Programlama Dilleri Laboratuvar Föyü
Haskell Programlama Dili 2
Fonksiyonlar ve Özyineleme

Şevket Umut ÇAKIR

1 Giriş

Bir saf fonksiyonel programlama dili olan Haskell'in en güçlü özelliği olan fonksiyonların tanımlanmasında kullanılan örüntü eşleme, muhafızlar(guards), where, let ve case gibi ifadelerin kullanımı anlatılacaktır. Fonksiyonel programlamada sıklıkla kullanılan öz yineleme üzerinde durulacaktır.

2 Konu Anlatımı ve Deney Hazırlığı

Haskell dilinde fonksiyon tanımlaması yapılırken bir çok farklı yol izlenebilir. Bu yollardan her biri farklı amaçlara hizmet etmektedir.

2.1 Örüntü Eşleme(Pattern Matching)

Fonksiyon tanımlarken farklı örüntüler için farklı fonksiyon gövdeleri tanımlanabilir. Bu yol basit ve okunabilir kod yazmayı sağlar. Örüntüler herhangi bir veri türünden olabilir(örn: sayılar, karakterler, listeler, çokuzlular vd.). Aşağıda örüntü eşleme ilgili örnek üç tane fonksiyon tanımı verilmiştir.

```
sansli 7 = "ŞANSLI SAYI 7!"
sansli x = "Üzgünüm, şansınız yok!"

soyle 1 = "Bir"
soyle 2 = "İki"
soyle 3 = "Üç"
soyle 4 = "Dört"
soyle x = "Bir ile Dört arasında değil"

faktoryel 0 = 1
faktoryel n = n * faktoryel (n - 1)
```

Haskell ilgili örüntüyü gördüğünde o satırdaki fonksiyonu çalıştıracaktır. Tanımlanan bu üç fonksiyonun kullanımları aşağıda verilmiştir.

```
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :l pattern1.hs
[1 of 1] Compiling Main             ( pattern1.hs, interpreted )
Ok, one module loaded.
*Main> putStrLn (sansli 7)
ŞANSLI SAYI 7!
*Main> putStrLn (sansli 4)
Üzgünüm, şansınız yok!
*Main> putStrLn (soyle 3)
Üç
*Main> faktoryel 20
2432902008176640000
```

Örüntü eşleme yöntemi listeler ve çokuzlular üzerinde de kullanılabilmektedir. Listelerde örüntü eşleme kullanılırken baş ve kuyruğu birbirinden ayırmak için **:** sembolleri kullanılır. Çokuzlular için de parantez içinde **,** sembolü kullanılarak elemanlar birbirinden ayırt edilebilir. Listelerde örüntü eşleme için ++ sembolü kullanılamaz. Aşağıda kullanım örnekleri mevcuttur.

```
vektorTopla a b = (fst a + fst b, snd a + snd b)

vektorTopla2 (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

soyle [] = "Liste boş"
soyle (x:[]) = "Listenin bir elemanı var: " ++ show x
soyle (x:y:[]) = "Listenin iki elemanı var: " ++ show x ++ " ve "
↪  ++ show y
soyle (x:y:_) = "Liste uzun. İlk iki eleman: " ++ show x ++ " ve
↪  " ++ show y

toplam [] = 0
toplam (x:xs) = x + toplam xs

ilkharf tumu@(x:xs) = tumu ++ " kelimesinin ilk harfi " ++ [x] ++
↪  " dir"
```

2.2 Muhafızlar(Guards)

Örüntü eşleme bir değerin bir biçime uygun olduğunu doğrular ve değerin parçalara ayrılarak kullanılmasını sağlar. Muhafızlar ise değerlerin bazı özelliklerinin doğru veya yanlışlığını test etme yoludur. Bu tanımla birlikte muhafız yapıları **if** yapısına benzer özellik gösterir fakat daha okunur ve anlaşılabilir bir yapıya sahiptir. Aşağıda vücut kitle indeksine göre geriye bir metin döndüren `vkiSoyle` fonksiyonu verilmiştir. Muhafız kullanımında girintiler önemlidir.

```
vkiSoyle vki
  | vki <= 18.5 = "Çok zayıfsınız"
```

```
  | vki <= 25 = "Normal kilodasınız"
  | vki <= 30 = "Kilolusunuz, biraz dikkat!"
  | otherwise = "Tebrikler!"
```

Vücut kitle indeksini boy ve kilodan hesaplayan vkiSoyle2 fonksiyonunun yapısı aşağıda verilmiştir. Bu yapının kullanımında formül ve değerler iç içe girmiş durumdadır. Bunu önüne geçmek adına `where` yapısı kullanılabilir. Aşağıdaki `vkiSoyle3` fonksiyonunun içinde `where` yapısı kullanımı bulunmaktadır. Muhafız kullanımında olduğu gibi `where` ifadesinde de girinti kullanımı önemlidir.

```
vkiSoyle2 kilo boy
  | kilo / boy ^ 2 <= 18.5 = "Çok zayıfsınız"
  | kilo / boy ^ 2 <= 25 = "Normal kilodasınız"
  | kilo / boy ^ 2 <= 30 = "Kilolusunuz, biraz dikkat!"
  | otherwise = "Tebrikler!"

vkiSoyle3 kilo boy
  | vki <= zayif = "Çok zayıfsınız"
  | vki <= normal = "Normal kilodasınız"
  | vki <= sisman = "Kilolusunuz, biraz dikkat!"
  | otherwise = "Tebrikler!"
  where vki = kilo / boy ^ 2
        zayif = 18.5
        normal = 25.0
        sisman = 30.0
--      (zayif, kilolu, sisman) = (18.5, 25.0, 30.0) --şeklinde
↪  de olabilirdi
--kullanımı: vkiSoyle3 80 1.75
vkiHesapla l = [vki k b | (k, b) <- l]
  where vki k b = k / b ^ 2
```

2.3 Let İfadesi

`where` ifadesine benzer özellikte olan, yerel bir kapsamda ifade ya da değerleri isimlere bağlamaya yarayan bir ifadedir. `let` ifadeleri içerisinde de örüntü eşleme kullanılabilir. Genel kullanımı `let <baglamalar> in <ifade>` şeklindedir. `in` içindeki ifade `let` ifadesinin tümünün alacağı değerdir. Aşağıda kullanım örneği verilmiştir.

```
silindir r h =
  let yanAlan = 2 * pi * r *h
      ustAlan = pi * r *r
  in yanAlan + 2 * ustAlan

vkiHesapla l = [vki | (k, b) <- l, let vki = k / b ^ 2]
```

`let` fonksiyon tanımlarında kullanıldığı gibi interaktif yorumlayıcı içinde de iki farklı şekilde kullanılabilir. Aşağıda kullanım örnekleri mevcuttur[2].

```
Prelude> 4 * (let a = 9 in a + 1) + 2
42
Prelude> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
Prelude> let zoot x y z = x * y + z
Prelude> zoot 3 9 2
29
```

2.4 Case İfadesi

Diğer dillerde olan `case` ifadelerine benzerdir. Bir değişkenin değeri bir kaç seçenek kullanılarak test edilir ve tüm ifade uygun seçeneğin değerini alır. Aşağıda örnek kullanımı mevcuttur.

```
listeTanimla x = "Liste " ++ case x of [] -> "boş."
                                       [e] -> "tek elemanlı."
                                       x -> "birden çok
                                        ↪  elemanlı."
```

2.5 Özyineleme

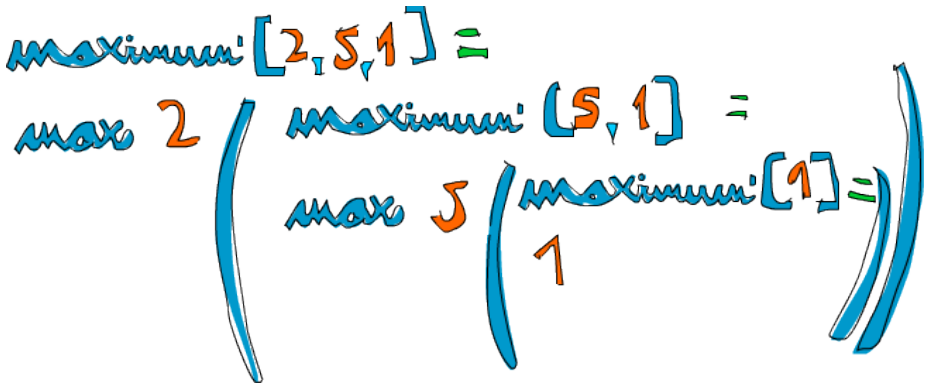
Özyineleme gövdesinde bir fonksiyonun kendisini çağırmasıdır. Matematikte tanımlamalar genellikle özyineli olarak verilir. Örneğin meşhur Fibonacci serisine bakıldığında, ilk iki sayı haricinde özyineli olarak tanımlanmıştır.

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned} \tag{1}$$

Fonksiyon tanımında bir veya daha fazla özyineli olmayan tanımlara **durma noktası**(edge condition) adı verilir. Durma noktası özyinelemeyi bitirmek için kullanılan koşullardır. Eğer özyineleme bir noktada sonlandırılmasaydı, F(-2000) gibi bir değere gelindiğinde bile fonksiyon bir sonuç üretmezdi. Özyinelemeli bir fonksiyon tasarlarken durma noktasına dikkat etmek gerekir. Aşağıda listedeki en büyük elemanı bulan `maximum'` fonksiyonunun tanımlanması verilmiştir. Listedeki elemanları baş ve kuyruk olarak ayırıp, kuyruğu özyinelemeli olarak fonksiyona gönderir. Dönen sonuçla baş elemanı birleştirerek geri döndürür. `maximum''` fonksiyonu da `max` fonksiyonunu kullanan kısa bir halidir. `maximum'` fonksiyonunun örnek çalışması Şekil 1'de verilmiştir.

```
maximum' [] = error "boş listenin en büyüğü"
maximum' [x] = x
maximum' (x:xs)
    | x > maxKuyruk = x
    | otherwise = maxKuyruk
    where maxTail = maximum' xs
```

```haskell
maximum'' [] = error "boş listenin en büyüğü"
maximum'' [x] = x
maximum'' (x:xs) = max x (maximum' xs)
```



Şekil 1: `maximum'` fonksiyonunu çalışması [1]

Aşağıda özyinelemeli olarak bazı fonksiyon tanımları verilmiştir.

```haskell
replicate' n x
    | n <= 0    = []
    | otherwise = x:replicate' (n-1) x

take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs

reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]

repeat' x = x:repeat' x

zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys

elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

3 Deneyin Uygulanması

Bu deneyde aşağıdaki fonksiyonların yazılması amaçlanmaktadır.

3.1 uzunluk Fonksiyonu

`uzunluk` fonksiyonu bir listedeki eleman sayısını vermelidir. Başka yollarla yapılabilmesine rağmen bu soruda çözümün özyinelemeli olması beklenmektedir.Kullanımı aşağıda verilmiştir.

```
Prelude> uzunluk [1, 10, 21, 4]
4
```

3.2 ciftFaktoryel Fonksiyonu

Faktoryel fonksiyonu 1 ile n arasındaki sayıların çarpımıdır. Çift faktoryelde ise eğer parametre olarak verilen sayı tek ise 1 ile n arasımdaki tek sayıların çarpımını, çift ise 2 ile n arasındaki çift sayıların çarpımını vermektedir. Örneğin 6 sayısı için $2 \cdot 4 \cdot 6 = 48$, 7 sayısı için $1 \cdot 3 \cdot 5 \cdot 7 = 105$ olacaktır. Bu işlevi gerçekleştiren `ciftFaktoryel` fonksiyonunu yazın.

3.3 palindrom Fonksiyonu

Palindrom metinler tersten ve düzden okunuşları aynı olan metinlerdir. Palindromluk sadece metinlerle sınırlı değildir, listeler için de düşünülebilir. Örneğin `[1,4,5,4,1]` listesi palindromdur. Parametre olarak verilen listenin palindrom olup olmadığını döndüren özyinelemeli `palindrom` fonksiyonunu yazınız.

3.4 indistekiEleman Fonksiyonu

Çokuzlu olarak verilen liste ve indis değerine göre belirtilen indisteki elemanı döndüren `indistekiEleman` fonksiyonunu yazınız. Örnek kullanımı aşağıda verilmiştir.

```
Prelude> indistekiEleman ([1,7,5,4,6], 2)
5
```

3.5 compress Fonksiyonu

`compress` fonksiyonu liste içindeki tekrar eden elemaları bir adete düşürecek bir yapıya sahiptir. Bu fonksiyonu yazınız. Örnek kullanımı aşağıda verilmiştir.

```
Prelude> compress "aaabbacccccdeed"
"abacded"
Prelude> compress [1,1,1,2,2,3,1,3,3]
[1,2,3,1,3]
```

Kaynaklar

[1] *Learn You a Haskell for Great Good! A Beginner's Guide - Recursion.* URL: `http://learnyouahaskell.com/recursion` (son erişim: 16.4.2019).

[2] *Learn You a Haskell for Great Good! A Beginner's Guide - Syntax in Functions.* URL: `http://learnyouahaskell.com/syntax-in-functions#let-it-be` (son erişim: 16.4.2019).