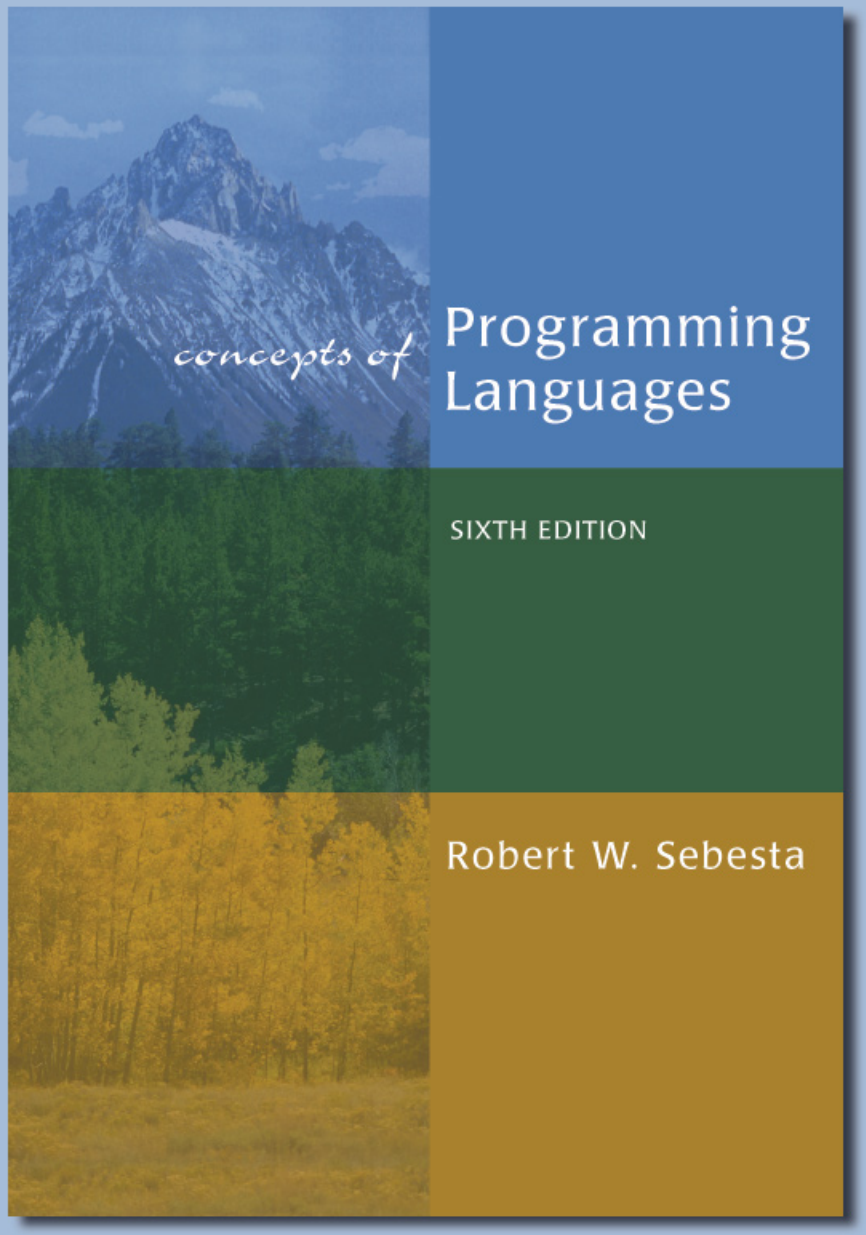
Bölüm 15

Fonksiyonel
Programlama Dilleri



ISBN 0-321-19362-8

PEARSON
Addison
Wesley

Bölüm 15 Topics

- Giriş
- Matematiksel Fonksiyonlar
- Fonksiyonel Programlama Dillerinin (Functional Programming Languages) Temelleri
- İlk Fonksiyonel Programlama Dili: LISP
- Scheme'e Giriş
- COMMON LISP
- ML
- Haskell
- Fonksiyonel dillerin uygulamaları
- Fonksiyonel(Functional) ve Buyurgan(zorunlu-Imperative) Dillerin Karşılaştırılması

Giriş

- Buyurgan(imperative) dillerin tasarımı doğrudan doğruya von Neumann mimarisine dayanır
  - İlgilenilen başlıca konu dilin yazılım geliştirme için uygunluğudan ziyade verimliliğidir

Giriş

- Fonksiyonel dillerin tasarımı Matematiksel Fonksiyonlara dayalıdır
  - Kullanıcıya da yakın olan sağlam bir teorik temel, fakat nispeten programların koşacağı makinelerin mimarileriyle ilgisizdir

Matematiksel Fonksiyonlar

- Tanım: Bir matematiksel fonksiyon, tanım kümesi(domain set) adı verilen bir kümenin(set) üyelerinin (members), değer kümesi(range set) adı verilen diğer bir küme ile eşlenmesidir(mapping)
- Bir lambda ifadesi(lambda expression) bir fonksiyonun parametresini/parametrelerini ve eşlenmesini(mapping) belirler

  $\lambda(x)\ x * x * x$

  cube (x) = x * x * x fonksiyonu için.

Matematiksel Fonksiyonlar

- Lambda ifadeleri adsız fonksiyonları tanımlar
- Lambda ifadelerinin parametreye(lere) uygulanması, parametrenin(lerin) ifadenin sonuna getirilmesiyle olur

  örn. $(\lambda(x)\ x * x * x)(3)$

  sonuç 27

Fonksiyon Biçimleri

1. Fonksiyon Bileşimi(Function Composition)
   - Parametre olarak iki fonksiyon alan ve sonuç olarak, değeri ilk gerçek(actual) parametre fonksiyonun ikincisine uygulanması olan bir fonksiyon veren fonksiyonel form

   Form: $h \equiv f \circ g$

   şu anlama gelir $h(x) \equiv f(g(x))$

   $f(x) \equiv x * x * x$ ve $g(x) \equiv x + 3$ için,

   $h \equiv f \circ g$ şu sonucu verir:

   $(x + 3) * (x + 3) * (x + 3)$

Fonksiyon Biçimleri

2. Yapım(Construction)
   - Parametre olarak fonksiyonlardan oluşan bir liste alan ve sonuç olarak her bir parametre fonksiyonunu verilen bir parametreye uygulama sonuçlarının listesini veren fonksiyonel form

Form: [f, g]

f (x) ≡ x * x * x ve g (x) ≡ x + 3 için,

[f, g] (4) in sonucu: (64, 7) dür

Fonksiyon Biçimleri

3. Tümüne uygula(Apply-to-all)
   - Parametre olarak bir tek fonksiyon alan ve sonuç olarak parametrelerden oluşan bir listenin her bir elemanına verilen fonksiyonun uygulanmasıyla elde edilen değerlerden oluşan bir liste döndüren fonksiyonel form

   Form: α

   $h(x) \equiv x * x * x$ için

   $\alpha(h, (3, 2, 4))$ in sonucu: $(27, 8, 64)$

Fonksiyonel Programlama Dillerinin Temelleri

- Bir FPL tasarlamanın amacı matematiksel fonksiyonları mümkün olduğunca taklit etmektir
- Bir FPL deki temel hesaplama işlemi buyurgan(imperative) dildekinden farklıdır
  - Bir buyurgan(imperative) dilde, işlemler yapılır ve sonuçlar daha sonra kullanım için değişkenlerde(variables) tutulur
  - Sürekli ilgilenilen ve buyurgan(imperative) programlamanın karmaşıklık kaynağı olan şey değişkenlerin(variables) yönetimidir
- Bir FPL de, değişkenler(variables), matematikte olduğu gibi gerekli değildir

Fonksiyonel Programlama Dillerinin Temelleri

- Bir FPL de, bir fonksiyon aynı parametreler verildiğinde daima aynı sonucu üretir
  - Buna referential transparency adı verilir

LISP

- Veri nesnesi tipleri(Data object types): atomlar(atoms) ve listeler(lists)
- Liste biçimi(List form): altliste(sublist) ve/veya atomların paranteze alınmış koleksiyonları

  örn., (A B (C D) E)
- LISP, tipsiz(typeless) bir dildir
- LISP listeleri, tek-bağlı liste(single-linked lists) olarak saklanır

LISP

- Lambda gösterimi fonksiyonları ve fonksiyon tanımlarını belirtmek için kullanılır. Fonksiyon uygulamaları ve veri aynı biçime sahiptir.

  örn., Eğer (A B C) listesi, veri(data) olarak yorumlanırsa, A, B, ve C diye üç atomdan oluşan basit bir listedir

  Eğer bir fonksiyon uygulaması olarak yorumlanırsa, A adındaki fonksiyonun B ve C adındaki iki parametreye uygulanması anlamına gelir

- İlk LISP yorumlayıcısı(interpreter), sadece gösterimin(notation) sayısal hesaplama yeteneklerinin evrenselliğinin ispatı olarak ortaya çıkmıştır

Scheme' e Giriş

- 1970lerin ortalarında çıkmış bir LISP diyalektidir, çağdaş LISP diyalektlerinin daha temiz, daha modern, ve daha basit bir versiyonudur
- Sadece statik kapsama(static scoping) kullanır
- Fonksiyonlar birinci-sınıf varlıklardır(entities)
  - İfadelerin(expressions) değerleri ve listelerin elemanları olabilirler
  - Değişkenlere(variables) atanabilirler ve parametrelere geçilebilirler

Scheme' e Giriş

- İlkel(Primitive) Fonksiyonlar

1. Aritmetik: **`+, -, *, /, ABS, SQRT, REMAINDER, MIN, MAX`**

örn., (+ 5 2) nin sonucu: 7

Scheme' e Giriş

2. **QUOTE** -bir parametreyi alır; onu değerlendirmeden geri döndürür
   - **QUOTE** gereklidir çünkü Scheme yorumlayıcısı(interpreter) olan **EVAL**, her zaman fonksiyonu uygulamadan önce parametreleri fonksiyon uygulamalarında değerlendirir. **QUOTE,** parametreler uygun olmadığı zaman onların değerlendirilmesini önlemek için kullanılır
   - **QUOTE** , kesme işareti(apostrophe) önek(prefix) operatörüyle kısaltılabilir

örn., '(A B) şuna eşittir: (**QUOTE** (A B))

Scheme' e Giriş

3. **`CAR`** bir parametre listesini alır; o listenin ilk elemanını döndürür

   örn., (CAR '(A B C)) sonucu: A

   (CAR '((A B) C D)) sonucu: (A B)

4. **`CDR`** bir parametre listesini alır; ilk elemanını kopardıktan sonra listenin kalanını döndürür

   örn., (CDR '(A B C)) sonucu: (B C)

   (CDR '((A B) C D)) sonucu: (C D)

Scheme' e Giriş

5. **`CONS`** iki parametre alır, bunların birincisi bir atom veya bir liste olabilir ve ikincisi bir listedir; sonuç olarak ilk elemanı olarak birinci parametreyi, sonucunun kalanı olarak da ikinci parametreyi içeren yeni bir liste döndürür

   örn., (CONS 'A '(B C)) sonuç: (A B C)

Scheme' e Giriş

6. **`LIST`** – herhangi bir sayıda parametre alır; elemanları bu parametreler olan bir liste döndürür

Scheme' e Giriş

- Lambda İfadeleri(Expressions)
  - Biçimi λ gösterimine(notation) dayalıdır

  örn., (LAMBDA (L) (CAR (CAR L)))

  L 'ye bağımlı değişken(bound variable) denir
- Lambda ifadeleri şu şekilde uygulanabilir

  örn.,

((LAMBDA (L) (CAR (CAR L))) '((A B) C D))

Scheme' e Giriş

- Fonksiyonları oluşturmak için bir fonksiyon:

  `DEFINE` – İki biçimdedir:

  1. Bir sembolü(symbol) bir ifadeye(expression) bağlama

     örn.,

     (DEFINE pi 3.141593)

     (DEFINE two_pi (* 2 pi))

Scheme' e Giriş

2. Adları(names) lambda ifadelerine bağlama

örn.,

(DEFINE (cube x) (* x x x))

örnek kullanım:

(cube 4)

Scheme' e Giriş

- Değerlendirme işlemi (normal fonksiyonlar için):
  1. Parametreler belli bir sıraya bağlı olmadan değerlendirilir
  2. Parametrelerin değerleri fonksiyon gövdesinde yerine konur
  3. Fonksiyon gövdesi değerlendirilir
  4. Gövdedeki son ifadenin değeri fonksiyonun değeridir

  (Özel biçimler farklı bir değerlendirme işlemi kullanır)

Scheme' e Giriş

- örnekler:

```
(DEFINE (square x) (* x x))

(DEFINE (hypotenuse side1 side1)
    (SQRT (+ (square side1)
    (square side2)))
)
```

Scheme' e Giriş

- Predicate(Hüküm) Fonksiyonlar: (**#T** true ve **()** false)
  1. **EQ?** İki sembolik parametre alır; eğer her ikisi de atomsa ve aynı ise **#T** döndürür

     örn., **(EQ? 'A 'A)** sonuç: **#T**

     (EQ? 'A '(A B)) sonuç: **()**
  - Dikkat edilmelidir ki eğer **EQ?** liste parametrelerle çağrılırsa, sonuç güvenilir olmaz
  - Bir de, **EQ?** sayısal(numeric) atomlar için çalışmaz

Scheme' e Giriş

- Predicate Fonksiyonlar:
  2. **`LIST?`** Bir parametre alır; parametre bir liste ise **`#T`**; değilse **`()`** döndürür
  3. **`NULL?`** Bir parametre alır; parametre bir boş(empty) liste ise **`#T`**; değilse **`()`** döndürür

Dikkat edilmelidir ki **`NULL?`** , eğer parametre **`()`** ise **`#T`** döndürür

4. Sayısal Hüküm(Numeric Predicate) Fonksiyonları

   `=, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?, NEGATIVE?`

Scheme' e Giriş

- Çıktı Yardımcı(Output Utility) Fonksiyonları:

```
(DISPLAY expression)
(NEWLINE)
```

Scheme' e Giriş

- Kontrol akışı

1. Seçim(Selection)- özel biçim, **`IF`**

`(IF predicate then_exp else_exp)`

örn.,

```
(IF (<> count 0)
        (/ sum count)
        0
    )
```

Scheme' e Giriş

- Kontrol akışı
  2. Çoklu Seçim – özel biçim, **COND**

     Genel biçim:

```
(COND
      (predicate_1  expr {expr})
      (predicate_2  expr {expr})
      ...
      (predicate_n  expr {expr})
      (ELSE expr {expr})
   )
```

     En sondaki expr nin değerini hükmü(predicate) true değeri veren birinci çiftte(pair) döndürür

COND Örneği

```
(DEFINE (compare x y)
   (COND
      ((> x y) (DISPLAY "x, y'den
 büyüktür"))
      ((< x y) (DISPLAY "y, x'ten
 büyüktür"))
      (ELSE (DISPLAY "x ve y
 eşittir"))
   )
 )
```

Örnek Scheme Fonksiyonları

1. **member** - bir atom ve bir basit liste alır; eğer atom listede varsa **#T**; yoksa **()** döndürür

```
(DEFINE (member atm lis)
  (COND
    ((NULL? lis) '())
    ((EQ? atm (CAR lis)) #T)
    ((ELSE (member atm (CDR
  lis)))
  ))
```

Example Scheme Fonksiyonlar

2. **`equalsimp`** – parametre olarak iki basit liste alır; iki liste birbirine eşitse **`#T`**; değilde **`()`** döndürür

```
(DEFINE (equalsimp lis1 lis2)
 (COND
  ((NULL? lis1) (NULL? lis2))
  ((NULL? lis2) '())
  ((EQ? (CAR lis1) (CAR lis2))
     (equalsimp(CDR lis1)(CDR lis2)))
  (ELSE '())
))
```

Example Scheme Fonksiyonlar

3. **`equal`** – parametre olarak iki genel liste alır; iki liste birbirine eşitse **`#T`**; değilse **`()`** döndürür

```
(DEFINE (equal lis1 lis2)
 (COND
  ((NOT (LIST? lis1))(EQ? lis1 lis2))
  ((NOT (LIST? lis2)) '())
  ((NULL? lis1) (NULL? lis2))
  ((NULL? lis2) '())
  ((equal (CAR lis1) (CAR lis2))
    (equal (CDR lis1) (CDR lis2)))
  (ELSE '())
))
```

Example Scheme Fonksiyonlar

4. **`append`** - parametre olarak iki liste alır; birinci parametre listesinin sonuna ikinci parametre listesinin elemanlarını ekleyerek geri döndürür

```
(DEFINE (append lis1 lis2)
 (COND
  ((NULL? lis1) lis2)
  (ELSE (CONS (CAR lis1)
        (append (CDR lis1) lis2)))
))
```

Scheme' e Giriş

- **`LET`** fonksiyonu
  - Genel biçim:

```
(LET (
   (name_1 expression_1)
   (name_2 expression_2)
   ...
   (name_n expression_n))
   body
)
```

  - Semantik(Semantics): bütün ifadeleri değerlendir, sonra değerleri(values) adlara(names) bağla; gövdeyi(body) değerlendir

Scheme' e Giriş

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
          (/ (SQRT (- (* b b) (* 4 a c)))
             (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))

  (DISPLAY (+ minus_b_over_2a
              root_part_over_2a))
  (NEWLINE)
  (DISPLAY (- minus_b_over_2a
              root_part_over_2a))
))
```

Scheme' e Giriş

- Fonksiyonel Biçimler
  1. Bileşim(Composition)
     - Önceki örnekler bunu kullandı
  2. Tümüne Uygula(Apply to All) - Scheme de bir biçim **`mapcar`**
     - Verilen fonksiyonu verilen bir listenin bütün elemanlarına uygular; sonuç, sonuçlardan oluşan bir listedir

```
(DEFINE (mapcar fun lis)
    (COND
      ((NULL? lis) '())
      (ELSE (CONS (fun (CAR lis))
                  (mapcar fun (CDR lis))))
  ))
```

Scheme' e Giriş

- Scheme'de, Scheme kodunu oluşturan ve yorumlanmasını(interpretation) isteyen bir fonksiyon tanımlamak mümkündür
- Bu mümkündür çünkü yorumlayıcı(interpreter) kullanıcı tarafından erişilebilen(user-available) bir fonksiyondur, **EVAL**
- örn., farzedelim ki birbiriyle toplanması gereken sayılardan oluşan bir listemiz var

```
(DEFINE (adder lis)
   (COND
      ((NULL? lis) 0)
      (ELSE (+ (CAR lis)
               (adder(CDR lis ))))
))
```

Bir Listedeki Sayıları Toplama

```
((DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis)))
))
```

- Parametre, eklenmesi gereken sayılardan oluşan bir listedir; **`adder`** araya bir + operatörü ekler ve sonuçtaki listeyi yorumlar

Scheme' e Giriş

- Scheme bazı buyurgan(zorunlu-imperative) özellikler içerir:

  1. **`SET!`** bir değeri(value) bir ada(name) bağlar(bind) veya yeniden bağlar(rebind)
  2. **`SET-CAR!`** bir listenin **`car`** 'ını değiştirir
  3. **`SET-CDR!`** bir listenin **`cdr`** kısmını değiştirir

COMMON LISP

- LISP'in popüler diyalektlerine ait çoğu özelliklerin 1980lerin başlarında ortaya çıkmış bir birleşimidir
- Büyük ve karmaşık bir dildir– Scheme'nin tersi

COMMON LISP

- İçeriği:
  - kayıtlar(records)
  - diziler(arrays)
  - karmaşık sayılar(complex numbers)
  - karakter stringleri(character strings)
  - güçlü I/O yetenekleri
  - erişim kontrollü(access control) paketler
  - Scheme'deki gibi buyurgan(imperative) özellikler
  - yinelenen(iterative) kontrol ifadeleri

COMMON LISP

- Örnek (yinelenen küme üyeliği, member (üye))

```
(DEFUN iterative_member (atm lst)
 (PROG ()
  loop_1
   (COND
    ((NULL lst)  (RETURN NIL))
    ((EQUAL atm (CAR lst))(RETURN T))
         )
  (SETQ lst (CDR lst))
  (GO loop_1)
))
```

ML

- Sentaksı LISP'den daha çok Pascala yakın olan bir statik-kapsamlı(static-scoped) fonksiyonel dildir
- Tip tanımlamaları(type declarations) kullanır, fakat aynı zamanda tanımsız(undeclared) değişkenlerin tiplerini belirlemek için tip çıkarma(type inferencing) kullanır (Bölüm 5 de anlatılacak)
- Strongly typed'tır (Scheme temelde tipsizdir(typeless)) ve tip baskısı(type coercions) yoktur
- İstisna yakalama(exception handling) ve soyut veri tipleri(abstract data types) oluşturmak için bir modül özelliği içerir

ML

- Listeler ve liste işlemleri içerir
- **`val`** ifadesi bir adı bir değere(value) bağlar (Scheme'deki **`DEFINE`**'a çok benzer)
- Fonksiyon tanımlama biçimi:

  `fun` fonksiyon_adı (formal_parametreler) =

  fonksiyon_govdesi_ifadesi;

örn.,

```
fun cube (x : int) = x * x * x;
```

Haskell

- ML'e benzer (sentaks, statik kapsamlı(static scoped), strongly typed, type inferencing)
- ML den(ve çoğu diğer fonksiyonel dillerden) farklıdır çünkü tamamen(purely) fonksiyoneldir (örn., değişkenler yoktur, atama ifadeleri yoktur, hiçbir çeşit yan etki yoktur)

Haskell

- En önemli özellikler
  - Tembel değerlendirme(lazy evaluation) kullanır (değer gerekmediği sürece hiçbir alt-ifadeyi değerlendirme)
  - Liste kapsamları(list comprehensions), sonsuz listelerle çalışabilmeye izin verir

Haskell örnekleri

1. Fibonacci sayıları (fonksiyon tanımlarını farklı parametre biçimleriyle gösterir)

```
fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1)
            + fib n
```

Haskell örnekleri

2. Faktoriyel(Factorial) (guardları gösterir)

```
fact n
 | n == 0 = 1
 | n > 0 = n * fact (n - 1)
```

Özel kelime **`otherwise`** guard olarak görünebilir

Haskell örnekleri

3. Liste işlemleri
   - Liste gösterimi(notation): elemanları köşeli parantez içine yaz

   örn., **`directions = [“north”, “south”, “east”, “west”]`**
   - Uzunluk(Length): #

   örn., **`#directions is 4`**
   - .. operatorü ile aritmetik seriler

   örn., **`[2, 4..10] is [2, 4, 6, 8, 10]`**

Haskell örnekleri

3. Liste işlemleri(devam)
   - ++ ile zincirleme(Catenation)

   örn., `[1, 3] ++ [5, 7]` şu sonucu verir:

   `[1, 3, 5, 7]`
   - İki nokta(colon) operatötü yoluyla CONS, CAR, CDR (Prolog'daki gibi)

   örn., `1:[3, 5, 7]` nin sonucu:

   `[1, 3, 5, 7]`

Haskell örnekleri

```
product [] = 1
product (a:x) = a * product x

fact n = product [1..n]
```

Haskell örnekleri

4. Liste kapsamları(comprehensions): küme gösterimi(set notation)

   örn., `[n * n | n ← [1..20]]`

   ilk 20 pozitif tamsayının karelerinden oluşan bir liste tanımlar

```
factors n = [i | i ← [1..n div
2],
                    n mod i == 0]
```

   Bu fonksiyon verilen parametrenin bütün faktörlerini(çarpan) hesaplar

Haskell örnekleri

- Quicksort(Hızlı Sıralama):

```
sort [] = []
sort (a:x) = sort [b | b ← x; b
  <= a]
 ++ [a] ++
sort [b | b ← x; b > a]
```

Haskell örnekleri

5. Tembel değerlendirme(Lazy evaluation)

örn.,

```
positives = [0..]
squares = [n * n | n ← [0..]]
```

(sadece gerekli olanları hesapla)

örn., **`member squares 16`**

True döndürür

Haskell örnekleri

- Üye(member) şu şekilde de yazılabilirdi:

```
member [] b = False
member(a:x) b=(a == b)||member x b
```

- Ancak, bu sadece kare(square) olan parametre tamkare olduğu zaman çalışacaktı; eğer değilse, sonsuza kadar üretmeye devam edecekti. Şu versiyon her zaman çalışır:

```
member2 (m:x) n
        | m < n= member2 x n
        | m == n    = True
        | otherwise = False
```

Fonksiyonel Dillerin uygulamaları

- APL atılan(throw-away) programlar için kullanılır
- LISP yapay zeka(artificial intelligence) için kullanılır
  - Bilgi gösterimi
  - Makine öğrenmesi(Machine learning)
  - Doğal Dil İşleme(Natural language processing)
  - Konuşma ve görmeyi modelleme
- Scheme birçok üniversitede programlamayı öğretmeye giriş için kullanılır

Fonksiyonel ve Buyurgan(imperative) Dillerin Karşılaştırılması

- buyurgan(imperative) diller:
  - Verimli çalışma
  - Karmaşık semantik(semantics)
  - Karmaşık sentaks(syntax)
  - Eşzamanlılık(Concurrency) programcı tarafından tasarlanır
- fonksiyonel diller:
  - Basit semantik(semantics)
  - Basit sentaks(syntax)
  - Verimsiz çalışma
  - Programlar otomatik olarak eşzamanlı(concurrent) yapılabilir