BMB214 Programlama Dilleri Prensipleri

Ders 15. Fonksiyonel Programlama Dilleri (Functional Programming Languages)

Erdinç Uzun
Tekirdağ Namık Kemal Üniversitesi
Bilgisayar Mühendisliği Bölümü

Konular

- Giriş
- Matematiksel Fonksiyonlar
- Fonksiyonel Programlama Dillerinin Temelleri
- İlk Fonksiyonel Programlama Dili: Lisp
- Scheme Giriş
- Common Lisp
- ML
- Haskell
- F#
- Emir Esaslı Fonksiyonel Programlama Desteği
- Fonksiyonel ve Emir Esaslı Dillerin Karşılaştırılması

Giriş

- Emir esaslı dillerin tasarımı doğrudan von Neumann mimarisine dayanmaktadır
  - Verimlilik, dilin yazılım geliştirmeye uygunluğundan ziyade birincil hususdur.
- Fonksiyonel dillerin tasarımı matematiksel fonksiyonlara dayanmaktadır
  - Kullanıcıya daha yakın olan, ancak programların üzerinde çalışacağı makinelerin mimarisiyle nispeten ilgisiz olan sağlam bir teorik temel

Matematiksel Fonksiyonlar

- Bir matematiksel fonksiyon, alan kümesi (domain set) adı verilen bir kümenin üyelerinin aralık kümesi (range set) adı verilen başka bir kümeyle eşleştirilmesidir.
- Bir lambda ifadesi (lambda expression), aşağıdaki biçimde bir fonksiyonun parametrelerini ve eşlemesini belirtir

```
(x) x * x * x
```

Fonksiyon: `cube(x) = x * x * x`

Lambda İfadesi (Lambda Expressions)

- Lambda ifadeleri isimsiz (nameless) fonksiyonlar tanımlar
- Lambda ifadeleri, parametrelerin ifadeden sonra yerleştirilmesiyle parametrelere uygulanır.

  örneğin, `(λ(x) x * x * x)(2)`

  8 olarak değerlendirilir

Fonksiyonel Formlar

- Daha yüksek dereceli bir fonksiyon veya fonksiyonel form, fonksiyonları parametre olarak alan veya sonucu olarak bir fonksiyon veren veya her ikisini birden yapan bir fonksiyondur.

Fonksiyon Birleşimi (Function Composition)

- Parametre olarak iki fonksiyonu alan ve değeri, ikincisinin uygulamasına uygulanan ilk gerçek parametre fonksiyonu olan bir fonksiyon veren fonksiyonel bir form

  Formu: `h ≡ f ° g`

  anlamı `h (x) ≡ f ( g ( x))`

  `f (x) ≡ x + 2` ve `g (x) ≡ 3 * x`, olduğunda

  `h ≡ f ° g` için sonuç `(3 * x)+ 2`

Tümünü Uygula

- Parametre olarak tek bir fonksiyonu alan ve verilen fonksiyonu bir parametre listesinin her bir öğesine uygulayarak elde edilen değerlerin bir listesini veren fonksiyonel bir form

  Formu: α

  `h(x) ≡ x * x`

  `α(h, (2, 3, 4))` sonuç olarak `(4, 9, 16)`

Fonksiyonel Programlama Dillerinin Temelleri

- Bir FPL (Functional Programming Language) tasarımının amacı, matematiksel fonksiyonları mümkün olan en geniş ölçüde taklit etmektir.
- Temel hesaplama süreci, bir FPL'de emir esaslı bir dilden temelde farklıdır.
  - Emir esaslı bir dilde, işlemler yapılır ve sonuçlar daha sonra kullanılmak üzere değişkenlerde saklanır.
  - Değişkenlerin yönetimi, zorunlu programlama için sürekli bir endişe ve karmaşıklık kaynağıdır
- Bir FPL'de, matematikte olduğu gibi değişkenler gerekli değildir
- Referans Şeffaflığı (*Referential Transparency)* - Bir FPL'de, bir fonksiyonun değerlendirilmesi aynı parametreler verildiğinde her zaman aynı sonucu verir

Lisp Veri Türleri ve Yapıları

- Veri nesnesi türleri (*Data object types*): orijinal olarak yalnızca atomlar (atom) ve listeler (list)
- Liste formu: alt listelerin ve / veya atomların parantez içine alınmış koleksiyonları

  örneğin (A B (C D) E)
- Başlangıçta Lisp, tipsiz (typeless) bir dildi
- Lisp listeleri dahili olarak tek bağlantılı listeler (single-linked lists) olarak saklanır

Lisp Yorumlama (Lisp Interpretation)

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

  Örneğin, list (A B C) veri olarak yorumlanırsa,

  A, B ve C olmak üzere üç atomdan oluşan basit bir liste

  Bir fonksiyon uygulaması olarak yorumlanırsa,

  bu, A adlı fonksiyonun B ve C olmak üzere iki parametreye uygulanır

- İlk Lisp yotumlayıcı, yalnızca notasyonun hesaplama yeteneklerinin evrenselliğinin bir göstergesi olarak ortaya çıktı.

Scheme'nın Kökenleri

- Lisp'in çağdaş lehçelerinden daha temiz, daha modern ve daha basit bir versiyon olarak tasarlanmış, 1970'lerin ortalarında bir Lisp lehçesi
- Yalnızca statik kapsam (static scope) kullanır
- Fonksiyonlar birinci sınıf varlıklardır
  - İfadelerin değerleri ve listelerin öğeleri olabilirler
  - Değişkenlere atanabilir, parametre olarak aktarılabilir ve fonksiyonlardan döndürülebilirler

Scheme Yorumlayıcı

- Etkileşimli modda (interactive mode), Scheme yorumlayıcısı sonsuz bir okuma-değerlendirme-yazdırma döngüsüdür (REPL)
  - Bu yorumlayıcı biçimi ayrıca Python ve Ruby tarafından da kullanılır.
- İfadeler, EVAL fonksiyonu tarafından yorumlanır
- Değişmezler (Literals) kendi kendilerine değerlendirir

İlkel Fonksiyon Değerlendirmesi

- Parametreler belirli bir sırayla değerlendirilmez
- Parametrelerin değerleri fonksiyon gövdesine değiştirilir
- Fonksiyon gövdesi değerlendirilir
- Gövdedeki son ifadenin değeri, fonksiyonun değeridir

İlkel Fonksiyonlar ve LAMBDA İfadeleri (Primitive Functions and LAMBDA Expressions)

- İlkel Aritmetik Fonksiyonlar: `+`, `-`, `*`, `/`, `ABS`, `SQRT`, `REMAINDER`, `MIN`, `MAX`
  e.g., `(+ 5 2)` yields `7`

- Lambda Expressions
  Form λ notasyonu dayanır
  e.g., `(LAMBDA (x) (* x x)`
  `x`, bağlı değişken olarak adlandırılır
- Lambda ifadeleri parametrelere uygulanabilir
  e.g., `((LAMBDA (x) (* x x)) 7)`

- `LAMBDA` ifadelerin herhangi bir sayıda parametresi olabilir
  `(LAMBDA (a b x) (+ (* a x x) (* b x)))`

Özel Form Fonkiyonu: DEFINE

- DEFINE - İki form:
- Bir sembolü bir ifadeye bağlamak için

  ör. (DEFINE pi 3.141593)

  Örnek kullanım: (DEFINE two_pi (* 2 pi))

  Bu semboller değişken değildir - Java'nın final bildirimleriyle bağlanan adlar gibidirler.
- Adları lambda ifadelerine bağlamak için (LAMBDA örtüktür)

  ör. (DEFINE (kare x) (* x x))

  Örnek kullanım: (kare 5)
- DEFINE için değerlendirme süreci farklıdır! İlk parametre asla değerlendirilmez. İkinci parametre değerlendirilir ve birinci parametreye bağlanır.

Output Fonksiyonu

- Genellikle gerekli değildir, çünkü yorumlayıcı her zaman en üst düzeyde değerlendirilen bir fonksiyonun sonucunu gösterir (iç içe olamayan)
- Scheme, C'nin printf fonksiyonuna benzer PRINTF'e sahiptir.
- Not: açık girdi ve çıktı, fonksiyonel programlama modelinin parçası değildir, çünkü girdi işlemleri programın durumunu değiştirir ve çıktı işlemleri yan etkilerdir.

Sayısal Dayanak Fonksiyonları (Numeric Predicate Functions)

- `T` (or `#t`) true olsun and `#F` (or `#f`) false olsun (bazen `()` false için kullanılır.)
- `=, <>, >, <, >=, <=`
- `EVEN?, ODD?, ZERO?, NEGATIVE?`
- NOT fonksiyonu, bir Boole ifadesinin mantığını tersine çevirir

Kontrol Akışı (Control Flow)

- Selection – özel formu:
- `IF`

`(IF` predicate then_exp else_exp`)`

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

- `COND` function:

```
(DEFINE (leap? year)
   (COND
     ((ZERO? (MODULO year 400)) #T)
     ((ZERO? (MODULO year 100)) #F)
     (ELSE (ZERO? (MODULO year 4)))
))
```

List Fonksiyonları

- QUOTE - bir parametre alır; parametreyi değerlendirmeden döndürür
  - QUOTE gereklidir çünkü EVAL adlı Scheme yorumlayıcısı, fonksiyonu uygulamadan önce uygulamaları çalıştırmak için parametreleri her zaman değerlendirir. QUOTE, uygun olmadığında parametre değerlendirmesinden kaçınmak için kullanılır
  - QUOTE, kesme işareti ön ek operatörü ile kısaltılabilir

    '(A B), (QUOTE (A B))' ye eşdeğerdir
- CAR, CDR ve CONS (Ders6)

List Fonksiyonları...

- LIST, herhangi bir sayıda parametreden bir liste oluşturmak için bir fonksiyondur

`(LIST 'apple 'orange 'grape)` sonucu:

```
(apple orange grape)
```

Dayanak Fonksiyonları (Predicate Function): EQ?

- EQ? parametre olarak iki ifade alır (genellikle iki atom); her iki parametre de aynı işaretçi değerine sahipse #T; aksi halde #F

  `(EQ? 'A 'A)` sonucu `#T`

  `(EQ? 'A 'B)` sonucu `#F`

  `(EQ? 'A '(A B))` sonucu `#F`

  `(EQ? '(A B) '(A B))` sonucu `#T` veya `#F`

  `(EQ? 3.4 (+ 3 0.4))` sonucu `#T` veya `#F`

Dayanak Fonksiyonları (Predicate Function): EQV?

- EQV? hem sembolik hem de sayısal atomlar için çalışması dışında EQ? gibidir; bu bir değer karşılaştırmasıdır, işaretçi karşılaştırması değildir

  `(EQV? 3 3)` sonucu `#T`

  `(EQV? 'A 3)` sonucu `#F`

  `(EQV 3.4 (+ 3 0.4))` sonucu `#T`

  `(EQV? 3.0 3)` sonucu `#F` (floats ve integers farklıdır)

Dayanak Fonksiyonları (Predicate Function): `LIST?` ve `NULL?`

- LIST? bir parametre alır; parametre bir liste ise #T; aksi halde #F

  (LIST? '()) #T sonucunu verir

- NULL? bir parametre alır; parametre boş bir listeyse #T döndürür; aksi halde #F

  (NULL? '(())) #F sonucunu verir

Örnek Scheme fonksiyonu: member

- member bir atom ve basit bir liste alır; Atom listede ise #T; #F Aksi takdirde

```
DEFINE (member atm a_list)
(COND
   ((NULL? a_list) #F)
   ((EQ? atm (CAR lis)) #T)
   ((ELSE (member atm (CDR a_list)))
))
```

Örnek Scheme fonksiyonu: `equalsimp`

- equalsimp, parametre olarak iki basit listeyi alır; iki basit liste eşitse #T; #F Aksi takdirde

```
(DEFINE (equalsimp list1 list2)
(COND
   ((NULL? list1) (NULL? list2))
   ((NULL? list2) #F)
   ((EQ? (CAR list1) (CAR list2))
         (equalsimp(CDR list1)(CDR
list2)))
   (ELSE #F)
))
```

Örnek Scheme fonksiyonu: `equal`

- equal, iki genel listeyi parametre olarak alır; İki liste eşitse #t döndürür; aksi takdirde #F döndürür.

```
(DEFINE (equal list1 list2)
    (COND
     ((NOT (LIST? list1))(EQ? list1 list2))
     ((NOT (LIST? lis2)) #F)
     ((NULL? list1) (NULL? list2))
     ((NULL? list2) #F)
     ((equal (CAR list1) (CAR list2))
            (equal (CDR list1) (CDR list2)))
     (ELSE #F)
  ))
```

Örnek Scheme fonksiyonu: append

- append, parametre olarak iki listeyi alır; sonuna ikinci parametre listesinin öğeleriyle birlikte ilk parametre listesini döndürür

```
(DEFINE (append list1 list2)
    (COND
     ((NULL? list1) list2)
     (ELSE (CONS (CAR list1)
                 (append (CDR list1) list2)))
    ))
```

Örnek Scheme fonksiyonu: `LET`

- LET konusununu 5. hafta görmüştük.
- LET aslında bir parametreye uygulanan LAMBDA ifadesinin kısaltmasıdır

```
(LET ((alpha 7))(* 5 alpha))
((LAMBDA (alpha) (* 5 alpha)) 7)
İkisi birbirine eşittir.
```

LET Örneği

```
(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)))
   (LIST (+ minus_b_over_2a
   root_part_over_2a))
        (- minus_b_over_2a
   root_part_over_2a))
))
```

Scheme'da Kuyruk Özyinelemesi (Tail Recursion)

- Tanım: Bir fonksiyon, özyinelemeli çağrısı fonksiyondaki son işlemse, kuyruk özyinelemelidir (tail recursive)
- Bir kuyruk özyinelemeli fonksiyonu, bir derleyici tarafından yinelemeyi kullanmak üzere otomatik olarak dönüştürülebilir ve böylece daha hızlı hale getirilebilir
- Scheme dili tanımı, Scheme dil sistemlerinin tüm arka arkaya tekrarlanan fonksiyonları yinelemeyi kullanacak şekilde dönüştürmesini gerektirir

Scheme'da Kuyruk Özyinelemesi (Tail Recursion)...

- Orginal Durum

```
(DEFINE (factorial n)
     (IF (<= n 0)
        1
       (* n (factorial (- n 1)))
))
```

- Tail Recursion Örneği

```
(DEFINE (facthelper n factpartial)
     (IF (<= n 0)
         factpartial
         facthelper((- n 1) (* n factpartial)))
   ))
 DEFINE (factorial n)
     (facthelper n 1))
```

Fonksiyonel Form: Birleştirme (Composition)

- Birleştirme
- h, f ve g'nin bileşimi ise, h (x) = f (g (x))

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
(DEFINE h x) (+ 2 (* 3 x)))
```

- Scheme'de, fonksiyonel birleştirme fonksiyonu oluşturabilir.

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
 ((compose CAR CDR) '((a b) c d)) sonuç c
  (DEFINE (third a_list)
    ((compose CAR (compose CDR CDR)) a_list))
```

Bu fonksiyon CADDR eşdeğerdir.

Fonksiyonel Form: Hepsine Uygulama (Apply-to-All)

- Scheme'daki hepsine uygulama formun bir tanesi map'tir.
  - Verilen fonksiyonu verilen listenin tüm öğelerine uygular

```
(DEFINE (map fun a_list)
        (COND
          ((NULL? a_list) '())
          (ELSE (CONS (fun (CAR a_list))
                                      (map fun (CDR a_list))))
    ))

(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

sonuç `(27 64 8 216)`

Kod Oluşturan Fonksiyonlar

- Scheme'de, Scheme kodunu oluşturan ve yorumlanmasını isteyen bir fonksiyon tanımlamak mümkündür.
- Bu mümkündür çünkü yorumlayıcı, kullanıcı tarafından kullanılabilen bir fonksiyondur, EVAL

Bir Sayı Listesine Ekleme

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

- Parametre, eklenecek sayıların listesidir; adder bir + operatörü ekler ve ortaya çıkan listeyi değerlendirir
  - Atom + 'yı sayılar listesine eklemek için CONS'u kullanın.
  - Değerlendirmeyi önlemek için + işaretinin verildiğinden emin olun
  - Yeni listeyi değerlendirme için EVAL'e gönderin

Common Lisp

- 1980'lerin başlarında Lisp'in popüler lehçelerinin birçok özelliğinin birleşimi
- Büyük ve karmaşık bir dil - Scheme'nin tersi
- Özellikler şunları içerir:
  - records
  - arrays
  - complex numbers
  - character strings
  - güçlü I/O özellikleri
  - Erişim kontrollü paketler
  - iterative control statements

Common Lisp...

- Makrolar (Macros)
  - Etkilerini iki adımda oluşturun:
    - Makroyu genişletin
    - Genişletilmiş makroyu değerlendirin
- Common Lisp'in önceden tanımlanmış fonksiyonlardan bazıları aslında makrolardır
- Kullanıcılar DEFMACRO ile kendi makrolarını tanımlayabilir

Common Lisp...

- Backquote operatörü (`)
- Scheme'nn QUOTE'una benzer, tek fark parametrenin bazı kısımlarının önüne virgül koyarak tırnaksız bırakılabilmektedir.
- `` `(a (* 3 4) c) ``
  - (a (* 3 4) c) olarak değerlendirilir
- `` `(a ,(* 3 4) c) ``
  - (a 12 c) olarak değerlendirilir

Coomon Lisp...

- Okuyucu Makroları (Reader Macros)
  - Lisp uygulamaları, Lisp'i bir kod gösterimine dönüştüren okuyucu (reader) adı verilen bir ön uca sahiptir. Daha sonra makro çağrıları kod gösterimine genişletilir.
  - Bir okuyucu makrosu, okuyucu aşamasında genişleyen özel bir makrodur.
  - Bir okuyucu makrosu, Lisp tanımına genişletilen tek bir karakterin tanımıdır.
  - Bir okuyucu makrosu örneği, QUOTE çağrısına genişleyen kesme işareti karakteridir.
  - Kullanıcılar kendi okuyucu makrolarını bir tür kısaltma olarak tanımlayabilir

Common Lisp...

- Common Lisp bir sembol veri türüne sahiptir (Ruby'ninkine benzer)
  - Ayrılmış kelimeler (reserved words), kendilerini değerlendiren sembollerdir
  - Semboller ya bağlı ya da bağlı değildir
    - Fonksiyon değerlendirilirken parametre sembolleri bağlanır
    - Değerler atanmış emir esaslı stil değişkenlerinin adları olan semboller bağlıdır
    - Diğer tüm semboller bağlı değildir

ML

- Lisp'ten Pascal'a daha yakın olan sözdizimine sahip statik kapsamlı bir fonksiyonel dil
- Tür bildirimlerini kullanır, ancak ayrıca bildirilmemiş değişkenlerin türlerini belirlemek için tür çıkarımı (*type inferencing)* yapar
- Güçlü bir tür yapısına sahiptir (oysa Scheme esasen tür yapısı yoktur) ve tür zorlaması yoktur.
- Emir esaslı stil değişkenleri yok
- Tanımlayıcıları, değerler için türsüz adlardır
- Soyut veri türlerini uygulamak için istisna işleme ve bir modül olanağı içerir
- Listeleri ve liste işlemlerini içerir

ML'e Özgü Özellikler

- Değerlendirme ortamı (*evaluation environment*) adı verilen bir tablo (table), bir programdaki tüm tanımlayıcıların adlarını türleriyle birlikte (bir çalışma zamanı sembol tablosu gibi) depolar.
- Fonksiyon bildirimi formu:

  `fun` *name* (*formal parameters*) = *expression*`;`

  e.g., `fun cube(x : int) = x * x * x;`
  - Tür, dönüş değerine eklenebilir:
    `fun cube(x) : int = x * x * x;`
  - Hiçbir tür belirtilmezse, varsayılan olarak int (sayısal değerler için varsayılan) kullanır
  - Kullanıcı tanımlı aşırı yüklenmiş fonksiyonlara izin verilmez, bu nedenle gerçek parametreler için bir cube fonksiyonu istiyorsak, farklı bir ada sahip olması gerekir

ML'e Özgü Özellikler...

- ML seçim işlemi (selection)

  `if` *expression* **`then`** *then_expression*
  `else` *else_expression*

  burada ilk ifadenin bir Boolean değeri olarak değerlendirilmesi gerekir

- Desen eşleştirme (Pattern matching), bir fonksiyonun farklı parametre formları üzerinde çalışmasına izin vermek için kullanılır

```
fun fact(0) = 1
|   fact(1) = 1
|   fact(n : int) : int = n * fact(n - 1)
```

ML'e Özgü Özellikler...

- List'ler

  Değişmez listeler parantez içinde belirtilir

  `[3, 5, 7]`

  `[]` : boş liste

  `CONS` ikili infix operatörüdür, `::`

  `4 :: [3, 5, 7]`, Sonuç: `[4, 3, 5, 7]`

  `CAR` is the unary operator `hd`

  `CDR` is the unary operator `tl`

```
fun length([]) = 0
|   length(h :: t) = 1 + length(t);

fun append([], lis2) = lis2
|   append(h :: t, lis2) = h :: append(t, lis2);
```

ML'e Özgü Özellikler...

- Val statement bir adı bir değere bağlar (Scheme içindeki DEFINE'a benzer)

```
val distance = time * speed;
```

- DEFINE'da olduğu gibi, val, emir esaslı bir dilde bir atama ifadesi gibi bir şey değildir.
- Aynı tanımlayıcı için iki val statement varsa, ilki ikincisi tarafından gizlenir
- val statement genellikle let yapılarında kullanılır

```
let
  val radius = 2.7
  val pi = 3.14159
in
  pi * radius * radius
end;
```

ML'e Özgü Özellikler...

- filter
  - Listeler için üst düzey bir filtreleme fonksiyonu
  - Bir yüklem fonksiyonu (predicate function) parametresi olarak, genellikle bir lambda ifadesi biçiminde alır
  - Lambda ifadeleri, ayrılmış kelime *fn* dışında fonksiyonlar gibi tanımlanır.

```
filter(fn(x) => x < 100, [25, 1, 711, 50, 100]);
```

Sonuç: `[25, 1, 50]`

ML'e Özgü Özellikler...

- Map
  - Tek bir parametre, bir fonksiyon alan daha yüksek dereceli bir fonksiyondur
  - Parametre fonksiyonu bir listenin her öğesine uygular ve bir sonuç listesi döndürür

```
fun cube x = x * x * x;
val cubeList = map cube;
val newList = cubeList [1, 3, 5];
```

  - newList değeri `[1, 27, 125]`
  - Alternatif: bir lambda ifadesi kullanın
  - `val newList = map (fn x => x * x * x, [1, 3, 5]);`

ML'e Özgü Özellikler...

- Fonksiyon Bileşimi
  - Tekli operatörü kullanma: o
  - `val h = g o f;`

ML'e Özgü Özellikler...

- Currying
- ML fonksiyonları aslında yalnızca bir parametre alır - daha fazla verilirse, parametreleri bir tuple olarak kabul eder (virgül gerekir)
- Currying işlemi, birden fazla parametreye sahip bir fonksiyon, orijinal fonksiyonunun diğer parametrelerini alan bir fonksiyona döndüren bir fonksiyonla değiştirir.
- Birden fazla parametre alan bir ML fonksiyonu, parametrelerde virgül bırakılarak Curried biçimde tanımlanabilir.

```
fun add a b = a + b;
```

- Tek parametreli bir fonksiyon, a parametre olarak b alan bir fonksiyon döndürür. Çağırmak için

```
add 3 5;
```

ML'e Özgü Özellikler...

- Kısmi Değerlendirme (Partial Evaluation)
  - Curried fonksiyonlar, kısmi değerlendirme ile yeni fonksiyonlar oluşturmak için kullanılabilir
  - Kısmi değerlendirme, fonksiyonun en soldaki gerçek parametrelerden biri veya daha fazlası için gerçek parametrelerle değerlendirildiği anlamına gelir.

```
fun add5 x add 5 x;
```

- Gerçek parametre 5'i alır ve add fonksiyonunu 5 ile ilk biçimsel parametresinin değeri olarak değerlendirir. Tek parametresine 5 ekleyen bir fonksiyon döndürür

```
val num = add5 10;
```

Num değeri 15 olur.

Haskell

- ML'ye benzer (syntax, static scoped, strongly typed, type inferencing, pattern matching)
- ML'den (ve diğer birçok fonksiyonel dilden) farklı olarak tamamen fonksiyoneldir (örneğin, değişken ve atama ifadesi yoktur)

Sözdizimi (Syntax)

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

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

Farklı Parametre Aralıklarına Sahip Fonksiyon Tanımları

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


sub n
  | n < 10 = 0
  | n > 100       = 2
  | otherwise     = 1


square x = x * x
```

- Haskell polimorfizmi desteklediğinden, bu herhangi bir sayısal x türü için çalışır.

Haskell
Lists

- Liste gösterimi: Öğeleri parantez içine koyun

```
directions = ["north", "south", "east", "west"]
```

- Length: `#`

```
#directions Sonuç: 4
```

- .. operatörü ile aritmetik seriler

`[2, 4..10]` Sonuç: `[2, 4, 6, 8, 10]`

- `++` ile birleştirme

`[1, 3] ++ [5, 7]` Sonuç: `[1, 3, 5, 7]`

- `:` operatörü ile CONS, CAR, CDR

`1:[3, 5, 7]` Sonuç: `[1, 3, 5, 7]`

Haskell

- Pattern Parametreleri

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

  - Factorial:

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

- List Comprehensions (Ders 6 konusu idi)

```
[n * n * n | n <- [1..50]]
```

Bu örnekteki niteleyici, bir oluşturucu biçimindedir. Bir test şeklinde olabilir

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

Ters işaretler (backticks), fonksiyonun ikili operatör olarak kullanıldığını belirtir

Quicksort
Haskell Gücü

```
sort [] = []
sort (h:t) =
      sort [b | b ← t; b <= h]
    ++ [h] ++
      sort [b | b ← t; b > h]
```

Tembel Değerlendirme (Lazy Evaluation)

- Bir dil, tüm gerçek parametrelerin tam olarak değerlendirilmesini gerektiriyorsa katıdır (*strict)*
- Bir dil, katı bir gereksinime sahip değilse, *nonstrict*'tir.
- Katı olmayan diller (Nonstrict languages) daha verimlidir ve bazı ilginç yeteneklere izin verir - sonsuz listeler (infinite lists)
- Tembel değerlendirme (Lazy evaluation) - Yalnızca gerekli olduğunda değerler hesaplanır. Başka bir deyişle bir değişkenin değerinin, mecbur kalınana kadar hesaplanmadığı programlama dili özelliğidir.
- Pozitif sayılar

```
positives = [0..]
```

- 16'nın kare sayı (square number) olup olmadığını belirleme

```
member [] b = False
member(a:x) b=(a == b)||member x b
squares = [n * n | n ← [0..]]
member squares 16
```

Member Revisited (Yeniden ziyaret)

- Member fonksiyonu şu şekilde yazılabilir:

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

- Ancak, bu yalnızca kareler parametresi tam bir kare ise işe yarar; değilse, sonsuza kadar üretmeye devam edecektir. Aşağıdaki sürüm her zaman çalışacaktır:

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

F#

- ML ve Haskell'in soyundan gelen Ocaml'a dayanmaktadır
- Temelde fonksiyonel bir dildir, ancak emir esaslı özelliklere sahiptir ve OOP'yi destekler
- Tam özellikli bir IDE'ye, kapsamlı bir yardımcı program kitaplığına sahiptir ve diğer .NET dilleriyle birlikte çalışır
- Tuple'lar, listeler, ayrıştırılmış birleşimler (discriminated unions), kayıtlar (records) ve hem değiştirilebilir hem de değiştirilemez dizileri içerir
- Değerleri oluşturucularla ve iterasyon yoluyla oluşturulabilen generic sequence'leri destekler

F#...

- Sequence

```
let x = seq {1..4};;
```

  - Sequence değerlerinin oluşturulması tembeldir (lazy)

```
let y = seq {0..10000000};;
```

    y sonucu `[0; 1; 2; 3;…]`
  - Geçerli adım sayısı (stepsize) 1'dir, herhangi bir sayıda olabilir

```
let seq1 = seq {1..2..7}
```

    `seq1` sonucu `[1; 3; 5; 7]`
  - İterasyon – array ve list için tembel (lazy) değildir

```
let cubes = seq {for i in 1..4 -> (i, i * i * i)};;
```

    cubes sonucu `[(1, 1); (2, 8); (3, 27); (4, 64)]`

- Fonksiyonlar
  - Eğer adlandırılmışsa, let ile tanımlanmışsa; eğer lambda ifadeleri, fun ile tanımlanmışsa
    ```
    (fun a b -> a / b)
    ```
  - Let ile tanımlanan bir isim ile parametresiz bir fonksiyon arasında fark yok
  - Bir fonksiyonun kapsamı girinti ile tanımlanır

```
let f =
        let pi = 3.14159
        let twoPi = 2.0 * pi;;
```

- Fonksiyonlar…
  - Bir fonksiyonu özyinelemeli ise, tanımı rec ayrılmış kelimeyi içermelidir
  - Fonksiyonlardaki adlar kapsam dışı bırakılabilir ve bu da kapsamlarını sona erdirir

```
let x4 =
      let x = x * x
      let x = x * x
```

Fonksiyonun gövdesindeki ilk let, x'in yeni bir versiyonunu yaratır; bu, parametrenin kapsamını sonlandırır; Gövdedeki ikinci let, ikinci x'in kapsamını sonlandıran başka bir x yaratır.

F#...

- Fonksiyonel Operatörler
  - Pipeline (|>)
  - Sol operand değerini çağrının son parametresine (sağ operanda) gönderen bir ikili operatör

```
let myNums = [1; 2; 3; 4; 5]
  let evensTimesFive = myNums
        |> List.filter (fun n -> n % 2 = 0)
        |> List.map (fun n -> 5 * n)
Sonuç: [10; 20]
```

F#...

- Fonksiyonel Operatörler…
  - Birleşim (Composition) (>>)
    - Sol operandı belirli bir parametreye (bir fonksiyon) uygulayan ve ardından fonksiyondan döndürülen sonucu sağ operanda (başka bir fonksiyon) geçiren bir fonksiyon oluşturur.
    - F# ifadesi (f >> g) x, matematiksel ifade g (f (x)) ile eşdeğerdir.
  - Curried Fonksiyonları

```
let add a b = a + b;;
let add5 = add 5;;
```

F#...

- F# Neden İlginç:
  - Önceki fonksiyonel diller üzerine inşa edilmiştir
  - Günümüzde yaygın olarak kullanılan hemen hemen tüm programlama metodolojilerini destekler
  - Yaygın olarak kullanılan diğer dillerle birlikte çalışabilirlik için tasarlanmış ilk fonksiyonel dildir.
  - Piyasaya sürüldüğünde, ayrıntılı ve iyi geliştirilmiş bir IDE'ye ve yardımcı yazılım kitaplığına sahipti.

Emir Esaslı Dillerde Fonksiyonel Programlama Desteği

- Fonksiyonel programlama desteği, zorunlu dillere giderek artıyor
  - Anonim fonksiyonlar (Anonymous functions - lambda expressions)
- JavaScript: fonksiyon adını tanımının dışında bırakır
- C #: i => (i% 2) == 0 (parametrenin çift veya tek olmasına bağlı olarak doğru veya yanlış döndürür)
- Python: lambda a, b: 2 * a - b

Emir Esaslı Dillerde Fonksiyonel Programlama Desteği...

- Python, üst düzey fonksiyonlar filter ve map destekler (genellikle ilk parametreleri olarak lambda ifadelerini kullanır)

```
map(lambda x : x ** 3, [2, 4, 6, 8])
```

Sonuç: `[8, 64, 216, 512]`

- Python kısmi fonksiyon (partial function) uygulamalarını destekler operatörden içe aktarma ekleme

```
from operator import add
add5 = partial (add, 5)
```

(ilk satır add fonksiyon olarak eklenir)

Kullanımı: add5 (15)

Emir Esaslı Dillerde Fonksiyonel Programlama Desteği...

- Ruby Blokları (Ruby Blocks), metotlara gönderilen etkili alt programlardır, bu da metodu daha yüksek dereceli bir alt program yapar
  - Bir blok, lambda ile bir alt program nesnesine dönüştürülebilir

```
times = lambda {|a, b| a * b}
```

Kullanım: `x = times.(3, 4)` (x, `12 olur`)

  - Times ile curry kullanılabilir

```
times5 = times.curry.(5)
```

Kullanım: `x5 = times5.(3)` (x5, `15 olur`)

Fonksiyonel ve Emir Esaslı Dillerin Karşılaştırılması

- Emir Esaslı Diller:
  - Etkili çalıştırma (Efficient execution)
  - Karmaşık anlambilim (Complex semantics)
  - Karmaşık sözdizimi (Complex syntax)
  - Eşzamanlılık (Concurrency) programcı tarafından tasarlanmıştır
- Fonksiyonel Diller:
  - Basit anlambilim (Simple semantics)
  - Basit sözdizimi (Simple syntax)
  - Daha az verimli çalıştırma (Less efficient execution)
  - Programlar otomatik olarak eşzamanlı (concurrent) yapılabilir