Programlama Dilleri

Onur Tolga Şehitoğlu

April 6, 2007

Contents

1 Giriş

Giriş

- Church-Turing hipotezi bütün programlama dillerinin ve bilinen berimsel cihazların (performans olarak değil, ifade ve problem çözme yeteneği olarak) eşlenik olduğunu söyler.
- Yani herhangi bir genel amaçlı programlama diliyle yazdığımız bir algoritmayı başka bir programlama diliyle de yazabiliriz.
- Peki neden bu kadar çok farklı programlama dili var?
- Çok fazla sayıda programlama dili bilmenin bir yararı var mı?

1.1 Dersin Amaçları

Dersin Amaçları

- Dil öğretmek **değil**. Haskell, C++ ve Java dillerini dersteki konuları göstermek için öğreneceğiz.
- Programlama dillerinin ortak kavramlarını incelemek.
- Farklı paradigmaları, yaklaşımları incelemek.
- Programlama dilinin "daha iyi" olması ne demektir, ne programlama dilini daha nitelikli yapar.
- Derleyici tasarımı, yazılım mühendisliği, nesneye yönelik tasarım, bilgisayar insan etkileşimi gibi diğer konulara temel oluşturmak için.

İlgili Alanlar

- İnsan bilgisayar etkileşimi
- İşletim sistemleri
- Bilgisayar mimarisi
- Veri tabanı ve bilgi erişimi

1.2 Programlama Dilleri ile Doğal Diller

Programlama Dilleri ile Doğal Diller

- Biçemli ile serbest (formal, informal)
- Katı kurallı ile hataya karşı esnek
- Kısıtlı alan ile büyük ve sınırsız alan

1.3 Programlama Dillerinin Özellikleri

Programlama Dili Nasıl Olmalı?

Herşey formal tanım programlama dili midir? (örneğin HTML?)

- **Evrensel** Bütün berimsel problemlerin çözümü olmalı. koşul+(döngü ya da özyineleme)
- **Doğal** Uygulama alanına yönelik özellikleri olmalı. Fortran: sayısal hesaplama, COBOL: dosya işleme, LISP ağaç ve liste işlemleri.
- **Uygulanabilir** Bilgisayar üzerinde çalışabilecek bir derleyicisi ya da yorumlayıcısı yapılabilmeli. Matematik, doğal dil "bana şu problemin çözümünü bul"
- **Etkili** Az kaynakla ve iyi başarımla çalışabilmeli.

Farklı dillerde "hello world" uygulamaları: `http://www.latech.edu/~acm/HelloWorld.shtml`

1.4 Paradigmalar

Paradigmalar

Paradigma: Kuramsal ya da model çerçevesi. Benzer yaklaşımlara temel oluşturan model.

- **Buyurgan(Imperative)** Fortran, Cobol, Algol, Pascal, C, C++, Java, Basic.
- **Fonksiyonel** Lisp, Scheme, ML, Haskell, Miranda
- **Nesneye Yönelik** Smalltalk, C++, Object Pascal, Eifell, Java, Csharp.
- **Eşzamanlı(Concurrent)** Ada, Occam, Par-C, Pict, Oz
- **Mantık** Prolog, Icon
- **Kısıt Programlama** CLP, CHR
- **Karışık** Parallel Prolog, Oz (Fonksiyonel, mantık, nesneye yönelik, eş zamanlı özellikleri olan bir dil: `http://www.mozart-oz.org`

1.5 Sözdizimi ve Anlam

Sözdizimi ve Anlambilim

- **Sözdizimi (Syntax)** Biçim. Dil yapılarının nasıl göründüğü, nasıl ifade edildiği.
- Sözdizimi çoğunlukla BNF (Bacus Naur Form) olarak ifade edilen Bağlam Bağımsız Gramerlerdir (Context Free Grammar).
- "Formal Diller ve Soyut Makinalar" derslerinde ve Derleyici Tasarımı derslerinde detaylı olarak anlatılır
- **Anlambilim (Semantics)** Programların anlamı, bilgisayarda nasıl çalıştığı

1.6 Dil İşleyicileri

Dil İşleyicileri

- Derleyiciler (compiler)
- Yorumlayıcılar (interpreter)
- Güzelleştiriciler (pretty printers, a2ps, ident)
- Sözdizim yetenekli düzenleyiciler (syntax directed editing, vim, anjuta, eclipse, visual studio)
- Program denetleyicileri (lint)
- Program doğrulayıcılar (verification)

2 Değerler ve Tipler

2.1 Değer ve Tip

Değer ve Tip Nedir?

- Değer "computation" sırasında varolan, hesaplanabilen, saklanabilen, veri yapılarında yer alan herşeydir. Sabit değerler, değişkenlerin değerleri, parametreler, fonksiyon dönüş değerleri, işlem sonuçları vs.
- Tip aynı türde olan değerlerin kümesidir. C'deki tipler
  - `int, char, long,...`
  - `float, double`
  - göstergeler (pointer)
  - yapılar: `struct, union`
  - diziler
- Haskell tipleri
  - `Bool, Int, Float, ...`
  - `Char, String`
  - ikililer,(N-liler), kayıtlar
  - listeler
  - fonksiyonlar
- Her tip bir değer kümesi belirtir, ama bu yeterli bir tanım mı? Alttaki küme bir tip mi? {`"ahmet", 1 , 4 , 23.453, 2.32, 'b'`}
- Kümedeki değerlerin benzer bir davranışı olması, üzerlerinde aynı temel bir grup işlemin yapılabilmesi gerekir.

2.2 İlkel ve Bileşik Tipler

İlkel Tipler

- İlkel Tipler: Alt değerlere bölünemeyen değerlerden oluşur C: `int, float, double, char, long, short`, göstergeler Haskell: `Bool, Int, Float`, fonksiyon değerleri
- tipin boyu: Bir veri tipinin boyu (kümenin kardinalitesi) o veri tipinde yer alabilecek farklı değerlerin sayısıdır. "#Tip" olarak gösterilir. #Bool = 2 #char = 256 #short = 2^{16} #int = 2^{32} #double = 2^{32}, ...
- tipin boyu ne ifade ediyor? En başta saklamak için kaç bitlik alana gerek olduğunu gösterir.

Kullanıcı Tanımlı İlkel Tipler

- sayımlı (enumerated) tipler `enum gunler {pzt, sal, car, per, cum, cts, paz}; enum aylar {ocak, subat, mart, nisan, .... };`
- aralıklar (Pascal ve Ada) `type Gun = 1..31;  var g:Gun;`

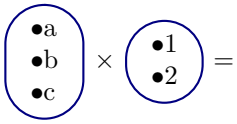
Bileşik Veri Tipleri

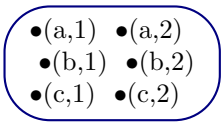
Bir ya da daha fazla veri tipinin birleştirilmesiyle kullanıcı tarafından oluşturulur. Birleştirme yöntemlerine göre farklılaşırlar:

- Kartezyen çarpım (struct, ikililer, kayıtlar)
- Ayrık bileşim (union (C), variant record (pascal), Data (haskell))
- Eşleşme (diziler, fonksiyonlar)
- Üstkümesi (küme veri tipi (Pascal))
- Özyinelemeli bileşimler (listeler, ağaçlar, karmaşık veri yapıları)

2.3 Kartezyen Çarpım

Kartezyen Çarpım

- $S \times T = \{(x, y) \mid x \in S, y \in T\}$
- Örnek: $S = \{a, b, c\}$ $T = \{1, 2\}$ $S \times T = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$ 



- $\#(S \times T) = \#S \cdot \#T$
- C `struct`, Pascal `record`, fonksiyonel dillerde İkili (Tuple)
- **C'de:** string × int

```
struct Kisi {
     char isim[20];
     int no;
} x = {"Osman Hamdi",23141};
```

- **Haskell'de**: string × int

```
type Kisi=(String,Int)
...
(x::Kisi) = ("Osman Hamdi",23141)
```

- Birden çok kartezyen çarpım: C: string × int × {ERKEK,KADIN}

```
struct Kisi {
     char isim[20];
     int no;
     enum Cinsiyet {ERKEK, KADIN} cinsiyet;
} x = {"Osman Hamdi",23141,ERKEK};
```

Haskel: string × int × float × String

```
x = ("Osman Hamdi",23141,3.98,"Yazar")
```

Eştürlü kartezyen tipler

- $S^n = \overbrace{S \times S \times S \times ... \times S}^{n}$ double^4 :

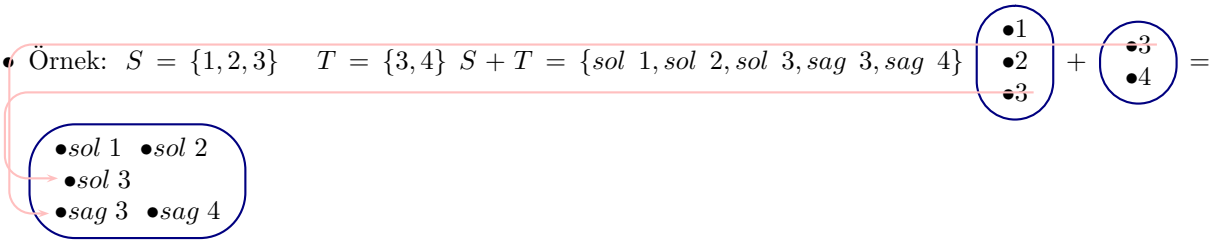
```
struct quad { double x,y,z,q; };
```

- $S^0 = \{()\}$ is 0-tuple.
- Boş küme *değil*. Tek elemanlı bir küme
- Fonksiyonel dillerde (nil) sonlandırıcı değer
- C `void`. Değer yok anlamında, değersiz anlamında, boş küme...

2.4 Ayrık Birleşim (Disjoint Union)

Ayrık Birleşim (Disjoint Union)

- $S + T = \{sol\ x \mid x \in S\} \cup \{sag\ x \mid x \in T\}$
- Örnek: $S = \{1,2,3\}$ $T = \{3,4\}$ $S + T = \{sol\ 1, sol\ 2, sol\ 3, sag\ 3, sag\ 4\}$



- $\#(S + T) = \#S + \#T$
- C `union`'lar ?
- **C:** int + double:

```
union sayi { double reel; int tam; } x;
```

- C `union`'lar güvenilir değildir. Aynı saklama alanını paylaşırlar ve gerçek verinin hangi alanda olduğu belli değildir:

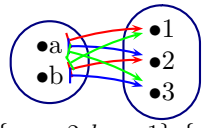
```
x.reel=3.14; printf("%d\n",x.tam);
```

- **Haskel:** Float + Int + (Int × Int):

```
data Sayi = Reel Float | Tam Int | Paydali (Int,Int)
x = Paydali (3,4)
y = Reel 3.14
z = Tam 12          {-- Farkli degerlere erisilemez --}
```

2.5 Eşleşmeler

Eşleşmeler

- Olası bütün eşleşmelerin kümesi
- $S \mapsto T = \{V \mid \forall(x \in S)\exists(y \in T), (x \mapsto y) \in V\}$
- Örnek: $S = \{1,2,3\}$ $T = \{a,b\}$ 

her renk bir eşleşme ve birçok eşleşme daha var $S \mapsto T = \{\{a \mapsto 1, b \mapsto 1\}, \{a \mapsto 1, b \mapsto 2\}, \{a \mapsto 1, b \mapsto 3\}, \{a \mapsto 2, b \mapsto 1\}, \{a \mapsto 2, b \mapsto 2\}, \{a \mapsto 2, b \mapsto 3\}, \{a \mapsto 3, b \mapsto 1\}, \{a \mapsto 3, b \mapsto 2\}, \{a \mapsto 3, b \mapsto 3\}\}$
- $\#(S \mapsto T) = \#T^{\#S}$

2.5.1 Diziler

Diziler

- `double a[3]={1.2,2.4,-2.1};` `a` $\in (\{0,1,2\} \mapsto$ `double`$)$ `a` $= (0 \mapsto 1.2, 1 \mapsto 2.4, 2 \mapsto -2.1)$
- Diziler bir tamsayı aralığından (ya da eşleşebilen bir tipten) eleman tipine bir eşleşme tanımlarlar.
- **C:** T`x`$[N]$ $\Rightarrow$ `x` $\in (\{0,1,...,N-1\} \mapsto T)$
- Tamsayı olmayan dizi indeksleri (Pascal):

```
type
   Gun = (Pzt,Sal,Crs,Prs,Cum,Cts,Paz);
   Ay = (Oca,Sub,Mar,Nis,May,Haz,Tem,Agu,Eyl,Eki,Kas,Ara);
var
   x : array Gun of real;
   y : array Ay of integer;
...
   x[Sal] := 2.4;
   y[Sub] := 28;
```

2.5.2 Fonksiyonlar

Fonksiyonlar

- C fonsiyonu:

```
int f(int a) {
    if (a%2 == 0) return 0;
    else return 1;
}
```

- `f` : `int` $\mapsto \{0,1\}$ fonksiyon içeriğine bakılmazsa: `f` : `int` $\mapsto$ `int`
- Haskell:

```
f a = if mod a 2 == 0 then 0 else 1
```

- C'de yalnızca `f` ifadesinin gerçek tipi göstergedir: `int (*)(int)` Haskell'de ise eşleşmedir: `int`$\mapsto$`int`

Eşleşmelerde dizi ile fonksiyon farkı

Diziler

- Bellekte değer
- Kısıtlı: sadece tam sayılardan
- `double`↦`double` ?

Fonksiyonlar

- Algoritma
- Hız, kaynak kullanımı
- Her türlü tipten eşleşme
- Yan etki, çıktı. Hata ve sonsuz döngü

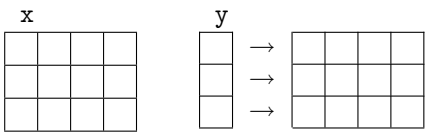
- Kartezyen eşleşmeler: `double a[3][4]; double f(int m, int n);`
- `int`×`int`↦`double` ve `int`↦(`int`↦`double`)

Kartezyen eşleşme ve iç içe eşleşme

- Pascal dizileri

```
var
   x : array [1..3,1..4] of double;
   y : array [1..3] of array [1..4] of double;
...
x[1,3] := x[2,3]+1;      y[1,3] := y[2,3]+1;
```

- Satır işlemi: `y[1] := y[2] ;` √ `x[1] := x[2] ;` ×



- Haskell fonksiyonları:

```
f (x,y) = x+y
g x y = x+y
...
f (3+2)
g 3 2
```

- `g 3` √ `f 3` ×
- Eski tanımı kullanarak yeni bir fonksiyon: `artir = g 1` `artir 1 2`

2.6 Üstkümesi

Üstkümesi

- $\mathcal{P}(S) = \{T \mid T \subseteq S\}$
- Bütün alt kümelerin kümesi.
- $S =$ •1 •2 •3 $\quad \mathcal{P}(S) =$ •$\emptyset$ •$\{1\}$ •$\{2\}$ •$\{3\}$ •$\{1,2\}$ •$\{1,3\}$ •$\{2,3\}$ •$\{1,2,3\}$
- $\#\mathcal{P}(S) = 2^{\#S}$
- Küme veri tipi. Kısıtlı ve özel bir tip. Pascal ve küme dili SetL
- küme işlemleri (Pascal)

```
type
   renk = (kirmizi,yesil,mavi,beyaz,siyah);
   kume = set of renk;
var
   a,b : kume;
...
a := [kirmizi,mavi];
b := a*b;                  (* kesisim *)
b := a+[yesil,kirmizi];    (* birlesim *)
b := a-[mavi];             (* fark *)
if (yesil in b) then ...   (* elemani mi? *)
if (a = []) then ...       (* kume esitligi *)
```

2.7 Özyinelemeli Tipler

2.7.1 Listeler

Özyinelemeli Tipler

- $S = ...S...$
- Kensini bileşen olarak içeren tipler. Tanımında kendisini gösteren tipler.

Listeler

- $S = Int \times S + \{bos\}$

$$\begin{aligned} S = \quad & \{sag\ bos\} \cup \{sol\ (x, bos) \mid x \in Int\} \cup \\ & \{sol\ (x, sol\ (y, bos)) \mid x, y \in Int\} \cup \\ & \{sol\ (x, sol\ (y, sol\ (z, bos))) \mid x, y, z \in Int\} \cup ... \end{aligned}$$

- $S = \{sag\ bos, sol(1, bos), sol(2, bos), sol(3, bos), ..., sol(1, sol(1, bos)), sol(1, sol(2, bos)), sol(1, sol(3, bos), ..., sol(1, sol(1, sol(1, bos))), sol(1, sol(1, sol(2, bos))), ...\}$
- C listeleri. Bağlı liste, göstergelerle. Gerçek özyineleme yok

```
struct List {
     int x;
     List *next;
} a;
```

- Haskell listeleri.

```
data Liste = Sol (Int,Liste) | Bos

x = Sol (1, Sol(2, Sol(3,Bos)))      {-- [1,2,3] Listesi --}
y = Bos                              {-- Bos liste, []   --}
```

- Çokşekilli listeler: Tek bir tanımda değişik tiplerin listesi tanımlanabilir.
- $Liste\ \alpha = \alpha \times (Liste\ \alpha) + \{bos\}$

```
data Liste alpha = Sol (alpha,Liste alpha) | Bos

x = Sol (1, Sol(2, Sol(3,Bos)))      {-- [1,2,3] Listesi --}
y = Sol ("ali",Sol("ahmet",Bos))     {-- ["ali","ahmet"] --}
z = Sol(23.1,Sol(32.2,Sol(1.0,Bos))){-- [23.1,32.2,1.0] --}
```

- $Sol(1, Sol(\text{“}ali\text{”}, Sol(15.23, Bos) \in Liste\ \alpha$? Hayır. Birçok dil sadece eştürlü (homogenous) listeye izin verir.

Haskell Listeleri

- İkili operatör şeklinde tanımlı tip yapıcısı “:” : `data [alpha] = (alpha :  [alpha]) | []`
- `x = (1:(2:(3:[])))`
- Gösterim kolaylığı: `[1,2,3]` $\equiv$ `(1:(2:(3:[])))` `["ali"]` $\equiv$ `("ali":[])`

2.7.2 Genel özyinelemeli tipler

Genel özyinelemeli tipler

- $T = ...T...$
- Bu formülün bir minimum çözümü olmalı: $T = Int \times T$ Bir değer yazmak mümkün mü? Minimum çözüm yok.
- Liste örneğinde: `x = Sol(1,Sol(2,x))` $x \in S$? Evet `[1,2,1,2,1,2,...]` değerini işlemek olası mı?
- Bazı diller sonsuz dizilerle de işlem yapabilir ama sonsuz döngüye girme tehlikesi var.
- Genelde diller sadece S'nin bir alt kümesini, sonlu listeler kümesine izin verir ve işlem yaparlar
- Haskell sonsuz dizi tanımına izin verir.
- $Agac\ \alpha = bos + dugum\ \alpha \times Agac\alpha \times Agac\alpha$

$$
\begin{aligned}
Agac\ \alpha = \quad & \{bos\} \cup \{dugum(x, bos, bos) \mid x \in \alpha\} \cup \\
& \{dugum(x, dugum(y, bos, bos), bos) \mid x, y \in \alpha\} \cup \\
& \{dugum(x, bos, dugum(y, bos, bos)) \mid x, y \in \alpha\} \cup \\
& \{dugum(x, dugum(y, bos, bos), dugum(z, bos, bos)) \mid x, y, z \in \alpha\} \cup ...
\end{aligned}
$$

- C++ (göstergelerle ve şablon tanımla)

```
template<class Alpha>
struct Agac {
     Alpha x;
     Agac *sol,*sag;
} kok;
```

- Haskell

```
data Agac alpha = Bos |
                  Dugum (alpha,Agac alpha, Agac alpha)

x = Dugum (1,Dugum (2,Bos,Bos),Dugum(3,Bos,Bos))
y = Dugum(3,Bos,Bos)
```

2.7.3 Söz dizileri (strings)

Söz dizileri (strings)

Dil tasarım tercihi:

1. İlkel tip (ML)
2. Karakter dizisi (array of char, C, Pascal, ...)
3. Karakter listesi (list of char, Haskell, Prolog, Lisp)

- Temel işlemler tercihe göre zor ya da kolay: birleştirme, atama, karşılaştırma, sıralama, parçalama,...

2.8 Tip Sistemleri

Tip Sistemleri

- Veri işleme, gösterim, erişimin olanaklı ve hızlı olması için tiplerin olması şart. Tiplerin uyumluluğu garanti edilmeli.
- Kullanıcı hataları → tip uyumsuzlukları olarak derleyiciler tarafından yakalanbilir.
- Anlamsız işlemler: `y=true * 12; x=12; x[1]=6; y=5; x.a = 4;`
- Tip denetlemesi ne zaman yapılabilir? İşlemi gerçekleştirmeden önce yapılmak zorunda. İki seçenek
  1. Derleme esnasında → statik tip denetlemesi
  2. Çalışma zamanında → dinamik tip denetlemesi

2.8.1 Statik Tip Denetlemesi

Statik Tip Denetlemesi

- Bütün değerlerin derleme esnasındaki tipleri üzerinden denetleme yapılır.
- Tip uyumsuzluklarının hepsi derleme anında tespit edilir. Değişkenlerin program boyunca sabit bir tipi vardır. Çoğu dil statik denetleme yapar.
- Kullanıcı tanımlı sabit, değişken ve fonksiyonların tipleri:
  - Tam tipi açıklamak zorunlu (C, C++, Fortran,...)
  - Zorunlu değil, çıkarım yapan diller (Haskell, ML,...)
- Derlenme sonrasında tiplerle ilgili başka işlem yapılmaz

2.8.2 Dinamik Tip Denetlemesi

Dinamik Tip Denetlemesi

- Çalışma sırasında işlem gerçekleşinceye kadar denetleme yapılmaz, işlem gerçekleşmeden önce yapılır.
- Lisp, Prolog, PHP, Perl, Python gibi yorumlanan bazı diller.
- Böyle bir dil olduğunu varsayalım:

```
int hangiay(girdi) {
        if (isinteger(girdi))   return girdi;
        else  if (isstring(girdi))
                switch(girdi) {
                case "Ocak": return 1;
                case "Subat": return 2;
                ...
                case "Aralik": return 12;}
}
...
read(girdi)      /* kullanicidan oku */
ay=hangiay(girdi)
```

- Kullanıcının girdiği değere göre çalışma zamanında karar verilebilir.
- Her değer için tip bilgisi çalışma zamanında tutulmak zorunda

Statik ile dinamik denetleme

- Statik denetleme daha hızlı çalışır. Dinamik denetleme çalışma zamanında tip işlemleri için zaman kaybeder. Ayrıca çalışma anında tip bilgisini saklamak için fazladan bellek tüketir.
- Statik denetleme daha güvenilirdir. Tip hatalarının derleme zamanında ayıklanması kullanıcı hataları önceden tespitte yararlı olur.
- Dinamik denetleme daha esnekdir. Bazı uygulamalarda verinin tipi önceden bilinmeden genel işlemler yapmak gerekebilir.

Tip Eşitliği

- $S \equiv T$? kararı neye göre verilmeli?
  - İsim eşitliği: İki tipin aynı yerde tanımlanması gerekir.
  - Yapısal eşiklik: İki tipin aynı değer kümesine sahip olması (matematiksel küme eşitliği).
- Çoğu dil isim eşitliği kullanır.
- C örneği:

```
typedef struct Kompleks { double x, y;}  Karmasik;
struct Complex { double x,y; };

struct Kompleks a;
Karmasik b;
struct Complex c;

/* ... */
a=b;   /* Dogru ifade, tipler esit */
a=c;   /* Derleme hatasi!!! tipler farkli */
```

Yapısal eşitlik

$S \equiv T$ ancak ve ancak:

1. S ve T ilkel tiplerse ve $S = T$ ise yani aynıysa,
2. $S = A \times B$, $T = A' \times B'$ ise, $A \equiv A'$ ve $B \equiv B'$ ise,
3. $S = A + B$, $T = A' + B'$ ise, ($A \equiv A'$ ve $B \equiv B'$) veya ($A \equiv B'$ ve $B \equiv A'$) ise
4. $S = A \mapsto B$, $T = A' \mapsto B'$ ise, $A \equiv A'$ ve $B \equiv B'$ ise,
5. $S = \mathcal{P}(A)$, $T = \mathcal{P}(A')$ ise, $A \equiv A'$ ise.

Diğer durumlarda $S \not\equiv T$

- Yapısal tip eşitliğini uygulamak daha zor. Özellikle özyinelemeli durumlarda.
- $T = \{nil\} + A \times T$, $T' = \{nil\} + A \times T'$ $T = \{nil\} + A \times T'$, $T' = \{nil\} + A \times T$
- `struct Daire { double x,y,r;}; struct Kare { double x,y,w;};` İki ayrı tip tanımlamasının anlamsal bir sebebi var. Kullanıcı hatalarını isim eşitliği bazı durumlarda daha iyi ayıklayabilir.
- Otomatik tip çevirimi farklı. İsim eşitliğini etkisiz kılmaz.

```
enum Gun {Pzt, Sal, Crs, Prs, Cum, Cts, Paz} x;
x=3;
```

2.9 Tip Tamamlığı

Tip Tamamlığı (type completeness)

- Birinci sınıf değerler:
  - Atama (assignment)
  - fonksiyon parametresi olabilme
  - karmaşık değlerlerde yer alabilme
  - fonksiyon değeri olarak dönülebilme
- Çoğu buyurgan dilde (Pascal, Fortran) fonksiyonlar ikinci sınıf değerdir. (C'de göstergeleri 1.sınıftır)
- Haskell ve fonksiyonel dillerde bütün değerler birinci sınıf değerdir.
- Diziler, yapı ve kayıtlar?
- Tip tamamlığı prensibi: Birinci sınıf değerler bu işlemlerin hepsinde yer alabilmelidir, keyfi kısıtlama olmamalıdır.

C tipleri:

	İlkel	Dizi	Yapı	Fonk.
Atama	√	×	√	×
Fonksiyon parametre	√	×	√	×
Fonksiyon dönüş	√	×	√	×
Birleşik tiplerde	√	√	√	×

Haskell tipleri:

	İlkel	Dizi	Yapı	Fonk.
Değişken tanım	√	√	√	√
Fonksiyon parametre	√	√	√	√
Fonksiyon dönüş	√	√	√	√
Birleşik tiplerde	√	√	√	√

Pascal tipleri:

	İlkel	Dizi	Yapı	Fonk.
Atama	√	√	√	×
Fonksiyon parametre	√	√	√	×
Fonksiyon dönüş	√	×	×	×
Birleşik tiplerde	√	√	√	×

2.10 İfadeler

Değerlendirildiğinde bir değeri olan değer veren program kısımları:

- Aynı değerler (sabitler)
- Değişken ve tanımlı sabit erişimi
- Yapılandırıcılar
- Fonksiyon çağrıları
- Koşullu ifadeler
- Yinelemeli ifadeler (Haskell)

2.10.1 Aynı değerler/Değişken ve sabit erişimi

- Sabit, ifadenin değeri gösterimiyle aynı (gösterimsel anlamına göre) olan ifadeler `123, 0755, 0xa12, 12451233L, -123.342, -1.23342e-2, 'c', '\021', "ayse", True, False`
- Değişken ve sabit erişimi: Kullanıcı tanımlı değişken ve sabitlerin isimleri, değerlendirildiklerinde içerdikleri değeri verir. `int x; #define pi 3.1416 x=pi*r*r`

2.10.2 Yapılandırıcılar

Yapılandırıcılar

- Program içerisinde birleşik değerler oluşturmak için kullanılırlar.

```
x=(12,"ali",True)              {-- 3 Tuple --}
y={isim="ali", numara=12}      {-- Kayit   --}
f=\x -> x*x                    {-- fonksiyon --}
l=[1,2,3,4]                    {-- ozyinelemi tip, liste --}
```

- C'de sadece tanım sırasında var. Çalıştırılabilir ifadelerde yapılandırıcı yok!

```
struct Kisi { char ad[20], int no } = {"Ali Cin", 332314};
double dizi[3][2] = {{0,1}, {1.2,4}, {12, 1.4}};
```

2.10.3 Fonksiyon Çağrıları

Fonksiyon çağrıları

- $F(Gp_1, Gp_2, ..., Gp_n)$
- Fonksiyon adı ve gerçek parametre listesi. Fonksiyon çağrılır ve döndüğü değer ifadenin yerine konur.
- Gerçek parametre: çağrı sırasında gönderilen ifadeler
- Tanım parametreleri: fonksiyon tanımı sırasında kullanılan parametre isimleri.
- Aritmetik, mantıksal ve karşılaştırma işlemleri de fonksiyon çağrısı olarak düşünülmelidir. Fark: içel (infix) sözdizimi.
- $\oplus(a, b)$ yerine $a \oplus b$
- birçok dil işlemleri dahili olarak hesaplar. Bazı diller işlemleri kullanıcıların tanımlamasına izin verir: C++, Haskell

2.10.4 Koşullu ifadeler

- Bir ifade ya da seçim sonucuna göre farklı değer dönen ifadeler
- Haskell: `if` *koşul* `then` *ifade1* `else` *ifade2* . `case` *değer* `of` *p1* `->` *ifade1* | *p2* `->` *ifade2* ...
- C: `(`*koşul*`)?`*ifade1*`:`*ifade2* `;`
- C'deki `if .. else` koşullu ifade değil, koşutlu komuttur! İfade olarak değer almaz.

```
x = (a>b)?a:b;
y = ((a>b)?sin:cos)(x);      /* dogru mu? deneyin... */
```

- Haskell:

```
x = if (a>b) then a else b
y = (if (a>b) then (+) else (*)) x y
data Gun = Pzt | Sal | Crs | Prs | Cum | Cts | Paz
cevir a = case a of
          Sol (x,devami) -> x : (cevir devami)
          Bos -> []
gunsayi g = case gun of
          Pzt -> 1
          Sal -> 2
          ...
          Paz -> 7
```

- `case` bir biçime uyup uymadığına bakar ve sol taraftaki değişkenleri şekle göre tanımlayarak sağdaki ifadeyi değerlendirir.

2.10.5 Yinelemeli ifadeler

Yinelemeli ifadeler

- Bir liste ya da veri yapısı üzerinde bir grup işlem yaparak yeni bir liste ya da veri yapısı dönen ifadeler.
- çok örneği yok. Haskell liste işlemleri ender örneklerden birisi.
- `[ ifade | değişken <- liste , koşul]`
- Matematik küme gösterimi benzeri: $\{ifade|degisken \in liste, kosul\}$
-

```
x=[1,2,3,4,5,6,7,8,9,10,11,12]
y=[ a*2 | a <- x ]                {-- [2,4,6,8,...24 ] --}
z=[ a | a <- x, mod a 3 == 1 ]    {-- [1,4,7,10] --}
```

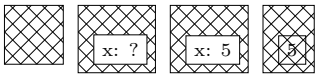
3 Bellek (Depo)

Bellek (Depo)

- Fonksiyonel dillerde değişken: matematik benzeri, tanımlanır ya da çözümlenir, o andan itibaren de ilgili deyimde değeri aynı kalır.
- Buyurgan dillerde değişken: değişken durumu ve değeri vardır. Aynı deyim içinde farklı değerlere atanabilir.
- İki temel işlem: gözlem ve güncelleme. Değişkenin değerini sorma ve değerini değiştirme.

Bilgisayar belleğini hücrelerden oluşan bir depo olarak düşünebiliriz.

- Bütün hücreler başlangıçta serbest.
- Sonra kullanılabilir/tanımsız. Bellekten yeri ayrılmış fakat henüz değeri yok.
- Sonra tanımlı
- İlgili deyim (fonksiyon, program) bitince tekrar serbest




```
f(); void f() {    int x;    ...       x=5;    ...       return; }
```

Toptan ya da özel güncelleme

- Bileşik tipteki değişkenlerin toptan ya da alan bazında seçilmesi ve değiştirilmesi mümkün.

```
struct Karmasik { double x,y; } a, b;
...
a=b;                    // Toptan guncelleme
a.x=b.y*a.x;            // ozel guncelleme
```

-

- İlkel tip değişkenleri: tek hücre Bileşik tip değişkenleri: iç içe girmiş bellek hücreleri

3.0.6 Dizi değişkenleri

Dizi değişkenleri

Dizi kavramı olan dillerde dizi değişkenlerinin boyları ile ilgili farklı yaklaşımlar vardır:

1. Statik diziler
2. Dinamik diziler
3. Esnek diziler

Statik diziler

- Dizinin boyu sadece sabit değerlerden ve sabit değer içeren ifadelerden oluşabilir. Derleme sırasında sabitlenir.
- Örnek: C

```
  #define MAXELS 100
  int a[10];
  double x[MAXELS*10][20];
}
```

Dinamik diziler

- Dizinin boyu değişken tanımı sırasında belirtilir. Sonrasında sabit kalır.
- Örnek: GCC eklentisi (ANSI'ye uymuyor!)

```
int f(int n) {
    double a[n]; ...
}
```

- örnek: C++'da template kullanarak yapılabilir.

```
template<class T>  class Dizi {
      T *icerik;
  public:
      Dizi(int s) { icerik=new T[s]; }
      ~Dizi()     { delete [] icerik; }
};
...
Dizi<int>  a(10);             Dizi<double> b(n);
```

Esnek diziler

- Dizilerin boyu tamamen değişkendir. Çalışma zamanında uzayıp kısalabilir. Perl, PHP, Python gibi betik (script) dillerde görülür.
- Örnek: Perl

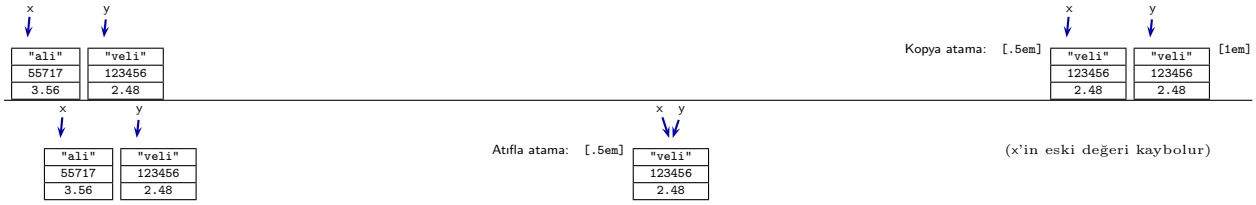
```
@a=(1,3,5);           # dizi boyu: 3
print $#a , "\n";     # cikti: 2 (0..2)
$a[10] = 12;          # dizi boyu 11 (ara elemanlar tanimsiz)
$a[20] = 4;           # dizi boyu 21
print $#a , "\n";     # cikti: 20 (0..20)
delete $a[20];        # son eleman silindi boy tekrar 11
print $#a , "\n";     # cikti: 10 (0..10)
```

- C++ ve nesneye yönelik dillerde `[]` operatörünü yükleyerek esnek dizi uygulamaları yapmak mümkün. C++ STL (Standart Template Library) içindeki vector, map gibi sınıflar esnek dizi uygulamaları olarak görülebilir.

3.1 Atama anlamı

Bileşik değişkenlerde atama anlamı

- Kopyalama ile Atıf (referans) yaklaşımları.
- Kopyalama: Ayrı bir bellek alanında dizinin aynı içerikle bir kopyası oluşur. Dizi elemanlarının iki kopyası vardır.
- Atıf: Atama işlemi sonrasında iki değişken aynı bellek alanını gösterir. Her elemanın bir kopyası vardır. Birinde yapılan değişiklik diğerine yansır.



- Dilin tasarımına göre atama mantığı belirlenir.
- C yapıları kopya ataması yapılır. Dizilerde atamaya izin verilmez. Göstergeler üzerinden atıf ataması yapılır. Dizi parametreleri atıf olarak geçirilir. C++ nesneleri de yapılar gibi davranır.
- Java ilkel tiplerde kopya ataması yapar. Bütün nesne ve dizi gibi bileşik tiplerde atıf ataması yapar.
- Kopya ataması daha yavaştır.
- Atıf atamasında aynı alanları paylaşması sorun yaratabilir. (x değişince y'nin de değişmesi). Bir değişkenin belleği iade edildiğinde diğeri yanlış bir bellek alanını gösterir (C'de göstergeler)
- Java `copy()` adlı özel bir üye fonksiyonuyla kopya atamasını olanaklı kılar. Bellek iade işlemlerini kendi atık toplama mekanizması ile kontrol ederek sorun çıkmasını engeller.

3.2 Değişken ömrü

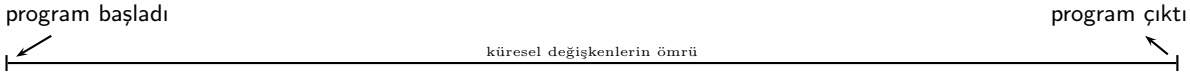
Değişken ömrü

- Değişken ömrü: Bir değişkenin bellekte yer ayrılmasıyla belleğe iade edilmesi arasında geçen süreç.
- 3 farklı değişken ömrü vardır:
  1. Küresel ömür (program çalıştıkça)
  2. Yerel ömür (ilgili program bloğu etkinken)
  3. Yığın ömrü (herhangi bir anda başlayıp bitebilen)
  4. Kalıcı ömür (program çıktıktan sonra yaşayan)

3.2.1 Küresel ömür

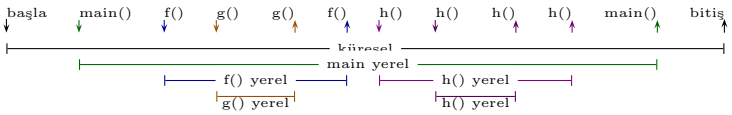
Küresel ömür

- küresel (global) değişkenlerin ömrü program çalıştığı an başlar, program çıkarken biter.
- C'de `main()`'de dahil herhangi bir fonksiyonun içinde tanımlanmamış. Fonksiyonların dışında genel bölgede tanımlanmış değişkenler küresel değişkenlerdir ve küresel ömre sahiptir.

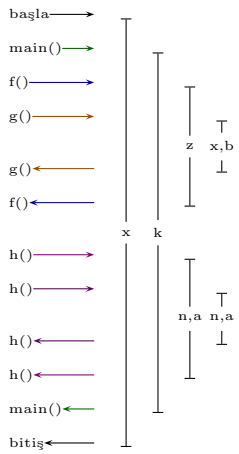


3.2.2 Yerel ömür

Yerel ömür

- Yerel değişkenlerin, yani bir fonksiyon ya da blok içerisinde yaratılan değişkenlerin ve parametre değişkenlerinin ömrü o fonksiyon bloğu çalışmaya başladığı andan çıktığı ana kadardır.
- Bir yerel değişkenin birden çok kopyası aynı anda yaşıyor olabilir. Özyinelemeli bir fonksiyon her çağrılışında ayrı bir yerel değişken kopyası yaratılır.
- 


```
double x;
int h(int n) {
   int a;
   if (n<1) return 1
   else return h(n-1);
}
void g() {
   int x;
   int b;
...
}
int f() {
   double z;
   ...
   g();
   ...
}
int main() {
    double k;
    f();
    ...
    h(1);
    ...;
    return 0;
}
```

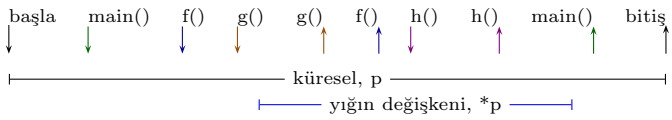


3.2.3 Yığın değişken ömrü

Yığın değişken ömrü

- Yığın (heap) değişkenler: bellekten ayırılması ve belleğe iadesi derleyici tarafından otomatik olarak gerçekleşmez. Kullanıcı harici çağrılarla değişkeni bellekten ayırır ve belleğe iade eder.
- C: `malloc(), free()`, C++: `new, delete`.
- Yığın değişkenlerine genelde gösterge aracılığıyla erişilir `double *p; p=malloc(sizeof(double)); *p=3.4; ... free(p);`
- *`p` ile `*p` farklı değişkenler!* `p` gösterge cinsinde yerel ya da küresel ömürlü bir değişken. `*p` yığın değişkeni.
- yığın değişkenlerinin ömürleri herhangi bir an başlayıp herhangi bir an bitebilir.

```
double *p;
int h() { ...
}
void g() { ...
   p=malloc(sizeof(double));
}
int f() { ...
   g(); ...
}
int main() { ...
    f();    ...
    h();    ...;
    free(p); ...
}
```



3.2.4 Başıboş atıf ve Atık değişkenler

Başıboş atıf

- başıboş atıf (dangling reference): Ömrü dolmuş, belleğe iade edilmiş bir değişkene erişilmeye çalışılması.

```
char *p, *q;

p=malloc(10);
q=p;
...
free(q);
printf("%s",p);
```

```
char *f() {
    char a[]="ali";
    ....
    return a;
}
....
char *p;
p=f();
printf("%s",p);
```

- iki örnekte de `p` iade edilmiş ya da ömrü dolmuş bir değişkeni gösteriyor ve erişince başıboş atıf yapılıyor
- başıboş atıflar bazen işletim sistemi tarafından tolere edilir, bazense "Koruma hatası", "segmentasyon hatası" gibi çalışma zamanı hatalarına yol açar.

Atık değişkenler

- atık değişkenler (garbage): ömürleri devam eden, fakat program içerisinde hiçbir şekilde ulaşılamaz hale gelen değişkenler.

```
char *p, *q;
...
p=malloc(10);
p=q;
...
```

```
void f() {
    char *p;
    p=malloc(10); ...
    return
}
....
f();
```

- İlgili göstergenin değeri değişince ya da ömrü bitince yığın değişkeni erişilmez halde yaşamaya devam ediyor (örneklerde *p)

Atık toplama

- Başıboş atıf ve atıklara çözüm olarak bazı diller yığın değişkenlerinin belleğe iadesini kendileri yapıp kullanıcıdan sorumluluğu alırlar. Bu işleme atık toplama denir. (Java ve Lisp, ML, Haskell gibi fonksiyonel diller)
- `free()` ya da `delete` benzeri çağrı yoktur.
- Her yığın değişken için kaç değişkenin bu alana erişebilecek durumda olduğu tutulur.
- Atıf sayısı 0 olunca atık toplama sistemi yığın değişkenini belleğe iade eder.
- Atık toplama sistemi genelde ayrı bir akım (thread) içinde işlemci boşken etkinleşir.
- atık toplamaya ek olarak: yerel bir değişkene yapılan atıf fonksiyondan dönemez (başıboş atıf sayfası ikinci örnekte `return a` yapılamaz).

3.2.5 Kalıcı değişken ömrü

Kalıcı değişken ömrü

- Ömrü program çıktıktan sonra devam eden değişkenler: dosya, veri tabanı, web servis nesnesi gibi...
- Disk alanı vb. ikincil belleklerde tutulurlar.
- çok az dilde şeffaf bir kalıcı değişken mekanizması vardır. Genelde özel kütüphane çağrılarıyla çalışır C dosyaları: `fopen(), fseek(), fread(), fwrite()`
- nesneye yönelik dillerde serileştirme: nesnenin diske yazılacak ya da ağ üzerinden iletilebilecek bir görüntüsünün çıkarılması.
- nesneler saklanabilir ve program tekrar çalışınca yüklenip kaldığı yerden devam edebilir.

3.3 Komutlar

Komutlar

İfade, değeri olan program parçacığı. Komut değeri olmayan, amacı program durumunu değiştirmek olan program parçacığı. Girdi çıktı, değişkenlerin değerini değiştirme, birden çok işlemi arka arkaya yapma...

1. Atama komutu
2. Altyordam çagrısı
3. Komut grupları
4. Koşullu komutlar
5. Yineleme komutları

3.3.1 Atama

Atama

- C: "`Var = Ifade;`", Pascal "`Var := Ifade;`".
- Soldaki değişkenin yeni değerini sağdaki ifadenin değeri olarak değiştirir.
- `x = x + 1` . Soldaki `x` ile sağdaki farklı!. Soldaki "değişkene atıf (ismi, yeri)" (l-value), sağdaki "değişkenin değeri".
- çoklu atama: `x=y=z=0;`
- paralel atama: (Perl, PHP) `($a,$b) = ($b, $a); ($ad, $soyad, $numara) = ("Onur","Şehitoğlu",55717);`
- işlemli atama: `x += 3; x *= 2;`

3.3.2 Altyordam çağrısı

Altyordam çağrısı

- Altyordam: kullanıcı tanımlı komutlar. Pascal: procedure, C: void dönen fonksiyon
- `void` *`fonksiyonadi`*`(`*`param1`*`, `*`param2`*`, ..., `*`paramn`*`)`
- Kullanım fonksiyonlara benzer fakat çağrı bir ifade pozisyonunda değil komut pozisyonunda (ayrı bir program cümlesinde)

3.3.3 Komut grupları

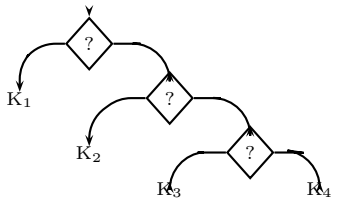
Komut grupları

- Birden çok komutun çalışması için beraber yazılması.
- Art arda çalıştırma: { K_1 ; K_2; ... ; K_n } Bir komut çalıştırılır, çalışmasın bitince sonrasındaki komut çalıştırılır ve bu şekilde devam eder.
- Blok olarak yazılan komut grubu sözdiziminde tek bir komut yerine geçer: "if" blokları, döngü içerikleri
- Eş zamanlı çalıştırma: eş zamanlı paradigma dillerinde bulunur: { K_1 | K_2 | ... | K_n }
- Bütün komutlar aynı zamanda çalışmaya başlar ve paralel olarak çalışır. Çalışmakta olan son komut bitince bütün blok bitmiş olur.
- Çok çekirdekli/işlemcili bilgisayarlar, paralel programlama
- Az sayıda dil tarafından şeffaf olarak desteklenir. Java, C, C++ gibi dillerde akım (thread) kütüphaneleriyle mümkündür.

3.3.4 Koşullu komutlar

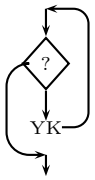
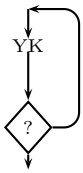
Koşullu komutlar

- Bir koşula göre bir grup komutu çalıştırmak üzere seçmeye yarayan komutlar
- C'de : `if (koşul)` K_1 `else` K_2 `;` `switch (değer) {` `case` D_1 `:` K_1 `;` `case` D_2 `:` K_2 `; ...}`
- `if` komutları içiçe yazılarak ve komut grupları kullanılarak çoklu seçim yapılabilir.
- `switch` benzeri komutlar seçim değerine göre doğrudan karşılık gelen komutu/komutları çalıştırır.



3.3.5 Yineleme komutlar

Yineleme komutları

- Aynı komut ya da komut bloğunun yinelenerek yapılması. Döngü komutları.
- Döngü sınıflandırması: en az tekrar sayısı 0 ya da 1. C: `while (...) { ... }`  C: `do {...} while (...);` 

- Diğer bir sınıflandırma: belirli ve belirsiz sayıda yineleme.
- Belirsiz yinelemeli döngüler: Döngünün kaç yineleme yapacağı döngüye girdikten sonra dahi bilinemez. Döngü içerisindeki herşey yineleme sayısını değiştirebilir.
- C döngüleri belirsiz yinelemeli döngülerdir.
- Belirli yinelemeli döngüler: Döngü başladığı anda kaç adet yineleme yapacağı (hata vb. durumlar dışında) bellidir.
- Pascal `for` döngüsü belirli yinelemeli bir döngüdür. `for i:= k to m do begin .... end;` (m-k+1) yineleme. Pascal döngü indeksinin değiştirilmesine izin vermez.
- Liste ya da küme üzerinde belirli yinelemeler: PHP, Perl, Python, Shell

```
$renkler=array('sari','mavi','yesil','kirmizi','beyaz');
foreach ($renkler as $i) {
    print $i," bir renktir","\n";
}
```

4 Eşleşim

Eşleşim

- Üst düzey dillerin önemli özelliklerinden birisi: Kullanıcıların programdaki varlıklara (değişken, sabit, fonksiyon, tip,...) isim verebilmesi. Bu isimlere belirteç (identifier) diyebiliriz.
- İsimlerin tanımlandıkları yerler ↔ İsimlerin kullanıldıkları yerler.
- Dilin bu eşleşimleri çözmesi gerekli.
- Belirteçler 1 yerde tanımlanır, n yerde kullanılır.
- Eşleşim: Belirteçlerin kullanım yeriyle tanım yeri arasındaki ilişkiyi bulma; her kullanım yerinin hangi tanıma karşılık geldiğini bulma.

Eşleşimlerin çözülmesi için:

1. Belirteç tanım alanlarının belirlenmesi. Blok yapısı var mı? Hangi bloklarda geçerli.
2. Aynı belirteç isminin tekrar kullanılması durumunda ne olacağına karar verilmesi. "C'de bir belirteç aynı alanda birden fazla kullanılamaz. İç içe farklı alanlarda kullanılabilir. İçteki belirteç dışarıdakini saklar."

```
double f,y;
int f() {   × yanlış!
   ...
}
double y; × yanlış!


double y;
int f() {
   double f;    √ tamam.
   int y ;      √ tamam.
}
```

4.1 Ortam

Ortam

- Ortam (çevre, environment): Programın bir noktasında olası eşleşimlerin (kullanılabilir belirteçlerin ve karşılık gelen tanımların) kümesi.
- Örnek:

```
struct Kisi { ... } x;
int f(int a) {
   double y;
   int x;
   ... ①
}
int main() {
   double a;
   ... ②
}
```

O(①)={struct Kisi ↦ tip, x ↦ int, f ↦ fonk, a ↦ int, y ↦ double}[2em] O(②)={struct Kisi ↦ tip, x ↦ struct Kisi, f ↦ fonk, a ↦ double, main ↦ fonk}

4.2 Blok Yapısı

Blok Yapısı

- Program blokları içerdiği belirteçlerin geçerli olduğu alanları belirleyen/sınırlayan bölümlerdir. Değişkenlerin ayrıca ömürlerini de belirler.
- Dile göre farklı blok yapıları olabilir:

  C Fonksiyon tanımları ve blok komutlar ({ ... }) yerel belirteç alanları tanımlar. Her bir kaynak dosya ayrı bir blok oluşturur (birden fazla kaynak dosyanın tek bir programa derlenmesi olası).

  Java Sınıf tanımları ve üye fonksiyon tanımları, blok komutlar yerel alanlar tanımlar. Fonksiyon tanımları iç içe yapılabilir, isim uzayı (namespace).

  Haskell '`let` *tanımlar* `in` *ifade*' blok ifade oluşturur. Yine ' *ifade* `where` *tanımlar*' blok ifade oluşturur. (belirtilen tanımlar sadece ifade için geçerli olur, dışarıdan erişilemez).

- Dilin blok yapısını blokların metin içerisindeki şekli belirler.

4.2.1 Tekli blok yapısı

Tekli blok yapısı

- Bütün program tek bir bloktur. Bütün belirteçlerin küresel bir alanı vardır.
- Cobol dili tekli blok yapısındadır.

```
int x;
int y;
....
....
```

- Uzun bir programda belirteç isimleri birbirine karışabilir. Fonksiyonların ve yerel alanların bulunması sorun.

4.2.2 Düz blok yapısı

Düz blok yapısı

- Programda küresel belirteç alanı ve fonksiyon tanımları gibi tek düzeyde yerel belirteç alanları vardır. İç içe birden fazla düzey tanımlanamaz.
- Fortran ve kısmen C dili düz blok yapısındadır.

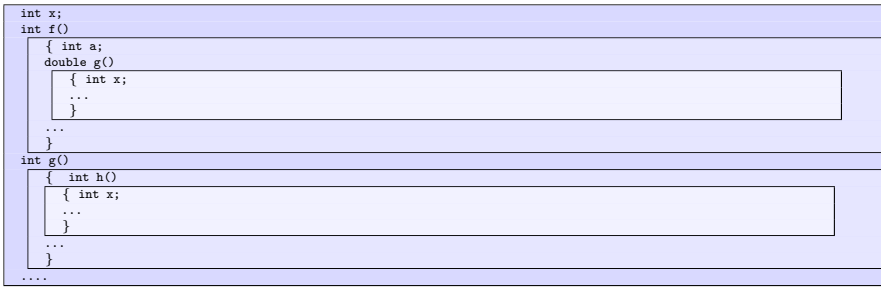
```
int x;
int y;
int f()
  { int a;
    double b;
  ...
  }
int g()
  {  int a;
     double b;
  ...
  }
....
```

4.2.3 İç içe blok yapısı

İç içe blok yapısı

- İçiçe birden çok düzeyde blok tanımlanabilir

- Pascal, Java dili düz blok yapısındadır.



- GCC'nin C eklentilerinde iç içe fonksiyon tanımlarına izin verilir.
- C'nin blok komutları iç içe değişken tanımına izin verir.

4.3 Gizleme

Gizleme

- Yerel blokta tanımlı belirteçler üst düzeydeki bloklarda tanımlı belirteçleri gizler. Bu belirteçler blok içerisinden erişilemez.

```
int x,y;
int f(double x) {
   ...              // parameter x hides global x in f()
}
int g(double a) {
    int y;          // local y hides global y in g()
    double f;       // local f hides global f() in g()
    ...
}
int main() {
    int y;         // local y hides global y in main()
}
```

4.4 Statik ve Dinamik Eşleşim/Alan

Statik ve Dinamik Eşleşim / Alan

Blok sınırlarının derleme sırasında ve çalışma zamanında çözümlenmesine göre iki çeşit eşleşim yöntemi uygulanabilir

1. Statik eşleşim, statik alan
2. Dinamik eşleşim, dinamik alan

- İlkinde programın şekline, blokların sırasına ve yerleşimine göre belirteçlerin alanları belirlenir ve eşleşimi yapılır.
- İkincisinde bloklar etkinleştirildikçe (fonksiyon çağırılınca) içindeki tanımlar etkinleşir ve çıkıncaya kadar etkin kalır. Ortam etkin olan fonksiyonların/bloklara göre değişir.

4.4.1 Statik eşleşim

Statik eşleşim

- Programın şekline bakılır. Ortam programın konumuna göre değişir (lexical scope)
- Birçok dil statik eşleşim yapar (C, Haskell, Pascal, Java, ...)

```
int x=1,y=2;
int f(int y) {
     y=x+y;
     return x+y;
}
int g(int a) {
    int x;
    y=x+x;    x=x+y;    y=f(x);
    return x;
}
int main() {
    int y=0;    int a=10;
    x=a+y;    y=x+a;    a=f(a);    a=g(a);
    return 0;
}
```

4.4.2 Dinamik eşleşim

Dinamik eşleşim

- Çalışma zamanında çağırılan her fonksiyon kendi tanımlarını ortama ekler. Çıkarken de siler. Fonksiyonun nerede olduğu değil çağrıldığı anda etkin olan fonksiyonlar önemlidir.
- Lisp ve bazı betik diller dinamik eşleşim yapar.

```
int x=1,y=2;
int f(int y) {
     y=x+y;
     return x+y;
}
int g(int a) {
    int x;
    y=x+x;    x=x+y;    y=f(x);
    return x;
}
int main() {
    int y=0;    int a=10;
    x=a+y;    y=x+a;    a=f(a);    a=g(a);
    return 0;
}
```

4.5 Tanımlar

Tanımlar

- Yeni adlandırmalar ve tanımlar
- Ardışık tanımlar
- Eş ortamlı tanımlar
- Özyinelemeli tanımlar
- Eş ortamlı öz yinelemeli tanımlar
- Blok komutlar
- Blok ifadeler.

4.5.1 Yeni adlandırmalar ve tanımlar

Yeni adlandırmalar ve tanımlar

- Yeni adlandırma: Var olan bir tanımı yeni bir adla erişilebilir kılmak.
- Tanım: Tamamen yeni bir eşleşim yaratmak.
- C'de : `struct Kisi` bir tanımdır. `typedef struct Kisi kisitipi` bir yeniden adlandırmadır.
- C++'da: `double x` bir tanımdır. `double &y=x;` bir yeniden adlandırmadır.
- Tanıma karşılık gelen bir varlık yaratma ya da yaratmama farkı. İngilizce'de "definition" ve "declaration". Genelde birbirleri yerine kullanılır.

4.5.2 Ardışık tanımlar

Ardışık tanımlar

- $\texttt{T}_1$; $\texttt{T}_2$; ... ; $\texttt{T}_n$
- Her tanım bir sonraki satırdan itibaren geçerlidir. $\texttt{T}_1$, $\texttt{T}_2$ ve sonrasında kullanılabilir, $\texttt{T}_2$, $\texttt{T}_3$ ve sonrasında kullanılabilir, ...
- Önceki satırlarda tanım erişilemez.
- Çogu programlama dili bu şekilde ardışık tanımlara izin verir.

4.5.3 Eş ortamlı tanımlar

Eş ortamlı tanımlar

- Baş; $\texttt{T}_1$ ve $\texttt{T}_2$ ve ... ve $\texttt{T}_n$; Son
- Her tanım bütün tanım grubundan önceki ortamda değerlendirilir. Ancak hepsi bittikten sonra bütün tanımlar kullanılabilir. $\texttt{T}_1$,... $\texttt{T}_n$, Baş noktasındaki ortamda tanımlanır. Son noktasında hepsi erişilebilir hale gelir.
- ML gibi bazı dillerde tanımlar isteğe göre bu şekilde yapılabilir.

4.5.4 Özyinelemeli tanımlar

Özyinelemeli tanımlar

- Tanım:Isim = Gövdesi
- Tanımın gövdesi tanımın kendisini kullanabilir. Yani tanım gövde ortamında mevcuttur.
- C fonksiyonları ve tip tanımları özyinelemelidir. Genelde değişken tanımları özyinelemeli değildir. ML gibi bazı diller özyinelemeli olmayan fonksiyonlara izin verir.

4.5.5 Eş ortamlı öz yinelemeli tanımlar

Eş ortamlı öz yinelemeli tanımlar

- Bütün tanımlar (sıralarına bakılmaksızın) bütün tanımların gövdesinden erişilebilir.
- Haskell'deki bütün tanımlar böyledir.
- Bütün tanımlar karşılıklı özyineleme yapabilirler.
- ML gibi diller bu şekilde tanımlar yapmaya izin verir.
- C'de prototip tanımlarla kısmen sağlanabilir.

4.5.6 Blok ifadeler

Blok ifadeler

- Bir ifadenin yerel bir ortamda, yani sadece o ifade için geçerli olan bir grup tanım ile hesaplanmasını sağlar. İfade bloğu dışında yerel tanımlar geçersiz olur.
- Haskell'de `let` T_1`;` T_2`; ... ;` T_n `in` ***Ifade*** şeklinde ya da ***Ifade*** `where` T_1`;` T_2`; ... ;` T_n şeklinde belirtilebilir.
-

```
x=5
t=(let xkare=x*x
       faktoriyel n = if n<2 then 1 else n*faktoriyel (n-1)
       xfakt = faktoriyel x
   in  (xkare+1)*xfakt/(xfakt*xkare))+2
```

- Gizleme blok ifadelerde beklenen şekilde çalışır:

```
x=5 ; y=6 ; z = 3
t=let x=1
  in let y=2
     in   x+y+z
{-- t burada 1+2+3 olur. yerel x ve y degerleri usteki --}
{-- tanimlari gizler --}
```

4.5.7 Blok komutlar

Blok komutlar

- Blok ifadelerdeki gibi yerel bir ortamda yapılan tanımlar sadece blok içerisinde geçerli olur. Blok içerisindeki komutlar bu ortamda çalıştırılır. Blok dışında tanımlar kaybolur.
-

```
int x=3,i=2;
x+=i;
while (x>i) {
    int i=0;
    ...
    i++;
}
/* i burada yeniden 2 degerine sahip */
```

5 Yüksek Derece Fonksiyonlar

Giriş

- Matematik: $(f \circ g)(x) = f(g(x))$, $(g \circ f)(x) = g(f(x))$
- "$\circ$" : iki fonksiyon alıp. yeni bir bileşik fonksiyon verir. İki fonksiyon alıp yeni bir fonksiyon dönen bir fonksiyon olarak düşünülebilir mi?
- Haskell'de

```
f x = x+x
g x = x*x
bilesik fonk1 fonk2 x = fonk1 (fonk2 x)
t = bilesik f g
u = bilesik g f
```

- `t 3 = (3*3)+(3*3) = 18 u 3 = (3+3)*(3+3) = 36`
- `bilesik`: $(\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma$
- "`bilesik`" fonksiyonu parametre olarak iki fonksiyon ve bir değer alıp yeni bir değer dönen bir fonksiyondur.
- Parametre olarak bir ya da daha fazla fonksiyon alan fonksiyonlara Yüksek Dereceli Fonksiyon denir.
- Fonksiyonel dillerde birçok işlem veri yapıları üzerinde temel işlemlerin tekrarı, fonksiyonların uygulaması şeklinde.
- Fonksiyonların birinci derece değer olması (parametre olarak kullanılabilmesi ve değer olarak dönülebilmesi) varolan fonksiyonları kullanarak yeni fonksiyonlar yazılmasını kolaylaştırıyor.

5.1 Fonksiyonlar

5.1.1 Çevir

Fonksiyonlar/Çevir (Map)

```
kare x = x*x
gun no = case no of 1 -> "pzt" ; 2 -> "sal" ; 3 -> "crs";
        4 -> "prs" ; 5 -> "cum" ; 6 -> "cts" ; 7 -> "paz"
cevir fonk [] = []
cevir fonk (el:devami) = (fonk el):(cevir fonk devami)
----
cevir kare [1,3,4,6]
[1,9,6,36]
cevir gun [1,3,4,6]
["pzt","crs","prs","cts"]
```

- `cevir`:$(\alpha \to \beta) \to [\alpha] \to [\beta]$
- Bir fonksiyon ve bir liste alır. Fonksiyonu bütün elemanlara teker teker uygulayarak sonuçların listesini döner.
- Haskell kütüphanesinde `map` adıyla mevcut.

5.1.2 Filtre

Fonksiyonlar/Filtre (Filter)

```
ciftmi x = if mod x 2 == 0 then True else False
buyukmu x = x>5

filtre fonk [] = []
filtre fonk (el:devami) = if fonk el then
                              el:(filtre fonk devami)
                          else  (filtre fonk devami)
----
filtre ciftmi [1,2,3,4,5,6,7]
[2,4,6]
filtre buyukmu [1,2,3,4,5,6,7]
[6,7]
```

- `filtre`:$(\alpha \to Boot) \to [\alpha] \to [\alpha]$
- Mantıksal değer dönen bir fonksiyon ve bir liste alır. Fonksiyonu bütün elemanlara teker teker uygular, sadece doğru sonuç dönen elemanların listesini döner.
- Haskell kütüphanesinde `filter` adıyla mevcut.

5.1.3 İndirge

Fonksiyonlar/İndirge (Reduce)

```
topla x y = x+y
carp x y = x*y

indirge fonk b [] = b
indirge fonk b (el:devami) = fonk el (indirge fonk b devami)
----
indirge topla 0 [1,2,3,4]
10                              // 1+2+3+4+0
indirge carp 1 [1,2,3,4]
24                              // 1*2*3*4*1
```

- `indirge`:$(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$
- İkili bir fonksiyon, bir liste ve bir başlangıç elemanı alır. Fonksiyonu bütün elemanlara sağdan sola uygulayarak, tek bir sonuç döner. $indirge\ f\ b\ [a_1, a_2, ..., a_n] = f\ a_1\ (f\ a_2\ (....\ (f\ a_n\ b)))$
- Haskell kütüphanesinde `foldr` adıyla mevcut.
- Bir sayı listesinin elemanlarının toplamı: `listetoplam = indirge topla 0`
- Bir sayı listesinin elemanlarının çarpımı: `listecarpim = indirge carp 1`
- Bir listenin karelerinin toplamı: `karetoplam x = indirge topla 0 (cevir kare x)`

5.1.4 İndirge

Fonksiyonlar/İndirge (soldan sağa)

```
cikart x y = x - y
solindirge fonk b [] = b
solindirge fonk b (el:devami) =
          solindirge fonk (fonk b el) devami
----
indirge cikart 0 [1,2,3,4]
-2                              // 1-(2-(3-(4-0)))
solindirge cikart 0 [1,2,3,4]
-10                             // ((((0-1)-2)-3)-4)
```

- `indirge`:$(\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$
- İndirgeme işlemi, soldan sağa: $solindirge\ f\ b\ [a_1, a_2, ..., a_n] = f\ (f\ (f\ ...(f\ b\ a_1)\ a_2\ ...))\ a_n$
- Haskell kütüphanesinde `foldl` adıyla mevcut.

5.1.5 Tekrar

Fonksiyonlar/Tekrar

```
iki x = 2*x

tekrar fonk b 0 = b
tekrar fonk b n = fonk (tekrar fonk b (n-1))
----
tekrar iki 1 4
16                              // iki (iki ( iki (iki 1))
tekrar kare 3 3
6561                                // kare (kare (kare 3))
```

- `tekrar`:$(\alpha \to \alpha) \to \alpha \to int \to \alpha$
- Aynı fonksiyonu belirtilen sayıda arka arkaya verilen parametreye uygular. $tekrar\ f\ b\ n = f^n\ b = \underbrace{f\ (f\ (f\ ...(f\ b))}_{n}$

5.1.6 Değerle tekrar

Fonksiyonlar/Değerle tekrar (for)

```
for fonk b m n =
      if m>n then b
      else for fonk (fonk b m) (m+1) n

----
for topla 0 1 4
10              // topla (topla (topla (topla 0 1) 2) 3) 4
for carp 1 1 4
24              // carp (carp (carp (carp 1 1) 2) 3) 4
```

- `for`:$(\alpha \to int \to \alpha) \to \alpha \to int \to int \to \alpha$
- İkili bir fonksiyonu verilen aralıktaki bir grup tamsayıya sırasıyla uygular. $for\ f\ b\ m\ n = f(f\ (f\ (f\ (f\ b\ m)\ (m+1))\ (m+2))\ ...)\ n$
- çarpma (toplama kullanarak): `carp x = tekrar (topla x) x`
- üst alma işlemi (Haskell $\hat{}$): `guc x = tekrar (carp x) x`
- 1'den n'e kadar olan sayıların toplamı: `dizitoplam = for topla 0 1`
- Faktoriyel işlemi: `faktoriyel = for carp 1 1`

5.2 C'de Yüksek Dereceden Fonksiyonlar

C'de Yüksek Dereceden Fonksiyonlar

C'de fonksiyon göstergeleri kullanarak benzeri tanımlar yapmak olası. Örnek: bsearch() ve qsort() fonksiyonları.

```
typedef struct KISI { char isim[30]; int no;} Kisi;
int numarakarsilastir(void *a, void *b) {
    Kisi *ka=(Kisi *)a; Kisi *kb=(Kisi *)b;
    return ka->no - kb->no;
}
int isimkarsilastir(void *a, void *b) {
    Kisi *ka=(Kisi *)a; Kisi *kb=(Kisi *)b;
    return strncmp(ka->isim,kb->isim,30);
}
int main() {       int i;
    Kisi liste[]={{"veli",4},{"ali",12},{"ayse",8},
                  {"osman",6},{"fatma",1},{"mehmet",3}};
    qsort(liste,6,sizeof(Kisi),numarakarsilastir);
    ...
    qsort(liste,6,sizeof(Kisi),isimkarsilastir);
    ...
}
```

5.3 İleri örnekler

5.3.1 Fibonacci

Fibonacci

Fibonacci sayıları: 1 1 2 3 5 8 13 21 .. $fib(0) = 1$; $fib(1) = 1$; $fib(n) = fib(n-1) + fib(n-2)$

```
fib n = let f (x,y) = (y,x+y)
            (a,b) = tekrar f (0,1) n
        in b
----
fib 5       // f(f(f(f(0,1))))
8           //(0,1)->(1,1)->(1,2)->(2,3)->(3,5)->(5,8)
```

5.3.2 Sıralama

Sıralama

Quicksort (hızlı sıralama):

1. Listenin ilk elamanını x. Geri kalanlar r.
2. r içinde x'den küçük olanları sırala ve x'in soluna koy
3. r içinde x'den büyük eşit olanları sırala ve x'in sağına koy.

```
buyuktur x y = x>y
kucukesit x y = x<=y
sort [] = []
sort (x:xs) = (sort (filter (buyuktur x) xs)) ++
              (x: (sort (filter (kucukesit x) xs)))
---
sort [5,3,7,8,9,3,2,6,1]
[1,2,3,3,5,6,7,8,9]
```

5.3.3 Listenin tersi

Listenin tersi

- Listenin tersini almak için:
  - Listenin ilk elamanını x. Geri kalanlar r.
  - r'yi ters çevir. x'i sonuna ekle.

  Her seferinde listenin sonuna x'i eklemek için zaman kaybediyor.
- Hızlı versiyon, fazladan parametre:
  - Listenin ilk elamanını al son parametrenin başına ekle.
  - listenin geri kalanını yeni parametre ile tekrar gönder.

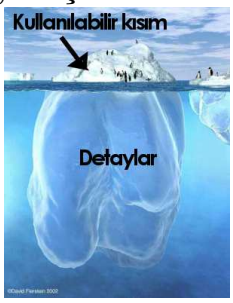
  Her seferinde listenin basina eklediği için hızlı.

```
ters1 [] = []
ters1 (x:xs) = (ters1 xs) ++ [x]

tersa [] x = x
tersa (x:devami) y = tersa devami (x:y)
ters2 x = tersa x []
---
ters1 [1..10000]        // yavas
ters2 [1..10000]        // hizli
```

6 Soyutlaştırma

Soyutlaştırma



- Buzdağı: Detaylar altta, kullanılabilir kısım üstte. Üstünde yaşayan hayvanlar detayla ilgilenmiyor.
- Kullanıcı: "nasıl kullanırım", Geliştirici: "nasıl çalışır?"
- Kullanıcı: "ne yapar?", Geliştirici: "nasıl yapar?"
- Soyutlaştırma: Geliştirilen programın/tasarımın detaylarını saklayarak bir mekanizma ile kolay kullanılır ve tekrar tekrar kullanılır hale getirmek.
- Program parçalarının kullanımı ve uygulanmasının ayırılması.
- Büyük ölçekli programlamanın vazgeçilmez gereksinimi.
- Genel olarak tasarımın söz konusu olduğu birçok branşta geçerli.
- Örnek: radyolarda sadece belirlenen frekanstaki dalgaların geçişini sağlayan bir devre elemanı vardır (tuner). Bu elemanın ayarını sağlayan potansiyometre radyonun dışında bir düğme olarak kontrol edilir. Radyonun ayar düğmesi "tuner" ayarı üzerine bir soyutlaştırmadır.
- Programlama dilleri makina dili üzerine bir soyutlaştırmadır.
- Grafik arayüzlü programlar karmaşık program mantığının üzerine bir soyutlaştırmadır.
- ...

Amaç

- Detaylar kafa karıştırıcı
- Detaylarda Hata yapması kolay
- Detayların tekrarlanması zor ve hata arttırıcı etmen
- Soyutlaştırma ile *Bir kere tanımla, istediğin kadar kullan!*
- Programın tekrar kullanılırlığı en önemli amaç
- Soyutlaştırmada parametre kullanarak etkinliği arttırmak mümkün

6.0.4 Fonksiyon ve altyordam soyutlaştırmaları

Fonksiyon ve altyordam soyutlaştırmaları

- Bir ifadenin hesaplanması detay (algoritma, değişken alanı kullanımı vs.)
- Fonksiyon çağrısı o detayı kullanma şekli.
- Fonksiyonlar ifadeler üzerinde bir soyutlaştırma
- `void` dönen fonksiyonlar ya da bazı dillerde `procedure` tanımları.
- Değerleri yok fakat bir grup komut detayından oluşur.
- Altyordamlar komutlar üzerinde bir soyutlaştırma

6.0.5 Seçici soyutlaştırması

Seçici soyutlaştırması

- Başka soyutlaştırmalar mümkün mü?
- dizilerde: `int a[10][20]; a[i]=a[i]+1;`
- `[..]` operatörü bir seçici. Dizinin elemanlarını seçiyor.
- Kullanıcı kendi veri yapısı üzerine seçici tanımlayabilir mi?
- Bağlı listelerde bir seçici (C++)

```
int & get(Liste *p, int el) {   /* bagli liste */
    int i;
    for (i=1;i<el;i++) {
        p = p->next;        /* sonraki elemana git */
    }
    return p->data;
}
get(bas,i) = get(bas,2) + 1; ...
```

- C++'ta nesne tanımları için `[..]` operatörünü kullanmak da mümkün.

6.0.6 Genel soyutlama

Genel soyutlama

- Aynı tür tanımların farklı veri tiplerine uygulanması mümkün.
- Tanımlar üzerinde soyutlaştırma. Bir fonksiyon ya da nesne tanımı farklı tiplere parametre aracılığıyla uyarlanabilir.

```
template <class T>
  class Liste {
        T icerik;
        Liste *sonraki;
  public:
        Liste() { sonraki=NULL };
        void ekle(T el) { ... };
        T al(int n) { ...};
  };
template <class U>
  void swap(U &a, U &b) { U tmp; tmp=a; a=b; b=tmp; }
...
Liste<int> a; Liste<double> b; Liste<Kisi> c;
int t,x; double v,y; Kisi z,w;
swap(t,x); swap(v,y); swap(z,w);
```

6.1 Soyutlaştırma prensibi

Soyutlaştırma prensibi

- Herhangi bir programlama dili varlığı hesaplama içeriyorsa bunun üzerine bir soyutlaştırma yapmak mümkündür

Varlık	→	**Soyutlaştırma**
İfade	→	Fonksiyon
Komut	→	Altyordam
Seçici	→	Seçici fonksiyon tanımı
Tanım	→	Genel soyutlaştırma

6.2 Parametreler

Parametreler

- Bir defa tanımla çok defa kullan prensibinin uygulanabilmesi için soyutlamanın her kullanımda farklı amaç ve davranışa izin vermesi gerekir.
- **Tanım kısmı:** `soyutlaştırma_ismi(Tp`$_1$`, Tp`$_2$`, ..., Tp`$_n$`)` **Kullanım kısmı:** `soylaştırma_ismi(Ap`$_1$`, Ap`$_2$`, ..., Ap`$_n$`)`
- Tanım parametreleri: belirteç ya da belirteç içeren yapılar (fonksiyonel dillerde)
- Asıl parametreler: soyutlama ve parametrenin türüne göre ifade ya da belirteç
- *Soru:* Asıl parametreler ve tanım parametreleri nasıl iletişim kurar?
- Programlama dili tasarımının yanıtlaması gerekir.
- Parametre iletme mekanizmaları

6.3 Parametre iletme mekanizmaları

Parametre iletme mekanizmaları

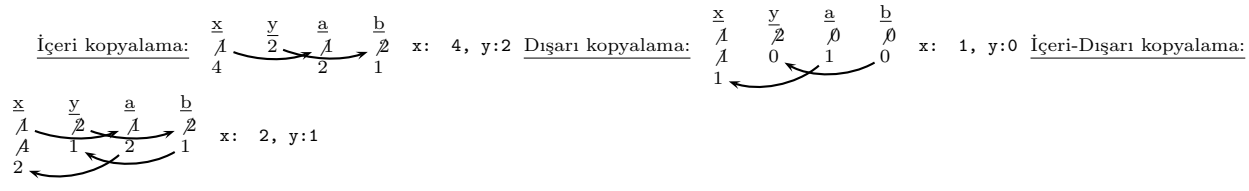
Programlama dili bir ya da birden fazla mekanizma destekliyebilir. 3 temel yöntem kullanılabilir

1. Kopyalama ile iletme (değişken değer atama temelli)
2. Eşleşim ile iletme (eşleşim temelli)
3. İsim olarak iletme (yerine koyma temelli)

6.3.1 Kopyalama ile iletme

- Fonksiyon ve altyordam soyutlaştırmalarında asıl parametre ve tanım parametresi arasında atama (assignment) işlemi. 3 çeşit yapılabilir:
- İçeri kopyalama: Fonksiyon çalıştığında: $Tp_i \leftarrow Ap_i$
- Dışarı kopyalama: Fonksiyon çıkarken: $Ap_i \leftarrow Tp_i$
- İçeri-dışarı kopyalama: Fonksiyon çalıştığında: $Tp_i \leftarrow Ap_i$, ve Fonksiyon çıkarken: $Ap_i \leftarrow Tp_i$
- C dili İçeri kopyalama mekanizması kullanır. Bu mekanizma aynı zamanda Değer ile iletme (pass by value) olarak da adlandırılır.

```
int x=1, y=2;
void f(int a, int b) {
    x += a+b;
    a++;
    b=a/2;
}
int main() {
    f(x,y);
    printf("x:%d␣,␣y:%d\n",x,y);
    return 0;
}
```



6.3.2 Eşleşim ile iletme

- Tanım parametresindeki değişkenin/belirtecin asıl parametredeki değer, değişken ya da belirteçle eşleşmesi.
- Tek bir değer ya da değişken var. Farklı isimlerle anılıyor.
- Sabit eşleşimi: Tanım parametresi fonksiyon içinde sabit. Değeri asıl parametrenin değeri ile eşleşir. Fonksiyonel diller ve Haskell bu mekanizmayı kullanır.
- Değişken eşleşimi: Fonksiyon çalıştığı sürece tanım değişkeni asıl değişkenin adresiyle eşleşir. Aynı bellek alanı tanım parametresi ismiyle anılır. Atıf ile iletim (pass by reference) olarak da anılır.
- Soyutlaştırmaya göre tip vb diğer varlıklar da benzeri şekilde eşleşim ile iletilir.

```
int x=1, y=2;
void f(int a, int b) {
    x += a+b;
    a++;
    b=a/2;
}
int main() {
    f(x,y);
    printf("x:%d␣,␣y:%d\n",x,y);
    return 0;
}
```

Değişken eşleşimi:

f():a / x	f():b / y
~~1~~	~~2~~
~~4~~	2
5	

`x: 5, y:2`

6.3.3 Yer değiştirme ile iletme

Yer değiştirme ile iletme

- Tanım parametresinin bütün göründüğü yerlere asıl parametre deyimi konur ve fonksiyon/altyordam bu hal ile çalıştırılır.
- C makroları bu mekanizma ile çalışır (ön-derleyici tarafından)
- Programlama dillerinin kuramsal analizi için kullanılır. Normal değerlendirme sırası olarak da adlandırılır.
- Örnek (Haskell benzeri)

```
f x y = if (x<12) then x*x+y*y+x
                  else x+x*x
```

Değerlendirme: `f (3*12+7) (24+16*3)` $\mapsto$ `if ((3*12+7)<12) then (3*12+7)*(3*12+7)+(24+16*3)*(24+16*3)+(3*12+7) else (3*12+7)+(3*12+7)*`
$\overset{*}{\mapsto}$ `if (43<12) then ...` $\mapsto$ `if (false) then ...` $\mapsto$ `(3*12+7)+(3*12+7)*(3*12+7)` $\overset{*}{\mapsto}$ `(3*12+7)+43*(3*12+7)` $\mapsto$ `...` $\mapsto$ `1892`

- Haskell Tembel değerlendirme sırası uygular. Diğer hemen hemen bütün diller Hırslı değerlendirme sırası (önce asıl parametreler değerlendirilir sonra iletilir) uygular.
- Değerlendirme (Hırslı sıra): `f (3*12+7) (24+16*3)` $\mapsto$ `f (36+7) (24+16*3)` $\overset{*}{\mapsto}$ `f 43 72` $\mapsto$ `if (43<12) then 43*43+72*72+43`
`else 43+43*43` $\mapsto$ `if (false) then ...` $\mapsto$ `43+43*43` $\overset{*}{\mapsto}$ `1892`
- Hırslı sıra çok daha hızlı olduğu için diller hırslı değerlendirme sırası ve uygun bir parametre iletme mekanizması uygular.
- Tembel değerlendirme hırslı sırayla aynı hızda ama normal değerlendirme sırasına benzer nitelikte değerlendirme sağlar. (farkı görmek için `f 15 (2/0)` ifadesini Haskell ve dilediğiniz dilde deneyin)

7 Modülleştirme

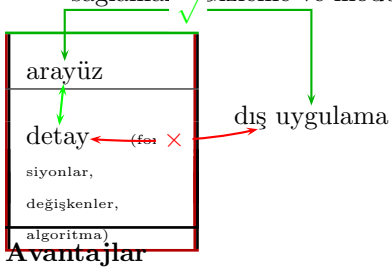
Modülleştirme

Karmaşıklığı yönetmek için: tekrar kullanılabilir kod ve soyutlaştırma:

50 satır	soyutlaştırmaya gerek yok, he
500 satır	fonksiyon/altyordam soyutlaşt
5,000 satır	fonksiyon grupları modüllerd bağlanmalı
500,000 satır	üst düzey soyutlaştırma/mod kullanıma göre tasarlanmış

Modülleştirme ve paketleme

- Kendi içinde tutarlı, bütün ve bağımsız bir fonksiyon/değişken tanımları kümesi yapmak. (Paketleme)
- Bu kümeye erişimin sadece belirli bir grup arayüz fonksiyon/değişken tanımları üzerinden yapılmasını sağlamal (Gizleme ve modülleştirme) (encapsulation)[2em]



Avantajlar

- Büyük hacimdeki detaylar sadece arayüz tanımlarına indirgenir (Kolay geliştirme/idame)
- Aynı arayüzü kullanarak farklı uygulamalar aynı modülleri kullanabilir (Tekrar kullanılabilirlik)
- Kodu yapıtaşları ile geliştirmeyi olanaklı kılar (Lego gibi) (Kolay geliştirme/idame)
- Detaylar sonradan değişse bile arayüz değişmedikçe program değişmez. (Kolay geliştirme/idame)
- Geliştirilen modül sonraki projelerde kullanılabilir (Tekrar kullanılabilirlik)

7.1 Paketler

Paketler

- Bir grup tanımı derli toplu hale getirip aynı erişim alanı altına toplayıp gruplamak.
- C dilinde farklı dosyalara yazarak kısmen yapmak mümkün.
- C++

```
namespace Trig {
    const double pi=3.14159265358979;
    double sin(double x) { ... }
    double cos(double x) { ... }
    double tan(double x) { ... }
    double atan(double x) { ... }
   ...
};
```

- `Trig::sin(Trig::pi/2+x)+Trig::cos(x)`
- C++: (::) Alan (scope) operatörü.
- Belirteç çakışmasını engeller. `Liste::ekle(...)` ve `Agac::ekle(...)` isimler çakışmıyor.

7.2 Gizleme

Gizleme

- Bir grup fonksiyon/değişken içeride gizli, bir grup fonksiyon/değişken arayüz. Paket içinde soyutlaştırma.

```
double taylorseri(double);
double sin(double x);
double pi=3.14159265358979;
double randombesleme;
double cos(double x);
double duzeltme(double x);
```

```
{-- sadece sin, pi ve cos'a erisilebilir --}
module Trig(sin,pi,cos) where
  taylorseri x = ...
  sin x = ...
  pi=3.14159265358979
  randombesleme= ...
  cos x = ...
  duzeltme x = ...
```

7.3 Soyut veri tipleri

Soyut veri tipleri

- Veri tipinin iç yapısı gizlenir ve sadece arayüz fonksiyonları ile erişim sağlanır.
- Örnek: paydalı kesirler: 3/4 , 2/5, 19/3 `data Paydali = Pay (Integer,Integer)` `x = Pay (3,4)` `topla (Pay(a,b)) (Pay(c,d)) = Pay (a*d+b*c,b*d)`
- *1. Geçersiz değer?* `Pay (3,0)`
- *2. Tek değer farklı gösterim?* `Pay (2,4) = Pay (1,2) = Pay(3,6)`
- Çözüm: Kullanıcının keyfi değer yaratmasını engellemek.

Soyut veri tiplerinde amaç veri bütünlüğü bozulmadan tanımlı veri tiplerinin dahili veri tipleri gibi kullanılabilmesidir.

```
module Payda(Paydali,pay,topla,cikart,carp,bol) where
  data Paydali = Pay (Integer,Integer)
  pay (x,y) = sadelestir (Pay(x,y))
  topla (Pay(a,b)) (Pay(c,d)) = pay (a*d+b*c,b*d)
  cikart(Pay(a,b)) (Pay(c,d)) = pay (a*d-b*c,b*d)
  carp(Pay(a,b)) (Pay(c,d)) = pay (a*c,b*d)
  bol (Pay(a,b)) (Pay(c,d)) = pay (a*d,b*c)
  obeb x y = if (x==0) then y
             else if (y==0) then x
             else if (x<y) then obeb x (y-x)
             else obeb y (x-y)
  sadelestir (Pay(x,y)) = if y==0 then Pay(div x y,1)
                          else let a=obeb x y
                               in Pay(div x a, div y a)
```

İlk değer? İlk değeri sağlamak için yapılandırıcı fonksiyon gerekli. `Pay (x,y)` yerine `Pay(x,y)`

7.4 Nesne ve Sınıf

7.4.1 Nesne

Nesne

- Değişkenler içerebilen, değişkenlerin gizlenerek erişimin sadece arayüz fonksiyonlarıyla yapıldığı paketler.
- Değişkenler güncellenebilen, değeri değişebilen değişkenler.

- Arayüz fonksiyonları veri bütünlüğünü ve soyutlaştırmayı gerçekleştirir.
- Yazılım içindeki varlıkların (sunucu, personel kaydı, dosya vs.) nesne olarak modellenmesi ve fonksiyonlarla haberleşmesi.
- Örnek (geçersiz sözdizimi! hayali C++)

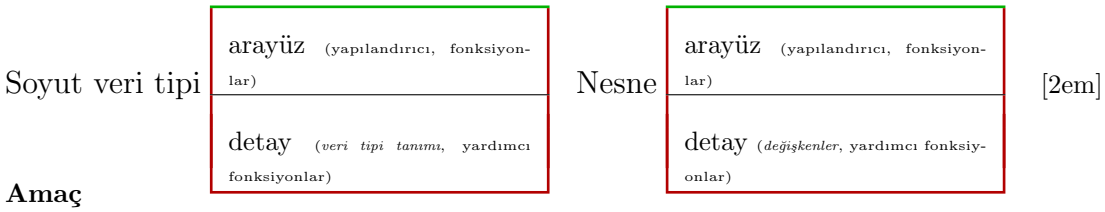
```
namespace Sayac {
private:   int sayac=0;
public:    int al() { return sayac;}
public:    void arttir() { sayac=sayac+1; }
};
Sayac::al()        Sayac::arttir()
```

7.4.2 Sınıf

Sınıf

- Aynı tür nesnelerin kümesi sınıf oluşturur.
- Bir nesne, ait olduğu sınıfın bir örneğidir. (tek bir sayaç yerine bir sayaç türü)
- Sınıflar soyut veri tiplerine benzer amaçlarla kullanılır.
- Bazı diller nesne/sınıf ayrımını yapar.
- C++ sınıf tanımı (geçerli sözdizimi):

```
class Sayac {
private:   int sayac;
public:    Sayac() { sayac=0; }
           int al() { return sayac;}
           void arttir() { sayac=sayac+1; }
} insanlar,tasitlar;
insanlar.arttir(); tasitlar.arttir();
insanlar.al(); tasitlar.al();
```



Amaç

- veri bütünlüğünü korumak,
- soyutlaştırma,
- tekrar kullanılabilir programlar.

8 Tip sistemleri

Tip sistemleri

Dilin tasarımında tipler konusunda verilmesi gereken kararlar:

- Dilin tasarımında tipler konusunda verilmesi gereken kararlar:
- tek türlü ile çok türlü değerler?
- aşırı yükleme var mı?
- otomatik tip çevirimi var mı, nasıl?

8.1 Çok türlülük

Çok türlülük (Polymorphism)

- Tek türlü tipler: her değerin sadece tek bir tipi olabilir. Fonksiyonlar tek tip üzerinde işlem yaparlar. C dili ve birçok dil tek türlüdür.
- Çok türlü tipler: değerler birden fazla tipe ait olabilir. Fonksiyonlar birden fazla tip üzerinde aynı işlemi yapabilir.
- `ayni x = x` fonksiyonu. tipi: $\alpha \to \alpha$ `ayni 4` : 4, `ayni "ali"` : "ali" , `ayni (5,"abc")` : (5,"abc") $int \to int$, $String \to String$, $int \times String \to int \times String$
- `bilesim f g x = f (g x)` .fonksiyonu tipi: $(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$ `bilesim kare ikikati 3` : 36, $(int \to int) \to (int \to int) \to int \to int$. `bilesim listetopla ters [1,2,3,4]` : 10 $([int] \to int) \to ([int] \to [int]) \to [int] \to int$
- `filtre f [] = []  filtre f (x:r) = if (f x) then x:(filtre f r) else (filtre r)` $(\alpha \to Bool) \to [\alpha] \to [\alpha]$ `filtre ((<) 3) [1,2,3,4,5,6]` : [4,5,6] $(int \to Bool) \to [int] \to [int]$ `filtre ayni [True, False, True, False]` : [True,True] $(Bool \to Bool) \to [Bool] \to [Bool]$
- İşlemler aynı, tipler farklı
- Tip değişkenleri olan tipler: çoklu tipler
- Fonksiyonel diller genelde çoktürlülüğü destekler.
- Nesneye yönelik dillerde nesne mirası üzerinden çoktürlülük mevcuttur.

8.2 Aşırı yükleme

Aşırı yükleme (overloading)

- Aşırı yükleme ≠ Çok türlülük
- Aşırı yükleme: Aynı belirteç isminin farklı varlıkları tanımlaması.
- Örn: Aynı alanda iki ayrı fonksiyon, iki ayrı tip, aynı isim.
- Çok türlü fonksiyon: tek fonksiyon, farklı tipleri işleyebiliyor.
- C++'da fonksiyonlar ve operatörler aşırı yüklenebilir.

```
typedef struct Karm { double x, y; } Karmasik;
double carp(double a, double b) { return a*b; }
Karmasik carp(Karmasik s, Karmasik u) {
    Karmasik t;
    t.x = s.x*u.x - s.y*u.y;
    t.y = s.x*u.y + s.y*u.x;
    return t;
}
Karmasik a,b; double x,y; ... ; a=carp(a,b) ; x=carp(y,2.1);
```

- Eşleşim problemi daha karmaşık: *sadece isme göre değil isim ve tipe göre*
- Fonksiyon tipi: `isim : parametreler` → *sonuç*
- Bağlama göre aşırı yükleme : fonksiyonun ismi, parametrelerinin tipi ve dönüş tipine göre yükleme yapar.
- Bağlama bakmadan aşırı yükleme : fonksiyonun ismi, ve parametrelerinin tipine göre yükleme yapar. Dönüş tipine bakmaz.

Bağlama bakarak aşırı yükleme

- Fonksiyon çağrısının geçtiği ifade pozisyonunda hangi tip bekleniyor (bağlam)

```
int f(double a) { ....① }
int f(int a) { ....② }
double f(int a) { ....③ }
double x,y;
int a,b;
```

- `y=f(x)`; ① (x double) `a=f(a)`; ② (a int, atama int) `x=f(a)`; ③ (a int, atama double) `x=2.4+f(a)`; ③ (a int, çarpma double) `a=f(f(x))`; ②(①) (x double, f(x):int, atama int) `a=f(f(a))`; ②(②) ya da ①(③) ???
- Problem karmaşıklaşıyor.

Bağlama bakmadan aşırı yükleme

- Bağlama bakarak aşırı yükleme pahalı, karmaşık ve kafa karıştırıcı.
- Gerekli mi?
- Aşırı yükleme yapan çoğu dil bağlama bakmaz.
- Bağlama bakmadan aşırı yükleme ② ve ③ fonksiyonlarının birlikte olmasına izin vermez.
- "`isim: parametreler → dönüş` " içerisinde "`isim: parametreler`" aynı olan tek fonksiyon olabilir.
- Aşırı yükleme çok yararlı değildir. Çoğu dil izin vermez.

  Kullanıma dikkat:

  Aşırı yükleme sadece fonksiyonlar benzeri anlamlı işlemleri yapıyorsa kullanılmalı. İlgisiz iki amacı yerine getiren fonksiyonlara aynı isim verilmemeli. Kafa karıştırır, hataya yol açar.

8.3 Otomatik tip çevirimi (coercion)

Otomatik tip çevirimi (coercion)

- Programcıya kolaylık olması için sıradan tip çevrimleri derleyici tarafından otomatik yapılır.

```
double x;
int k;
k = x+3.45;     /* k=(int) (x+3.45); */
k = x+2;        /* k=((int) x )+2; */
x = x+k-2;      /* x= x+ (double)k - (double) 2 ; */
```

- C dili *int* ↔ *double* çevrimlerini ve gösterge tiplerini çevirir (uyarı vererek).
- Başka tip çevrimleri yapılabilir (A * → A, A → A * gibi)
- Otomatik tip çevrimi hatalara yol açabilir: `x=(k=3.25)` : x, 3.0 olacaktır.
- Otomatik tip çevrimi ve aşırı yükleme birlikte çok karmaşıklaşır.
- Yeni diller bu yüzden çevirim yapmaz. (Katı tip denetimi)