CENG 218 Programlama Dilleri

Bölüm 3: Fonksiyonel Programlama

Öğr.Gör. Şevket Umut Çakır

Pamukkale Üniversitesi

Hafta 3

Hedefler

- Fonksiyonel programlama kavramlarını anlamak
- Scheme diline aşina olmak
- ML diline aşina olmak
- Gecikmeli değerlendirmeyi anlamak
- Haskell ile tanışmak
- Fonksiyonel programlamanın matematiğini anlamak

Arka Plan

- Aşağıdakiler dahil birkaç farklı programlama stili vardır:
  - Fonksiyonel programlama
  - Mantıksal programlama
  - Nesne yönelimli programlama
- Her programlama stilini desteklemek için farklı diller gelişti
  - Her dil türü, von Neumann modelinden farklı olan farklı bir hesaplama modeline dayanır.

Arka Plan

- Fonksiyonel programlama:
  - Fonksiyonlar olarak programların tek tip bir görünümünü sağlar
  - Fonksiyonları veri olarak ele alır
  - Yan etkilerin önlenmesini sağlar
- Fonksiyonel programlama dilleri genellikle daha basit semantiğe ve daha basit bir hesaplama modeline sahiptir
  - Hızlı prototipleme, yapay zeka, matematiksel kanıt sistemleri ve mantık uygulamaları için kullanışlıdır

Arka plan

- Yakın zamana kadar, çoğu fonksiyonel dil verimsiz yürütmeden muzdaripti
  - Çoğu, derlemek yerine orijinal olarak yorumlandı
- Bugün, fonksiyonel diller genel programlama için çok caziptir
  - Paralel çalıştırmaya aşırı yardımcı olmaktalar
  - Çok çekirdekli donanım mimarilerinde zorunlu(imperative) dillerden daha verimli olabilir
  - Olgun uygulama kitaplıklarına sahiplerdir

Arka Plan

- Bu avantajlara rağmen, fonksiyonel diller çeşitli nedenlerle ana diller haline gelmemiştir:
  - Programcılar önce zorunlu veya nesne yönelimli dilleri öğrenirler
  - Nesneye yönelik diller, gerçek nesnelerin günlük hayattaki kullanımını yansıtan güçlü düzenleme presiplerine sahiplerdir
- Özyineleme, fonksiyonel soyutlama ve yüksek mertebeden fonksiyonlar gibi fonksiynel yöntemler, birçok programlama dilinin parçası haline gelmiştir

Fonksiyon Olarak Programlar

- Bir program, belirli bir hesaplamanın açıklamasıdır
- "Nasıl" ı görmezden gelirsek ve sonuca veya hesaplamanın "nesine" odaklanırsak, program girdiyi çıktıya dönüştüren sanal bir kara kutu haline gelir
  - Bu nedenle bir program, matematiksel bir fonksiyona esasen eşdeğerdir
- **Fonksiyon**: X değer kümesindeki her bir x ile bir dizi Y değer kümesinden benzersiz bir y ile ilişkilendiren bir kural

Fonksiyon Olarak Programlar

- Matematiksel terminolojide, fonksiyon $y = f(x)$ veya $f : X \to Y$ şeklinde yazılabilir
- f **alanı(domain)**: X kümesi
- f **aralığı(range)**: Y kümesi
- **Bağımsız değişken(Independent variable)**: $f(x)$ 'deki x, X kümesindeki herhangi bir değeri temsil eder
- **Bağımlı değişken(Dependent variable)**: Y kümesinden $y = f(x)$ ile tanımlanan y
- **Kısmi fonksiyon(Partial function)**: X'teki tüm x'ler için f tanımlanmadığında oluşur

Fonksiyon Olarak Programlar

- **Toplam(Total) fonksiyon**: X kümesindeki tüm x'ler için tanımlanan bir fonksiyon
- Programlar, prosedürler ve fonksiyonların tümü, bir fonksiyonun matematiksel konseptiyle temsil edilebilir
  - Program düzeyinde, x girdiyi temsil eder ve y çıktıyı temsil eder
  - Prosedür veya fonksiyon düzeyinde, x parametreleri temsil eder ve y döndürülen değerleri temsil eder

Fonksiyon Olarak Programlar

- **Fonksiyonel tanım(Functional definition)**: bir değerin biçimsel parametreler kullanılarak nasıl hesaplanacağını açıklar
- **Fonksiyonel uygulama(Functional application)**: gerçek parametreleri kullanan tanımlı bir fonksiyona yapılan çağrı veya belirli bir hesaplama için biçimsel parametrelerin varsaydığı değerler
- Matematikte, bir parametre ile bir değişken arasında her zaman net bir ayrım yoktur
  - Bağımsız değişken terimi genellikle parametreler için kullanılır

Fonksiyon Olarak Programlar

- Zorunlu(imperative) programlama ile fonksiyonel programlama arasındaki temel fark, değişken kavramıdır.
  - Matematikte, değişkenler her zaman gerçek değerleri temsil eder
  - Zorunlu programlama dillerinde, değişkenler, değerleri depolayan bellek konumlarına karşılık gelir.
- Atama ifadeleri, bellek konumlarının yeni değerlerle sıfırlanmasına izin verir
  - Matematikte bellek konumu ve atama kavramları yoktur

Fonksiyon Olarak Programlar

- Fonksiyonel programlama, değişken kavramına matematiksel bir yaklaşım getirir
  - Değişkenler değerlere bağlıdır, bellek konumlarına değil
  - Bir değişkenin değeri değiştirilemez, bu da kullanılabilir bir işlem olarak atamayı ortadan kaldırır
- Fonksiyonel programlama dillerinin çoğu, bir çeşit atama kavramını korur
  - Değişkenlere kesinlikle matematiksel bir yaklaşım getiren saf bir fonksiyonel program oluşturmak mümkündür

Fonksiyon Olarak Programlar

- Atama eksikliği döngüleri imkansız kılar
  - Bir döngü, döngü yürütülürken değeri değişen bir kontrol değişkeni gerektirir
  - Döngüler yerine özyineleme kullanılır
- Bir fonksiyonun iç durumu kavramı yoktur
  - Değeri yalnızca bağımsız değişkenlerinin değerlerine (ve muhtemelen yerel olmayan değişkenlere) bağlıdır
- Bir fonksiyonun değeri, argümanlarının değerlendirme sırasına bağlı olamaz
  - Eşzamanlı uygulamalar için bir avantaj

Fonksiyon Olarak Programlar

Örnek: Değerlendirme Sırası

- Aşağıdaki kodun çıktısı nedir?

```
#include <stdio.h>

int topla(int a, int b) {
    return a+b;
}

void main(){
    int x = 5;
    printf("%d\n", topla(x, x+=5));
}
```

```
sonuc=15
ama topla(x+=5,x)olsa da
sonuc = 20 olacak
cunku ilk parametre 2.
metreye bagli ve ilk
parametre 10 oldu
```

Fonksiyon Olarak Programlar

```
void gcd(int u, int v, int* x) {
    int y, t, z;
    z = u; y = v;
    while (y != 0) {
        t = y;
        y = z % y;
        z = t;
    }
    *x = z;
}
```

Şekil: Döngü kullanan zorunlu sürüm

```
int gcd(int u, int v) {
    if (v == 0) return u;
    else return gcd(v, u%v);
}
```

Şekil: Özyinelemeli fonksiyonel sürüm

Fonksiyon Olarak Programlar

- **Referans şeffaflığı(Referential transparency)**: bir fonksiyonun değerinin yalnızca değişkenlerinin (ve yerel olmayan değişkenlerin) değerlerine bağlı olduğu özellik
- Örnekler:
  - `gcd` fonksiyonu referans olarak şeffaftır
  - `rand` fonksiyonu, makinenin durumuna ve kendisine yapılan önceki çağrılara bağlı olduğu için değildir
- Parametre içermeyen, referans olarak şeffaf bir fonksiyon her zaman aynı değeri döndürmelidir
  - Böylece sabitten farklı değildir

Fonksiyon Olarak Programlar

- Referans şeffaflığı ve atama eksikliği, semantiği basitleştirir
- **Değer semantiği(Value semantics)**: adların bellek konumlarıyla değil, yalnızca değerlerle ilişkilendirildiği semantik
- Fonksiyonel programlamada yerel durumun olmaması, onu nesneye yönelik programlamanın zıttı yapar; burada hesaplama, nesnelerin yerel durumunu değiştirerek ilerler
- Fonksiyonel programlamada, fonksiyonlar genel dil nesneleri olmalı ve kendileri değer olarak görülmelidir

Fonksiyon Olarak Programlar

- Fonksiyonel programlamada, fonksiyonlar **birinci sınıf veri değerleridir(first-class data values)**
  - Fonksiyonlar diğer fonksiyonlar tarafından hesaplanabilir
  - Fonksiyonlar, diğer fonksiyonların parametreleri olabilir
- **Bileşim(Composition)**: fonksiyonlar üzerinde temel işlemler
  - Bir fonksiyon iki fonksiyonu parametre olarak alır ve döndürülen değer olarak başka bir fonksiyon üretir
- Matematikte, bileşke operatörü $\circ$ tanımlanmıştır:
  - Eğer $f : X \to Y$ ve $g : Y \to Z$ ise $f \circ g : X \to Z$ aşağıdaki gibi tanımlanır:
    $(g \circ f)(x) = g(f(x))$

Fonksiyon Olarak Programlar

- Fonksiyonel program dillerinin ve fonksiyonel programların nitelikleri:
  - Tüm prosedürler, gelen değerleri (parametreleri) giden değerlerden (sonuçlar) ayıran fonksiyonlardır
  - Tamamen(saf) fonksiyonel programlamada, hiçbir atama yoktur
  - Saf fonksiyonel programlamada döngü yoktur
  - Bir fonksiyonun değeri, değerlendirme sırasına veya yürütme yoluna değil, yalnızca parametrelerine bağlıdır
  - Fonksiyonlar birinci sınıf veri değerleridir

Scheme: Bir Lisp Lehçesi

- **Lisp (LISt Processing)**: modern fonksiyonel dillerin birçok özelliğini içeren ilk dil
  - Lambda hesabına göre
- Dahil olan özellikler:
  - Tek bir genel yapı kullanarak programların ve verilerin tek tip gösterimi: liste
  - Aynı dilde yazılmış bir tercüman kullanarak dilin tanımı (**metacircular interpreter**)
  - Çalışma zamanı sistemi tarafından otomatik bellek yönetimi

Scheme: Bir Lisp Lehçesi

- Lisp için tek bir standart geliştirilmedi ve birçok varyasyon var
- Statik kapsam belirlemeyi ve fonksiyonların daha tekdüze bir şekilde ele alınmasını kullanan iki lehçe standart hale geldi:
  - Common Lisp
  - Scheme

Scheme Unsurları

- Scheme'deki tüm programlar ve veriler ifade olarak kabul edilir
- İki tür ifade:
  - **Atomlar(Atoms)**: bir zorunlu dilin değişmez sabitleri ve tanımlayıcıları gibi
  - **Parantezli ifade**: boşluklarla ayrılmış ve parantezlerle çevrili sıfır veya daha fazla ifadeden oluşan bir dizi
- Sözdizimi, **genişletilmiş Backus-Naur form(extended Backus-Naur form)** gösterimiyle ifade edilir

Scheme Unsurları

Genişletilmiş Backus-Naur formunda kullanılan semboller	
$\rightarrow$	"Şu şekilde tanımlanır" anlamına gelir
|	Bir alternatifi gösterir
{ }	Sıfır veya daha fazla kez görülebilen bir öğeyi çevreler
' '	Değişmez(literal) bir öğeyi çevreler

Scheme Unsurları

- Scheme sözdizimi:

$$expression \rightarrow atom \mid '(' \; expression \; ')'$$
$$atom \rightarrow number \mid string \mid symbol \mid character \mid boolean$$

- Parantezli ifadeler veri olarak görüntülendiğinde, bunlar listeler olarak adlandırılır
- **Değerlendirme kuralı(Evaluation rule)**: Scheme ifadesinin anlamı
- Scheme'deki bir ortam, tanımlayıcıları değerlerle ilişkilendiren bir sembol tablosudur

Scheme Unsurları I

- Şema ifadeleri için standart değerlendirme kuralı:
  - Atomik değişmezler kendilerine değerlendirilir
  - Anahtar sözcükler dışındaki semboller, geçerli ortamda aranan ve orada bulunan değerlerle değiştirilen tanımlayıcılar veya değişkenler olarak kabul edilir
  - Parantezli bir ifade veya liste iki yoldan biriyle değerlendirilir:
    - İlk öğe bir anahtar kelimeyse, ifadenin geri kalanını değerlendirmek için özel bir kural uygulanır
    - Bir anahtar kelimeyle başlayan ifadeye **özel(special) form** denir
    - Aksi takdirde, parantezli ifade bir fonksiyon uygulamasıdır
    - Parantez içindeki her bir ifade yinelemeli(recursively) olarak değerlendirilir
    - İlk ifade, daha sonraki değerlere(ilk ifadenin argümanları) uygulanan bir fonksiyon olarak değerlendirilmelidir
- Şema değerlendirme kuralı, tüm ifadelerin önek(prefix) biçiminde yazılması gerektiğini belirtir

Scheme Unsurları II

- Örnek: (+ 2 3)
  - + bir fonksiyondur ve 5 değerini döndürmek için 2 ve 3 değerlerine uygulanır
- Değerlendirme kuralı ayrıca bir fonksiyonun değerinin (bir nesne olarak) fonksiyona yapılan bir çağrıdan açıkça ayırt edildiğini ima eder
  - Fonksiyon, bir uygulamadaki ilk ifade ile temsil edilir
  - İşlev çağrısı parantez içine alınır
- Değerlendirme kuralı, uygulama sırası değerlendirmesini(applicative order evaluation) temsil eder:
  - Önce tüm alt ifadeler değerlendirilir
  - Karşılık gelen bir ifade ağacı, yapraklardan köke doğru değerlendirilir

Scheme Unsurları

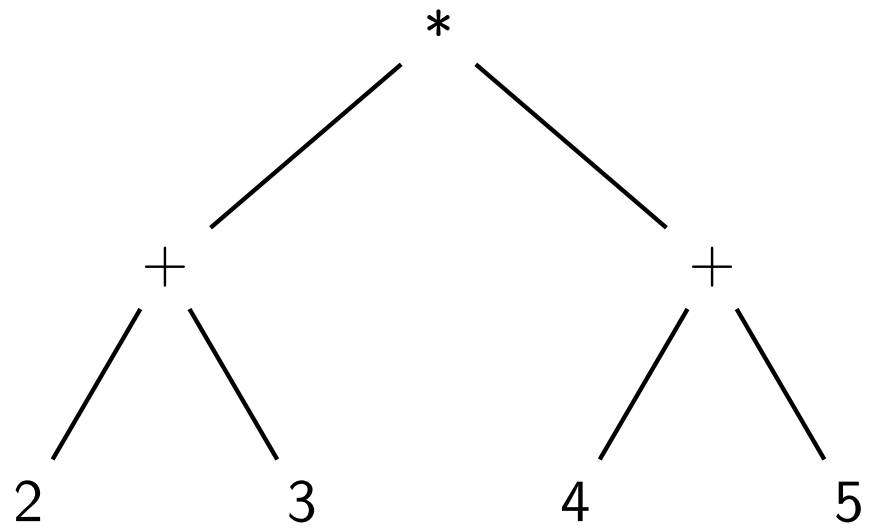
```
3 + 4 * 5
(a == b) && (a != 0)
gcd(10,35)
gcd
getchar()
```

```
(+ 3 (* 4 5))
(and (= a b) (not (= a 0)))
(gcd 10 35)
gcd
(read-char)
```

C ve Scheme'de bazı ifadeler

Scheme Unsurları

- Örnek: `(* (+ 2 3) (+ 4 5))`
  - Önce iki toplama, ardından çarpma işlemi yapılır



Şekil: Scheme ifadesi için ifade ağacı

Scheme Unsurları

- Veriler, doğrudan bir programda temsil edildiğinde bir sorun ortaya çıkar, sayı listesi gibi
- Örnek: (2.1 2.2 3.1)
  - Scheme bunu bir fonksiyon çağrısı olarak değerlendirmeye çalışacak
  - Bu önlenmeli ve bir liste olarak kabul edilmelidir `quote` anahtar kelimesi ile özel bir form kullanarak
- Örnek: `(quote (2.1 2.2 3.1))`
- `quote` özel formunun değerlendirme kuralı, `quote` ifadesinden sonraki ifadeyi değerlendirmeden basitçe döndürmektir.

Scheme Unsurları

- Döngüler özyinelemeli çağrı tarafından sağlanır
- Seçim özel formlarla sağlanır:
  - `if` formu: if-else yapısı gibi
  - `cond` formu: if-elseif yapısı gibi; `cond` koşullu ifade anlamına gelir

```
(if (= a 0) 0          ;eğer a=0 ise 0 döndür
    (/ 1 a))           ;aksi halde 1/a döndür

(cond ((= a 0) 0)      ;eğer a=0 ise 0 döndür
      ((= a 1) 1)      ;eğer a=1 ise 1 döndür
      (else (/ 1 a)))  ;aksi halde 1/a döndür
```

Scheme Unsurları

- Ne `if` ne de `cond` özel formu standart değerlendirme kuralına uymaz
  - Uysalardı, tüm argümanlar her seferinde değerlendirilecek ve onları kontrol mekanizmaları olarak işe yaramaz hale getirecekti
  - Özel formlara yönelik tartışmalar uygun ana kadar **ertelenir(delayed)**
- Scheme fonksiyon uygulamaları değerle geçişi(pass by value) kullanırken, Scheme ve Lisp'teki özel formlar gecikmiş değerlendirmeyi(delayed evaluation) kullanır

Scheme Unsurları

- Özel biçim `let`: bir değişkeni bir ifade içindeki bir değere bağlar
  - Örnek: `(let ((a 2) (b 3)) (+ 1 b))`
    - Bir `let`'teki ilk ifade bir bağlama listesidir
- `let` bir dizi değişken adı için yerel bir ortam ve kapsam sağlar
  - Blok yapılı dillerdeki geçici değişken bildirimlerine benzer
  - Değişkenlerin değerlerine sadece `let` formu içinde erişilebilir, onun dışında değil

Scheme Unsurları

- `lambda` özel formu: belirtilen biçimsel parametrelerle bir fonksiyon ve fonksiyon uygulandığında değerlendirilecek bir kod gövdesi oluşturur
  - Örnek: `(lambda (radius) (* 3.14 (* radius radius)))`
- Fonksiyon ve argüman farklı bir paranteze alınarak fonksiyon parametreye uygulanabilir

```
((lambda (radius) (* 3.14 (* radius radius))) 10)
```

Scheme Unsurları

- Bir `let` içindeki lambda'ya bir isim bağlayabilir:

```
(let ((circlearea (lambda (radius) (* 3.14 (* radius
↪  radius))))) (circlearea 10))
```

- `let` bağlamalar kendilerine veya birbirlerine atıfta bulunamayacağı için özyinelemeli fonksiyonları tanımlamak için kullanılamaz
- `letrec` özel formu: bir `let` gibi çalışır, ancak bağlama listesi içinde keyfi özyinelemeli referanslara izin verir

```
(letrec ((factorial (lambda (n) (if (= n 0) 1 (* n
↪  (factorial (- n 1)))))))) (factorial 10))
```

Scheme Unsurları

- `let` ve `letrec` formları, let veya letrec'in kapsamı ve ömrü dahilinde görülebilen değişkenler oluştur
- `define` özel biçimi: en üst düzey ortamda görünen bir değişkenin genel bir bağlantısını oluşturur

Dinamik Tip Kontrolü

- Scheme dilinin semantiği, dinamik veya gizli tür kontrolünü içerir
  - Sadece değerlerin türü vardır, değişkenler hariç
  - Çalışma zamanında gerekli olana kadar değer türleri kontrol edilmez
- Otomatik tip kontrolü basit bir fonksiyon çağrısının hemen öncesinde kontrol edilir
- Programcı tanımlı fonksiyonlara yönelik bağımsız değişkenler(argümanlar) otomatik olarak kontrol edilmez
- Yanlış türdeyse, Scheme bir hata mesajıyla durur

Dinamik Tip Kontrolü

- `number?` ve `procedure?` gibi yerleşik tür tanıma fonksiyonları değerin tipini kontrol etmek için kullanılır
  - Bu, programcı üretkenliğini ve kodun yürütme hızını yavaşlatır

Kuyruk ve Kuyruksuz Özyineleme

- Prosedür çağrıları için çalışma zamanı ek yükü nedeniyle, zorunlu dillerde döngüler her zaman özyinelemeye tercih edilir
- **Kuyruk özyinelemeli(Tail recursive)**: özyinelemeli adımlar herhangi bir fonksiyondaki son adımlar olduğunda
  - Scheme derleyicisi bunu, üst düzey çağrı dışındaki fonksiyon çağrıları için ek yük olmadan bir döngü olarak yürütülen koda çevirir.
  - Özyinelemenin performans kaybını ortadan kaldırır

Kuyruk ve Kuyruksuz Özyineleme

```
(define factorial
    (lambda (n)
    (if (= n 1)
        1
        (* n (factorial (- n 1))))))

(factorial 6) ;->720
```

Şekil: Kuyruksuz özyinelemeli

```
(define factorial
    (lambda (n result)
        (if (= n 1)
            result
            (factorial (- n 1) (* n result)))))

(factorial 6 1) ;->720
```

Şekil: Kuyruk özyinelemeli

Kuyruk ve Kuyruksuz Özyineleme

- Şekil 39'deki kuyruksuz özyinelemeli(non-tail recursive) fonksiyon örneği:
  - Her özyinelemeli çağrının ardından, çağrının döndürdüğü değer n ile çarpılmalıdır (önceki çağrının argümanı)
  - Özyineleme çözülürken her çağrı için bu bağımsız değişkenin değerini izlemek için bir çalışma zamanı yığıtı(stack) gerektirir
  - Doğrusal bir bellek büyümesi ve önemli bir performans kaybına/azalmasına neden olur

Kuyruk ve Kuyruksuz Özyineleme

- Şekil 39'deki kuyruk özyinelemeli(tail recursive) fonksiyon örneği:
  - Değerleri hesaplamanın tüm işi, her özyinelemeli çağrıdan önce argümanlar değerlendirildiğinde yapılır
  - Argüman sonucu, özyinelemeli çağrılar yoluyla ara ürünleri biriktirmek için kullanılır
  - Her özyinelemeli çağrıdan sonra yapılacak iş kalmaz, bu nedenle önceki çağrıların argümanlarını hatırlamak için çalışma zamanı yığıtı gerekmez

Scheme Veri Yapıları

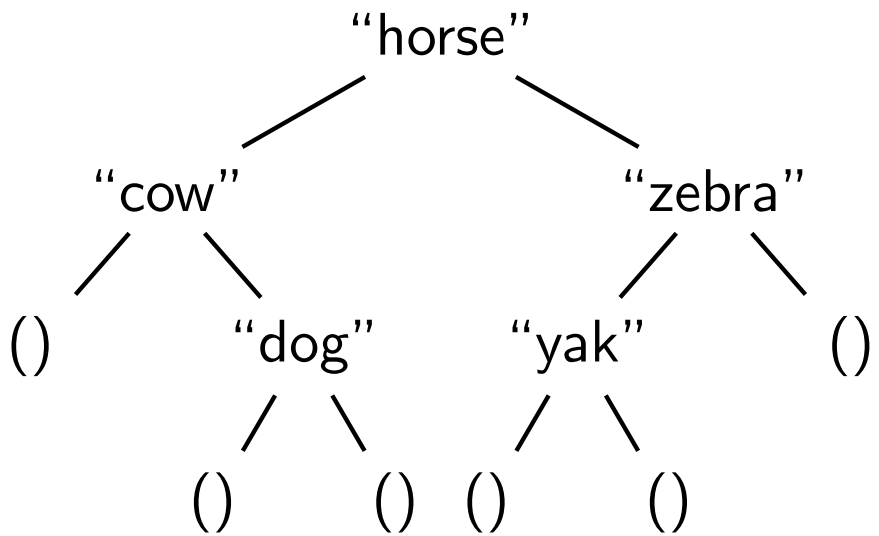
- Scheme'deki temel veri yapısı listedir
  - Bir sekansı/koleksiyonu, kaydı veya başka herhangi bir yapıyı temsil edebilir
- Şema ayrıca vektörler (tek boyutlu diziler) ve dizeler(string) için yapılandırılmış türleri destekler
- Liste fonksiyonları:
  - `car`: listenin başına erişir
  - `cdr`: listenin kuyruğunu döndürür (baş hariç)
  - `cons`: mevcut bir listeye yeni bir başlık(ilk eleman) ekler

Scheme Veri Yapıları

- Örnek: Bir ikili arama ağacının liste temsili

```
("horse" ("cow" () ("dog" () ())) ("zebra" ("yak" () ()) () ))
```

- Bir ağaç düğümü üç elemanlı bir listedir(ad sol sağ)



Şekil: Metinsel veriyi içeren bir ikili arama ağacı

Scheme Veri Yapıları

- Liste bir çift değer olarak görselleştirilebilir: `car` ve `cdr`
  - L listesi, iki işaretçinin bulunduğu kutuya bir işaretçidir; bir tanesi `car`'a(baş), diğeri `cdr`'ye(kuyruk)

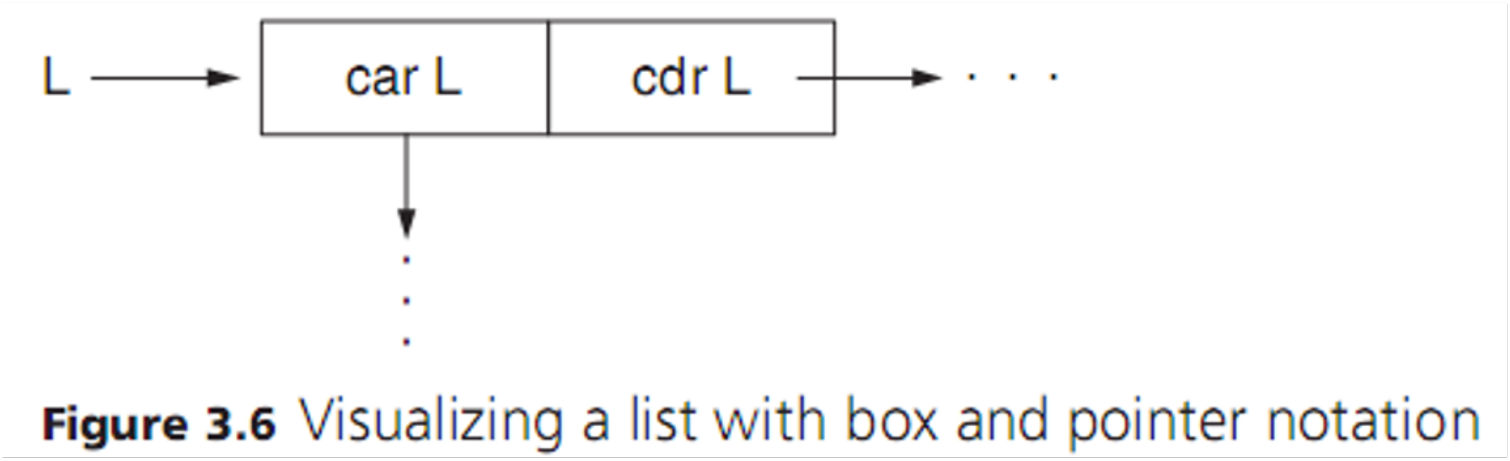


Figure 3.6 Visualizing a list with box and pointer notation

Scheme Veri Yapıları

- Basit bir liste için, `(1 2 3)`, **kutu ve işaretçi gösterimi(box and pointer notation)**

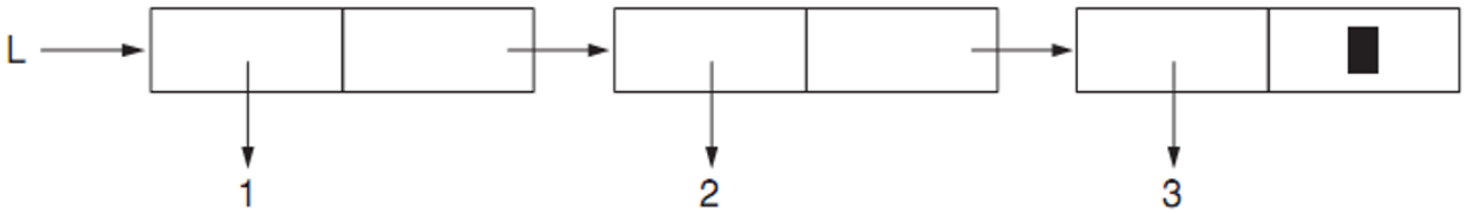


Figure 3.7 Box and pointer notation for the list `(1 2 3)`

Scheme Veri Yapıları

- `((a b) c (d))` için kutu ve işaretçi gösterimi

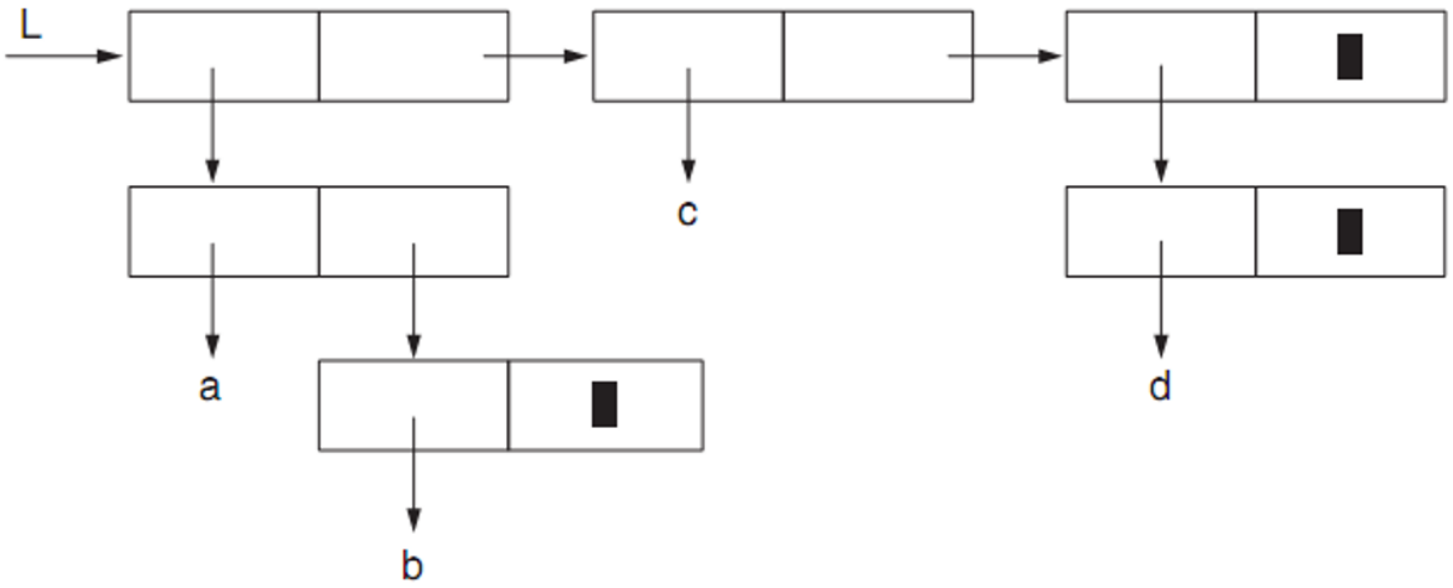


Figure 3.8 Box and pointer notation for the list `L = ((a b) c (d))`

Scheme Veri Yapıları

- Tüm temel liste işleme işlemleri `car`, `cdr`, `cons` ve `null?` kullanılarak fonksiyonlar olarak yazılabilir
  - `null?` liste boşsa doğru(true), aksi halde yanlış(false) döndürür

Scheme Programlama Teknikleri

- Scheme, döngüleri ve diğer tekrarlayan işlemleri gerçekleştirmek için özyinelemeye dayanır
  - Bir listeye tekrarlanan işlemleri uygulamak için, "`cdr` aşağı and `cons` yukarı": işlemi bir listenin kuyruğuna özyinelemeli olarak uygulayın ve ardından mevcut sonuçla yeni bir liste oluşturmak için cons operatörünü kullanın
- Örnek:

```
(define square-list (lambda (L)
        (if (null? L) '()
        (cons
                (* (car L) (car L))
                (square-list (cdr L))))))
```

Yüksek Mertebeden Fonksiyonlar

- **Yüksek mertebeden fonksiyonlar(Higher-order functions)**: diğer fonksiyonları parametre olarak alan fonksiyonlar ve fonksiyonları değer olarak döndüren fonksiyonlar
- Örnek: Bir fonksiyon döndüren, fonksiyon parametreli bir fonksiyon

```
(define make-double (lambda (f) (lambda (x) (f x x))))
```

- Artık bunu kullanarak fonksiyonlar oluşturabilir:

```
(define square (make-double *))
(define double (make-double +))
```

Yüksek Mertebeden Fonksiyonlar

- Fonksiyonel dillerin çalışma zamanı ortamı, standart bir blok yapılı zorunlu dilin yığıt(stack) tabanlı ortamından daha karmaşıktır
- **Çöp toplama(Garbage collection)**: fonksiyonlar tarafından kullanılan belleği döndürmek için otomatik bellek yönetimi tekniği

Statik(Sözcüksel) Kapsam

- Lisp'in erken lehçeleri dinamik kapsamlıydı
- Scheme ve Common Lisp dahil olmak üzere modern lehçeler statik kapsamlıdır
- **Statik (veya sözcüksel) kapsam(static (or lexical) scope)**: bir değişken bildiriminin görünür olduğu bir program alanı
  - Statik kapsam belirleme için, bir değişkenin anlamı veya değeri kaynak kodu okunarak belirlenebilir
  - Dinamik kapsam için anlam, çalışma zamanı bağlamına bağlıdır

Statik(Sözcüksel) Kapsam

- Değişkenlerin bildirimi blok yapılı dillerde yuvalanabilir
- Bir değişkenin kapsamı, herhangi bir iç içe blok dahil olmak üzere bildirildiği bloğun sonuna kadar uzanır (bir iç içe geçmiş blok içinde yeniden bildirilmediği sürece)

```
(let ((a 2) (b 3))
        (let ((a (+ a b)))
                (+ a b)))
>> 8
```

Statik(Sözcüksel) Kapsam

- **Serbest değişken(Free variable)**: bir fonksiyon içinde referans verilen, aynı zamanda bu fonksiyona resmi bir parametre olmayan ve iç içe geçmiş bir fonksiyonla sınırlı olmayan bir değişken
- **Bağlı değişken(Bound variable)**: bir fonksiyonun içindeki değişken, aynı zamanda o fonksiyona yönelik biçimsel bir parametre
- Statik kapsam, serbest değişkenlerin anlamını kodda tek bir yerde düzelterek, bir programı dinamik kapsam belirlemeden daha kolay okunur ve doğrulanır hale getirir

Sembolik Bilgi İşleme ve Dilbilimsel Güç

- **Dilbilimsel güç(Metalingiustic power)**: daha sonra programlar olarak değerlendirilen sembol listelerini oluşturma, kullanma ve dönüştürme kapasitesi
- Örnek: `let` form aslında bir `lambda` formunun argümanlarına uygulanması için sözdizimsel şekerdir(syntactic sugar)

```
(let ((a 3) (b 4)) (* a b))
```

Şekil: let formu

```
((lambda (a b) (* a b)) 3 4)
```

Şekil: lambda formu

Scheme Örnek Uygulamalar

- Aşağıdaki fonksiyonları Scheme dilinde yazınız:
  - 1'den verilen n sayısına kadar olan tek sayıların toplamını bulan fonksiyon
  - Listedeki en küçük değeri bulan fonksiyon
  - Listedeki elemanların toplamını hesaplayan fonksiyon
  - Listedeki çift sayıların kareleri toplamını hesaplayan fonksiyon
  - Listedeki tek sayıları *filtreleyen(kabul eden)* fonksiyon
  - Bir fonksiyon ve listeyi parametre olarak alan ve verilen fonksiyona göre listeyi filtreleyen fonksiyon
  - Bir listedeki elemanların toplamını kuyruk özyinelemeli olarak hesaplayan fonksiyonu yazınız
  - Collatz sanısını kuyruk özyinelemeli olarak bulan fonksiyon(1'e ulaşmak için gereken adım sayısı)

ML: Statik Tip Kontrolü ile Fonksiyonel Programlama

- **ML (veya MetaLanguage)**: Lisp'in lehçelerinden oldukça farklı fonksiyonel bir programlama dili
  - Daha çok Algol benzeri sözdizimine sahiptir, bu da birçok parantezin kullanılmasını önler
  - Statik tip kontrolüne izin verir
- Avantajlar:
  - Yürütmeden önce daha fazla hata bulunduğundan dili daha güvenli hale getirir
  - Çalışma zamanında tip kontrolünü gereksiz hale getirerek verimliliği artırır

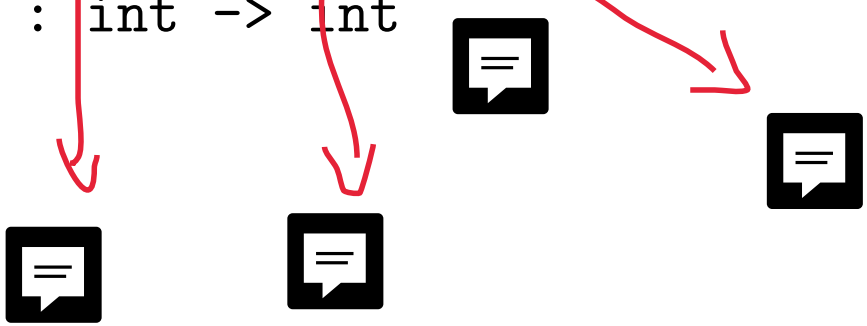
ML: Statik Tip Kontrolü ile Fonksiyonel Programlama

- ML ilk olarak 1970'lerin sonunda programların doğruluğunu kanıtlamak için geliştirildi
  - Edinburgh Logic for Computable Functions (LCF) sisteminin parçası olarak
- Daha sonra HOPE diliyle birleştirildi ve Standard ML veya SML olarak adlandırıldı
- Mevcut standart, 1997'de SML97 veya ML97 olarak adlandırılan başka bir revizyonu yansıtır

ML Unsurları

- ML'de, temel program bir fonksiyon bildirimidir
- `fun`: bir fonksiyon bildirimi tanıtan ayrılmış sözcük
- Elemanların anlamı yalnızca konumlarına göre belirlendiğinden parantezler neredeyse tamamen gereksizdir

```
- fun fact (n : int): int = if n = 0 then 1 else n * fact(n
↪   - 1);
val fact = fn : int -> int
```



ML Unsurları

- Bildirilen bir fonksiyon adıyla çağrılabilir

```
- fact 5;
val it = 120 : int
```

- ML, döndürülen değer ve türüyle yanıt verir
  - `it`, değerlendirilmekte olan mevcut ifadenin adıdır
- Değerler, `val` anahtar sözcüğü kullanılarak tanımlanabilir

```
- val Pi = 3.14159;
val Pi = 3.14159 : real
```

ML Unsurları

- Aritmetik operatörler infix operatörleri olarak yazılır
  - Lisp'in prefix gösteriminden farklı
  - Operatör önceliği(precedence) ve ilişkilendirilebilirlik(associativity) bir sorundur
  - ML aritmetik operatörler için standart matematik kurallarına uyar
- `op` anahtar sözcüğünü kullanarak infix operatörlerini prefix operatörlerine dönüştürebilir:

```
- op + (2, op * (3, 4));
val it = 14 : int
```

ML Unsurları

- İkili aritmetik operatörlerin argüman olarak tamsayı çiftlerini aldığına dikkat edin
  - Çiftler, kartezyen çarpım türünün veya **tuple türünün** öğeleridir

`int * int`

```
- (2, 3);
val it = (2,3) : int * int
- op +;
val it = fn : int * int -> int
```

ML Unsurları

- ML'de programlar Lisp'te olduğu gibi kendileri liste değildir
- ML'de bir liste. virgülle ayrılmış elemanlarla köşeli parantezlerle gösterilir
  - Bir listenin tüm elemanlarının aynı türde olması gerekir

```
- [1, 2, 3];
val it = [1,2,3] : int list
```

- Veri türlerini karıştırmak için bir demet kullanmalısınız:

```
- (1, 2, 3.1);
val it = (1,2,3.1) : int * int * real
```

ML Unsurları

- `::` operatörü Scheme'deki `cons` operatörüne karşılık gelir, bir elemandan(baş) ve önceden oluşturulmuş bir listeden(kuyruk) bir liste oluşturmak için
  - Her liste, `::` operatörünün bir dizi uygulaması tarafından oluşturulur; burada `[]` boş listedir

```
- 1 :: 2 :: 3 :: [];
val it = [1,2,3] : int list
```

- **Tip değişkeni(Type variable)**: `'a` ile gösterilir

```
- op :: ;
val it = fn : 'a * 'a list -> 'a list
```

ML Unsurları

- ML operatörleri `hd` (baş/head için) ve `tl` (kuyruk/tail için) Scheme'nin `car` ve `cdr` operatörlerine karşılık gelir

```
- hd [1, 2, 3];
val it = 1 : int
- tl [1, 2, 3];
val it = [2,3] : int list
```

- ML'nin desen eşleştirme(pattern matching) yeteneği bu fonksiyonları gereksiz kılar
  - Bir listenin başını ve sonunu tanımlamak için `h::t` kullanabilir

ML Unsurları

- Desen eşleştirme, `if` ifadelerinin çoğu kullanımını ortadan kaldırabilir
- Örnek: desen eşleştirmeyi kullanan özyinelemeli faktöryel fonksiyonu:

```
- fun fact 0 = 1 | fact n = n * fact(n -1);
val fact = fn : int -> int
```

- Desenler ayrıca alt çizgi karakteri (_) olarak yazılan joker karakterler içerebilir

```
fun hd (h::_) = h | hd [] = raise Empty;
```

ML Unsurları

- Güçlü tip sistemi(strong typing) nedeniyle veri türleri arasında bir dönüştürme fonksiyonu kullanarak elle dönüştürmeniz gerekmektedir

```
- fun square x: real = x * x;
val square = fn : real -> real
- square (real 2);
val it = 4.0 : real
```

- ML, fonksiyonların aşırı yüklenmesine izin vermez

ML Unsurları

- `rev` fonksiyonu: bir listeyi ters çeviren yerleşik fonksiyon
- ML eşitlik açısından karşılaştırılabilecek türler ile karşılaştırılamayan türler arasında güçlü bir ayrım yapar
  - Gerçek sayılar eşitlik açısından karşılaştırılamaz
- Bir polimorfik fonksiyon tanımı bir eşitlik karşılaştırması içerdiğinde, tür değişkenleri iki tırnakla yazılmış yalnızca eşitlik türleri arasında değişebilir

```
- op =;
val it = fn : ''a * ''a -> bool
```

ML Unsurları

- **Structure**: ML'nin kütüphane paketi sürümü
  - Girdi ve çıktı için yararlı olan birkaç standart önceden tanımlanmış kaynak içerir
  - Örnekler: `TextIO` yapısı, `inputLine` ve `output` fonksiyonları
- ML'deki `unit` türü, C'deki `void` türüne benzer
  - "Gerçek değer olmadığını" temsil eden bir değere () sahiptir
- `ToString` ve `fromString` fonksiyonlarıyla dizeler ve sayılar arasında dönüştürme yapabilir

ML Unsurları

- **İfade dizisi(Expression sequence)**: değeri, listelenen son ifadenin değerine sahip, parantez içine alınmış noktalı virgülle ayrılmış bir ifade dizisi

```
open TextIO;
- fun printstuff () =
          ( output(stdOut, "Hello\n");
            output(stdOut, "World!\n")
          );
val printstuff = fn : unit -> unit
- printstuff ();
Hello
World!
val it = () : unit
```

ML'de Veri Yapıları

- ML numaralandırılmış türler, kayıtlar, demetler ve listeler dahil olmak üzere zengin bir veri türü kümesine sahiptir
- `type` anahtar kelimesi: mevcut bir veri türünün eş anlamlısını verir
- `datatype` anahtar sözcüğü, kullanıcı tanımlı bir veri türü üretir
- **Değer oluşturucular(Value constructors)** (veya **veri oluşturucular(data constructors)**): model olarak kullanılabilecek veri türlerinin oluşturulmasında kullanılan isimler
  - Dikey çubuk, alternatif değerleri belirtmek için kullanılır

ML'de Veri Yapıları

- Değer oluşturucu örneği:

```
- datatype Direction = North | East | South | West;
datatype Direction = East | North | South | West
- fun heading North = 0.0 |
      heading East = 90.0 |
      heading South = 180.0 |
      heading West = 270.0;
val heading = fn : Direction -> real
```

- İkili arama ağacı `datatype` ile bildirilebilir:

```
- datatype 'a BST = Nil | Node of 'a * 'a BST * 'a BST;
```

ML'de Yüksek Mertebeden Fonksiyonlar ve Körileme(Currying)

- `fn` anahtar sözcüğü: bir fonksiyon ifadesini belirtir ve ardından => gelir
  - Anonim fonksiyonlar ve fonksiyon dönüş değerleri oluşturmak için kullanılabilir
  - `fun` tanımı, bir `fn` ifadesinin kullanımı için sadece sözdizimsel şekerdir
- Örnek:
  - `fn square x = x * x`
  - eşdeğerdir
  - `val square = fn x => x * x;`

ML'de Yüksek Mertebeden Fonksiyonlar ve Körileme(Currying)

- `rec` anahtar sözcüğü: `fn` kullanılırken özyinelemeli bir fonksiyon bildirmek için kullanılır
  - Scheme `letrec` fonksiyonuna benzer

```
val rec fact = fn n => if n=0 then 1 else n*fact(n-1);
```

- o harfi ile bileşke fonksiyon oluşturulabilir

```
- fun double x = x+x;
val double = fn : int -> int
- fun square x = x*x;
val square = fn : int -> int
- val double_square = double o square;
val double_square = fn : int -> int
- double_square 3;
val it = 18 : int
```

ML'de Yüksek Mertebeden Fonksiyonlar ve Körileme(Currying)

- **Körileme(Currying)**: Birden fazla parametrenin bir fonksiyonunun, kalan parametrelerin bir fonksiyonunu döndüren tek bir parametrenin daha yüksek mertebeden bir fonksiyonu olarak görüldüğü bir süreç
  - Bu işlemin uygulandığı bir fonksiyonun **körilenmiş(curried)** olduğu söyleniyor
- Matematik ve bilgisayar bilimlerinde körileme(currying), birden fazla argümanı alan bir fonksiyonu, her biri tek bir argüman alan bir dizi fonksiyona dönüştürme tekniğidir.

$$x = f(a, b, c) \text{ aşağıdaki şekilde olur}$$
$$h = g(a)$$
$$i = h(b)$$
$$x = i(c)$$

ML'de Yüksek Mertebeden Fonksiyonlar ve Körileme(Currying)

- Bir fonksiyonun **körilenmemiş(uncurried)** versiyonunu veya körilenmiş versiyonu için iki ayrı parametre elde etmek için bir tuple kullanılabilir
- Fonksiyon tanımları otomatik olarak körilenmiş olarak değerlendirilirse ve tüm çok parametreli yerleşik fonksiyonlar körilenmiş ise bir dilin tamamen curried olduğu söylenir
  - ML tüm yerleşik ikili operatörler tuple alacak şekilde tanımlandığından tam olarak körilenmiş değil

ML Örnek Uygulamalar

- Aşağıdaki işlevleri gerçekleştiren fonksiyonları yazınız:
  - 1 ile parametre olarak verilen n sayısı arasındaki tek sayıların toplamını hesaplayan fonksiyon
  - Parametre olarak verilen listedeki elemanların toplamını döndüren fonksiyon
  - Parametre olarak verilen listenin en küçük elemanını veren fonksiyon
  - Parametre olarak verilen listedeki tek sayıların bulunduğu listeyi döndüren fonksiyon
  - Bir fonksiyon ve bir listeyi parametre olarak alan ve listeyi fonksiyona göre filtreleyen fonksiyon
  - `map` fonksiyonunu kullanan ve parametre olarak verilen listedeki sayıların karesini alıp liste olarak döndüren fonksiyon
  - İkili arama ağacında arama yapan fonksiyon
  - İkili arama ağacına ekleme yapan fonksiyon

Gecikmeli Değerlendirme(Delayed Evaluation)

- Uygun sıralı değerlendirme(applicative order evaluation) sahip bir dilde kullanıcı tanımlı bütün fonksiyonlara verilen parametreler çağrı esansında değerlendirilir
- Uygun sıralı değerlendirmeyi kullanmayan örnekler:
  - `and` ve `or` Boole özel formları
  - `if` özel formu
- Boole ifadelerinin kısa devre değerlendirmesi, ikinci parametreyi değerlendirmeden bir sonuca izin verir

Gecikmeli Değerlendirme

- `if` özel formu için gecikmeli değerlendirme gereklidir
- Örnek: `(if a b c)`
  - `b` ve `c`'nin değerlendirmesi `a`'nın sonucu belli olana kadar geciktirilir; sonra `b` veya `c` değerlendirilir, her ikisi değil
- Standart değerlendirme(fonksiyon uygulamaları) kullanan formlar ile kullanmayanlar(özel formlar) arasında ayrım yapılmalıdır
- Fonksiyonlar için uygun sıralı değerlendirmenin kullanılması, semantik ve gerçekleştirmeyi kolaylaştırır

Gecikmeli Değerlendirme

- **Katı olmayan(Nonstrict)**: Alt ifadeler veya parametreler tanımlanmamış olsa bile, gecikmeli değerlendirmenin iyi tanımlanmış bir sonuca yol açtığı bir fonksiyon özelliği
- Katı olmayan fonksiyonlar istenen bir özellik olsa da, fonksiyonların katı olduğu özelliğe sahip dillerin uygulanması daha kolaydır
- Algol60, isim olarak gönderme(pass by name) parametre geçme kuralı ile gecikmeli yürütme içeriyordu
  - Bir parametre, yalnızca çağrılan bir prosedürün kodunda gerçekten kullanıldığında değerlendirilir

Gecikmeli Değerlendirme

- Örnek: Algol60 gecikmeli yürütme

```
function p(x: boolean; y: integer): integer;
begin
    if x then p := 1
    else p := y;
end
```

- p(true, 1 div 0) olarak çağrıldığında, p kodunda y'ye hiçbir zaman ulaşılmadığı için 1 döndürür
  - Tanımlanmamış ifade 1 div 0 hiçbir zaman hesaplanmaz

Gecikmeli Değerlendirme

- Fonksiyon değerlerine sahip bir dilde, bir parametrenin değerlendirilmesini, onu bir "kabuk" fonksiyonunun (parametresiz bir fonksiyon) içine alarak geciktirmek mümkündür
- C'de isim olarak gönderme(pass by name) eşdeğeri

```
typedef int (*IntProc) ();
int divByZero() {
    return 1/0;
}
int p(int x, IntProc y){
    if (x) return 1;
    else return y();
}
```

Gecikmeli Değerlendirme

- Bu tür "kabuk" prosedürlerine bazen, **thunks** denir
- Scheme ve ML'de, `lambda` ve `fn` fonksiyon değer yapıcıları, parametreleri fonksiyon kabuklarıyla çevrelemek için kullanılabilir
- Örnek:

```
(define (p x y) (if x 1 (y)))
```

- aşağıdaki gibi çağrılabilir

```
(p #T (lambda () (/ 1 0)))
```

Gecikmeli Değerlendirme

- `delay` özel formu: argümanlarının değerlendirilmesini geciktirir ve lambda "kabuğu" gibi bir nesne döndürür veya argümanlarını değerlendirme sözü verir
- `force` özel formu: gecikmiş bir nesne olan parametresinin değerlendirilmesine neden olur
- Önceki fonksiyon artık şu şekilde yazılabilir:

```
(define (p x y) (if x 1 (force y)))
```

- ve aşağıdaki gibi çağrılır

```
(p #T (delay (/ 1 0)))
```

Gecikmeli Değerlendirme

- Gecikmiş değerlendirme, aynı gecikmiş ifade tekrar tekrar değerlendirildiğinde verimsizliğe neden olabilir
- Scheme, geciken nesnenin değerini ilk kez zorlandığında(`force`) saklamak için bir **hafızaya alma(memoization)** işlemi kullanır ve ardından sonraki her `force` çağrısı için bu değeri döndürür
  - Bu bazen **ihtiyaca göre gönderme(pass by need)** olarak adlandırılır

Gecikmeli Değerlendirme

- **Tembel değerlendirme(Lazy evaluation)**: bir ifadeyi yalnızca gerçekten ihtiyaç duyulduğunda değerlendirir
- Bu, fonksiyonel bir dilde, `delay` ve `force`'a yönelik açık çağrılar olmadan gerçekleştirilebilir
- Tembel değerlendirme için gerekli çalışma zamanı kuralları:
  - Kullanıcı tanımlı işlevlere yönelik tüm argümanlar geciktirilir
  - `let` ve `letrec` ifadelerindeki yerel adların tüm bağlantıları geciktirilir
  - Yapıcı fonksiyonlarına yönelik tüm argümanlar geciktirilir
  - Diğer önceden tanımlanmış fonksiyonlara yönelik tüm argümanlar zorunludur
  - Fonksiyon değerli tüm argümanlar zorunludur
  - Seçim formlarındaki tüm koşullar zorunludur(`if` vb.)
- Tembel değerlendirmeye uyan listelere **akışlar(streams)** denebilir
- Tembel değerlendirmeye sahip fonksiyonel bir dilin birincil örneği Haskell'dir

Gecikmeli Değerlendirme

- **Üretici filtreli programlama(Generator-filter programming)**: hesaplamanın akışları oluşturan prosedürler ve akışları argüman olarak alan diğer prosedürlere ayrıldığı bir fonksiyonel programlama tarzı
- **Üreteçler(Generators)**: akışları oluşturan prosedürler
- **Filtreler(Filters)**: akışları değiştiren prosedürler
- Listeler için **aynı sınır(same fringe)** problemi: aynı sırada aynı boş olmayan atomları içeren iki liste aynı sınıra sahiptir

Gecikmeli Değerlendirme

- Örnek: bu listelerin sınırları aynıdır

  `((2 (3)) 4)` ve `(2 (3 4 ()))`
- İki listenin aynı sınıra(same fringe) sahip olup olmadığını belirlemek için, yalnızca atomlarının listelerine göre **düzleştirilmelidir(flatten)**
- düzleştirme(`flatten`) fonksiyonu: bir filtre olarak görülebilir; bir listeyi atomlarının bir listesine indirger
- Tembel değerlendirme, öğeleri uyuşmadan önce gerektiği kadar düzleştirilmiş listeleri hesaplayacaktır

Gecikmeli Değerlendirme

- Gecikmeli değerlendirme, semantiği karmaşıklaştırır ve çalışma zamanı ortamında karmaşıklığı artırır
  - Gecikmeli değerlendirme, bir tür paralellik olarak tanımlanmıştır; `delay` bir süreç askıya alma biçimi ve bir tür sürecin devamı olarak `force`
- Yan etkiler, özellikle atama işlemi, tembel değerlendirmeyle pek iyi uyuşmaz

Haskell — Aşırı Yüklemeyle Tamamen Körilenmiş Tembel Bir Dil

- **Haskell**: 1980'lerin sonunda geliştirilmiş saf(purely) fonksiyonel bir dil
- Tamamen fonksiyonel bir dizi tembel dil üzerine kurulur ve bunları genişletir
- Fonksiyon aşırı yükleme ve G/Ç gibi yan etkilerle başa çıkmak için `monad` adı verilen bir mekanizma dahil olmak üzere bir dizi yeni özellik içerir

Haskell Unsurları

- Haskell'in sözdizimi ML'ninkine çok benzer
  - Belirsizlikleri çözmek için girinti ve çizgi biçimlendirmeye sahip bir **düzen kuralı(layout rule)** kullanır
- ML'den farklılıkları:
  - Önceden tanımlanmış herhangi bir fonksiyonu yeniden tanımlayamaz
  - `cons` operatörü tek iki nokta üst üste olarak yazılır(`:`)
  - Türler, çift iki nokta üst üste kullanılarak verilmiştir(`f::Int->Int`)
  - Desen eşleştirme(pattern matching) `.` sembolünün kullanımına ihtiyaç duymaz
  - Liste birleştirme `++` operatörü ile gerçekleştirilir

Haskell Unsurları

- Haskell, önceden tanımlanmış tüm operatörlerin körilenmiş(curried) olduğu, tamamen körilenmiş bir dildir
- **Bölüm yapısı(Section construct)**: bir ikili operatörün parantezler kullanılarak her iki bağımsız değişkene de kısmen uygulanmasına izin verir
- Örnekler:
  - `plus2 = (2 +)`, soldaki bağımsız değişkenine 2 ekleyen bir fonksiyonu tanımlar
  - `times3 = (* 3)`, sağdaki bağımsız değişkeninin 3 katını çarpan bir fonksiyonu tanımlar

Haskell Unsurları

- Infix fonksiyonları, parantez içine alınarak prefix fonksiyonlarına dönüştürülebilir

```
Prelude> (+) 2 3
5
Prelude> (*) 4 5
20
```

- Haskell, lambda'yı temsil eden ters eğik çizgi ile anonim fonksiyonlara veya lambda formlarına sahiptir

```
Prelude> (\x -> x * x) 3
9
```

Yüksek Mertebeden Fonksiyonlar ve Liste Üreteçleri

- Haskell, tümü körilenmiş biçimde olan `map` gibi önceden tanımlanmış birçok üst düzey fonksiyonu içerir.
- Yerleşik listeler ve demetler(tuple), tür eşanlamlıları ve kullanıcı tanımlı polimorfik türler vardır

```
type ListFn a = [a] -> [a]
type Salary = Float
type Pair a b = (a, b)
data BST a = Nil | Node a (BST a) (BST a)
```

Yüksek Mertebeden Fonksiyonlar ve Liste Üreteçleri

- Tür değişkenleri ML'deki alıntı(`'a`) olmadan yazılır ve veri türü adından önce değil, sonra yazılır
- `data` anahtar kelimesi, ML'nin `datatype` anahtar kelimesinin yerini alır
- Tür ve yapıcı adları büyük harf olmalı, fonksiyon ve değer adları ise küçük olmalıdır
- Yeni veri türündeki fonksiyonlar, ML'de olduğu gibi veri yapıcılarını desen olarak kullanabilir

```
data BST a = Nil | Node a (BST a) (BST a)
flatten :: BST a -> [a]
flatten Nil = []
flatten (Node val left right) = flatten left ++ [val]
↪   ++ flatten right
```

```
*Main> flatten (Node 4 (Node 3 Nil Nil) (Node 7 Nil Nil))
[3,4,7]
```

Yüksek Mertebeden Fonksiyonlar ve Liste Üreteçleri

- **Liste üreteçleri(List comprehensions)**: listelere uygulanan işlemler için özel bir gösterim
- Örnek: bir tam sayılar listesinin karesini alma

```
square_list lis = [x * x | x <- lis]
square_list_positive lis = [x * x | x <- lis, x>0]
```

Tembel Değerlendirme ve Sonsuz Listeler

- Haskell tembel bir dildir — gerçekten gerekli olmadıkça hiçbir değer hesaplanmaz
  - Haskell'deki listeler akışlarla aynıdır ve potansiyel olarak sonsuz olabilir
- Haskell, sonsuz listeler için birkaç kısaltma işaretine sahiptir, örneğin `[n..]`, `n` ile başlayan tamsayıların listesi anlamına gelir
- `take` fonksiyonu: bir listeden ilk n öğeyi alır
- `drop` fonksiyonu: listedeki ilk n öğeyi atar

Tip Sınıfları ve Aşırı Yüklenmiş Fonksiyonlar

- Haskell, fonksiyonların aşırı yüklenmesine izin verir
- **Tip sınıfı(Type class)**:
  - Tümü belirli işlevleri tanımlayan bir dizi tür
  - Kendisine ait her tipin tanımlaması gereken fonksiyonların isimlerini ve tiplerini (imza denilir) belirtir
  - Java arayüzlerine(interface) benzer

```
class Num a where
        (+), (-), (*)   :: a -> a -> a
        negate          :: a -> a
        abs             :: a -> a
```

Tip Sınıfları ve Aşırı Yüklenmiş Fonksiyonlar

- **Örnek tanımı(Instance definition)**: gerekli işlevlerin her biri için fiili çalışma tanımlarını içerir

```
instance Num Int where
    (+)           = primPlusInt
    (-)           = primMinusInt
    negate        = primNegInt
    (*)           = primMulInt
    abs           = absReal
```

- Birçok tür sınıfı, diğer tür sınıflarının parçası olacak şekilde tanımlanır
  - Bu bağımlılığa tür sınıfı mirası denir

Tip Sınıfları ve Aşırı Yüklenmiş Fonksiyonlar

- Tür mirası, bir tür sınıfları hiyerarşisine dayanır
- `Eq` ve `Show` sınıfları temel sınıflardır
  - Önceden tanımlanmış tüm Haskell türleri, `Show` sınıfının örnekleridir
  - `Eq` sınıfı, bir üye türünün iki değerinin == operatörü kullanılarak karşılaştırılabilme yeteneğini oluşturur

---

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x == y     = not (x/=y)
    x /= y     = not (x==y)
```

Tip Sınıfları ve Aşırı Yüklenmiş Fonksiyonlar

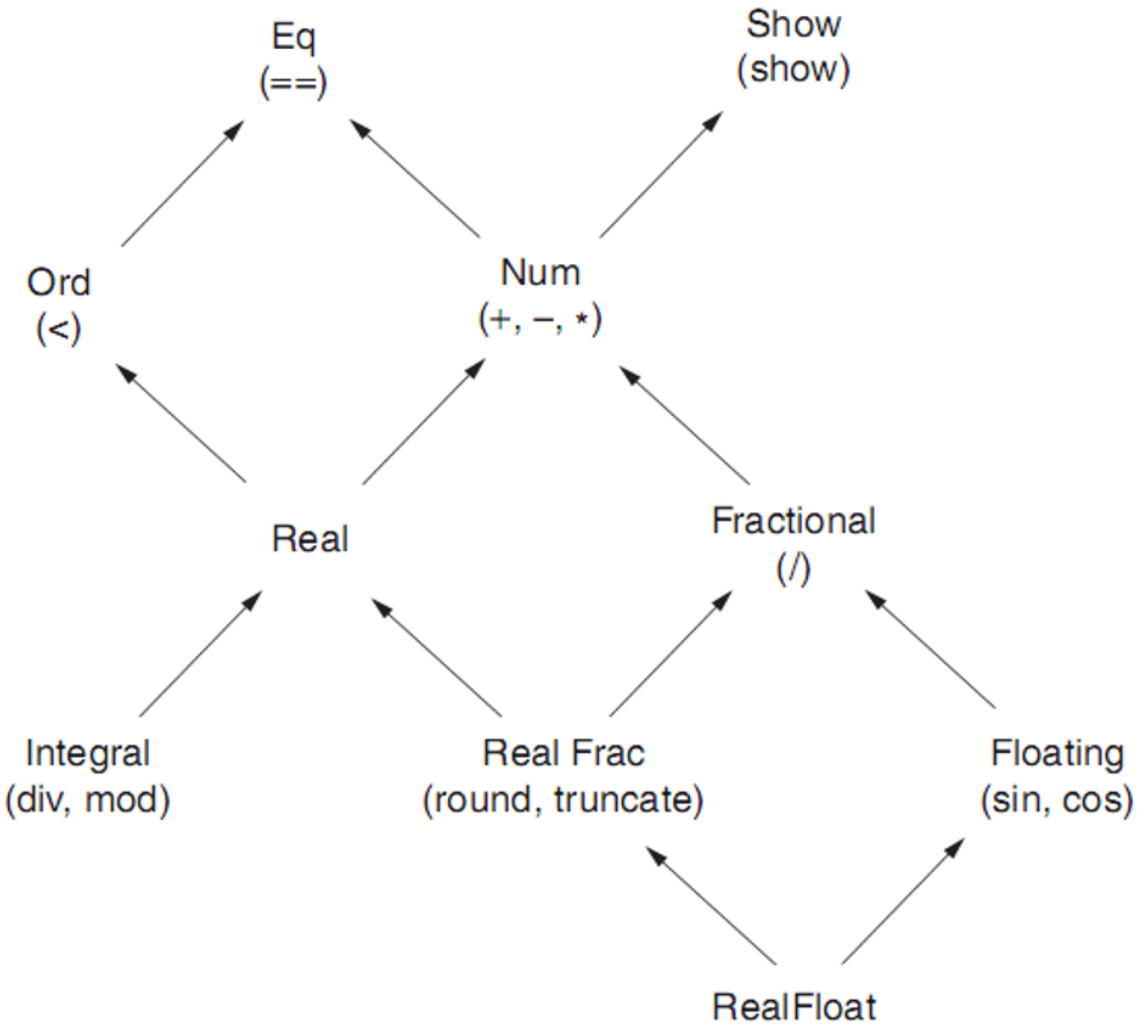


Figure 3.10 The numeric type class hierarchy in Haskell, with sample functions required by some of the classes in parentheses

Haskell Örnek Uygulamaları

- Aşağıdaki işlevleri gerçekleştiren fonksiyonları yazınız:
  - 1 ile parametre olarak verilen n sayısı arasındaki tek sayıların toplamını hesaplayan fonksiyon
  - Parametre olarak verilen listedeki elemanların toplamını döndüren fonksiyon
  - Parametre olarak verilen listenin en küçük elemanını veren fonksiyon
  - Parametre olarak verilen listedeki tek sayıların bulunduğu listeyi döndüren fonksiyon
  - Bir fonksiyon ve bir listeyi parametre olarak alan ve listeyi fonksiyona göre filtreleyen fonksiyon
  - `map` fonksiyonunu kullanan ve parametre olarak verilen listedeki sayıların karesini alıp liste olarak döndüren fonksiyon
  - İkili arama ağacında arama yapan fonksiyon
  - İkili arama ağacına ekleme yapan fonksiyon
  - İkili arama ağacından değer silen fonksiyon

Fonksiyonel Programlamanın Matematiği: Lambda Hesabı

- **Lambda hesabı(Lambda calculus)**: 1930'larda Alonzo Church tarafından icat edildi
  - Hesaplamayı fonksiyonlarla ifade etmek için matematiksel bir biçimcilik
  - Tamamen fonksiyonel programlama dilleri için bir model olarak kullanılabilir
- Lisp, ML ve Haskell dahil birçok fonksiyonel dil, lambda hesabına dayanır

Lambda Hesabı

- **Lambda soyutlaması(abstraction)**: lambda hesabının temel yapısı: $(\lambda x.+\ 1\ x)$
- Tam olarak bu Scheme lambda ifadesi olarak yorumlanabilir: `(lambda (x) (+ 1 x))`
  - x parametresinin x'e 1 ekleyen adlandırılmamış bir fonksiyonu
- Lambda hesabının temel işlemi, lambda soyutlaması gibi ifadelerin **uygulanmasıdır(application)**.

Lambda Hesabı

- Bu ifade: $(\lambda x.+\ 1\ x)\ 2$
  - Sabit 2'ye 1'i x'e ekleyen fonksiyonun uygulamasını temsil eder
- Bir **indirgeme kuralı(reduction rule)**, 2'nin lambda'da x'in yerine geçmesine izin verir ve bunu verir:

  $(\lambda x.+\ 1\ x)\ 2 \Rightarrow (+12) \Rightarrow 3$

Lambda Hesabı

- Lambda hesabı için sözdizimi:

$$
\begin{aligned}
exp \rightarrow & constant \\
& \mid variable \\
& \mid (exp\ exp) \\
& \mid (\lambda\ variable\ .\ exp)
\end{aligned}
$$

- Üçüncü kural, fonksiyon uygulamasını temsil eder
- Dördüncü kural lambda soyutlamaları verir
- Burada tanımlanan Lambda hesabı tamamen körilenmiştir

Lambda Hesabı

- Lambda hesabı değişkenleri hafızayı işgal etmez
- Sabitler kümesi ve değişkenler kümesi dilbilgisi tarafından belirtilmez
  - Birçok **lambda taşından(lambda calculi)** bahsetmek daha doğrudur
- $(\lambda\ x.E)$ ifadesinde
  - x lambda ile bağlıdır
  - E ifadesi, bağlamanın kapsamıdır
  - **Serbest oluşum(Free occurence)**: kapsam dışındaki herhangi bir değişken oluşumu
  - **Bağlı oluşum(Bound occurence)**: özgür olmayan bir oluşum

Lambda Hesabı

- Bir değişkenin farklı oluşumları farklı lambdalar tarafından bağlanabilir
- Bir değişkenin bazı oluşumları bağlı olabilirken diğerleri serbesttir
- Lambda hesaplaması fonksiyonel programlamayı modelleme olarak düşünülebilir:
  - Fonksiyon tanımı olarak bir lambda soyutlaması
  - Fonksiyon uygulaması olarak iki ifadenin yan yana getirilmesi

Lambda Hesabı

- **Tipli lambda hesabı(Typed lambda calculus)**: veri türü kavramını içeren daha kısıtlayıcı form, böylece izin verilen ifade kümesini azaltır
- İfadeleri dönüştürmek için kesin kurallar verilmelidir
- **Değiştirme(Substitution)** (veya **fonksiyon uygulaması(function application)**): lambda hesabında **beta indirgeme(beta-reduction)** olarak adlandırılır
- **Beta-soyutlama(Beta-abstraction)**: yer değiştirme sürecini tersine çevirmek
- **Beta dönüşümü(Beta conversion)**: beta indirgeme veya beta soyutlama

Lambda Hesabı

- **Ad yakalama sorunu(Name capture problem)**: beta dönüştürme yapılırken ve iç içe kapsamlarda oluşan değişkenleri değiştirirken, yanlış bir azaltma meydana gelebilir
  - İç lambda soyutlamasındaki değişkenin adını değiştirmeli (**alfa dönüştürme(alpha conversion)**)

$$
\begin{aligned}
&(\lambda\ f.\lambda\ x.f\ x)\ (\lambda\ x.+\ 1\ x)\ 3 \\
&\quad = (\lambda\ x.(\lambda\ i0.+\ 1\ i0)\ x)\ 3 \\
&\quad = (\lambda\ x.+\ 1\ x)\ 3 \\
&\quad = +\ 1\ 3 \\
&\quad = 4
\end{aligned}
$$

- **Eta dönüşümü(Eta-conversion)**: "gereksiz" lambda soyutlamalarının ortadan kaldırılmasına izin verir
  - Fonksiyonel dillerde körili tanımları basitleştirmede faydalıdır

Lambda Hesabı

- Uygun sıralı değerlendirme(applicative order evaluation)(değer olarak gönderme) ve normal sıralı değerlendirme(normal order evaluation)(isim olarak gönderme)
- Örnek: bu ifadeyi değerlendirin: $(\lambda x.* \ x \ x)(+ \ 2 \ 3)$
  - Uygun sıra(applicative order) kullanıldığında; (1 2 3) 'ü değeriyle değiştirilir ve ardından beta inditgeme uygulanır:
    $(\lambda x.* \ x \ x)(+ \ 2 \ 3) \Rightarrow (\lambda x.* \ x \ x) \ 5 \Rightarrow (* \ 5 \ 5) \Rightarrow 25$
  - Normal sıra kullanıldığında; önce beta indirgeme uygulamak ve sonra değerlendirmek şunları verir:
    $(\lambda x.* \ x \ x)(+ \ 2 \ 3) \Rightarrow (* \ (+ \ 2 \ 3) \ (+ \ 2 \ 3)) \Rightarrow (* \ 5 \ 5) \Rightarrow 25$
  - Normal sıralı değerlendirme bir tür **gecikmiş(delayed)** değerlendirmedir

Lambda Hesabı

- Parametre değerlendirmesinin tanımlanmamış bir sonuç vermesi gibi farklı sonuçlar ortaya çıkabilir
  - Normal sıra yine de doğru değeri hesaplayacaktır
  - Uygun sıra(applicative order) tanımlanmamış bir sonuç verecektir
- Parametreler tanımsız olsa bile bir değer döndürebilen işlevlerin **katı olmadığı(nonstrict)** söylenir
- Parametreler tanımsız olduğunda tanımlanmamış olan fonksiyonların **katı(strict)** olduğu söylenir
- Church-Rosser teoremi: indirgeme dizileri esasen gerçekleştirildikleri sıradan bağımsızdır

Lambda Hesabı

- **Sabit nokta(Fixed point)**: başka bir işleve bağımsız değişken olarak iletildiğinde işlev döndüren işlev
- Lambda hesabında özyinelemeli bir fonksiyon tanımlamak için, fonksiyonun lambda ifadesinin sabit bir noktasını oluşturmak için bir Y fonksiyonuna ihtiyaç vardır
  - Y sabit nokta birleştirici olarak adlandırılır
- Çünkü doğası gereği Y aslında bir anlamda "en küçük" olan bir çözüm inşa edecek; lambda hesabında özyinelemeli fonksiyonların **en küçük sabit noktalı semantiğine(least-fixed-point semantics)** başvurulabilir

Lambda Hesabı Örnekleri

- Aşağıdaki işlevleri gerçekleştiren Lambda λ ifadelerini yazınız
  - Mantıksal **değil** işlemi

Lambda Hesabı Örnekleri

- Aşağıdaki işlevleri gerçekleştiren Lambda λ ifadelerini yazınız
  - Mantıksal **değil** işlemi
    - λk.if k false true
  - Mantıksal **ve** işlemi

Lambda Hesabı Örnekleri

- Aşağıdaki işlevleri gerçekleştiren Lambda λ ifadelerini yazınız
  - Mantıksal **değil** işlemi
    - λk.if k false true
  - Mantıksal **ve** işlemi
    - λa.λb.if a (if b true false) false
  - Mantıksal **veya** işlemi

Lambda Hesabı Örnekleri

- Aşağıdaki işlevleri gerçekleştiren Lambda λ ifadelerini yazınız
  - Mantıksal **değil** işlemi
    - λk.if k false true
  - Mantıksal **ve** işlemi
    - λa.λb.if a (if b true false) false
  - Mantıksal **veya** işlemi
    - λa.λb.if a true (if b true false)
  - Bölümden kalanı bulan ifade(modulo)

Lambda Hesabı Örnekleri

- Aşağıdaki işlevleri gerçekleştiren Lambda λ ifadelerini yazınız
  - Mantıksal **değil** işlemi
    - λk.if k false true
  - Mantıksal **ve** işlemi
    - λa.λb.if a (if b true false) false
  - Mantıksal **veya** işlemi
    - λa.λb.if a true (if b true false)
  - Bölümden kalanı bulan ifade(modulo)
    - λa.λb.- a (* b (/ a b))
  - Bir değerin çift olup olmadığını veren ifade(cift)

Lambda Hesabı Örnekleri

- Aşağıdaki işlevleri gerçekleştiren Lambda λ ifadelerini yazınız
  - Mantıksal **değil** işlemi `if k is true return false`
    - λk.if k false true `if false return true`
  - Mantıksal **ve** işlemi `if a is false return false, if true look at b if b is true return true if false return false`
    - λa.λb.if a (if b true false) false
  - Mantıksal **veya** işlemi
    - λa.λb.if a true (if b true false)
  - Bölümden kalanı bulan ifade(modulo)
    - λa.λb.- a (* b (/ a b))

e.g: 7 3 `7/3=2...2*3=6...7-6=1`

`this / returns int`

  - Bir değerin çift olup olmadığını veren ifade(cift)
    - mod tanımlı ise:
    - λx.if (= 0 (mod x 2)) true false
    - aksi halde(/ işlemi tamsayı bölmesi)
    - λx.if (= 0 (- x (* 2 (/ x 2)))) true false

Lambda Hesabı Örnekleri

- Aşağıdaki ifadeler değerlendirildiğinde sonuç ne olur?
  - $(\lambda f.\lambda x.\ f\ x\ x) + 5$
  - $(\lambda a.\lambda b.\lambda c.\lambda d.4)\ 5\ 7\ 9\ 11$
  - $(\lambda a.\lambda b.\text{if}\ (=\ a\ b)\ (+\ a\ b)\ (*\ a\ b))\ 7\ 4$
  - $(\lambda x.\lambda y.\lambda z.\lambda f.f\ (f\ x\ y)\ z)\ 2\ 3\ 4\ (\lambda a.\lambda b.\ *\ a\ (+\ b\ 2))$
  - $(\lambda f.\lambda x.\lambda y.f\ x\ y)(\lambda a.\lambda b.\ +\ (*\ a\ b)\ (+\ a\ b))\ 5\ 6$
  - $(\lambda f.\lambda x.\lambda y.f\ x\ y)\ (\lambda a.\lambda b.\text{if}\ (<\ a\ b)\ a\ b)\ 7\ 4$

Lambda Hesaplayıcı

- Dr. Carl Burch tarafından yazılan
  - Java Uygulaması
  - Çevrimiçi hesaplayıcı