F# ile Fonksiyonel Programlama

İçindekiler

    - Gevşek Diziler (Sequences)
    - Sorgu İfadeleri (Query Expressions)
- 5.Bölüm : Genel Amaçlı Programlama
    - Değişken ve Değişmeyen Kavramları (Immutability and Mutability)
    - .NET Bellek Yönetimi
    - Değişken İçeriğini Değiştirme
    - Diziler
    - .NET Yığın Yapıları Kullanımı
    - Döngü Yapıları (For ve While)
    - Koşullu Dallanma Yapıları (If/Else)
    - İstisna Yönetimi (Exceptions)
- 6.Bölüm : Nesne Tabanlı Programlama ve Sınıflar
    - Fonksiyonel Bir Dilde Neden Nesne Tabanlı Programlama Desteği Var?
    - Sınıf Tanımlama
    - Sınıf Özellik ve Üyeleri
    - Sınıflar Arası Kalıtım
    - Ara Birim Kullanımı (Interfaces)
- 7.Bölüm : İleri Seviye Fonksiyonel Programlama Yöntemleri
    - Aktif Desenler (Active Patterns)
    - Liste Modülü
    - Kuyruk Özyenilemeli Fonksiyonlar
    - Fonksiyonlar ile Programlama
    - Fonksiyonel Programlama Desenleri
- 8.Bölüm : Asenkron ve Paralel Programlama
    - İşletim Sistemi İplikleri ile Çalışma (Thread)
    - Asenkron Programlama
    - Asenkreon Programlama Kütüphanesi
    - Paralel Programlama
    - Paralel Programlama Kütüphanesi
- 9.Bölüm : Örnek Uygulamalar

- Veritabanı Uygulaması
- Veri Ayıklama ve Analiz Uygulaması
- Web Programlama Uygulaması
- Finansal Uygulalma : Kredi Puanı Hesaplayıcı
- UrhoSharp İle Örnek Oyun

1. Bölüm : Giriş

Bu bölümün ilk kısmında matematiksel anlamda fonksiyonları ve fonksiyonların bazı ilginç özelliklerini ele alacağız. Bölümün ikinci kısmında ise fonksiyonel programlamanın tanımını yaparak F#'ın kısa tarihçesini aktarıp "Neden F#?" ve "F# programlama dili neye benzer?" sorularının cevaplarını arayacağız.

1.1 Fonksiyonlara Matematiksel Bakış

Fonksiyonel programlamanın temeli matematiksel fonksiyonlar ve fonksiyonların bazı özellikleri üzerine inşa edilmiştir. Matematiksel açıdan **fonksiyon** tanımlarından bir tanesi aşağıdaki gibi yapılır

> X ve Y iki küme, $f \subset X \times Y$ bir bağıntı olsun. Aşağıdaki koşullar sağlanırsa f bağıntısına bir fonksiyon denir:
>
> 1. $\forall x \in X, \exists y \in Y : (x,y) \in f$,
> 2. $(x,y),(x,y') \in f \Rightarrow y = y'$
>
> Burada X'e tanım kümesi, Y'ye ise değer kümesi denir. Tanımından da anlaşılacağı gibi fonksiyon, tanım kümesindeki her elemanı, değer kümesindeki tek bir elemanla eşleştiren bir bağıntıdır. Bu yüzden fonksiyonlarda xfy veya $(x,y) \in f$ gösterimi yerine $y = f(x)$ gösterimi kullanılır. Bir fonksiyona bazen dönüşüm de denir. Eğer f, X'den Y'ye bir fonksiyon ise bu durum $f : X \rightarrow Y$ ile ya da $X \rightarrow fY$ ile gösterilir.

Yukarıdaki tanımda belirtilen 1. koşuldaki **∀x∈X** ifadesini "X kümesinin elemanı olan tüm x değerleri", **∃y∈Y** ifadesini ise "Y kümesinin elemanı olan bir y değeri" şeklinde okuyabilirsiniz. **∀** ve **∃** sembolleri matematikte nicelik/miktar belirten sembollerdir, **∀** sembolü **tüm** ve **∃** sembolü de **bir** anlamında miktar belirtir. Bu tanımda yer alan diğer iki sembolden **∈** sembolü bir değerin bir kümenin elemanı olduğunu ifade eder, **⊂** sembolü ise **alt küme** anlamına gelir ve tanımda (x,y) değer çiftinin f fonksiyonunun üreteceği sonuç kümesinin bir alt kümesi olduğu anlamını taşır.

Tanımın ikinci koşulu olan "$(x,y),(x,y')\in f \Rightarrow y=y'$" ifadesini ise şöyle yorumlarız; f fonksiyonu, X değer kümesinin bir x elemanını Y kümesinin y ve y′ şeklinde iki elemanı ile eşleştiriyorsa y ve y′ değerleri birbirine eşittir. Başka bir deyişle, f fonksiyonu X değer kümesinin elemanı olan bir x değerini her zaman Y kümesinin bir elemanı olan aynı y değeri ile eşleştirir.

Şimdi gelin bu fonksiyon tanımını görselleştirerek basit bir örnek ile somutlaştıralım.

f(x) = x * x şeklinde bir fonksiyon tanımı olsun. Bu fonksiyon girdi olarak verilen x değerinin karesini hesaplar. Daha matematiksel bir şekilde ifade edecek olursak; bu fonksiyon doğal sayılar kümesinin elemanı olan tüm **x** değerlerini yine doğal sayılar kümesinin elemanı olan bir **x*x** değeri ile eşleştirmektedir.

Yukarıdaki şekilde yer alan **tanım kümesi** ve **değer kümesi** kavramları önemlidir, zira fonksiyonları tanım kümesindeki elemanları değer kümesindeki elemanlar ile eşleştiren birer dönüşüm olarak da ifade edebiliriz.

Yukarıdaki örnekte

- Tanım Kümesi A : A{1,2,3}
- Değer Kümesi B : B{a,b,c,d}
- Görüntü Kümesi : f(A) = {a,d}

f fonksiyonunu da f(A) = {(1,a),(2,a),(3,d)} şeklindeki eşleştirmelerin kümesi olarak tanımlarız.

1.2 Fonksiyonların İlginç Özellikleri

Matematiksel fonksiyonların fonksiyonel programlama dillerinin yapısını yakından etkileyen belirleyici iki önemli özelliğinden bahsedebiliriz, bunlar

- Fonksiyonlar tanım kümesindeki bir elemanı her zaman değer kümesindeki aynı eleman ile eşleştirir
- Fonksiyonların yan etkileri yoktur

f(x) = x * x şeklindeki fonksiyon tanımını örnek olarak ele alırsak, bu fonksiyonun tanım kümesindeki 2 değerini değer kümesindeki 4 değeri ile (f(2)=4), 3 değerini de 9 değeri ile eşleştirdiğini (f(3) = 9) söyleriz. Bu fonksiyonun **f(2) ≠ 4** veya **f(3) ≠ 9** şeklinde bir eşleştirme yapması asla mümkün değildir. Programcı terimleri ile ifade edecek olursak fonksiyonlar **girdi parametresi olarak kullanılan bir değer için her zaman aynı çıktıyı üretir**.

f(x) = x * x fonksiyonunun F# ile matematiksel tanımına uygun olarak basit bir eşleştirme dönüşümü olarak aşağıdaki gibi ifade edebiliriz.

```
(* 01_01.fsx *)

let f (x) =
```

```
    match x with
    | 1 -> 1
    | 2 -> 4
    | 3 -> 9
    | _ -> -1 //Diğer olası tüm değerler
```

Dikkat ederseniz fonksiyonları bu noktaya kadar hep *eşleştirme yapan birer dönüşüm* olarak tanımlamaya özen gösterdik. Eğer fonksiyonel olmayan programlama dilleri ile tecrübeniz varsa fonksiyonların veya metodların hesaplama yapmak için kullanıldığını düşünüyor olabilirsiniz. Ancak yükarıdaki **f(x) = x * x** örneğinde de görebileceğiniz gibi fonksiyonlar aslında herhangi bir hesaplama yapmazlar, fonksiyonlar basitçe iki kümenin elemanlarını birbirleri ile eşleştirirler. Bu nedenle fonksiyonları programcı bakış açısıyla herhangi bir hesaplama yapmayan basit birer switch/case (C,C++, Java, C#, JavaScript gibi dillerin hepsinde olan koşullu dallanma yapısı) kod bloğu olarak düşünebilirsiniz.

Ancak, switch/case benzeri yapılar yazım açısından zahmetli olup genellemeye uygun değildirler. Tanım kümesinin tüm elemanlarının switch/case ile değer kümesinden bir eleman ile eşleştirilmesi pratik olarak mümkün değildir. Bu nedenle fonksiyonları, bir hesaplama yaptığı izlenimine kapılmamıza da neden olan, aşağıdaki şekilde yazarak genelleştirilebilir.

```
(* 01_02.fsx *)

let f (x) = x * x
```

Fonksiyonların ikinci ilginç özelliği ise yan etkilerinin olmamasıdır. **Yan etki** fonksiyonun eşleştirme dönüşümünü yaparken giridi olarak

verilen tanım kümesindeki değerin de değişmesi anlamına gelir. Bu durumda fonksiyon sadece tanım kümesindeki değeri değer kümesi ile eşleştirmiş olmaz yan etki olarak tanım kümesindeki değeri de değiştirmiş olur.

Örneğin **f(x) = x * x** fonksiyonuna girdi olarak verilen değer kümesindeki **x = 5** değerinin **y = f 5** ifadesi ile yapılan dönüşüm işlemi sonrasında hala 5'e eşit olması f(x) fonksiyonunun yan etkisi olmadığını gösterir.

```
(* 01_03.fsx *)

let f(x) = x * x   // fonksiyon tanımı

let x = 5          // Tanım kümesinden 5 değeri
let y = f 5        // y = f(5)

printfn "x = %d" x // x değeri değişmiş mi kontrolü

printfn "y = %d" y // y = f(5) dönüşümü yapılmış m
ı kontrolü
```

Bu iki özelliği sağlayan fonksiyonları matematikçiler ve fonksiyonel programcılar **saf fonksiyonlar** olarak adlandırır. Saf fonksiyonlar aynı girdi değerleri için her zaman aynı çıktıyı üretir ve bu işlem sonsuza dek farklı değerler ile tekrarlansa bile fonksiyonun davranışı değişmez, ikinci olarak ise saf fonksiyonlar hiç bir zaman girdinin değerini değiştirmez.

Saf fonksiyonlar fonksiyonel programlama çerçevesinde aşağıdaki yöntemlerin uygulanmasını mümkün kılar

- Örneğin 100 çekirdekli bir işlemciniz varsa 1 ile 100 arasındaki sayıların karelerini aynı anda her bir çekirdekte tanım kümesinden bir elemanı değer kümesinden bir elemana eşleştirecek şekilde **paralel** olarak programlayabilirsiniz. Bu fonksiyonların birinci özelliği sayesinde mümkün olur

- Bir fonksiyonu çıktısına ihtiyaç duyduğunuz anda gevşek olarak (lazy)çalıştırabilirsiniz. Fonksiyonel olmayan programlama dillerinde program akışı bir methoda veya fonksiyona geldiği anda o method veya fonksiyon hemen çalıştırılır ve sonuç alanının bir bellek konumunda saklamanız gerekir. Fonksiyonel programlama dillerinde program akışı bir fonksiyona geldiğinde eğer fonksiyonun sonucuna hemen ihtiyacınız yoksa bu fonksiyonun çalışmasını geciktirebilirsiniz. Buna **gevşek**(lazy) çalıştırma denir. Gevşek çalıştırma da fonksiyonların birinci özelliği sayesinde mümkündür, çünkü bir fonksiyonu ne zaman çalıştırırsanız çalıştırın tanım kümesindeki aynı değeri her zaman değer kümesindeki aynı eleman ile eşleştirir (aynı girdi için her zaman aynı çıktıyı üretir)

- Yine fonksiyonların birinci özelliği sayesinde bir fonksiyonun tanım kümesindeki bir değerin eşleştirildiği değer kümesindeki değeri daha sonra tekrar kullanılmak üzere bellemesini sağlayabilirsiniz. Fonksiyonel programlama dillerinde bu özelliğe **belleme** memoization denir. Belleme davranışı doğrudan fonksiyon tanımında ifade edilebilir ve fonksiyon eğer daha önce bellediği bir eşleştirme işlemini yapacaksa bu işlemi gerçekten yapmadan sonucunu hazır olarak bellekten okuyarak döndürebilir.

- Fonksiyonların ikinci özelliği sayesinde (yan etkisinin olmaması) birden fazla fonksiyonu istediğiniz sıra ile

değerleyebiliriz (evaluate). Fonksiyonlar çalıştırıldığında tanım kümesindeki girdi değeri değişmediği için (girdi değeri bozulmadığı için de diyebiliriz) değer kümesindeki eşleşen değer de değişmez.

Değerleme Sırası Önemli Mi Değil Mi?

Fonksiyonların ikinci özelliğine istinaden fonksiyonları istediğimiz sırada değerleyebileceğimizi ve sonucun değişmeyeceğini söylemiştik. Ancak matematiksel olarak f(g(x)) = g(f(x)) önermesi her zaman doğru değildir. Bu önerme sadece bazı özel f ve g fonksiyonları için doğru olabilir (örneğin birim fonksiyon), bu özel fonksiyonlar dışındaki fonkisyonlar için f(g(x)) ≠ g(f(x)) önermesi geçerlidir.

Fonksiyonların çalıştırma sırasını önemli olduğuğunu aşağıdaki örnek programımızda da hızlıca görebiliriz. Sıralama değiştirildiğinde sonuç da kaçınılmaz olarak değişebilmektedir.

```
(* 01_04.fsx *)
let f(x)  = x + 1 // bir arttırma fonksiyonu tanımı

let g(x) = x * x // kare alma fonksiyonu tanımı

printfn "Sonuç 1 = %d" (f(g(1))) // Sonuç 1 = 2
printfn "Sonuç 2 = %d" (g(f(1))) // Sonuç 2 = 4
```

Ancak fonksiyonel programlama açısından değerleme (evaluate) ve çalıştırma (execute) aynı kavramlar değildir. Değerleme sırası kavramı daha çok derleyici seviyesinde geçerli olan bir kavramdır ve yazdığınız kodun çalıştırılma sırası ile doğrudan bir ilişkisi yoktur. Bu nedenle matematiksel ve programatik olarak yukarıdaki örnekteki

f(g(x)) ve **g(f(x))** çağırıları eş çağırılar değildirler. Bu nedenle fonksiyonel programlamada değerleme sırası önemli olmamakla birlikte çalıştırma sırası diğer tüm programlama yaklaşımlarında olduğu gibi önemlidir.

Şimdi gelelim derleyici açısından değerleme sırasının neden önemli olmadığına. Yime yukarıdaki örneğimizdeki f ve g fonksiyonlarını örnek olarak kullanalım. f(g(1)) ifadesi için iki farklı şekilde değerleme yapılabilir. İlk değerleme (Normal Sıralı Değerleme - Normal Order Evaluation)yaklaşımı şöyle olabilir

```
// Normal Değerleme

f(g(1))
= g(1)+1  // f(x) = x + 1 olduğu için f(x) g(1) +
1 olarak değerlendi
= (1*1)+1 // g(1) -> 1*1 olarak değerlendi
= 1 + 1   // g(1) = 1 olduğu için ifade 1 + 1 olar
ak değerlendi
= 2
```

İkinci değerleme yaklaşımı (Uygun Sıralı Değerleme - Applicative Order Evaluation) ise şöyle olabilir

```
f(g(1))
= f(1*1) // önce g(1) ifadesi değerlendi -> 1*1
= f(1)   // sonuç f(1)
= 1+1    // sonra da f(1) ifadesi değerlendi -> 1
+ 1
= 2      // sonuç
```

Hangi değerleme yaklaşımı uygulanırsa uygulansın **f(g(1))** ifadesinin

sonucu değişmez ve 2'ye eşittir.

> **BİLGİ**
>
> **Normal Sıralı Değerleme (Normal Order)** yapılırken bir fonksiyonun en soldaki görünümü öncelikli olarak değerlenir. f(g(1)) ifadesinde en solda f fonksiyonu var ve f(x) = x + 1 oladuğu için f(g(1)) ifadesi açılarak g(1) + 1 olarak yazılır. Programlama terminolojisinde buna *isimle çağırma - call by name* de denir
>
> **Uygun Sıralı Değerleme (Applicative Order)** yapılırken en içteki fonksiyonun görünümü öncelikli olarak değerlenir. f(g(1)) ifadesinde en içteki fonksiyon g fonksiyonu olduğu için g(1) ifadesi değerlendi (1 * 1 = 1) ve f(g(1)) ifadesi f(1*1) olarak yazıldı. Programlama terminolojisinde buna *değerle çağırma - call by value* de denir

Kullandığınız fonksiyonel programlama dilinin derleyicisi her zaman yukarıdaki değerleme yöntelerinden birini kullanabileceği gibi yazdığınız ifadelere veya derleyicinin çalıştırıldığı donanımın yeteneklerine göre iki değerleme yöntemini de değişimli olarak duruma göre kullanabilir.