Fonksiyonel Programlama'da List Yapısı Yardımıyla Cebirsel ve Kategoriksel Yapılar

Prof. Dr. Zekeriya ARVASİ

Matematik ve Bilgisayar Bilimleri Bölümü

Eskişehir Osmangazi Üniversitesi

20 Mayıs 2013

List Yapısı Yardımıyla Cebirsel ve Kategoriksel Yapılar

List Yapısı

List Yapısı

Fonksiyonel Programlama Dili'nde herhangi bir tip alındığında, bu tipin elemanlarının bir sonlu listesi yazılabilir.

Örneğin, *Int* tipi için,

$$[1, 2, 2, 1, 4]$$

bir sonlu listedir.

List Yapısı

Fonksiyonel Programlama Dili'nde herhangi bir tip alındığında, bu tipin elemanlarının bir sonlu listesi yazılabilir.

Örneğin, *Int* tipi için,

$$[1, 2, 2, 1, 4]$$

bir sonlu listedir.

List(S) , S tipinin elemanları yardımıyla elde edilebilen tüm mümkün listelerin kümesi olsun.

Örneğin, *Int* tipi için,

$$List\ (Int) = \{\ [\ ], [1], [1, 2], [1, 2, 2], [3, 3], \ldots\}$$

şeklinde sonsuz elemanlı bir kümedir.

List Yapısı

Hatta tek elemanlı bir $S = \{s\}$ tipi alınsa dahi,

$$List\ (S) = \{\ [\ ], [s], [s, s], [s, s, s], [s, s, s, s], \ldots\}$$

şeklinde yine sonsuz elemanlı bir kümedir.

List Yapısı

Hatta tek elemanlı bir $S = \{s\}$ tipi alınsa dahi,

$$List\ (S) = \{\ [\ ], [s], [s, s], [s, s, s], [s, s, s, s], \ldots\}$$

şeklinde yine sonsuz elemanlı bir kümedir.

Böylece *List* ile tipler üzerinde etkisi her tipin aslında bir küme olması itibariyle dolaylı olarak; $C(\mathcal{L})$; $\mathcal{L}$ dilinin kategorisi olmak üzere,

$$List : C(\mathcal{L}) \longrightarrow C(\mathcal{L})$$

şeklinde tanımlıdır.

List Yapısı

Fakat *List* in genel olarak fonksiyonel programlama mantığı altında, sadece tipler üzerinde değil, aynı zamanda aşağıda açıkça inceleneceği gibi,

$$f : S \longrightarrow S'$$

biçimdeki, iki tip arasındaki fonksiyonların üzerinde,

List Yapısı

Fakat *List* in genel olarak fonksiyonel programlama mantığı altında, sadece tipler üzerinde değil, aynı zamanda aşağıda açıkça inceleneceği gibi,

$$f : S \longrightarrow S'$$

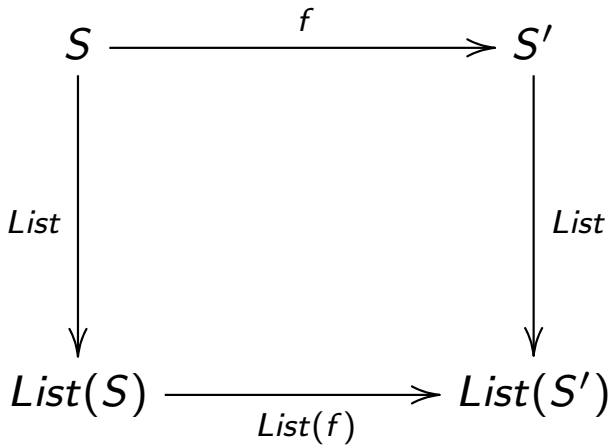
biçimdeki, iki tip arasındaki fonksiyonların üzerinde,

$$List(f) : List\ (S) \longrightarrow List\ (S')$$

şeklinde bir indirgenmiş fonksiyondur.

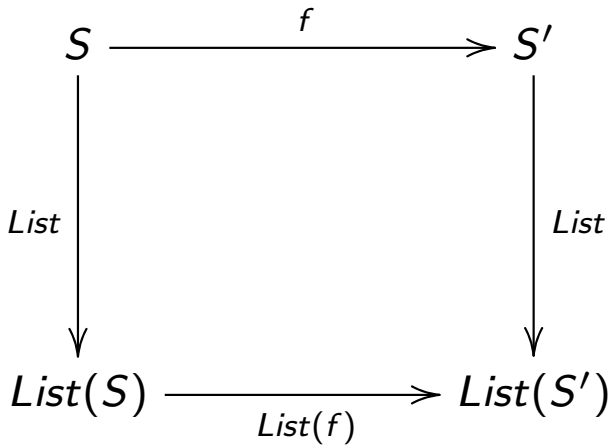
List Yapısı

Yani diyagramatik olarak,



şeklindedir.

Yani diyagramatik olarak,



şeklindedir.

Örneğin,

$$\begin{array}{lccl} List\ (f): & List\ (S) & \longrightarrow & List\ (S') \\ & [a,b] & \longmapsto & List\ (f)\ [a,b] = [f(a), f(b)] \end{array}$$

şeklindedir.

List Yapısı

List dönüşümünün fonksiyonlara etkisini, herhangi bir fonksiyonel programlama dilinde genel bir örnekle görelim. Herhangi bir fonksiyonel programlama dilinde *f* fonksiyonu,

Fonksiyonel Programlama Dilleri

```
f :: Int -> Int
f x = x*x
```

şeklinde tanımlanacak olursa,

List Yapısı

List dönüşümünün fonksiyonlara etkisini, herhangi bir fonksiyonel programlama dilinde genel bir örnekle görelim. Herhangi bir fonksiyonel programlama dilinde *f* fonksiyonu,

Fonksiyonel Programlama Dilleri

```
f :: Int -> Int
f x = x*x
```

şeklinde tanımlanacak olursa,

$$List(f) : List(Int) \longrightarrow List(Int)$$

dönüşümü için,

List Yapısı

List dönüşümünün fonksiyonlara etkisini, herhangi bir fonksiyonel programlama dilinde genel bir örnekle görelim. Herhangi bir fonksiyonel programlama dilinde *f* fonksiyonu,

Fonksiyonel Programlama Dilleri

```
f :: Int -> Int
f x = x*x
```

şeklinde tanımlanacak olursa,

$$List(f) : List(Int) \longrightarrow List(Int)$$

dönüşümü için,

Fonksiyonel Programlama Dilleri

```
> List(f)([1,2,3])
[1,4,9]
```

sonucu elde edilir.

List Yapısı

Şimdi yukarıdaki örneği özel olarak Haskell diline taşıyalım.

List Yapısı

Şimdi yukarıdaki örneği özel olarak Haskell diline taşıyalım.

List dönüşümünün f fonksiyonuna etkisi burada *fmap* fonksiyonu yardımıyla;

$$List(f) = fmap\ f$$

şeklinde tanımlanmaktadır.

List Yapısı

Şimdi yukarıdaki örneği özel olarak Haskell diline taşıyalım.

List dönüşümünün *f* fonksiyonuna etkisi burada *fmap* fonksiyonu yardımıyla;

$$List(f) = fmap\ f$$

şeklinde tanımlanmaktadır.

Bu durumda yukarıdaki fonksiyon Haskell üzerinde,

Haskell

```
f :: Int -> Int
f x = x*x
```

biçiminde tanımlanacak,

List Yapısı

ve ayrıca,

Haskell

```
fmap f :: [Int] -> [Int]
```

dönüşümü ile birlikte,

ve ayrıca,

Haskell
```
fmap f :: [Int] -> [Int]
```

dönüşümü ile birlikte,

Haskell
```
> fmap f [1,2,3]
[1,4,9]
```

sonucu elde edilecektir.

List Yapısının Özellikleri ve Funktor

List Yapısının Özellikleri ve Funktor

Hatırlatma: Kategori Teori'deki Funktor tanımı hatırlanacak olursa, $\mathcal{C}$ ve $\mathcal{D}$ herhangi iki kategori olmak üzere,

$$F : \text{Mor}(\mathcal{C}) \rightarrow \text{Mor}(\mathcal{D})$$

fonksiyonu;

Hatırlatma: Kategori Teori'deki Funktor tanımı hatırlanacak olursa, $\mathcal{C}$ ve $\mathcal{D}$ herhangi iki kategori olmak üzere,

$$F : \mathrm{Mor}(\mathcal{C}) \to \mathrm{Mor}(\mathcal{D})$$

fonksiyonu;

F1) (Birimlerin korunması) $\mathcal{C}$ nin her A objesi için,

$$F(1_A) = 1_{F(A)}$$

F2) (Kompozisyonların korunması) $g \circ f$, $\mathcal{C}$ nin bir kompozisyonu ise,

$$F(g \circ f) = F(g) \circ F(f) \quad ,$$

List Yapısının Özellikleri ve Funktor

Hatırlatma: Kategori Teori'deki Funktor tanımı hatırlanacak olursa, $\mathcal{C}$ ve $\mathcal{D}$ herhangi iki kategori olmak üzere,

$$F : \mathrm{Mor}(\mathcal{C}) \to \mathrm{Mor}(\mathcal{D})$$

fonksiyonu;

F1) (Birimlerin korunması) $\mathcal{C}$ nin her A objesi için,

$$F(1_A) = 1_{F(A)}$$

F2) (Kompozisyonların korunması) $g \circ f$, $\mathcal{C}$ nin bir kompozisyonu ise,

$$F(g \circ f) = F(g) \circ F(f) \quad ,$$

aksiyomlarını sağlıyor ise F ye, $\mathcal{C}$ den $\mathcal{D}$ ye bir funktor denir.

List Yapısının Özellikleri ve Funktor

Haskell dilinde bir listeye eleman eklenmesi : işlemi ile sağlanır. Yani, bir x elemanının herhangi bir xs listesine eklenmesi x : xs şeklinde ifade edilir.

List Yapısının Özellikleri ve Funktor

Haskell dilinde bir listeye eleman eklenmesi : işlemi ile sağlanır. Yani, bir x elemanının herhangi bir xs listesine eklenmesi x : xs şeklinde ifade edilir.

Örneğin;

Haskell

```
[2,1,3]
> [2,1,3]

2 : [1,3]
> [2,1,3]

2 : 1 : [3]
> [2,1,3]

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

şeklindedir.

List Yapısının Özellikleri ve Funktor

fmap fonksiyonunun tanımına ve özelliklerine bakacak olursak, bu fonksiyon Haskell üzerinde,

Haskell

```
fmap :: (a -> b) -> f a -> f b
```

şeklinde tanımlanmaktadır.

List Yapısının Özellikleri ve Funktor

fmap fonksiyonunun tanımına ve özelliklerine bakacak olursak, bu fonksiyon Haskell üzerinde,

Haskell

```
fmap :: (a -> b) -> f a -> f b
```

şeklinde tanımlanmaktadır.

Yukarıdaki tanımı basit şekilde örneklendirecek olursak,

List Yapısının Özellikleri ve Funktor

fmap fonksiyonunun tanımına ve özelliklerine bakacak olursak, bu fonksiyon Haskell üzerinde,

Haskell
```
fmap :: (a -> b) -> f a -> f b
```

şeklinde tanımlanmaktadır.

Yukarıdaki tanımı basit şekilde örneklendirecek olursak,

Haskell
```
fmap f [ ]
>[ ]

fmap f [4]
>[16]

fmap f [2,3,4]
>[4,9,16]
```

sonucu elde edilecektir.

List Yapısının Özellikleri ve Funktor

Bu fonksiyon aslında listeler için,

Haskell

```
map _ [ ] = [ ]
map f (x:xs) = f x : map f xs
```

şeklinde de tanımlanabilmektedir.

Bu fonksiyon aslında listeler için,

Haskell

```
map _ [ ] = [ ]
map f (x:xs) = f x : map f xs
```

şeklinde de tanımlanabilmektedir.

Yani yine örneklendirecek olursak,

Bu fonksiyon aslında listeler için,

Haskell
```
map _ [ ] = [ ]
map f (x:xs) = f x : map f xs
```

şeklinde de tanımlanabilmektedir.

Yani yine örneklendirecek olursak,

Haskell
```
map f (2 : [3,4])
>[4,9,16]

f 2 : map f [3,4]
>[4,9,16]

f 2 : f 3 : map f [4]
>[4,9,16]
```

sonucu elde edilecektir.

List Yapısının Özellikleri ve Funktor

Ayrıca yine alternatif olarak,

List Yapısının Özellikleri ve Funktor

Ayrıca yine alternatif olarak,

Haskell

```
map _ [ ] = [ ]
map f [a] = [f a]
map f (xs ++ ys) = map f xs ++ map f ys
```

biçiminde de tanımlanabilir.

Ayrıca yine alternatif olarak,

Haskell

```
map _ [ ] = [ ]
map f [a] = [f a]
map f (xs ++ ys) = map f xs ++ map f ys
```

biçiminde de tanımlanabilir. Yine bu tanımı da basit şekilde örneklendirirsek,

Ayrıca yine alternatif olarak,

Haskell

```
map _ [ ] = [ ]
map f [a] = [f a]
map f (xs ++ ys) = map f xs ++ map f ys
```

biçiminde de tanımlanabilir. Yine bu tanımı da basit şekilde örneklendirirsek,

Haskell

```
map f [ ]
> [ ]

map f [5]
> [25]

map f ( [1,2] ++ [3,4] )
> [1,4,9,16]

map f [1,2] ++ map f [3,4]
> [1,4,9,16]
```

sonucu elde edilecektir.

List Yapısının Özellikleri ve Funktor

Yukarıda tanımladığımız *map* fonksiyonunun aslında listeler için bir fmap yapısında olduğunu söylemiştik. Benzer şekilde farklı tipler için de tanimlanabilecek,

List Yapısının Özellikleri ve Funktor

Yukarıda tanımladığımız *map* fonksiyonunun aslında listeler için bir fmap yapısında olduğunu söylemiştik. Benzer şekilde farklı tipler için de tanimlanabilecek,

Haskell

```
map :: (a -> b) -> [a] -> [b]
treemap :: (a -> b) -> Tree a -> Tree b
mmap :: (a -> b) -> Maybe a -> Maybe b
iomap :: (a -> b) -> IO a -> IO b
funmap :: (a -> b) -> (c -> a) -> (c -> b)
```

tüm bu fonksiyonlar aslında *fmap* fonksiyonunun ozel halleridir.

List Yapısının Özellikleri ve Funktor

Yukarıda tanımladığımız *map* fonksiyonunun aslında listeler için bir fmap yapısında olduğunu söylemiştik. Benzer şekilde farklı tipler için de tanimlanabilecek,

Haskell

```
map :: (a -> b) -> [a] -> [b]
treemap :: (a -> b) -> Tree a -> Tree b
mmap :: (a -> b) -> Maybe a -> Maybe b
iomap :: (a -> b) -> IO a -> IO b
funmap :: (a -> b) -> (c -> a) -> (c -> b)
```

tüm bu fonksiyonlar aslında *fmap* fonksiyonunun ozel halleridir.

İşte bu noktada en önemli soru, üzerinde tanımlanan tiplerden bağımsız olarak hareket edebilecek, en genel bir *map* fonksiyonunun tanımlanıp tanımlanamayacağıdır.

List Yapısının Özellikleri ve Funktor

Yukarıda tanımladığımız *map* fonksiyonunun aslında listeler için bir fmap yapısında olduğunu söylemiştik. Benzer şekilde farklı tipler için de tanimlanabilecek,

Haskell

```
map :: (a -> b) -> [a] -> [b]
treemap :: (a -> b) -> Tree a -> Tree b
mmap :: (a -> b) -> Maybe a -> Maybe b
iomap :: (a -> b) -> IO a -> IO b
funmap :: (a -> b) -> (c -> a) -> (c -> b)
```

tüm bu fonksiyonlar aslında *fmap* fonksiyonunun ozel halleridir.

İşte bu noktada en önemli soru, üzerinde tanımlanan tiplerden bağımsız olarak hareket edebilecek, en genel bir *map* fonksiyonunun tanımlanıp tanımlanamayacağıdır.

Bu sorunun cevabı da aslında, bizim yukarıda vermiş olduğumuz, herhangi iki tip arasındaki bir $h : * -- > *$ fonksiyonu için,

List Yapısının Özellikleri ve Funktor

Yukarıda tanımladığımız *map* fonksiyonunun aslında listeler için bir fmap yapısında olduğunu söylemiştik. Benzer şekilde farklı tipler için de tanimlanabilecek,

Haskell

```
map :: (a -> b) -> [a] -> [b]
treemap :: (a -> b) -> Tree a -> Tree b
mmap :: (a -> b) -> Maybe a -> Maybe b
iomap :: (a -> b) -> IO a -> IO b
funmap :: (a -> b) -> (c -> a) -> (c -> b)
```

tüm bu fonksiyonlar aslında *fmap* fonksiyonunun ozel halleridir.

İşte bu noktada en önemli soru, üzerinde tanımlanan tiplerden bağımsız olarak hareket edebilecek, en genel bir *map* fonksiyonunun tanımlanıp tanımlanamayacağıdır.

Bu sorunun cevabı da aslında, bizim yukarıda vermiş olduğumuz, herhangi iki tip arasındaki bir $h : * -- > *$ fonksiyonu için,

Haskell

```
fmap :: (a -> b) -> f a -> f b
```

şeklinde tanımlanan *fmap* fonksiyonudur.

List Yapısının Özellikleri ve Funktor

fmap fonksiyonunun yukarıda incelediğimiz yapısı ve özellikleri göz önüne aldığında aklımıza ilk olarak, bu fonksiyonun kategoriksel olarak nasıl davrandığı, yani,

List Yapısının Özellikleri ve Funktor

fmap fonksiyonunun yukarıda incelediğimiz yapısı ve özellikleri göz önüne aldığında aklımıza ilk olarak, bu fonksiyonun kategoriksel olarak nasıl davrandığı, yani,

- *gf* gibi bir kompozisyon üzerindeki etkisi için hangi özelliklere sahip olduğu

List Yapısının Özellikleri ve Funktor

fmap fonksiyonunun yukarıda incelediğimiz yapısı ve özellikleri göz önüne aldığında aklımıza ilk olarak, bu fonksiyonun kategoriksel olarak nasıl davrandığı, yani,

- *gf* gibi bir kompozisyon üzerindeki etkisi için hangi özelliklere sahip olduğu
- Özel olarak birim fonksiyonlar için nasıl bir etkisi olduğu

List Yapısının Özellikleri ve Funktor

fmap fonksiyonunun yukarıda incelediğimiz yapısı ve özellikleri göz önüne aldığında aklımıza ilk olarak, bu fonksiyonun kategoriksel olarak nasıl davrandığı, yani,

- *gf* gibi bir kompozisyon üzerindeki etkisi için hangi özelliklere sahip olduğu
- Özel olarak birim fonksiyonlar için nasıl bir etkisi olduğu

soruları gelmektedir.

List Yapısının Özellikleri ve Funktor

Bu soruların cevapları incelenecek olursa,

List Yapısının Özellikleri ve Funktor

Bu soruların cevapları incelenecek olursa,

Haskell

```
f :: a -> b

g :: b -> c

g.f :: a -> c
```

Bu soruların cevapları incelenecek olursa,

Haskell

```
f :: a -> b

g :: b -> c

g.f :: a -> c
```

şeklindeki fonksiyonlar ve kompozisyonlar için *fmap* fonksiyonunun,

Bu soruların cevapları incelenecek olursa,

Haskell
```
f :: a -> b

g :: b -> c

g.f :: a -> c
```

şeklindeki fonksiyonlar ve kompozisyonlar için *fmap* fonksiyonunun,

Haskell
```
fmap :: (Functor f) => ( a -> b ) -> f a -> f b
```

yapısı gereği asosyatifliği sağladığı,

yani basit bir örnekle,

Haskell

```
f :: Int -> Int
f x = x*x

g :: Int -> Int
g y = y+2
```

fonksiyonları için,

yani basit bir örnekle,

Haskell

```
f :: Int -> Int
f x = x*x

g :: Int -> Int
g y = y+2
```

fonksiyonları için,

Haskell

```
fmap (g.f) [1,2]
> [3,6]

( fmap g . fmap f ) [1,2]
> [3,6]
```

olduğu,

ve dolayısıyla da,

ve dolayısıyla da,

$$\boxed{\text{fmap g.f} = \text{fmap g . fmap f}} \qquad \text{.....(2)}$$

şartını doğruladığı görülür.

Diğer yandan,

Haskell

```
idS :: S -> S
idS x = x
```

şeklinde tanımlı id_S ve,

List Yapısının Özellikleri ve Funktor

Diğer yandan,

Haskell

```
idS :: S -> S
idS x = x
```

şeklinde tanımlı id_S ve,

Haskell

```
idListS :: [a] -> [a]
idListS y = y
```

şeklinde tanımlı $id_{List(S)}$ için,

List Yapısının Özellikleri ve Funktor

Diğer yandan,

Haskell
```
idS :: S -> S
idS x = x
```

şeklinde tanımlı id_S ve,

Haskell
```
idListS :: [a] -> [a]
idListS y = y
```

şeklinde tanımlı $id_{List(S)}$ için,

Haskell
```
fmap idS (x:xs) = idS x : fmap idS xs
```

olmaktadır.

List Yapısının Özellikleri ve Funktor

Bunu basit bir örnekle incelersek,

List Yapısının Özellikleri ve Funktor

Bunu basit bir örnekle incelersek,

Haskell

```
fmap idS [1,2,3]
> [1,2,3]

idListS [1,2,3]
> [1,2,3]
```

olduğu ve dolayısıyla da,

Bunu basit bir örnekle incelersek,

Haskell

```
fmap idS [1,2,3]
> [1,2,3]

idListS [1,2,3]
> [1,2,3]
```

olduğu ve dolayısıyla da,

$$\boxed{\text{fmap idS} = \text{idListS}} \qquad \text{.....(1)}$$

şartını doğruladığı görülür.

Sonuç:

List yapısının ve funktor tanımının sahip olduğu özellikler göz önüne alındığında,

List Yapısının Özellikleri ve Funktor

Sonuç:

List yapısının ve funktor tanımının sahip olduğu özellikler göz önüne alındığında,

- (1) ile F1 şartının,

List Yapısının Özellikleri ve Funktor

Sonuç:

List yapısının ve funktor tanımının sahip olduğu özellikler göz önüne alındığında,

- (1) ile F1 şartının,
- (2) ile de F2 şartının

Sonuç:

List yapısının ve funktor tanımının sahip olduğu özellikler göz önüne alındığında,

- (1) ile F1 şartının,
- (2) ile de F2 şartının

birebir aynı olduğu görülmektedir.

Sonuç:

List yapısının ve funktor tanımının sahip olduğu özellikler göz önüne alındığında,

- (1) ile F1 şartının,
- (2) ile de F2 şartının

birebir aynı olduğu görülmektedir.

Bu durumda *List* in $C(\mathcal{L})$ küçük kategorisinden $C(\mathcal{L})$ küçük kategorisine,

Sonuç:

List yapısının ve funktor tanımının sahip olduğu özellikler göz önüne alındığında,

- (1) ile F1 şartının,
- (2) ile de F2 şartının

birebir aynı olduğu görülmektedir.

Bu durumda *List* in $C(\mathcal{L})$ küçük kategorisinden $C(\mathcal{L})$ küçük kategorisine,

$$\begin{array}{lccl} List: & C(\mathcal{L}) & \longrightarrow & C(\mathcal{L}) \\ & A & \rightsquigarrow & List(A) \sim A \text{ nın tüm listeleri} \\ & f & \rightsquigarrow & List(f) = fmap\ f \end{array} \quad \text{.....(3)}$$

şeklinde bir **funktor** olduğu sonucuna ulaşılmaktadır.

Monoid

Monoid

Hatırlatma: M bir küme ve $\circ$ da bu küme üzerinde bir ikili işlem olsun. Eğer bu M kümesi, üzerinde tanımlı $\circ$ işlemi altında, asosyatiflik ve birimlilik şartlarını sağlıyorsa bu durumda M ye, $\circ$ işlemine göre bir **monoid** denir.

Monoid

Hatırlatma: M bir küme ve $\circ$ da bu küme üzerinde bir ikili işlem olsun. Eğer bu M kümesi, üzerinde tanımlı $\circ$ işlemi altında, asosyatiflik ve birimlilik şartlarını sağlıyorsa bu durumda M ye, $\circ$ işlemine göre bir **monoid** denir.

Hatırlatma: **(Monoid Homomorfizmi) :** M ve M' birer monoid olmak üzere,

Monoid

Hatırlatma: M bir küme ve $\circ$ da bu küme üzerinde bir ikili işlem olsun. Eğer bu M kümesi, üzerinde tanımlı $\circ$ işlemi altında, asosyatiflik ve birimlilik şartlarını sağlıyorsa bu durumda M ye, $\circ$ işlemine göre bir **monoid** denir.

Hatırlatma: **(Monoid Homomorfizmi) :** M ve M' birer monoid olmak üzere,

$$\phi : (M, \circ, e) \longrightarrow (M', \circ', e')$$

fonksiyonunun bir homomorfizm olabilmesi için gerek ve yeter şart, $\forall a, b \in M$ için,

Monoid

Hatırlatma: M bir küme ve $\circ$ da bu küme üzerinde bir ikili işlem olsun. Eğer bu M kümesi, üzerinde tanımlı $\circ$ işlemi altında, asosyatiflik ve birimlilik şartlarını sağlıyorsa bu durumda M ye, $\circ$ işlemine göre bir **monoid** denir.

Hatırlatma: **(Monoid Homomorfizmi) :** M ve M' birer monoid olmak üzere,

$$\phi : (M, \circ, e) \longrightarrow (M', \circ', e')$$

fonksiyonunun bir homomorfizm olabilmesi için gerek ve yeter şart, $\forall a, b \in M$ için,

$$\phi(a \circ b) = \phi(a) \circ' \phi(b)$$

olmasıdır.

Monoid

Hatırlatma: M bir küme ve $\circ$ da bu küme üzerinde bir ikili işlem olsun. Eğer bu M kümesi, üzerinde tanımlı $\circ$ işlemi altında, asosyatiflik ve birimlilik şartlarını sağlıyorsa bu durumda M ye, $\circ$ işlemine göre bir **monoid** denir.

Hatırlatma: **(Monoid Homomorfizmi) :** M ve M' birer monoid olmak üzere,

$$\phi : (M, \circ, e) \longrightarrow (M', \circ', e')$$

fonksiyonunun bir homomorfizm olabilmesi için gerek ve yeter şart, $\forall a, b \in M$ için,

$$\phi(a \circ b) = \phi(a) \circ' \phi(b)$$

olmasıdır. Ayrıca bu homomorfizmin birimi koruduğu, yani $\phi(e) = e'$ olduğu açıktır.

Monoid

Hatırlatma: Kategori Teori'deki $\mathcal{C} = Monoid$ kategorisi hatırlandığında, bu kategori,

Monoid

Hatırlatma: Kategori Teori'deki $\mathcal{C} = Monoid$ kategorisi hatırlandığında, bu kategori,

- $Ob(\mathcal{C}) \sim$ Monoidlerin Sınıfı

Monoid

Hatırlatma: Kategori Teori'deki $\mathcal{C} = Monoid$ kategorisi hatırlandığında, bu kategori,

- $Ob(\mathcal{C}) \sim$ Monoidlerin Sınıfı
- $Mor(\mathcal{C}) \sim$ Monoid Homomorfizmleri

Monoid

Hatırlatma: Kategori Teori'deki $\mathcal{C} = Monoid$ kategorisi hatırlandığında, bu kategori,

- $Ob(\mathcal{C}) \sim$ Monoidlerin Sınıfı
- $Mor(\mathcal{C}) \sim$ Monoid Homomorfizmleri
- $Komp(\mathcal{C}) \sim$ Bileşke İşlemi

Monoid

Hatırlatma: Kategori Teori'deki $\mathcal{C} = Monoid$ kategorisi hatırlandığında, bu kategori,

- $Ob(\mathcal{C}) \sim$ Monoidlerin Sınıfı
- $Mor(\mathcal{C}) \sim$ Monoid Homomorfizmleri
- $Komp(\mathcal{C}) \sim$ Bileşke İşlemi

olmak üzere,

Monoid

Hatırlatma: Kategori Teori'deki $\mathcal{C} = Monoid$ kategorisi hatırlandığında, bu kategori,

- $Ob(\mathcal{C}) \sim$ Monoidlerin Sınıfı
- $Mor(\mathcal{C}) \sim$ Monoid Homomorfizmleri
- $Komp(\mathcal{C}) \sim$ Bileşke İşlemi

olmak üzere,

$$\mathcal{C} = (Ob(\mathcal{C}), Mor(\mathcal{C}), Komp(\mathcal{C}))$$

şeklindedir.

Monoid

Keyfi bir S tipi (kümesi) için $List(S)$ aslında,

Haskell

```
[ ] ++ xs = xs

xs ++ [ ] = xs

xs ++ ( ys ++ zs ) = ( xs ++ ys ) ++ zs
```

Monoid

Keyfi bir S tipi (kümesi) için $List(S)$ aslında,

Haskell

```
[ ] ++ xs = xs

xs ++ [ ] = xs

xs ++ ( ys ++ zs ) = ( xs ++ ys ) ++ zs
```

Haskell

```
[1,2,3] ++ [4,5]
>[1,2,3,4,5]

[1,2,3] ++ [ ]
>[1,2,3]

[1,2,3] ++ ( [4,5] ++ [6,7] )
> [1,2,3,4,5,6,7]

( [1,2,3] ++ [4,5] ) ++ [6,7]
> [1,2,3,4,5,6,7]
```

şeklinde tanımlı $++$ işlemi yardımıyla şu ana kadar farkında olmadığımız ekstra bir yapıya sahiptir.

Monoid

Öyle ki,

- Herhangi iki listeyi uç uca birleştirme anlamındaki $++$ işlemi,

Monoid

Öyle ki,

- Herhangi iki listeyi uç uca birleştirme anlamındaki $++$ işlemi,
- Bu işlemin herhangi üç liste için asosyatifliği koruduğu,

Monoid

Öyle ki,

- Herhangi iki listeyi uç uca birleştirme anlamındaki $++$ işlemi,
- Bu işlemin herhangi üç liste için asosyatifliği koruduğu,
- Bu işlem üzerinde $[\,]$ boş listesinin bir etkisi olmadığı,

Monoid

Öyle ki,

- Herhangi iki listeyi uç uca birleştirme anlamındaki $++$ işlemi,
- Bu işlemin herhangi üç liste için asosyatifliği koruduğu,
- Bu işlem üzerinde $[\,]$ boş listesinin bir etkisi olmadığı,

bu işlemin tanımı gereği açıktır.

Monoid

O halde artık S herhangi bir tip (küme) olmak üzere;

(List (S) , ++ , [ ])

yapısı,

Monoid

O halde artık S herhangi bir tip (küme) olmak üzere;

(List (S) , ++ , [ ])

yapısı,

- G1) Asosyatiflik

Monoid

O halde artık S herhangi bir tip (küme) olmak üzere;

(List (S) , ++ , [ ])

yapısı,

- G1) Asosyatiflik
- G2) Birimlilik

Monoid

O halde artık S herhangi bir tip (küme) olmak üzere;

(List (S) , ++ , [ ])

yapısı,

- G1) Asosyatiflik
- G2) Birimlilik

şartlarını sağlayarak bir **monoid** yapısı oluşturmaktadır.

Monoid

O halde peki *List* nin küme yapısından bir üste çıkarak, aslında bir monoid olması dolayısıyla (3) de tanımlanan funktoru da bir adım daha ilerleterek;

Monoid

O halde peki *List* nin küme yapısından bir üste çıkarak, aslında bir monoid olması dolayısıyla (3) de tanımlanan funktoru da bir adım daha ilerleterek;

$$\boxed{List : C(\mathcal{L}) \longrightarrow Monoid} \qquad \text{.....(4)}$$

şekline dönüştürebilir miyiz?

Monoid

O halde peki *List* nin küme yapısından bir üste çıkarak, aslında bir monoid olması dolayısıyla (3) de tanımlanan funktoru da bir adım daha ilerleterek;

$$\boxed{List : C(\mathcal{L}) \longrightarrow Monoid} \qquad \text{.....(4)}$$

şekline dönüştürebilir miyiz?

Bu ifadenin gerçeklenebilmesi için, *List* in kümeleri monoidlere taşımasının yanında ayrıca, fonksiyonları da monoid homomorfizmlerine taşımalıdır.

Monoid

Burada yukarıdaki hatırlatma altında herhangi iki,

$$(List(S), ++, [\,]) \quad \& \quad (List(S'), ++, [\,])$$

monoidi ve bu iki monoid arasındaki homomorfizm de,

Monoid

Burada yukarıdaki hatırlatma altında herhangi iki,

$$(List(S), ++, [\,]) \quad \& \quad (List(S'), ++, [\,])$$

monoidi ve bu iki monoid arasındaki homomorfizm de,

$$f : S \longrightarrow S'$$

fonksiyonu için,

Monoid

Burada yukarıdaki hatırlatma altında herhangi iki,

$$(List(S), ++, [\,]) \quad \& \quad (List(S'), ++, [\,])$$

monoidi ve bu iki monoid arasındaki homomorfizm de,

$$f : S \longrightarrow S'$$

fonksiyonu için,

$$fmap\ f \ : \ (List(S), ++, [\,]) \longrightarrow (List(S'), ++, [\,])$$

şeklinde alındığı takdirde bu fonksiyonun bir homomorfizm olabilmesi için sağlaması gereken şartlar,

Monoid

- fmap _ [ ] = [ ]

Monoid

- fmap _ [ ] = [ ]
- fmap f (xs ++ ys) = fmap f xs ++ fmap f ys

Monoid

- fmap _ [ ] = [ ]
- fmap f (xs ++ ys) = fmap f xs ++ fmap f ys

olacak olup, bu şartların geçerli olduğu zaten *fmap* fonksiyonunun yapısı ve özellikleri kısmında gösterilmiştir.

Monoid

Sonuç:

Tüm bu özellikler altında *List* in $C(\mathcal{L})$ kategorisinden *Monoid* kategorisine,

Monoid

Sonuç:

Tüm bu özellikler altında *List* in $C(\mathcal{L})$ kategorisinden *Monoid* kategorisine,

$$\boxed{List : C(\mathcal{L}) \longrightarrow Monoid} \qquad \text{.....(4)}$$

şeklinde bir **funktor** 'dur.

Doğal Transformasyon

Doğal Transformasyon

Hatırlatma: $F : \mathcal{C} \to \mathcal{D}$ ve $G : \mathcal{C} \to \mathcal{D}$ küçük kategoriler arasında iki funktor olsun.

Doğal Transformasyon

Hatırlatma: $F : \mathcal{C} \to \mathcal{D}$ ve $G : \mathcal{C} \to \mathcal{D}$ küçük kategoriler arasında iki funktor olsun.

(i) $\mathcal{C}$ nin her A objesi için,

$$\begin{array}{rccl} \eta : & \mathrm{Ob}(\mathcal{C}) & \longrightarrow & \mathrm{Mor}(\mathcal{D}) \\ & A & \longmapsto & \eta(A) = \eta_A : F(A) \longrightarrow G(A) \end{array}$$

şeklinde tanımlı η_A , $\mathcal{D}$ nin bir morfizmi;

Doğal Transformasyon

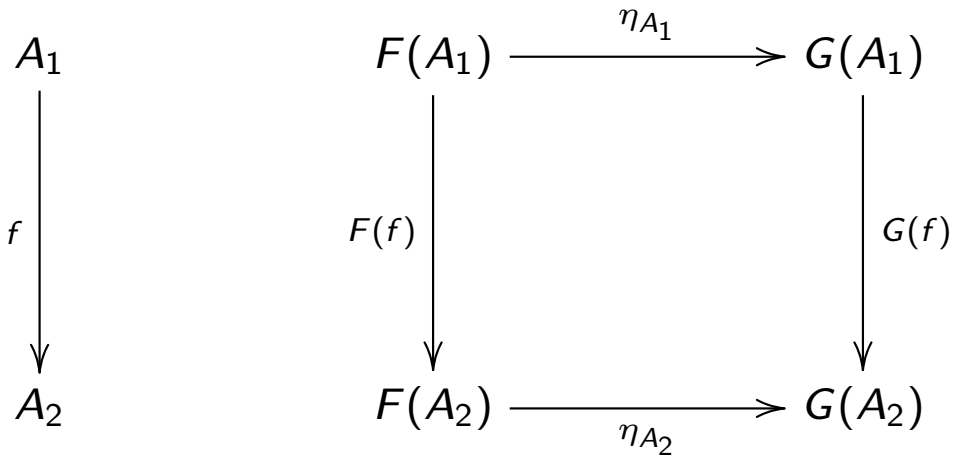
Hatırlatma: $F : \mathcal{C} \to \mathcal{D}$ ve $G : \mathcal{C} \to \mathcal{D}$ küçük kategoriler arasında iki funktor olsun.

(i) $\mathcal{C}$ nin her A objesi için,

$$\begin{array}{rccc} \eta : & \mathrm{Ob}(\mathcal{C}) & \longrightarrow & \mathrm{Mor}(\mathcal{D}) \\ & A & \longmapsto & \eta(A) = \eta_A : F(A) \longrightarrow G(A) \end{array}$$

şeklinde tanımlı η_A , $\mathcal{D}$ nin bir morfizmi;

(ii) $\mathcal{C}$ nin her $f : A_1 \to A_2$ morfizmi için



diyagramı değişmeli;

Doğal Transformasyon

şartları sağlanıyorsa;

Doğal Transformasyon

şartları sağlanıyorsa;

(F, η, G) üçlüsüne bir **doğal transformasyon** denir.

Doğal Transformasyon

şartları sağlanıyorsa;

(F, η, G) üçlüsüne bir **doğal transformasyon** denir.

(Ayrıca $\eta : F \Rightarrow G$ şeklinde de gösterilebilir.)

Doğal Transformasyon

Fonksiyonel Programlama Dilinde keyfi bir S tipi (kümesi) üzerindeki listeler için, bu listeyi ters çeviren, yani elemanları sıra anlamında yansıtarak yeni bir liste oluşturan *reverse* adında bir fonksiyon bulunmaktadır.

Doğal Transformasyon

Fonksiyonel Programlama Dilinde keyfi bir S tipi (kümesi) üzerindeki listeler için, bu listeyi ters çeviren, yani elemanları sıra anlamında yansıtarak yeni bir liste oluşturan *reverse* adında bir fonksiyon bulunmaktadır.

Bu fonksiyonun yapısını Haskell için inceleyecek olursak,

Doğal Transformasyon

Fonksiyonel Programlama Dilinde keyfi bir S tipi (kümesi) üzerindeki listeler için, bu listeyi ters çeviren, yani elemanları sıra anlamında yansıtarak yeni bir liste oluşturan *reverse* adında bir fonksiyon bulunmaktadır.

Bu fonksiyonun yapısını Haskell için inceleyecek olursak,

Haskell

```
reverse :: [a] -> [a]
reverse [ ] = [ ]
reverse ( x : xs ) = reverse xs ++ [x]
```

şeklinde tanımlı olup, örneğin herhangi bir [1,4,5] listesi için, bu listenin tersi,

Haskell

```
reverse [1,4,5]
> [5,4,1]
```

elde edilir.

Doğal Transformasyon

Bu durumda *reverse* fonksiyonu,

$$reverse_S : List(S) \longrightarrow List(S)$$

veya

Doğal Transformasyon

Bu durumda *reverse* fonksiyonu,

$$reverse_S : List(S) \longrightarrow List(S)$$

veya

$$reverse\ (S) : List(S) \longrightarrow List(S)$$

şeklinde tanımlıdır.

Doğal Transformasyon

Bu durumda *reverse* fonksiyonu,

$$reverse_S : List(S) \longrightarrow List(S)$$

veya

$$reverse\ (S) : List(S) \longrightarrow List(S)$$

şeklinde tanımlıdır. Hatta dikkat edilirse,

$$reverse : List \longrightarrow List$$

şeklinde olduğu görülmektedir.

Doğal Transformasyon

O halde peki *List* yapısının bir funktor olması, ve ayrıca *reverse* dönüşümünün iki funktor arasında tanımlı olduğundan faydalanarak acaba bu *reverse* fonksiyonunun kategoriksel olarak bir doğal transformasyon olduğunu söyleyebilir miyiz?

Doğal Transformasyon

Yukarıdaki doğal transformasyon tanımı için,

Doğal Transformasyon

Yukarıdaki doğal transformasyon tanımı için,

- $\mathcal{C} = \mathcal{D} = C(\mathcal{L})$
- $F = G = List$
- $\eta = reverse$

alınsın.

Doğal Transformasyon

Bu durumda,

Doğal Transformasyon

Bu durumda,

$$\begin{aligned} List &: C(\mathcal{L}) \longrightarrow C(\mathcal{L}) \\ List &: C(\mathcal{L}) \longrightarrow C(\mathcal{L}) \end{aligned}$$

funktorları ve $S, S' \in Ob(C(\mathcal{L}))$ için,

Doğal Transformasyon

Bu durumda,

$$\begin{aligned} List &: C(\mathcal{L}) \longrightarrow C(\mathcal{L}) \\ List &: C(\mathcal{L}) \longrightarrow C(\mathcal{L}) \end{aligned}$$

funktorları ve $S, S' \in Ob(C(\mathcal{L}))$ için,

$$f : S \longrightarrow S'$$

fonksiyonu tanımlı olduğunda,

Doğal Transformasyon

(i) $C(\mathcal{L})$ in her S objesi için,

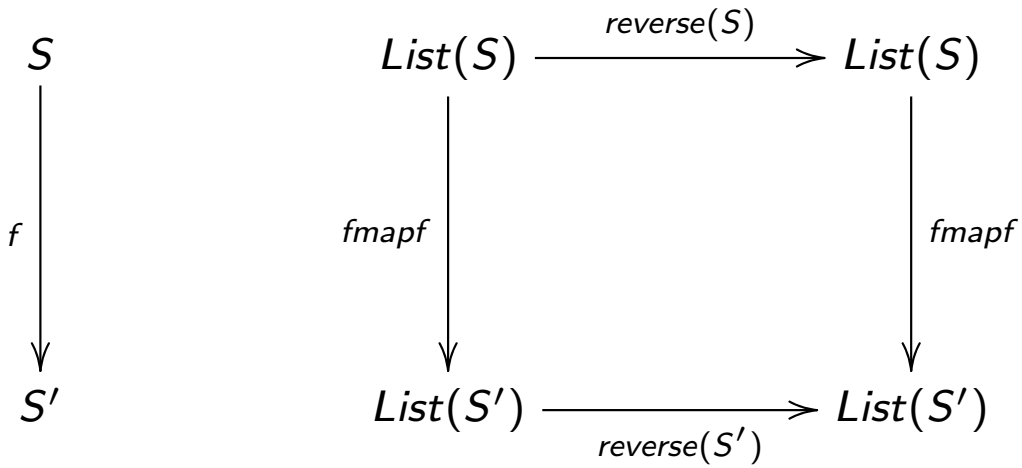
(i) $C(\mathcal{L})$ in her S objesi için,

$$\begin{array}{lccl} reverse: & Ob(C(\mathcal{L})) & \longrightarrow & Mor(C(\mathcal{L})) \\ & S & \longmapsto & reverse\ (S) : List\ (S) \longrightarrow List\ (S) \end{array}$$

şeklinde tanımlı olduğu *reverse* fonksiyonunun yapısı gereği açıkça görülmektedir.

Doğal Transformasyon

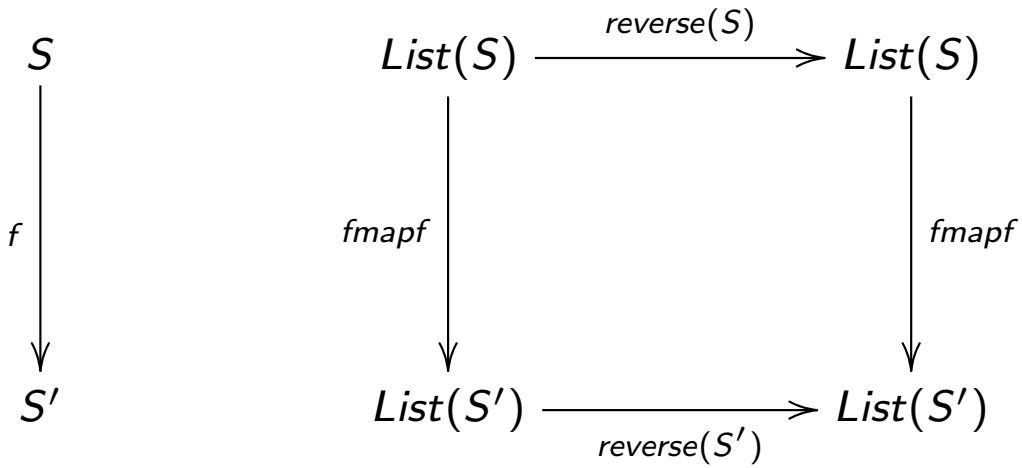
(ii) $\mathcal{C}$ nin her $f : S \to S'$ morfizmi için,



diyagramının değişmeli

Doğal Transformasyon

(ii) $\mathcal{C}$ nin her $f : S \to S'$ morfizmi için,



diyagramının değişmeli yani,

$$reverse\ (S')\ .\ fmap\ f\ =\ fmap\ f\ .\ reverse\ (S)$$

olması gerekmektedir.

Doğal Transformasyon

Örneğin,

Doğal Transformasyon

Örneğin,

Haskell

```
reverse ( fmap f [1,2,3] )
> [9,4,1]
```

ve ayrıca,

Doğal Transformasyon

Örneğin,

Haskell

```
reverse ( fmap f [1,2,3] )
> [9,4,1]
```

ve ayrıca,

Haskell

```
fmap f ( reverse [1,2,3] )
> [9,4,1]
```

olup,

Doğal Transformasyon

Örneğin,

Haskell

```
reverse ( fmap f [1,2,3] )
> [9,4,1]
```

ve ayrıca,

Haskell

```
fmap f ( reverse [1,2,3] )
> [9,4,1]
```

olup,

$$reverse\ (S')\ .\ fmap\ f\ =\ fmap\ f\ .\ reverse\ (S)$$

olduğu açıktır.

Sonuç:

Yani tüm bu özellikler altında *reverse* fonksiyonu,

Doğal Transformasyon

Sonuç:

Yani tüm bu özellikler altında *reverse* fonksiyonu,

$$reverse : List \longrightarrow List$$

şeklinde bir **doğal transformasyon**'dur.

Ödevler :))

Ödev 1

Yukarıda, List yapısı yardımıyla bir monoid yapısı elde ettik.

Peki bu yapı neden sadece bir monoid'de kalarak, daha da ötesinde bir grup yapısı oluşturamadı?

İnceleyiniz.

Ödev 2

tail ve **init** fonksiyonlarının yapısını, Haskell üzerinde vereceğiniz örneklerle araştırınız.

Ödev 3

- $\mathcal{C} = \mathcal{D} = Set$
- $F = G = List$
- $\eta = tail$

yapısının bir **doğal transformasyon** olduğunu gösteriniz.

Ödev 4

- $\mathcal{C} = \mathcal{D} = Set$
- $F = G = List$
- $\eta = init$

yapısının bir **doğal transformasyon** olduğunu gösteriniz.

Haskell Programı:

`http://hackage.haskell.org/platform/windows.html`