

F# ile Fonksiyonel Programlama

İçindekiler

- 1.Bölüm : Giriş
 - 1.1 Kısa F# Tarihçesi
 - 1.2 Neden F#?
 - 1.3 F# Sözdizimine Hızlı Bakış
 - 1.4 Fonksiyonlara Matematiksel Bakış
 - 1.5 Fonksiyonların İlginç Özellikleri
 - 1.6 Fonksiyonel Programlama Nedir?
- 2.Bölüm : Kurulum ve Hazırlık
 - F# Geliştirme Platformu Temel Bileşenleri
 - Windows ve Visual Studio
 - OSX ve Visual Studio for Mac
 - Linux ve Visual Studio Code
 - Merhaba Dünya!
- 3.Bölüm : F# Temelleri
 - Basit Veri Tipleri
 - Karşılaştırma ve Eşitlik
 - Fonksiyonlar
 - Temel Veri Tipleri
 - Kod Organizasyonu
- 4.Bölüm : Fonksiyonel Programlama
 - Fonksiyonlar ve Özellikleri
 - Desen Eşleştirme (Pattern Matching)

- Küme Teorisi ve F# Tipleri
 - Değişkenler Grubu (Tuple)
 - Ayrıcılık Bileşim (Discriminated Union)
 - Kayıt (Record)
- Gevşek Değerlendirme (Lazy Evaluation)
- 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
 - Asenkron Programlama Kütüphanesi
 - Paralel Programlama
 - Paralel Programlama Kütüphanesi
- 9.Bölüm : Örnek Uygulamalar
 - Veritabanı Uygulaması
 - Veri Ayıklama ve Analiz Uygulaması
 - Web Programlama Uygulaması
 - Finansal Uygulama : Kredi Puanı Hesaplayıcı
 - UrhoSharp İle Örnek Oyun

1. Bölüm : Giriş

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

1.1 Kısa F# Tarihçesi

F#, Türkçe **efsharp** olarak telafuz edilen yabancı kaynaklarda da **FSharp** veya **F Sharp** olarak da rastlayabileceğiniz şart odaklı (imperative) ve fonksiyonel programlama (declarative/functional) yaklaşımlarının her ikisini de (multi-paradigm) destekleyen fonksiyonel bir programlama dilidir.

DİKKAT

"Fonksiyonel programlama dili" ifadesindeki **fonksiyonel** ibaresi ilk etapta "çok faydalı", "işe yarayan" benzeri anlamlar çağırırsa da kitapta bu anlamlarda kullanılmamıştır.

"Fonksiyonel programlama" ifadesi programlama dilleri tasarımında matematikteki fonksiyonları ve özelliklerini temel alan bir yaklaşımı ifade eder. Bölümün sonunda bu tanım ayrıntılı olarak ele alınmaktadır.

F# programlama dili Microsoft tarafından tasarlanıp geliştirilen açık kaynak kodlu fonksiyonel bir programlama dilidir. Microsoft'un F# gibi bir dili geliştirmesinin altındaki temel motivasyon Microsoft'un geliştirdiği bir platformu olan .NET Framework'ün 90'lı yılların sonundaki temel tasarım amacına kadar uzanır. Microsoft'un .NET Framework'ünü Java'nın sanal ortamına (JVM) benzetebilirsiniz. .NET Framework farklı programlama dilleri ile geliştirilmiş programların MSIL (Microsoft Intermediate Language) adı verilen ara bir dile derlenmesi sonrasında üretilen kodu çalıştıran sanal bir ortam sunar. .NET Framework'ü hedefleyen herhangi bir dilde geliştirilen ve MSIL'e derlenen programlar .NET Framework'ün desteklediği Windows, Linux veya OSX işletim sistemlerinde çalıştırılabilir. F# da .NET Framework'ü destekleyen dillerden birisidir.

BİLGİ

MSIL, işletim sistemi ve bilgisayar mimarisi bağımsız bir dildir ve .NET Framework'ü hedefleyen programlama dillerinin (C#, VB.NET ve F#) derleyicileri tarafından üretilir, elle kodlama yapılmaz.

BİLGİ

.NET Framework ilk çıktığında sadece Windows işletim

sistemini destekliyordu. Kısa bir süre sonra bağımsız bir grup yazılımcı Linux ve OSX'de de çalışabilen Mono isimli açık kaynak bir .NET Framework geliştirdi. 2015 yılı itibariyle Microsoft Mono'ya kod katkısı sağlamaya başlayarak diğer yandan da Windows, Linux ve OSX'de çalışan .NET Core isimli işletim sistemi bağımsız bir .NET Framework versiyonu geliştirmektedir.

F#'ın Microsoft içindeki yaratıcısı olarak adlandırılan Don Syme F#'ın ortaya çıkışını kendi sözleri ile şöyle anlatmaktadır

.NET platformunun vizyonunda başlangıçtan itibaren birden fazla programlama dilinin desteklenmesi bekleniyordu. 1998 yılında, programlama dilleri ile ilgili araştırma grubumdan 10 kişi ile birlikte Microsoft'a dahil olduğumuz zaman, Project 7 kod adlı projeyi başlatan James Plamondon isimli birisi bizimle irtibata geçti. Project 7, yedi adet akademik ve yedi adet de yazılım sektöründe kullanılan genel amaçlı programlama dilinin .NET'i desteklemesinin sağlanmasını hedefleyen bir projeydi. Project 7 ile Microsoft .NET'in gelecekte farklı programlama dillerini destekleyebilmek için hangi alanlarda ne tür esneklikler sağlaması gerektiğini erken safhada anlamasını sağlayacaktı.

.NET'in Generic'leri üzerinde çalışırken elde ettiğim tecrübey ML benzeri bir fonksiyonel programlama dilinin .NET'i destekleyip desteklemeyeceğini araştırmak için ".NET için Haskell" üzerinde çalışmaya başladım. Bu çalışmada önemli gelişmeler sağlamamıza rağmen Haskell ile .NET'in yapısı arasındaki ciddi uyumsuzluklar nedeni ile bu çalışmayı sonlandırmadan durdurduk.

Don Syme ve ekibi yukarıda da aktardığımız Project 7 kapsamında

Haskell ve ML'in de aralarında bulunduğu bazı fonksiyonel dilleri .NET'e taşıma çalışmalarına başladılar. Çalışma yapılan diller arasında ML basitliği ve .NET ile olan uyumu ile ön plana çıkmaktaydı. Caml ve OCaml dilleri de ML'in varyantları olarak ML'in sadeliğini ve basitliğini bir üst seviyeye taşıyan yapıları barındırmaktaydı. Don Syme ve ekibi o dönem için en popüler ML varyantı olan OCaml'ı .NET'e taşıma çabalarına yoğunlaştılar ve 2005 yılında temelinde OCaml olan F# dilinin ilk versiyonu yayınlandı. Aşağıdaki örnekte F# için verilen faktöriyel hesaplama kodu OCaml ile birebir aynıdır.

```
(* 01_1_01.fsx *)  
  
let rec fact x = if x <= 1 then 1 else x * fact (x  
    - 1);;  
fact 5
```

BİLGİ

OCaml kodunu online olarak <https://try.ocamlpro.com> adresinden deneyebilirsiniz. Deneme yaparken her bir satırın sonuna ;; eklemeyi unutmayın

2017 yılı itibarıyla F# 4.1 versiyonuna ulaşmış arkasında Microsoft gibi dev bir firmanın bulunduğu açık kaynak kodlu fonksiyonel bir programlama dili olarak varlığını sürdürmektedir. .NET Framework'ün çalıştığı platformların çeşitliliği arttıkça F# dilinin ulaştığı kitleler ve farklı alanlardaki popülerliği de artmaktadır.

2017 yılı itibarıyla F# versiyon tarihçesini ve diğer ayrıntıları aşağıdaki çizelgede inceleyebilirsiniz.

1.4 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 xy 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 $X \rightarrow Y$ ile gösterilir.

Yukarıdaki tanımda belirtilen 1. koşuldaki $\forall x \in X$ ifadesini " X kümesinin elemanı olan tüm x değerleri", $\exists y \in Y$ ifadesini ise " Y kümesinin elemanı olan bir y değeri" şeklinde okuyabilirsiniz. \forall ve \exists sembolleri matematikte nicelik/miktar belirten sembollerdir, \forall sembolü **tüm** ve \exists sembolü de **bir** anlamında miktar belirtir. Bu tanımda yer alan diğer iki sembolden \in sembolü bir değer bir kümenin elemanı olduğunu ifade eder, \subset 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.

1.5 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 $f(2) \neq 4$ veya $f(3) \neq 9$ 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_2_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 yukarıdaki $f(x) = x * x$ 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_2_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_2_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
1 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 tekrarlanırsa 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 fonksiyonlar için $f(g(x)) \neq g(f(x))$ önermesi geçerlidir.

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

```
(* 01_2_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 değerlendirme (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 değerlendirme sırası önemli olmamakla birlikte çalıştırma sırası diğer tüm programlama yaklaşımlarında olduğu gibi önemlidir.

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

```
// Normal Değerleme  
  
f(g(1))
```

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

İkinci değerlendirme yaklaşımı (Uygun Sıralı Değerleme – Applicative Order Evaluation) ise şöyle olacaktır

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

Hangi değerlendirme 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 değerlendirilir. $f(g(1))$ ifadesinde en solda f fonksiyonu var ve $f(x) = x + 1$ olduğu 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 değerlendirilir. $f(g(1))$ ifadesinde en içteki fonksiyon g fonksiyonu olduğu için $g(1)$ ifadesi değerlendirildi ($1 * 1 = 1$) 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 değerlendirme yöntemlerinden birini kullanabileceği gibi yazdığınız ifadelerle veya derleyicinin çalıştırıldığı donanımın yeteneklerine göre iki değerlendirme yöntemini de değişimli olarak duruma göre kullanabilir.

Fonksiyonların ilginç iki özelliğine ilave olarak pek de ilginç olmayan iki özelliğinden daha bahsedebiliriz. Bunlar

- Fonksiyonların girdisi olan tanım kümesinden bir elemanını değeri ve çıktısı olan değer kümesindeki bir elemanın değeri değiştirilemez. Buna **değerin değişmezliği (immutability)** denir.
- İkinci olarak fonksiyonların tek bir girdi değerinin ve tek bir çıktı değerinin olmasıdır.

Bu iki özellik ilk başta çok önemli değilmiş hatta biraz da kısıtlayıcıymış gibi görünebilir. Ancak, bu özellikler fonksiyonel programlama dillerinin tasarımını doğrudan etkiler. Örneğin F# (ef şarp – F sharp) programlama dilinde derleyici yazdığınız tüm fonksiyonları tek bir giriş parametresi alan ve tek bir çıktı üreten birer fonksiyon olarak değerler, benzer şekilde F# programlama dilinde varsayılan davranış tanımladığınız değişkenlerin tanımlandığı andaki değerlerinin daha sonra değiştirilmesine izin verilmemesi şeklindedir.

BİLGİ

F# programlama dilinde aslında **değişken (variable)** terimi yerine **değer ifadesi (value expression)** terimi kullanılır. Örneğin aşağıdaki a,b ve pi değer ifadeleri değişken değildir

çünkü değerlerini bir defa tanımlandıktan sonra değiştiremeyiz (*değişmezlik – immutability*)

```
(* 01_2_05.fsx *)

let a = 42
a = 43 // Hata

let b = "F# ile Fonksiyonel Programlama"
b = "F# ile fonksiyonel programlama" // Hata

let pi = 3.14
pi = 3.0 // Hata
```

Ancak F# dilinde dilin yaklaşımı nedeni (multi paradigm bir dil) ile değeri değiştirilebilen (mutable) değer ifadeleri tanımlamak da mümkündür

```
(* 01_2_06.fsx *)

let mutable a = 42
printfn "a = %d" a

a <- 43 // Değer ifadesinin değerini değiştir
printfn "a = %d" a

let mutable b = "F# ile Fonksiyonel Programla
ma"
printfn "b = %s" b

b <- "F# ile fonksiyonel programlama" // Değ
er ifadesinin değerini değiştir
printfn "b = %s" b
```



```
let mutable pi = 3.14
printfn "pi = %f" pi
pi <- 3.0 // Değer ifadesinin değerini değiştirir
printfn "pi = %f" pi
```

1.6 Fonksiyonel Programlama Nedir?

Fonksiyonel programlama, saf fonksiyonları (pure functions) ve değeri sonradan değiştirilemeyen değer ifadelerini (value expressions) kullanarak paylaşılan program durumuna (shared program state) ve yan etkilere (side effect) mahal vermeden yapılan kodlama faaliyetidir. Bazı kaynaklar fonksiyonel programlamayı fonksiyonların birinci sınıf vatandaş (first class citizen) olarak kabul edildiği kodlama faaliyeti olarak da tanımlamaktadır. Fonksiyonel programlama bir araç veya dile bağlı değildir ve bir paradigma (yaklaşım) olarak değerlendirilir. Fonksiyonel olmayan programlama dilleri ile de (eğer dilin yapısı müsait ise) fonksiyonel programlama yaklaşımına ve ilkelerine uygun kod yazmak mümkün olabilir.

Fonksiyonel programlama yaklaşımına göre tasarlanmış programlama dilleri **bildirim odaklı (declarative)** diller sınıfında yer alır. Bildirim odaklı dilleri sınıfının karşıtı olarak ise C, C++, Java, Pascal ve C# gibi **şart odaklı (imperative)** diller yer alır.

NOT

Programlama dilleri sınıflandırılırken bakış açısına bağlı olarak farklı yöntemler uygulamak ve farklı sınıflandırmalar yapmak mümkündür. Bildirim odaklı ve şart odaklı şeklindeki sınıflandırma bunlardan en genel geçer sınıflandırmayı temsil eder. Bunun dışında prosedürel diller, makina dilli, üst seviye diller, görsel diller, domain spesifik diller vs gibi

sınıflandırmalar da yapılabilmektedir.

Şimdi gelin basit bir F# kod parçası ile fonksiyonel programlama dili ile geliştirilen kodun neye benzediğini hızlıca deneyimleyelim

```
(* 01_2_07.fsx *)

let liste = [1..10] // 1 ile 10 arasındaki sayılar
                  1 barındıran liste
let kare x = x * x // Bir sayının karesini alan f
                  onksiyon tanımı

let sonuc = List.map kare liste // List modülü içi
                  ndeki map fonksiyonu
printfn "Sonuç = %A" sonuc
// val sonuc : int list = [1; 4; 9; 16; 25; 36; 49
; 64; 81; 100]
```

Yukarıdaki kod parçasında **list** isimli bir değer ifadesi ve **kare** isimli bir fonksiyon tanımı yapılmaktadır. **List.map kare liste** ifadesi ile de **List** modülü içindeki **map** isimli **yüksek dereceli** fonksiyon birinci parametresi **kare** fonksiyonu ikinci parametresi de **liste** olacak şekilde çalıştırılmaktadır.

Şimdi gelin bu örnek kod parçasındaki bazı satırların fonksiyonel programlama yöntemine uygunluğunu değerlendirelim. Şöyle ki

- **kare** fonksiyonu saf bir fonksiyondur çünkü tanım kümesindeki her bir değer için sonuç olarak her zaman aynı çıktıları üretir. İlave olarak fonksiyon girdi veya çıktının değerini değiştirmez
- **liste** değer ifadesinin değeri 1 ile 10 arasındaki sayılardır ve liste değer ifadesinin içeriği tanımlandığı andan sonra

değiştirilemez

- **List.map** fonksiyonu yüksek dereceli bir fonksiyondur çünkü **kare** fonksiyonunu parametre olarak kabul eder

BİLGİ

Yüksek dereceli fonksiyonlar başka bir fonksiyonu girdi parametresi olarak kabul eden fonksiyonlardır. Yukarıdaki örnekte kullanılan **List.map** fonksiyonu **kare** fonksiyonunu parametre olarak alabildiği için **yüksek dereceli (higher order)** bir fonksiyondur.

Bildirim ve Şart Odaklı Programlama Yaklaşımları

F#, OCaml, Scala, Haskell gibi fonksiyonel programlama dilleri bildirim odaklı (declarative) diller sınıfında yer alan dillerdir. C, C#, Java, Pascal ve Cobol gibi diller ise ana yaklaşımları nedeni ile şart odaklı (imperative) diller sınıfında yer alır. Ancak programlama dillerinin bu iki yaklaşıma göre hangi sınıfta yer aldığı belirlenmesi için çok net kriterler yoktur. Bazı diller (örneğin JavaScript, C# veya Java 8) destekledikleri programlama yapılarına göre her iki sınıfta da yer alabilmektedir. Tüm bu kriter belirsizliği ve karmaşasına rağmen bir programcı olarak bu iki sınıf arasındaki temel farkları bilmeniz hem F# öğrenirken hem de diğer diller ile çalışırken sizin için oldukça faydalı olacaktır.

Şimdi gelin her iki yaklaşımın tanımını yaparak aralarındaki farkları ortaya koyalım.

Şart odaklı programlama dillerinde yazdığınız kod bir işlemin **nasıl (how)** yapılacağını tarif eder. Bu yüzden bu tür dillerin temel yapı taşları **tümcelerdir (sentence)**. Bu tümceler ile adım adım programın hangi işlemi **nasıl** yapması gerektiği tarif edilir ve bilgisayar bu adımları takip ederek programı çalıştırır. Bu sınıftaki dillere

prosedürel diller de denir. Bu tür dillerde adım adım bir tarif söz konusu olduğu için genellikle akış kontrolü için **while** ve **for** gibi döngü yapıları, koşullu dallanma için **if/else** ve **switch** yapıları ve her bir adım sonrasında ulaşılan durumun takip edilmesi ve kayıt altına alınması için de **değişkenler** kullanılır.

Bildirim odaklı programlama dillerinde ise yazdığınız kod bir işlemin nasıl yapılacağına değil işlem sonucunun **ne olacağına(what)** odaklanmıştır. Bu sınıftaki dillere fonksiyonel diller de denir. Bu tür dillerin temel yapı taşı **değer ifadeleridir (expression)** ve bilgisayar programınızdaki bu değer ifadelerini çalıştırarak sonucun üretilmesini sağlar. Bildirim odaklı dillerde akış kontrolü için **öz yinelemeli (recursive) fonksiyonlar**, koşullu dallanma için **yüksek dereceli fonksiyonlar (higher order functions)** ve **match** benzeri yapılar kullanılır. Bildirim odaklı dillerde işlem sonucuna odaklanılır ve önceki adımlarda ulaşılan durumun takip edilmesi için değişkenlere ihtiyaç duyulmaz. Bu nedenle daha önce de değindiğimiz gibi bu dillerde doğrudan değişken tanımlanmasına izin verilmez.

F# ağırlıklı olarak bildirim odaklı fonksiyonel bir dil olmakla birlikte şart odaklı yapıları da desteklediği için gelin şimdi örnekler ile her iki yaklaşım için yazmamız gereken kodunun nasıl görüneceğine bakalım

```
(* 01_2_08.1.fsx *)
(* Şart odaklı (fonksiyonel olmayan) stil *)
let liste = [1..10]

let mutable ikiyeBölünenler = []
let mutable ikiyeBölünmeyenler = []

for d in liste do
```

```
if d % 2 = 0 then
    ikiyeBölünenler <- ikiyeBölünenler @ [d]
else
    ikiyeBölünmeyenler <- ikiyeBölünmeyenler @
[d]
printfn "İkiye bölüneneler = %A" ikiyeBölünenler
printfn "İkiye bölünmeyenler = %A" ikiyeBölünmeyen
ler
```

```
(* 01_2_08.1.fsx *)
(* Bildirim odaklı (fonksiyonel) stil *)
let liste = [1..10]
let ikiyeBolünebilirMi x = x % 2 = 0

let ikiyeBölünenler = liste |> List.filter ikiyeBo
lünebilirMi
printfn "İkiye bölüneneler = %A" ikiyeBölünenler

let ikiyeBölünmeyenler = liste |> List.filter (iki
yeBolünebilirMi >> not)
printfn "İkiye bölünmeyenler = %A" ikiyeBölünmeyen
ler
```

Yukarıdaki kod örneklerini de göz önünde bulundurarak her iki yaklaşım arasındaki temel farkları şöyle ifade edebiliriz

- İki yaklaşımın kodalama stilleri birbirinden farklıdır. Şart odaklı dillerde yapılacak her işlem adım adım belirtilmek durumunda olduğu için genelde yazılması gereken kod miktarı fazla olur. Yukarıdaki örnek kodlarda da göreceğiniz gibi fonksiyonel yaklaşım ile en basit bir programda bile %40 (10 satıra karşılık 6 satır) seviyesinde daha az kod yazılması mümkün

- Şart odaklı dillerde çalıştırılan adımlar sonrasında varılan durumun takip edilmesi için değişkenler kullanılır ve bu değişkenlerin değerleri herhangi bir aşamada değiştirilebilir. Ancak fonksiyonel dillerde değişken kavramı yoktur bunun yerine değer ifadeleri (value expression) kullanılır ve bu ifadelerin değerleri ilk atandıkları andan sonra değiştirilemez.
- Çalıştırma sırası şart odaklı dillerde önemlidir çünkü durum takibi değişkenler ile yapılır ve her adım çalıştırıldıktan sonra bu değişkenlerin değeri değişebilir. Bu nedenle şart odaklı dillerde kodun çalışma sırası önemlidir. Ancak, fonksiyonel dillerde değer ifadelerinin değerleri atandıktan sonra değiştirilemediği için ve fonksiyonel programlar durumsuz oldukları için çalışma sırası önemli değildir. Daha önceki bölümlerde bu sıralamanın derleyici seviyesinde de esnek olarak ayarlandığından örnekler ile bahsetmiştik
- Fonksiyonel dillerde fonksiyonlar birinci sınıf vatandaşlar ve bir fonksiyon başka bir fonksiyonu girdi parametresi olarak alıp çıktı olarak geri döndürebilir. Şart odaklı dillerin bir kısmında da bu mümkündür ancak genel olarak fonksiyonları girdi ve çıktı olarak kullanmak daha fazla kod yazılmasını ve hata kontrollerinin düzgün yapılmasını gerektirir.
- Şart odaklı dillerde akış kontrolü için döngü (for/while), koşullu dallanma (if/else, switch) ve metod tanımları kullanılır, programcılar bu yapıları kullanarak program akışını kontrol altında tutarlar. Fonksiyonel dillerde ise akış kontrolü için genel olarak fonksiyonlar ve öz yinelemeli (recursive) fonksiyonlar kullanılır, bu dillerde akış kontrolü alt seviyede derleyici tarafından en optimum şekilde otomatik oluşturulur.
- Şart odaklı dillerde kullanılan temel veri yapıları değişkenler ve diziler (array) gibi içeriği değiştirilebilen yapılarıdır. Fonksiyonel diller ise genel olarak fonksiyonları ve veri yapıları olarak yığınları (collection) kullanırlar.

BİLGİ

Diziler(array) ve yığınlar(collection) arasındaki temel fark dizilerin boyunun sabit ve değiştirilemez olması buna karşın yığınların boyutunun fiziksel kapasitenin izin verdiği sınırlara kadar büyüebilmesidir. Diziler ve yığınlar hem şart odaklı dillerde hem de fonksiyonel dillerde yer alan veri yapılarıdır, ancak fonksiyonel dillerde yığın kullanımı tavsiye edilen pratiklerden birisidir.

Şart odaklı diller bir çok sektörde yoğun olarak kullanılan ana dillerdir bu nedenle fonksiyonel dillere oranla popülerliği ve üretilen kod miktarı daha fazladır. Ancak, bulut tabanlı sistemlerin ve büyük veri odaklı veri işleme uygulamalarının popüler hale gelmesi ile birlikte F#, Clojure ve Haskell gibi fonksiyonel programlama dilleri de geliştiricilerin ilgisini çekmeye başlamış ve kullanımı gün geçtikçe yaygınlaşmaktadır. Değer ifadelerinin değerlerinin atandıktan sonra değiştirilememesi(immutability) ve fonksiyonların prensip olarak yan etkisinin (side effect) olmaması gibi temel yapısal özellikler bu dillerin paralel ve eş zamanlı işleme kabiliyeti gerektiren büyük veri projelerinde her geçen gün daha fazla tercih edilmesini sağlamaktadır.

Sizler de bulut tabanlı büyük veri işleme uygulamaları veya benzer uygulamalar geliştirmek istiyorsanız F# veya farklı bir fonksiyonel programlama dilini öğrenerek kariyerinize pozitif bir katkı yapabilir, farklı mücadele ve fırsatlara açılan kapıları aralayabilirsiniz.

NOT

Şart odaklı (imperative) ve Bildirim odaklı (declarative,fonksiyonel) dillere ilave olarak nesne tabanlı (object oriented) diller de günümüzde diğerlerinde daha popüler olan üçüncü yaklaşımı temsil etmektedir.

