება. ინტერპრეტატორის ბრძანება განსხვავდება Haskell-ის გამოსახულებისგან იმით, რომ იწყება სიმბოლო ორიწერტილით (`:`). მაგალითად, ინტერპრეტატორის ბრძანებაა `:quit`, რომელის შესრულებაც იწვევს ინტერპრეტატორის მუშაობის დამთავრებას. ინტერპრეტატორის ბრძანებები შეიძლება შემოკლდეს ერთ ასომდე. ასე, რომ, ბრძანებები `:quit` და `:q` ექვივალენტურია. ბრძანება `:set` გამოიყენება ინტერპრეტატორის სხვადასხვა ოფციების დასაყენებლად. ბრძანება `:?`-ს გამოჰყავს ინტერპრეტატორის ყველა ბრძანების სია. შემდგომში ჩვენ სხვა ბრძანებებსაც განვიხილავთ.

3 ტიპები

Haskell ენის პროგრამები არიან გამოსახულებები, რომელთა გამოთვლებს მივყავართ მნიშვნელობებამდე. თითოეულ მნიშვნელობას აქვს *ტიპი*. ინტუიციურად,

ტიპი შეიძლება გავიგოთ როგორც გამოსახულების დასაშვები მნიშვნელობების სიმრავლე. იმისათვის, რომ განვსაზღროთ, რა ტიპი აქვს მოცემულ გამოსახულებას, საჭიროა გამოვიყენოთ ინტერპრეტატორის ბრძანება `:type` (ან `:t`). ამას გარდა, შესაძლოა გამოვიყენოთ ბრძანება `:set +t`, რათა ინტერპრეტატორმა ავტომატურად დაბეჭდოს ყოველი გამოთვლილი გამოსახულების ტიპი.

ენა Haskell-ის ტიპებს წარმოადგენს:

- ტიპები `Integer` და `Int` - გამოიყენება მთელი რიცხვების წარმოსადგენად, ამასთან ტიპი `Integer`-ის სიგრძე არ არის შეზღუდული.
- ტიპები `Float` და `Double` გამოიყენება ნამდვილი რიცხვების წარმოსადგენად.
- ტიპი `Bool` შეიცავს ორ მნიშვნელობას: `True` და `False` და დანიშნულია ლოგიკური გამოსახულებების შედეგის წარმოსადგენად.
- ტიპი `Char` გამოიყენება სიმბოლოების წარმოსადგენად.

ტიპების სახელები Haskell ენაში ყოველთვის იწყება დიდი (მთავრული) ასოებით.

ენა Haskell წარმოადგენს *ძლიერ ტიპიზირებულ* პროგრამირების ენას. მიუხედავად ამისა, ხშირ შემთხვევებში პროგრამისტი არ არის ვალდებული განაცხადოს, თუ რომელ ტიპს ეკუთვნის მის მიერ შემოტანილი ცვლადი. ინტერპრეტატორს თვითონ აქვს შესაძლებლობა *გამოიყვანოს* მომხმარებლის მიერ გამოყენებული ცვლადის ტიპი. თუმცა, თუ რაიმე მიზნისთვის საჭიროა გამოცხადდეს, რომ მნიშვნელობა ეკუთვნის რომელიღაც ტიპს, გამოიყენება კონსტრუქცია: *ცვლადი* `::` *ტიპი*. თუ ჩართულია ინტერპრეტატორის ოფცია `+t`, მაშინ ინტერპრეტატორი ამავე ფორმატში დაბეჭდავს მნიშვნელობებს.

ქვემოთ მოყვანილია ინტერპრეტატორთან მუშაობის სესია. იგულისხმება, რომ ტექსტი მოსაწვევი `Prelude>`-ის შემდეგ, შეყვანილია მომხმარებლის მიერ, ხოლო მისი მომდევნო ტექსტი არის სისტემის პასუხი.

```
Prelude>:set +t
Prelude>1
1 :: Integer
Prelude>1.2
1.2 :: Double
Prelude>'a'
'a' :: Char
Prelude>True
True :: Bool
```

მოცემული ოქმიდან ჩანს, რომ ტიპები `Integer`, `Double` და `Char`-ის მნიშვნელობები მოიცემა იგივე წესებით, როგორც ენა C-ში.

ტიპების სისტემის განვითარებისა და მკაცრი ტიპიზაციის შედეგად Haskell ენის პროგრამები არის *უსაფრთხო ტიპების* მიხედვით. გარანტირებულია, რომ Haskell ენის სწორ პროგრამაში ტიპები სწორად გამოიყენება. პრაქტიკულად, ეს ნიშნავს, რომ Haskell ენაზე პროგრამა შესრულებისას ვერ გამოიწვევს შეცდომას მეხსიერების შეღწევადობაზე (Access violation). ასევე, გარანტირებულია, რომ პროგრამაში ცვლადების გამოყენება საწყისი ინიციალიზების გარეშე არ მოხდება. ამრიგად, პროგრამაში მრავალი შეცდომის არსებობა მოწმდება კომპილაციის ეტაპზე და არა შესრულების ეტაპზე.

4 არითმეტიკა

ინტერპრეტატორი **Hugs** შეიძლება გამოყენებული იყოს არითმეტიკული გამოსახულებების გამოსათვლელად. ამასთან, შესაძლოა გამოყენებული იყოს ოპერატორები: `+`, `-`, `*`, `/` (მიმატება, გამოკლება, გამრავლება, გაყოფა) პრიორიტეტების ჩვეულებრივი წესებით. ამასთან, შეიძლება გამოყენებული იყოს ოპერატორი `^` (ხარისხში აყვანა). ამრიგად, მუშაობის სეანს შეიძლება ჰქონდეს შემდეგი სახე:

```
Prelude>2*2
4 :: Integer
Prelude>4*5 + 1
21 :: Integer
Prelude>2^3
8 :: Integer
```

ამასთან, შეიძლება გამოვიყენოთ სტანდარტული მათემატიკური ფუნქციები `sqrt` (კვადრატული ფესვი), `sin`, `cos`, `exp` და ა.შ. პროგრამირების სხვა ენებისგან განსხვავებით, Haskell-ში ფუნქციის გამოძახებისას არ არის აუცილებელი არგუმენტის ფრჩხილებში ჩასმა. ამრიგად, შეიძლება მარტივად დაიწეროს `sqrt 2`, და არა `sqrt(2)`. მაგალითი:

```
Prelude>sqrt 2
1.4142135623731 :: Double
Prelude>1 + sqrt 2
2.4142135623731 :: Double
Prelude>sqrt 2 + 1
```

```
2.4142135623731 :: Double
Prelude>sqrt (2 + 1)
1.73205080756888 :: Double
```

ამ მაგალითიდან შეიძლება დავასკვნათ, რომ ფუნქციის გამოძახებას უფრო მაღალი პრიორიტეტი აქვს, ვიდრე არითმეტიკულ ოპერაციებს, ასე, რომ გამოსახულება `sqrt 2 + 1` ინტერპრეტირდება როგორც `(sqrt 2) + 1`, და არა როგორც `sqrt (2 + 1)`. გამოთვლების ზუსტი რიგის მისათითებლად საჭიროა გამოყენებული იყოს ფრჩხილები. ფუნქციის გამოძახებას ყოველთვის უფრო მაღალი პრიორიტეტი აქვს, ვიდრე ნებისმიერ ბინარულ ოპერაციას.

საჭიროა ასევე აღინიშნოს, რომ პროგრამირების ბევრი სხვა ენისგან განსხვავებით, მთელრიცხვებიანი გამოსახულება Haskell-ზე გამოითვლება თანრიგების შეუზღუდავი რიცხვით. (შეეცადეთ გამოთვალოთ გამოსახულება `2^5000`). C ენისგან განსხვავებით, სადაც `int` ტიპის მაქსიმალური მნიშვნელობა შეზღუდულია მანქანის თანრიგებრიობით (თანამედროვე მანქანებზე იგი ტოლია $2^{31}-1$ = 2147483647), Haskell ენაში ტიპმა `Integer`-მა შეიძლება წარმოადგინოს ნებისმიერი სიგრძის მთელი რიცხვი.

5 კორტეჟები

ზემოთ ჩამოთვლილი მარტივი ტიპების გარდა Haskell-ში შეიძლება განვსაზღვროთ შედგენილი ტიპის მნიშვნელობებიც. მაგალითად, სიბრტყეზე წერტილების მოსაცემად აუცილებელია ორი რიცხვი, რომლებიც მათ კოორდინატებს შეესაბამება. ენა Haskell-ში რიცხვების წყვილი შეიძლება მოვცეთ ასე: ჩამოვთალოთ კომპონენტები, გამოვყოთ მძიმეებით და ავიღოთ ფრჩხილებში: `(5,3)`. არ არის აუცილებელი, რომ რიცხვების კომპონენტები იყოს ერთიდაიგივე ტიპის. შეიძლება შევადგინოთ წყვილი, რომლის პირველი კომპონენტი შეიძლება იყოს სტრიქონი, მეორე მთელი რიცხვი და ა.შ.

ზოგადად, თუ `a` და `b` Haskell ენის ნებისმიერი ტიპია, მაშინ იმ წყვილის ტიპი, რომელშიც პირველი ელემენტი ეკუთვნის ტიპს `a`, ხოლო მეორე - ტიპს `b`, აღინიშნება ასე: `(a,b)`. მაგალითად, წყვილს `(5,3)`აქვს ტიპი `(Integer, Integer)`; წყვილი `(1,'a')` ეკუთვნის ტიპს `(Integer, Char)`. შეიძლება მოვიყვანოთ უფრო რთული მაგალითი: წყვილი `((1,'a'),1.2)` ეკუთვნის ტიპს `((Integer,Char),Double)`. შეამოწმეთ ეს ინტერპრეტატორის საშუალებით.

ყურადღება მივაქციოთ იმას, რომ თუმცა შემდეგი სახის კონსტრუქციები `(1,2)` და `(Integer,Integer)` მსგავსად გამოიყურება, ენა Haskell-ში ისინი აღნიშნავენ სხვადასხვა ცნებებს. პირველი მათგანი წარმოადგენს მნიშვნელობას, მაშინ როცა მეორე არის ტიპი.

წყვილებთან სამუშაოდ ენა Haskell-ში არსებობს სვანდარტული ფუნქციები `fst` და `snd`, რომლებიც, შესაბამისად, აბრუნებენ სიის პირველ და მეორე ელემენტებს. ამ ფუნქციების დასახელება წარმოშობილია ინგლისური სიტყვებიდან „first“ (პირველი) და „second“ (მეორე). ამრიგად, ისინი შეიძლება გამოვიყენოთ შემდეგნაირად:

```
Prelude>fst (5, True)
5 :: Integer
Prelude>snd (5, True)
True :: Bool
```

ანალოგიურად, გარდა წყვილებისა, შეიძლება განვსაზღვროთ სამეულები, ოთხეულები და ა.შ. მათი ტიპები ჩაიწერება შესაბამისი სახით:

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

მონაცემების ასეთ სტრუქტურას უწოდებენ *კორტეჟს*. კორტეჟში შეიძლება შენახული იყოს ფიქსირებული რაოდენობის სხვადასხვა მონაცემი. ფუნქციები `fst` და `snd` განსაზღვრულია მხოლოდ წყვილებისთვის და არ მუშაობს სხვა კორტეჟებთან. თუ მათ გამოვიყენებთ, მაგალითად, სამეულთან, ინტერპრეტატორი შეგვატყობინებს შეცდომის შესახებ.

კორტეჟის ელემენტი შეიძლება იყოს ნებისმიერი ტიპის, მათ შორის სხვა კორტეჟიც. იმ კორტეჟის ელემენტებთან წვდომისთვის, რომლებიც წყვილებს წარმოადგენენ, შეიძლება გამოვიყენოთ `fst` და `snd` ფუნქციების კომბინაცია. შემდეგი მაგალითი გვიჩვენებს `'a'` ელემეტის ამოღებას კორტეჟიდან `(1, ('a', 23.12))`:

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

6 სიები

კორტეჟისაგან განსხვავებით, სიამ შეიძლება შეინახოს ელემენტების ნებისმიერი რაოდენობა. ენა Haskell-ში სია განისაზღვრება ასე: კვადრატულ ფრჩხილებში ჩამოითვლება ელემენტები და ერთმანეთისგან მძიმით გამოიყოფა. ელემენტები აუცილებლად უნდა ეკუთვნოდეს ერთიდაიგივე ტიპს. სიის ტიპი, რომელიც შედგება a ტიპის ელემენტებისგან, აღინიშნება როგორც `[a]`.

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

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

სიაში შეიძლება არცერთი ელემენტი არ შედიოდეს. ცარიელი სია აღინიშნება როგორც `[]`.

ოპერატორი `:` (ორი წერტილი) გამოიყენება სიის თავში ელემენტის დასამატებლად. მისი მარცხენა არგუმენტი უნდა იყოს ელემენტი, მარჯვენა - სია:

```
Prelude>1:[2,3]
[1,2,3] :: [Integer]
Prelude>'5':['1','2','3','4','5']
['5','1','2','3','4','5'] :: [Char]
Prelude>False:[]
[False] :: [Bool]
```

ოპერატორი `(:)` და ცარიელი სიისგან შეიძლება ავაგოთ ნებისმიერი სია:

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

ოპერატორი `(:)` მარჯვნიდან ასოციატიურია, ამიტომ ზემოთ მოყვანილ გამოსახულებაში შეიძლება გამოვტოვოთ ფრჩხილები:

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

სიის ელემენტები შეიძლება იყოს ნებისმიერი მნიშვნელობები - რიცხვები, სიმბოლოები, კორტეჟები, სხვა სიები და ა.შ.

```
Prelude>[(1,'a'),(2,'b')]
[(1,'a'),(2,'b')] :: [(Integer,Char)]
Prelude>[[1,2],[3,4,5]]
[[1,2],[3,4,5]] :: [[Integer]]
```

ენა Haskell-ში სიებთან სამუშაოდ არსებობს ფუნქციათა დიდი რაოდენობა. ჩვენ მხოლოდ ზოგიერთ მათგანს განვიხილავთ.

- ფუნქცია `head` აბრუნებს სიის პირველ ელემენტს.
- ფუნქცია `tail` აბრუნებს სიას პირველი ელემენტის გარეშე.
- ფუნქცია `length` აბრუნებს სიის სიგრძეს.

ფუნქციები `head` და `tail` განსახღვრულია არაცარიელი სიებისთვის. იმ შემთხვევაში, თუ ისინი გამოიყენება ცარიელ სიებთან, ინტერპრეტატორს გამოაქვს შეტყობინება შეცდომის შესახებ. ამ ფუნქციებთან მუშაობის მაგალითებია:

```
Prelude>head [1,2,3]
1 :: Integer
Prelude>tail [1,2,3]
[2,3] :: [Integer]
Prelude>tail [1]
[] :: Integer
Prelude>length [1,2,3]
3 :: Int
```

შევნიშნოთ, რომ ფუნქცია `length` ეკუთვნის ტიპს `Int` და არა `Integer`-ს.

სიების გაერთიანებისთვის (კონკატენაციისთვის) Haskell-ში განსაზღვრულია ოპერატორი `++`.

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

7 სტრიქონები

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

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

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

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

რიცხვითი მნიშვნელობების გარდაქმნისთვის სტრიქონებად და პირიქით, არსებობს ფუნქციები `read` და `show`:

```
Prelude>show 1
"1" :: [Char]
Prelude>"Formula " ++ show 1
"Formula 1" :: [Char]
Prelude>1 + read "12"
13 :: Integer
```

იმ შემთხვევაში, თუ ფუნქცია `show` ვერ გარდაქმნის სტრიქონს რიცხვად, გამოდის შეცდომა.

8 ფუნქციები

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

- ბრძანება `:load` საშუალებას იძლევა ჩაიტვირთოს Haskell პროგრამა მოცემული ფაილიდან.
- ბრძანება `:edit` უშვებს ბოლო ჩატვირთული ფაილის რედაქტირების პროცესს.
- ბრძანება `:reload` თავიდან კითხულობს ბოლოს ჩატვირთულ ფაილს.

მომხმარებლის მიერ განსაზღვრული ფუნქცია უნდა იყოს ფაილში, რომელიც ჩაიტვირთება **Hugs** ინტერპრეტატორით ბრძანება `:load`-ის საშუალებით. ჩატვირთული ფაილის რედაქტირებისთვის შეიძლება გამოვიყენოთ ბრძანება `:edit`. ის გაუშვებს გარე რედაქტორს (შეთანხმების პრინციპით - ეს არის Notepad). რედაქტირების პროცესის დამთავრების შემდეგ აუცილებელია დავხუროთ რედატორი. თუმცა, ფაილი შეიძლება შევცვალოთ უშუალოდ Windows ოპერაციული სისტემის გარსიდან. ამ შემთხვევაში, იმისთვის რომ ინტერპრეტატორმა თავიდან წაიკითხოს ფაილი, საჭიროა ცხადად გამოვიძახოთ ბრძანება `:reload`.

განვიხილოთ მაგალითი. შევქმნათ რომელიღაც კატალოგში ფაილი `lab1.hs`. დავუშვათ, ამ ფაილის სრული გზაა - `c:\labs\lab1.hs` . **Hugs** ინტერპრეტატორში შევასრულოთ შემდეგი ბრძანება:

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

თუ ჩატვირთვა წარმატებით დამთავრდა, ინტერპრეტატორის მოსაწვევი იცვლება `Main>`-ით. საქმე იმაშია, რომ თუ არ მივუთითებთ მოდულის სახელს, ითვლება, რომ ის არის `Main`.

```
Main>:edit
```

აქ უნდა გაიხსნას რედაქტორის ფანჯარა, რომელშიც უნდა შევიტანოთ პროგრამის ტექსტი. შევიტანოთ:

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

შევინახეთ ფაილი და დავხუროთ რედაქტორი. ინტერპრეტატორი **Hugs** ჩატვირთავს ფაილს `c:\labs\lab1.hs` და ეხლა `x` ცვლადის მნიშვნელობა იქნება განსაზღვრული:

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

ყურადღება მივაქციოთ, რომ ფაილის სახელის ჩაწერისას `:load` ბრძანების არგუმენტში სიმბოლო `\` დუბლირდება. ისევე, როგორც ენა C-ში, ენა Haskell-შიც სიმბოლო `\`–ით იწყება მოსამსახურე სიმბოლოები (`'\n'` და ა.შ.). თვითონ სიმბოლო `\`–ის გამოყენებისთვის საჭიროა კიდევ ერთი `\`–ის მითითება, ისევე, როგორც C ენაშია.

გადავიდეთ ფუნქციის განსაზღვრაზე. ზემოთ აღწერილი პროცესის შესაბამისად შევქმენათ რაიმე ფაილი და ჩავწეროთ მასში შემდეგი ტექსტი:

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

პირველი სტრიქონი `square :: Integer -> Integer` მიუთითებს, რომ ჩვენ შემოგვაქვს ფუნქცია `square`–ს განსაზღვრება, რომლის პარამეტრია `Integer` ტიპის და აბრუნებს `Integer` ტიპის მნიშვნელობას. მეორე სტრიქონი `square x = x * x` წარმოადგენს უშუალოდ ფუნქციის აღწერას. ფუნქცია `square` ღებულობს ერთ არგუმენტს და აბრუნებს მის კვადრატს.

ფუნქციები ენა Haskell–ში წარმოადგენენ „პირველი კლასის“ მნიშვნელობებს. ეს ნიშნავს, რომ ისინი არიან „თანაბარუფლებიანი“ ისეთი მნიშვნელობების, როგორიცაა მთელი და ნამდვილი რიცხვები, სიმბოლოები, სტრიქონები, სიები და ა.შ. ფუნქციები შეიძლება გადაეცეს სხვა ფუნქციებს არგუმენტად, დაბრუნდეს როგორც მნიშვნელობები და ა.შ. ისევე როგორც ყველა მნიშვნელობას Haskell–ში, ფუნქციასა აქვს ტიპი. ფუნქციის ტიპი, რომელიც იღებს `a` ტიპის მნიშვნელობას და აბრუნებს `b` ტიპის მნიშვნელობას, აღინიშნება ასე: `a->b`.

შექმნილი ფაილი ჩავტვირთოთ ინტერპრეტატორში და შევასრულოთ შემდეგი ბრძანებები:

```
Main>:type square
```

```
square :: Integer -> Integer
Main>square 2
4 :: Integer
```

შევნიშნოთ, რომ `square` ფუნქციის ტიპის გამოცხადება არ იყო აუცილებელი: ინტერპრეტატორს თვითონ შეუძლია აუცილებელი ინფორმაციის გამოყვანა ფუნქციის ტიპის შესახებ მისი აღწერიდან. თუმცა, ჯერ ერთი, გამოყვანილი ტიპი იქნებოდა უფრო დიდი, ვიდრე `Integer -> Integer`, მეორეც, Haskell ენაზე პროგრამირებისას ფუნქციის ტიპის ცხადად მითითება ითვლება „კარგ ტონად“, რადგანაც ტიპის გამოცხადება ემსახურება რაღაც აზრით ფუნქციის დოკუმენტაციას და ეხმარება პროგრამირების შეცდომების გამოვლენას.

მომხმარებლია მიერ განსაზღვრული ფუნქციებისა და ცვლადების სახელები უნდა იწყებოდეს პატარა (ქვედა რეგისტრის) ლათინური ასოებით. სახელებში სხვა სიმბოლოები შეიძლება იყოს დიდი ან პატარა ლათინური ასოები, ციფრები ან სიმბოლო `_` და `'` (ხაზგასმა და აპოსტროფი). ასე, რომ ქვემოთ ჩამოთვლილია ცვლადების სწორი სახელები:

```
var
var1
variableName
variable_name
var'
```

9 პირობითი გამოსახულებები

ენა Haskell–ში ფუნქციის განსაზღვრებისას შესაძლებელია გამოყენებული იყოს პირობითი გამოსახულებები. ჩავწეროთ ფუნქცია `signum`, რომელიც თვლის გადაცემული არგუმენტის ნიშანს:

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

პირობითი გამოსახულება ჩაიწერება ასე:

`if` *პირობა* `then` *გამოსახულება* `else` *გამოსახულება*.

ყურადღება გავამახვილოთ იმაზე, რომ თუმცა გარეგნულად ეს გამოსახულება ჰგავს C ან Pascal ენების ოპერატორს, ენა Haskell–ში აუცილებლად უნდა იყოს

როგორც `then`, ასევე `else` ნაწილები. გამოსახულებები პირობითი ოპერატორის `then` და `else` ნაწილებში აუცილებლად ერთიდაიგივე ტიპის უნდა იყოს.

პირობა პირობითი ოპერატორის განსაზღვრებაში წარმოადგენს ნებისმიერ `Bool`–ის ტიპის გამოსახულებას. ასეთი გამოსახულებების მაგალითად გამოდგება შედარება. შედარებისას შეიძლება გამოვიყენოთ შემდეგი ოპერატოები:

- `<, >, <=, >=` – ამ ოპერატორებს იგივე აზრი აქვთ, რაც ენა C-ში (ნაკლებია, მეტია, ნაკლებია და ტოლია, მეტია და ტოლია).
- `==` – ტოლობაზე შედარების ოპერატორი.
- `/=` – უტოლობაზე შედარების ოპერატორი.

`Bool`–ის ტიპის გამოსახულებები შეიძლება გავაერთიანოთ ლოგიკური ფუნქციებით `&&` და `||` („და“ და „ან“), და უარყოფის ფუნქციით `not`. დასაშვები პირობების მაგალითებია:

```
x >= 0 && x <= 10
x > 3 && x /= 10
(x > 10 || x < -10) && not (x == y)
```

რა თქმა უნდა, შესაძლებელია განვსაზღვროთ ფუნქციები, რომლებიც აბრუნებენ `Bool`–ის ტიპის მნიშვნელობებს და გამოვიყენოთ ისინი პირობებად. მაგალითად, შესაძლებელია განვსაზღვროთ ფუნქცია `isPositive`,რომელიც აბრუნებს `True`–ს, თუ მისი არგუმენტი არის არაუარყოფითი და `False` წინააღმდეგ შემთხვევაში:

```
isPositive :: Integer -> Bool
isPositive x = if x > 0 then True else False
```

ეხლა ფუნქცია `signum` შეიძლება განვსაზღროთ შემდეგნაირად:

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

შევნიშნოთ, რომ ფუნქცია `isPositive` შეიძლება განვსაზღვროთ მარტივადაც:

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

10 მრავალი ცვლადის ფუნქცია და ფუნქციების განსაზღვრის რიგი

აქამდე ჩვენ განვსაზღვრავდით ფუნქციებს, რომლებიც ერთ არგუმენტს იღებდნენ. რა თქმა უნდა, Haskell ენაში შესაძლებელია განვსაზღვროთ ფუნქციები, რომლებიც იღებენ ნებისმიერი რიცხვის არგუმენტებს. ფუნქცია `add`–ის

განსაზღვრას, რომელიც იღებს ორ მთელ რიცხვს და აბრუნებს მათ ჯამს, აქვს სახე:

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

ფუნქცია `add`-ის ტიპი გამოიყურება „უცნაურად“. Haskell ენაში ითვლება, რომ ოპერაცია `->` ასოციატიურია მარჯვნიდან. ასე, რომ `add` ფუნქციის ტიპი შეიძლება ასე წავიკითხოთ `Integer -> (Integer -> Integer)`, ანუ კარირების წესების მიხედვით, `add` ფუნქციის გამოყენების შედეგი ერთ არგუმენტთან იქნება ფუნქცია, რომელიც მიიღებს `Integer` ტიპის ერთ პარამეტრს. საზოგადოდ, ფუნქციის ტიპი, რომელიც ღებულობს `n` არგუმენტს, რომლებიც ეკუთვნის `t1, t2, . . . , tn`, ტიპებს და აბრუნებს `a` ტიპის შედეგს, ჩაიწერება სახით `t1->t2->...->tn->a`.

საჭიროა გავაკეთოთ კიდევ ერთი შენიშვნა ფუნქციების განსაზღვრის რიგის შესახებ. წინა პარაგრაფში ჩვენ განვსაზღვრეთ ორი ფუნქცია – `signum` და `isPositive`, ამათგან ერთი მათგანი იყენებდა თავის განსაზღვრებაში მეორეს. ისმის კითხვა, რომელი მათგანი უნდა განისაზღვროს ადრე? თითქოსდა `isPositive`–ის განსაზღვრება უნდა უსწრებდეს `signum`–ის განსაზღვრებას, მაგრამ Haskell ენაში ფუნქციების განსაზღვრის რიგს არ აქვს მნიშვნელობა! ასე, რომ ფუნქცია `isPositive` შეიძლება განისაზღვროს როგორც `signum` ფუნქციის განსაზღვრამდე, ისე მის შემდეგ.

11 დავალებები

მოიყვანეთ არატრივიალური გამოსახულებების მაგალითები, რომლებიც ეკუთვნის ტიპებს:

1) `((Char,Integer), String, [Double])`
2) `[(Double,Bool,(String,Integer))]`
3) `([Integer],[Double],[(Bool,Char)])`
4) `[[[(Integer,Bool)]]]`
5) `(((Char,Char),Char),[String])`
6) `(([Double],[Bool]),[Integer])`
7) `[Integer, (Integer,[Bool])]`
8) `(Bool,([Bool],[Integer]))`
9) `[([Bool],[Double])]`
10) `[([Integer],[Char])]`

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

2. განსაზღვრეთ შემდეგი ფუნქციები:

1) ფუნქცია `max3`, რომელიც სამი მთელი რიცხვიდან აბრუნებს მათ შორის უდიდესს.

2) ფუნქცია `min3`, რომელიც სამი მთელი რიცხვიდან აბრუნებს მათ შორის უმცირესს.

3) ფუნქცია `sort2`, რომელიც ორი მთელი რიცხვიდან აბრუნებს წყვილს, რომელშიც პირველ ადგილას დგას ამ ორი რიცხვიდან უმცირესი, მეორეზე კი – უდიდესი.

4) ფუნქცია `bothTrue :: Bool -> Bool -> Bool`, რომელიც აბრუნებს `True`–ს, მაშინ და მხოლოდ მაშინ, როცა ორივე არგუმენტი არის `True`. ფუნქციის განსაზღვრისათვის არ გამოიყენოთ ლოგიკური ოპერაციები (`&&`, `||` და ა.შ.)

5) ფუნქცია `solve2::Double->Double->(Bool,Double)`, რომელიც ორი რიცხვის მიხედვით, რომლებიც წარმოადგენენ `ax + b = 0` წრფივი განტოლების კოეფიციენტებს, აბრუნებს წყვილს, რომლის პირველი ელემენტი არის `True`, თუ არსებობს ამონახსნი და `False` – წინააღმდეგ შემთხვევაში; წყვილის მეორე ელემენტი კი არის ან ფესვის მნიშვნელობა, ან `0.0`.

6) ფუნქცია `isParallel`, რომელიც აბრუნებს `True`–ს, თუ ორი მონაკვეთი, რომლებიც წარმოადგენენ ფუნქციის არგუმენტებს, არის პარალელური (ან დევს ერთ წრფეზე). მაგალითად, მნიშვნელობა გამოსახულებისა `isParallel (1,1) (2,2) (2,0) (4,2)` არის `True`, ვინაიდან მონაკვეთები `(1, 1) – (2, 2)` და `(2, 0) – (4, 2)` პარალელურია.

7) ფუნქცია `isIncluded`, რომლის არგუმენტებია სიბრტყეზე ორი წრეწირის პარამეტრები (ცენტრის კოორდინატები და რადიუსები); ფუნქცია აბრუნებს `True`–ს, თუ მეორე წრეწირი მთლიანად თავსდება პირველის შიგნით.

8) ფუნქცია `isRectangular`, რომელიც პარამეტრად ღებულობს სიბრტყეზე სამი წერტილის კოორდინატებს და აბრუნებს `True`–ს, თუ მათ მიერ შედგენილი სამკუთხედი არის მართკუთხა სამკუთხედი.

9) ფუნქცია `isTriangle`, რომელიც განსაზღვრავს, შეიძლება თუ არა მოცემულ `x, y` და `z` სიგრძის მონაკვეთებზე აიგოს სამკუთხედი.

10) ფუნქცია `isSorted`, რომელიც შესასვლელზე ღებულობს სამ რიცხვს და აბრუნებს `True`, თუ ეს რიცხვები დალაგებულია ზრდადობით ან კლებადობით.

12 საკონტროლო შეკითხვები

1. რით განსხვავდება ინტერპრეტატორის ბრძანებები `Haskell`–ის გამოსახულებებისგან?
2. ენა `Haskell`–ის ძირითადი ტიპები.
3. კორტეჟებთან მუშაობის ფუნქციები.
4. სიებთან მუშაობის ფუნქციები.
5. ცვლადებისა და ფუნქციების დასაშვები სახელები.
6. ინტერპრეტატორის ბრძანებები პროგრამების ფაილებთან სამუშაოდ.
7. პირობითი გამოსახულებები ენა `Haskell`–ში.
8. ფუნციების განსაზვრება ენა `Haskell`–ში.