Дәріс № 5-6

Haskell тіліндегі жоғарғы ретті функциялар

Функционалды тілдерде функциялар басқа функцияларға аргумент ретінде берілуі немесе нәтиже ретінде қайтарылуы мүмкін.

Анықтама

Функционалды аргуметтерді қабылдайтын функциялар ***жоғарғы реттті функциялар немесе функционалдар*** деп аталады.

Мысал. Қандай да бір тізімнің барлық элементтерінің квадратын табу **:**

map функцияны қолданамыз **.**

```
s q u a r e (N) = N _ N
s q u a r e L i s t = map ( squar e , [ 1 , 2 , 3 , 4 ] )
```

Нәтижеде[1; 4; 9; 16] тізім пайда болады.

Есеп.

Сандар тізімі берілген. Екі функцияны анықтау керек.

1) Тізім сандарының квадрат түбірлер тізімі
2) Тізім сандарының логарифм тізімі.

Функцияларды анқтайық:

1) sqrtList [] = []

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

2) logList [] = []

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

Осы 2 функцияны біріктірейік

Haskell тілінде функцияларды басқа функцияларға параметр ретінде беруге болады.

- Екі параметр қабылдайтын **transformList** функциясын анықтайық: түрлендіру функциясын және түрлену тізімін.
- **transformList f [] = []**
- **transformList f (x:xs) = f x : transformList f xs**

Енді **sqrtList** және **logList** функцияларды келесі түрде анықтайық:

sqrtList l = transformList sqrt l

logList l = transformList log l

Немесе, каррированияны ескере отырып:

sqrtList = transformList sqrt

logList = transformList log

map функциясы

(ағылш. map – бейнелеу) стандартты кітапханасында анықталған және **transformList** функцияға үқсас**.**

map функциясының түрі:

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

Бұл, **b** типті мәндеріне **a** ерікті типінің мәндерін бейнелейтін, оның бірінші аргументі **a->b** типті функция болатынын білдіреді (бұл типтер сай келуі мүмкін).

Функцияның екінші аргументі **a** типті мәндерінің тізімі болып есептеледі. Сонда функция нәтижесі **b** типті мәндер тізімі болады.

Аргумент ретінде басқа функцияларды қабылдайтын **map** тәрізді функциялар жоғарғы ретті функциялар деп аталады.

Оларды функционалды программаларды жазуда кеңінен қолданады. Олардың көмегімен алгоритмнің жүзеге асыру жеке детальдарын (мысалы, нақты функцияны **map** –қа түрлендіру) оның жоғары деңгейлі құрылымынан (тізімнің әр элементті түрленуін) ажыратуға болады.Жоғарғы ретті функцияларды қолданумен берілген алгоритмдер, көрнекті әрі жинақы, бағытталған нақты бөліктерден гөрі. чем реализации, ориентированные на конкретные частности.

Filter функциялар

Берілген предикат (булдік мәнді қайтаратын функциясы) және тізім бойынша берілген предикатты қанағаттандыратын элементтер тізімін қайтарады:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter xs
                | otherwise = filter xs
```

Мысалы, сандар тізімінен оның оң элементтерін алатын функция, былай анықталады :

```
getPositive = filter isPositive
isPositive x = x > 0
```

Foldr және foldl функциялар

(ағыл. **fold** – қайыру, ретпен қою (мысалы, парақты), **r-right** (оң), **l**-(сол))

Тізім элементтерінің қосындысын және көбейтіндісін (произведение) қайтаратын функцияларын қарастырайық :

```
sumList [] = 0
sumList (x:xs) = x + sumList xs

multList [] = 1
multList (x:xs) = x * multList xs
```

Мұнда жалпы элементтері: бастапқы мән (0- қосу, 1 - көбейту үшін) және мәндерді өзара келістіретін функция. **foldr** функциясы арқылы оларды біріктірейік Обобщим **:**

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

foldr функциясы бірінші аргумент ретінде аралас функцияны қабылдайды (ескереміз, ол түрлі типтің аргументтерін қабылдай алады,бірақ тип нәтижесі екінші аргумент типімен сай келуі керек). **foldr** функциясының екінші аргументі аралас функцияның бастапқы мәні болып табылады. Үшінші аргумент ретінде тізім беріледі. Функция берілген параметрлерге сай тізімге «жыймалау» жасайды.

foldr функциясы қалай жұмыс атқаратынын білу үшін **foldr** функциясының анықтамасын инфикстік нотация колдану арқылы жазамыз:

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

[a,b,c,...,z] элементтер тізімін **:** операторды қолданып ұсынайық. **foldr** функциясының қолданылу ережесі мынадай: : барлық операторлар : **f** функцияны (`**f**`) инфиксті түрде қолдануға ауыстырылады , ал **[]** бос тізім символы аралас функцияның бастапқы мәнге ауыстырылады. Түрлендіру қадамдарын былай бейнелеуге болады (бастапқы мән **init** –ке тең, дейік)

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

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

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

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

foldr функциясының көмегімен тізім элементтерін қосындылау және көбейту функциялары былай анықталады:

sumList = foldr (+) 0

multList = foldr (*) 1

[1,2,3] тізімі мысалы ретінде, осы функциялардың мәндері қалай есептелетінін карастырайық:

[1,2,3]

1 : 2 : 3 : []

1 : (2 : (3 : []))

1 + (2 + (3 + 0))

Сол сияқты көбейту үшін:

[1,2,3]

1 : 2 : 3 : []

1 : (2 : (3 : []))

1 * (2 * (3 * 1))

foldr функцияның қолданылуы жыймалауда пайдалану функциясының ассоциативтігін яғни фнкцияның қолданылуы оңға топталатынын көрсетеді.

Foldl функцияның пайдалануын қарастырайық **,** бұл жерде **l** әріпі операцияны қолдану солға топталатынын нұсқайды :

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

Түрлендіру қадамдары жоғарыдағыдай орындалады :

```
[a,b,c,...,z]
[]:a : b : c : ... : z
(([]: a) : b) : c) : ... : z
(((init `f` a) `f` b) `f` c) `f` ... `f` z
```

Қосу және көбейту сияқты ассоциативті операциялар үшін, **foldr** және **foldl** функциялары эквивалентті, бірақ егер операция ассоциативті болмаса, олардың нәтижесінде айырмашылық болады :

```
Main>foldr (-) 0 [1,2,3]
2
Main>foldl (-) 0 [1,2,3]
-6
```

Шынында, бірінші жағдайда **1 - (2 - (3 - 0)) = 2** шамасы есептеледі, ал екіншіде **((0 - 1) - 2) - 3 =** -6 шамасы есептеледі

Жоғары реттіктің басқа да функциялары

zip функцияс екі тізімді жұп тізіміне түрлендіреді:

zip :: [a] -> [b] -> [(a,b)]

zip (a:as) (b:bs) = (a,b):zip as bs

zip _ _ = []

Қолдану мысалы:

Prelude>zip [1,2,3] ['a','b','c']

[(1,'a'),(2,'b'),(3,'c')]

Prelude>zip [1,2,3] ['a','b','c','d']

[(1,'a'),(2,'b'),(3,'c')]

Ескеріңіз, нәтижелі тізім ұзындығы ең қысқа алғашқы тізім ұзындығына тең.

Бұл функция обобщениесі көрсетілген функция көмегімен екі тізімді «қосатын», жоғарғы ретті **zipWith** функциясы болып табылады :

```
zipWith        :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _   _    = []
```

Мына функция көмегімен, мысалы, екі тізімнің жеке элементтік қосынды функциясын оңай анықтауға болады :

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

немесе, каррирование есебімен :

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

Лямбда-абстракциялар

Жоғары дәрежелі функцияларды қолданғанда атаусыз функциялардың пайдалану қажеттілігін тұғызады болады. Haskell тілінде атаусыз функцияларды абстракция лямбда конструкциясының көмегімен анықтауға болады.

Мысалы, өз аргументін ің квадратын, екіге көбейтетін, бірді қосатын атаусыз функциялар келесі түрде жазылады:

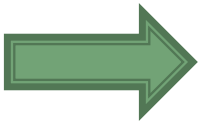
\x -> x * x

\x -> x + 1

\x -> 2 * x

Оларды ендігі жерде жоғары дәрежелі функциялардың аргументтері қатарында қолдануға болады.Мысалы, тізім элементтерінің квадратын есептеу функциясын былай жазуға болады:

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



getPositive функция

келесі түрде анықтауға болдады:

-- Тізімнің оң элементтерін анықтау функциясы

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

Лямбда-абстракцияларды келесі бірнеше айнымалылар үшін анықтауға болады :

\x y -> 2 * x + y

Лямбда-абстракцияларды кәдімгі функциялардың қатарында қолдануға болады, мысалы :

```
Main>(\x -> x + 1) 2
3
Main>(\x -> x * x) 5
25
Main>(\x -> 2 * x + y) 1 2
4
```

Лямбда-абстракциялар көмегімен функцияларды анықтауға болады. Мысалы, мына жазба

square = \x -> x * x

толық сәйкес келеді

quare x = x * x

Секциялар

Функцияларды бөлшектеп қолдануға болады яғни барлық аргументтердің мәндерін бермеуге. Мысалы,

егер **add** функциясы былайша анықталса

add x y = x + y

онда өз аргументін 1 ге арттыратын **inc** функциясын келесі түрде анықтауға болады:

inc = add 1

Тілде ендірілген және пайдаланушылар анықтаған бинарлы операторларды өз аргументерінің тек белгілі бір бөліктеріне қолдануға болады (себебі бинарлы операторлардың аргументтерінің саны екіге тең, ал бұл бөлім бір агрументтен тұрады). Бір аргументке қолданылған бинарлы операция ***секция*** деп аталады.

Мысалы :

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

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

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

Жақшалар мұнда міндетті түрде болу тиіс. Сонымен, add және inc функцияларын былайша анықтауға болады:

add = (+)

inc = (+1)

Секциялар әсіресе оларды жоғары дәрежелі функциялардың аргументтері ретінде қолданған кезде пайдалы. Тізімнің оң элементтерін анықтау функциясын еске түсірейік :

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

Секцияларды қолданып қысқаша жазуға болады:

getPositive = filter (>0)

Тізім элементтерін екі еселеу функциясы:

doubleList = map (*2)