Fonksiyonel Programlama Jargonu

Arity

Bir fonksiyonun aldığı argüman sayısıdır. Bir fonksiyon aldığı argüman sayısına göre *unary* (1 argüman), *binary* (2 argüman), *ternary* (3 argüman)... olarak adlandırılır. Eğer bir fonksiyon değişken sayıda argüman alıyorsa *variadic* olarak adlandırılır.

```
Prelude> let sum a b = a + b
Prelude> :t sum
sum :: Num a => a -> a -> a

-- sum fonksiyonunun arity'si 2dir.
```

Higher-Order Functions (HOF)

Argüman olarak bir fonksiyon alan ya da bir fonksiyonu çıktı veren fonksiyonlardır.

```
Prelude> let add3 a = a + 3
Prelude> map add3 [1..4]
[4,5,6,7]
```

```
Prelude> filter (<4) [1..10]
[1,2,3]
```

Closure

Kapanış, bir fonksiyona bağlı değişkenleri koruyan bir kapsamdır. Kısmi uygulama için önemlidir.

```
Prelude> let f x = (\y -> x + y)
```

`f` fonksiyonunu bir sayı ile çağıralım.

```
Prelude> let g = f 5
```

Bu durumda `x = 5` değeri `g` fonksiyonunun kapanışında korunur. Şimdi `g` fonksiyonunu bir `y` değeri ile çağırırsak:

```
Prelude> g 3
8
```

Partial Application

Kısmi uygulama, bir fonksiyonun bazı argümanlarını önceden doldurarak yeni bir fonksiyon oluşturmaktır.

```
-- Orjinal fonksiyonumuz
Prelude> let add3 a b c = a + b + c

--`2` ve `3` argümanlarını `add3` fonksiyonumuza vererek `fivePlus` fonksiyonumuzu
 oluşturuyoruz
Prelude> let fivePlus = add3 2 3

Prelude> fivePlus 4
9
```

Kısmi uygulama, kompleks fonksiyonlardan daha basit fonksiyonlar oluşturmaya yardım eder. Curried fonksiyonlar otomatik olarak kısmi uygulanmış fonksiyonlardır.

Currying

Birden çok parametre alan bir fonksiyonu, her defasında sadece bir parametre alan bir fonksiyona dönüştürmektir.

Fonksiyon her çağrıldığında sadece bir argüman kabul eder ve tüm argümanlar verilene kadar sadece bir argüman alan bir fonksiyon döndürür.

```
Prelude> let sum (a, b) = a + b
Prelude> let curriedSum = curry sum
Prelude> curriedSum 40 2
42
Prelude> let add2 = curriedSum 2
Prelude> add2 10
12
```

Function Composition

İki farklı fonksiyonu bir araya getirerek, bir fonksiyonun çıktısı diğer fonksiyonun girdisi olan üçüncü bir fonksiyon oluşturmaktır.

```
-- fonksiyonları bir araya getirmek için '.' operatörü kullanılır
Prelude> let floorAndToString = show . floor
Prelude> floorAndToString 32.123
"32"
```

Purity

Bir fonksiyonun çıktısı sadece girdi veya girdilerine bağlı ve fonksiyon yan etki oluşturmuyor ise, fonksiyon *saftır* denir.

```
Prelude> let greet name = "Hello " ++ name
Prelude> greet "Brianne"
"Hello Brianne"
```

Saf olmayan fonksiyona bir örnek:

```
Prelude> let name1 = "Brianne"
Prelude> let greet = "Hello " ++ name1
Prelude> greet
"Hello Brianne"
```

Yukarıdaki fonksiyonun çıktısı fonksiyonun dışarısında tanımlı bir değişkene bağlıdır.

Side effects

Bir fonksiyon veya ifade, dışarısındaki bir durum ile etkileşime geçiyor ise (okuma veya yazma), *yan etki* ye sahiptir denir.

Haskell'deki tüm fonksiyonlar saftır.

Idempotent

Bir fonksiyon, sonucuna tekrar uygulandığında sonuç değişmiyorsa *idempotent* olarak adlandırılır.

```
f(f(x)) ≍ f(x)
```

```
Prelude> abs (abs (-1))
1
```

```
Prelude Data.List> sort (sort [1,4,3,1,5])
[1,1,3,4,5]
```

Point-Free Style

Argümanların açıkca tanımlanmadığı fonksiyonlar yazmaktır. *Tacit programming* olarak da bilinir.

```
Prelude> let add a b = a + b

-- incrementAll fonksiyonunu tanımlayalım

-- Point-free değildir - `numbers` argümanı belirtilmiştir
Prelude> let incrementAll numbers = map (+1) numbers

-- Point-free - Fonksiyonun aldığı argüman açıkca belirtilmemiştir
Prelude> let incrementAll = map (+1)
```

`incrementAll` fonksiyonunun `numbers` argümanını aldığı belirtilmiştir, bu nedenle point-free değildir. `incrementAll2` fonksiyonu ise, fonksiyon ve değerlerin bir bileşimidir ve argüman bilgisi belirtilmemiştir. Yani *point-free* dir.

Predicate

Verilen bir değer için doğru veya yanlış değerini dönen fonksiyonlardır. Genellikle *filter* ile beraber kullanılırlar.

```
Prelude> let predicate a = a < 3
Prelude> filter predicate [1..10]
[1,2]
```

Referential Transparency

Bir ifade değeri ile yer değiştirildiğinde programın davranışı değişmiyor ise, ifade *referentially transparent* olarak adlandırılır.

Lambda

Anonim (isimsiz) fonksiyonlardır.

`\x -> x + 1`

Çoğunlukla yüksek mertebeden fonksiyonlar ile birlikte kullanılırlar.

```
Prelude> map (\x -> x + 1) [1..4]
[2,3,4,5]
```

Lazy evaluation

Lazy evaluation, bir ifadenin, ifade sonucuna ihtiyaç duyulana kadar hesaplanmamasıdır. Böylece, sonsuz listeler gibi yapılar tanımlanabilir.

```
Prelude> let lst0 = [1..]
Prelude> take 5 lst0
[1,2,3,4,5]
```

Category

Bir C kategorisi üç şeyden oluşur:

- Nesnelerin bir kümesi,
- Her bir nesne çifti için, morfizmaların bir kümesi,
- Birbiriyle uyumlu morfizma çiftleri arasında tanımlı bir ikili işlem.

ve aşağıdaki iki beliti sağlar:

- A, B, C, D nesneler ve f, g, h morfizmalar olmak üzere, $f : A \to B$, $g : B \to C$ ve $h : C \to D$ ise $h \circ (g \circ f) = (h \circ g) \circ f$ dir,
- Her x nesnesi ve her $f : a \to x$ ve $g : x \to b$ için, $1_x \circ f = f$ ve $g \circ 1_x = g$ koşullarını sağlayan bir $1_x \circ x = x$ morfizması vardır.

Aşağıdaki tabloda bir kaç kategori örneği verilmiştir.

Kategori	Nesneler	Morfizmalar
Set	Kümeler	Fonksiyonlar
Grp	Gruplar	Grup homomorfizmaları
Top	Topolojik uzaylar	Sürekli fonksiyonlar
Uni	Düzgün uzaylar	Düzgün sürekli fonksiyonlar

Haskell'de kategoriler

Hask, Haskell tiplerinin ve fonksiyonlarının bir kategorisidir.

Hask kategorisinin nesneleri Haskell'deki *tipler*, `A` nesnesinden `B` nesnesine tanımlı morfizmalar ise `A -> B` şeklindeki fonksiyonlardır.

Daha Fazla Kaynak

- [Category Theory for Programmers]

Morphism

Morfizma, bir kategorideki iki nesne arasındaki eşlemedir.

1. *Homomorfizma*, aynı tipteki iki cebirsel yapı arasındaki bir eşlemedir. Morfizmanın daha genel halidir.
2. Bir kategorideki $f : Y \to X$ morfizması ve bu kategorideki her $u, v : Z \to Y$ morfizmaları için $fu = fv \Rightarrow u = v$ ise `f` morfizması *monomorfizma* olarak adlandırılır.
3. Bir kategorideki $f : Y \to X$ morfizması ve bu kategorideki her $u, v : X \to Z$ morfizmaları için $uf = vf \Rightarrow u = v$ ise `f` morfizması *epimorfizma* olarak adlandırılır.
4. Birebir ve örten morfizmalar, *isomorfizma* olarak adlandırılır.
5. Bir nesneden kendisine ve örten morfizmalar, *endomorfizma* olarak adlandırılır.
6. Bir nesneden kendisine isomorfizmalar, *otomorfizma* olarak adlandırılır.

Functor

C ve D iki kategori olsun. Bir $F : C \to D$ funktoru

- C kategorisindeki her X nesnesi için $F(id_X) = id_{F(X)}$,
- C kategorisindeki her $f : X \to Y$ ve $g : Y \to Z$ morfizmaları için $F(g \circ f) = F(g) \circ F(f)$

koşullarını sağlayan bir eşlemedir.

Haskell'de funktorlar

Funktor, üzerine `map` fonksiyonu uygulanabilen bir tiptir (listeler üzerine uygulanan map fonksiyonunu genelleştirir) ve tek bir metoda sahiptir:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Applicative Functor

Applicative functor, funktor ve monad arasında konumlanan ve

```
pure id <*> v = v                            -- Identity
pure f <*> pure x = pure (f x)               -- Homomorphism
u <*> pure y = pure ($ y) <*> u              -- Interchange
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

koşullarını sağlayan bir yapıdır.

Haskell'de applicative funktorlar aşağıdaki gibi tanımlanır:

```
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Monad

Bir $M : C \to C$ funktoru verilsin. C kategorisindeki her X nesnesi için - $unit_X^M : X \to M(X)$ - $join_X^M : M(M(X)) \to M(X)$ morfizmalarını tanımlayalım. `M` funktoru, `unit` ve `join` morfizmaları ile birlikte bir *monad* olarak adlandırılır.

Haskell'de monadlar

Monadlar, bir araya getirilebilen hesaplama adımları olarak düşünülebilirler. Ayrıca monadlar, saf hesaplamalara G/Ç, ortak ortam, güncellenebilir durumlar vb. özellikler ekler.

Algebraic data type

Cebirsel veri tipleri, bileşik tiplerdir - diğer tiplerin bir araya getirilmesiyle oluşurlar.

En yaygın cebirsel veri tipleri toplamsal tipler (sum types) ve çarpımsal tipler (product types) dir.

Sum type

Toplamsal tipler, basit olarak `veya` bağlacı ile oluşturulan tipler denilebilir. En basit toplamsal tip olan `Bool` tipinin tanımına bakalım:

```
data Bool = False | True
```

Bu tanım şunu söylemektedir: Bir `Bool`, `False` veya `True` değerlerinden herhangi birini alabilir.

Product type

Çarpımsal tipler, `ve` bağlacı ile oluşturulan tiplerdir. Bir örnek verelim:

```
data Color = Color Int Int Int
```

`Color` tipi üç `int` değerinden oluşmaktadır.

References

- Haskell - Wiki
- Haskell - Wikibooks
- hemanth/functional-programming-jargon.
- Wolfram MathWorld