

BIM 202 - Bahar 2011

Programlama Dilleri

Prof. Dr. Tuğrul Yılmaz
e-posta: tyilmaz@mu.edu.tr

Pazartesi 9:00-12:00 Z33 nolu sınıf
13:30-16:15 PCLAB1

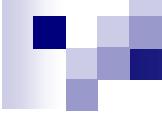
Akademik Dürüstlük

- Ad'ımız her şeyimizdir, koruyalım.
 - Başarılı olmanızı beklerim, gerekli çabayı gösterirseniz olursunuz.
 - Hile ile başarıyı yakalayamazsınız, çalışın.
 - Ödevleri hemen çalışmaya başlayın.
 - Takılırsanız sorun.
 - Paniklemeyin bazı sorular zor olacaktır.
 - Zor sorular iyi öğrencilerin limitlerini görmek içindir.
 - Verdiğim ödevleri yaparsanız, derslere devam edip anlamaya çalışırsanız faydasını görürsünüz.

Dersin Amacı

Programlama dili nedir?

- Programları anlatmak için bir simgeler kümesi ile gösterilmesi (kaynak kod).
- Programın hazırlanabilmesi için bir çatı.
- Derleme (compilation) - Program başka bir dile tercüme edilir, sonra çalıştırılır.
- Yorumlayıcı (interpreter) - Program çalışırken yorumlanır.
- İşletim sistemi değildir.



Neden programlama dilleri dersi?

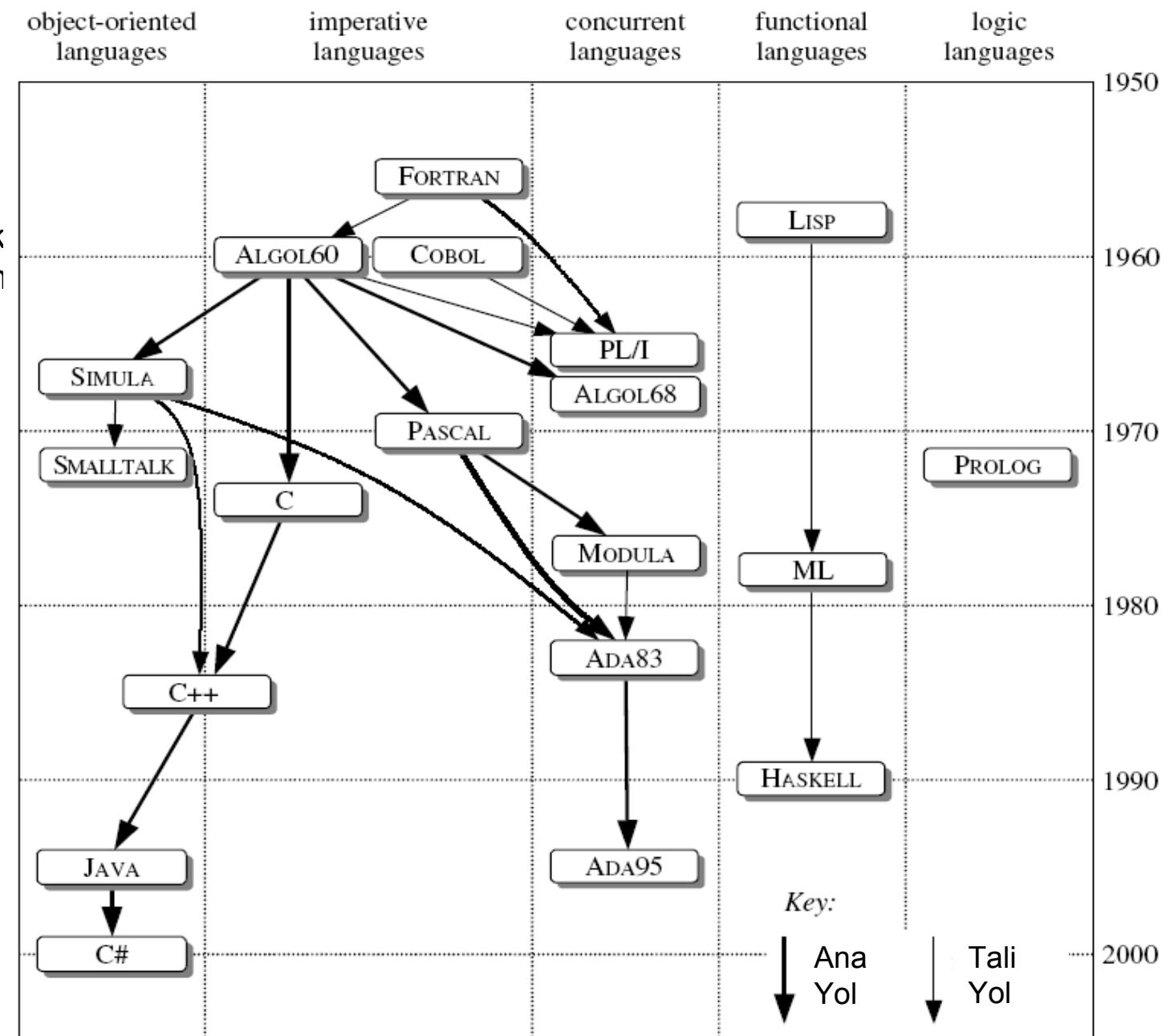
- Fikirlerimizi uygularken daha kolay ve daha iyi yapabilmek için. Seçeneklerimizin ne olduğunu bilirsek iyiyi seçebiliriz.
- Dil öğrenmede yetkinlik. Dillerin özelliklerini bilmeyen, belli bir dille çalışmaya alışmiş kişi, farklı bir dili öğrenmesi gerektiğinde zorlanır.
- Belli bir dilin önemli özelliklerini anlayarak daha iyi kullanabilmek için.
- Berimin (Hesaplamanın) gelişmesi için. Dilleri daha iyi değerlendirebilirsek, doğru seçimler yaparız, doğru teknolojilerin gelişmesine destek olmuş oluruz.
- Hata ayıklarken özelliklerini bilmemiz faydalıdır.
- Özellikleri öğreniriz, olmayan özelliklerine öykünürüz (emulate).

Programlama dilleri kullanım tercihleri nedenleri

- Teknik özellikler
 - Kolay kullanım.
 - Problem tipi (şekil işlevi takip eder)
 - Anlatımcılığı (ne kadar genel?)
 - Yazma kolaylığı
 - Performans
 - Esneklik/Gelişmişlik(established languages)
- Teknik olmayan faktörler
 - Eylemsizlik (eskiden vazgeçememe).
 - Büyük destekçiler/görünürlük (yeni diller için).
 - Derleyici/yorumlayıcı erişilebilirliği.
 - Kişisel tercihler/inanışlar.

Dillerin tarihçesi

- **Object-oriented:** Nesne tabanlı.
- **Imperative:** Buyurgan dil. İşlevsel dillerden farklı olarak değişkenlere değerler atayan dil.
- **Concurrent:** Koşut zamanlı diller, koşut zamanlı görev/komut dizisi ile belirginleşir.
- **Functional:** fonksiyonel dil. Lisp, Postscript, Haskell örneklerinde olduğu gibi, bir veri işleme işinde yapılacak işleri salt fonksiyon çağrıları ile ifade eden programlama dili.
- **Logic:** mantıksal dil.



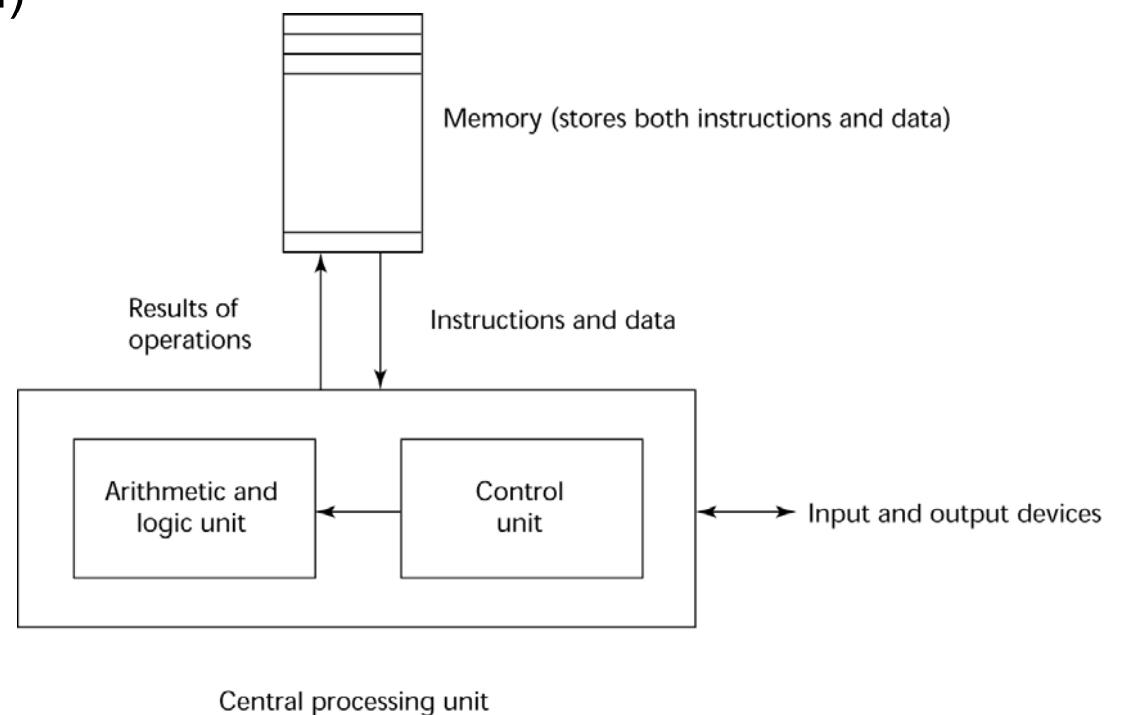
Programlama Dilleri Taksonomisi

■ Zorunlu/buyurgan (imperative) (akış kontrolüne odaklı)

- Yordamsal (Procedural) (von Neuman)
- Nesneye yönelik (Object Oriented)

■ Bildirimci (declarative)

- Fonksiyonel
- Veri akışı
- Mantıksal, kısıt tabanlı



Taksonomi: Bir bilgi dağarcığını bölüntüleyip, parçaları arasındaki ilişkiyi tanımlayan yöntem

Zorunlu/buyurgan (imperative) diller

- akış kontrolüne odaklı (Focus on Control Flow) komutlar (Instructions)
 - Yordamsal/yöntemsel (Procedural) (von Neumann)
 - Veri üzerindeki eylemi belirler.
 - Assembly, Fortran, Basic, Pascal, C, Bourne Shell
 - Nesneye yönelik (Object Oriented)
 - Veriyi gruplandırmaya ve işlemeye dil desteği (Kılıflama (Encapsulation))
 - Simula 67, Small Talk, C++ (Hybrid), Eiffel, Java

Bildirimci (declarative) diller

- bildirimci = veri güdümlü (Data Driven)
 - Fonksiyonel (Functional) - Alonzo Church (June 14, 1903 – August 11, 1995) tarafından geliştirilen Lamda hesabına dayanır: Lisp, ML, Haskell
 - Veri akışı (Data Flow) - boru hattı veri işlemleri (Pipelined data operations)
 - Mantıksal, kısıt tabanlı (Logical, Constraint Based) – Kuralları koy, başlangıç koşullarını belirle, sonuç için komutları belirle.
 - Prolog and Spread Sheets (Visicalc/Lotus/Excel)
 - İlişkisel (relational) – Veri tabanı sorgulaması (Database Query) - SQL

İlişkilendirme zamanı (Binding Time)

- İlişkilendirme dil nesnelerine değerler atar

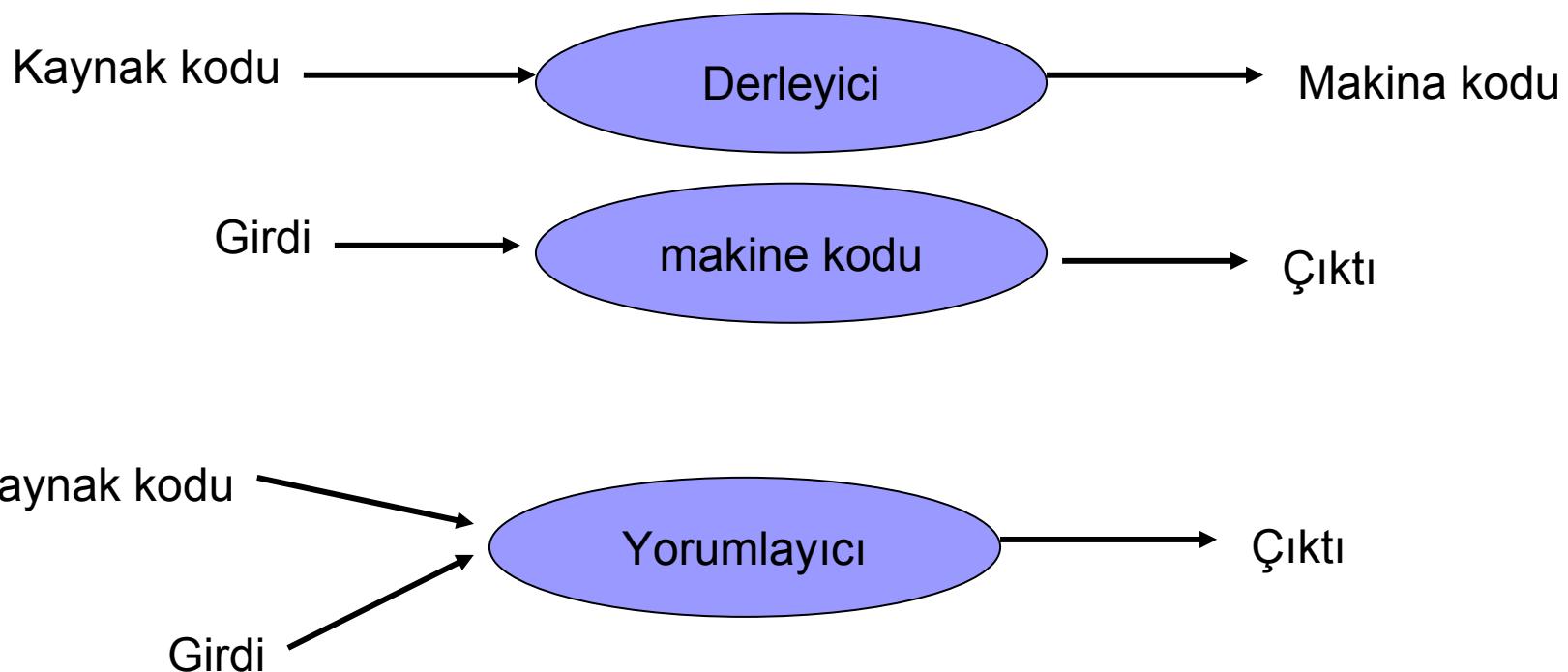
- komut adresleri
 - veri değerleri
 - veri adresleri

- İlişkilendirme:

- erken olursa – performans artar,
 - geç olurse - esneklik artar.

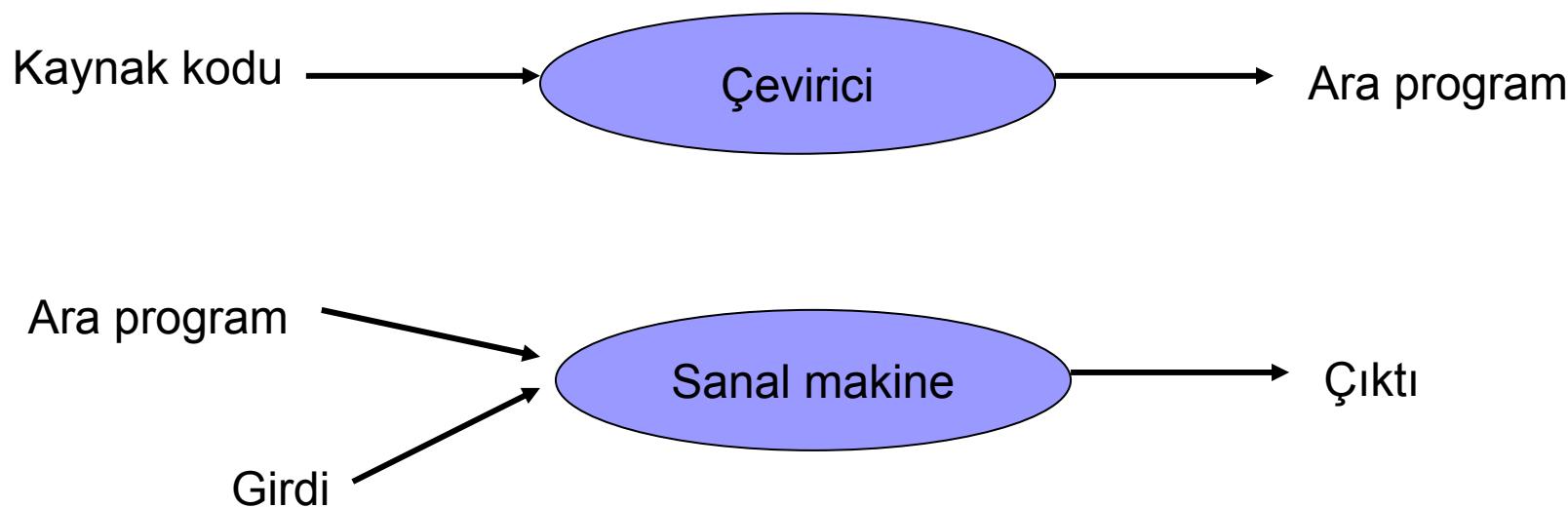
Derleyicilere karşı yorumlayıcılar (Compilers Vs. Interpreters)

- Çeviri yürütmeden ayrı mı?
 - Evet – Derleyici (Compiler)
 - Hayır – Yorumlayıcı (Interpreter)
- Birleşik uygulama (örneğin Java)



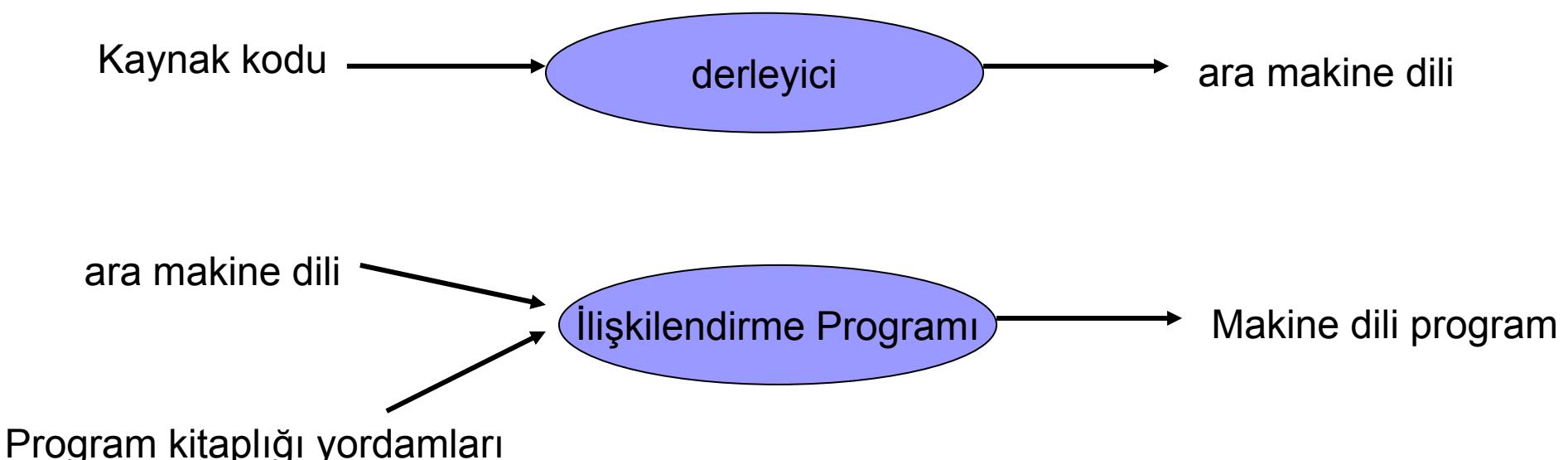
Neden yorumlayıcı?

- Esneklik (yürütüm zamanı oluşan geç bağlantılar nedeniyle)
- Yürütüm zamanı durum desteği
 - komut dosyaları (Perl, Shells, Python, TCL)
 - dinamik ortamlar (Basic, APL, LISP)
 - sanal makineler (JVM, Emulators, CPUs).

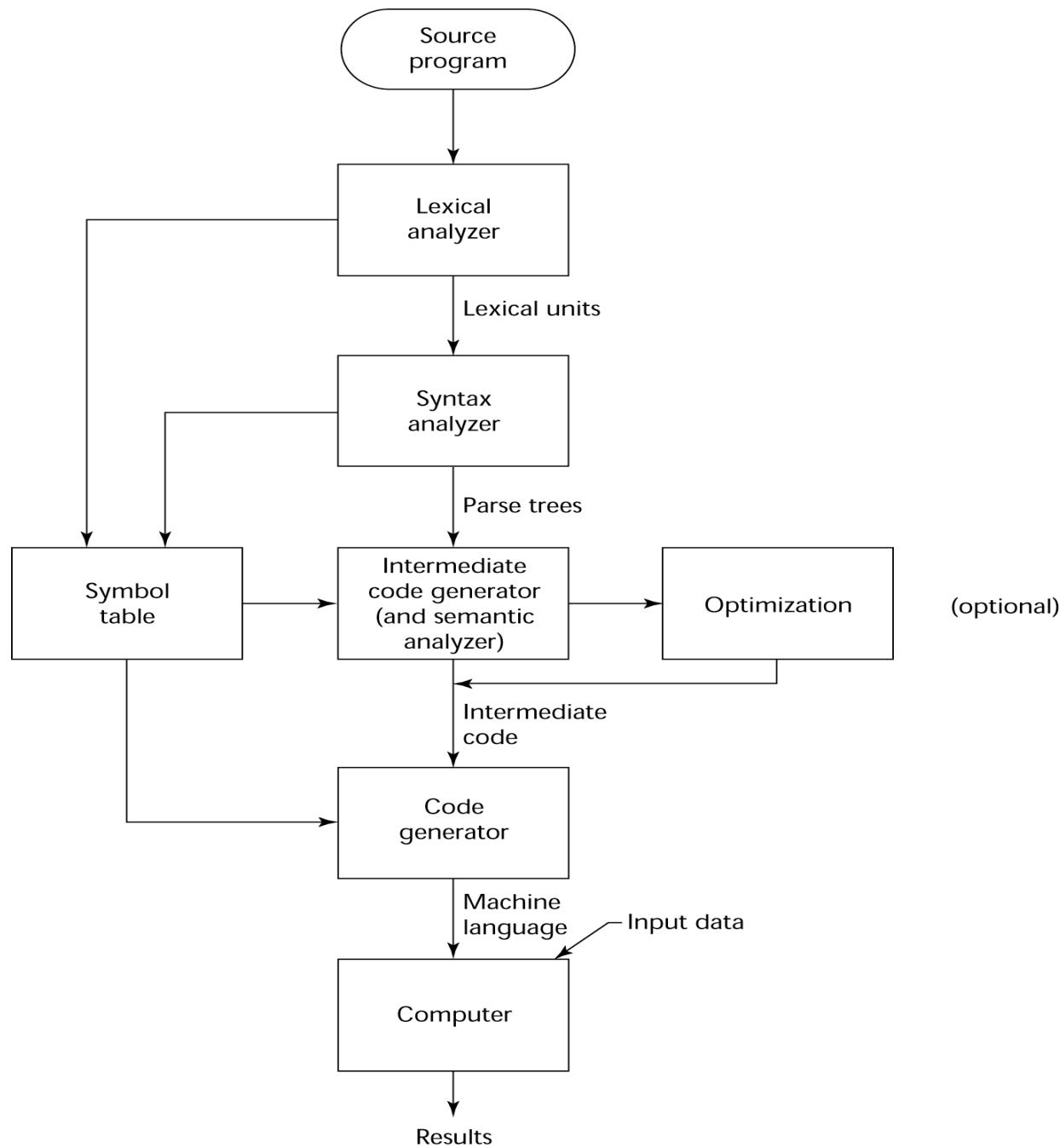


Neden derleyici?

- Temel mühendislik prensipleri
 - doğruluk – erken statik hata kontrolü
 - maliyet – derleme program dağıtma maliyetini düşürür.
 - performans – hızlı çalışır
 - bir defa derle (maliyet) , birçok defa yürüt (fayda)
- Yazılımın/fikirlerin korunması



Derleme



Çok geçişli derleyiciler

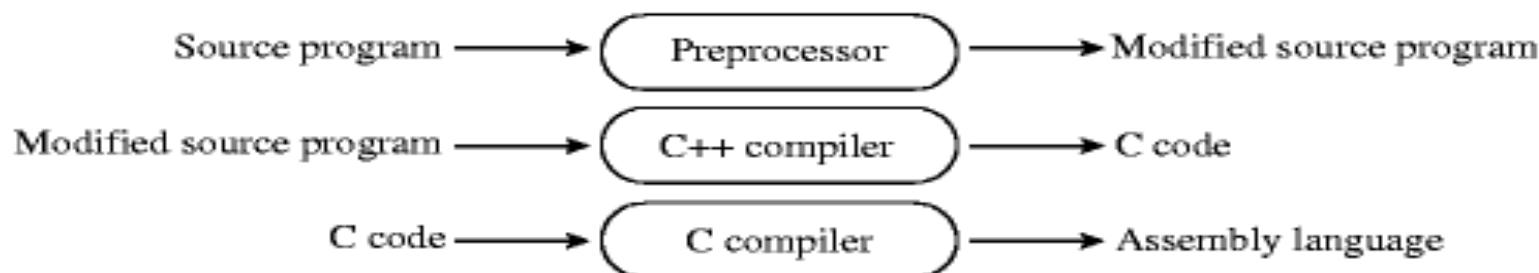
- Karmaşıklığı nasıl çözümleyelim?
 - Program kitabı ile (dili basit tutarak, örneğin Java).
 - Katmanlaştırarak (her seferinde bir problem üzerinde yoğunlaşarak)
 - Peş peşe fazlalarдан oluşan çözüm.

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

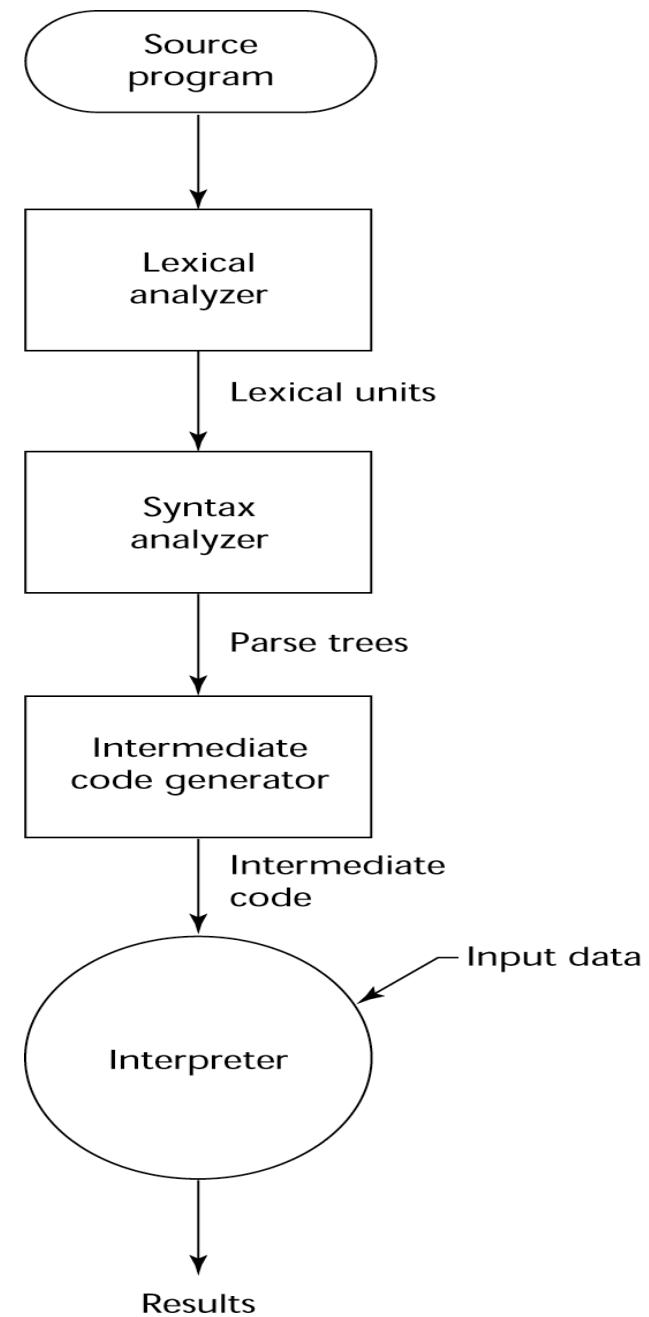


Başka dile derleme

- Bazı derleyiciler çeviriçi diline derler (assembly code)
 - yüksek oranda optimize olur
- Bazı diğer derleyiciler başka üst seviye dile derleyebilir.
 - Var olan dilin optimizasyonunu kullanılır.
 - Taşınabilirliği artırır, karmaşıklığı azaltır.



Birleşik uygulama (derleme+yorumlama)



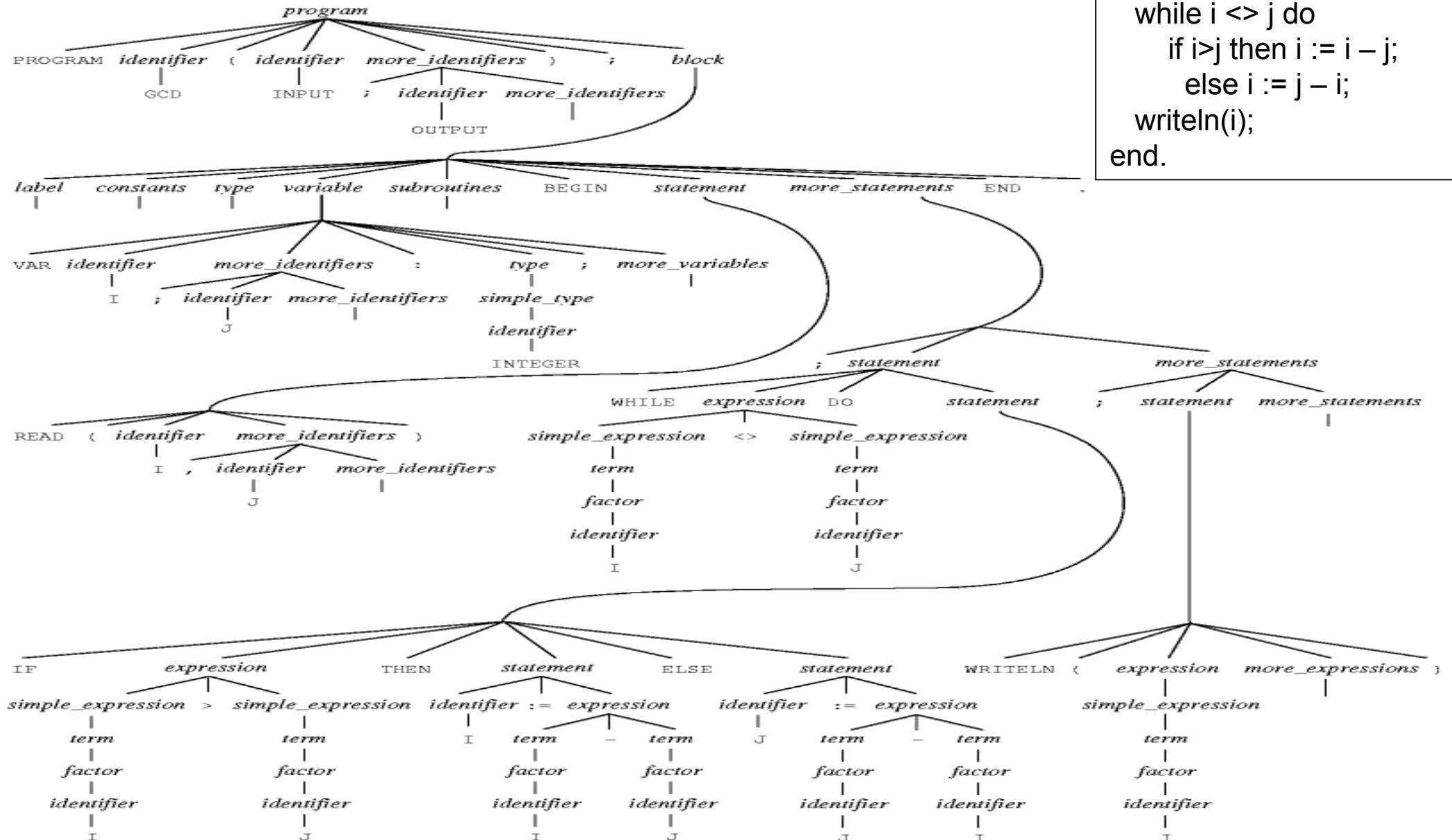
Bir örnek

- Bir Pascal Programı düşünelim.

```
program gcd(input, output);
var i, j: integer;
begin
  read(i, j);
  while i <> j do
    if i>j then i := i - j;
    else i := j - i;
  writeln(i);
end.
```

Sözdizim analiz (Syntax Analysis)

- Tarama uçları belirler (tokens)
- Avrıldırma uc olmavanları belirler.



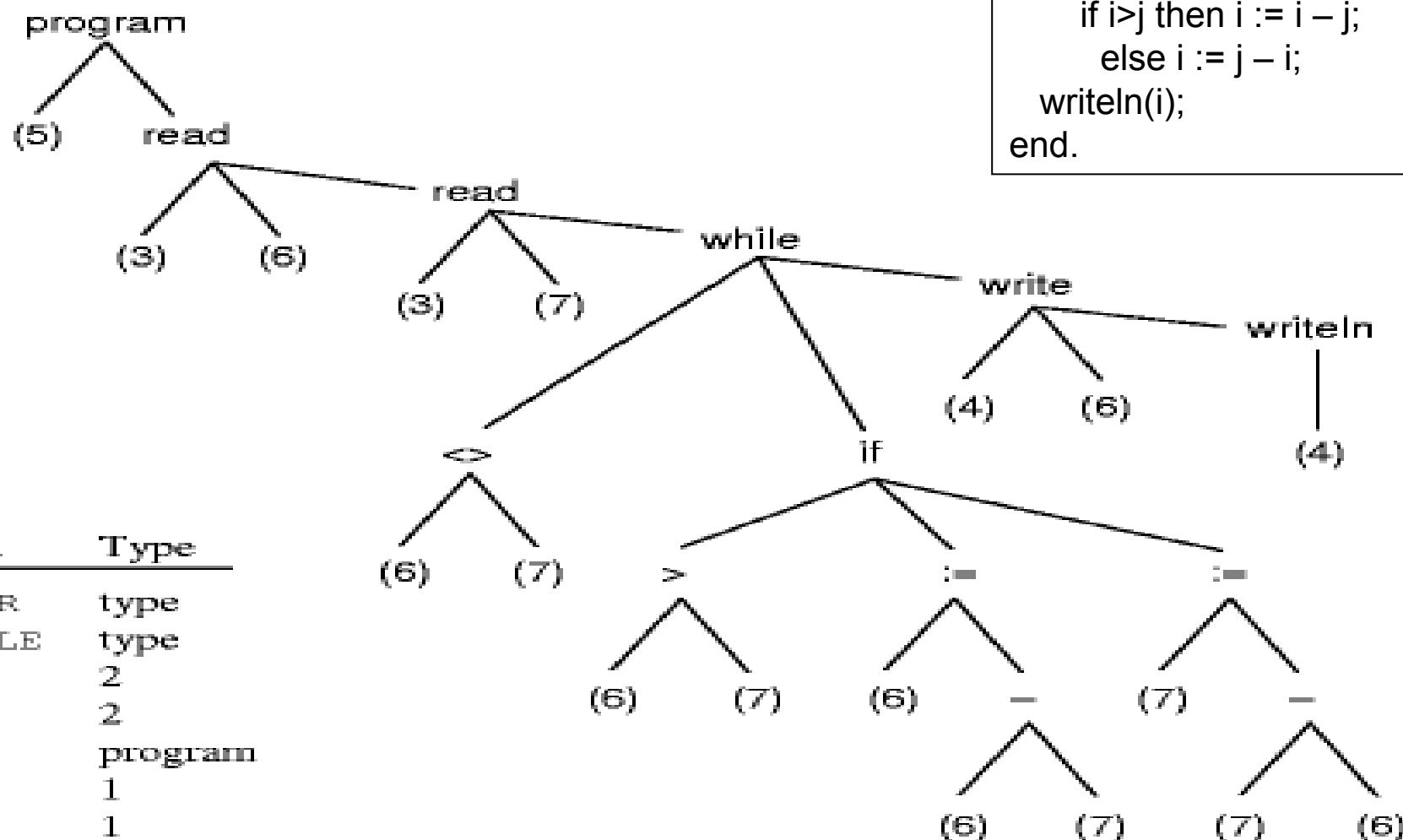
```

program gcd(input,
output);
var i, j: integer;
begin
  read(i, j);
  while i <> j do
    if i>j then i := i - j;
    else i := j - i;
  writeln(i);
end.
  
```

Anlambilimsel (semantic) analiz

- Anlambilimsel analiz son kısımdır.
- Özeti sözdizim analiz ağacı kullanılır.

```
program gcd(input,  
           output);  
var i, j: integer;  
begin  
  read(i, j);  
  while i <> j do  
    if i>j then i := i - j;  
    else i := j - i;  
    writeln(i);  
end.
```



Optimizasyon

■ Hedef kaynak tüketimini azaltmak

- bellek (veri veya kod)
- yürütme zamanı

Programlama ortamları

-Yazılım geliştirilmesinde kullandığımız araçlar

1. UNIX, Linux

- Geliştirilen birçok araç bulunmaktadır.

2. Borland JBuilder

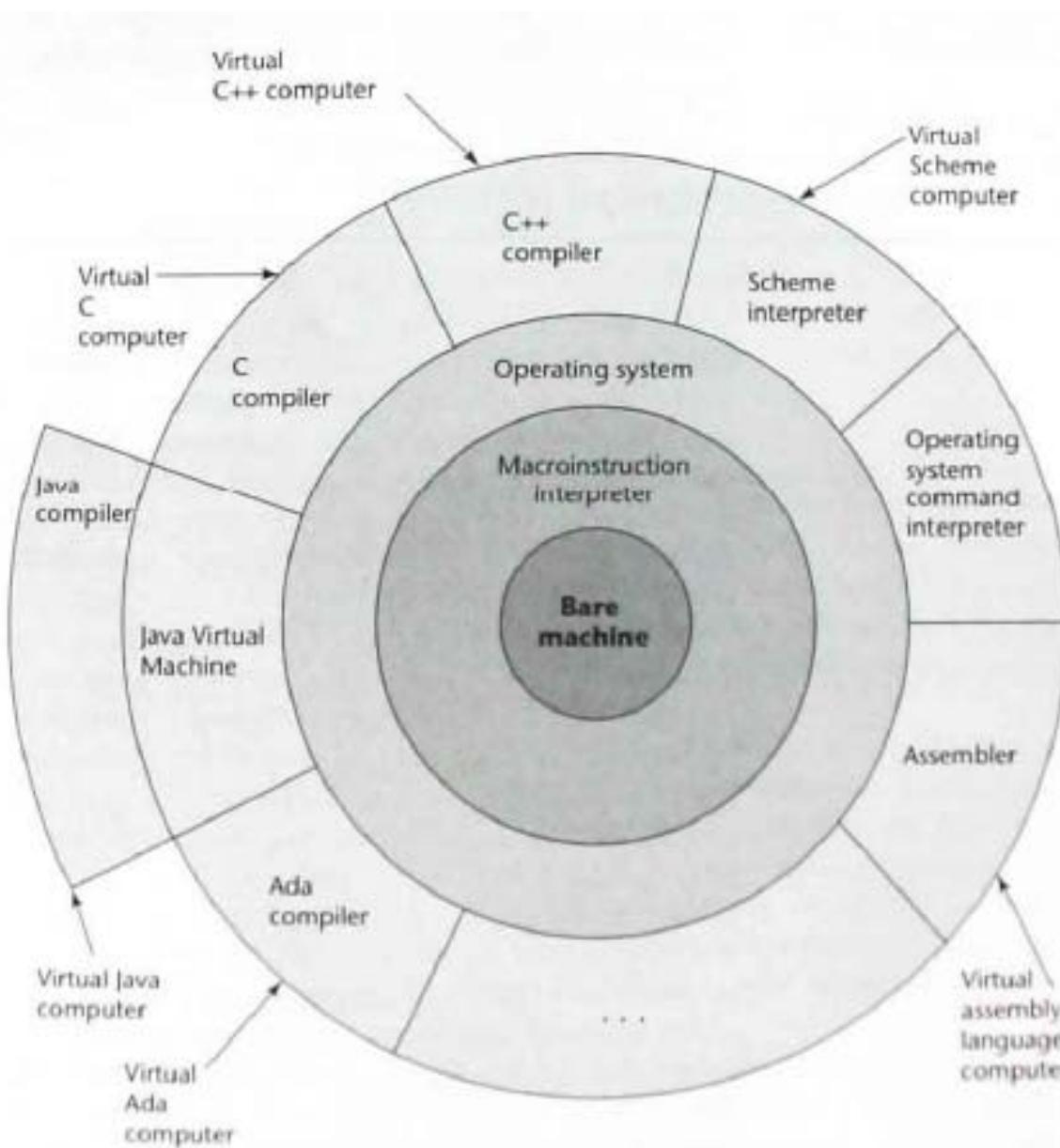
- Java için entegre yazılım geliştirme ortamı

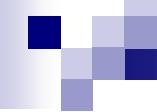
3. Microsoft Visual Studio

- Bütün .NET dilleri için geliştirme ortamı

4.

Bilgisayarların kademelendirilmiş arayüzü





PROGRAMLAMA DİLLERİ DEĞERLENDİRME KRİTERLERİ

- Programlama Dillerinin Gelişimi bölümünde tanıtıldığı gibi, çeşitli programlama amaçlarına uygun çok sayıda programlama dili vardır. Programlama dilleri arasında seçim yapmak için çeşitli değerlendirme kriterlerine ihtiyaç duyulmaktadır.
- Bir programlama dilinin değerlendirilmesinde göz önüne alınması gereken kavramlar çok sayıdadır ve dilin kullanım amacına göre farklılık gösterebilmektedir. Ancak genel olarak bir programlama dilinin değerlendirilmesi için bazı kriterler belirlenmiştir. Bu kriterlerin en önde gelenleri, **okunabilirlik, yazılabilirlik ve güvenilirlik** olarak sayılabilir.

Okunabilirlik

- Bir programlama dilinin değerlendirilmesinde en önemli kriterlerden birisi, programların okunabilme ve anlaşılabilme kolaylığıdır. Programlama dillerinin okunabilir olmaları, programlarda hata olasılığını azaltır ve programların bakımını kolaylaştırır.
- Bir programlama dilinde yer alan kavramlar, yapılar ve dilin sözdizimi, dilin okunabilirliğini doğrudan etkiler. Karmaşık bir sözdizim, bir programın yazımı sırasında kısa yollar sağlayabilir ancak programın daha sonra değiştirilmek amacıyla okunmasını ve anlaşılmasını zorlaştırır.

Yazılabilirlik

- Bir programlama dilinde program yazma kolaylığını belirleyen en önemli etkenlerden birincisi, programlama dilinin sözdizimidir. Buna ek olarak, programlama dilinin soyutlama yeteneği, dilin yazılabılırlığını önemli ölçüde etkilemektedir.
- Programlama dilleri, işlem veya veri olarak iki ayrı şekilde soyutlama sağlayabilirler. Bunlara ek olarak, yazılabılırlik açısından söz edilmesi gereken bir diğer konu, seçilen dilin, eldeki problem konusuna uygunluğudur.

Güvenilirlik

- Bir programlama dilinin güvenilirliği, o dil kullanılarak geliştirilen programların güvenilir olmasıdır.
- Programlama dillerinde güvenilirlik, çeşitli faktörler tarafından belirlenir. Bunlara örnek olarak, dilde bulunan tip denetimi ve istisnai durum işleme verilebilir.
- Programın doğruluğunu sağlanması için programlama ortamında sunulan araçlar da, güvenilir programlar geliştirilmesini etkiler.

1. Plankalkül - 1945

- Hiç gerçekleştirilmedi.
- Gelişmiş veri yapıları
- Notasyon:

$$A[7] = 5 * B[6]$$

		5	*	B	=>	A
V			6	7		(indeks)
S			1.n	1.n		(veri tipi)

2. sözde program(Pseudocodes) - 1949

Bilgisayara özgü komut deyimleriyle yazılmamış ancak anlaşılabilirliği artıran ve yürütümünden önce çevrilmesi ya da yorumlanması gereken bilgisayar programı.

- kısa kod; 1949; BINAC; Mauchly
- bazı operasyonlar:
 - 1n => (n+2)nd power
 - 2n => (n+2)nd root
 - 07 => addition

3 IBM 704 ve FORTRAN (FORmula TRANslation)

- ***FORTRAN I - 1957***

(FORTRAN 0 - 1954 - gerçekleştirilmemiş)

- Yeni IBM 704 için gerçekleştirildi. Dizin yazmacı (index registers) ve kayar noktalı aritmetik donanımı vardı.
- isimler 6 karaktere kadardı
- DO loop
- Formatted i/o
- alt programlar
- arithmetic IF: if(aritmetik ifade) N1,N2,N3
- veri tipi yok
- 400 satırından uzun program nadiren derlendi. Bunun nedeni 704'ün güvenilmezliğiydı.
- kod hızlıydı.
- hızla kullanılmaya başladı.

3 IBM 704 ve FORTRAN (devam)

- *FORTRAN II* - 1958
- *FORTRAN IV* - 1960-62
 - veri tipi deklerasyonu
 - Mantıksal if,

FORTRAN 77 - 1978

- karakter dizgisi
- mantıksal döngü kontrolü
- IF-THEN-ELSE deyimi
- *Fortran 90* - 1990
 - Moduller, dinamik dizilim, gösterici (pointer)
- *Fortran 95* – 1995
- Fortran 2003
- *FORTRAN değerlendirme*
 - Çok büyük ölçüde değişti ve hala kullanılıyor.

```
! Fortran 95 Example program
! Input: An integer, List_Len, where List_Len is less
!         than 100, followed by List_Len-Integer values
! Output: The number of input values that are greater
!         than the average of all input values

Implicit none
Integer :: Int_List(99)
Integer :: List_Len, Counter, Sum, Average, Result
Result= 0
Sum = 0
Read *, List_Len
If ((List_Len > 0) .AND. (List_Len < 100)) Then
! Read input data into an array and compute its sum
    Do Counter = 1, List_Len
        Read *, Int_List(Counter)
        Sum = Sum + Int_List(Counter)
    End Do
! Compute the average
    Average = Sum / List_Len
! Count the values that are greater than the average
    Do Counter = 1, List_Len
        If (Int_List(Counter) > Average) Then
            Result = Result + 1
        End If
    End Do
! Print the result
    Print *, 'Number of values > Average is:', Result
Else
    Print *, 'Error - list length value is not legal'
End If
End Program Example
```

4 LISP - 1959

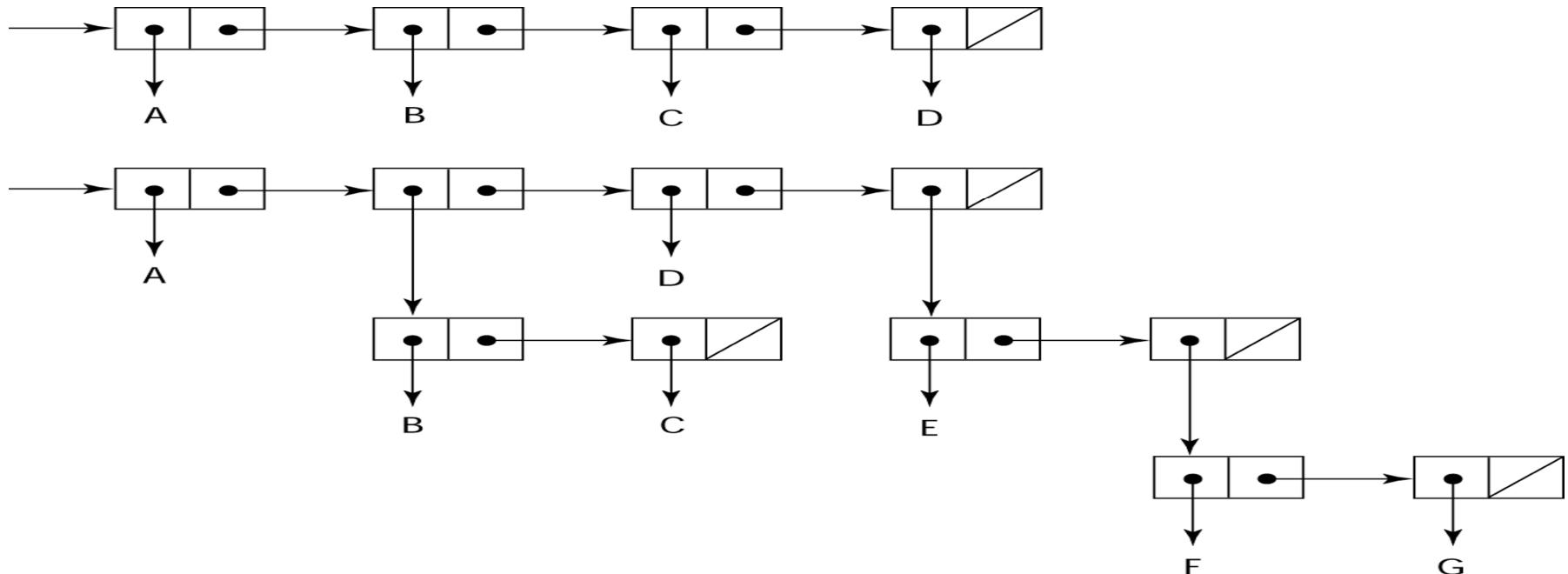
- LISt Processing language
(Designed at MIT by McCarthy)

- İki veri tipi var: atom ve list
- Sözdizim lambda calculus'a dayanır
- *Fonksiyonel programlamada öncü*
 - Değişkenlere gerek yok.
 - Özyineleme (recursion) ve koşullu ifadeler ile kontrol.
- Yapay zeka için hala dominant.
- Common LISP, Standard Lisp ve Scheme çağdaş lehçeleri.
- ML, Miranda, ve Haskell ilgili diller.

Standart Lisp'de faktöriyel hesaplama örneği verelim:

```
(de factorial (n)
  (cond ((zerop n) 1)
        (t (times n (factorial (difference n 1))))))
  )
```

4 LISP – 1959 (devam)



list (A B C D) ve (A (B C) D (E (F G)))'nin gösterimi

5 ALGOL 58 ve 60

- ACM and GAMM 4 günlük toplantıda kararlaştırıldı.
- *ALGOL 58 özellikleri:*
 - type kavramı
 - isim boyu serbest
 - Array indeksleri serbest
 - Compound statements (begin . . . end)
 - noktalı virgül komut ayıracı.
 - atama operatorü :=
 - if else-if
- **Başta IBM desteği vardı orta-1959 da kestiler.**

5 ALGOL 58 ve 60 (devam)

- ALGOL 60

- Pariste bir toplantıda 6 günde geliştirildi.
- *Yeni özellikler:*
 - Blok yapısı (local scope)
 - İki tip parametre geçirme yöntemi
 - Altprogram özyineleme (Subprogram recursion)

- Başarıları:

- Algoritmaları açıklamak için kullanılması > 20 yıl
- Sonraki bütün buyurgan diller takip etti
- İlk makineden bağımsız dil
- Sözdizimi (syntax) resmen tanımlanan ilk dil (BNF)
- Bir komite tarafından tasarlanan ilk dil

- Başarısızlıklar:

- Geniş olarak kullanılamadı, özellikle ABD'de.

Nedenleri:

1. i/o yetersizde, karakter seti programların taşınabilirliğini azaltıyordu
3. Çok esnekti, gerçekleştirmi zordu
4. FORTRAN'ın direnişi
5. Resmi sözdizimi açıklamaları
6. IBM desteğinin olmaması

```
comment ALGOL 60 Example Program
Input: An integer, listlen, where listlen is less than
       100, followed by listlen-integer values
Output: The number of input values that are greater than
       the average of all the input values ;
begin
  integer array intlist [1:99];
  integer listlen, counter, sum, average, result;
  sum := 0;
  result := 0;
  readint (listlen);
  if (listlen > 0) ^ (listlen < 100) then
    begin
comment Read input into an array and compute the average;

    for counter := 1 step 1 until listlen do
      begin
        readint (intlist[counter]);
        sum := sum + intlist[counter]
      end;
comment Compute the average;
    average := sum / listlen;
comment Count the input values that are > average;
    for counter := 1 step 1 until listlen do
      if intlist[counter] > average
        then result := result + 1;
comment Print result;
    printstring("The number of values > average is:");
    printint (result)
    end
  else
    printstring ("Error--input list length is not legal");
end
```

6 COBOL - 1960

- ***FLOW-MATIC'e dayanır***

- FLOW-MATIC özellikleri:

- İsimler 12 karaktere kadar, tire dahil
 - Aritmetik işlemler için İngilizce isimler. *, - gibi işlemciler yok
 - Veri ve kod tamamen farklı
 - Eylem her cümlenin ilk kelimesi

- ***İlk tasarım toplantısı (Pentagon) - Mayıs 1959***

- Tasarım komitesi üyelerinin tamamı ya bilgisayar üreticilerinden ya da ABD savunma bakanlığından
 - Tasarım problemleri: aritmetik ifadeler? indeksler? Üreticiler arasında çekişme

- ***Dillere kavramsal katkıları:***

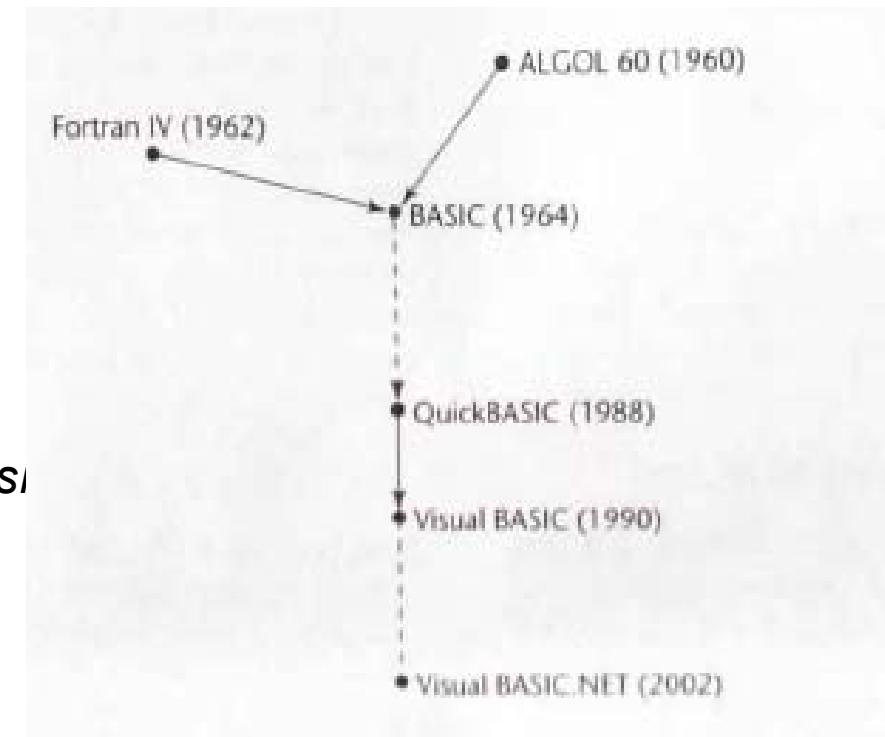
- Üst seviye dillerde ilk makro kavramı
 - Hiyerarşik veri yapıları (records)
 - İçiçe seçme ifadeleri
 - Uzun isimler (30 karaktere kadar), tire karakteri dahil
 - ayrı veri bölgesi

7 BASIC - 1964

- Kemeny & Kurtz at Dartmouth tarafından tasarlandı
- Güncel popüler lehçesi (dialect): Visual BASIC.NET
- İlk çok kullanılan zaman paylaşımı dil

8 PL/I - 1965

- IBM ve SHARE tarafından tasarlandı
- 1964'de hesaplama durumu tespiti (*IBM'in görüş açısı*)
 1. Bilimsel hesaplama
 - IBM 1620 ve 7090 bilgisayarları
 - FORTRAN
 - SHARE kullanıcı grupları
 2. İş hesaplaması
 - IBM 1401, 7080 bilgisayarları
 - COBOL
 - GUIDE kullanıcı grupları
- 1963'e kadar:
 - Bilimsel kullanıcılar daha çok i/o istiyordu
 - İş kullanıcıları ise kayan noktalı hesaplama ve dizimler (MIS)
- *Açık çözüm belliidi!:*
 1. Yeni bir bilgisayar yap, her tür uygulama olsun
 2. Yeni bir dil tasarıla hem iş dünyasına hem de bilimsel dünyaya hitap etsin

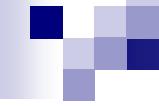


```

/* PL/I PROGRAM EXAMPLE
INPUT: AN INTEGER, LISTLEN, WHERE LISTLEN IS LESS THAN
       100, FOLLOWED BY LISTLEN-INTEGER VALUES
OUTPUT: THE NUMBER OF INPUT VALUES THAT ARE GREATER THAN
       THE AVERAGE OF ALL INPUT VALUES */

PLIEX: PROCEDURE OPTIONS (MAIN);
DECLARE INTLIST (1:99) FIXED.
DECLARE (LISTLEN, COUNTER, SUM, AVERAGE, RESULT) FIXED;
SUM = 0;
RESULT = 0;
GET LIST (LISTLEN);
IF (LISTLEN > 0) & (LISTLEN < 100) THEN
  DO;
/* READ INPUT DATA INTO AN ARRAY AND COMPUTE THE SUM */
  DO COUNTER = 1 TO LISTLEN;
    GET LIST (INTLIST (COUNTER));
    SUM = SUM + INTLIST (COUNTER);
  END;
/* COMPUTE THE AVERAGE */
  AVERAGE = SUM / LISTLEN;
/* COUNT THE NUMBER OF VALUES THAT ARE > AVERAGE */
  DO COUNTER = 1 TO LISTLEN;
    IF INTLIST (COUNTER) > AVERAGE THEN
      RESULT = RESULT + 1;
  END;
/* PRINT RESULT */
  PUT SKIP LIST ('THE NUMBER OF VALUES > AVERAGE IS:');
  PUT LIST (RESULT);
  END;
ELSE
  PUT SKIP LIST ('ERROR-INPUT LIST LENGTH IS ILLEGAL');
END PLIEX;

```



8 PL/I (devam)

- Beş ayda 3 X 3 lük bir komite tarafından tasarlandı

- *PL/I 'nın dil tasarımına katkıları:*

1. İlk birim bazında koşut zamanlı (concurrency)
2. İlk ayrılık yönetimi (exception handling)
3. Anahtar seçmeli özyineleme (Switch-selectable recursion)
4. İlk gösterici veri tipi (pointer data type)
5. İlk dizilim kesişmesi (First array cross sections)

- *notlar:*

- Birçok yeni özellik iyi tasarılanmamıştı.
- Çok büyük ve çok karmaşıktı.
- İş ve bilimsel amaçlarla kullanılıyordu.

9 APL ve SNOBOL

- Dinamik tipleme ve dinamik bellek tahsis etme.

- APL (A Programming Language) 1962

- Donanım betimleme dili olarak tasarlandı (IBM'de, Ken Iverson tarafından)
- Çok detaylı (değişik amaçlar için birçok işlev)
- Programları okumak çok zor.

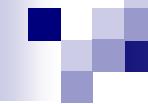
- SNOBOL(1964)
 - Dizgi (string) işlemek için tasarlanmış bir dil.
(Bell Lab'da, Farber, Griswold, ve Polensky tarafından)
 - Dizgi örüntü eşleştirmeleri için güçlü işleyiciler.

10 SIMULA 67 - 1967

- Sistem simülasyonu için tasarlanmış bir dil.
(Norveç'de, Nygaard ve Dahl tarafından)
- ALGOL 60 ve SIMULA I'e dayanır.
- *Dillerde en önemli katkısı:*
Sınıflar, nesneler ve kalıtım (Classes, objects, and inheritance).

11 ALGOL 68 - 1968

- ALGOL 60'dan geliştirilmiştir, fakat daha büyük kapsayıcı bir dil değildir.



12 Önemli ALGOL soyundan gelen programlama dilleri

- Pascal – 1971 - Wirth

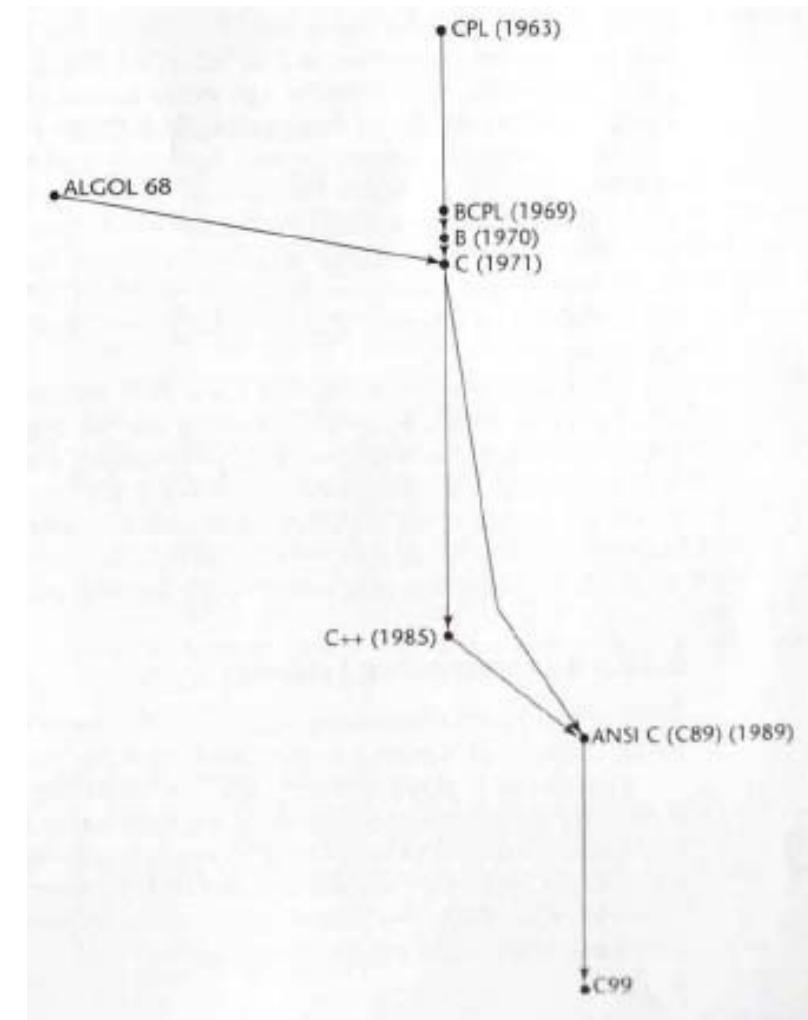
- Yapısal programlama öğretmek için tasarlandı.
- Küçük, basit, gerçek anlamda yeni birşey yok.
- 1970'lerin ortalarından 1990'ların sonuna kadar dil eğitiminde en çok kullanılan dildi.

- C - 1972

- Sistem programlama için tasarlandı – Richie.
- ALGOL 68 ve temelde B dilinden geliştirildi.
- Güçlü işleyçiler fakat zayıf tip kontrolü.
- Başlangıçta UNIX kanalıyla dağıtıldı.

- Perl – 1987 – Larry Wall

- ALGOL'e C üzerinden bağlı.
- Betik dili olarak da tanımlanır.
- Perl değişkenleri statik tiplidir ve örtülü tanımlanır.
 - İlk karakterle belirlenen üç tip değişken alanı.
- Genel ve web amaçlı programlama dili olarak geniş bir şekilde kullanılır.



```
{Pascal Example Program
Input: An integer, listlen, where listlen is less than
       100, followed by listlen-integer values
Output: The number of input values that are greater than
       the average of all input values }

program pasex (input, output);
type intlisttype = array [1..99] of integer;
var
  intlist : intlisttype;
  listlen, counter, sum, average, result : integer;
begin
  result := 0;
  sum := 0;
  readln (listlen);
  if ((listlen > 0) and (listlen < 100)) then
    begin
{ Read input into an array and compute the sum }
    for counter := 1 to listlen do
      begin
        readln (intlist[counter]);
        sum := sum + intlist[counter]
      end;
{ Compute the average }
    average := sum / listlen;
{ Count the number of input values that are > average }
    for counter := 1 to listlen do
      if (intlist[counter] > average) then
        result := result + 1;
{ Print the result }
    writeln ('The number of values > average is:',
             result)
  end { of the then clause of if (( listlen > 0 ... )
else
  writeln ('Error--input list length is not legal')
end.
```

```
/* C Example Program
Input: An integer, listlen, where listlen is less than
       100, followed by listlen-integer values
Output: The number of input values that are greater than
       the average of all input values */
void main (){
    int intlist[98], listlen, counter, sum, average, result;
    sum = 0;
    result = 0;
    scanf("%d", &listlen);
    if ((listlen > 0) && (listlen < 100)) {
        /* Read input into an array and compute the sum */
        for (counter = 0; counter < listlen; counter++) {
            scanf("%d", &intlist[counter]);
            sum += intlist[counter];
        }
        /* Compute the average */
        average = sum / listlen;
        /* Count the input values that are > average */
        for (counter = 0; counter < listlen; counter++)
            if (intlist[counter] > average) result++;
        /* Print result */
        printf("Number of values > average is:%d\n", result);
    }
    else
        printf("Error-input list length is not legal\n");
}
```

```
# Perl Example Program
# Input: An integer, $listlen, where $listlen is less
#         than 100, followed by $listlen-integer values.
# Output: The number of input values that are greater than
#         the average of all input values.
($sum, $result) = (0, 0);
$listlen = <STDIN>

if (($listlen > 0) && ($listlen < 100)) {
    # Read input into an array and compute the sum
    for ($counter = 0; $counter < $listlen; $counter++) {
        $intlist[$counter] = <STDIN>;
    } #- end of for (counter ...
    # Compute the average
    $average = $sum / $listlen;
    # Count the input values that are > average
    foreach $num (@intlist) {
        if ($num > $average) { $result++; }
    } #- end of foreach $num ...
    # Print result
    print "Number of values > average is: $result \n";
} #- end of if (($listlen ...
else {
    print "Error--input list length is not legal \n";
}
```

13 Prolog – 1972

- Aix-Marseille Üniversitesinde Comerauer ve Roussel tarafından geliştirildi. Edinburgh Üniversitesinden Kowalski yardım etti.
- Mantıga dayanır.
- Yordamsal değildir (Non-procedural).
- Akıllı veri tabanına dayalı, sorgulardan doğru sonuca olaşma olarak özetlenebilir.

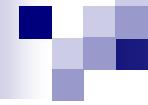
Prolog veri tabanı iki tip deyimden oluşur: olgular (facts) ve kurallar (rules). Faktöriyel hesaplama örneği verelim:

```
factorial(0,1).          %% olgu: 0! = 1 dir.  
factorial(N,F) :- N>0,    %% kural: N! = N*(N-1)!  
                  N1 is N-1, %% virgül 've' mantıksal işlecidir.  
                  factorial(N1,F1),  
                  F is N * F1.
```

Tanımlamalar yukarıdaki şekilde yapılmınca, aşağıdaki sorguda 'W' değişkeni sonucu döner:

```
?- factorial(3,W).
```

W=6



14 Ada - 1983 (1970'lerin ortaları)

- Yüzlerce insanın çalıştığı büyük geliştirme çalışması, para ve sekiz yıl.

- *Environment*: ABD savunma bakanlığında kullanılan sistemler için 450'den fazla dil.

- *Dillerde katkılar*:

1. Paketler – veri soyutlamasına destek olmak için;

2. Ayrıılıkların yönetimi (Exception handling);

3. Cinsine özgü program birimleri (Generic program units);

4. İş modeliyle dönemdeş erişim (Concurrency).

- *notlar*:

- Rekabetçi tasarım.

- Yazılım mühendisliği ve dil tasarımı hakkında bilinen herşey eklendi.

- İlk derleyiciler çok zordu. İlk kullanılabılır derleyici dilin tasarımının bitiminden 5 yıl sonra geldi.

- **Ada 95 (1988'de başladı)**

- Nesneye dayalı programlamaya destek.

- Paylaşılan veri için daha iyi destek.

- Daha esnek kütüphaneler.

```

-- Ada Example Program
-- Input: An integer, List_Len, where List_Len is less
--         than 100, followed by List_Len-integer values
-- Output: The number of input values that are greater
--          than the average of all input values
with Ada.Text_IO, Ada.Integer.Text_IO;
use Ada.Text_IO, Ada.Integer.Text_IO;
procedure Ada_Ex is
    type Int_List_Type is array (1..99) of Integer;
    Int_List : Int_List_Type;
    List_Len, Sum, Average, Result : Integer;
begin
    Result:= 0;
    Sum := 0;
    Get (List_Len);
    if (List_Len > 0) and (List_Len < 100) then
        -- Read input data into an array and compute the sum
        for Counter := 1 .. List_Len loop
            Get (Int_List(Counter));
            Sum := Sum + Int_List(Counter);
        end loop;
        -- Compute the average
        Average := Sum / List_Len;
        -- Count the number of values that are > average
        for Counter := 1 .. List_Len loop
            if Int_List(Counter) > Average then
                Result:= Result+ 1;
            end if;
        end loop;
        -- Print result
        Put ("The number of values > average is:");
        Put (Result);
        New_Line;
    else
        Put_Line ("Error-input list length is not legal");
    end if;
end Ada_Ex;

```

15 Smalltalk - 1972-1980

- Önceleri Alan Kay, sonra Adele Goldberg tarafından Xerox PARC'da geliştirildi.
- İlk Nesne Tabanlı Dil (NTD) (Object Oriented Language (OOL)) gerçekleştirmi (veri soyutlaması, kalıtım ve dinamik bağlama) (data abstraction, inheritance, and dynamic binding).
- Şimdi herkesin kullandığı grafik kullanıcı ara yüzünde öncü.

16 C++ - 1985

- Developed at by Stroustrup tarafından Bell Labs'da geliştirildi.
 - C ve SIMULA 67'den türetildi.
 - Kısmen SIMULA 67'den alınan Nesne Tabanlı Programlama (NTP) özellikleri C'ye eklendi.
 - Ayrılık yönetimi de var.
 - Büyük ve karmaşık bir dil.
 - NTP ile birlikte hızla popülerliği arttı.
 - ANSI standartları Kasım 1997'de kabul edildi.
-
- **Eiffel** – NTP desteği veren ilgili dil.
 - Bertrand Meyer tarafından tasarlandı - 1992
 - Hiçbir dilden doğrudan türetilmedi.
 - C++'dan küçük fakat onun gücünü koruyor.

16 C++ - 1985 (continued)

- **Delphi** – başka bir ilgili dil - Anders Hejlsberg tarafından tasarlandı (Turbo Pascal ve C#)
- Pascal'a dayanan melez bir dil.

17 Java (1995)

- 1990 başlarında Sun tarafından geliştirildi.
- C++'a dayanır
 - Önemli ölçüde basitleştirildi
(C++ 'da bulunan struct, union, enum, ve atama zorlamalarının yarısı yoktur.)
 - Sadece NTP destekler.
 - Referanslar vardır fakat göstericiler (pointers) yoktur.
 - Dönemdeşlige (concurrency) ve uygulamacıklara (applet) destek verir.

18 Betik dilleri (Scripting Languages)

- **JavaScript (1985)**

- Netscape'in LiveScript dili olarak başladı.
- Çoğunlukla kullanıcı tarafı, HTML içinde gömülü, tarayıcıda çalışan betik dilidir.
- Kullanıcı tarafında dinamik web dökümanlarında ve veri girişi kontrolünde çok kullanılır.
- Tamamen yorumlanan bir dildir.

```
// Java Example Program
// Input: An integer, listlen, where listlen is less
//         than 100, followed by length-integer values
// Output: The number of input data that are greater than
//         the average of all input values
import java.io.*;
class IntSort {
    public static void main(String args[]) throws IOException {

        DataInputStream in = new DataInputStream(System.in);
        int listlen,
            counter,
            sum = 0,
            average,
            result = 0;
        int[] intlist = new int[99];
        listlen = Integer.parseInt(in.readLine());
        if ((listlen > 0) && (listlen < 100)) {
/* Read input into an array and compute the sum */
            for (counter = 0; counter < listlen; counter++) {
                intlist[counter] =
                    Integer.valueOf(in.readLine()).intValue();
                sum += intlist[counter];
            }
/* Compute the average */
            average = sum / listlen;
/* Count the input values that are > average */
            for (counter = 0; counter < listlen; counter++)
                if (intlist[counter] > average) result++;
/* Print result */
            System.out.println(
                "\nNumber of values > average is:" + result);
        } /** end of then clause of if ((listlen > 0) ...
        else System.out.println(
            "Error-input list length is not legal\n");
    } /** end of method main
} /** end of class IntSort
```

JavaScript Örneği

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.1 //EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11-strict.dtd">
<!-- example.html
      Input: An integer, listLen, where listLen is less
             than 100, followed by listLen-numeric values
      Output: The number of input values that are greater
              than the average of all input values
-->
<html>
<head><title> Example </title>
</head>
<body>
<script type = "text/javascript">
<!--
var intList = new Array(99);
var listLen, counter, sum = 0, result = 0;

listLen = prompt (
    "Please type the length of the input list", "");
if ((listLen > 0) && (listLen < 100)) {
// Get the input and compute its sum
    for (counter = 0; counter < listLen; counter++) {
        intList[counter] = prompt (
            "Please type the next number", "");
        sum += parseInt(intList[counter]);
    }
// Compute the average
    average = sum / listLen;
// Count the input values that are > average
    for (counter = 0; counter < listLen; counter++)
        if (intList[counter] > average) result++;
// Display the results
    document.write("Number of values > average is: ",
                  result, "<br />");
} else
    document.write(
        "Error - input list length is not legal <br />");
// -->
</script>
</body>
</html>
```

18 Betik dilleri

- **PHP – 1994 - Rasmus Lerdorf**

- Sunucu tarafı, HTML içinde gömülü, betik dilidir.
- Çoğunlukla web üzerinden form işleme veritabanı erişimleri için kullanılır.
- Tamamen yorumlanan bir dildir.

- **Python – 1990’lı yılların başları – Guido Van Rossum**

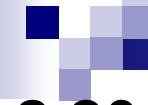
- Sistem yönetimi, CGI programlama
- Tip kontrollü dinamik tipleme.
- Dizilimler (array) yerine listeler, değiştirilemez listeler (tuples), and kıymılı listeler (dictionaries)

- **Ruby – 1990’lı yılların ortaları – Yukihiro Matsumoto**

- Tam NTL – hersey nesne.
- A scripting language.
- Tip kontrolü olmadan dinamik tipleme.
- Sınıflar ve nesneler dinamik.

2.19 C# - 2000 - Microsoft

- .NET platformları için ana dil.
- Java ve C++ takipcisi.
- Java'nın çoğu özelliğini bir kısım değişikliklerle ve bazı C++ özelliklerini kapsar.
- web üzerinde .NET uygulamaları için kullanılabileceği gibi genel amaçla da kullanılabilir.



2.20 Bağlantılı metin/Programlama melez dilleri (Markup/Programming Hybrid Languages)

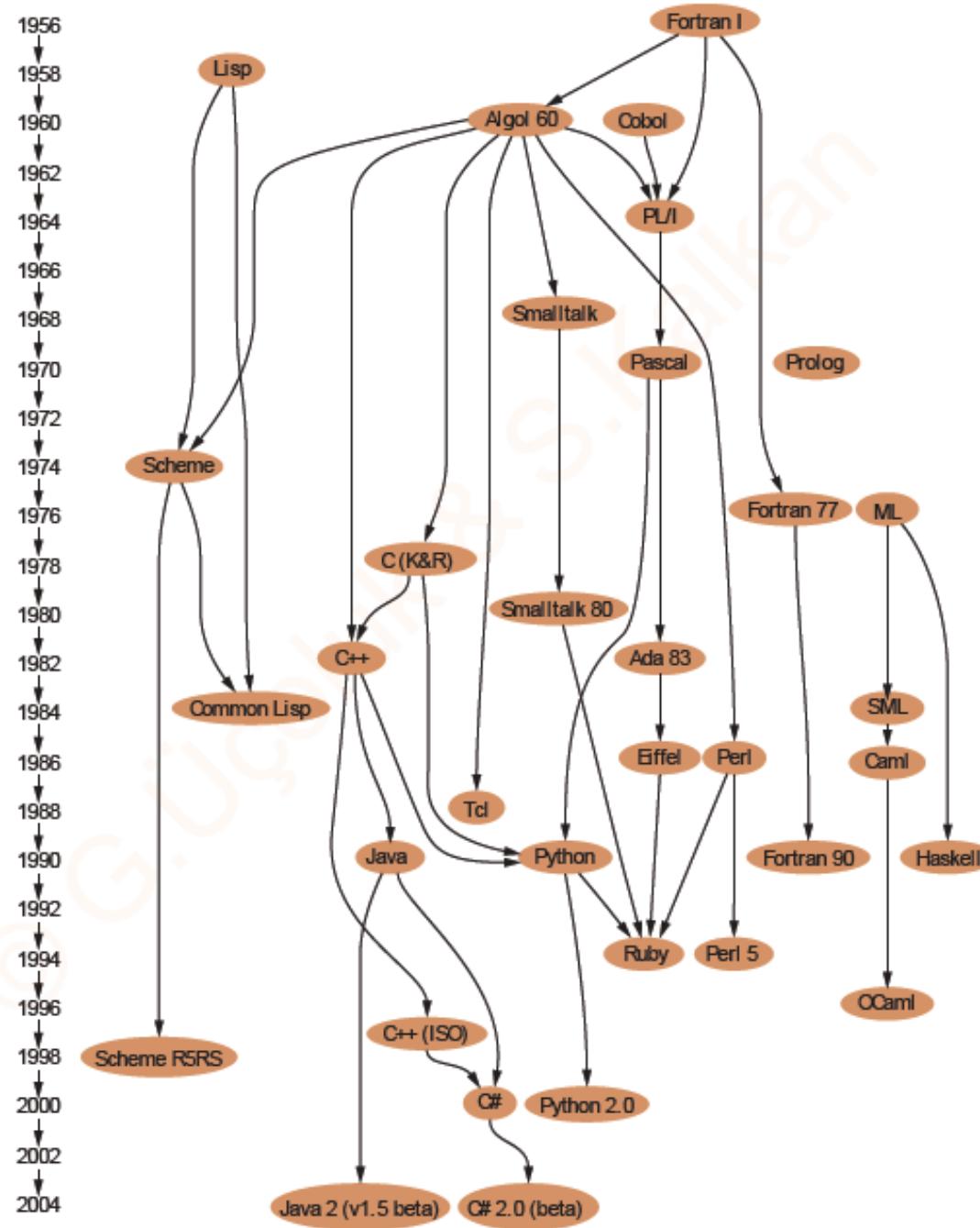
- XSLT

- XML dokümanlarının ekranında gösterimi için kullanılır.
- Etiketler şeklinde kontrol yapıları içerir, örneğin: <for-each>

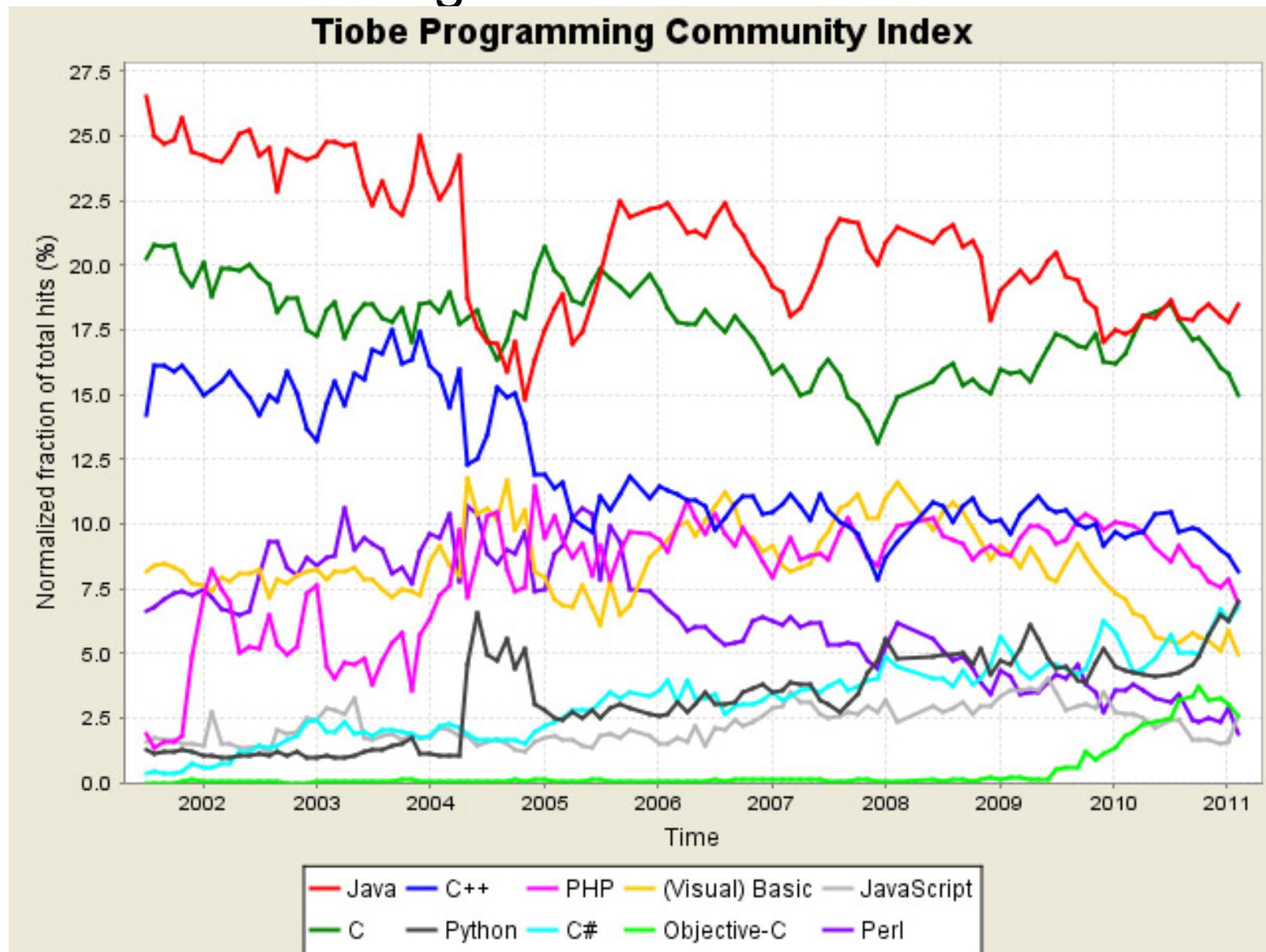
- JSP

- “Java Server Page” XHTML ve Java’nın karışımıdır.
- Sayfalar JSP işlemcisi tarafından işlenerek sunucu java uygulamaları haline getirilir.
- JSTL, JSP dokümanının işlemesini kontrol eden XML hareket elemanlarını tanımlar.
- Örnek hareket elemanları: <if>, <forEach>, vs.

Üst seviye
programlama
dillerinin şeceresi

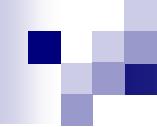


Günümüz Programlama Dilleri Kullanıcı Oranları



<http://www.tiobe.com/index.php/content/paperinfo/tpci/>

Bölüm Sonu...



BIM 202 - Bahar 2011

Programlama Dilleri

Prof. Dr. Tuğrul Yılmaz
e-posta: tyilmaz@mu.edu.tr

Sözdizim (syntax) ve Anlambilim (Semantics)

- Her programlama dilindeki geçerli programları belirleyen bir dizi kural vardır. Bu kurallar **sözdizim** (syntax) ve anlambilim (semantics) olarak ikiye ayrılır.
- Her deyimin sonunda noktalı virgül bulunması sözdizim kurallarına örnek oluştururken, bir değişkenin kullanılmadan önce tanımlanması bir anlam kuralı örneğidir.
- Bir ya da daha çok dilin sözdizimini anlatmak amacıyla kullanılan dile **metadil** adı verilir.
- Bu derste programlama dillerinin sözdizimini anlatmak için BNF (Backus-Naur Form) adlı metadil kullanılacaktır. Öte yandan, anlam tanımlama için böyle bir dil bulunmamaktadır.

Sözdizim (Syntax)	Anlam (Semantics)
Bir dilin sözdizim kuralları, bir deyimdeki her kelimenin nasıl yazılabileceğini belirler.	Bir dilin anlam kuralları ise, bir program çalıştırıldığında gerçekleşecek işlemleri tanımlar.

Sözdizim ve Anlambilim (devam)

- Sözdizim ve anlam arasındaki farkı, programlama dillerinden bağımsız olarak bir örnekle incelersek:
- Tarih gg.aa.yyyy şeklinde gösteriliyor olsun.

Sözdizim	Anlam	
10.06.2007	10 Haziran 2007	Türkiye
	6 Ekim 2007	ABD

- Ayrıca sözdizimindeki küçük farklar anlamda büyük farklılıklara neden olabilir. Bunlara dikkat etmek gereklidir:

```
while (i<10)  
{ a[i]= ++i; }
```

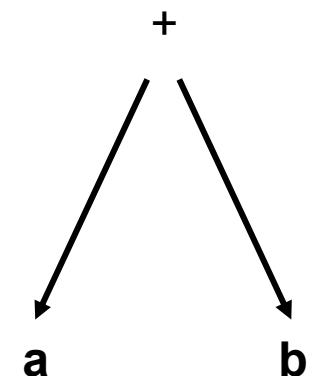
```
while (i<10)  
{ a[i]= i++; }
```

Soyut Sözdizim

- Bir dilin **soyut sözdizimi**, o dilde bulunan her yapıdaki anlamlı bileşenleri tanımlar. Örneğin; $+ab$ prefix ifadesi, $a+b$ infix ifadesi, ve $ab+$ postfix ifadesinde $+$ işlemcisi ve a ve b alt-ifadelerinden oluşan aynı anlamlı bileşenleri içermektedir. Bu nedenle ağaç olarak üçünün de gösterimi yandaki şekildeki gibidir.

■ Soyut Sözdizim Ağaçları

Bir ifadedeki işlemci/işlenen yapısını gösteren ağaçlara **soyut sözdizim ağaçları** adı verilir. Soyut sözdizim ağaçları, bir ifadenin yazıldığı gösterimden bağımsız olarak sözdizimsel yapısını gösterebilmeleri nedeniyle bu şekilde isimlendirilirler.



Soyut sözdizim ağaçları, uygun işlemcilerin geliştirilmesiyle diğer yapılar için de genişletilebilir.

Örneğin

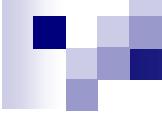
if $a > b$ then a else b

METİNSEL SÖZDİZİM

sonuc = 2* say + 18;

Sözlük-birim	Andaç
sonuc	Tanımlayıcı
say	
2	Tamsayı sabit
18	
=	Eşit işaretçi
*	Çarpım işlevci
+	Toplama işlevci
;	Noktalı virgül

- Hem doğal diller (Türkçe) hem de programlama dilleri (Java), bir alfabeteki karakter dizilerinden oluşurlar. Bir dilin karakter dizilerine **cümle** veya **deyim** adı verilir. Bir dilin sözdizim kuralları, o dilin alfabetesinden hangi karakter dizilerinin o dilde bulunduğularını belirlerler. En büyük ve en karmaşık programlama dili bile sözdizimsel olarak çok basittir.
- Bir programlama dilindeki en düşük düzeyli sözdizimsel birimlere **sözlükbirim (lexeme)** adı verilir. Programlar, karakterler yerine **sözlükbirimler** dizisi olarak düşünülebilir. Bir dildeki **sözlükbirimlerin** gruplanması ile dile ilişkin **andaçlar (token)** oluşturulur.
- Bir programlama dilinin metinsel sözdizimi, andaçlar ile tanımlanır. Örneğin bir tanımlayıcı; *toplam* veya *sonuc* gibi **sözlükbirimleri** olabilen bir andaçtır. Bazı durumlarda, bir andaçın sadece tek bir olası **sözlükbirimini** vardır. Örneğin, *toplama İşlemci* denilen aritmetik işlemci "+" simbolü için, tek bir olası **sözlükbirim** vardır.
- Boşluk (*space*), ara (*tab*) veya yeni satır karakterleri, **andaçlar** arasına yerleştirildiğinde bir programın anlamı değişmez.
- Yandaki örnekte, verilen C deyimi için **sözlükbirim** ve **andaçlar** listelenmiştir.

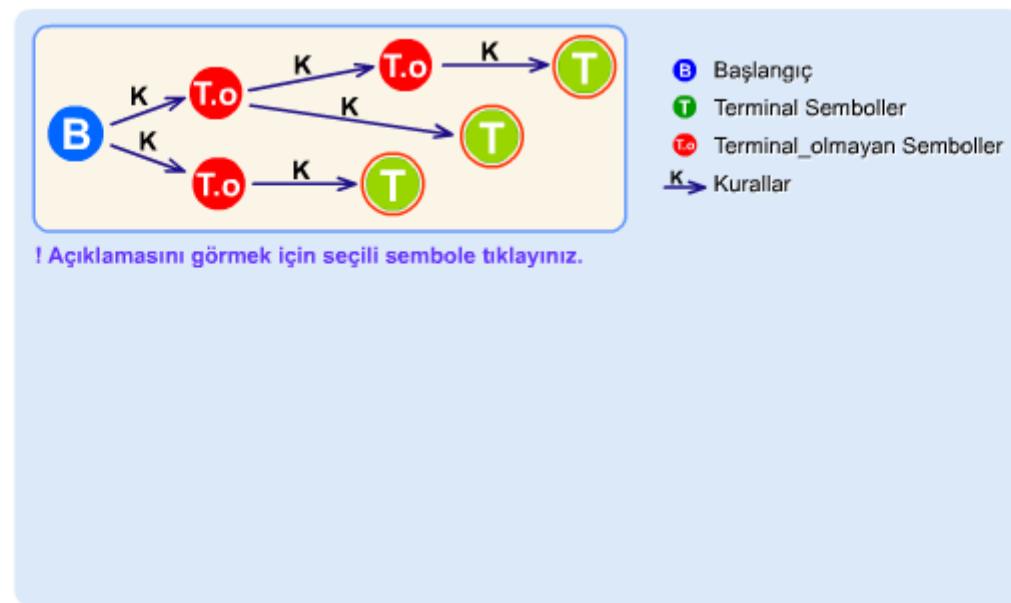


PROGRAMLAMA DİLLERİNDE GRAMER

- **Gramer**, bir programlama dilinin metinsel (somut) sözdizimini açıklamak için kullanılan bir gösterimdir. Gramerler, anahtar kelimelerin ve noktalama işaretlerinin yerleri gibi metinsel ayrıntılar da dahil olmak üzere, bir dizi kuraldan oluşur.

BNF: Backus-Naur Form

- **BNF (Backus-Naur Form)**, 1950'li yıllarda çeşitli gruplar tarafından yapılan çalışmaların sonucu olarak geliştirilen ve 1960 yılından beri programlama dilleri için standart olarak kullanılan metadildir. BNF kullanılarak sözdizimi tanımlanan ilk programlama dili ALGOL60'tır. Daha sonraları BNF'e yapılan eklemelerle oluşan dil ise **genişletilmiş BNF (extended BNF)** olarak adlandırılmıştır.
- BNF ile açıklanan bir gramer dört bölümden oluşur. Başlangıç (amaç) sembolü, terminal semboller, terminal_olmayan semboller ve kurallar'dan oluşan BNF gramer yapısı aşağıdaki şekildeki gibi gösterilebilir:



Gramer

- BNF kullanılarak, bir dilde yer alan cümleler oluşturulabilir. Bu amaçla, başlangıç sembolünden başlayarak, dilin kurallarının sıra ile uygulanması gereklidir. Bu şekilde cümle oluşturulmasına **türetme** (*derivation*) denir ve BNF türetmeli bir yöntem olarak nitelendirilir.
- Örnek gramer animasyonunda görülen dilin, atama görevini gören tek bir deyimi vardır. Bir **program**, **begin** ile başlar, **end** ile biter.

! Örnek Gramer yapısını görmek için
"Başla" düğmesini tıklayınız.

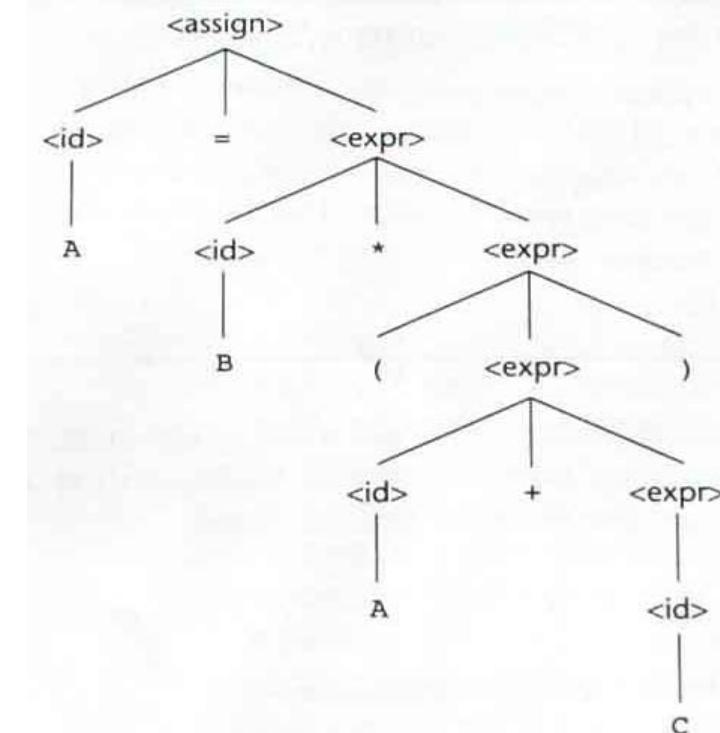
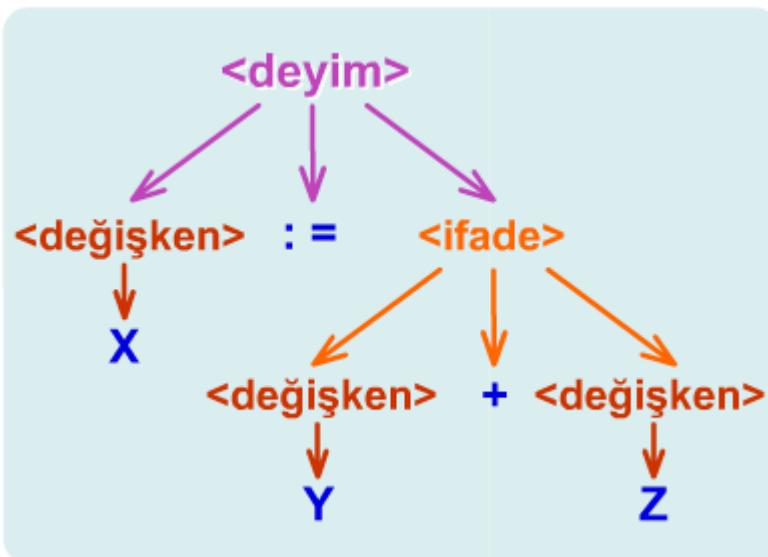
Başla

begin
z:= y+y ;
x:= z+y
end

Başla

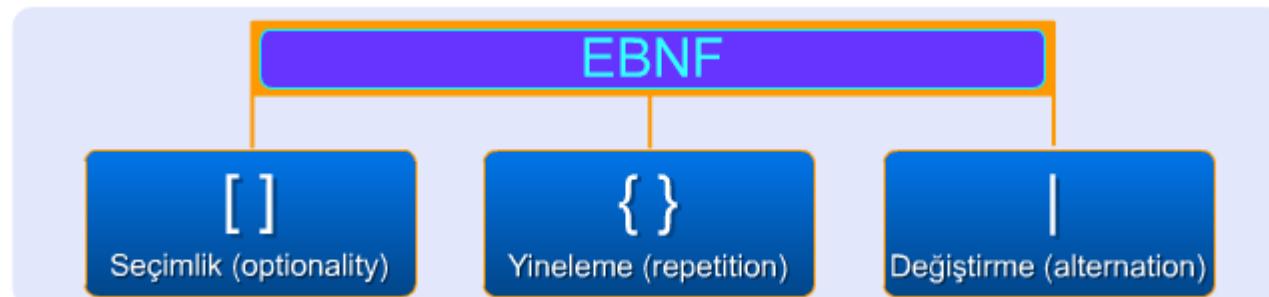
Ayrıştırma Ağaçları

- Gramerler, tanımladıkları dilin cümlelerinin hiyerarşik sözdizimsel yapısını tarifleyebilirler. Bu hiyerarşik yapılara **ayrıştırma (parse) ağaçları** denir. Bir ayrıştırma ağaçının en aşağıdaki düğümlerinde terminal semboller veya *andaçlar* yer alır. Ayrıştırma ağaçının diğer düğümleri, dil yapılarını gösteren terminal olmayanları içerir.
- Ayrıştırma ağaçları ve türetmeler birbirleriyle ilişkili olup, birbirlerinden türetilabilirler. Aşağıdaki şekilde yer alan ayrıştırma ağıacı, "Gramerler ve Türetmeler" bölümünde tanımlanan gramere göre, bir önceki sayfada türetilmiş olan "örnek türetme" deyiminin, " $x := y + z$ ", yapısını göstermektedir.



Genişletilmiş BNF

- BNF'nin okunabilirliğini ve yazılabilirliğini artırmak amacıyla, BNF'e bazı eklemeler yapılmış ve yenilenmiş BNF sürümlerine **genişletilmiş BNF** veya kısaca **EBNF** adı verilmiştir. EBNF'te Seçimlik (optionality), Yineleme (repetition) ve Değiştirme (alternation) olmak üzere üç özellik yer almaktadır:



```
<seçimlik_deyim> -> If (<mantıksal>) <deyim> [else <deyim>];
```

Genişletilmiş BNF - EBNF

BNF:

```
<expr> → <expr> + <term>
      | <expr> - <term>
      | <term>
<term> → <term> * <factor>
      | <term> / <factor>
      | <factor>
<factor> → <exp> ** <factor>
      | <exp>
<exp> → ( <expr> )
      | id
```

EBNF:

```
<expr> → <term> [(+ | -) <term>]
<term> → <factor> [(* | /) <factor>]
<factor> → <exp> { ** <exp> }
<exp> → ( <expr> )
      | id
```

BNF devam

- Bir kuralın sağ tarafında, istenilen sayıda yinelenebilecek veya hiç yer almayabilecek bir bölümü göstermek için { } kullanımı eklenmiştir.
- Bir grup içinden tek bir eleman seçilmesi gerektiği zaman seçenekler, parantezler içinde birbirlerinden "veya" işlemcisi "|" ile ayrılarak yazılabilir. Aşağıda pascal örneği verilmiştir.

```
<for_deyimi> -> for <değişken> := <ifade> (to | down to) <ifade> do  
    <deyim> ;
```

BNF uygulaması

Örnek

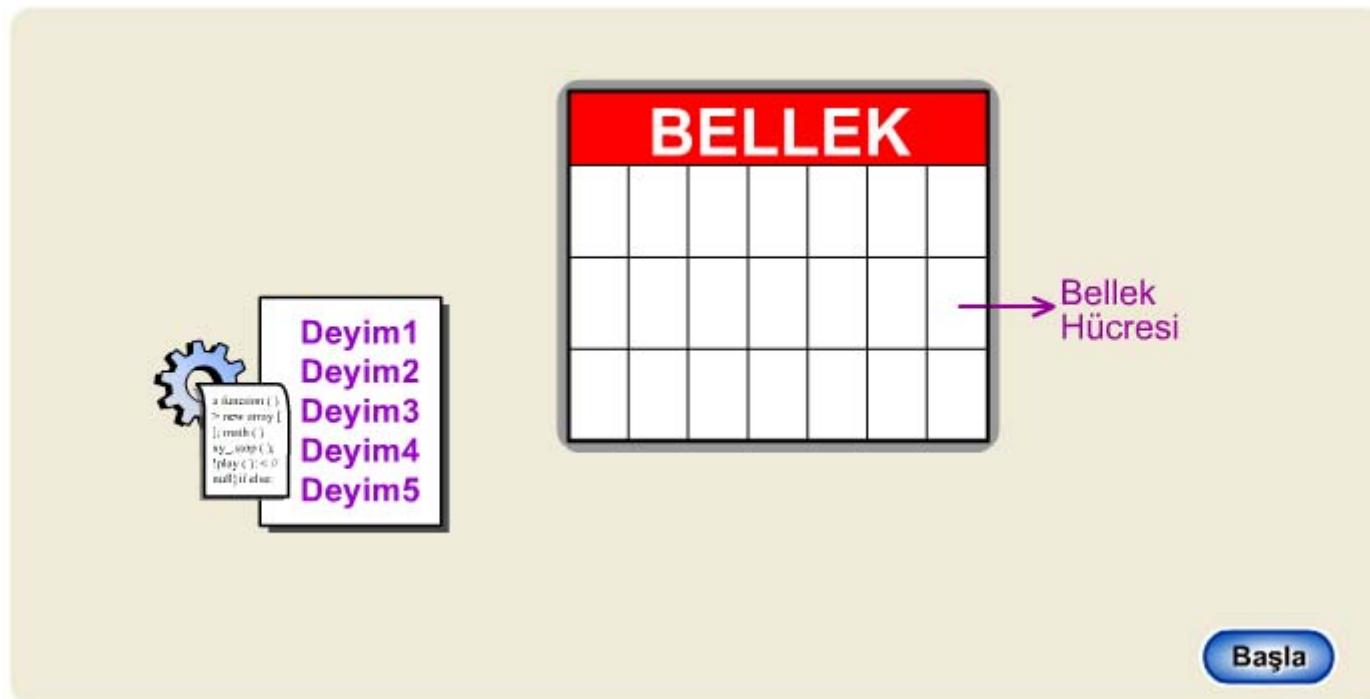
$<\text{amaç}> := [\text{x}] \text{ y } \{\text{z}\}$

! " $<\text{amaç}> := [\text{x}] \text{ y } \{\text{z}\}$ " kuralına göre geçerli olan karakter dizgilerini görmek için "Başla" düğmesini tıklayınız.

Başla

Temel Programlama Elemanları

- Geleneksel bilgisayar mimarisi *von Neumann* mimarisi olarak adlandırılır. Bu mimari, her bellek hücresinin özgün bir adres ile tanımlandığı ana bellek kavramına dayanmaktadır. Bir bellek hücresinin içeriği, bir değerin belirli bir yönteme göre kodlanmış gösterimidir. Bu içerik, programların çalışması sırasında okunabilir ve değiştirilebilir.
- *Buyurgan (Imperative)* programlama, *von Neumann* mimarisindeki bilgisayarlara uygun olarak programların işlem deyimleri ile bellekteki değerleri değiştirmesine dayanır.



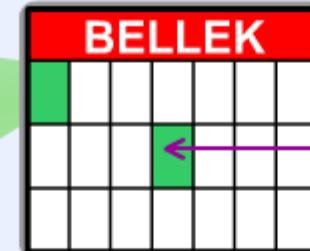
Atama İşlemi

- buyurgan (Imperative) programlamada en temel işlem atama işlemidir. Atama sembolü programlama dillerinde farklı şekilde gösterilebilir, ancak tüm programlama dillerinde atama sembolünün anlamı, sağ taraftaki değerin sol taraftaki değişkene aktarılmasıdır.
- **sum = 0; java, C, C++ vs.**
- **sum := 0; pascal, algol, vs.**

Değişkenler (identifiers, names)

ÖZELLİKLER

İsim
Adres
Değer
Tip
Yaşam Süresi
Kapsam



- *l-value*: Değişkenin adresidir
- *r-value*: Değişkenin değeridir.

İsimler

- İsimler, programlama dillerinde, değişkenlerin yanısıra, etiketler, altprogramlar, parametreler gibi program elemanlarını tanımlamak için kullanılırlar.
- İsimleri tasarlamak için programlama dillerinde farklı yaklaşımlar uygulanmaktadır.
 - en fazla uzunluk
 - büyük küçük harf duyarlılığı
 - özel kelimeler

İsimler – en fazla uzunluk

- Programlama dillerinde bir ismin en fazla kaç karakter uzunluğunda olabileceği konusunda farklı yaklaşımlar uygulanmıştır. Önceleri programlama dillerinde bir isim için izin verilen karakter sayısı daha sınırlı iken, günümüzdeki yaklaşım, en fazla uzunluğu kullanışlı bir sayıyla sınırlamak ve çoklu isimler oluşturmak için altçizgi "_" karakterini kullanmaktadır.
- örnek: cok_uzun_bir_degisken_olabilir_ama_yine_de_kisa_mi
- 70 ve 80 li yıllarda moda olan altçizgi karakteri günümüzde yerini "**camel**" notasyonu tabir edilen bir yönteme bırakmıştır.
- örnek: cokUzunBirDegiskenOlabilirAmaYineDeKisaMi

İsimler – en fazla uzunluk

■ Dil örnekleri

- FORTRAN I: maksimum 6
- COBOL: maksimum 30
- FORTRAN 90 ve ANSI C: maksimum 31
- Ada ve Java: limit yok, bütün karakterler kullanılıyor
- C++: limit yok ama derleyici hazırlayanların koydukları limitler var
- LISP: limit yok

Küçük-Büyük Harf Duyarlılığı (Case Sensitivity)

- Birçok programlama dilinde, isimler için kullanılan küçük ve büyük harfler arasında ayırım yapılmazken, bazı programlama dilleri (Örneğin; C, Java) isimlerde küçük-büyük harf duyarlığını uygulamaktadır. Bu durumda, aynı harflerden oluşmuş isimler derleyici tarafından farklı olarak algılanmaktadır.
- Dezavantaj: okunabilirlik (birbirine benzeyen isimler fakat farklılar)
 - C++ ve Java da daha kötü çünkü önceden tanımlanmış böyle isimler var (örneğin `IndexOutOfBoundsException`)
- C, C++, ve Java isimleri büyük-küçük harf duyarlı
- Fortran I-77: Harfler sadece büyük. 77'de okunurken küçük varsa büyütülüyor.
- Diğer dillere bakmak lazım.

Özel Kelimeler

- Özel kelimeler, bir programlama dilindeki temel yapılar tarafından kullanılan kelimeleri göstermektedir.
- **Anahtar Kelime:** Bir anahtar kelime (*keyword*), bir programlama dilinin sadece belirli içeriklerde özel anlam taşıyan kelimelerini göstermektedir. Örneğin FORTRAN'da *REAL* kelimesi, bir deyimin başında yer alıp, bir isim tarafından izlenirse, o deyimin tanımlama deyimi olduğunu gösterir. *REAL ELMA* gibi. Eğer *REAL* kelimesi, atama işlemcisi "=" tarafından izlenirse, bir değişken ismi olarak görülür. *REAL = 87.6* gibi. **Bu durum dilin okunabilirliğini azaltır.**
- **Ayrılmış Kelime:** Öte yandan, ayrılmış kelime (*reserved word*), bir programlama dilinde bir isim olarak kullanılamayacak özel kelimeleri göstermektedir. Örneğin Pascal'da, *for*, *begin*, *end* gibi kelimeler, C'de *int*, *float*, *double* gibi kelimeler isim olarak kullanılamaz ve ayrılmış kelime olarak nitelendirilir.

Özel Kelimeler

- Bazı dillerde de önceden tanımlanmış kelimelerin değiştirilmesine izin verilebilir. Örneğin Ada dilinde Integer ve Float önceden tanımlanmıştır. Ancak program içinde yeniden tanımlanabilirler.

Veri Tipi

- Bir **veri tipi**, aynı işlemlerin tanımlı olduğu değerler kümesini göstermektedir.
- Bir değişkenin tipi, değişkenin tutabileceği değerleri ve o değerlere uygulanabilecek işlemleri gösterir. Örneğin; tamsayı (*integer*) tipi, dile bağımlı olarak belirlenen en küçük ve en büyük değerler arasında tamsayılar içerebilir ve sayısal işlemlerde yer alabilir.
- Veri tipleri, programlama dillerinde önemli gelişmelerin gerçekleştiği bir alan olmuş ve bunun sonucu olarak, programlama dillerinde çeşitli veri tipleri tanıtılmıştır. **Tipler temel (primitive) ve yapısal (composite) olarak gruplandırılabilir.**
- **Temel tipler**, çoğu programlama dilinde yer alan ve diğer tiplerden oluşmamış veri tiplerini göstermektedir.
 - **C Ada**
 - boolean Boolean = {*false*, *true*}
 - char Character = { . . . , 'a', . . . , 'z', . . . , '0', . . . , '9', . . . , '?', . . . }
 - int Integer = { . . . , -2, -1, 0, +1, +2, . . . } → {-2 147 483 648, . . . , +2 147 483 647}
 - float Float = { . . . , -1.0, . . . , 0.0, . . . , +1.0, . . . }
- **Yapısal tipler** ise çeşitli veri tiplerinde olabilen bileşenlerden oluşmuştur. Bir yapısal tipin elemanları, tipin bileşenlerini oluşturmaktadır. Bir yapısal tipteki her bileşenin, tip ve değer özellikleri bulunmaktadır.

Temel tipler

- Bütün dillerde Boolean farklı bir veri tipi değildir. Örneğin, C++ 'da **bool** vardır fakat bunlar “small integer”dır. 0 false, diğer sayılar true anlamına gelir.
- Bütün dillerde ayrı bir Character tipi yoktur. Örneğin C, C++ ve JAVA'da **char**, tipi vardır ama bunlar aslında “small integer” olurlar; içerde aralarında fark yoktur.
- Bazı dillerde birden çok “integer” temel tipi vardır. Örneğin Java'da
 - **byte** {-128, . . . , +127},
 - **short** {-32 768, . . . , +32 767},
 - **int** {-2 147 483 648, . . . , +2 147 483 647},
 - **long** {-9 223 372 036 854 775 808, . . . , +9 223 372 036 854 775 807}.
- Bazı dillerde birden çok gerçek sayı tipi vardır. Örneğin C, C++ ve JAVA'da **float** ve **double**..

Yapısal tiplere örnekler

Ada örneği

```
type Month is (jan, feb, mar, apr, may,  
    jun,jul, aug, sep, oct, nov, dec);  
type Day_Number is range 1 .. 31;  
type Date is  
    record  
        m: Month;  
        d: Day_Number;  
    end record;
```

Java Örneği

```
enum Month {jan, feb, mar, apr, may, jun,  
    jul, aug, sep, oct, nov, dec};  
struct Date {  
    Month m;  
    byte d;  
};
```

Sabitler

- Bir **sabit**, belirli bir tipteki bir değerin kodlanmış gösterimini içeren ancak programın çalıştırılması sırasında değiştirilemeyen değerlere verilen isimdir. Bir sabit genellikle ilkel tipte bir değerdir. Örneğin 568, bir tamsayı sabittir.
- Bir değişken, bir bellek yerine bağlandığında bir değere de bağlıyorsa ve daha sonra bu değer değiştirilemiyorsa o değişkene **isimlendirilmiş sabit** denir. Sabit bir değerin programda birçok kez yinelenmesi durumunda, isimlendirilmiş sabitlerin kullanılması yararlıdır. Örneğin 3.14159 değeri yerine *pi* isminin kullanılması, programın okunabilirliğini artırır.
- Bir başka örnek olarak, 50 elemanlı bir diziyi işleyen bir programı düşünelim. Bu programda birçok kez (örneğin, dizi tanımlamada, döngülerde vb.) dizi sınırlına başvuru yer alır. Bu değerin programın başında isimlendirilmiş sabit olarak tanımlanması, programın okunabilirliğini ve güvenilirliğini artırır.
- Isimlendirilmiş sabitlerin tanımlanması için, Pascal'da **const** tanımlayıcısı kullanılır. C'de ise isimlendirilmiş sabit tanımlamak için **#define** kullanılır.

İşleçlerin (İşlemci) genel özellikleri

- İşleçlerin genel özellikleri işlenen sayısı, işlemcin yerı, öncelik ve birleşmeliğ (associativity) olmak üzere dört tanedir.
- **İşlenen Sayısı (arity)** : Bir işaret, alabileceği işlenen sayısına göre **tekli** (unary), **ikili**(binary) veya **çoklu** (nary) olabilir.
 - $p = &a;$ (C – unary, prefix)
 - $i = -5;$ (C – unary, prefix)
 - $i = j - 5;$ (C – binary, infix)
 - $i = i + j;$ (C – binary, infix)
 - $i = i + j + 5;$ (C – binary, infix)
 - $(plus i j 5)$ (LISP – nary, prefix)
 - $i++;$ (C – unary, postfix)
- **İşleçin Yeri:** Çoğu işaret işlenenleri arasında yazılmakla birlikte, bazı işaretler, işlenenlerinden önce veya sonra da yazılabılır.
 - İşlemciler bir ifadede, işlenenlerden önce(*prefix*), işlenenler arasında (*infix*) ve işlenenlerden sonra (*postfix*) olmak üzere üç şekilde yer alabilirler.

İşleçlerin öncelikleri

- İşlemcilerin öncelikleri, birden çok işaretin yer aldığı bir ifadede, parantez kullanılmadığında, bir ifadenin bileşenlerinin değerlendirilme sırasını belirler.
- İkili işaretler, *infix* gösterimde, " $a+b$ " de olduğu gibi işlenenleri arasına yazılır. Ancak *infix* gösterimdeki sorun, birden çok işaretin birlikte yer aldığı bir ifadede görülür. Örneğin; " $a+b*c$ " gibi bir ifadenin değerlendirilmesi nasıl olacaktır? Sonuç, " a " ve " $b*c$ " nin toplamı mı yoksa " $a+b$ " ve " c " nin çarpımı mıdır? Bu soruların yanıtları işaretlerin **öncelik** ve **birleşmeliilik** kavramları ile açıklanabilir.
- Her işaretin, programlama dili tasarılanırken önceden belirlenmiş bir önceliği vardır. Daha yüksek bir öncelik düzeyinde yer alan bir işaret, işlenenlerini daha düşük bir düzeydeki bir işaretden önce alır. Geleneksel bir kural olarak, sayısal işaretlerden "*" ve "/", toplama "+" ve çıkardan "-" daha yüksek önceliğe sahiptir. Bu nedenle yukarıdaki ifadede, "*" işaretci, işlenenlerini "+" dan önce alır ve " $a+b*c$ ", " $a+(b*c)$ " ye eşittir.

İşleçlerin öncelikleri

Örnek

$$a+b*c$$

Başla

Birleşmelilik

- Bir ifadede aynı öncelik düzeyinde iki işlemci bulunuyorsa, hangi işlemcinin önce değerlendirileceği dilin **birleşmelilik** kuralları ile belirlenir. Bir işlemci, sağ veya sol birleşmeli olabilir.
 - **Sol** birleşmeli: Bir işlemcinin birden çok kez yer aldığı bir ifadedeki alt ifadeler, soldan sağa olarak grüplenirse, işlemci sol birleşmeli olarak adlandırılır. Aritmetik işlemcilerden "+", "-", "*" ve "/" sol birleşmelidir.
 - **Sağ** birleşmeli: Öte yandan, eğer bir işlemcinin birden çok kez yer aldığı bir ifadedeki alt ifadeler, sağdan sola grüplenirse işlemci, sağ birleşmeli olarak adlandırılır. Üs alma işlemcisi sağ birleşmelidir.

Örnek

$$4 - 2 - 1$$

1.İşlem

2.İşlem

Örnek

$$2^{3^4} = 2^{81}$$

Niteliğine Göre İşlemciler

- **İşlemciler**, işlenenlerin niteliğine göre *sayısal*, *ilişkisel* veya *mantıksal işlemciler* olabilirler.
- **Sayısal işlemciler**, sayısal işlenenlere uygulanan işlemcilerdir. Kullanılan semboller programlama dillerinde farklılık gösterebilmekle birlikte, genel olarak üs alma, toplama, çıkarma, mod, çarpma, bölme gibi işlemcilerdir.
- **Bir ilişkisel işlemci**, iki işleneninin değerlerini karşılaştırır ve eğer mantıksal (Boolean) tipi dilde tanımlı bir veri tipi ise mantıksal tipte bir sonuç oluşturur.
 - Genellikle, ilişkisel işlemcilerle kullanılabilen işlenenler, sayısal, karakter veya sıralı (ordinal) tiplerde olurlar.

	Pascal	C
Eşit	=	==
Eşit değil	<>	!=
Küçük	<	<
Büyük	>	>
Küçük veya eşit	<=	<=
Büyük veya eşit	>=	>=

Niteliğine Göre İşlemciler

- Mantıksal işlemciler, sadece mantıksal (Boolean) işlenenleri alırlar ve mantıksal değerler, DOĞRU ve YANLIŞ üretirler.
 - Mantıksal işlemciler, AND (ve), OR(veya), NOT (değil) XOR(dışlayan veya) gibi işlemleri de içerebilirler. Mantıksal işlemcilerde öncelik sıralaması genellikle NOT, AND ve OR şeklindedir.

Örnek

$$a+10 > b^*5$$

önce önce

NOT: İlk önce aritmetik işlem, ardından mantıksal işlem yapılır.

Örnek

$$a > 0 \text{ or } b < 100$$

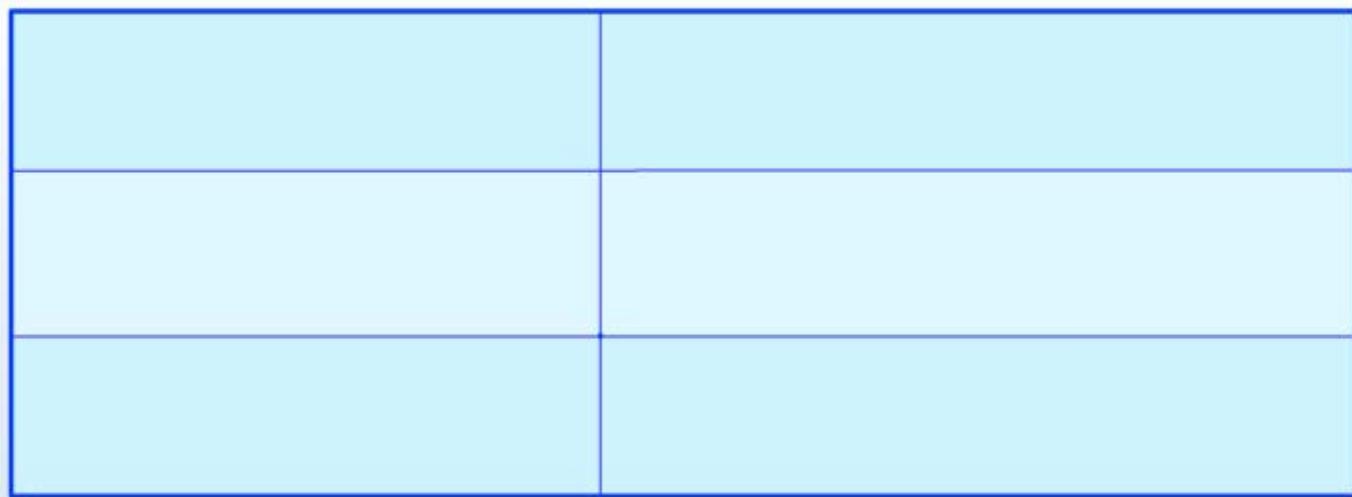
NOT: Öncelik "or"da olduğu için ifade geçersizdir. Bu yüzden doğru söz dizimi;

$$(a > 0) \text{ or } (b < 100)$$

İşlemci (işleç) yükleme

- İşlemcilerin anlamlarının, işlenenlerin sayısına ve tipine bağlı olarak belirlenmesine **işlemci yüklemesi** (*operator overloading*) denir.
- Sayısal işlemciler, programlama dillerinde sıkılıkla birden çok anlamda kullanılırlar. Örneğin "+", hem tamsayı hem de kayan-noktalı toplama için kullanılır ve bazı dillerde, sayısal işlemlere ek olarak karakter dizgilerin birleştirilmesi için de kullanılır. İşlemci yüklemelerinde, "+" da olduğu gibi, benzer anlamlarda olmayıabilir.
- Bir diğer örnek olan '-' işlemcisi, hem bir sayısal değerin negatif olduğunu belirtmek için tekli işlemci olarak, hem de ikili bir işlemci olan sayısal çıkarma işlemini göstermek için kullanılır. Her ne kadar işlemcinin iki kullanımında anlamsal yakınlık varsa da, bir ifadede yanlışlıkla birinci işlenenin unutulması, derleyici tarafından hata olarak algılanmayacak ve ikinci işlenen negatif değer olarak kabul edilecektir. Bu ve benzeri işlemci yüklemeleri, fark edilmesi güç hatalara neden olabilmektedir.

Örnek



! + işlemcisinin hangi anlamlarda kullanıldığını görmek için
"Başla" düğmesini tıklayınız.

Başla

Deyimler (Statements)

- **Deyimler**, bir programdaki işlemleri göstermek ve akışı yönlendirmek için kullanılan yapılardır. Deyimler **basit** veya **birleşik** olabilirler. Basit deyimlere örnek olarak atama deyimi verilebilir. Birleşik deyimler ise bir dizi deyimin tek bir deyime soyutlanmasılığını sağlarlar. Birleşik bir deyimde yer alan deyimleri belirlemek için basit deyimlerden ayrı bir sözdizime gereksinim vardır. Örneğin Pascal'da, birleşik deyimler *begin* ve *end* anahtar kelimeleri arasında, C de “{}” parantezleri arasında gruplanır.
- Programlarda akışı yönlendirmek için seçimli deyimler (*if-then- else*, *case* deyimleri gibi) ve yinelemeli deyimler (*while*, *for* deyimleri gibi) kullanılabilir.
- **Altprogramlar**, bir dizi deyimin gruplanmasıını ve bir isim ile gösterilmesini sağlarlar. Altprogramlar, bir başlık, yerel tanımlamalar bölümü ve işlemlerin yer aldığı bir gövde ile tanımlanır. Altprogram başlığı, altprogram ismini ve varsa tipleriyle birlikte altprogramın parametrelerini belirtir. Altprogram gövdesi, altprogram etkin olduğunda çalıştırılacak deyimlerden oluşur. Altprogramlar ve parametre aktarımları daha sonra inceleneciktir.

Bağlama (Binding) Kavramı

- Programlarda yer alan tüm program elemanlarının - örneğin; değişkenler, altprogramlar vb.- çeşitli özellikleri vardır. Değişkenlerin isim, adres, değer gibi çeşitli özelliklerini incelemiştik.
- Bir özellikle bir program elemanı arasında ilişki kurulmasına **bağlama** (*binding*) denir. Çeşitli programlama dilleri, özelliklerin program elemanlarına **bağlanma zamanı** ve bu özelliklerin **durağan** (*static*) veya **dinamik** (*dynamic*) olması açısından farklılıklar göstermektedir.

Program elemanları

Değişkenler (x,y,z,...)

Alt programlar

...



Özellikler

Adres

Değer

Bağlama Zamanı

- Bir programlama dilinde çeşitli bağlamalar farklı zamanlarda gerçekleşebilir.



İlk üç grupta olduğu gibi bağlamanın çalışma zamanından önce gerçekleştiği durumda bağlama, çalışma zamanında değiştirilemez ve **durağan (static) bağlama** olarak adlandırılır. Çalışma zamanında gerçekleşen bağlamalar ise, çalışma süresince değiştirilebilirler ve **dinamik (dynamic) bağlama** olarak adlandırılırlar.

Örnek

```
int puan;  
.....  
puan = puan + 3;
```

- puan değişkeninin alabileceği tipler
- puan değişkeninin alabileceği değerler
- + sembolünün alabileceği anlamlar

- puan değişkeninin programdaki tipi
- puan değişkeninin o deyimdeki değeri
- + sembolünün anlamı

Dil Tasarım
Zamanında Bağlama

Dil Gerçekleştirme
Zamanında Bağlama

Derleme
Zamanında Bağlama

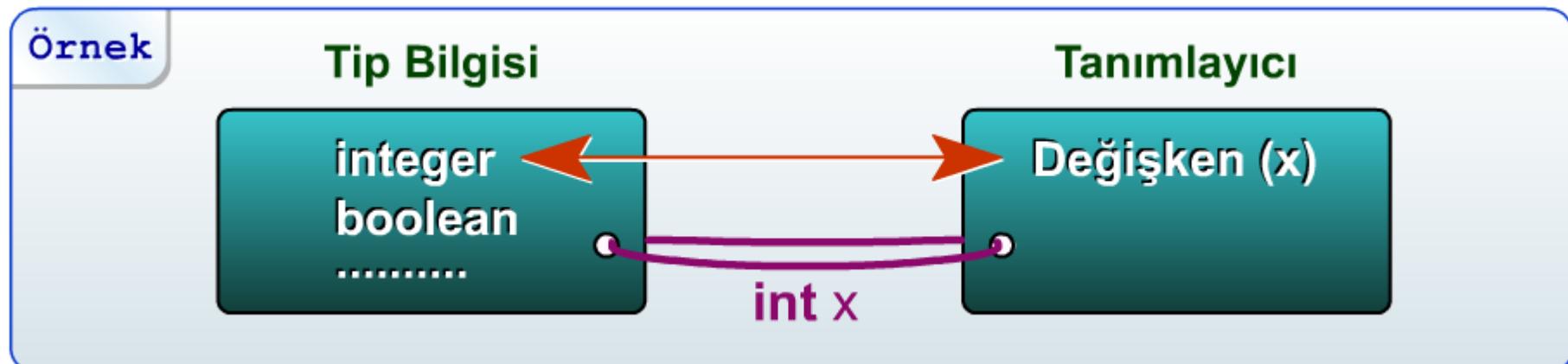
Çalışma
Zamanında Bağlama

! Yukarıdaki listede bulunan bağlamaların gerçekleştiği zamanı görmek için "Başla" düğmesini tıklayınız.

Başla

Tip Bağlama (Type Binding)

- Tip bilgisi, bir tanımlayıcı ile ilişkilendirilince, tanımlayıcı, o tiple bağlanmış olur. Örneğin, birçok programlama dilinde bir programda bir değişkene başvuru yapılmadan önce, değişkenin bir veri tipi ile bağlanması gereklidir.



Bağlanması kavramı

■ Bağlanması tipleri

- tip nasıl belirlenir?
- bağlanması ne zaman gerçekleşir?

Bağlanması kavramı

Aşağıdaki C atama komutunu ele alalım:

say = say + 5;

- “say” değişkeninin tipi derleme sırasında bağlanır (belirlenir).
- “say” değişkeninin alabileceği değerler derleyicinin tasarıımı sırasında belirlenir.
- “+” operatör symbolünün anlamı işaretlerin tipleri ile birlikte derleme zamanında bağlanır.
- “5”in iç gösteriminin nasıl olacağı derleyicinin tasarlandığı zaman bağlanır.
- “say” değişkeninin değeri yukarıdaki komutla program çalışırken bağlanır.

Durağan Tip Bağlama (Static type binding)

- Tiplerin isimlerle derleme zamanında bağlandığı diller, **statik tip bağlamalı diller** olarak nitelendirilirler. statik tip bağlamalı dillerde bir değişken, *integer* tipi ile bağlanmışsa, değişkenin gösterdiği değer çalışma zamanında değişse de, gösterdiği değerin tipi her zaman *integer* olmalıdır. Örneğin FORTRAN, Pascal, C ve C++'da bir değişkenin tip bağlaması durağan olarak gerçekleşir ve çalışma süresince değiştirilemez.
- Statik tip bağlamalı dillerde derleyici, tip hatalarını, program çalıştırılmadan önce yakalar.
- Statik tip bağlamaları, **örtülü (*implicit*)** ve **açıkça (*explicit*)** olmak üzere iki tür tanımlama ile gerçekleştirilebilir.

Statik Tip Bağlama

- **Örtülü tanımlama:** Örtülü tanımlamada, tanımlama deyimleri kullanılmaz ve değişkenlerin tipleri, varsayılan (*default*) kurallar ile belirlenir. Bu durumda bir değişken isminin programda ilk kullanıldığı deyim ile değişkenin örtülü tip bağlaması oluşturulur.
- Örtülü tanımlamalar, program geliştirme sırasında programcıya yardımcı olsalar da, yazım yanlışlığı gibi hataların derleme sırasında yakalanmasını engelledikleri için ve programcının tanımlamayı unuttuğu değişkenlere varsayılan olarak tip bağlanması ile programda fark edilmesi güç hatalara yol açabildikleri için programlama dilinin güvenilirliğini azaltırlar. Ancak örtülü ve dışsal tanımlama ile tip bağlama, anlamsal açıdan aynıdır.
- PL/I, BASIC ve FORTRAN gibi dillerde örtülü tanımlamalar bulunmasına karşın günümüzde çoğu programlama dili, değişkenlerin dışsal olarak tanımlanmasını gerektirmektedir. Örnek:
 - Fortran'da I-N arasındaki harflerle başlayan değişkenler tam sayı değişkenlerdir.
 - Basic de son karakter \$ ise karakter dizisi değişkenidir.
- **Dışsal tanımlama**
- Dışsal tanımlamada, bir değişken, programda yer alan bir tanımlama deyimi ile belirli bir tip ile bağlanır. Örnek:
 - int toplam;
 - float ortalama;

Dinamik Tip Bağlama

- Bir programlama dilinde bir değişkenin tipi çalışma zamanında, değişkenin bağlandığı değer ile belirleniyorsa, dil, **dinamik tip bağlamalı** olarak nitelendirilir. Dinamik tip bağlamalı dillerde bir değişken, atama sembolünün sağ tarafında bulunan değerin, değişkenin veya ifadenin tipine bağlanır ve değişkenin tipi, çalışma zamanında değişkenin yeni değerler alması ile değiştirilir.
- Değişkenlere Dinamik olarak tip bağlanması, programlama açısından esneklik sağlar. Örneğin, Dinamik tip bağlamalı bir dilde bir sıralama programındaki değişkenlerin tipleri çalışma zamanında belirleneceği için, tek bir program, farklı tipteki değerlerin sıralanması amacıyla kullanılabilir. Halbuki, C veya Pascal gibi durağan tip bağlamalı programlama dillerinde, bir sıralama programı sadece tek bir veri tipi için yazılabilir ve bu veri tipi program geliştirilirken bilinmelidir.
- Dinamik tip bağlamalı diller, genellikle yorumlayıcı ile gerçekleştirilirler.
- Bu tip dillerde hata yakalama özelliği zayıftır.

Dinamik bağlanma

- Örnek: JavaScript, perl, PHP
- Atama cümleleri ile yapılır, örneğin, JavaScript'de

```
list = [2, 4.33, 6, 8];  
list = 17.3;
```

- Avantaj: esneklik (tip bağımsız program birimleri)
 - Dezavantaj:
 - yavaş (dinamik veri tipi kontrolü ve yorumlaması)
 - Bilgisayarın tip hatalarını kontrol etmesi zor.
- Tip çıkarsaması (ML, Miranda, ve Haskell)
 - Atama yerine referansın sağladığı ortam nedeniyle tipin belirlenmesi.

Bellek bağlama

- Bir değişkene bağlanan bir bellek hücresi, kullanılabilir bellek hücreleri arasından seçilir ve bu işlemeye **bellek yeri ataması** (*allocation*) denir.
- Bir değişkenin kullandığı bellek hücresini geri vermesi ise **belleğin serbest bırakması** (*deallocation*) işlemi olarak nitelendirilir.
- Bir değişkenin **yaşam süresi**, o değişkenin belirli bir bellek yerine bağlı kaldığı süredir. Bu nedenle bir değişkenin yaşam süresi, bir bellek hücresi ile bağlandığı zaman başlar ve bellek hücresini bıraktığı zaman sona erer.

Statik bellek

- Statik bellekler programın yürütmesi başlamadan bağlanır, ve yürütme sürdüğü sürece bağlı kalırlar.
Örnek: bütün FORTRAN 77 değişkenleri, C statik değişkenleri.
- C++, C#, ve Java'da “class” içinde yapılan “static” tanımlamaları belleğin yaşam süresini değil, onun bir “class” değişkeni olduğunu, bir nesnenin anlık değişkeni olmadığını gösterir.
- Avantajları: verimli, hızlı (doğrudan adresleme), geçmişe duyarlı alt program desteği.
- Dezavantaj: esnek değil (özyineleme yok).

Yığıt dinamik bağlama

- Yığıt dinamik bağlama (Stack-dynamic—Storage binding) değişkenlerin tanımlanması ile ilgili komutlar yürütülürken gerçekleştirilir. Tanımlandığı blok aktif kaldığı sürece yaşar.
- Sayısal değişkenlerin bellek adresi hariç bütün özellikleri statik olarak belirlenmiştir.
Örnek: C alt programlarının veya java metodlarının lokal değişkenleri gibi.
- Bazı dillerde her noktada yığıt dinamik tanımlama yapılabilir.
- Metotlar içinde tanımlanan Java, C++ ve C# değişkenlerinin hepsi yığıt dinamiktir. Aynı şekilde Ada'da altyordamlarda tanımlanan değişkenlerde (bellek yığını hariç) yığıt dinamiktir.
- Avantaj: öz yinelemeye izin verir; depolamayı korur; az bellek harcanmasına neden olur.
- Dezavantaj:
 - Bellekten yer almak ve geri vermenin yarattığı işlem yükü.
 - Alt programlar geçmişe hassas değildir. Çıkılıncá bütün bilgiler unutulur.
 - Inefficient references (indirect addressing)

Bellek yığını (heap)

- **Açıkça bellek yığını bağlama** – Program yürütülürken programcı tarafından komutla bellek alınması.
- Sadece gösterici (pointer) ile erişilir.

Örnek: C++ dinamik nesneleri(new ve delete ile):

```
int *intp; // gösterici tanımla  
...  
intp = new int; //yeni bellek ayır  
...  
delete intp;// yeni belleği sil (gerekli)
```

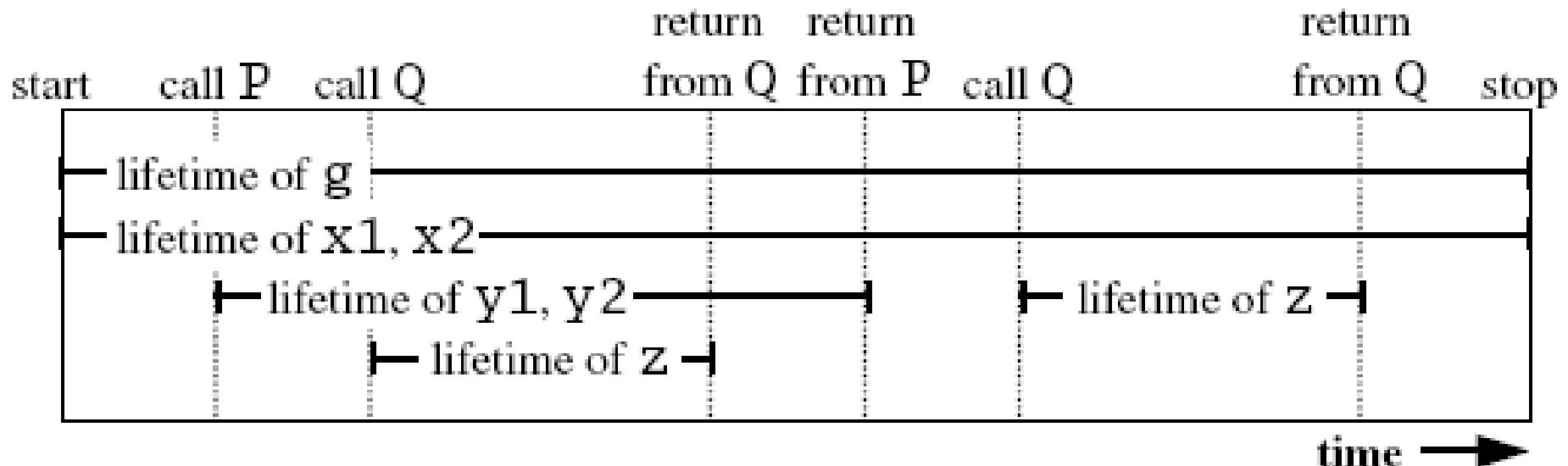
Örnek: Bütün Java nesneleri: “delete” yok ancak kullanılmayan bellekleri toplayan örtülü bir çöp toplama yapısı var (implicit garbage collection).

Örnek: C#’da da “delete” yok ancak örtülü olarak kullanılmayacaklar sisteme geri iade ediliyor.

- Avantaj: dinamik bellek yönetimi sağlar.
- Dezavantaj: yönetimi zor bu nedenle güvenilir değil.

Örtülü dinamik bellek yiğini

- **Örtülü dinamik bellek yiğini** – Bellek bağlamaları atama komutları ile.
Örnek: Perl ve JavaScript’de bütün dizgi (string) ve dizilim (array) atamaları.
- avantaj: esneklik.
- Dezavantaj:
 - Yetersiz çünkü bütün özellikler dinamik.
 - Hata farketme yetersizliği.



```

int g;
void main () {
    int x1; float x2;
    . . . P(); . . . Q(); . . .
}
void P () {
    float y1; int y2;
    . . . Q(); . . .
}
void Q () {
    int z;
    . . .
}

```

Tip Kontrolü

- İşlenen (operands) ve işaretç (operators) tanımlarını genişletirsek: altprogramlar (subprograms) işletmen, parametreleri işlenenler; atamalar (assignments) işletmen, değişkenler ve ifadeler işlenenler şeklinde tanımlanabilir.
- Tanım: İşlenenlerin işletmenlere uygunluğuna bakmak tip kontrolü ([Type checking](#)) olarak adlandırılır.
- Tanım : Bir [uygun tipli işlenen](#), ya işletmenin tanımına uygundur veya dilin yapısı içinde örtülü olarak uygun tipe çevrilebilir.
- Tanım : Tip hatası: işlenen işletmene uygun değilse tip hatası oluşur.
- Örnek:

```
int i;  
float f;  
...  
f = 3.14 * i;
```

```
int *ip;  
float f;  
...  
f = 3.14 * ip;
```

Tip kontrolü

- Eğer tip bağlanmaları statikse, tip kontrolü de statiktir.
- Eğer tip bağlanmaları dinamikse, tip kontrolü de dinamik olmak zorundadır.
- Tanım: Bir programlama dili eğer tip hatalarının hepsini fark ediyorsa bu dile **kesin tiplendirilmiş** (*strongly typed*) dil denir.
- Bir dildeki bütün isimlerin önceden tanımlanmış olması kesin tiplendirilmiş olması için yeterli değildir. Çünkü programın yürütülmesi sırasında farklı veriler konulabilir.

Kesin tiplendirme

- Kesin tiplendirmenin avantajı: değişkenlerin yanlış kullanımının fark edilmesini ve hatalı sonuçları engeller.
- Bazı dillerin kesin tiplendirme durumu:
 - FORTRAN 77 kesin tiplendirilmiş değildir: parametreler, EQUIVALENCE
 - Pascal değildir.
 - C ve C++ değildir: parametre tip kontrolü engellenebilir, “union” tipler kontrol edilmez.
 - Ada hemen hemen (kontrol edilmeyen çevrimlerin istenebiliyor olması zayıflık) (Java, C# benzer şekilde)
- Çevirme kuralları kesin tip kontrolünü zayıflatır. (C++ karşı Ada)
- Java'nın çevirme kuralları C++'ın yarısı kadar olsa da, kesin tip kontrolü Ada'nın yanında zayıftır. Örneğin Java'da bir tam sayı değişkeni ile gerçek sayı değişkeni toplanırken tam sayı otomatik gerçele çevrilir ve bu yapılmırken kesin tipleme bozulmuş olur.

Tip Uyumluluğu

- İki tip ‘tip’ uyumluluğu bulunmaktadır:
 - İsim tipi uyumluluğu;
 - Yapısal tip uyumluluğu.
- Tanım: **İsim tipi uyumluluğu**: eğer iki değişken aynı tanımlamada tanımlanmış veya tanımlamalarında aynı tip tanımlama kullanılmışsa.
- Uygulanması kolay ancak hayatı sınırlandırıcı, Ada örneğini inceleyelim:
 - Sınırlı tam sayılar ile tam sayılar uyumlu değil:

```
type indextype is 1..100;
count: integer;
index: indextype;
```
 - Fonksiyona geçirilen yapısal parametrelerin isim tipi uyumluluğu olması gerekirse, bu tanımlama her fonksiyonda yapılamayacağından, bir kez global tanımlanması gereklidir (Pascal).

Tip Uyumluluğu

- Tanım: Yapısal tip uyumluluğu (*Structure type compatibility*): Eğer yapıları (*structure*) aynıysa, iki değişken uyumlu tiplerdir.
- Daha esnek fakat uygulaması zor.

Tip Uyumluluğu

- yapısal tiplerle ilgili aşağıdaki problemleri tartışalım:
 - Yapısal olarak aynı ama farklı alan adları kullanmış iki kayıt uyumlu mudur?
 - Diğer bütün özellikleri aynı ama indeksleri farklı iki dizilim aynı mıdır?
(örneğin [1..10] and [0..9])
 - Elemanları farklı yazılmış iki enumeration tip uyumlu mudur?
 - Tip uyumluluğu ile aynı yapıdaki farklı tipleri ayırd edemezsınız (Örneğin farklı birimlerde hız (mph – km/h), ikisi de float).
 - Ada bu problemin üstesinden aşağıdaki gibi gelir:

```
type mph is new float;
type kmph is new float;
```
- Uyumlu ada örnekleri:
Aşağıdaki örnek integer tipi ile uyumludur:

```
subtype small_int is integer range 1..99;
```

Aşağıdaki iki vector de uyumludur.

```
type vector is array (integer range <>) of integer;
vector 1: vector (1..10);
vector 2: vector (11..20);
```

Tip Uyumluluğu

■ Dil örnekleri:

- Pascal: genellikle yapısal tip uyumluluğu, fakat bazı durumlarda isim uyumluluğu da kullanılır (alt program parametrelerinde).
- C: yapısal, “structure”, “union” hariç.
- Ada: sınırlandırılmış isim uyumluluğu. Tanımlanmış yapılar uyumlu.
 - Türetilmiş tiplerde aynı yapıların farklı olması mümkün.
 - Genel tiplerin hepsi tek, hatta:
A, B : array (1..10) of INTEGER:
uyumlu değil.
 - Uyumlu olması için:
`type list10 is array (1 ..10) of Integer;
A, B : list10;`

Tip Uyumluluğu (notlar)

- C, C++, Java, vs, için toplama operatörünün iki işlecinin farklı nümerik tiplerde olması önemli değildir. Uygun şekilde örtülü olarak çevrilirler.
- Ada'da aritmetik operatörler için örtülü çevirme yoktur.
- C 'struct' ve 'union' hariç bütün tipler için yapılsal uyumluluk kullanır. Her 'struct' ve 'union' ayrı bir tiptir ve dolayısıyla uyumlu değildir. 'typedef' yeni bir tip yaratmaz, sadece isimlendirir.
- C++ isim tipi uyumluluğu kullanır.
- Yeni tip tanımlamaya izin vermeyen dillerde (Fortran, Cobol) isim tipi uyumluluğu kullanılamaz.

İsimler ve Kapsam

- Tanım: Bir değişkenin **kapsamı (scope)** değişkenin görülebilir olduğu komutların alanıdır. Görülebilir olduğu alan, bir komut içinde belirlenen değerle kullanılabilen alandır.
- Tanım: **Lokal değişkenler**, bir program biriminde kullanılan ve orada tanımlanmış değişkenlerdir.
- Tanım: **Lokal olmayan değişkenler**, bir program biriminde görülebilir olan ancak orada tanımlanmamış değişkenlerdir.
- Kapsam kuralları, isimlere yapılan referansların değişkenlerle nasıl bağlanacağının kurallarını belirler.

Statik Kapsam (static scope)

- Program metnine dayanır.
- Bir isim referansını değişkene bağlayabilmek için isim tanımlanmış olmalıdır.
- Arama işlemi: lokalden başlayarak ve her seferinde kapsamı genişleterek, verilen ismin tanımını arama. Bu durumda kapsam en içteki alt programdan onu çevreleyen üst alt programlara doğrudur. Bazı diller iç içe alt programları desteklerken (Ada, JavaScript, PHP), bazı diller desteklemez (C tabanlı diller gibi).
- İç içe alt programları desteklemeyen dillerde bile blok içleri ayrı kapsama alanlarıdır.
- Statik kapsamı çevreleyen kapsam onun atasıdır. En yakın ataya, ebeveyn (parent) denir.

Blok kapsamı

- Program birimleri içerisinde statik kapsam yaratma yöntemi - ALGOL 60 ile başlar
- Örnekler:

C and C++: **for** (...)

```
{  
    int index;  
    ...  
}
```

Ada: **declare LCL : FLOAT;**
 begin
 ...
 end

Kapsam

- Lokalde tanımlanmış aynı isimli değişken, dışarda ata tarafından tanımlı değişkene erişimi keser:

```
void sub() {  
    int count;  
  
    ...  
    while ( ... ) {  
        int count=1;  
        count ++;  
  
        ...  
    }  
}
```

- C++ ve Ada bu tip erişilmez verilere kapsamı belirterek erişim imkanı sağlar.
 - In Ada: `unit.name`
 - In C++: `class_name::name`

Statik kapsam, örnekler

- C++ değişken tanımlarının fonksiyon içinde herhangi bir yerde yapılmasına izin verir. Fonksiyonun içinde ama bir blok içinde olmayan tanımlar, fonksiyon içinde tanımlandığı noktadan fonksiyonun sonuna kadar tanımlanmış sayılırlar.
- C'de benzer tanımların fonksiyon başında yapılması zorunludur.
- C++, Java ve C# "class"ları içinde tanımlanan değişkenler farklılıklar gösterir:
 - Eğer herhangi bir metodun içinde tanımlanmadıysa, bütün class içinde tanımlıdır. "public"se dışardan da erişilebilir.
 - Bir metot içinde tanımlanıysa, tanımlandığı blokdaki değerini kullanır.
 - C#, C++ tipi göstergicileri destekler. Ancak bunlar güvenliği bozduklarından bunları kullanan 'metot'ların 'unsafe' olarak tanımlanması zorunludur.

1. A,B :ornek; X: Sub1;
2. A,B :ornek; Y: Sub 2; X: Sub3;
3. A,B, :ornek; X,Y: Sub2;
4. A,B, :ornek;

```

procedure ornek is
    A, B : Integer;
    ...
    procedure Sub1 is
        X: Integer;
        begin -- Sub1
        ...
        <===== 1
        end;
    procedure Sub2 is
        X,Y : Integer;
        ...
        procedure Sub3 is
            X : Integer;
            begin -- Sub3
            ...
            <===== 2
            end;
        begin -- Sub2
        ...
        <===== 3
        end;

    begin -- Example
    ...
    <===== 4
    endl;

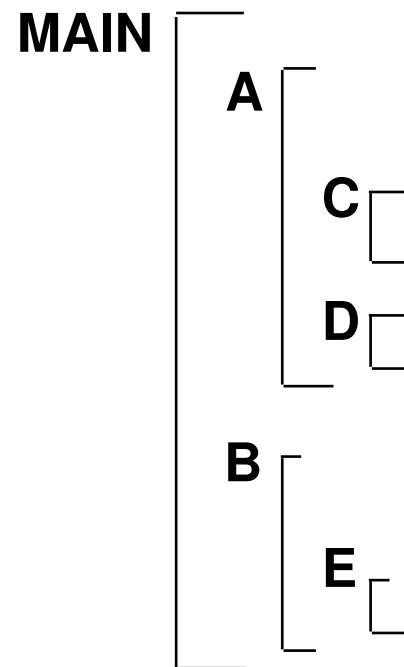
```

Kapsam

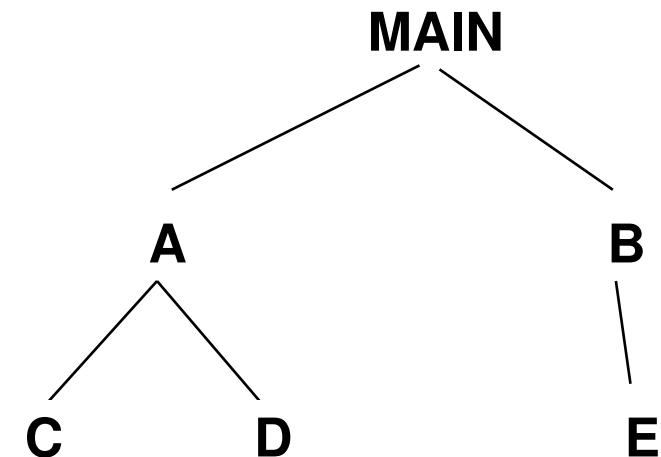
- Statik kapsamın değerlendirilmesi
- Örnek: Bütün kapsamlar MAIN program ve alt programlarca belirlenir.
 - MAIN A ve B yi çağırır
 - C ve D'yi A çağırır
 - A ve E'yi B çağırır

Statik kapsam örneği

Programın yapısı

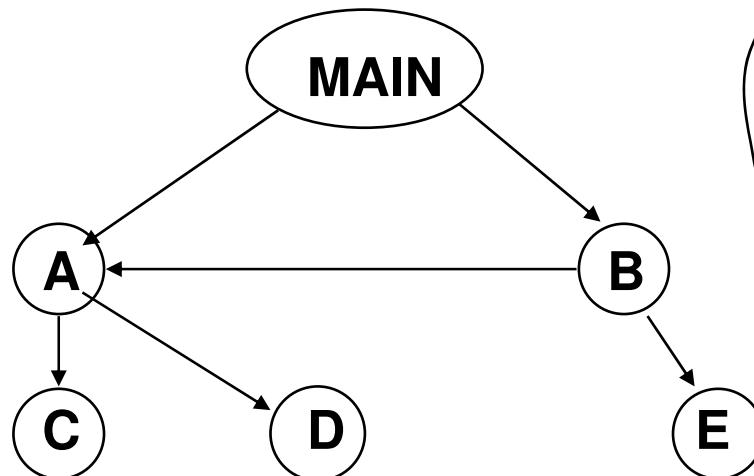


Programın ağaç yapısı

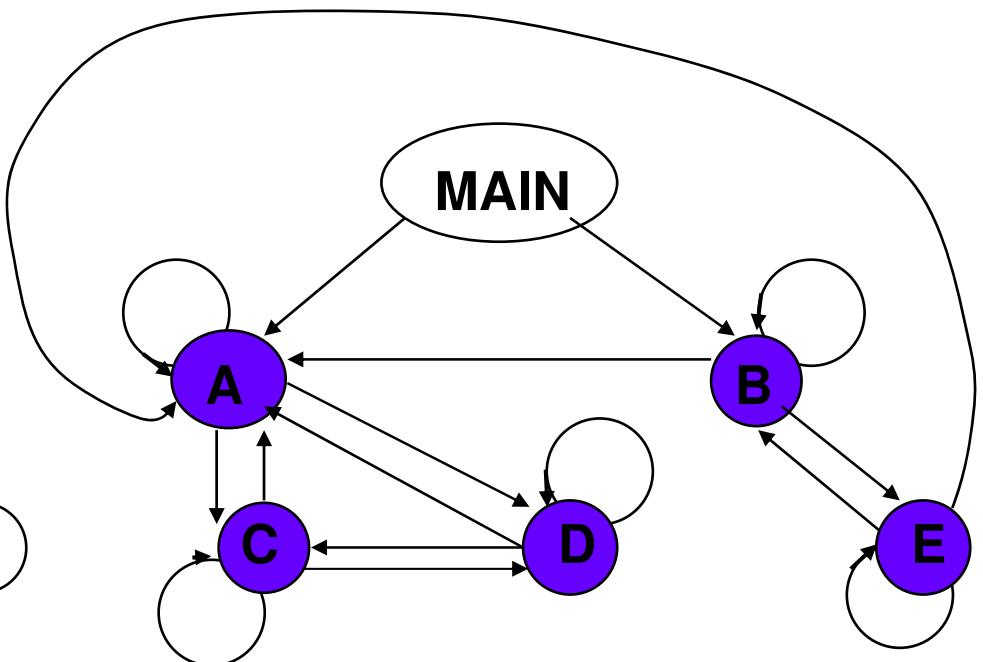


Statik kapsam örneği

Programın istenen altprogram
çağıırma yapısı

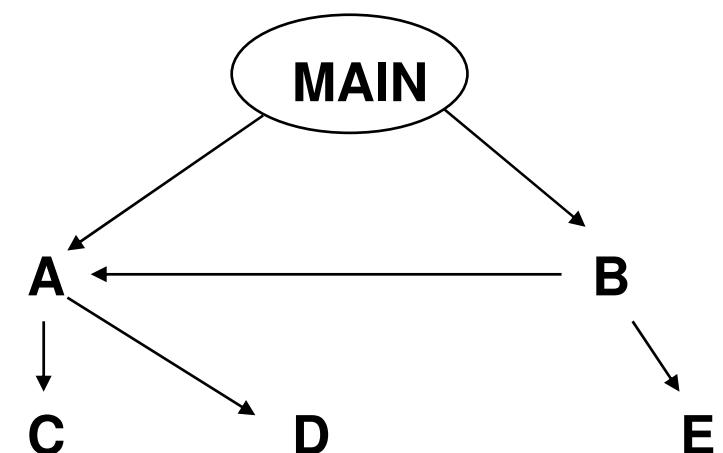
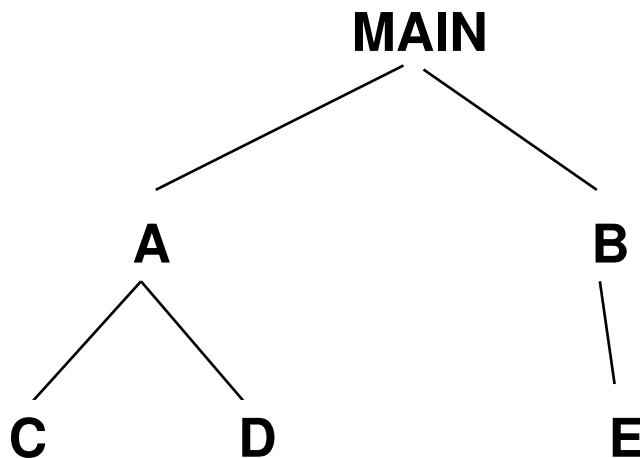


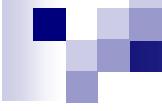
Programın potansiyel altprogram
çağıırma yapısı



Statik kapsam örneği

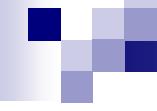
- Varsayıalım D, B'nin içindeki veriye ulaşmalı.
- Olası çözümler:
 - D'yi B'nin altına koy (Fakat artık C, D'yi çağrıramaz, D, A'nın değişkenlerine erişemez).
 - B'den D'nin gereksinim duyduğu veriyi MAIN'e koy (bütün alt programlar erişebilir)
- Sonuç: statik kapsam global değişkenlere neden olur.





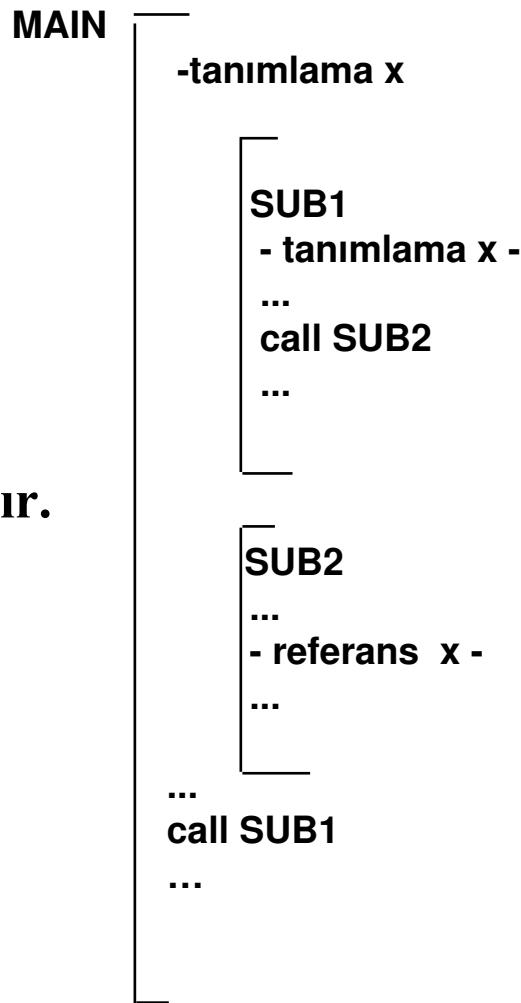
Dinamik kapsam

- Program birimlerinin çağrıma sırasında dayanır; onların programdaki yerleşme şekillerine değil.
- Değişkenlere erişim altprogramların herhangi bir andaki çağrı zincirine bağlıdır.
- APL, SNOBOL4, Perl ve bazı Lisp versiyonları dinamik kapsam kullanır. Perl ve Common Lisp her iki kapsamı da kullanabilirler.



Dinamik kapsam örneği

**MAIN SUB1'i
SUB1 SUB2'yi
çağırıır.
SUB2 x'i kullanır.**



- Statik kapsama
 - MAIN içindeki x'e referans
- Dynamic scoping
 - SUB1 içindeki x'e referans
- Dinamik kapsamanın değerlendirmesi:
 - avantaj: kolaylık;
 - dezavantaj: zor okunabilirlik.
- Perl ve Common Lisp'de dinamik kapsam vardır.

Dinamik kapsam değerlendirmesi

- Bir altprogramdaki içerisinde tanımlanmamış değişken programın sürecine göre farklı altprogramlardaki farklı tanımlara gönderme yapıyor olabilir.
- Altprogramlardaki değişkenleri başka altprogramların beklenmedik değiştirmelerinden korumak çok zor. Güvenilirlik çok düşüyor.
- Yerel olmayan değişkenlerin kullanım sırasında tip kontrolünü yapmak zor.
- Dinamik kapsamlı bir programı okumak pratikte çok zor. Her türlü dinamik kapsam öngörülemez.
- Yerel olmayan değişkenlere erişim çok fazla zaman aldığından, program yavaşlıyor.

Kapsam ve yaşam süresi

- Kapsam ve yaşam süresi bir birleriyle ilgili ancak farklı kavramlardır.
- Bir Java metodunun içinde tanımlanmış değişken, metot içinde geçerlidir, yaşam süresi de metot çalıştığı sürecedir.
- C ve C++’da altprogram içinde **static** değişkenleri düşünün. Kapsamı sadece altprogramdır, fakat ana program çalıştığı sürece korunur; yaşam süresi programın yaşam süresi kadardır.
- Aşağıdaki örneğe bakarsak, ‘sum’ değişkeninin kapsamı ‘compute’ fonksiyonu ile sınırlıysa da, ‘printhead’ çalışırken de yaşamaya devam eder.

```
void printhead() {  
    ...  
} /* end of printhead */  
void compute() {  
    int sum;  
    ...  
    printhead();  
} /* end of compute */
```

Referans Çevre

- Tanım:Bir komutun referans çevresi o komut tarafından erişilebilen bütün isimlerdir.
- Bir statik kapsamlı dilde bu, bütün lokal değişkenler ile çevreleyen kapsamlardan görünür değişkenlerdir.
- Bir alt programın yürütülmesi başlamış ve henüz sonuçlanmamışsa, o alt program aktif demektir.
- Bir dinamik kapsamlı dilde referans çevre,bütün lokal değişkenler ile aktif alt programlardaki bütün görünür değişkenlerdir.

Referans Çevre örneği (Ada iskelet programı) Statik Örnek

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
    ... <----- 1
    end; -- of Sub1
  procedure Sub2 is
    X : Integer;
    ...
  procedure Sub3 is
    X : Integer;
    begin -- of Sub3
    ... <----- 2
    end; -- of Sub3
  begin -- of Sub2
  ... <----- 3
  end; -- of Sub2
begin -- of Example
... <----- 4
end. -- of Example
```

Nokta	Referans Çevre
1	Sub1(X,Y), Example(A,B)
2	Sub3(X), Example(A,B)
3	Sub2(X), Example(A,B)
4	Example(A,B)

Referans Çevre örneği (C iskelet programı) Dinamik Örnek

```
void sub1() {  
    int a, b;  
    ... <----- 1  
} /* end of sub1 */  
void sub2() {  
    int b, c;  
    ... <----- 2  
    sub1;  
} /* end of sub2 */  
void main() {  
    int c, d;  
    ... <----- 3  
    sub2();  
} /* end of main */
```

Nokta	Referans Çevre
1	Sub1(a,b), Sub2(c), main(d)
2	Sub2(b,c), main(d)
3	main(c,d)

İsimli sabitler

- Tanım: Bir değişken sadece bir belleğe bağlıyken bir değere de sahipse buna **isimli sabit** (named constant) denir.
- Avantajı: okunabilirlik ve değiştirilebilirlik.
- Programları parametrize etmek için kullanılır.
- Değerlerin sabitlere bağlanması statik veya dinamik olabilir.
- Diller:
 - Pascal: sadece kalıp değerler
 - FORTRAN 90: sabit değerli ifadeler
 - Ada, C++, and Java: her türlü ifade.

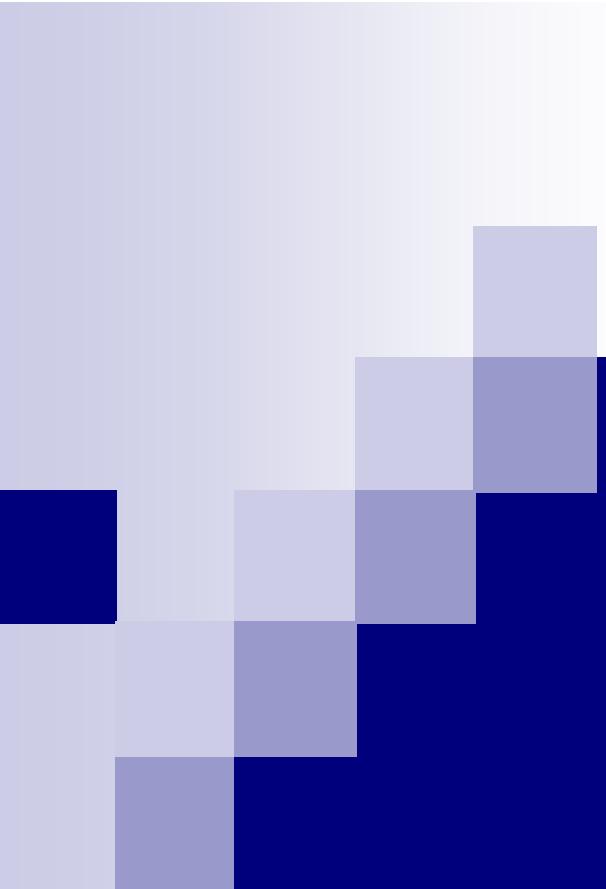
```
void example() {  
    int[] intList = new int[100];  
    String[] strList = new String[100];  
    ...  
    for (index = 0; index < 100; index++) {  
        ...  
    }  
    ...  
    for (index = 0; index < 100; index++) {  
        ...  
    }  
    ...  
    average = sum / 100;  
    ...  
}
```

```
void example() {  
    final int len = 100;  
    int[] intList = new int[len];  
    String[] strList = new String[len];  
    ...  
    for (index = 0; index < len; index++) {  
        ...  
    }  
    ...  
    for (index = 0; index < len; index++) {  
        ...  
    }  
    ...  
    average = sum / len;  
    ...  
}
```

Değişkenin başlangıç değeri

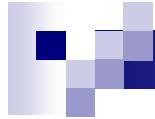
- Tanım: Bir değişkene bellekle bağlanırken verilen değere denir.
- Başlangıç değeri tanımlanırken verilir:
örneğin Java

```
int sum = 0;
```



Ders 6

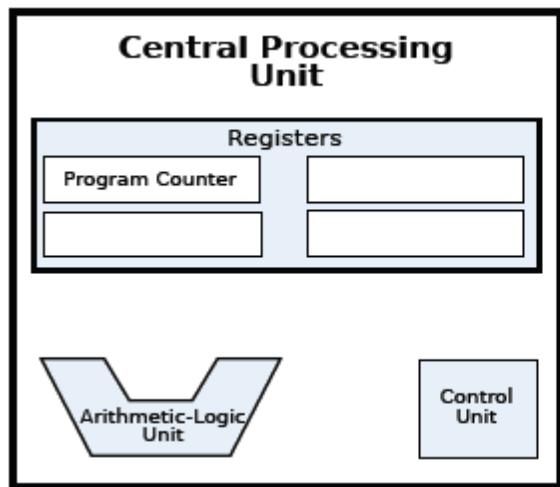
Veri Tipleri



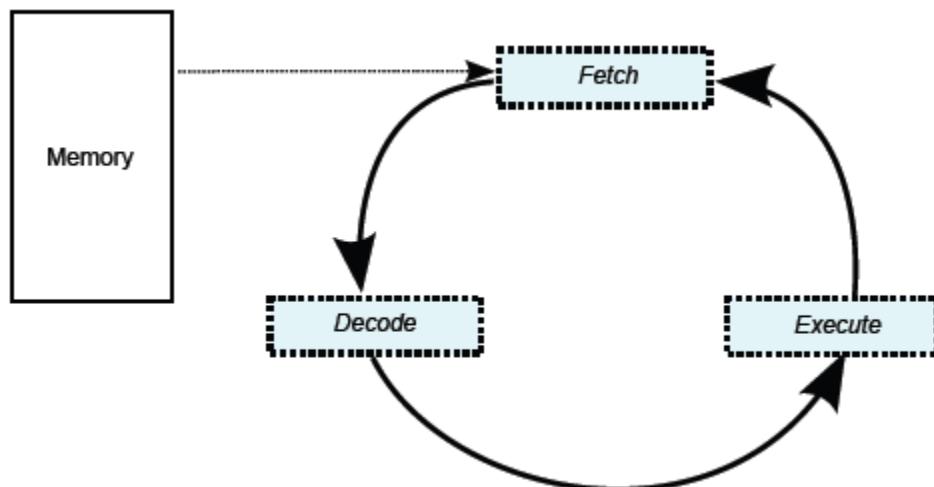
Konu Başlıkları

- Giriş
- Temel veri tipleri
- Karakter dizgi tipleri (character string)
- Kullanıcı tanımlı sıralı tipler (user defined ordinal types)
- Dizimler (arrays)
- İlişkili dizimler (associative arrays)
- Kayıt tipleri (Record Types)
- Bileşim tipi (Union Types)
- İşaretçi/gösterici tipleri (Pointer Types)

Giriş



Von Neumann mimarisi



Getir (Fetch), kodu çöz (decode), yap (execute)

main:

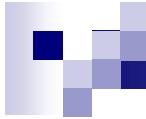
```
pushq %rbp  
movq %rsp, %rbp  
movl alice(%rip), %edx  
movl bob(%rip), %eax  
imull %edx, %eax  
movl %eax, carol(%rip)  
movl $0, %eax  
leave  
ret  
alice:  
.long 123  
bob:  
.long 456
```

Çevirici dili (assembly language)

İki ayrı bellek noktasındaki tam sayıları çarpıp sonucu başka bir bellek noktasına koyan program

```
01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011  
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000  
00001111 10101111 11000010 10001001 00000101 10110011 00000011 00100000  
00000000 10111000 00000000 00000000 00000000 11001001 11000011  
...  
11001000 00000001 00000000 00000000 00000000 00000000 00000000
```

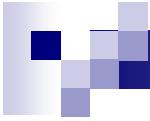
Yukarıdaki programın ikili sayı sisteminde bellekteki durumu.



Giriş

Von Neumann makinesinin yapısının programlama dilleri üzerindeki etkileri önemlidir.

- Von Neumann mimarisi CPU ile belleği belirgin bir şekilde ayırrır.
- Bellek içeriği oldukça karışiktır. Bellekte:
 - Her türlü CPU komutları: yazmaç (register) ↔ bellek transferleri, aritmetik işlemler, karşılaştırma, vs.
 - Bazı işlemleri yapabilmek için ek bilgi: transfer etmek için adresler, vs.
 - İşlenecek değerler, adres bilgileri gibi **veriler (data)**.
- Belleğe erişim tamamen adrese göredir.
- Von Neumann makinesinde işlenecek bütün bilgiler ikili sayısal sisteme dönüştürülmek zorundadır.



Bilgisayarda Veri (data) nedir?

Bir problemin çözümü esnasında bilgisayarda tutulan, CPU komutu olmayan her türlü bilgiye veri denir.

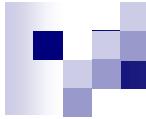
- Bazı tür veriler CPU tarafından doğrudan algılanır ve işlenir ancak bunlar çok fazla değildir: tam sayılar ve kayar noktalı sayılar. Ayrıca bu sayıların gerçekleştirmesi limitlidir (4 – 8 bayt gibi).
- Başka birçok veri tipi vardır ki CPU tarafından doğrudan işlenemez: kesirli sayılar, limitsiz tam ve kayar noktalı sayılar, kompleks sayılar, matrisler, karakterler, cebirsel veriler, dizgiler, gibi. Şüphesiz bu tip verilerin işlenmesi sağlayacak programlar yazılabilir.
- Bazı programlama dillerinde bu tip verileri işleyecek yapılar kurulmuştur. Örneğin Lisp, Prolog, Python ve ML limitsiz tam sayıları destekler; FORTRAN'da kompleks sayılar desteklenir; Mathematica, Matlab, Reduce, ve Maple cebirsel veriler işlenebilir; Hemen hemen bütün üst düzey dillerde karakter dizgileri işlenebilir.
- Şüphesiz daha az kullanılan bazı veri tipleri de bazı dillerde tanımlanabilir. Örneğin renklerin veri tipi olarak tanımlanması.



Bu derste uygulama dili olarak python kullanıyoruz. Bu dil 1980'lerin sonunda tasarlanıyordu, 1991'de ilk sürümü çıktı, 2000 yılındaki ikinci sürümünden sonra yaygınlaşmaya başladı.

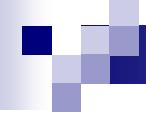
Python bm makinesinde çalıştırıldığında aşağıdaki satırlar görülür:

```
[tugrul@bm ~]$ python
Python 2.5.2 (r252:60911, Sep 30 2008, 15:41:38)
[GCC 4.3.2 20080917 (Red Hat 4.3.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> "hello world"
'hello world'
>>> 7*2
14
>>> 7/2
3
>>> 7.0/2
3.5
>>> 3**3
27
>>> 23**23
20880467999847912034355032910567L
>>> 23**60
5054406430037885272981046135356839275715416508617402090310185410509132307928805601L
```



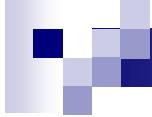
Giriş

- Bir veri tipi, bir değerler kümesini ve bu değerler üzerindeki işlemleri tanımlar. Bir programlama dilindeki veri tipleri, programlardaki ifade yeteneğini ve programların güvenilirliğini doğrudan etkiledikleri için, bir programlama dilinin değerlendirilmesinde önemli bir yer tutarlar.
- Temel ve yapısal veri tipleri arasındaki en önemli fark, temel veri tiplerinin başka veri tiplerini içermemesidir.
- Veri tiplerinin gelişimi:
 - FORTRAN I (1957) - INTEGER, REAL, arrays
 - Ada (1983) – Kullanıcı her kategori değişken için yeni tip tanımlar, sistemin tipleri zorlamasını, kontrol etmesini kullanır.
- Tanım: **Betimleyici/Niteleyici (descriptor)** bir değişkenin bütün özelliklerinin toplamıdır. Bir uygulamada bu, değişkenin özelliklerinin saklandığı belleklerdir.
 - Eğer özellikler statikse, bu bilgiler sadece derleme süresince saklanır.
 - Betimleyici derleyici tarafından, genellikle simbol tablosunun parçası olarak yaratılır ve kullanılır.
 - Dinamik özellikler programın yürütülmesi sırasında da saklanır ve kullanılır.
 - Betimleyici tip kontrolü gereken her yerde kullanılır.



Giriş

- Bütün veri tipleri için tasarımla ilgili önemli noktalar:
 1. Değişkenlere referanslarda sözdizim nedir?
 2. Hangi operasyonlar tanımlanmıştır, nasıl?



Temel veri tipleri

- Başka veri tipleri aracılığıyla tanımlanmayan veri tiplerine **Temel (*primitive*) veri tipleri** denir.
- Önceleri programlama dillerinde sadece sayısal temel veri tipleri tanımlanmışken, günümüzde popüler olan programlama dillerinde, karakter, mantıksal, karakter dizgi, kullanıcı tanımlı sıralı tipler gibi çeşitli temel veri tipleri bulunmaktadır.
- 1. Integer
 - Hemen her zaman CPU özelliklerine göredir ve eşlemleme kolaydır.
 - Birçok çeşidi olabilir:
 - Java: byte, short, int, long
 - C, C#, C++ (ek olarak) : unsigned int

Tam sayılar (integer)

İşaret ve büyüklük
sign and magnitude

İkili (binary)	signed	unsigned
00000000	0	0
00000001	1	1
...
01111111	127	127
10000000	-0	128
...
11111111	-127	255

IBM 7090 gibi ilk bilgisayarlarda
kullanılmıştır.

bire tümler
ones' complement

İkili (binary)	signed	unsigned
00000000	0	0
00000001	1	1
...
01111101	125	125
01111110	126	126
01111111	127	127
10000000	-127	128
10000001	-126	129
10000010	-125	130
...
11111110	-1	254
11111111	-0	255

ikiye tümler
two's complement

İkili (binary)	signed	unsigned
00000000	0	0
00000001	1	1
...
01111110	126	126
01111111	127	127
10000000	-128	128
10000001	-127	129
10000010	-126	130
...
11111110	-2	254
11111111	-1	255

binary decimal (bire tümler)

$$\begin{array}{r}
 11111110 \quad -1 \\
 + 00000010 \quad +2 \\
 \hline
 \dots \quad \dots \\
 1 \ 00000000 \quad 0 \quad \text{-- doğru değil} \\
 \quad \quad \quad 1 \quad +1 \quad \text{-- eldekini (carry) ekle} \\
 \hline
 \dots \quad \dots \\
 00000001 \quad 1 \quad \text{-- doğru cevap}
 \end{array}$$

ikiye tümler örneği

Örnek 1

Örnek 2

1. Sağdan başlayarak ilk '1' i bul

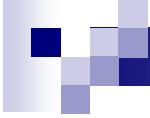
0101001

0101100

2. Onu takip edenleri tersine
çevir

1010111

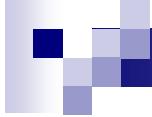
1010100



Temel veri tipleri

2. Kayan noktalı (Floating Point)

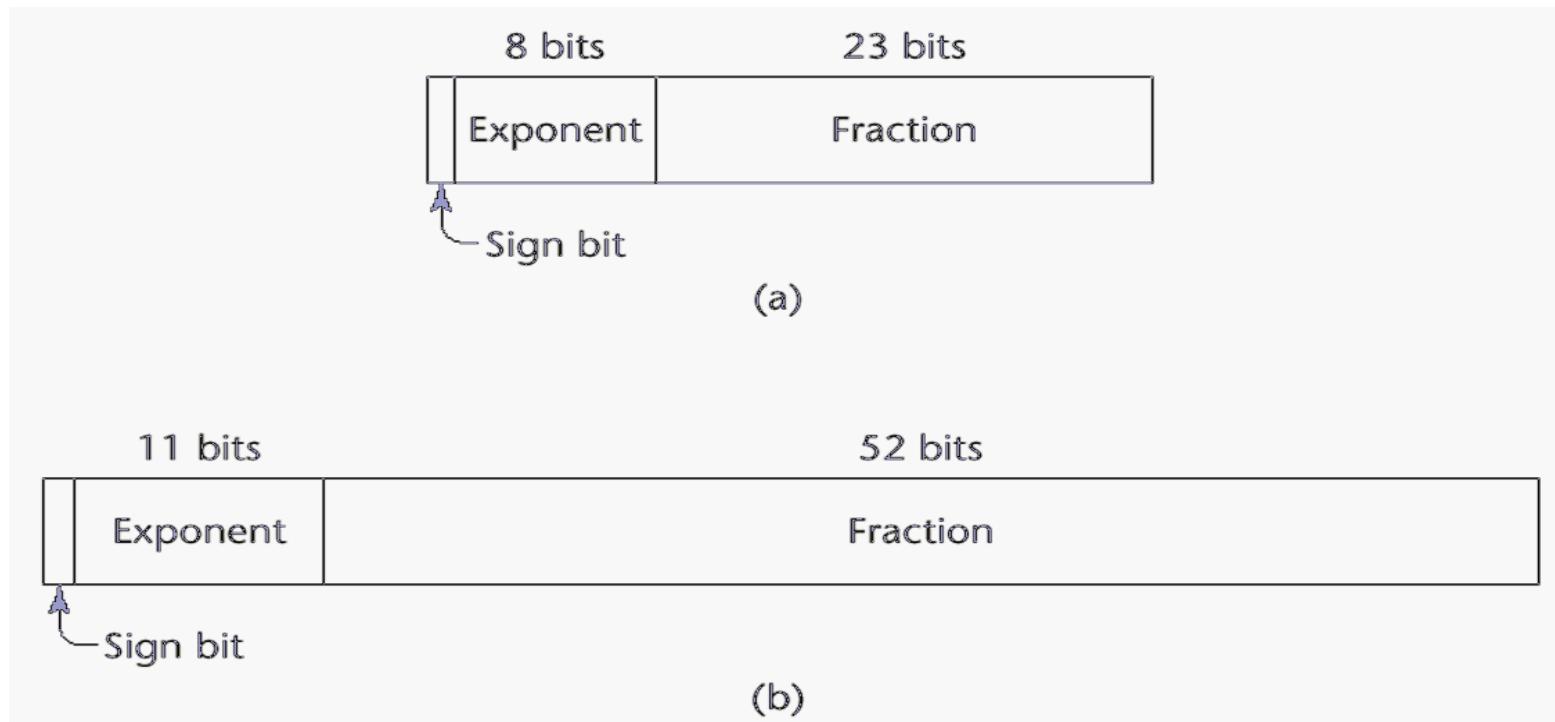
- **Kayan noktalı** (*floating point*) veri tipleri, gerçek sayıları **modellerler**.
- Modelleme yaklaşaktır. Örneğin π veya e sayıları gösterilemez.
- Diller genellikle iki tip kayar noktalı veri tipini desteklerler ancak bazen daha fazla da olabilir.
- Genellikle tamamen donanım (CPU) ile uyumludur.



IEEE Kayar noktalı formatı

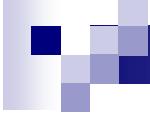
- Kayan noktalı sayılar, kesirler ve üsler olarak iki bölümde ifade edilirler.
- Kayan noktalı tipler, kesir (fraction) ve üst (exponent) açısından tanımlanırlar.
- Aşağıdaki şekilde görüldüğü gibi kayan noktalı veri tipi, gerçel (real) ve çift-duyarlılık (double-precision) olmak üzere iki tiple gösterilebilirler.

$$\left(1 + \sum_{n=1}^{23} \text{bit}_{[23-n]} \times 2^{-n} \right) \times 2^{\text{exponent}-127}$$



- Eğer $E=255$ ve $F \neq 0 \Rightarrow V=SD$ ("Sayı değil")
- Eğer $E=255$ ve $F = 0$ ve $S = 1 \Rightarrow V=-\infty$
- Eğer $E=255$ ve $F = 0$ ve $S = 0 \Rightarrow V=\infty$
- Eğer $0 < E < 255 \Rightarrow V=(-1)^S * 2^{E-127} * (1.F)$
- Eğer $E=0$ ve $F \neq 0, \Rightarrow V=(-1)^S * 2^{-126} * (0.F)$ Normalize edilmemiş değerler.
- Eğer $E=0$ ve $F = 0$ ve $S = 1, \Rightarrow V=-0$
- Eğer $E=0$ ve $F = 0$ ve $S = 0, \Rightarrow V=0$
- Diğer sayılar için: <http://babbage.cs.qc.edu/IEEE-754/Decimal.html>

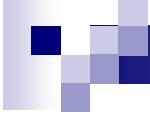
0 00000000 00000000000000000000000000000000 = 0	$\left(1 + \sum_{n=1}^{23} \text{bit}_{[23-n]} \times 2^{-n}\right) \times 2^{\text{exponent}-127}$
1 00000000 00000000000000000000000000000000 = -0	
0 11111111 00000000000000000000000000000000 = sonsuz	
1 11111111 00000000000000000000000000000000 = -sonsuz	
0 11111111 00001000000000000000000000000000 = SD	
1 11111111 00100010001001010101010 = SD	
0 10000000 00000000000000000000000000000000 = $+1 * 2^{-(128-127)} * 1.0 = 2$	
0 10000001 10100000000000000000000000000000 = $+1 * 2^{-(129-127)} * 1.101 = 6.5$	
1 10000001 10100000000000000000000000000000 = $-1 * 2^{-(129-127)} * 1.101 = -6.5$	
0 00000001 00000000000000000000000000000000 = $+1 * 2^{-(1-127)} * 1.0 = 2^{-(126)}$	
0 00000000 10000000000000000000000000000000 = $+1 * 2^{-(126)} * 0.1 = 2^{-(127)}$	
0 00000000 00000000000000000000000000000001 = $+1 * 2^{-(126)} *$ 0.00000000000000000000000000000001 = $2^{-(149)}$ (en küçük pozitif sayı)	



Temel veri tipleri

3. Onlu (Decimal, BCD)

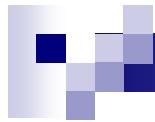
- İş uygulamaları için (para)
- Bu veri tipi, az sayıda programlama dilinde (Örneğin; PL/I, Cobol, C# gibi) tanımlanmıştır.
- Sabit sayıda on tabanlı karakterleri saklar (kodlanmış – BCD binary coded decimal)
- **Avantaj:** kesinlik
- **dezavantaj:** kapsama sınırlıdır, çok bellek kullanılır. Onlu veri tipi, onlu değerleri tam olarak saklayabilirse de, üsler bulunmadığı için gösterilecek değer alanı sınırlıdır. Her basamak için bir sekizli (byte) gereklili olması nedeniyle, belleği etkin olarak kullanmaz.
- İşlemler CPU desteği varsa CPU tarafından yapılır (Intel de yok), yoksa yazılımla benzetimlenir (simulate).



Temel veri tipleri

4. Boolean

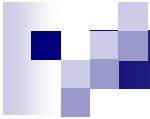
- Bir mantıksal değer bellekte bir ikili ile gösterilebilirse de, çoğu bilgisayarda bellekteki tek bir ikiliye etkin olarak erişim güç olduğu için, bir sekizlide (byte) saklanırlar.
- **Mantıksal(boolean)** veri tipi, ilk olarak ALGOL 60 tarafından tanıtılmış ve daha sonra çoğu programlama dilinde yer almıştır. Mantıksal veri tipi, sadece *doğru (true)* veya *yanlış (false)* şeklinde ifade edilen iki değer alabilir.
- İlişkisel işlemciler, mantıksal tipte bir değer döndürdükleri için ve seçimli deyimler gibi programlamadaki birçok yapı, mantıksal tipte bir ifade üzerinde çalıştığı için mantıksal veri tipinin dilde yer almasının önemi büyektür.
- ALGOL 60'dan sonraki çoğu dilde yer alan mantıksal veri tipinin yer almadiği bir programlama dili C dilidir. C'de ilişkisel işlemciler, ifadenin sonucu doğru ise 1, değilse 0 değeri döndürürler. C'de *if* deyimi, sıfır değeri için yanlış bölümünü, diğer durumlarda ise doğru bölümünü işler.
- **avantaj:** okunabilirlik.



Temel veri tipleri

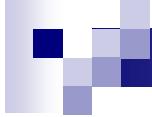
5. Karakter

- **Karakter**(character) veri tipi, tek bir karakterlik bilgi saklayabilen ve bilgisayarlarda sayısal kodlamalar olarak saklanan bir veri tipidir.
- Karakter veri tipinde en yaygın olarak kullanılan kodlamalardan biri ASCII (American Standard Code for Information Interchange) kodlamasıdır. ASCII kodlaması, 128 farklı karakteri göstermek için, 0..127 arasındaki tamsayı değerleri kullanır.
- ASCII kodlamasıyla bağlantılı olarak bazı programlama dilleri, karakter veri tipindeki değerlerle tamsayı tipi arasında ilişki kurarlar. C'de, *char* veri tipi, int ile dönüşümlü olarak kullanılabilmektedir.
- ISO 8859-1 başka bir karakter kodudur ve 256 karakterden oluşur.
- Daha çok dilin karakter setini göstermek amacıyla daha sonra Unicode (UTF) geliştirilmiştir. Bu kodlamaya bütün diller eklenmiştir. Burada karakterler 1-4 bayt ile gösterilirler. ASCII kodu bu gösterimde tek bayt olarak dahil edilmiştir. Java, JavaScript ve C# bu karakter kodlarını kullanabilmektedir.



Karakter dizgi tipi (Character String Types)

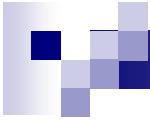
- Bir **karakter dizgi** veri tipinde, nesneler karakterler dizisi olarak bulunur.
- Karakter dizgi veri tipi bazı programlama dillerinde temel bir veri tipi olarak (dolayısıyla dizilim tipi indeksli kullanım yok), bazlarında ise özel bir karakter dizilimi olarak yer almıştır.
- FORTRAN77, FORTRAN90 ve BASIC'te karakter dizgiler temel bir veri tipidir ve karakter dizgilerin atanması, karşılaştırılması vb. işlemler için işlemciler sağlanmıştır.
- Pascal, C, C++ ve Ada'da ise karakter dizgi veri tipi, tek karakterlerden oluşan dizilimler şeklinde saklanır.
- Önemli tasarım özelliklerinden birisi de boyunun statik veya dinamik olmasıdır.



Karakter dizgi tipi

■ İşlemler:

- Atama, tanımlama (char *str = “özellikler”;)
- Karşılaştırma (=, >, strcmp, vs.)
- Birleştirme
- Alt dizgiye erişim
- Örüntülü eşleme (Pattern matching)



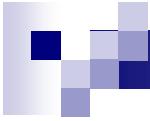
Karakter dizgi tipi

- Örnekler:
 - Pascal
 - temel veri tipi değil; sadece atama ve karşılaştırma (dizilimde duran verinin)
 - Ada, FORTRAN 90, and BASIC
 - Temel veri.
 - Atama, karşılaştırma, birleştirme, alt dizgiye erişim
 - FORTRAN da örüntülü eşleme var.

örneğin (Ada)

N := N1 & N2 (birleştirme)

N(2..4) (alt dizgiye erişim)



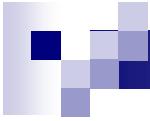
Karakter dizgi tipi

■ C and C++

- temel değil
- `char` dizilimleri ve kütüphane fonksiyonları kullanılır.
- örnek: `strcpy (src, dst);`
- C++'da string sınıfı kullanmak daha iyi; kontrol var.

■ SNOBOL4 (bir dizgi işleme dili)

- temel
- Ayrıntılı örüntülü eşleme dahil birçok operatör.



Karakter dizgi tipi

■ Perl, JavaScript, C++, Java, C#, vs

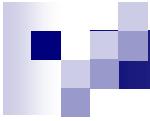
- Örüntüler (Patterns) düzenli ifadeler (regular expressions) ile tanımlanır.
- Çok güçlü bir özellik
- Örneğin :

/ [A-Za-z] [A-Za-z\d]+ /

veya / \d+ \. ? \d* | \. \d+ /

■ Java

- **String** class (karakter dizilimleri değil (not arrays of **char**)) statik dizgi nesneleri yaratır. Nesneler değiştirilemez (kesin).
- **Buna karşılık StringBuffer** sınıfı değiştirilebilir dizgi nesnelerinin sınıfıdır.



Karakter dizgi tipi

- Dizgi boyu seçenekleri (String Length Options):

1. **Statik** - FORTRAN 77, Ada, COBOL

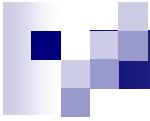
Örneğin. (FORTRAN 90)

```
CHARACTER (LEN = 15) NAME ;
```

2. **Limitli Dinamik Uzunluk** - C ve C++ da geçek boy sondaki null karakterinden anlaşılır.

3. **Dinamik** - SNOBOL4, Perl, JavaScript

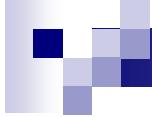
4. Ada her üç tip dizgiyi de ayrı ayrı destekler.



Karakter dizgi tipi

■ Değerlendirme

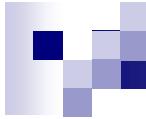
- yazabilmek için araç.
- Sabit boylu temel tip olarak desteklenmesi kolay, neden olmasın.
- Dinamik boy güzel ancak harcanan kaynaklara değer mi? Dinamik dizgiler genelde yorumlanan dillerde tercih ediliyorlar.



Karakter dizgi tipi

■ Gerçekleştirim (implementation):

- Statik boy – derleme zamanı niteleyici/betimleyici.
- Sınırlı dinamik boy – yürütme zamanı betimleyici gerekli olabilir (fakat C ve C++ da yok.)
- Dinamik boy – Yürütme zamanı betimleyici gereklidir. Bellekten yer alma ve geri verme en zor uygulama problemi.



Karakter dizgi tipi

Statik dizgi

boyu

adresi

Sınırlı dinamik dizgi

Maksimum boyu

Şimdiki boyu

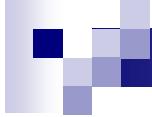
Adresi

Statik dizgiler için
derleme zamanı
betimleyici

Sınırlı dinamik
dizgiler için
yürütme zamanı
betimleyici

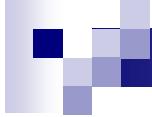
Kullanıcı Tanımlı Sıralı Tipler (User-Defined Ordinal Types)

- Bir **sıralı (ordinal) tip**, olası değerlerin pozitif tamsayılar kümesi ile ilişkilendirilebildiği veri tipidir.
- Bir çok programlama dilinde kullanıcılar, **sayılıma (enumeration)** ve **altalan (subrange)** olmak üzere iki tür sıralı tip tanımlayabilir. Bu tip tanımlarındaki amaç, programcılara modellenen gerçek dünya nesnelerine karşı gelebilecek yeni tipler oluşturma olanağı sağlamaktır.
- Örnekler:
 - C++: enum renkler {kirmizi, mavi, yesil, sari, siyah};
renkler benimRenk = mavi;
benimRenk++;
benimRenk = 4; (C++ kural dışı; C kabul)
benimRenk = (renkler) 4; (kabul)
 - C: typedef enum {kirmizi, mavi, yesil, sari, siyah} renkler;



Kullanıcı Tanımlı Sıralı Tipler

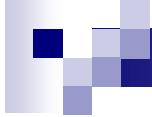
1. **Sayılama (*enumeration*) tipi**, gerçek hayataki verilerin tamsayı (*integer*) veri tipine eşleştirilmesi için kullanılan veri tipidir.
- Tasarım sorunu:
 - Sembolik sabitlerin birden çok tip tanımında kullanılmasına izin verilmeli mi?
 - Sayılama değerleri tam sayı olmaya zorlanmalı mı?
 - Diğer tipler sayılama tipine zorlanmalı mı?
 - Bu sorular tip kontrolü ile ilişkilidir. Eğer sayılama tipi tam sayılarla zorlanırsa, alabilecekleri değerleri kontrol etmek için fazla bir şey yapılamaz.



Kullanıcı Tanımlı Sıralı Tipler

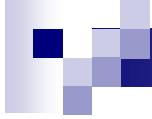
■ Örnekler:

- Pascal – sabitleri tekrar kullanılamaz; Dizilim indeksi, değişken, “case” seçicisi olarak kullanılabilirler. Karşılaştırılabilirler. girdi, çıktı değeri olarak kullanılamaz.
- Ada – sabitler tekrar kullanılabilir (overloaded literals); kullanıldıkları bağlamda veya tip adından farkları anlaşılır (birinden biri). Pascal'daki gibi kullanılabilir, girdi çıktı değeri olabilirler. Tam sayılarla zorlanmazlar.
- C and C++ - Pascal gibi; ayrıca tam sayılar olarak girdi ve çıktı değişkeni olabilirler.
- C#: C++ gibi. Ancak tam sayılarla zorlanmazlar.
- Java sıralı tipi desteklemez, fakat **Enumeration** arayüzüne sağırlar.



Kullanıcı Tanımlı Sıralı Tipler

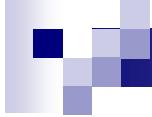
- Değerlendirme (Sıralı Tiplerin):
 - a. Okunabilirliği arttırır – örneğin renkleri sayı olarak girmeye gerek yok.
 - b. Güvenilirliği artırır – Örneğin derleyici aşağıdakileri kontrol edebilir:
 - i. işlemler (renklerin toplanmasına izin vermez)
 - ii. alabileceği değerler (eğer 7 renge izin verdiyseniz, 9 geçerli bir sayı ve renk değildir.)



Kullanıcı Tanımlı Sıralı Tipler

2. altalan (*subrange*) tipi

- Bir **altalan tipi**, bir sıralı tipin bir alt grubudur.
- Bir altalan tipinin tanımındaki değerler, daha önceden tanımlanmış veya dilde tanımlı olan (*built-in*) sıralı tiplerle ilişkilendirir. Böylece, yeni tanımlanan tip ile alt grubu olduğu ana sınıf arasında bağ kurulur. Altalan tipinin ana sınıfına uygulanabilen tüm işlemciler, altalan tiplerine de uygulanabilmektedir.
- Tasarım sorusu: nasıl kullanılabılırler?

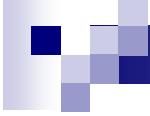


Kullanıcı Tanımlı Sıralı Tipler

■ Örnekler:

- Pascal – Altalan tipleri ebeveynlerine benzer ve onlar gibi davranışları; “for” döngüsünde kullanılabilir, dizilik indeksi olabilirler.

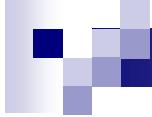
```
type pos = 0 .. MAXINT;
```



Kullanıcı Tanımlı Sıralı Tipler

- Ada – Altalan tipleri yeni tipler değildir. Sınırlandırılmış varolan tiplerdir. Pascal'daki gibi kullanılabildiği gibi “case” içinde de kullanılabilir.

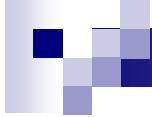
```
subtype POS_TYPE is
    INTEGER range 0 .. INTEGER'LAST;
```



Kullanıcı Tanımlı Sıralı Tipler

■ Kullanıcı Tanımlı Sıralı Tiplerin uygulanması:

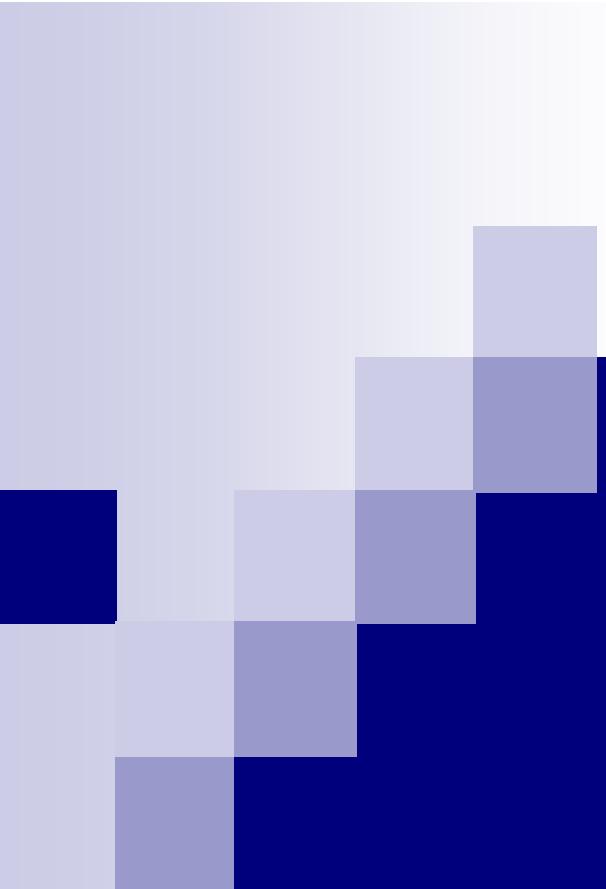
- Kullanıcı Tanımlı Sıralı Tipler tam sayılarla uygulanır.
- Altalan tipleriyse derleyici tarafından sınırlar eklenmiş ebeveyn tiplerdir.



Kullanıcı Tanımlı Sıralı Tipler

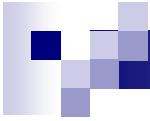
- Altalan tiplerinin değerlendirmesi:

- Okunabilirliği arttırır.
 - Güvenilirlik – sınırlandırılmış değerler hata kontrolünü sağlar.



Ders 6

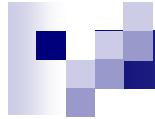
Bölüm 6: Veri Tipleri (devam)



Dizilimler (Arrays)

- Dizilik bir toplam homojen veri alanıdır. İçindeki her bir elemana ilk elemana göre olan pozisyonuna göre erişilir.
- Örnek: C:

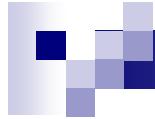
```
int aa[4][3][7];  
sum += aa[i][j][k];
```



Dizilimler

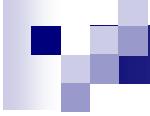
Tasarım konuları:

1. Hangi tip altsimgeler/indeksler (subscript) kullanılabilir?
2. İfade içinde altsimgen değerlerinin geçerlilikleri kontrol edilecek mi?
3. Ne zaman altsimgen geçerli değerleri sınırlandırılır?
4. Yer ayrılması (bellekte) ne zaman yapılır?
5. Altsimgelerin maksimum sayısı kaçtır?
6. Dizilimlere başlangıç değerleri atanabilir mi?
7. Dilimleme yapılabilmeli mi?



Dizimler

- İndeksleme, altsimgeden elemanlara bir eşlemlemedir.
eşlemleme(dizim adı, altsimge (indeks) değeri) → bir eleman
- İndeks söz dizim (syntax)
 - Bütün dillerde genel bir kabul görmüş yapı vardır. Önce dizimin adı sonra parantezler veya köşeli parantezler gelir, bunların arasında da altsimge bulunur.
 - FORTRAN, PL/I, Ada parantez kullanır.
 - Diğer dillerin çoğu köşeli parantez kullanır.



Dizimler

- Altsimgeler:
 - FORTRAN, C, Java – sadece “integer”
 - Pascal – Her türlü sıralı tip (integer, boolean, char, enum)
 - Ada - integer veya enum (boolean ve char dahil)
- Altsimgeler C tabanlı dillerde 0'dan başlar. Fortran'da 1 dir. Bazı dillerde ise tamamen programcı tarafından belirlenir.

Dizilimler

■ Dizilimlerin kategorileri (Altsimge bağlanmaları ve bellekteki bağlanmalarına göre)

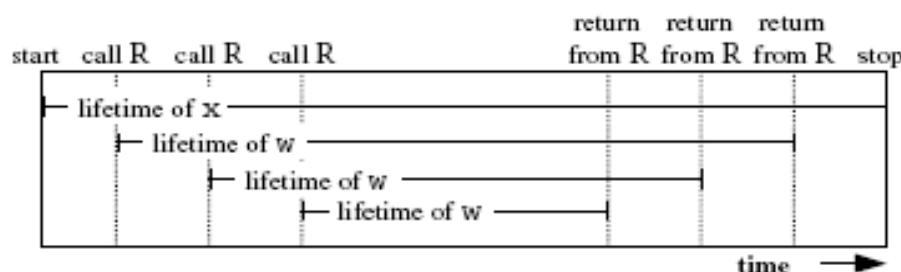
1. **Statik** - Altsimge limitlerinin bağlanmaları ve dizilimin bellekteki bağlanmaları statik.

FORTRAN 77, Ada'da bazı dizilimler. C'de statik dizilimler.

- Avantaj: Yürütme verimi (bellekten yer alma, geri verme yok)
- 2. **Sabit yiğit dinamik (Fixed stack dynamic)** - Altsimge limitlerinin bağlanmaları statik fakat dizilimin bellekteki bağlanması yürütme sırasında dizilim gerçekleştirilirken.

Örneğin "statik" olmayan çoğu Java, C lokal değişkenleri.

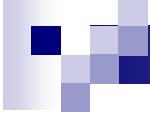
- avantaj: bellek verimi



C++ (veya C) programı:

```
double z[8];
void main () {
    int x[23];
    static float y[10];
    ... R(); ...
}

void R () {
    int w[5];
    ... R(); ...
}
```



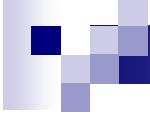
Dizilimler

3. **Yığıt dinamik (Stack-dynamic)** - Altsimge bağlanmaları ve bellekteki bağlanmalarına dinamik, fakat değişkenlerin hayatı boyunca sabit.

□ Örnek: Ada declare blokları:

```
declare
  STUFF : array (1..N) of FLOAT;
begin
  ...
end;
```

□ Avantaj: Esneklik – Dizilim kullanılıncaya kadar boyutlarını bilmeye gerek duyulmaz.



Dizilimler

4. **Sabit Yığın dinamik (Fixed Heap-dynamic)** - Altsimge bağlanmaları ve bellekteki bağlanmaları dinamik, fakat bir kez belirlendikten sonra sabit. Bellek geri verilebilir.

□ **FORTRAN 90**

INTEGER, ALLOCATABLE, ARRAY (:,:) :: MAT

(MAT’ı dinamik 2-boyutlu dizilim olarak tanımlar)

ALLOCATE (MAT (10,NUMBER_OF_COLS))

(MAT’a 10 satır ve NUMBER_OF_COLS sütun ayırır)

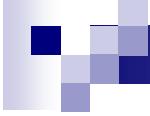
DEALLOCATE MAT

(MAT için ayrılmış belleği iade eder).

□ C, C++: malloc, free ...

□ C++: new, delete.

□ Java’da bütün dizilimler sabit yığın dinamikdir. C# da sabit yığın dinamiği destekler.



Dizimler

4. Yığın dinamik (Heap-dynamic) - Altsimge bağlanmaları ve bellekteki bağlanmalarına dinamik, ve sabit değil.

□ APL, Perl ve JavaScript'de dizimler isteğe göre büyüyebilir veya küçülebilir.

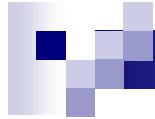
Perl örneği: @list = (1 , 3, 7, 10);

push(@list, 13 , 17); // → (1 , 3, 7, 10 13 , 17)

@list = (); // belleği boşaltır ve iade eder.

□ C# ArrayList class, yığın dinamik dizimleri sağlar. Bu class'ın nesneleri elemansız yaratılır, sonra dinamik olarak elemanlar eklenir:

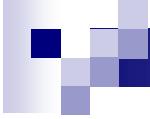
```
ArrayList intList = new ArrayList();  
intList.Add(nextone);
```



Dizimler

■ Altsimgelerin sayısı

- FORTRAN I: 3'e kadar.
- FORTRAN 77: 7'ye kadar.
- Diğerleri – altsimge limitlerinde.



Dizilimler

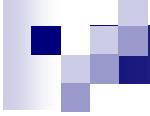
- Dizilim başlangıcı:
 - Genellikle sırayla yazılmış başlangıç değerleri listeleri şeklinde.
- Dizilim başlangıç örnekleri:
 1. FORTRAN - DATA deyimini kullanır veya değerleri tanımlama sırasında / . . . / içine koyar.

Integer, Dimension (3) :: List = (/0,5,5/)

veya

```
Integer List(3)
```

```
DATA List /0,5,5/
```



Dizimler

■ Dizim başlangıç örnekleri :

2. C ve C++ - Kırık parantezler içinde, derleyici elemanları sayar ve buna göre yer ayarlar:

```
int stuff [] = {2, 4, 6, 8};  
char name[] = "tuğrul";  
char *names[] = {"ali", "veli"}
```

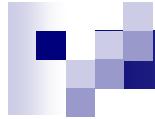
3. Java

```
String[] names = {"ali", "veli"};
```

4. Ada – değerler için yer belirlenebilir:

```
SCORE : array (1..14, 1..2) :=  
        (1 => (24, 10), 2 => (10, 7),  
         3 => (12, 30), others => (0, 0));  
List: array(1..5) of Integer := (1, 3, 5, 77)
```

5. Pascal başlangıç değerine izin vermez.



Dizilimler

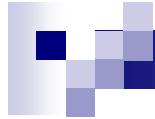
■ Dizilim işlemleri

1. Ada

- **Atama**; Sağ tarafta birden çok değer veya dizilim olabilir.
- **Birleştirme**; bütün tek boyutlu dizilimler için.
- **İlişkisel operatörler** (sadece = ve /=)

2. FORTRAN 90

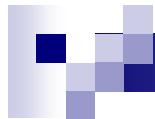
- Derleyici kütüphanelerinden altprogramlarla birçok dizilim işlemi; örneğin matris toplamı, çarpımı, vektör iç çarpımı gibi.



Dizilimler

■ Dilimler (Slices)

- Dilimler, dizilimlerin alt parçalarıdır. Belli bölgelere erişmek için kullanılabilecek yöntemlerden biridir.
- Dizilim işlemleri olan dillerde faydalıdır.



Dizimler

■ Dilim örnekleri:

1. FORTRAN 90

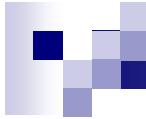
```
INTEGER MAT (1:3, 1:3)
```

MAT (1:3, 1) – birinci sütun

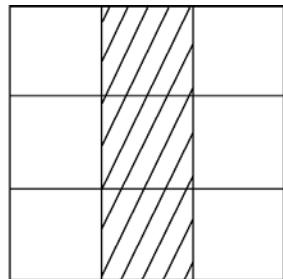
MAT (2, 1:3) – ikinci satır

2. FORTRAN 95

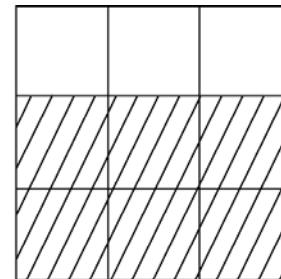
```
Integer, Dimension (3, 3, 4) :: Cube
```



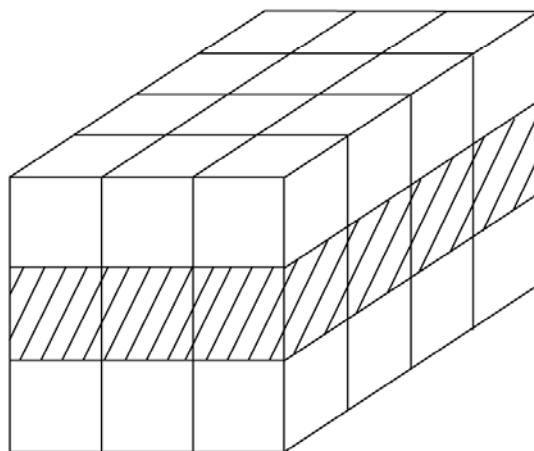
FORTRAN örnekleri



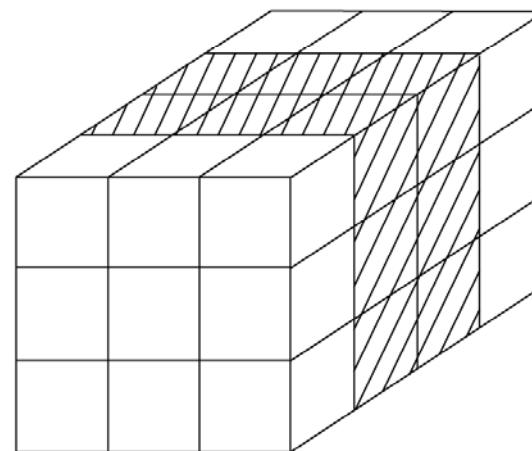
MAT (1:3, 2)



MAT (2:3, 1:3)

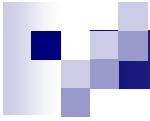


CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

MAT = CUBE (:,:, :, 2) // cube dizininin ikinci dilimini mat'a koyar.

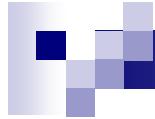


Dizimler

■ Dilim örnekleri:

3. Ada – sadece tek boyutlu dizimler

LIST (4 .. 10)



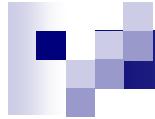
Dizilimler

■ Dizilim işlemleri:

Bazı diller dizilimler üzerinde tanımlı işaretleri tanımlamıştır.

Örneğin Fortran 95'de A ve B adlarında iki aynı boyutlu dizilimin toplamı ($A + B$), elemanları iki dizilimin aynı yerindeki elemanlarının toplamı olan başka bir dizilimdir.

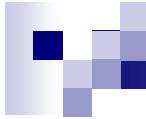
Dizilim işlemlerinde en yetenekli dil APL'dir. 4 farklı işlem tanımlanmıştır.



Dizimler

■ Dizimlere erişimin gerçekleştirilmesi

- Erişim fonksiyonu altsimgeleri hesaplayarak dizim içindeki hücrenin bellekte durduğu adrese ulaşır.
- Bunu yaparken
 - sütün öncelikli (Fortran) veya
 - satır öncelikli (Diğer diller) yöntemini kullanır.



Bir elemanın yerini bulmak

Eğer altsimgeler 1'den başlıyorsa ve satır öncelikli yerleştirme varsa:

Adres(eleman(i,j)) = Adres(eleman(1,1)) +
 $((i - 1) * n + (j - 1)) * \text{eleman_boyu}$

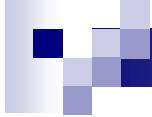
Not 1: Satır önceliklide döngüleri sütun üzerinden, sütun önceliklide satır üzerinden yapmak bellek erişimini hızlandırır.

Not 2: Çok boyutlu dizilimlerde işaretçi kullanımına dikkat.

Örnek:

```
int * ip;  
int i;  
int aa[100][150];  
ip = &aa;  
for(i = 0; i < 100*150; i++)  
    *ip++ = i;
```

	1	2	...	$j-1$	j	...	n
1							
2							
:							
$i-1$							
i					⊗		
:							
m							



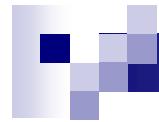
Derleme zamanı betimleyicisi

Dizilim
Eleman tipi
Altsimge tipi
Altsimge alt limit
Altsimge üst limit
adres

Tek boyutlu dizilim

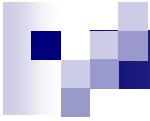
Cök boyutlu dizilim	
Eleman tipi	
Altsimge tipi	
Boyutu = n	
Altsimge 1 alt limit	Altsimge 1 üst limit
...	...
Altsimge n alt limit	Altsimge n üst limit
Adres	

Cök boyutlu dizilim



İlişkili Dizimler (associative array)

- Bir ilişkili dizim kendisiyle aynı sayıdaki anahtar denilen değerlerle indekslenmiş rasgele verilerdir.
- Anahtar değerlerinden bir kiyim (hash) algoritması ile değerler bulunur.
- Tasarım sorunları:
 1. Dizim elemanlarına olan referans şekli nedir?
 2. Boyut statik mi yoksa dinamik mi?



İlişkili Dizimler

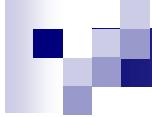
- Java ve C++ “class” kütüphaneleri ilişkili dizimleri destekler.
- Perl’de kullanımı ve yapısı
 - Değişkenler “%” ile başlar. Boyu değişkendir.
 - Sabit değerler parantezler arasına yazılır:

```
%YuksekSicakliklar = ("Pazartesi" => 45,
                           "Salı" => 49, ...);
```
 - Altsimgeme parantezler ve anahtar verilerle yapılır:

```
$YuksekSicakliklar{"Çarşamba"} = 42;
```
 - Elemanlar **delete** ile silinebilir:

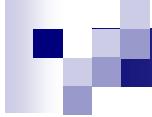
```
delete $YuksekSicakliklar{"Salı"};
```
 - Dizimi tamamen boşaltmak için:

```
@YuksekSicakliklar = ();
```
- PHP dizimleri hem normal hem ilişkili dizimlerdir.
- Eğer dizimde arama yapılacaksa ilişkili dizimler çok verimlidir. Sıralı erişim için normal dizim kullanılmalıdır.



Tutanaklar (Records)

- Tutanak olasılıkla heterojen veri yapılarından oluşmuş, içindeki unsurların isimleriyle ayırtırıldığı bir yapıdır.
- Tasarım sorunları:
 1. Referansların şekli nasıl olacak?
 2. Hangi birim işlemleri tanımlanacak?



Tutanaklar

■ Tutanak tanımlama sözdizimi

□ COBOL seviye numaraları kullanır, diğerleri özyinelemeli olarak tanımlarlar:

01 ISCI-KAYDI.

 02 ISCI-ADI

 05 ILK PICTURE IS X(20)

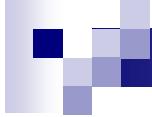
 05 ORTA PICTURE IS X(10)

 05 SOY PICTURE IS X(20)

 02 SAAT-UCRET PICTURE IS 99V99

□ ADA

```
type IsciAdiTipi is record
    ilk: String (1..20);
    orta: String (1..10);
    soy: String (1..20);
end record;
type IsciKaydiTipi is record
    IsciAdi: IsciAdiTipi;
    SaatUcret: Float;
end record;
IsciKaydi: IsciKaydiTipi;
```

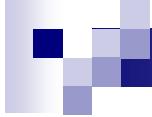


Tutanaklar (devam)

■ Tutanak tanımlama sözdizimi

□ C dilleri

```
struct IsciAdiTipi{  
    char ilk[20];  
    char orta[10];  
    char soy[20]; }  
  
struct IsciKaydiTipi{  
    IsciAdiTipi IsciAdi;  
    float SaatUcret; }  
  
struct IsciKaydiTipi IsciKaydi;
```



Tutanaklar

■ Tutanak alanlarına referans:

1. COBOL

AlanAdi OF TutanakAdi_1 OF ... OF TutanakAdi_n

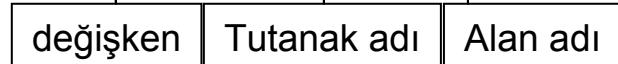
2. Diğerleri (nokta notasyonu)

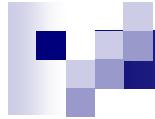
TutanakAdi_1. TutanakAdi _2. ... TutanakAdi _n.AlanAdi

■ Örnekler:

1. ORTA OF ISCI-ADI OF ISCI-KAYDI

2. IsciKaydi.IsciAdi.orta

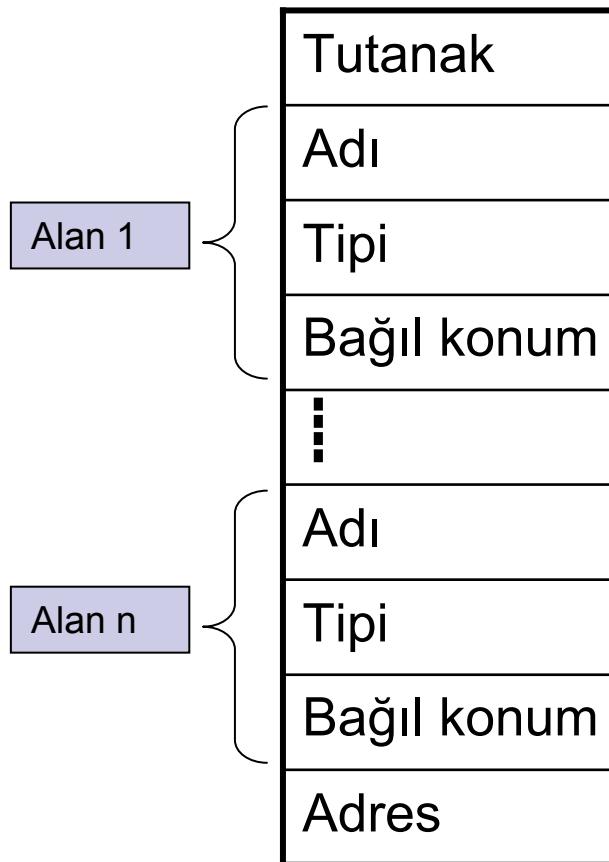




Tutanaklar

- Tüm koşulları sağlayan bir tutanak kaydına erişim bütün tutanak adlarını içermelidir.
- Cobol'da erişilen alan adında bir belirsizlik olmaması kaydıyla bazı tutanak adları atlanabilir. Örneğin ORTA OF ISCI-ADI, ORTA OF ISCI-KAYDI, ORTA daha önce tanımladığımız örneğe göre aynı veriyi işaret eder.
- Pascal'da **with** ile referanslar kolaylaştırılabilir.

Tutanaklar



Derleme zamanı tutanak (record) betimleyicisi. Yürütme zamanında gereksiz çünkü gerçek adresler hesaplanmış oluyor.

Tutanak (Record) İşlemleri

1. Atama

- Pascal, Ada ve C: tipler aynıysa izin verilir.
- Ada: Sağ taraf sabitler topağı (aggregate constant) olabilir.

2. Başlangıç

- Ada'da sabitler topağı ile.

3. Karşılaştırma

- Ada'da = ve /=; biri sabitler topağı olabilir.

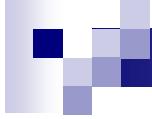
4. MOVE CORRESPONDING

- COBOL'da – aynı isimli alanları kopyalar.

```
01 INPUT-RECORD.  
  02 NAME.  
    05 LAST      PICTURE IS X(20).  
    05 MIDDLE    PICTURE IS X(15).  
    05 FIRST     PICTURE IS X(20).  
  02 EMPLOYEE-NUMBER PICTURE IS 9(10).  
  02 HOURS-WORKED   PICTURE IS 99.
```

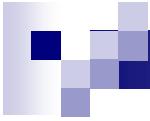
```
01 OUTPUT-RECORD.  
  02 NAME.  
    05 FIRST      PICTURE IS X(20).  
    05 MIDDLE    PICTURE IS X(15).  
    05 LAST      PICTURE IS X(20).  
  02 EMPLOYEE-NUMBER PICTURE IS 9(10).  
  02 GROSS-PAY   PICTURE IS 999V99.  
  02 NET-PAY    PICTURE IS 999V99.
```

```
MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD.
```



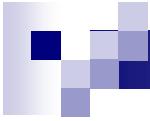
Tutanakları ve dizilimleri karşılaştırma (Comparing records and arrays)

Dizilim elemanlarına erişim tutanak alanlarına erişimden daha yavaştır. Bunun nedeni dizilim altsimgelerinin dinamik, buna karşılık tutanak alanlarının statik olmasındandır.



Ortaklık (Unions)

- **union tipi:** programın yürütülmesi sırasında değişik zamanlarda değişik tip değerler alabilen değişkenlerdir.
- Tasarım problemleri:
 1. Bir tip kontrolü yapılacaksa, nasıl bir kontrol yapılacak? Bu kontrolün dinamik olması zorunlu?
 2. Tutanaklarla bütünlüğe sahip olmalıdır.



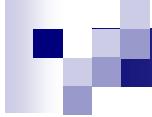
Ortaklık Örnekler

- FORTRAN – EQUIVALENCE ile
 - tip kontrolü yok
- C ve C++’da da tip kontrolü yok. Tipi belirleyen bir etiket veriye eklenmemiş.

```
union ornek { //boyu 8 byte.  
    int i;  
    double d;  
    char ch; } mu, *muptr = &mu;
```

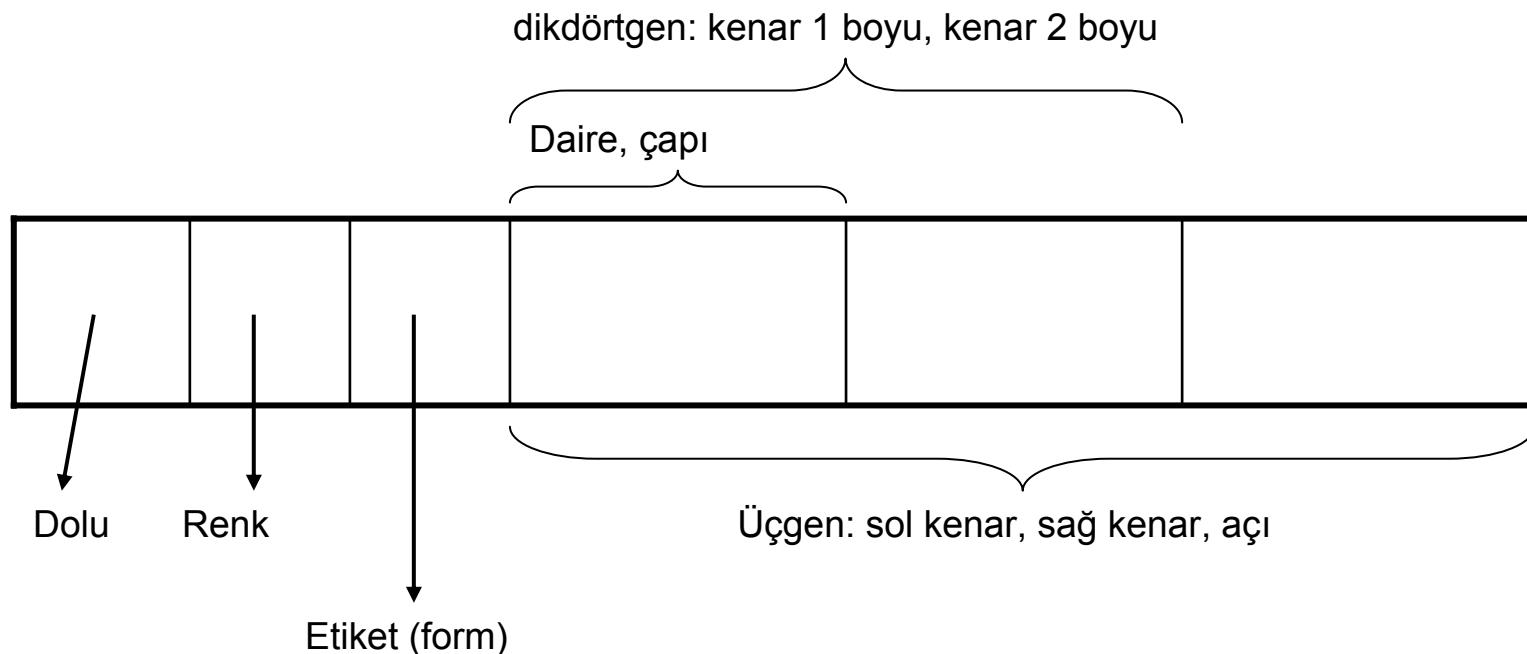
```
mu.d=3.1416; // bir alan adıyla yazar başka biriyle okursanız beklenmedik sonuç  
mu.i=3;  
mu.ch='a' ;
```

- Bu tip ortaklıklara serbest ortaklık (free union) denir.



Etiketlendirilmiş Ortaklık

Etiketlendirilmiş ortaklıkta (discriminated union) veri tipi verinin içine ayrıca yazılır. Tip kontrolünü olanaklı kılar.

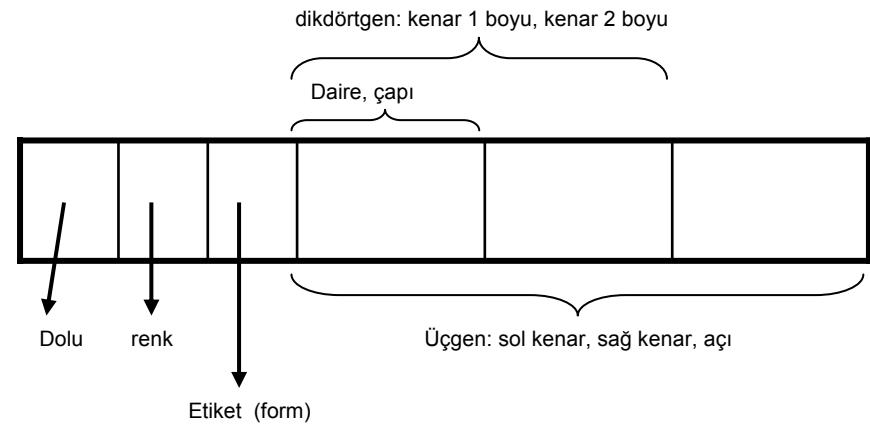


Etiketlendirilmiş Ortaklık (Ada örneği)

```
type Gsekil is (Daire, Ucgen, Dortgen);
type Renkler is (Kirmizi, Yesil, Mavi);
type Sekil (Form : Gsekil)
record
  Dolu : Boolean;
  Renk : Renkler;
  case Form is
    when Daire =>
      Capi : Float;
    when Ucgen =>
      Sol_kenar : Integer;
      Sag_kenar : Integer;
      Aci: Float;
    when Dortgen =>
      Kenar_1: Integer;
      Kenar_2: Integer;
  end case;
end record;

Sekil_1 : Sekil;
Sekil_2 : Sekil(Form => Daire);

Sekil_1 := ( Dolu => True,
             Renk => Mavi,
             Form => Dortgen,
             Kenar_1 => 15,
             Kenar_2 => 5);
```



- Sekil_1 sınırlanmadan tanımlandığından her değer atamasında farklı şekiller içine konulabilir. Ancak bu tanımlamanın tam olması gereklidir. Aynı alanlar tüm veri tipleri için kullanılabilir. Bunun için yeterince yer ayrıılır.
- Buna karşılık Sekil_2 "Daire" ile sınırlandırıldığından, sadece Daire ile ilgili veri saklanmasıında kullanılabilir. Sınırlandırıldığı için, statik olarak gerektiği kadar yer ayrıılır.

Ortaklık Örnekler

■ Pascal – hem etiketli hem de etiketsiz ortaklığını destekler.

```
type intreal =  
  record tagg : Boolean of  
    true : (blint : integer);  
    false : (blreal : real);  
  end;
```

□ Pascal'da ortaklıkda tip kontrolü etkili değildir.

■ Neden etkili değildir?:

Kullanıcı tutarsız ortaklıklar yaratabilir (çünkü etiket her bir elemana ayrıca veriliyor):

```
var blurb : intreal;  
      x : real;  
blurb.tagg := true; {integer }  
blurb.blint := 47; { doğru }  
blurb.tagg := false;{real }  
x := blurb.blreal; { real'a integer  
atanıyor. }
```

Aynı şey Ada'da:

```
Type Intreal (Tag: Boolean) is  
record  
case Tag is  
  when True => blint : Integer;  
  when False => blreal : Float;  
end case;  
end record;
```

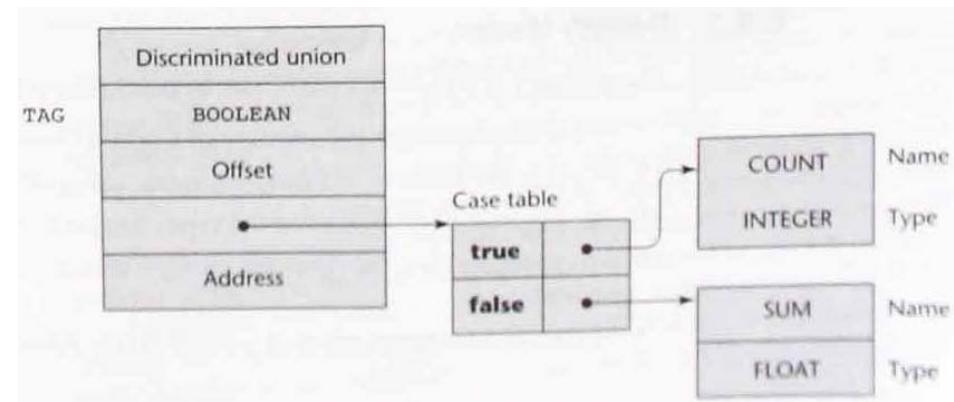
Ortaklık Örnekler

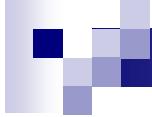
■ Ada – etiketlendirilmiş ortaklık

Neden Pascal'dan daha güvenlidir:

- a. Tag mutlaka bulunur.
- b. Kullanıcı tarafından tutarsız ortaklık yaratmak mümkün değildir, çünkü sadece etiket atanamayacağı gibi, veri de atanamaz; bu tip değişkenlere birlikte atanırlar.

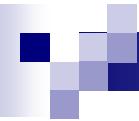
```
Type Intreal (Tag: Boolean) is
  record
    case Tag is
      when True => COUNT : Integer;
      when False => SUM : Float;
    end case;
  end record;
```





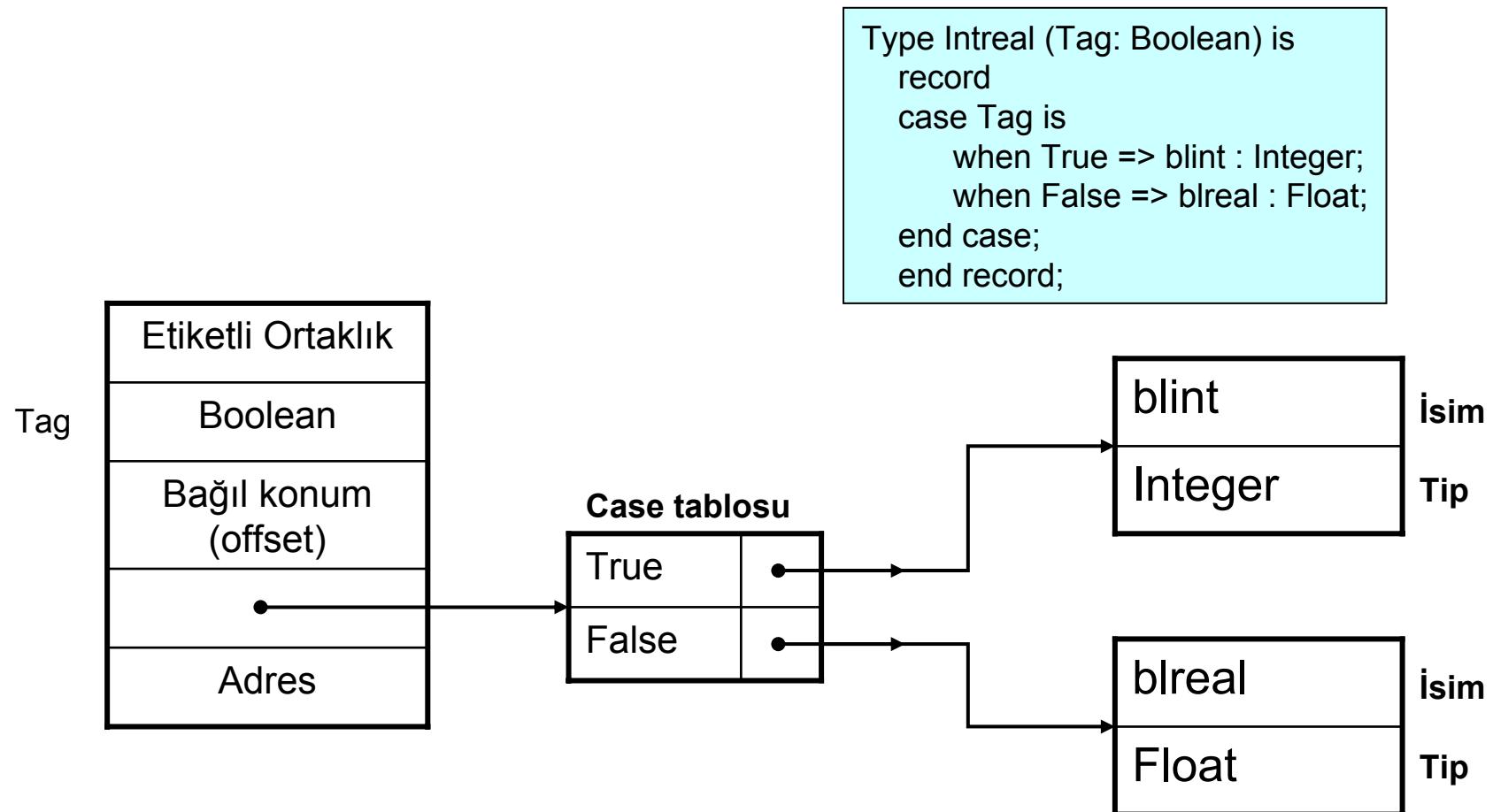
Ortaklık -- Unions

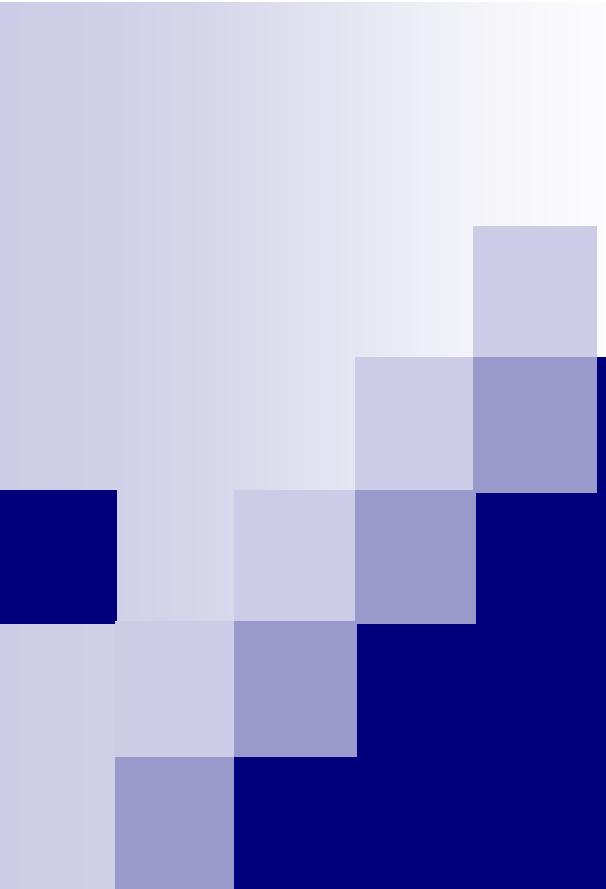
- Java'da ne tutanak (record), ne de ortaklık var.
- C# ortaklık tipini desteklemiyor.
- Sonuç olarak – Ada hariç diğer dillerde potansiyel olarak güvensiz.



Ortaklık – Unions

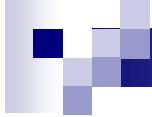
Derleme zamanı betimleyicisi





Ders 7

Veri Tipleri (devam 2)



Göstericiler (İşaretciler)

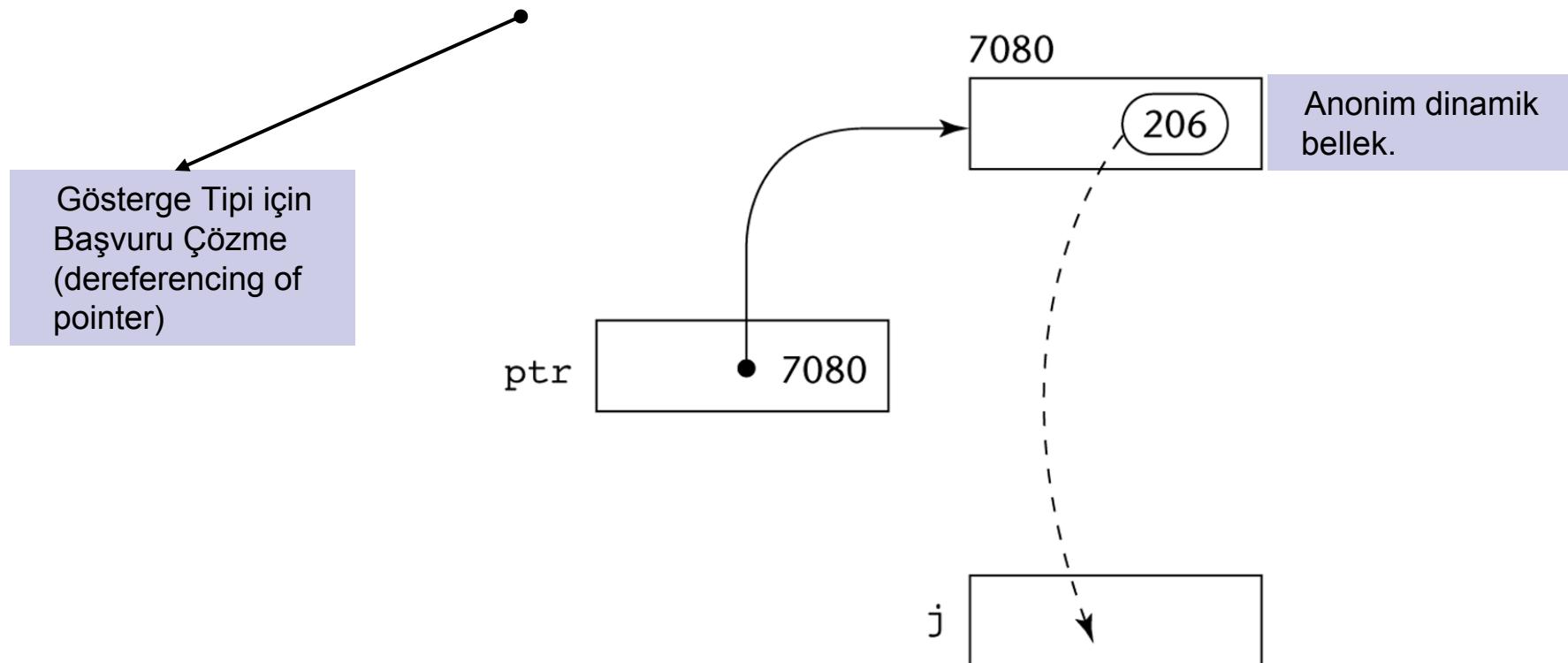
- **Gösterici** (*pointer*) tipi, belirli bir veriyi içermek yerine başka bir veriye başvuru amacıyla kullanılır. Bu nedenle, hem temel tiplerden hem de yapısal tiplerden farklı bir veri tipidir. Bir Gösterici tipi sadece bellek adreslerinden oluşan değerler veya boş(*null*) değerini içerebilen bir tiptir.
- Gösterici tipindeki değerler, gösterdikleri veriden bağımsız olarak sabit bir büyüklüktedirler

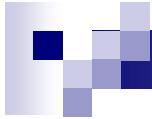
Göstericiler (İşaretciler)

- Gösterge tipindeki değişkenler iki amaçla kullanılabilir:

1. Dolaylı adresleme: Göstergeler, bellekteki değerlere erişim yolunu göstermek için dolaylı adresleme aracı olarak kullanılabilirler.

Örnek: atama işlemi: int *ptr; j = *ptr;

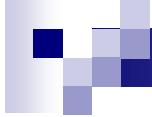




Göstericiler (İşaretçiler)

2. Dinamik bellek yönetimi: Programlarda çalışma zamanında büyüyen/küçülen veri yapılarını içeren yiğin belleğe erişmek için gösterge tipinde değişkenler kullanılabilir. Yiğin bellekte saklanan değişkenlerin tanımlayıcı isimleri olmadığı için, sadece gösterge değişkenler ile başvurulabilir.

- Yiğin bellek yönetimi için gösterge kullanımını sağlayan programlama dillerinde yiğin bellekten yer almak için bir işlemciye veya fonksiyona gereksinim vardır. Örneğin C'de malloc fonksiyonu, C++'da new işlemcisi, bellekten yer almak için kullanılabilir. Bazı dillerde, belleği serbest bırakmak için de ayrı bir işlemci veya fonksiyon vardır. C'de free fonksiyonu, C++'da delete işlemcisi bu amaçla kullanılabilir.



Göstericiler (İşaretçiler)

- C'de gösterici tipi için, atama ve başvuru çözme (*dereferencing*) olmak üzere iki temel işlem tanımlanmıştır.
- **Gösterici Tipi için Atama:** Atama işleminde bir gösterge değişkene belirli bir nesnenin adresi verilir ve bunun için bir değişkenin adresini veren & işlemcisi kullanılır.
- **Gösterici Tipi için Başvuru Çözme:** * simbolü başvuru çözme işlemcisidir. Gösterici değişkenler, kayıtlara başvuru için kullanıldığında çeşitli programlama dillerinde farklı söz dizimleri kullanılır.

Göstericilerle ilgili problemler

1. Sallanan (dangling) gösterciler (**tehlikeli**)

- Bir göstericinin gösterdiği belleğin bir şekilde sisteme iade edildiği durumlar.
- Bir tane yapalım (bellek iade ederek):

```
int *jp1, *jp2;  
jp1 = malloc(sizeof(int));  
jp2 = jp1;  
*jp2 = 5;  
free jp1;
```

```
int *jp1;  
int *jp2 = new int[50];  
jp1 = jp2;  
delete *jp2;
```

Bu durumda jp1 sallantıda çünkü gösterdiği bellek silindi.

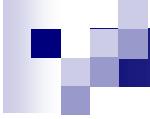
- Başka bir örnek verelim:

```
int* f () {  
    int fv = 42;  
    return &fv;  
}
```

Aşağıdaki kod gösterici dönen f'i çağırır, sonucu bir gösterici değişkene koyar, sonra bunu değiştirmeye çalışır.

```
int* p = f();  
*p = 0;
```

Fakat göstericinin gösterdiği fv f'nin içinde tanımlıdır, f'nin yaşamı bitince onun bellekte kullandığı yer de iade edilmiştir, bu şekilde kullanılması beklenmedik sonuçlar doğurur.



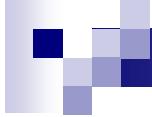
Göstericilerle ilgili problemler - 2

2. Kayıp yığın dinamik (Heap-Dynamic) değişkenler.

- Yığın dinamik değişken herhangi bir program göstERICisi tarafından göSTERİLMEMEKtedir.
- ÖRNEK:
 - a) GöSTERİCI p1 ONCE bir yığın değişKENİ göSTERİR:

```
int *p1;
p1 = (int *)malloc(sizeof(int));
```
 - b) DAHA SONRA, YENİ YARATILMIŞ BAŞKA BİRİNİ:

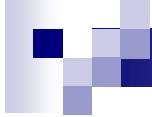
```
p1 = (int *)malloc(sizeof(int));
```
- Bu ŞEKLİDEKİ KAYİPLARA BELLEK KAÇAĞI Veya SİZİNTİSİ DENİR.



Göstericiler

1. Pascal: sadece dinamik bellek yönetimi için kullanılır.

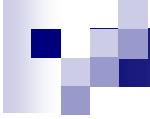
- Açıkça gösterici çözme var (postfix ^).
- Sallanan gösterciler var (**dispose** komutu nedeniyle).



Göstericiler

2. Ada: Pascal'dan biraz daha iyi

- Dinamik değişkenler göstericinin tip yaşam döngüsü bittiğinde otomatik olarak serbest bırakıldığından, bazı sallanan göstercilere izin verilmiyor.
- Bütün göstercilerin otomatik başlangıç değeri null.
- Benzer sallanan değişkenler problemi (fakat açıkça serbest bırakma nadir olduğundan daha az oluyor).

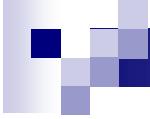


Göstericiler

3. C ve C++

- Dinamik bellek ve adres yönetimi için kullanılır.
- Açık başvuru çözme, adres kaldırma işlemleri.
- Alan tipinin sabitleştirilmesi şart değil (**void ***).
- **void *** - herseyi gösterebilir ve tip kontrolü yapılabılır (başvuru çözme işlemi yapılamaz)
- Sınırlı olarak adres aritmetiği yapılır, örneğin:

```
float stuff[100];  
float *p;  
p = stuff;  
*(p+5) → stuff[5] ve p[5]  
*(p+i) → stuff[i] ve p[i]  
(örtülü ölçekleme)
```



Göstericiler

4. FORTRAN 90 Göstericileri

- Yığın bellek veya statik bellekteki verileri gösterebilirler.
- Göstericiler sadece **TARGET** özniteliği (attribute) olan değişkenleri gösterebilirler.
- **TARGET** özniteliği tanımlama sırasında verilir:

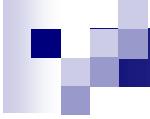
INTEGER, TARGET :: NODE

- Örtülü gösterici çözme yapılır. Bir gösterici bir ifade içinde yer alduğunda otomatik olarak çözme işlemi yapılır.
- Örtülü çözmenin yapılmaması için özel bir komut yöntemi kullanılır:

REAL, POINTER :: ptr (POINTER özniteliği)

ptr => target (burada target ya bir gösterici ya da TARGET öznitelikli bir değişken)

- Bu **ptr**'ın **target** ile aynı değeri almasına neden olur.

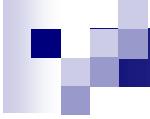


Göstericiler

5. C++ Referans Tipleri

- Çoğunlukla fonksiyonların tanımlarında göstermelik parametre (programlamada işlevi olmayıp yer tutma için kullanılan değişken) olarak kullanılırlar.
 - Hem adresle geçirme, hem değeriyle geçirme avantajı vardır.
- Örtülü olarak başvuru çözülen sabit göstericilerdir.
- Sabit olduklarıdan bir değişkenin adresiyle başlatılmaları gereklidir ve daha sonradan bu adres değişimeyecektir.
- Referans tip değişken çağrıran fonksiyonda tanımlanınca, çağrıran ve çağrılan arasında iki yönlü haberleşme olanağı sağlanır. Aynı amaçla göstericiler de kullanılabilir ama bunların başvuru çözümlerinin açıkça yapılması gereklidir ki programın okunabilirliğini azaltır.
- Referans tip değişkenler “ve” işaretini ile başlar:

```
int result;  
int &ref_result = result;  
...  
ref_result = 100;
```



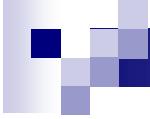
Göstericiler

6. Java – Sadece referans tip göstericiler bulunur.

- C++ referans tiplerinden geliştirilmiştir.
- Sınıf nesnelerini gösterirler (yığın bellekte duran).
- Bu nedenle gösterici aritmetiği yoktur.
- Kullanılmayan belleği sisteme iade eden açık bir komut yoktur, bunun yerine çöp toplama kullanılır (garbage collection).
- Bu nedenle sallanan referanslar da yoktur.
- Referans çözme her zaman örtülüdür (implicit).
- C++’dan farklı olarak gösterdiği nesneler değişebilir.
- Örnekte “String” standart Java sınıfı (class):

```
String str1;  
...  
str1 = "Bu bir dizgidir";
```

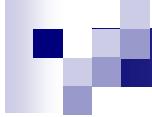
İlk satırda str1 bir dizgi nesnesine referans tip gösterici olarak tanımlanıyor.
Başlangıç değeri “null”. Daha sonra “Bu bir dizgidir” nesnesini gösterecek atama yapılıyor.



Göstericiler

6. C# – Java referans tip göstericiler, C++ göstericilerle birlikte bulunur.

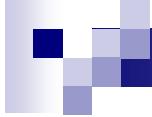
- Ancak kullanılmaları istenmez.
- Kullanan “metot”ların “unsafe” olarak tanımlanması gereklidir.
- Esas olarak C ve C++ kodları ile birlikte çalışabilmeleri için konulmuştur.



Göstericiler

■ Değerlendirilmesi:

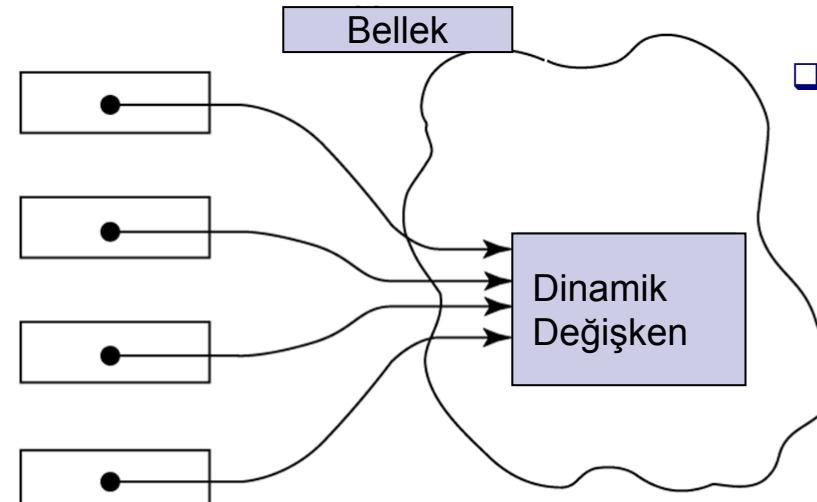
- 1) Sallanan göstericiler, sallanan nesneler, yığın bellek yönetimi ile birlikte problemler.
- 2) Göstericiler programlama dillerinde “go to” gibidirler:
 - “go to” programda bir sonraki işlemde gidilebilecek noktayı genişletir;
 - “pointer” da aynı şekilde erişilebilecek bellek yerini genişletir.
- 3) Dinamik veri yapıları için gerekiğinden onlardan vazgeçemeyiz.



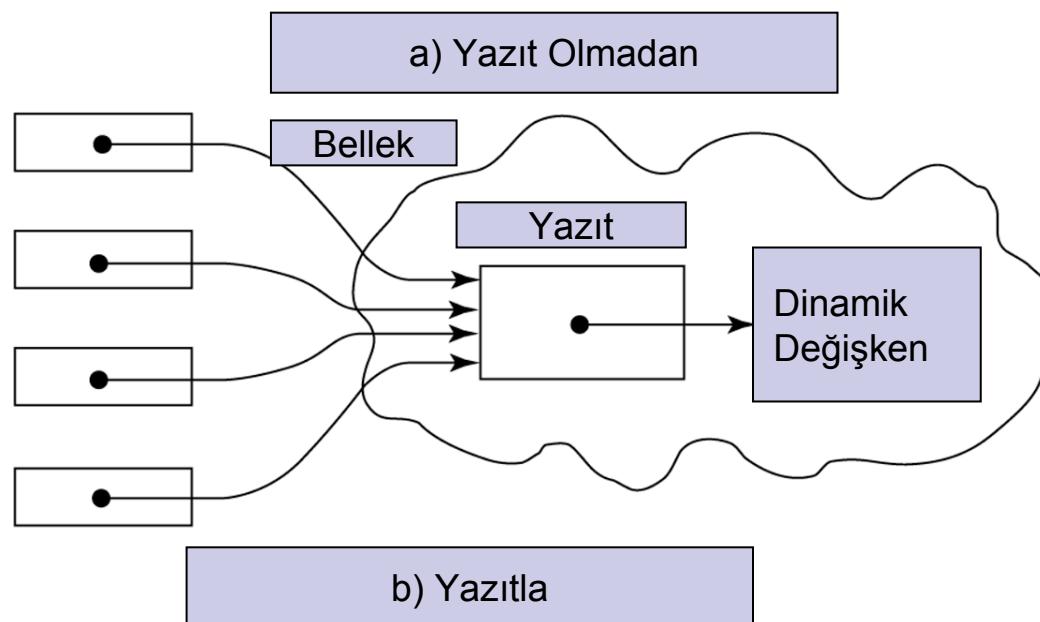
Göstericiler

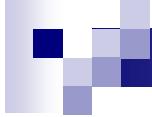
- Referans ve göstericilerin temsili
 - Büyük bilgisayarlar tek bir değer kullanır.
 - Intel işlemcileri “segment” ve “offset” kullanır.
- Sallanan gösterici problemine çözüm
 1. **Yazıt (Tombstone)**: Yığın dinamik bellek değişkenini gösteren ek bellek hücresi (Lomet 1975).
 - Gerçek gösterici yazıt üzerindeki belleği gösterir; o da gerçek yığın dinamik değişkeni.
 - Yığın dinamik değişken sisteme geri verildiğinde yazıt kalır ancak içeriği “nil” olur. Böylelikle bu değişkenin artık olmadığını bildirir.

Dinamik değişkenlerin gerçekleştirilmesi Sallanan gösterici problemine çözüm



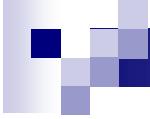
Bellek ve zaman kaybına neden olduğundan kullanılmadı.





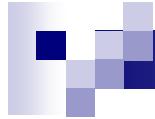
Sallanan gösterici problemine çözüm

2. Kilit ve Anahtar (Locks and keys) (Fisher ve LeBlanc, 1977, 1980):
 - Göstericiler [anahtar, adres] çiftlerinden oluşturulur.
 - Yığın dinamik değişken ise [kilit, değişken] çiftinden oluşur.
 - Yığın dinamik değişken yaratıldığında kilit ve anahtara aynı (tam sayı) değer atanır. Bu değişkene referans verildiğinde anahtar ve kilit değerlerinin eşit olması kontrolü yapılır.
 - Değişken silindiğinde kilit değeri de kullanılmayan bir değer yapılır. Böylelikle başka bir gösterici aynı değişkeni daha sonra gösterdiğinde anahtar ve kilit değerler tutmayacağından hata mesajı verilir.
- **Sallanan gösterici probleminin esas çözümü programcıdan bellek iade yetkisini almaktır. Java ve C# referans değişkenleri ve Lisp bu yetkiyi vermeyip, kullanılmayan bellegi daha sonra kendisi belirleyerek sisteme iade etme/yeniden kullanma yöntemini kullanmaktadır.**



Göstericiler

- Yığın bellek yönetimi (Heap management)
 - Dilin tasarım problemi olmaktan çok, dilin uygulanması problemi.
 - Sabit boylu hücre (cells) – Değişken boylu hücre
 - Sabit boyuta örnek Lisp.
- Kullanılmayan belleği toplamanın iki yöntemi vardır: **Referans sayıcıları** (Reference counters) (*aceleci yaklaşım*) ve **çöp toplama** (garbage collection) (*tembel yaklaşım*)
 1. Referans sayıcıları: dinamik yığın bellekteki her değişken için bir sayı verilir ve kaç gösterici tarafından değişken gösteriliyorsa bu bilgi saklanır. Sayıcı sıfır olduğu zaman bellek kullanılmıyor demektir, sisteme iade edilir.
 - Dezavantajı: bellek alanı ve işlem zamanı kullanılır (özellikle hücreler küçükse). Dairesel tanımlanmış göstericiler için problem.



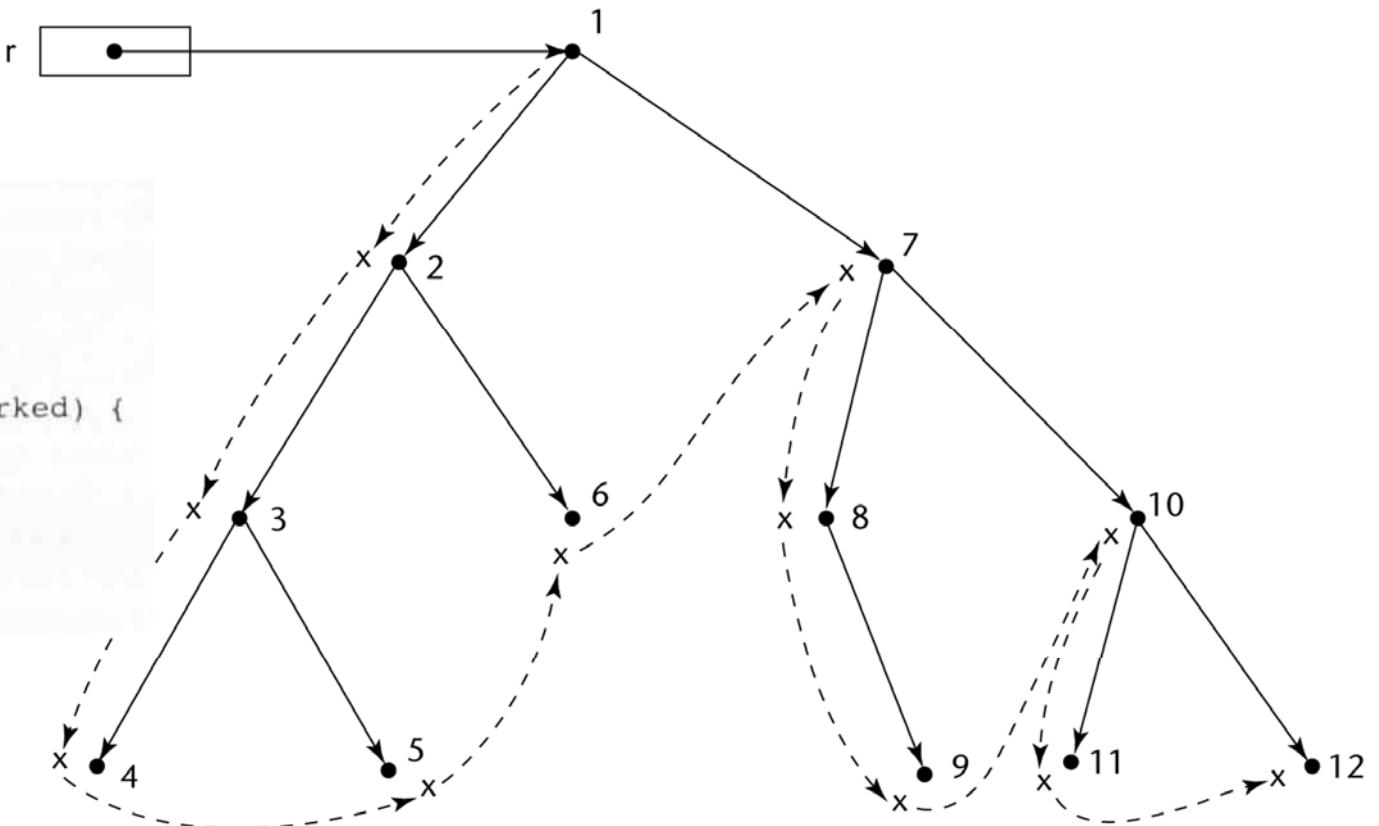
Göstericiler

1. Çöp toplama: bellekte yer olduğu sürece yeni bellek al ve kullan, işin bitince terket; bellek tükenince çöpleri topla.
 - Her hücrenin işaretleme için kullanılan bir ek bit'i olur.
 - Bütün dinamik yiğin bellekten kullanılan hücreler çöp olarak işaretlenir.
 - Bütün göstericiler taranır ve yiğin bellekte bu şekilde gösterilebilen hücreler “çöp değil” şeklinde işaretlenir.
 - Sonuçta çöp olarak işaretli kalan hücreler, kullanılabilir hücrelere eklenir.
 - Dezavantajı: en çok ihtiyacınız olduğunda, en yavaş çalışır. Programın bellek ihtiyacı artınca, büyük bellekte çöp toplama daha uzun sürdüğünden yavaşlar.

İşaretleme algoritması

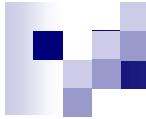
```
for every pointer r do
    mark(r)

void mark(void * ptr) {
    if (ptr != 0)
        if (*ptr.tag is not marked) {
            set *ptr.tag
            mark(*ptr.llink)
            mark(*ptr.rlink)
        }
}
```



Dashed lines show the order of node_marking

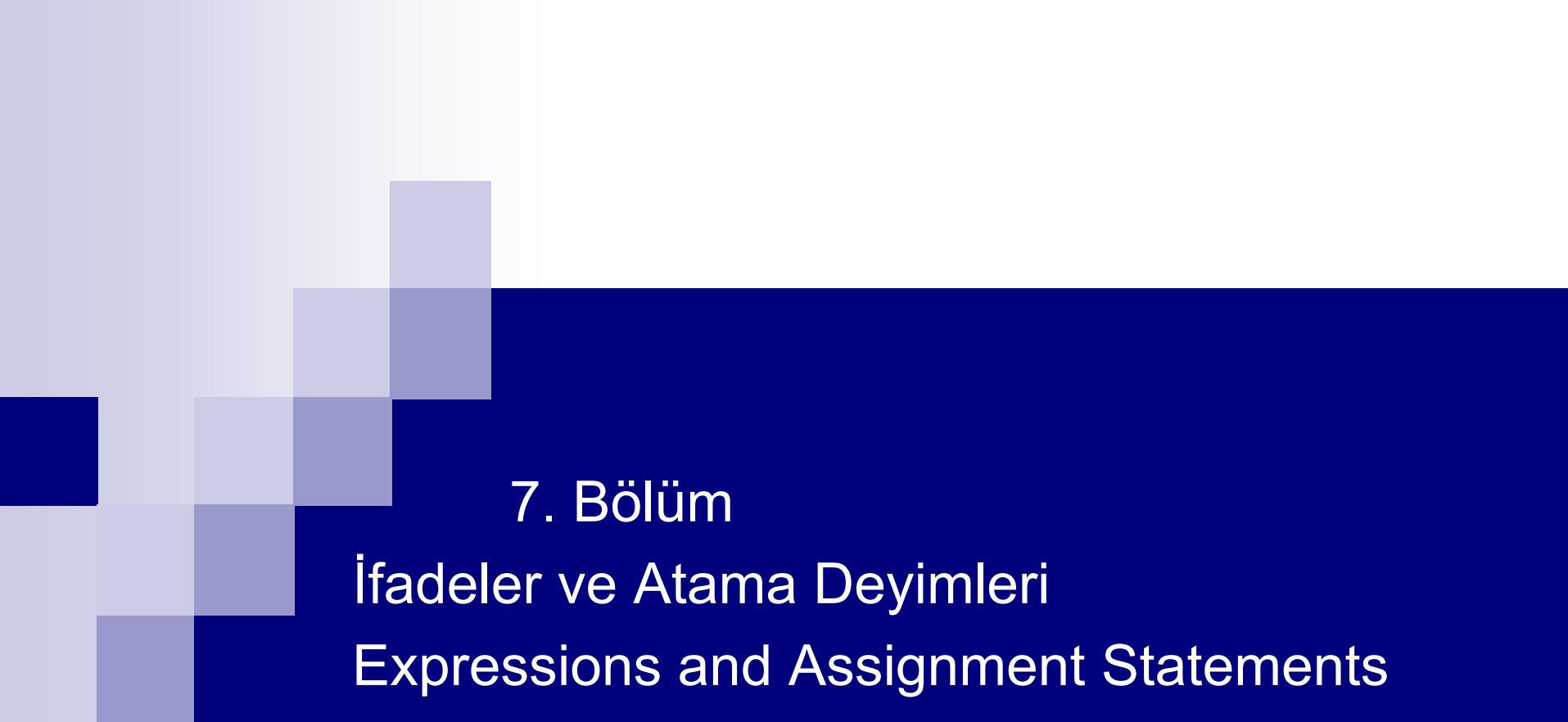
Kesikli çizgi işaretlemenin sırasını gösteriyor.



Göstericiler

■ Değişken boylu hücreler

- Hücre boyları değişkense bahsedilen güçlükler ilaveten başka güçlükler de gözlenir.
- Eğer çöp toplama yöntemi kullanılıyorsa ek olarak aşağıdaki zorluklar gözlenir:
 - Hücrelerin boyları sabit olmadığından işaretlenen hücrenin boyutları bilinemez. Çözüm için her hücrenin başına hücrenin boyu yazılabilir.
 - Kullanılan hücrelerin zincirini takip etmek de zor. Çünkü her hücre farklı ve bir sonraki hücreyi gösteren gösterici farklı yerde olabilir.
 - Kullanılabilir alanın listesinin tutulması da hücre boyları ve yapıları farklı olunca daha zor.
- Eğer referans sayacı yöntemi kullanılırsa ilk iki problem ortadan kalksa da kullanılabilir alanların yönetim zorlukları kalır.



7. Bölüm

İfadeler ve Atama Deyimleri

Expressions and Assignment Statements

Başlıklar

- Başlangıç
- Aritmetik ifadeler
- Yüklenmiş işleyiciler (Overloaded Operators)
- Tip çevirme (Type Conversions)
- İlişkisel ve mantıksal ifadeler (Relational and Boolean Expressions)
- Kısa-devre değerlendirmesi (Short-Circuit Evaluation)
- Atama komutları (Assignment Statements)
- Karışık mod atamaları (Mixed-Mode Assignment)

Başlangıç

- Bu bölümde esas olarak buyurgan (imperative) dilleri inceleyeceğiz.
- İfadeler programlama dillerinde berimin (computation) temel yöntemidir.
- İfadelerin değerlendirilmesini anlamak için işaretlerin (operator) sırasını ve işlenenlerin (operand) değerlendirilmesini bilmemiz gereklidir.
- Buyurgan dillerde temel olan atama komutlarıdır.

Aritmetik ifadeler

- Aritmetik ifadelerin işlenmesi ilk programlama dillerinin geliştirilmesi için önemli motivasyon kaynaklarından biri olmuştur.
- Aritmetik ifadeler; işaretler, işlenenler, parantezler ve fonksiyon çağrılarından oluşur.

Aritmetik ifadelerde tasarımla ilgili hususlar

- 1) İşleçlerin öncelik kuralları nelerdir?
- 2) İşleçlerin birleşmelilik (associativity) kuralı nedir?
- 3) İşlenenlerin değerlendirilmesinde sıra nedir?
- 4) İşlenenlerin değerlendirilmesinde yan etkilerde sınırlama var mıdır?
- 5) Dil, kullanıcı tanımlı fazla yüklenmiş işaretlere izin veriyor mu?
- 6) İfadelerde hangi tip karıştırmasına izin veriliyor?

Aritmetik ifadeler

- Birli (**unary**) işlecin bir işleneni olur.
- İkili (**binary**) işlecin iki işleneni olur.
- Üçlü (**ternary**) işlecin üç işleneni olur.
- n'li (**n-ary**) işlecin “n” tane işleneni olur.
- Örnekler:
 - $A + (-B) * C$ (ikili + ve *, birli -)
 - Java'da birli + var. Short ve byte'ı int yapıyor, tip değiştiriyor.
 - <mantıksal ifade>?<doğruysa ifade> : <yanlıssa ifade>; (üçlü işaret)

İşleç Öncüllüğü (operator precedence)

- İşleç öncüllüğü kuralları peşpeşe gelen, aralarında en çok bir işlenen olan işaretlerin hangi sıra ile işleneceğini belirler.
- Tipik öncüllük sıralaması
 1. () Parantezler.
 2. Tekli işaretler: $A + (-B) * C$ doğru, $A + -B * C$ yanlış.
 - ikili işaretler
 3. ** (kuvveti işlevi, eğer dil destekliyorsa).
 4. *, / Çarşı ve bölü.
 5. +, - Artı ve eksı.

Öncülük	Fortran	C tabanlı dil	Ada
Yüksek	**	art takı ++,--	**, abs
	*, /	öntakı ++,--, birli +,-	*, /, mod, rem
	+,-	*, /, %	birli +, -
Düşük		ikili +, -	ikili +, -

İşleçlerin birleşmeliğin (associativity) kuralı

- İşleçlerin birleşmeliğin (associativity) kuralı, aynı öncüllükte, aralarında en fazla bir işlenen olan işaretlerin hangi sıra ile işleneceğini belirler.
- Tipik birleşmeliğin kuralları:
 - Genelde soldan sağa, ** hariç, ** sağdan sola.
 - Bazen bir işaretler sağdan sola (örneğin FORTRAN)
- APL değişik; bütün işaretler aynı öncüllükte ve sağdan sola işleniyorlar.
- Öncüllük ve birleşmeliğin kuralları parantezlerle değiştirilebilir.
- Örnekler:
 - $A^{**} B^{**} C$: (visual basic de $^$)(Ada'da birleşmeliğin yok, parantez şart)
 - $-A - B$: tekli ve ikili eksilerin birleşmeliği aynı da olsa, farklı da olsa, sonuç aynı.

Dil	Birleşmeliğin kuralı
Fortran	soldan: *, /, +, - sağdan: **
C tabanlı diller	soldan: *, %, /, ikili +, ikili - sağdan: ++, --, birli +, birli -
Ada	soldan: ** hariç hepsi tanımsız: **

İşlenenlerin işlenme sırası(Operand evaluation order)

İşlem (yan etkiler yoksa sıra önemli değildir):

1. Değişkenler: değerini bellekten al.
2. Sabitler: bazen bellekten getirilir, bazen de makine kodunun içine konulmuş olur ve kendiliğinden gelir.
3. Parantezli ifadeler: içindekiler öncelikle işlenir.
4. Fonksiyonlara referanslar: yan etkilerinden dolayı önemli
 - İşlenme sırası çok önemli.

Fonksiyonel yan etkiler

- Fonksiyonel yan etkiler – bir fonksiyon iki yönlü veya lokal olmayan bir değişkeni değiştirdiğinde ortaya çıkar.
- Fonksiyonel yan etkilere örnek :

```
int fun(int *u)
    {   *u = *u/2;
        return *u; }

a = 10;
b = a + fun(&a);
/* fun parametresini değiştiriyor */
```

- Global değişkenlerle de aynı sorun.
- Fonksiyonların birden çok değer dönmeleri gerektiğinden, fonksiyonları parametrelerini değiştiremez veya global değişkenleri değiştiremez yapmak pratik değil.
- İşlenenlerin işlenmesi sırasını belirlemek de derleyicilerin optimizasyonlarını bozacağından pratik değil.
- Bununla birlikte, Java da işlenenler soldan sağa işlenirler ki, burada bahsettiğimiz sorunla karşılaşmaz.

Yüklenmiş İşleçler

- Bir işlecin birden fazla kullanılmasına işaret (fazla) yüklenmesi (*operator overloading*) denir.
- Bazıları çok bilinir (örneğin C'de int ve float için +, ek olarak Java'da dizin birlestirmesi)
- Bazıları potansiyel olarak problemdir
 - örneğin C ve C++ da *, &: ikili olarak çarpı ve "bit bazında ve" olduğu gibi, tekli olarak da adres çözme ve adres işaretleri olarak da kullanılırlar.
 - Derleyicinin hata bulması zorlaşır (işlenenin eksikliği fark edilebilmeli).
 - x & y yerine &y yazmak gibi. Çok farklı anlamları var.
 - ortalama = toplam / n; gibi. Toplam ve n int'se ne olacak.
 - Farklı amaçlarla aynı işaretlerin kullanılması okunabilirliği zorlaştırır.
 - Farklı amaçlar için farklı sembollerle düzeltilebilir (Örneğin Pascal'da tam sayı bölme için div kullanılması gibi).

Yüklenmiş İşleçler

- C++ ve Ada kullanıcı tarafından tanımlanmış yüklenmiş işaretlere izin verir.
- Örneğin * ve + matris çarpımı ve toplamı için ek olarak tanımlanırsa

MatrixAdd (MatrixMult (A, B) , MatrixMult (C, D)) ; yerine
A * B + C*D yazılabilir.

- Potansiyel problemler:
 - Kullanıcılar anlamsız işaretler tanımlayabilirler.
 - İşleçler anlamlı olsa bile okunabilirlik zarar görebilir.
- C++'da . (class ve structure member operator) ve :: (scope resolution operator) işaretlerine yeni yüklemeler yapılamaz.
- C++'da olan bu özellik Java'ya konulmamış, daha sonra C#'da yeniden ortaya çıkmıştır.

Tip Çevirme (Type conversion)

- Daraltıcı çevirme: eğer çevrilen tip başlangıç tipinin bütün değerlerini kapsamıyorsa buna daraltıcı çevirme denir. Örneğin **float** → **int**.
- Genişletici çevirme: eğer çevrilen tip başlangıç tipinin bütün değerlerinden fazlasını kapsiyorsa buna genişletici çevirme denir. Örneğin **int** → **float**.

Tip Çevirme

- Eğer bir işlecin işlenenleri farklı tiplerdense buna karışık biçimli işlem denir.
- Karışık biçimli işlemler genellikle işlemciler tarafından desteklenmediğinden işlenenlerin derleyici tarafından belli bir tipte toplanması gereklidir. Bu işlem açıkça yapılabileceği gibi örtülü olarak da yapılabilir.
- Örtülü çevirmenin avantaj/dezavantajı:
 - Derleyicinin hata bulma yeteneğini azaltır. Dezavantaj.
 - Esnekliği artırır. Avantaj.
- Çoğu dilde örtülü çevirmeler genişletici yönde olur.
- Ada'da örtülü çevirme yoktur.
- Örtülü çevirme örneği:

C (hata vermez)

```
int a;  
float b, c, d;  
  
...  
d=b*a;
```

Ada (hata verir)

```
a: Integer;  
b, c, d: Float;  
  
...  
d:=b*a; //bu satırda "c"  
// yanlışlıkla "a" yazılmış.
```

Açıkça tip çevirme (type casting)

Ada: fonksiyon çağrıma gibidir.

FLOAT (INDEX) –INDEX, INTEGER tipidir.

C benzeri diller:

(long int) speed /*speed, float tipidir*/

İlişkisel ve mantıksal ifadeler

Relational and Boolean Expressions

■ İlişkisel ifadeler:

- İlişkisel işaretler kullanılır.
- Mantıksal bir sonuç alınır.
- İşaretler dillere göre değişir
(Örneğin eşit değildir: !=, /=, .NE., <>, #)

İlişkisel ve mantıksal ifadeler

■ Mantıksal ifadeler

- İşlenenler mantıksal ve sonuç da mantıksal.
- İşleçler:

FORTRAN 77	FORTRAN 90	C	Ada
.AND.	and	&&	and
.OR.	or		or
.NOT.	not	!	not
		^	xor

İlişkisel ve mantıksal ifadeler

- C'de mantıksal tip yok – int tipini kullanır ve sıfırsa yanlış (false), değilse doğru (true) olur.
- C'nin tuhaf bir özelliği:
 - $a < b < c$ doğru bir ifadedir fakat bekledığınız sonucu vermeyebilir.
 - Soldan sağa doğru işlendiği için önce a'nın b'den küçük olup olmadığına bakılır. Diyelim ki bu ifade yanlış olsun. Sonuç 0 olur, c'nin sıfırdan büyük olup olmadığı bir sonraki ilişkisel ifadedir. c, hiçbir zaman b ile karşılaştırılmaz.

İlişkisel ve mantıksal ifadeler

- Ada işleçlerinin öncelikleri:

- ** , abs , not

- * , / , mod , rem

- unary - , +

- binary + , - , &

- relops , in , not in

- and , or , xor , and then , or else

- C, C++, ve Java'nın 40 dan fazla işlemci ve 15 den fazla öncelik seviyesi vardır.

Kısa devre değerlendirmesi

Short Circuit Evaluation

- Bir ifadede bütün işlemleri yapmadan sonucun bulunmasına kısa devre değerlendirmesi denir.
- Örneğin:
 $a = 0;$
 $c = (13*a)*(b/13-1);$ //bu ifadede ikinci parantezin hesaplanması gereksiz çünkü birincisi 0.
- Problem: tablodan değer bulma

```
index = 1;  
while(index <= length) && (LIST[index] != value)  
    index++;
```

while mantıksal kısmında ilk test yanlış olunca durulmasa, ikincisi işlenecek ve "array out of range" gibi bir hata verilecek.

Kısa devre değerlendirmesi

- C, C++, ve Java: kısa devre değerlendirmeyi bütün mantıksal işaretler için yapar (`&&` ve `||`), fakat bit bazında mantıksal işaretler için yapmaz (`&` ve `|`)
- Ada: programcı kısa devre istiyorsa kendisi kullanır (`and →and then`, `or →or else`)

```
Index := 1;  
while (Index <= Listlen) and then (List (Index) /= Key)  
  loop  
    Index := Index + 1;  
  end loop;
```

- Kısa devre değerlendirmesi ifadelerde potansiyel yan etki kaynağı da olabilir:
Örneğin: `(a > b) || (b++ / 3)`
a, b den büyük olduğu sürece kısa devre nedeniyle b'nin bir büyütülmesi yapılmayacak.

Atama Komutları

■ İşleç sembolü:

1. $=$ FORTRAN, BASIC, PL/I, C, C++, Java
2. := ALGOLs, Pascal, Ada

Atama Komutları

■ Daha karmaşık atamalar:

1. Çok hedefe (PL/I)

A, B = 10

2. Koşullu hedefler (C, C++, ve Java)

(first==1) ? total : subtotal = 0

3. Birleşik atama komutları (C, C++, ve Java)

sum += next;

4. Tekli atama komutu (C, C++, ve Java)

a++;

- **a++** : iki tekli işaret bir değişkene uygulandığında değerlendirme sağdan soladır. Bu nedenle:

- **a++ → - (a++)** şeklinde değerlendirilir.

5. C, C++, ve Java "=" işaretini aritmetik işaret olarak görürler.

Örneğin:

a = b * (c = d * 2 + 1) + 1

Bu yapı ALGOL 68 den alınmıştır.

Atama Komutları

■ Bir ifade olarak atama komutu

- C, C++, ve Java'da, atama komutu sonuç döner.
- Bu nedenle ifadelerde işlenen olarak da yer alabilir.
Örneğin:

```
while ((ch = getchar()) !=EOF) {...}
```

□ Dezavantaj

■ Başka tip yan etkiler:

```
a = b + (c = d/b++) - 1;
```

■ Yazım hatası olasılığı:

```
if(x = y) ...
```

```
if(x == y) ...
```

Cok farklı anlamları var. Java ve C# "if" içinde atamaya izin vermez.

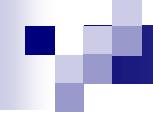
Karışık biçimli atamalar

- FORTRAN, C, ve C++, her türlü sayısal değerler, sayısal değişkene atanabilir, gerekli çevrimler yapılır.
- Pascal'da interger'lar real'lara atanabilir fakat tersi doğru değil. Bunun için gerçek sayının üste mi tamlanacağı, aşağıya mı indirileceğinin belirlenmesi gereklidir.
- C# ve Java'da sadece genişletici örtülü çevrimler atamalarda çalışır.
- Ada'da atama için örtülü çevrim yoktur.



Bölüm 8

Komut seviyesi kontrol yapıları
Statement-Level Control Structures



Chapter 8 Topics

- Başlangıç
- Seçme komutları
- Döngü komutları
- Koşulsuz atlama komutu
- Korunan komutlar
- Sonuçlar

Başlangıç

■ Akış kontrolünün seviyeleri:

1. İfadeler içinde,
2. Program birimleri arasında,
3. Program komutları arasında.

Başlangıç

■ Gelişimi:

- FORTRAN I kontrol komutları (aritmetik if) doğrudan IBM 704 donanımını tasarlayanlar tarafından hazırlanmıştır.
- 1960lardan 70lerin ortalarına kadar bu konudaki çalışmalar devam etmiştir.
- Önemli teorem: **Bütün akış diyagramlarının bir mantıksal döngü ve bir de iki yön seçmeli mantıksal ifadelerle kodlanabileceği ispat edilmiştir** (Böhm ve Jacopini, 1966).
- Bu ispat koşulsuz “go to” komutuna, potansiyel olarak kimi zaman faydalı olabilse de, gerek olmadığını göstermektedir.
- Programcılar programın kolay yazılp okunabilirliğine teorik araştırmalardan daha fazla önem verirler. Bu nedenle örneğin sadece while döngüsü yetebilecekken, for, do, vs. gibi başka döngüler de programlama dillerine girebilmektedir.
- Bir dilde çok sayıda akış denetim deyiminin bulunmasının dilin yazılabilirliğini artırması nedeniyle, tüm programlama dilleri çok sayıda akış denetim deyimi içerirler. Öte yandan, bir dildeki akış denetim deyimlerinin sayısı arttıkça dilin okunabilirliği azalmaktadır.
- Esas sorun, yazılabilmesini kolaylaştmak uğruna bir dilin basitlik, boyut ve okunabilirliğinden ne kadar feragat edileceğidir.

Başlangıç

- Bir kontrol yapısı (*control structure*) bir kontrol komutu ve onun kontrolündeki komutlardan oluşur.
- Önemli sorulardan biri, kontrol yapılarının birden çok girişi olmalı mıdır?
 - Birden çok girişin dile fazla bir şey katmadığı genel bir inanıştır.
- Birden çok çıkış ise tehlike yaratmadığından genel kabul görmüş bir özelliklektir.
- Esas tasarım sorusu:
 - Bir dil seçme ve ön sınamalı döngülerden (pretest logical loops) öte hangi kontrol komutlarına sahip olmalıdır?

Seçme komutları

- Bir seçme komutu (**selection statement**) yürümekte olan programda iki veya daha fazla yoldan birini seçmemizi sağlar.
- İki sınıfa ayrılır:
 - İki yollu seçimciler
 - Çok yollu seçimciler

İki yollu seçme komutları

■ Genel şekli:

```
if <kontrol ifadesi>
    then <ifade>
        else <ifade>
```

■ Tasarımla ilgili hususlar:

1. Kontrol ifadesinin şekli ve tipi ne olacak?
2. “**then**” ve “**else**” terimleri nasıl belirlenecek?
3. İç içe geçmiş seçeneklerin anlamları nasıl belirlenecek?

İki yolu seçme komutları

- FORTRAN Örneği:
 - FORTRAN IF: **IF** (<mantıksal ifade>) <komut>
- Problem: sadece bir komut seçebilir. Programı bir komut grubuna yönlendirmek için **GOTO** kullanmak zorunlu.
- **IF (.NOT. <koşul>) GOTO 20**

... }
 ... }
 } Koşul sağlanırsa çalışacak komutlar

20 CONTINUE

- Olumsuz mantık okunabilirliği azaltıyor.
- Bu problem FORTRAN 77'de çözüldü.
- Sonraki dillerde birleşik belgelerin kullanılması yöntemleri geliştirildi.

İki yollu seçme komutları

■ Örnek: ALGOL 60 **if**:

if (<koşul>)

then <komut> (then terimi)

else <komut> (else terimi)

Komutlar tek başına da olabilir, birleşik de.

İki yolu seçme komutları

- İçiçe seçiciler

- Örneğin Java:

```
if (toplam==0)  
    if (sayi==0) ...  
    ...  
else ...
```

- **else** hangi **if**'e bağlanır?

- İlk **if** ile aynı hizada, ona mı bağlı?

- Java'nın statik anlam kuralı: **else** en yakın **if**'e gider.

- Perl'de bu problem yoktur çünkü komut kısımları mutlaka kıvrık parantezler içine konur.

- Else'in bağlı olduğu if kesinleştirilmek isteniyorsa:

```
if (toplam==0) {  
    if (sayi==0) ...  
}  
else ...
```

İki yollu seçme komutları

■ FORTRAN 90 ve Ada çözümü – özel kelime ile bitirmek

□ Örneğin Ada:

```
if ... then  
  if ... then  
    ...  
  else  
    ...  
  end if  
else  
  ...  
end if
```

```
if ... then  
  if ... then  
    ...  
  end if  
else  
  ...  
end if
```

□ Avantaj: Okunabilirlik

Çoklu seçme komutları

- Bir programdaki akışı belirlemek için ikiden fazla yol olduğu zaman **çoklu seçim komutları** kullanılır.
- Tasarımla ilgili hususlar:
 1. Kontrol ifadesinin tipi ve şekli nasıl olacak?
 2. Seçilebilir bölümler nasıl belirlenecek?
 3. Programın çoklu yapıdaki akışı sadece bir bölge ile mi sınırlı olacak?
 4. Seçimde temsil edilmeyen ifadelerle ilgili ne yapılacak?

Çoklu seçme komutları

■ Erken çoklu seçme komutları:

1. FORTRAN arithmetic **IF** (üçlü seçme komutu)

IF (<aritmetik ifade>) **N1** , **N2** , **N3**

■ Kötü özellikleri:

- Kılıflanmamış (not encapsulated); seçilebilir bölgeler programın içinde her yerde olabilir.
- **GOTO** gereklidir.

2. FORTRAN hesaplanmış **GOTO** ve atanmış **GOTO**

Çoklu seçme komutları

■ Modern çoklu seçme komutları

1. Pascal case (Hoare'in ALGOL W'ya katkısı)

```
case <ifade> of
```

```
    SabitListe_1 : Komut_1;
```

```
    ...
```

```
    SabitListe_n : Komut_n
```

```
end
```

Çoklu seçme komutları

■ Tasarım seçimleri (Pascal):

1. Seçme ifadesi herhangi bir sıra tipinde(**int**, **boolean**, **char**, **enum**) olabilir.
2. Seçilen bölgede tek komut da olabilir, bir grup komutta.
3. Yapının her yürütülmesinde bir bölgesi kullanılır.
4. Pascal'da (Wirth), karşılığı olmayan seçme yapıldığında sonuç belirsizdir, 1984 ISO Standardına göre bu işlem hata mesajı verir.
5. Birçok dilde bu durumlar için **otherwise** veya **else** terimleri konulmuştur.

Çoklu seçme komutları

2. C, C++ ve Java **switch**

```
switch (<fade>)
```

```
{
```

```
    SabitIfade_1 : ifade_1;
```

```
...
```

```
    SabitIfade_n : ifade_n;
```

```
    [default : ifade_n+1]
```

```
}
```

Çoklu seçme komutları

■ Tasarım seçimleri: (**switch** için)

1. Kontrol ifadesi sadece tam sayı tipi olabilir.
2. Seçilebilir kısımda da birkaç komut peş peşe veya bir blok içinde olabilir.
3. Birden çok kısım peş peşe çalıştırılabilir. Seçilebilir kısımların sonunda örtülü bir sapma yok. Açıkça "break" komutu ile "switch" sonuna gidilebilir
4. **default** terimi tanımlanmamış kontrol ifadesi değerleri içindir; eğer default olmazsa uygun kontrol değeri çalışmaması durumunda switch bir şey yapmaz.

```
switch (indeks) {  
    case 1:  
    case 3: tek +=1;  
        toplamtek += indeks;  
    case 2:  
    case 4: cift += 1;  
        toplamcift += indeks;  
    default: printf("switch içinde hata, indeks = %d\n",  
                    indeks);  
}
```

```
switch (indeks) {  
    case 1:  
    case 3: tek +=1;  
        toplamtek += indeks;  
        break;  
    case 2:  
    case 4: cift += 1;  
        toplamcift += indeks;  
        break;  
    default: printf("switch içinde hata, indeks = %d\n",  
                    indeks);  
}
```

Çoklu seçme komutları

3. C# Tasarım seçimleri: (**switch** için)

1. C gibidir, sadece birden çok kısmın yürütülmesine izin vermez.
2. Her kısmın mutlaka "break" veya "goto" ile sonlandırılması gereklidir.

```
switch (indeks) {  
    case 1: goto case 3;  
    case 3: tek +=1;  
        toplamtek += indeks;  
        break;  
    case 2: goto case 4;  
    case 4: cift += 1;  
        toplamcift += indeks;  
        break;  
    default: Console.WriteLine("switch içinde hata, indeks = %d\n",  
                               indeks);  
}
```

Çoklu seçme komutları

4. Ada'nın **case**'i Pascal'ın **case** 'ine çok benzer, şu farkla ki:

1. Sayısal değişkenlere (Integer, Boolean, Char, Enum) ek olarak aşağıdaki tipleri de kabul eder:

- Alt değer kümeleri, örneğin: 10..15.
- Mantıksal OR işleçleri,
örneğin: 1..5 | 7 | 15..20.

2. Seçme ifadesi sabitlerinin tamamen kapsanması gereklidir:

- Genellikle **others** terimi ile sağlanır.
- Bu komutu güvenilir yapar.

3. Seçilen komut grubu bitince case komutunu takip eden komut ile program devam eder. "break" veya "goto" kullanılmaz.

```
case <ifade> is
    when SabitListe_1 => Komut_1 grubu;
    ...
    when SabitListe_n => Komut_n grubu;
    [when others => Komut_diger grubu;]
end case;
```

Çoklu seçme komutları

5. PHP Tasarım seçimleri: (**switch** için)

1. C gibidir, sadece indeks ifadesi "string" "integer" veya "double" olabilir..
2. Yürütülmesi sırasında indeks değerine tam uyan case sabiti aranır.
Hiçbiri uymazsa switch bir şey yapmadan bir sonraki komuta devreder..

```
switch (indeks) {  
    case 1: goto case 3;  
    case 3: tek +=1;  
        toplamtek += indeks;  
        break;  
    case 2: goto case 4;  
    case 4: cift += 1;  
        toplamcift += indeks;  
        break;  
    default: printf("switch içinde hata, indeks = %d\n", indeks);  
}  
}
```

Çoklu seçme komutları

- Birçok durumda switch veya case komutları yetersizdir:
 - Örneğin basit bir sabit yerine mantıksal bir ifade gerekiğinde.
- Çok seçimeler iki yolu seçme komutlarının genişletilmesi ile de elde edilebilir:(ALGOL 68, FORTRAN 90, Ada)

Ada

```
if ...  
  then ...;  
elsif ...  
  then ...;  
else ...;  
end if;
```

Ada

```
if ...  
  then ...;  
else  
  if ...  
    then ...;  
  else ...;  
  end if;  
end if;
```

C

```
if (...)  
...;  
else if (...)  
...;  
else ...;
```

Çoklu seçme komutları

■ Ada'nın çoklu seçme komutu:

- İç içe geçmiş if'lerden çok daha kolay okunabilir.
- Her aşamada mantıksal değerlendirme yapılmasını sağlar.



Part 1: Linguistics and the Birth of Perl

LARRY WALL

Larry Wall wears many hats—he is involved in book publishing, language publishing, software publishing, child rearing (he has four kids). He has spent time studying at Seattle Pacific University, at Berkeley, and at UCLA. He has also worked at Unisys, Jet Propulsion Laboratories, and Seagate. Language publishing has brought him the most fame ("and the least money" he adds): Larry is the author of the Perl scripting language.

FIRST, A BIT ABOUT YOUR PROFESSIONAL BACKGROUND

What was your best job? Working at JPL [Jet Propulsion Laboratories] was a lot of fun. I did both system administration and software development there. The sysadmin job was particularly nice because it was one of those 90 percent boredom, 10 percent panic jobs, so I had plenty of time to work on Perl.

What was your weirdest job? That's a hard one. Most of my jobs turn out weird, one way or another, so it's hard to pick. Let's see . . . I've been a camp counselor called "Crazy Horse." I've written business software in BASIC on a Wang system that could be manipulated only via menus. I was the concertmaster for several years at MusiComedy Northwest. Perhaps my weirdest gig is the recording session at which I was the only one who could play the violin part, so they recorded me eight or ten times so I'd sound like the whole violin section.

What's your current job? My current job is to design Perl 6. Unfortunately nobody is paying me for that at the moment. Of course, nobody was officially paying me for the first five versions either. But it would be nice if someone were at least unofficially paying me for it this time. Perhaps by the time you read this someone will have hired me to be a walter/actor in Hollywood.

LINGUISTICS AND COMPUTING LANGUAGES

You studied linguistics. At the time you were engaged in these studies, what career paths were you considering? My wife and I were going to be field linguists in Africa and were for a time associated with

Wycliffe Bible Translators, until my (newly developed) food allergies put the kibosh on that. In any event, I don't know that I'd have made a very good field linguist—I've probably done linguists more good by staying out of the field and writing Perl. Nevertheless, I still love linguistics and have been teaching myself Japanese for the last few years just to keep the synapses from coagulating, or curdling, or whatever it is they most naturally do.

How did your interest in spoken languages translate to program languages? [Before] I wrote Perl, I have to admit that my linguistics and my computer science were pretty much compartmentalized. On the computer side, I wrote several compilers without asking myself why most computer languages seem so unnatural. On the linguistics side, I wrote a few natural language programs in Lisp, but generally didn't consider natural languages amenable to analysis by computer. (Anyone who has used Babblefish will discover this is still the case.) But it wasn't until I started working on Perl that I realized there are many principles that make a natural language feel natural, and some of those principles can be taught to computers without driving them nuts.

THE BIRTH OF PERL

What were you working on at Unisys in the days leading up to the creation of Perl? I was a system administrator and systems programmer in support of a secret project for the NSA [National Security Agency]. If I told you about it, you'd have to kill me. Er, that seems wrong. . . . Anyway, I spent a lot of time locked in "copper rooms," as they call them, playing with computers that didn't officially exist.

Döngülü Komutlar (Iterative Statements)

- Bir komutun veya bir komut grubunun tekrarlanan yürütülmesi döngü veya özyineleme ile elde edilir.
- Biz burada döngüyü inceleyeceğiz çünkü özyineleme daha çok fonksiyonel programlama dilleri tarafından kullanılan ve o kapsamda inceleyeceğimiz bir metot.
- Döngü Plankalkül'den beri bütün dillerde var.
- Döngünün komutunun ne şekilde olacağı iki temel sorunun cevabına göre belirlenir:
 1. Döngü kontrolü nasıl yapılacak? Mantıksal, sayarak veya ikisi birden.
 2. Döngü mantıksal ifadesi nerede olacak? Başlangıkta mı, sonda mı, programcı mı karar verecek?
 - Bu sorunun nedeni fiziksel olarak kontrol ifadesine yer aramak değil, mantıksal ifadenin döngünün komutlar kısmından önce mi yoksa sonra mı test edileceğile ilgilidir.
- Döngü ifadeleri kimi zaman makine koduyla da desteklenmiştir. Ancak bu yaniltıcı da olabilmiştir. Örneğin bir zamanlar Fortran sonda testli döngü içerdiginden vax makinelere böyle makine kodu eklenmiş, sonradan Fortran testi başa alınınca bu makine kodu boşta kalmıştır. IBM 704 ve aritmetik if ilişkisi de benzer bir durumdur.

Döngülü Komutlar

1. Sayıcı kontrollü döngüler

■ Tasarım hususları:

1. Döngü değişkeninin tipi ve kapsamı nedir?
2. Döngü değişkeninin döngü bittiğinde değeri nedir?
3. Döngü değişkeninin değeri döngü içinde değiştirilebilmeli midir?
Değiştirilebilirse bu döngü kontrolünü etkilemeli midir?
4. Döngü parametreleri bir kez mi değerlendirilmelidir yoksa her döngüde mi?

Döngülü Komutlar

1. FORTRAN 95

- Sözdizim (Syntax): DO etiket **var** = **baslangic**, **bitis** [, **adim**]
 - değişken
 - parametreler
- adim 0'dan farklı tam sayı
- parametreler ifadeler olabilir.
- Tasarım hususları:
 1. Değişken **INTEGER** olmalıdır.
 2. Değişken sonda bitis değerini alır.
 3. Değişken döngü içinde değiştirilemez ancak parametreler değiştirilebilir. Bunun nedeni parametrelerin başda bir kez değerlendirilmesidir. Döngü başladıkтан sonra değiştirilmeleri döngüyü etkilemez.
- Diğer DO
 - Sözdizim:
[isim:] DO var = baslangic, bitis [, adim]
...
END DO [isim]
 - Döngü değişkeni **INTEGER** olmalıdır.

Döngülü Komutlar

2. Pascal

■ sözdizim:

for var := başlangıç (**to** | **downto**) son **do** <komut>

■ Tasarım hususları:

1. Döngü değişkeni bir sayı tipi (ordinal type).
2. Döngü bittikten sonra döngü değişkeni tanımsız.
3. Değişken döngü içinde değiştirilemez ancak parametreler değiştirilebilir. Bunun nedeni parametrelerin başda bir kez değerlendirilmesidir. Döngü başladıkтан sonra değiştirilmeleri döngüyü etkilemez.
4. Parametreler bir kez başta hesaplanır.
5. Döngüye sadece başından girilebilir.

Döngülü Komutlar

4. Ada

■ Sözdizim:

```
for var in [reverse] deger_kumesi loop
```

...

```
end loop
```

■ Ada döngü tasarıımı:

1. Döngü değişkeni var'ın tipi deger_kumesi'nin tipidir. Döngü içinde tanımlıdır (örtülü tanımlama).
2. Döngü değişkeni döngü dışında tanımlı değil.
3. Döngü değişkeni döngü içinde değiştirilemez fakat deger_kumesi değiştirilebilir; bu döngüyü etkilemez.
4. Deger_kumesi bir kez hesaplanır.
5. Döngüye sadece başından girilebilir.

```
Say : Float := 1.35;  
for Say in 1..10 loop  
    toplam=toplam+Say;  
end loop;
```

Döngülü Komutlar

5. C

■ Sözdizim:

for ([ifade_1] ; [ifade_2] ; [ifade_3]) komut

- ifade'ler tek olabilecekleri gibi, virgül ile ayrılmış birden çok ifade de olabilirler.
- Birden çok ifadeli kısımlarda değer, son ifadenin değeridir.

örneğin,

for (*i* = 0, *j* = 10; *j* != *i*; *j*--, *i*++) ...

- İkinci ifade yoksa döngünün sonu yoktur.

Döngülü Komutlar

■ C tasarım seçimleri:

1. Açık bir döngü değişkeni yoktur.
2. Döngü değişkeni olmadığından tanımlı olduğu bölge de tanımlı değil.
3. Döngü içinde herşey değiştirilebilir.
4. İlk ifade başta bir kez hesaplanır ancak diğerleri her döngüde hesaplanırlar.

■ Döngü komutu daha esnek, buna karşılık daha çok hataya açık.

Döngülü Komutlar

6. C++

- C'den iki şekilde ayrılır:
 1. Kontrol ifadesi mantıksal da olabilir.
 2. Başlangıç ifadesi döngü içinde geçerli değişken tanımlaması da içerebilir.

7. Java, C#

- C++'dan farkı kontrol ifadesi mantıksal olmak zorunda. Java döngüye ortadan girişe izin vermezken C# verir.

Döngülü Komutlar

2. Mantıksal kontrollü döngüler

■ Tasarımla ilgili hususlar:

1. Test başta mı olacak yoksa sonda mı?
2. Sayıcı kontrollü döngülerin özel hali mi olacak yoksa ayrı mı olacak?

Döngülü Komutlar

■ Dil örnekleri:

1. Pascal iki türlü başta ve sonda kontrollü mantıksal döngüsü var (**while-do** ve **repeat-until**).
2. C ve C++'da da iki tür var (**while - do** ve **do - while**).
- 3 Java, C gibi; sadece kontrol ifadesi mantıksal olmak zorunda ve döngüye baştan girilebiliyor (goto komutu olmadığından).
4. Ada'da sadece başta kontrol var.
5. FORTRAN 77 ve 90'da mantıksal döngü yok.
6. Perl'ün iki baştan kontrollü döngüsü, iki de sondan kontrollü döngüsü var. Hepsinde kontrol kısmı **while** veya **until** ile oluşturuluyor.

do { } until (...);	do { } while (...);	while (...) { };	until (...) { };
-------------------------------------	-------------------------------------	----------------------------------	----------------------------------

Döngülü Komutlar

3. Kullanıcı tarafından yerleştirilen döngü kontrol düzenekleri:

1. Koşul ve döngüden çıkış tek bir kısım mı olmalı?
2. Kontrol birden çok döngüden dışarı çıkabilmeli mi?

Kullanıcı tarafından yerleştirilen döngü kontrol düzenekleri

Örnekler:

1. Ada – koşullu ve koşulsuz; her türlü döngü için; her seviyede

```
for ... loop          LOOP1:  
    ...                  while ... loop  
    exit when ...        ...  
    ...                  LOOP2:  
    end loop             for ... loop  
    ...  
    exit LOOP1 when ..  
    ...  
    end loop LOOP2;  
    ...  
end loop LOOP1;
```

Kullanıcı tarafından yerleştirilen döngü kontrol düzenekleri

2. C , C++, Java, Perl ve C#

- C , C++: koşulsuz, etiketsiz bir kademe çıkış **break**.
- Java, C#: bir öncekine ilaveten, koşulsuz etiketli birkaç kademeli çıkış **break**.
- Perl: koşulsuz etiketli birkaç kademeli çıkış **last**.
- Bütün bu dillerde ayrıca, döngüyü bitirmeyen, ancak kontrol kısmına gönderen, **break** ile aynı özelliklerde **continue**.
- Java ve Perl de **continue** komutlarının etiketi de olabilir.

C# örneği:

```
dongul:  
for(satir = 0; satir<satirsayi; satir++)  
    for(sutun = 0; sutun<sutunsayi; sutun++) {  
        toplam += mat[satir] [sutun];  
        if(toplam > 1000.0) break dongul;  
    }
```

C örneği

```
while (toplam < 1000) {  
    sonraki(deger);  
    if (deger < 0) continue;  
    toplam += deger;  
}
```

Kullanıcı tarafından yerleştirilen döngü kontrol düzenekleri

3. FORTRAN 90 - **EXIT**

- Koşulsuz, her türlü döngü için, her seviyede.
- Ayrıca C'deki **continue** ile aynı anlama gelen **CYCLE**.

Döngülü Komutlar

■ Veri yapılarına dayalı döngüler

- Kavram: bir veri yapısını (data structure) ve sırasını döngünün kontrolü için kullanmak.
- Kontrol mekanizması: varsa veri yapısının bir sonraki elemanını dönen bir fonksiyon, yoksa döngü biter.
- C'de **for** bu amaçla kullanılabilir:
 - örneğin: **for (p=hdr; p; p=sonraki(p)) { ... }**

□ Perl bu tip durumlar için özel bir döngü kullanır:

```
foreach $name (@names) { print $name }
```

□ PHP örneği: reset, current, prev ve next fonksiyonları.

```
reset $list; // current göstericisini başa alır.  
print ("ilk sayı: " + current($list) + "<br />");  
while ($current_value = next ($list))  
    print ("sonraki sayı: " + $current_value + "<br />");
```

Veri yapılarına dayalı döngüler

■ Örnekler devam...

- Java: Iterator metotları kullanılarak (next, hasNext ve remove) yapılabileceği gibi, "foreach" gibi davranışlı "for" kullanılarak veri yapısına dayalı döngü yapılabilir. Örneğin önceden tanımlanmış "Listemiz" isimli bir "String" dizilimimiz (array) varsa:

```
for (String Eleman : Listemiz) { . . . . }
```

- C#

```
String[] Listemiz = { "Ankara", "İstanbul", "Denizli" };  
. . . .  
foreach (String Eleman in Listemiz)  
    Console.WriteLine("Şehir: {0} ", Eleman);
```

{0} yazılan yere "Eleman"ın değeri yazılır.

Koşulsuz sapma ("goto")

- GÜCÜ: Programın akışı değiştirilebilir, her türlü kontrol komutu "goto" ve seçici ile yapılabilir. Çok etkili bir komut.
- SORUN: Okunabilirlik ve bağlı olarak oluşan güvenlik sorunları, desteklenmesinin zorlukları.
- Bazı dillerde yoktur: örneğin Java.
- Döngü çıkış komutları (**break**, **last**) kamufla edilmiş **goto**'lardır. Ancak çıkış döngünün hemen sonuna olduğundan okunabilirliği ve güvenliği bozmazlar.
- Dijkstra (1968): "the goto statement as it stands is just too primitive; is too much an invitation to make a mess of one's program".

Koşulsuz sapma

■ Etiket şekilleri:

1. İşaretsiz tam sayı sabitler: Pascal (üstüste nokta ile)
FORTRAN (üstüste noktasız)
2. Üstüste nokta ile değişkenler: ALGOL 60, C
3. Değişkenler << ... >> içinde: Ada
4. Değişkenler etiket: PL/I
 - Bu değişkenlere değer atanabilir ve parametre olarak alt programlara geçirilebilirler.
 - Çok esnek yapar ama aynı ölçüde de okunamaz ve gerçekleştirilemez yapar.

Önlemli Komutlar (Guarded Commands)

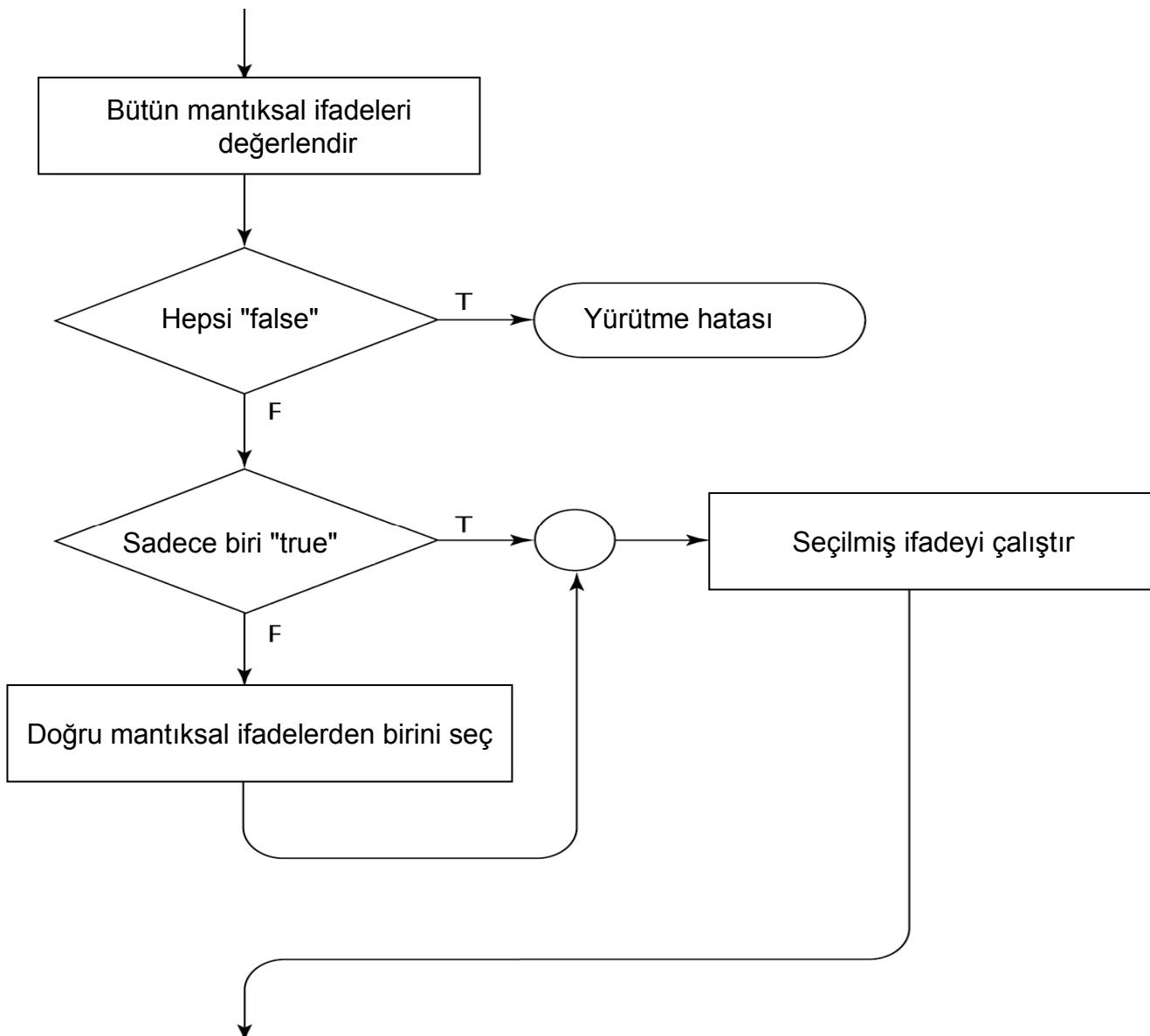
- Edsger Wybe Dijkstra, 1975: Yeni ve farklı seçme ve döngü yapıları önermiştir.
- Amaç: yeni bir programlama metodolojisini desteklemek – program geliştirmesi sırasında doğrulama.
- Paralel programlama için önemli.

Önlemli Komutlar

1. Seçme: **if** <boolean> -> <statement>
 [] <boolean> -> <statement>
 . . .
 [] <boolean> -> <statement>
 fi

- Anlamı: Bu yapıya erişildiğinde,
 - Bütün boolean ifadeleri değerlendirir;
 - Eğer birden fazlası doğruysa birini belirleme imci olmadan (nondeterministically) seç;
 - Eğer hiçbirini doğru değilse, yürütme hatası ver.
- Düşünce: Eğer değerlendirme sırası önemli değilse, program böyle bir seçim yapmamalıdır.

Önlemli Komutlar



Önlemli Komutlar

2. Loops **do** <boolean> -> <statement>
 [] <boolean> -> <statement>
 . . .
 [] <boolean> -> <statement>
 od

■ Anlamı: Her döngüde,

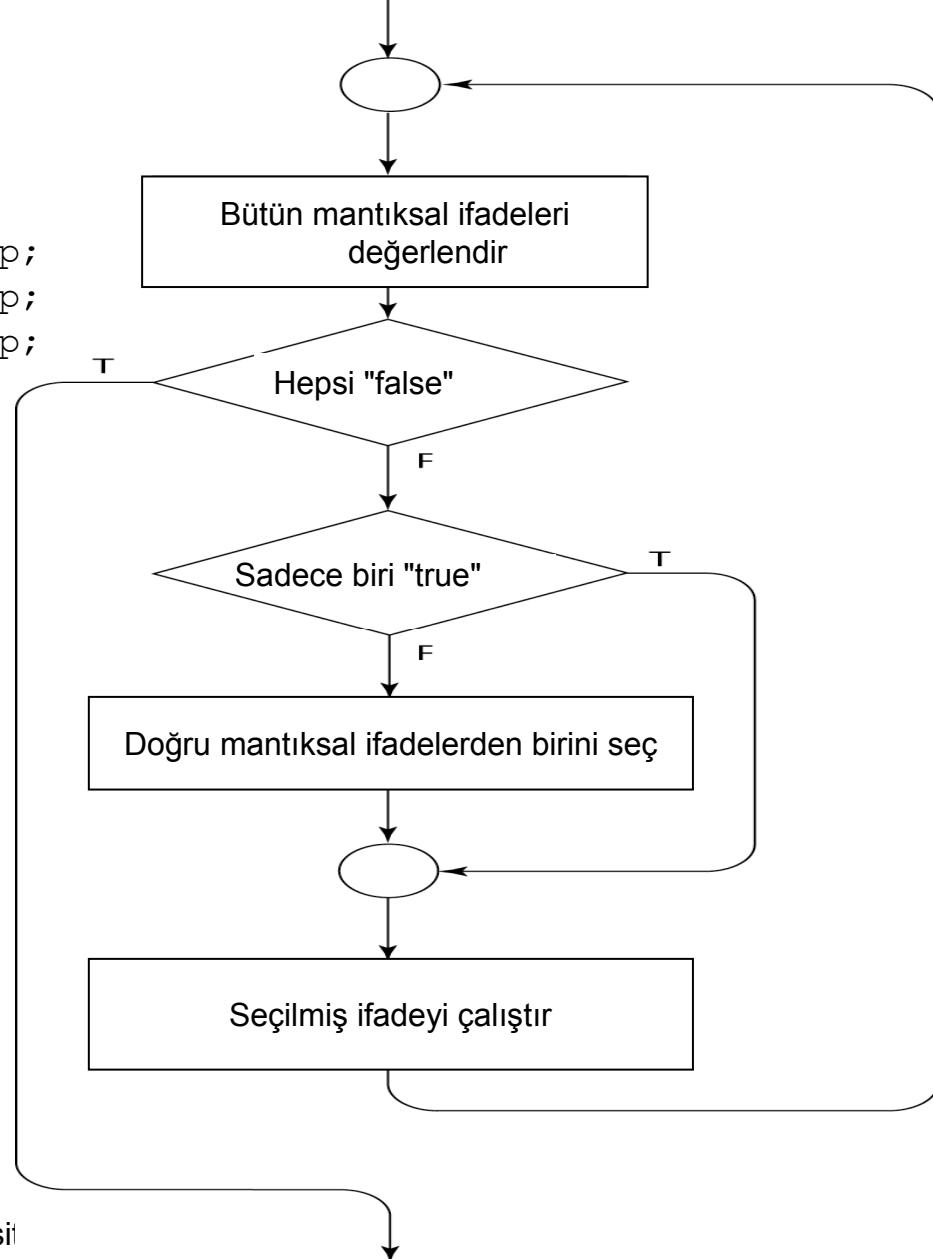
- Bütün boolean ifadeleri değerlendir;
- Eğer birden fazlası doğruysa birini belirlenimci olmadan (nondeterministically) seç;
- Eğer hiçbirini doğru değilse, döngüden çık.

■ Düşünce: Eğer değerlendirme sırası önemli değilse, program böyle bir seçim yapmamalıdır.

Önlemlİ Komutlar -- döngü

$q_1 \leq q_2 \leq q_3 \leq q_4$ olmasını sağlayan program.

```
do  q1>q2 -> temp:= q1; q1 := q2; q2 := temp;  
[] q2>q3 -> temp:= q2; q2 := q3; q3 := temp;  
[] q3>q4 -> temp:= q3; q3 := q4; q4 := temp;  
od
```



Önlemli Komutlar

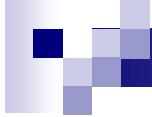
- Kontrol komutları ve program doğrulaması (verification) arasında güçlü bir etkileşme vardır.
- goto kullanılan bir programda doğrulama çok zorlaşır.
- Doğrulama seçimler ve mantıksal döngüler kullanılırsa yapılabilir.
- Doğrulama "önlemli komutlarla" görece olarak daha basittir.

Son söz

- Bu kısımda bahsettiğimiz seçme ve ön kontrollü döngüler dışındaki diğer kontrol komutları dilin büyülüğu ile kolay yazılabılırlik arasındaki tercih sorunudur.
- Fonksiyonel ve mantıksal dillerdeki kontrol yapıları bu kısımda bahsettiğimiz yapılardan farklıdır.

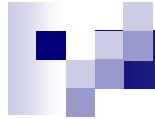
9. Bölüm

Altprogramlar (Subprograms)



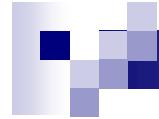
9. Bölüm Başlıkları

- Başlangıç
- Altprogramların esasları
- Altprogramların tasarım hususları
- Lokal referans ortamları (Local Referencing Environments)
- Parametre geçirme metodları (Parameter-Passing Methods)
- Altprogram adı olan parametreler (Parameters that are Subprogram Names)
- Fazla yüklü altprogramlar (Overloaded Subprograms)
- Cinsine özgü altprogramlar (Generic Subprograms)
- Fonksiyonlar için tasarım hususları (Design Issues for Functions)
- Kullanıcı tanımlı fazla yüklü işaretçiler (User-Defined Overloaded Operators)
- Eşyordamlar (coroutine): çağrıran programa dönmesi gerekmeyen ve işlemin durdurulduğu yerden tekrar yürütülmü başlatan program



Başlangıç

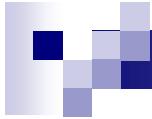
- Bir programda birden çok kez etkin duruma getirilebilecek bir dizi deyimin bir isim altında grüplanması ile altprogramlar oluşturulur. Böylece tek bir isim, bir dizi deyimi göstermekte ve bir soyutlama gerçekleştirmektedir.
- Altprogramlar ile işlev soyutlaması amaçlanır.
- İki temel soyutlama (abstraction) kolaylığı
 - İşlev soyutlaması (Process abstraction) – bu kısımda
 - Veri soyutlaması (Data abstraction) – daha sonra.



Altprogramların esasları (Fundamentals of Subprograms)

■ Altprogramların genel özellikleri:

1. Altprogramın bir tane giriş yeri olur.
2. Altprogram çağrırlığından ve yürütülürken çağrıran program bekler.
3. Çağırılan program bitince kontrol her zaman çağrıran programa geri döner.



Temel tanımlamalar

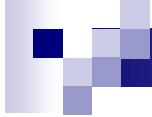
- Altprogram tanımı: Altprogram soyutlamasının betimlenmesi.
- Altprogram çağrıması (**subprogram call**): altprogramın yürütülmesinin talep edilmesi.
- Altprogram başlığı (**subprogram header**): adı, parametreleri ve ne tip bir altprogram olduğuna dair bilgiler.
- Parametre profili: Altprogramın parametrelerinin sayısı, sırası ve tipleri.
- Altprogram protokolü: Parametre profili ve eğer fonksiyonsa döndüğü değer tipi.
- Altprogram bildirimi (**subprogram declaration**): protokol belirlenir ancak altprogramın gövdesi belirlenmez.
- Resmi parametre (**formal parameter**): Altprogram başlığında listelenip altprogram içinde kullanılan parametre.
- Gerçek parametre (**actual parameter**): Altprogram çağrırlarken adres veya değeri için yazılan parametreler.

Altprogramların esasları

■ Gerçek/Resmi parametrelerin eşleşmesi:

1. Konumsal (Positional): Gerçek ve resmi parametreler yerleştiriliş sırasına göre eşleşirler.
 - Dezavantaj: parametre sayısı artınca karıştırılabilir.
 - Birçok dilde resmi ve gerçek parametreler eşleşmelidir. Ancak, C, C++, Java ve Perl'de eksiklik durumunda eşleşme programcının sorumluluğundadır. Bu durum belirsiz parametreli fonksiyonlara izin verir.
2. Anahtar kelime: Resmi parametreler anahtar kelime olarak kullanılarak gerçek parametrelerle eşleştirilirler.
 - ADA örneği, `SORT(LIST => A, LENGTH => N)` .
 - Avantaj: sıra önemli değil.
 - Dezavantaj: kullanıcı resmi parametre adını bilmelidir.
 - C++, Fortran 95, Ada ve PHP'de resmi parametrelerin varsayılan değeri olabilir:

```
procedure SORT(LIST : LIST_TYPE;
               LENGTH : INTEGER := 100);
               ...
SORT(LIST => A);
```



Altprogramların esasları

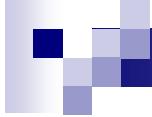
■ Gerçek/Resmi parametrelerin eşleşmesi:

3. Değişken sayılı parametreler

- C#

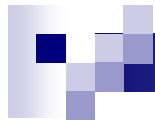
```
public void DisplayList(params int[] list) {  
    ... }  
myObject.DisplayList(2, 4, 6, 8);
```

- Ruby, Python gibi dillerde de bu özellik var.



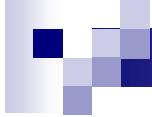
Altprogramların tipleri

- Bir altprogram, bir **yordam** (*Procedure*) veya bir **fonksiyon** (*Function*) tipinde olabilir.
- İki altprogram türü arasındaki fark, fonksiyonların çağrıran program birimine bir sonuç değeri döndürmelerinin şart olmasıdır.
- Fonksiyon altprogramlar bir programlama dilinde varolan işlevlere benzer.
- Yordam altprogramlar ise bir programlama dilinde varolan komutlara benzer olarak düşünülebilir.
- Çoğu popüler *buyurgan* dilde hem fonksiyonlar hem de yordamlar bulunur.
- Örnekler:
 - **Pascal ve Ada'da Yordamlar:**
Yordamlar, Pascal ve Ada'da procedure anahtar kelimesi ile belirtilirler.
 - **C'de Yordamlar:**
 - C'de yordamların belirtilmesi için özel bir anahtar kelime kullanılmaz.
 - C ve C++'da sadece fonksiyonlar yer alırsa da, bu fonksiyonlar yordamların işlevlerini de gerçekleştirebilirler (void fonksiyonlar).
 - **FORTRAN'da** genel özellikleri diğer dillerdeki yordamlara benzer olmakla birlikte, yordamlara SUBROUTINE ismi verilir.



Altprogramlarda tasarım hususları

1. Hangi parametre geçirme (parameter passing) metodları uygulanacak?
2. Parametrelerin tipleri kontrol edilecek mi?
3. Lokal değişkenler statik mi, dinamik mi olacak??
4. Altprogram tanımlamaları başka altprogramların tanımlamalarında yer alabilecek mi (iç içe tanımlama)?
5. Eğer altprogramlar parametre olarak başka altprogramlara geçirilebilirse ve altprogramlar iç içe tanımlanabilirse bu durumda parametre ile geçen altprogramın referans çevresi (referencing environment) (erişilebilir tüm isimler ve bunların kapsamlarıdır) ne olacak?
6. Altprogramlar fazla yüklü (overloaded) (aynı isimli başka altprogramlar) olabilecek mi?
7. Altprogramların cinsine özgü (generic) (farklı çağrılarda farklı veri tiplerini işleyebilen) olmasına izin verilecek mi?



Lokal Referans Çevresi

■ Eğer lokal değişkenler yığıt dinamikse (stack-dynamic):

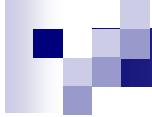
□ Avantaj:

- a. Özyinelemeye destek.
- b. Lokaller için ayrılan alan altprogramlar tarafından paylaşılabilir.

□ Dezavantajları:

- a. Tahsis/geri verme zamanı.
- b. Dolaylı adresleme. Çoğu bilgisayarda yavaş.
- c. Altprogramlar geçmişe hassas değil. Altprogram bitince bütün lokal değerler unutuluyor.

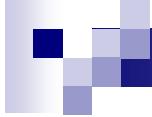
■ Statik lokallerde avantaj ve dezavantajlar yer değiştirir.



Lokal Referans Çevresi

■ Dil Örnekleri:

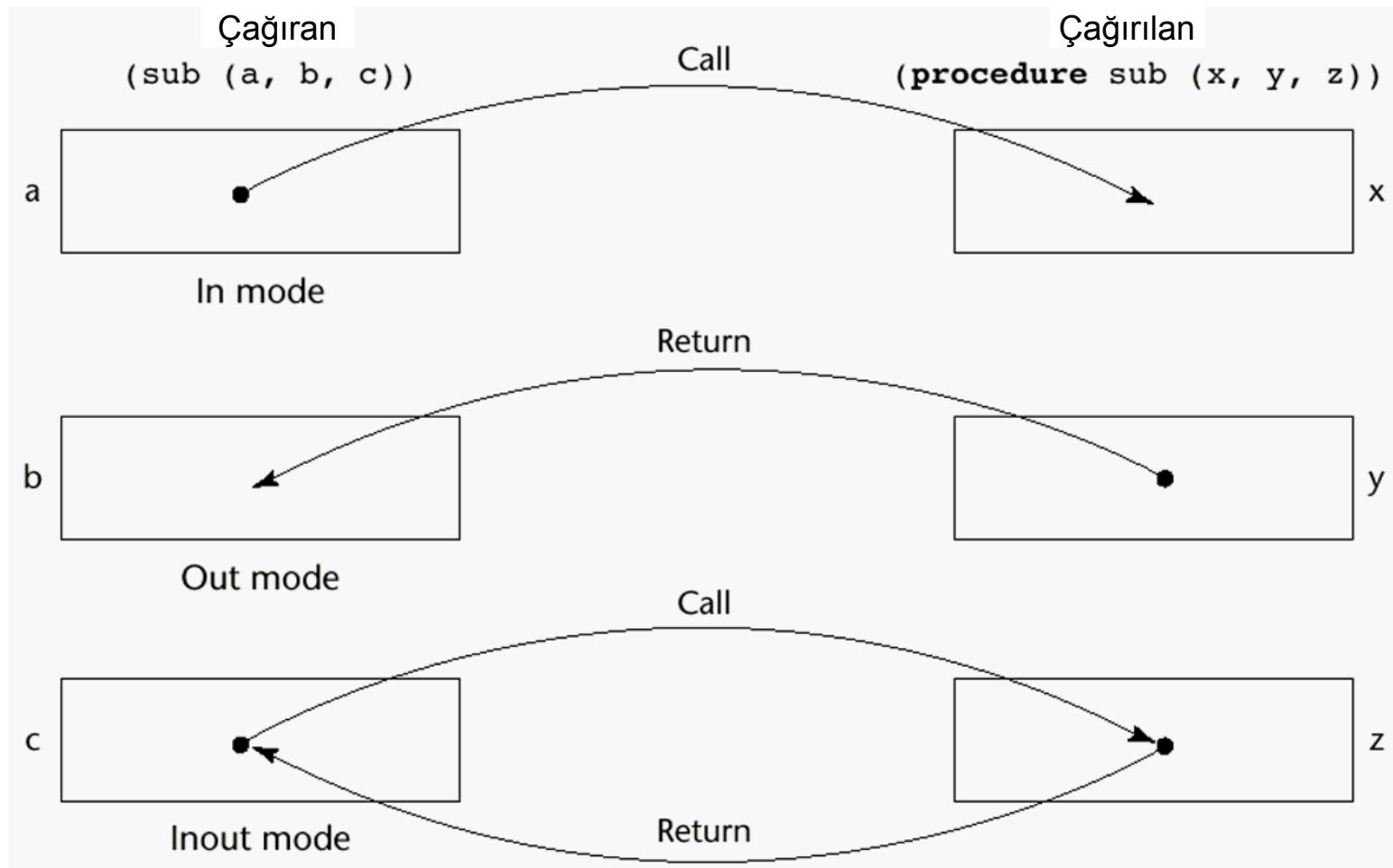
1. FORTRAN 77, 90, 95 – Özyineleme öngörülmediğinden lokal değişkenler statik. Dinamik için "recursive subroutine" tanımlaması gereklidir. Bu durumda "save" tanımlanmış değişkenler yine de statik olur.
2. C – ikiside: **static** tanımlanmış değişkenler statik; varsayılan yığıt dinamik(stack dynamic).
3. Pascal, Java, ve Ada – sadece dinamik.

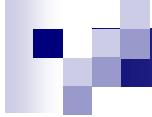


Parametre geçirme metodları

- Bir yordam etkin duruma getirilirken, çağrım deyimindeki gerçek parametreler ile yordamın resmi parametrelerine farklı değerler aktarılabilir.
- Gerçek parametreler yordamlara aktarılacak değerleri gösterdikleri için, değişken,sabit veya ifade olabilir. Resmi parametreler ise bu değerleri tutacak bellek yerlerini gösterdikleri için değişken olmak durumundadır.
- Veri Akışı Modelleri:
 - Bir yordam, resmi parametrelerinin değerini ve/veya resmi parametre olmayan ancak erişebildiği değişkenlerin değerlerini değiştirebilir.
- Bunu farklı seviyelerde tartışacağız:
 - anlamsal modeller: içeri modeli (in mode), dışarı modeli (out mode), içeri-dışarı modeli (inout mode).
 - Kavramsal modeller:
 1. Değerin aktarılması.
 2. Değere erişim yolunun (adresinin) aktarılması.

Parametre geçirme metodları



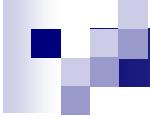


Parametre geçirme metodları

■ Gerçekleştirme metodları:

1. Değer ile çağrıma (Pass-by-value) (in mode)

- Değer ile çağrıma yöntemi, içeri modelinin gerçekleştirmidir. Fiziksel taşıma veya erişim adresinin aktarılması yoluyla olabilir.
 - Fiziksel taşımada resmi parametre, karşı gelen gerçek parametrenin değeriyle ilklendikten sonra, altprograma yerel bir değişken olarak nitelendirilir.
 - Gerçek parametre, sabit, değişken veya ifade olabilir.
 - Bu yöntem, sadece gerçek parametreden resmi parametreye değer geçisi olduğu için en güvenilir parametre aktarım yöntemidir. Ancak, bu değer geçisi sırasında fiziksel olarak veri kopyalanması gerçekleşir. Yani, resmi parametre için de bellek ayrılması gereklidir.
 - Erişim adresinin aktarıldığı durumda kaynak adresin yazma korumalı olması gereklidir.
- Erişim adresi aktarılmasının dezavantajları:
 - Çağırılan altprogramda geçilen adresler yazma korumalı olmalıdır.
 - Erişim zaman alıcı (dolaylı adreslemeden dolayı).
- Fiziksel aktarımın dezavantajları:
 - Daha çok depolama alanı (aktarılanın yeni kopyası gerektiğinden)
 - Taşımanın aldığı zaman (Özellikle geçilen parametre büyükse, dizilim gibi).



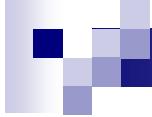
Parametre geçirme metodları

2. Sonuç ile çağrıma (Pass-by-result) (out mode)

- Sonuç ile çağrıma yöntemi, dışarı modelinin gerçekleştirimidir. Bu yöntemde çağrılmış deyimi ile altprograma bir değer aktarılmazken, gerçek bir parametreye karşı gelen resmi parametrenin değeri, altprogram sonunda, denetim yeniden çağrıran programa geçmeden önce, gerçek parametreyi gösteren değişkene aktarılır. Bu tanımlamadan anlaşıldığı gibi, gerçek parametrenin değişken olması zorunludur. Gerçek parametreye karşı gelen resmi parametre, altprogramın çalışması süresince yerel değişkendir.
- Bu yöntemin uygulanmasındaki güçlük, gerçek parametrenin değerinin resmi parametreye aktarılmasının önlenmesidir.
- Dezavantajları:
 - Eğer değer geçilirse zaman ve yer kaybı.
 - Her durumda sıraya bağımlılık sorun olabilir:

```
procedure sub1(y: int, z: int);  
...  
sub1(x, x);
```

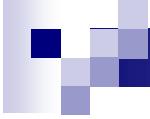
 - **x** 'in değeri dönüşteki atama sırasına bağlı.



Parametre geçirme metodları

3. Değer ve sonuç ile çağrıma yöntemi (Pass-by-value-result) (inout mode)

- Değer ile çağrıma ve sonuç ile çağrıma yöntemlerinin birleşimidir.
- Bu yöntemde, gerçek parametrenin değeri ile karşı gelen resmi parametrenin değeri ilklendikten sonra resmi parametre, altprogramın çalışması süresince yerel değişken gibi davranışır ve altprogram sona erdiğinde resmi parametrenin değeri gerçek parametreye aktarılır. Bu yöntemde de gerçek parametrenin değişken olması zorunludur.
- Bu yöntemin dezavantajları, parametreler için birden çok bellek yeri gerekliliği ve değer kopyalama işlemlerinin zaman almasıdır.



Parametre geçirme metodları

4. Referans ile çağrıma (Pass-by-reference) (pass-by-sharing) (inout mode)

- Referans ile çağrıma yöntemi de gerçek ve resmi parametreler arasında iki yönlü veri aktarımı sağlar. Ancak önceki yöntemlerden en önemli farkı, altprograma verinin adresinin aktarılmasıdır. Bu adres aracılığıyla altprogram, çağrıran program ile aynı bellek yerine erişebilir ve gerçek parametre, çağrıran program ve altprogram arasında ortak olarak kullanılır.
- Yöntemin Avantajı:
 - Referans ile çağrıma yönteminin en önemli üstünlüğü, aktarımın hem yer hem de zaman açısından etkin olmasıdır. Değer ve sonuç ile çağrıma yönteminde olduğu gibi bellek yeri veya kopyalama zamanı gereklidir.
- Yöntemin Dezavantajı:
 - Bu yöntemin dezavantajı, resmi parametrelere erişim için verilerin aktarıldığı yöntemlere göre fazladan bir dolaylı erişim gerektirmesidir.
 - Ayrıca eğer sadece çağrıran programdan altprograma değer aktarılması isteniyorsa, bu yöntemde gerçek parametrenin bellekteki yerine altprogram tarafından ulaşılabilirliği için, değerinde istenmeyen değişiklikler yapılabilir.

Parametre geçirme metodları

- Referans ile çağrıma (Pass-by-reference) dezavantajlar(devam)
 - örtüşmelere neden olabilir, okunabilirliği ve anlaşılabilirliği çok azaltır:
 - i. gerçek parametre örtüşmesi:

```
void sub1(int &a, int &b);  
...  
sub1(x, x);
```

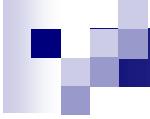
- ii. Dizilim elemanlarının çarpışması:

```
sub1(a[i], a[j]); /* eğer i = j */  
sub1(a, a[i]);
```

- iii. global değişkenlerle resmi parametrelerin çarpışması

- temel neden: çağrılan program lokal olmayanlara daha geniş kullanım hakkı vermiştir.
- Örnek:

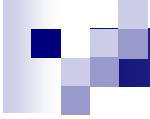
```
int * global;  
void main(){  
    extern int *global;  
    ...  
    sub(global);  
    ...  
}  
  
void sub(int *param){  
    extern int *global;  
    ...  
}
```



Parametre geçirme metodları

5. İsim ile çağrıma yöntemi (Pass-by-name) (inout mode)

- İsim ile çağrıma yöntemi de bir içeri-dışarı modeli için gerçekleştirildir.
- Resmi parametreler çağrıma yöntemine altprogram çağrııldığı zaman bağlanır, fakat parametrelerin gerçek parametrelerle eşleşmesi o resmi değişkene referans veya atama olduğu zaman oluşur.
- Eğer gerçek parametre bir sabit değerse, isim ile çağrıma yöntemi, değer ile çağrıma yöntemi ile aynı şekilde gerçekleşir.
- Eğer gerçek parametre bir değişkense, isim ile çağrıma yöntemi referans ile çağrıma yöntemi ile aynı şekilde gerçekleşir.
- Eğer gerçek parametre bir dizilim elemanı ise çağrıma yöntemi hiçbir yönteme benzemez.
- İsim ile çağrıma yönteminin gerçekleştirilmesi güçtür ve kullanıldığı programların hem yazılmasını hem de okunmasını karmaşıklaştırabilir. Bu nedenle ALGOL 60 ile tanıtılan isim ile çağrıma yöntemi, günümüzde popüler olan programlama dillerinde uygulanmamaktadır.
- Amaç: geç bağlamanın esnekliği.



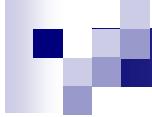
Parametre geçirme metodları

5. İsim ile çağrıma yöntemi (Pass-by-name) (devam)



- Gerçek parametreler resmi parametrelerin yerlerine yerleştirilmiş gibi davranış gösterir.
- gerçek parametre bir değişkense: referans ile çağrıma (pass-by-reference)
- gerçek parametre sabitse: değer ile çağrıma (pass-by-value)
- Eğer gerçek parametre bir dizilim elemanı ise çağrıma yöntemi hiçbir yönteme benzemez.

```
procedure sub1(x: int; y: int);  
begin  
    x := 1;  
    y := 2;  
    x := 2;  
    y := 3;  
end;  
sub1(i, a[i]);
```



Parametre geçirme metodları

■ İsimle çağrımanın dezavantajları:

- Referanslar çok yavaş.
- Anlaması çok zor.

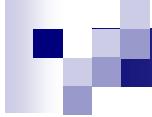
■ Dil örnekleri:

1. FORTRAN

- 77 öncesi, referans ile çağrıma
- 77 – değer ve referansla çağrıma (inout model).

2. ALGOL 60

- isimle çağrıma varsayılan; değerle çağrıma opsiyon.



Parametre geçirme metodları

3. ALGOL W

- ❑ değer ve sonuçla çağrıma

4. C

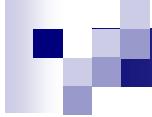
- ❑ değer ile çağrıma

5. Pascal ve Modula-2

- ❑ Varsayılan değer ile; referans ile çağrıma isteğe bağlı.

6. C++

- ❑ C gibi, ama referans tip parametrelere izin verir.



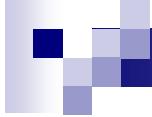
Parametre geçirme metodları

7. Ada

- anlamsal modellerin üçünü de destekler (içeri, dışı, içeri-dışı modeller).
- Eğer dışarıysa, referans verilemez.
- Eğer içeriyse, değer atanamaz.

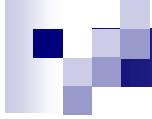
8. Java

- C++ gibi sadece referans tipi parametreleri kabul eder.



Parametre geçirme metodları

- Parametrelerde tip kontrolü (Şimdilerde güvenilirlik için önemli)
 - FORTRAN 77 ve orijinal C: yok.
 - Pascal, FORTRAN 90, Java, ve Ada: Her zaman var.
 - ANSI C and C++: kullanıcı karar verir.



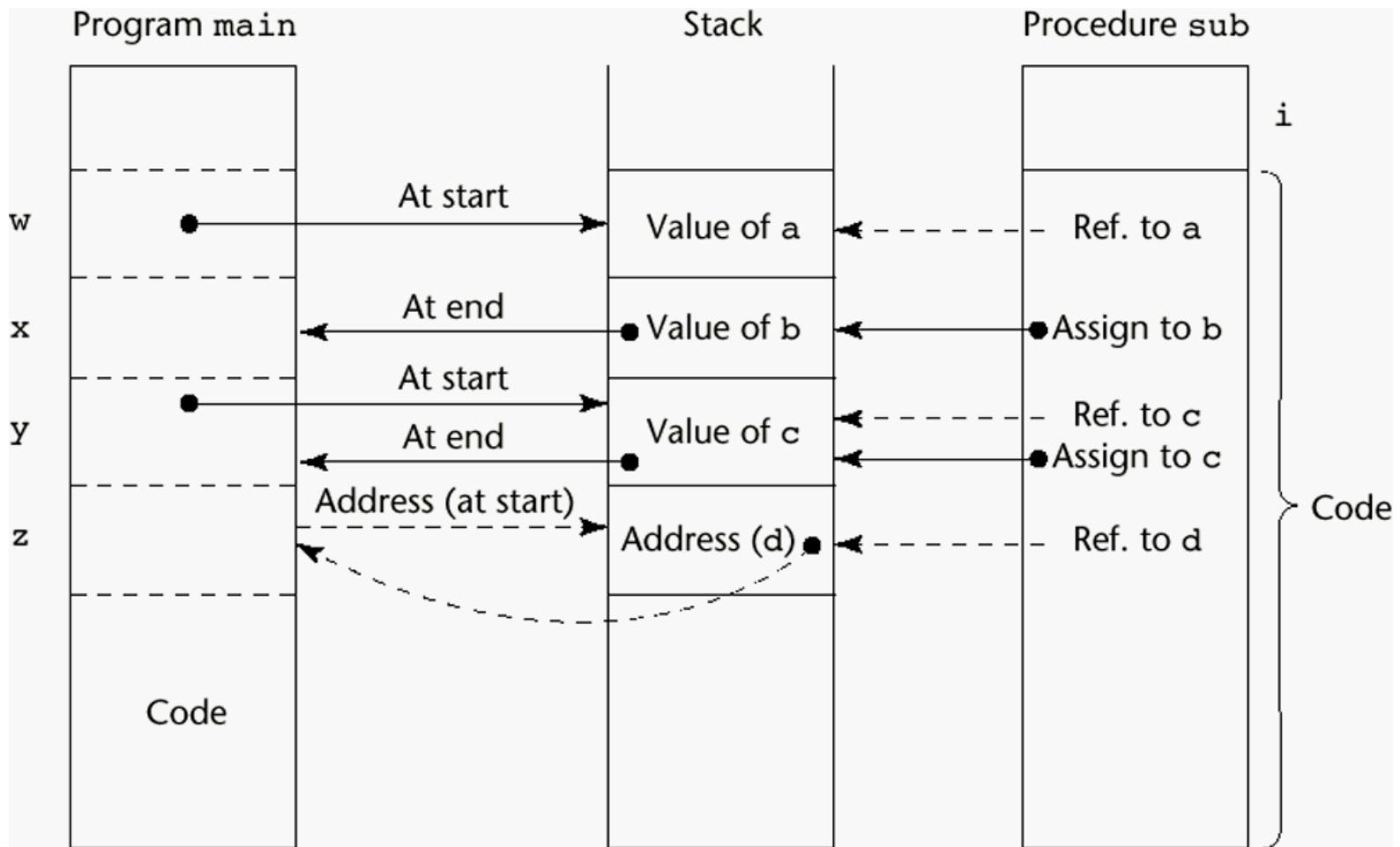
Parametre geçirme metodları

■ Parametre geçirmenin gerçekleştirimi

- ALGOL 60 ve onun takipçileri yürütme zamanı yığıt bellek kullanır.
- değer geçirme – değeri yığıt belleğe kopyala.
- sonuç – sonucu çağrıran programdan alınmak üzere yığıt belleğe kopyala.
- referans – parametrelerin tiplerine bakmadan adresi yığıt belleğe kopyala.

Yığıt bellek üzerinden parametre geçirme metodları

call sub(w, x, y, z): w değeri, x sonucu, y değeri ve sonucu, z referansı üzerinden.
a, b, c, d: resmi parametreler, w, x, y, z gerçek parametreler.



Parametre geçirme metodları

1. Fortran

- 77'den önce, referans ile çağrıma
- 95 – sayılar değer sonuç çağrımasıyla.

2. C

- Değer ile.
- Eğer gösterici geçiliyorsa referans ile.

3. C++

- C gibi, fakat referans tip parametrelere de izin verir. Bu durum örtülü çözmenin (dereferencing) verimliliğini sağlar.

Örnek:

```
void fun (const int &p1, int p2, int &p3) { . . . }
```

p1: referans ile çağrıma ama fonksiyon içinde değiştirilemez;

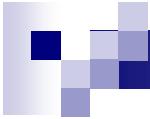
p2: değer ile çağrıma;

p3: referans ile çağrıma.

Ne p1 ne de p3'ün açıkça çözülmesi gerekmektedir. Doğrudan isimleri ile kullanılabilirler.

4. Java

- Değer ile
- Tüm nesnelere referans tiplerine göre erişilir ve referans ile erişme kullanılır.



Parametre geçirme metodları

5. C#

- Varsayılan metot değer ile geçirme.
- Nesneler Java'daki gibi.
- Nesne olmayanların referans ile geçirilmeleri resmi parametrelerde "ref" ile belirlenir.
- Nesne olmayanların dışarı modeli resmi parametrede "out" ile belirlenir.

Örnek:

```
void sumer(ref int oldsum, int newone) { . . . }
```

. . .

```
Sumer(ref sum, newvalue);
```

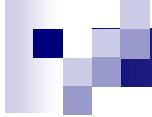
- İlk parametre referansla, ikinci değerle çağrılır.

6. PHP

- C# gibi, sadece referans tipi resmi ve gerçek parametrelerde olabileceği gibi her ikisinde de olabilir. Parametrenin başına & konularak gösterilir.

7. Perl

- Parametreler @_ dizilim değişkeni ile fonksiyona geçirilir. Buradaki gerçek parametreler altprogramda değişirse, dışında da değişir.
- Gerçek ve resmi parametre sayının tutması gerekmekz.



Parametre geçirme metodları

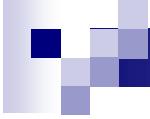
8.Ada

- Basit değişkenler değer-sonuç olarak geçilir (value-result)
- Yapısal tipler (Structured types) referans olarak geçilir.
- Bütün anlamsal modeller var: içeri modeli (in mode), dışarı modeli (out mode), içeri-dışarı modeli (inout mode).

□ Örnek:

```
procedure Adder( A: in out integer;  
                 B: in integer;  
                 C: out Float)
```

- Ada hariç bütün programlama dillerinde içeri (in) modelindeki değişkenler altprogram içinde kullanılabilir ancak bu değerler gerçek parametreye yansıtılmaz. Ada'da kullanılamaz.



Parametre geçirme metodları

■ Parametrelerin tip kontrolü

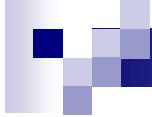
- FORTRAN 77 ve original C: yok
- Perl, JavaScript, PHP, Python, ve Ruby hariç, birçok çağdaş dilde var.
- C örneği:

```
double sin(x)
double x;
{ . . . }
kontrol yok
```

```
double sin(double x)
{ . . . }
kontrol var
```

```
void main()
{ int count;

deger = sin(count);
```



Parametre geçirme metodları

■ Parametre olarak çok boyutlu dizilimler (arrays)

- Eğer çok boyutlu bir dizilik bir altprograma parametre olarak geçirilirse, derleyici doğru adresleme yapabilmek için dizilikin tanımlama boyutlarını bilmek ister.

Parametre olarak çok boyutlu dizimler (arrays)

■ C ve C++

- Programcı birinci boyu hariç diğer boyutların boyalarını resmi parametrede vermek zorundadır.

```
□ adres(mat[i,j]) = adres(mat[0,0]) + i * sutun_sayisi + j
```

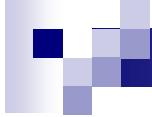
```
void fun(int matrix[][][10]) { . . . }
```

```
void main() {
    int mat[5][10];
    .
    .
    fun(mat);
    .
}
}
```

- Bu durum esnek program yazmayı engeller. Her boyutta matrisle çalışabilecek bir altprogram yazılamaz.

- Çözüm: matrisin işaretçisini parametre olarak geç ve bilgi olarak da boyutları ayrı parametreler olarak geç. Programcı adres hesaplayan fonksiyonu da sağlamalıdır.

```
void fun(float *mat_ptr, int num_rows, int num_cols);
#define mat_ptr(i,j) = (* (mat_ptr + ((i) * num_cols) + (j)))
```



Parametre olarak çok boyutlu dizimler (arrays)

■ Pascal

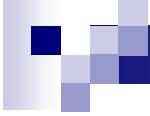
- Problem değil, dizimin boyu dizimin parçası.

■ Ada

- Kısıtlanmış dizimler – Pascal gibi.
- Kısıtlanmamış dizimler – boyutlar nesnenin bir parçası (Java da benzer).

■ Java, C#

Ada gibi.

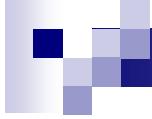


Parametre olarak çok boyutlu dizilimler (arrays)

■ Pre-90 FORTRAN

- Resmi parametrelerde dizimle birlikte boyutları da geçilebilir.

```
SUBPROGRAM SUB(MATRIX, ROWS, COLS, RESULT)
    INTEGER ROWS, COLS
    REAL MATRIX (ROWS, COLS), RESULT
    ...
END
```



Parametre geçirme metodları

■ Parametre geçirirmede tasarım etmenleri

1. Verimlilik

2. bir yönlü mü, iki yönlü mü?

□ Yukarıdaki iki etmen birbirleri ile çelişirler!

□ İyi programlama => değişkenlere limitli erişim; altprogramların dışarıdaki veriye erişimi mümkün olduğu kadar kısıtlanmalıdır. Mükünse tek yönlü erişim.

□ Verimlilik => belli bir büyüklüğü olan yapıları (structures) geçirmek için referans ile geçirme (pass by reference) en hızlı yoldur. Ancak bu durumda da altprogram dışarıdaki bir veriye erişmiş olur. Ada'da girdi, çıktı sınırlaması, C++'da "const" tipi parametre erişimi kısıtlar.

Parametre geçirme örnekleri

C: değer ile çağrıma

```
void swap1(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
swap1(c, d); // c ve d'nin içerikleri yer  
değişitmez.
```

C: referans ile çağrıma

```
void swap2(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
swap2(&c, &d); // c ve d'nin içerikleri yer  
değiştirir.
```

C++: referans ile çağrıma

```
void swap3(int &a, int &b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
swap3(c, d); // c ve d'nin içerikleri  
yer değiştirir.
```

Ada: değer ile çağrıma

```
procedure swap4(a : in out Integer,  
                b: in out Integer) is  
    temp : Integer;  
    temp := a;  
    a := b;  
    b := temp;  
end swap4;  
  
swap4(c, d); // c ve d'nin içerikleri yer değiştirir.
```

Parametre geçirme örnekleri

C gibi sözdizim: referans ile veya değer sonuç ile çağrıma karşılaştırması

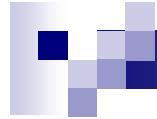
```
int i = 3; /* global değişken */
void fun(int a, int b) {
    i = b;
}
void main() {
    int list[10];
    list[i] = 5;
    fun(i, list[i]);
}
```

değer-sonuç ile çağrıma

```
addr_i = &i           // adresleri al
addr_listi = &list[i]
a = *addr_i           // a = 3
b = *addr_listi       // b = 5
i = b                 // i = 5
*addr_i = a            // i = 3
*addr_listi = b        // list[3]=5
```

referans ile çağrıma

```
a = &i
b = &list[i]
i = *b    // i = 5
```

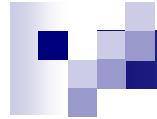


Parametre olan altprogram isimleri

Etmenler:

1. Altprogramın parametre tipleri kontrol edilecek mi?

- Erken Pascal ve FORTRAN 77 kontrol etmez.
- Sonraki Pascal versiyonları ve FORTRAN 90 kontrol eder.
- Ada altprogramların parametre olarak geçilmesine izin vermez.
- Java metod isimlerinin parametre olarak geçilmesine izin vermez.
- C ve C++ parametre olarak fonksiyon geçer (fonksiyon göstericileri ile) ve parametrelerinin tipleri kontrol edilebilir.



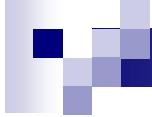
Parametre olan altprogram isimleri

2. Parametre olarak geçen bir altprogramın doğru referans çevresi (referencing environment)?

- Olasılıklar:

- a. Çağırılan altprogramın referans çevresi:
 - yüzeysel bağlama (shallow binding)
- b. Parametre olarak çağrılan altprogramın referans çevresi:
 - derin bağlama (Deep binding)
- c. Çağırılan altprogramın, çağrılan altprograma parametre olarak geçtiği referans çevresi
 - Özel bağlama (Ad hoc binding) (Hiç kullanılmamış).

- Statik kapsamlı (static-scoped) diller için derin bağlama en doğalıdır.
- Dinamik kapsamlı (dynamic-scoped) diller için yüzeysel bağlama en doğalıdır.

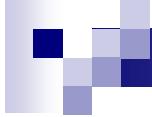


Parametre olan altprogram isimleri

Örnek:

```
sub1
  └─sub2
    └─sub3
      └─call sub4(sub2)
    └─sub4(subx)
      └─call subx
  └─call sub3
```

- **sub2 sub4**'ün parametresi olarak çağrııldığından referans çevresi nedir?
 - Yüzeysel bağlama (Shallow binding) => sub2, sub4, sub3, sub1
 - Derin bağlama (Deep binding) => sub2, sub1



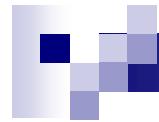
Fazla yüklü altprogramlar

Overloaded Subprograms

- Aynı referans çevresinde başka bir altprogramla aynı isimde olan altprograma "Fazla yüklü altprogram" (*overloaded subprogram*) denir.
- C++, Java, C# ve Ada yerleşik fazla yüklü altprogramlara sahiptirler ve yazılımcılar isterlerse kendi fazla yüklü altprogramlarını yazabilirler.
- C++, Java ve C#'da parametrelerinin farklılığından hangi altprogramın kullanılacağına karar verilir.
- Ada'da altprogramın geleceği değer de seçimde rol oynayabilir. Ada karışık modlu ifadelere izin vermediğinden bu özellik faydalıdır.
- Örneğin iki parametreleri aynı ama "integer" ve "float" dönen iki `fun` fonksiyonu olsun.

```
A, B : Integer;  
      . . . .  
A := B + Fun(7);
```

Bu durumda derleyici tam sayı dönen fonksiyonu kullanır.



Cinsine özgü (jenerik) altprogramlar (Generic Subprograms)

- Farklı etkinleştirmelerde farklı tiplerde parametre alabilen altprogramlara cinsine özgü (*generic*) veya çok biçimli (*polymorphic*) altprogram denir. Fazla yüklü altprogramlar özel çok biçimlilik oluştururlar.
- Hazırlamış bir altprogramın değişik veri yapıları için kullanılması, farklı veri yapıları için farklı altprogramlar hazırlanmaması ana motivasyon kaynağıdır.
- Olası parametre tipleri bir tip ifadesinde tanımlanmış, cinsine özgü (*generic parameter*) parametre alan altprogramlara parametrik çok biçimliliği olan altprogram denir.

Parametrik çok biçimliliğe (parametric polymorphism) örnekler

1. Ada

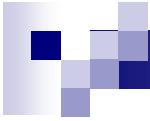
- Ada altprogramları ve paketlerinde tipler (Types), altsimge kapsamı (subscript range), sabit değerler gibi özellikleri cinsine özgü olabilir.

```
generic
  type INDEX is (<>);
  type ELEMENT is private;
  type VECTOR is array (INTEGER range <>)
    of ELEMENT;
  procedure GENERIC_SORT(LIST: in out VECTOR);
```

Cinsine özgü altprogramlar (örnekler)

```
generic
    type INDEX is (<>);
    type ELEMENT is private;
    type VECTOR is array (INTEGER range <>)
        of ELEMENT;
    procedure GENERIC_SORT(LIST: in out VECTOR);

procedure GENERIC_SORT(LIST: in out VECTOR) is
    TEMP : ELEMENT;
begin
    for Top in LIST'FIRST ..
                    Top'PRED(LIST'LAST) loop
        for Bottom in INDEX'SUCC(Top) ..
                        LIST'LAST loop
            if LIST(Top) > LIST(Bottom) then
                TEMP := LIST (Top);
                LIST(Top) := LIST(Bottom);
                LIST(Bottom) := TEMP;
            end if;
            end loop; -- for Top ...
        end loop; -- for Bottom ...
end GENERIC_SORT;
```



Cinsine özgü altprogramlar (örnekler)

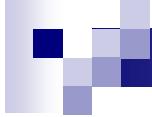
```
procedure INTEGER_SORT is new GENERIC_SORT (
    INDEX => INTEGER;
    ELEMENT => INTEGER;
    VECTOR => INT_ARRAY);
```

- Yukarıdaki somutlaştırılmaya kadar jenerik tanım (bir önceki sayfa) için bir kod üretilmez. Ne zaman ki yukarıdaki somutlaştırma gerçekleştirilir, derleyici GENERIC_SORT'un tamsayıları sıralayan versiyonu olan INTEGER_SORT alt fonksiyonunu üretir.
- Ada'da fonksiyon isimlerinin parametre olamadığı düşünülürse, bu yöntem çok amaçlı fonksiyon hazırlamak açısından da faydalıdır. Bir sonraki örneğe bakalım.

Cinsine özgü altprogramlar (örnekler)

```
generic
  with function FUN(X : FLOAT) return FLOAT;
  procedure INTEGRATE(LOWERBD : in FLOAT;
                      UPPERBD : in FLOAT;
                      RESULT : out FLOAT);
procedure INTEGRATE(LOWERBD : in FLOAT;
                      UPPERBD : in FLOAT;
                      RESULT : out FLOAT) is
  FUNVAL : FLOAT;
begin
  ...
  FUNVAL := FUN(LOWERBD);
  ...
end;
INTEGRATE_FUN1 is new INTEGRATE(FUN => FUN1);
```

FUN1 fonksiyonunun entegralini alan özel program.



Cinsine özgü altprogramlar (örnekler)

2. C++

- Şablon fonksiyonlar (Template functions)

- **template <şablon parametreleri>**

şablon parametreleri:

class tanımlayıcı (identifier) – tip isimleri için kullanılır.

tip_adı tanımlayıcı – şablon fonksiyona değer geçmek için kullanılır.

- Örneğin

```
template <class Type>
```

```
Type max(Type first, Type second) {  
    return first > second ? first : second;  
}
```

- "Type" fonksiyonun işleyeceği veri tipini gösterir. Bu fonksiyon ">" işlecinin işleyebileceği bütün veri tipleri ile çalışır.

Cinsine özgü altprogramlar (örnekler)

□ Örnek:

```
template <class Type>
Type max(Type first, Type second) {
    return first > second ? first : second;
}
```

□ Yukarıdaki işlem makrolar ile de yapılabilirse de, yan etkilerine dikkat etmek gereklidir:

- `#define max(a,b) ((a) > (b)) ? (a) : (b)`
- ancak bu makro `max(x++,y)` şeklinde kullanılırsa:
- `((x++) > (y)) ? (x++) : (y)` şeklinde hatalı kod üretir. (`x` iki defa büyütülüyor!)
- Aynı sorun şablon alt programda yok.
- C++ şablon programları örtülü olarak, ismi ile çağrıldığında veya & işaretci ile adresi alındığında somutlaştırılır (instantiated).

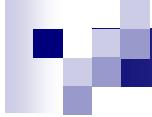
```
□
int a,b,c;
float d,e,f;
...
c = max(a,b);
f = max(d,e);
```

Cinsine özgü altprogramlar (örnekler)

```
template <class Type>
void generic_sort(Type list[], int len) {
    int top, bottom;
    Type temp;

    for (top = 0; top < len - 2; top++)
        for (bottom = top + 1; bottom < len - 1;
            bottom++) {
            if (list[top] > list[bottom]) {
                temp = list [top];
                list[top] = list[bottom];
                list[bottom] = temp;
            } //** for (bottom = ... sonu
    } //** generic_sort sonu

/** şablon fonksiyonun somutlaştırılması
float flt_list[100];
...
generic_sort(flt_list, 100);
```



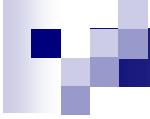
Jenerik Altprogramlarda Tasarımla ilgili etmenler

1. Yan etkilere izin verilecek mi?

- a. İki yönlü parametreler (Ada izin vermez)
- b. Lokal olmayan referans (hepsi izin verir)

Yukarıdakilere izin verildiği zaman yan etkiler (parametrelerin değişme olasılığı) kaçınılmazdır.

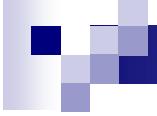
2. Ne tip dönülen değere izin verilecek?



Altprogramlarda Tasarımla ilgili etmenler

■ Olası dönülen tipler için Dil örnekleri:

1. FORTRAN, Pascal – sadece basit tipler dönerler.
2. C – fonksiyonlar ve dizilimler hariç her tip (fonksiyonlar ve dizilimler göstericiyle dönülür).
3. Ada – her tip (altprogramlar tip değildir bu nedenle dönülmeyecektir ama göstericisi dönülebilir).
4. C++ – C gibi, ayrıca kullanıcı tanımlı tipler ve "class" dönülebilir.
5. Java ve C# – Bu iki dilde fonksiyonlar yoktur ancak onlar gibi işlem gören nesnelerin metodları vardır. Bu dillerde bütün tipler ve sınıflar ("class") dönülebilir. Metotlar tip olmadığından dönülemez.
6. JavaScript – Bazı dillerde fonksiyonlar veri nesneleri gibidir ve bu nedenle parametrede geçilebileceği gibi değer olarak da dönülebilirler. Birçok fonksiyonel dilde de bu böyledir.



Kullanıcı tanımlı fazla yüklü işaretçiler (overloaded operators)

- Fazla yüklü işaretçiler bütün programlama dillerinde bulunur.
- Kullanıcılar C++ ve Ada'da bunları daha fazla da yükleyebilirler (C++'ın devamı niteliğindeki Java'ya bu özellik taşınmamıştır).

Kullanıcı tanımlı fazla yüklü işaretçiler

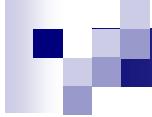
- Örnek vektör iç çarpımı (vektörün aynı yöndeki elemanlarının çarpımlarının toplamı) (Ada) varsayıyalım **VECTOR** tipi **INTEGER** tipi elemanları olan bir dizilim (array) olsun:

```
function "*" (A, B : in VECTOR) return INTEGER is
    SUM : INTEGER := 0;
begin
    for INDEX in A'range loop
        SUM := SUM + A(INDEX) * B(INDEX);
    end loop;
    return SUM;
end "*";
```

- Aynı örneğin C++ benzerinin prototipi ise aşağıdaki gibi olurdu:

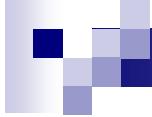
```
int operator * (const vector A, const vector B, int boy);
```

- Kullanıcı tanımlı fazla yüklü işaretçiler iyi mi yoksa kötü müdür? Kişisel olarak beğenisiye kalmış.
- Ancak programın okunabilirliğini azaltır. "a*b" ifadesinin sayısal bir çarpım mı yoksa vektör çarpımı mı hatırlamak zor olabilir.
- Farklı ekiplerin hazırladığı program parçacıkları işaretlere çelişen görevler yükleyebilirler, ayırt edilmesi zor olur.



Eşyordamlar (Coroutines)

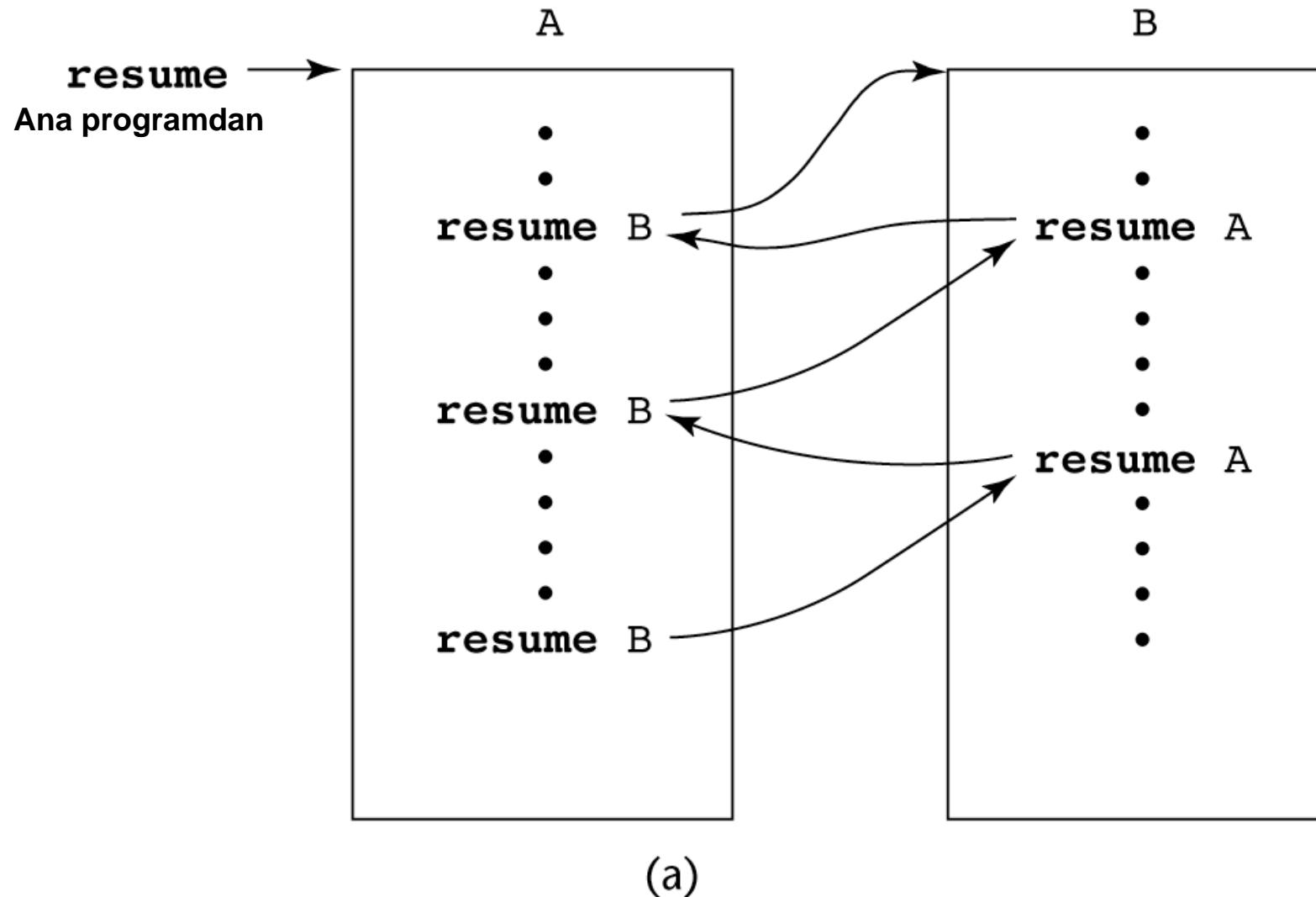
- İlk eşyordam kullanan programlama dili Simula 67 dir. Simula sistem simulasyonları (benzetim, gerçek sistemin bilgisayarda benzerini yaparak incelemek) için geliştirilmiştir. Daha sonra kullanan diller BLISS (Wulf et al., 1971), INTERLISP (Teitelman, 1975) ve Modula-2 (Wirth, 1985).
- Bir eşyordam (**coroutine**) birden çok giriş noktası olan ve bunları kendisi kontrol eden, daha önceki kullanılışlarını hatırlayan, çağrıranla eşit yetkilere sahip bir program birimidir.
- Bu nedenle simetrik kontrol da (**symmetric control**) da denir.
- Bu nedenle eşyordam çağrılmamasında sürdür (**resume**) kullanılır.
- Eşyordama ilk giriş başından olur fakat daha sonraki girişler hep bırakılan yerden devam şeklinde olur.

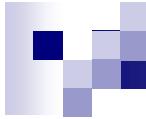


Eşyordamlar (Coroutines)

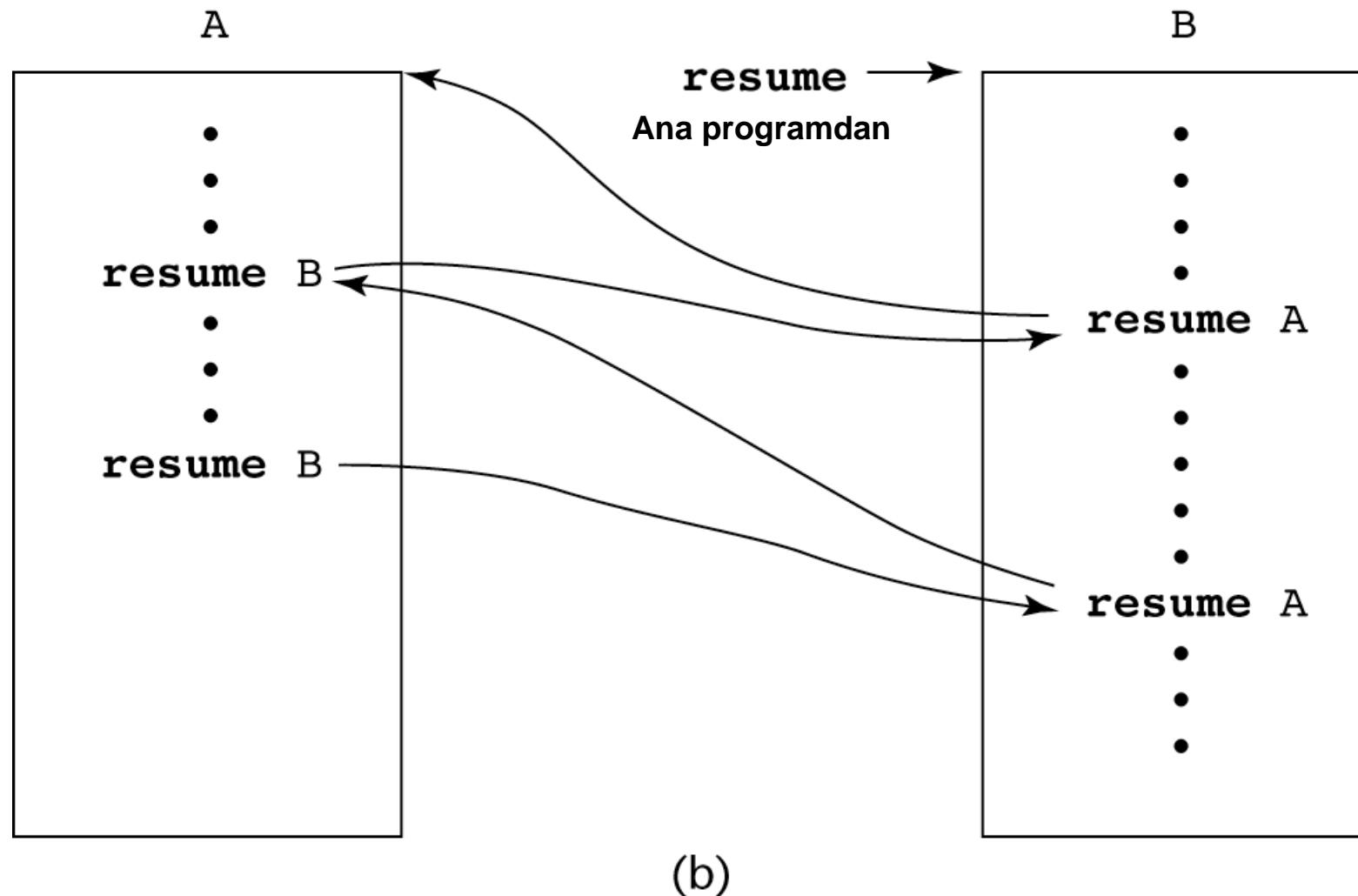
- Tipik olarak eşyordamlar birbirlerin tekrar tekrar yürütürler.
- Eşyordamlar paralel işlemcili makinelerde yarı koşut zamanlı (*quasi-concurrent execution*) olarak çalışabilirler.

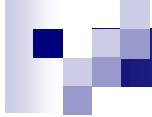
Eşyordamlar (Coroutines)



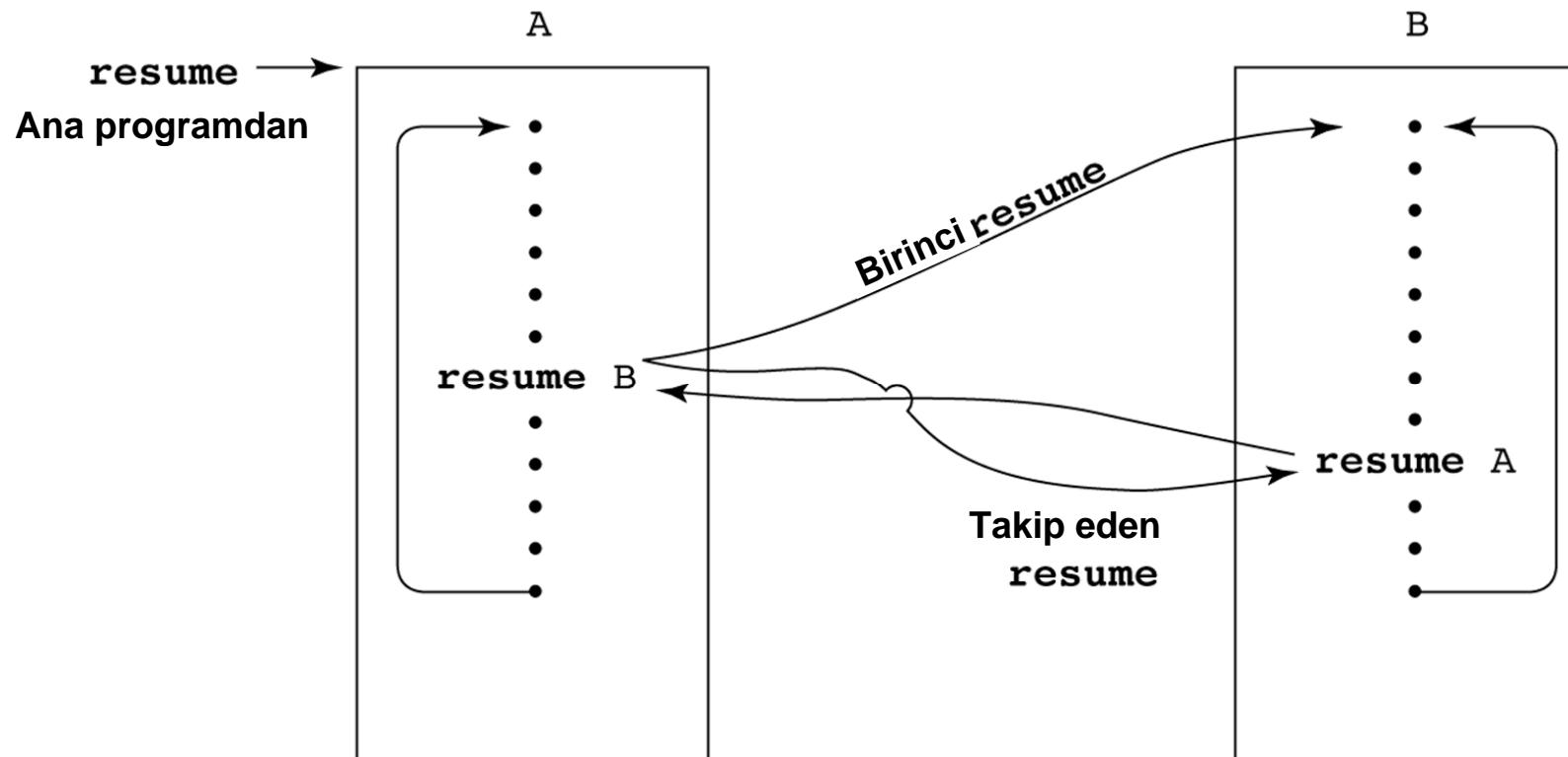


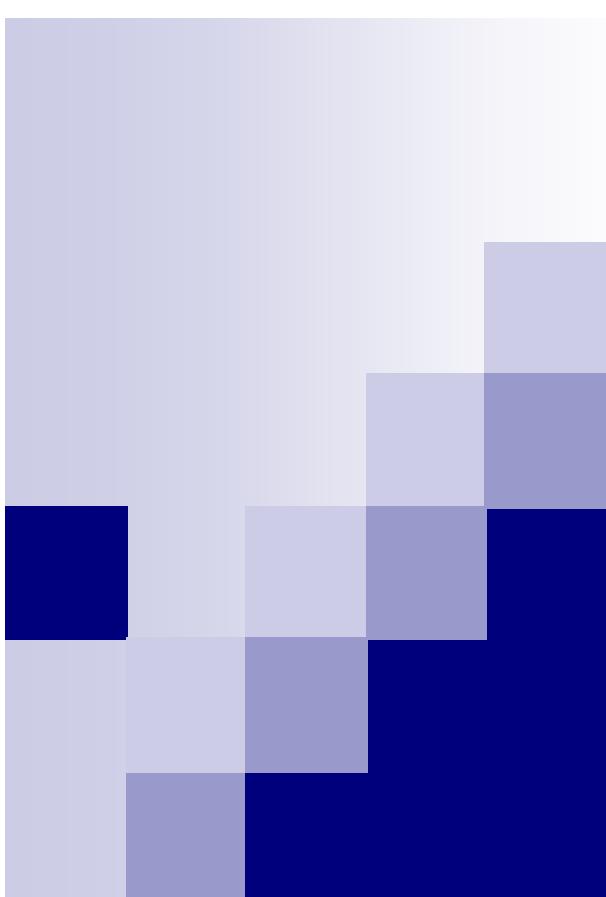
Eşyordamlar (Coroutines)





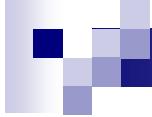
Eşyordamlar (Coroutines)





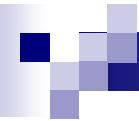
Bölüm 10

Altprogramları Uygulamak

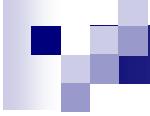


Altprogramları Uygulamak -- Başlıklar

- Çağrıların genel anlam analizi
- Basit altprogramların gerçekleştirilmesi
- Yığıt dinamik yerel değişkenlerle altprogramları uygulamak
- İç içe altprogramlar
- Bloklar
- Dinamik kapsamlı uygulamalar

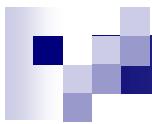


Bilgisayar mühendisleri neden
cerrah olamaz



Çağrıların ve döndükleri değerlerin genel anlam analizi

- Bir dilde altprogramların çağrılmaması ve dönmesi işlemlerine altprogram bağlanması ([subprogram linkage](#)) denir.
- Parametrelerin nasıl alt programa geçileceği belirlenmelidir.
- Çağırılanın değerleri korunmalıdır.
- Alt program yerel değişkenleri statik değilse yerel olarak saklanmalıdır.
- Dönüşte çağrıranın doğru noktasına, doğru değerlerle dönülmesi temin edilmelidir.
- İç içe altprogramlar varsa, yerel olmayan değişkenlere erişim sağlanmalıdır.

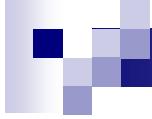


"Basit" altprogramların gerçekleştirilmesi Implementing “Simple” Subprograms

Basit altprogramdan kastımız, bütün yerel değişkenleri statik olan ve iç içe çağrılamayan altprogramlar (örn. Fortran I).

Çağrı anlam analizi:

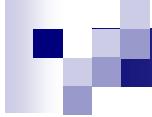
1. Çağırılan programın çalışma durumunu sakla (Save the execution status of the caller).
2. Parametre geçme işlemlerini yap (Carry out the parameter-passing process).
3. Dönüş adresini çağrılna geç (Pass the return address to the callee)
4. Kontrolü çağrılna ver (Transfer control to the callee).



"Basit" altprogramların gerçekleştirilmesi Implementing “Simple” Subprograms

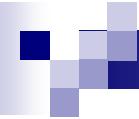
■ Dönüş analizi (Return Semantics):

1. Eğer sonucu değeri ile geç (pass-by-value-result) parametreler kullanılmışsa değerlerinin gerçek parametrelere geçir.
2. Eğer dönen fonksiyonsa, döneceği değeri çağrıranın bulması gereken yere yerleştir.
3. Çağırılanın çalışma ortamını yeniden yapılandır.
4. Kontrolü çağrıvana geri ver.



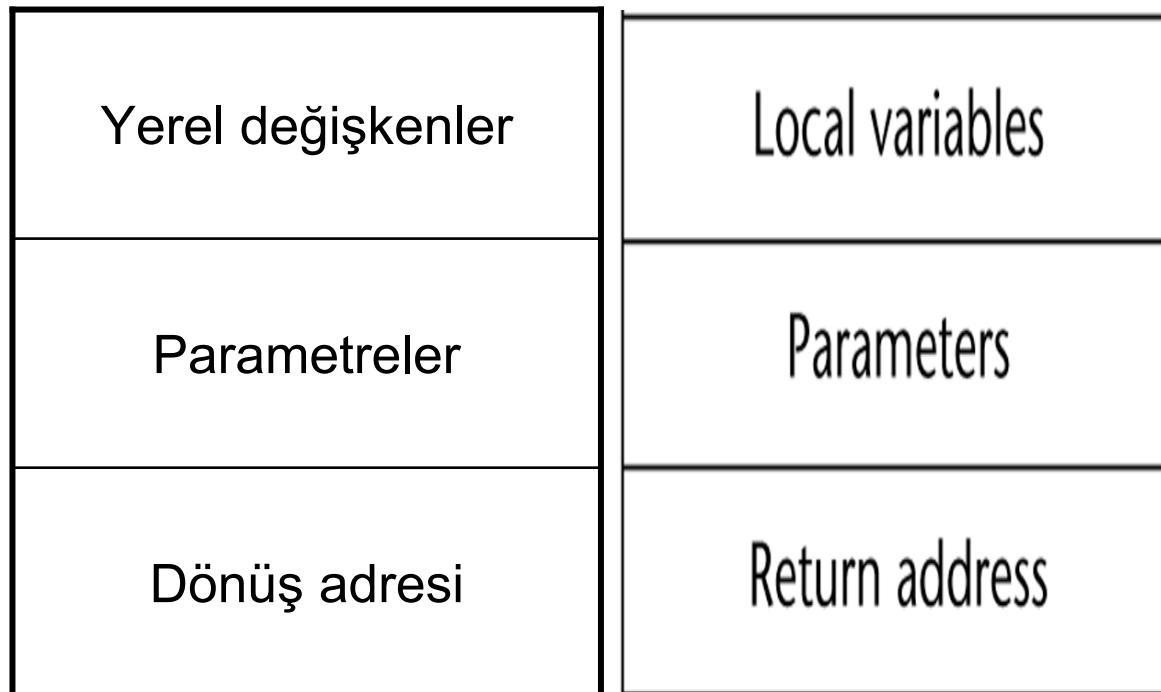
"Basit" altprogramların gerçekleştirilmesi Implementing “Simple” Subprograms

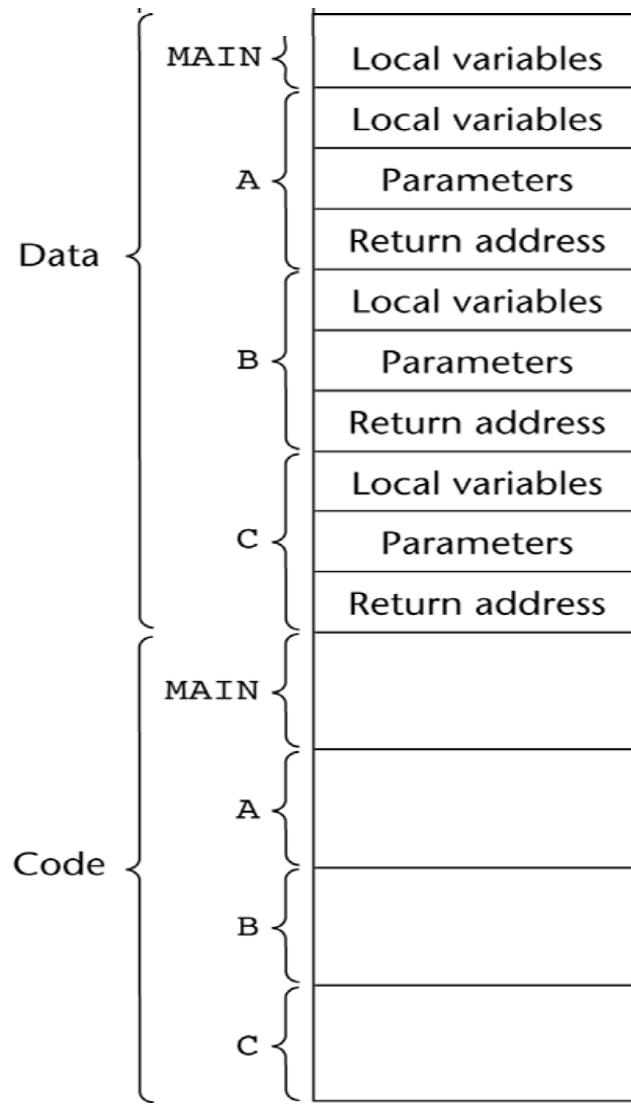
- Gerekli depolama: Çağırılan durum bilgileri, parametreleri, dönüş adresi ve doneceği değer (eğer fonksiyonsa).
- Altprogramın çalışan kod olmayan, altprogram verilerinin tutulduğu kısmının biçimi veya yerleşim planına **etkinleştirme kaydı (activation record)** denir. Bu kaydın sonraki sayfalarda göreceğimiz gibi belli bir formatı olur.
- Altprogram çağrııldığı zaman belli değerlerle doldurulmuş haline somutlaşan **gerçekleşme** kaydı denir (**instance of activation record**).



"Basit" altprogramlarının etkinleştirme kaydı

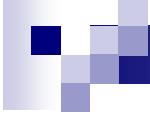
An Activation Record for “Simple” Subprograms





"Basit" altprogramlarının etkinleştirme kaydı ve kodu

- Etkinleştirme kaydı verileri statik bellekte statik olarak saklanıyor.
- Bu nedenle somutlaşmış tek etkinleştirme kaydı olabilir.
- Bu nedenle kullanım olarak kolay, erişim daha hızlı, ancak özyinelemeyi desteklemez.
- İlk Fortran derleyicileri bu tip etkinleştirme kaydı tutuyordu.

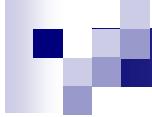


Altprogramları yığıt dinamik yerel değişkenlerle gerçekleştirmek

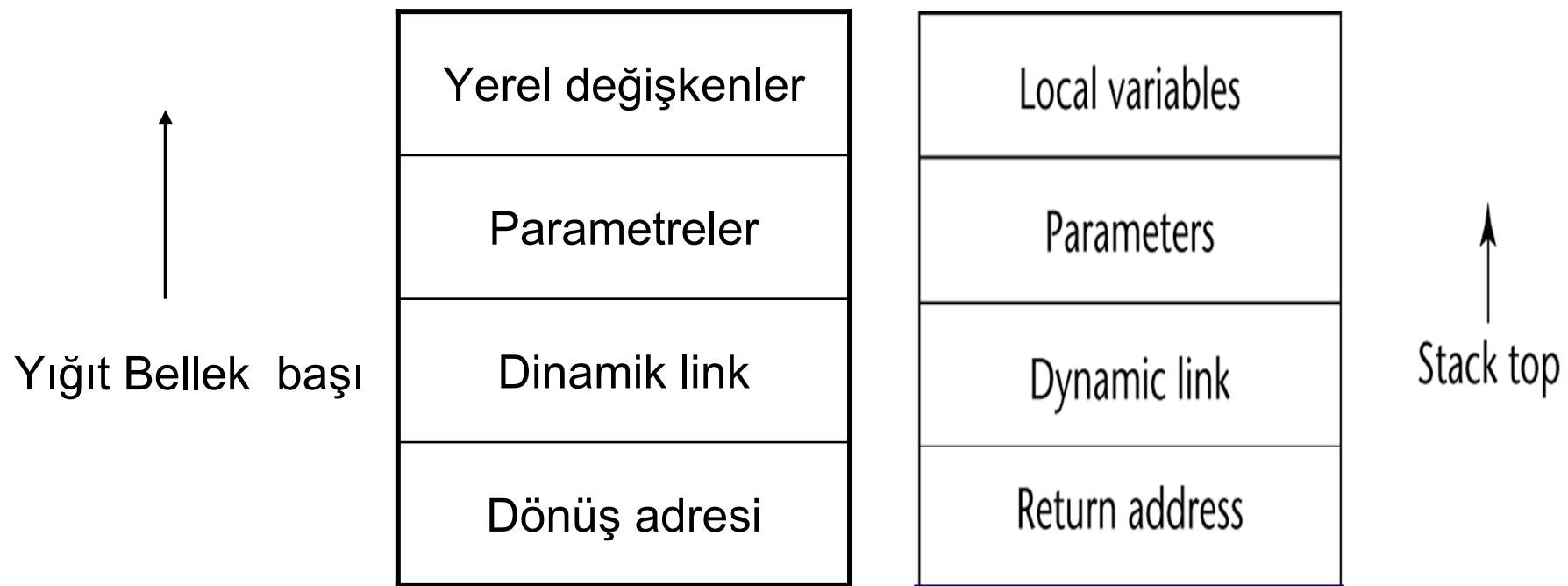
Implementing Subprograms with Stack-Dynamic Local Variables

■ Daha karmaşık çünkü:

- Derleyici altprogram yerel değişkenlerine yığıt bellekte örtülü bellek tahsis ve geri alma işlemleri için kod hazırlamalıdır.
- Özyineleme desteklenebildiğinden altprogramın aynı anda birden çok peş peşe çağrırlabilmesini destekler, bu da birden çok etkinleştirme kodu somutlaşmasına neden olur.
- Bazı dillerde yerel dizilimlerin (local arrays) boyu çağrı sırasında belirlendiğinden, etkinleştirme kodunun içinde yer alan bu tip değişkenler etkinleştirme kodunun boyunun dinamik olmasını gerektirir.



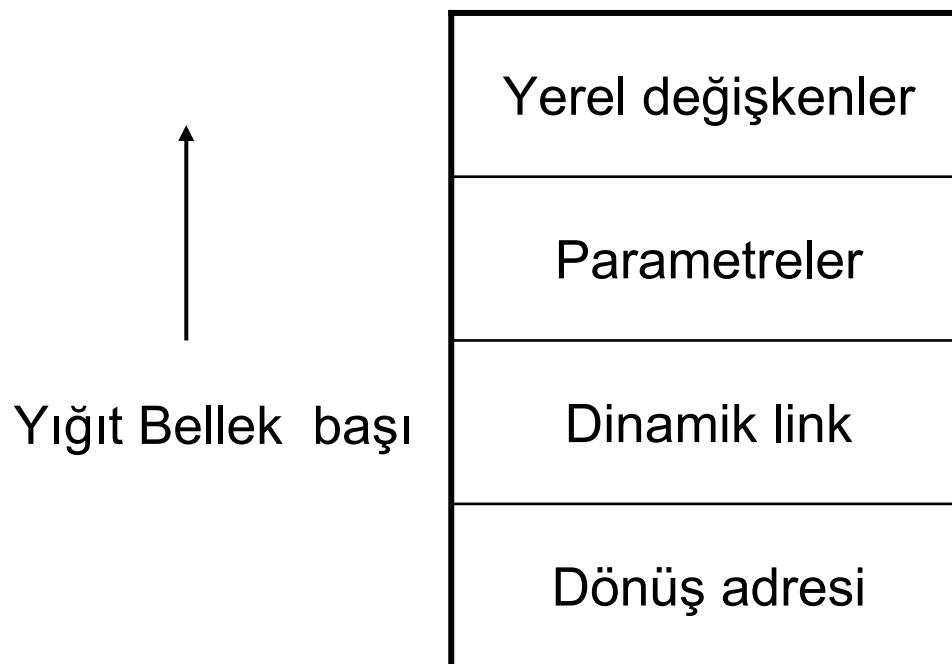
Tipik Altprogramları yiğit dinamik yerel değişkenlerle gerçekleştirmeye etkinleştirme kaydı





Altprogramları yiğit dinamik yerel değişkenlerle gerçekleştirmek

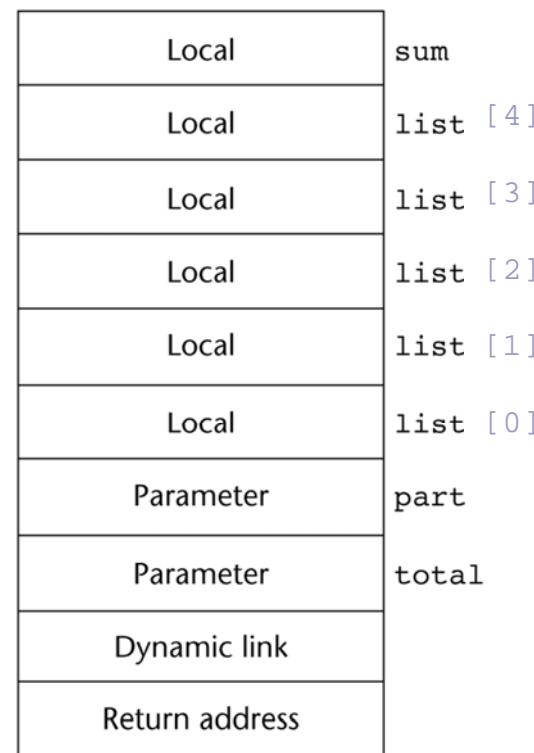
- Etkinleştirme kaydı formatı statik ve fakat boyutları dinamiktir.
- "dynamic link" çağrıran programın etkinleştirme kaydının başını gösterir.
- Etkinleştirme kaydı dinamik olarak altprogram çağrııldığında üretilir.



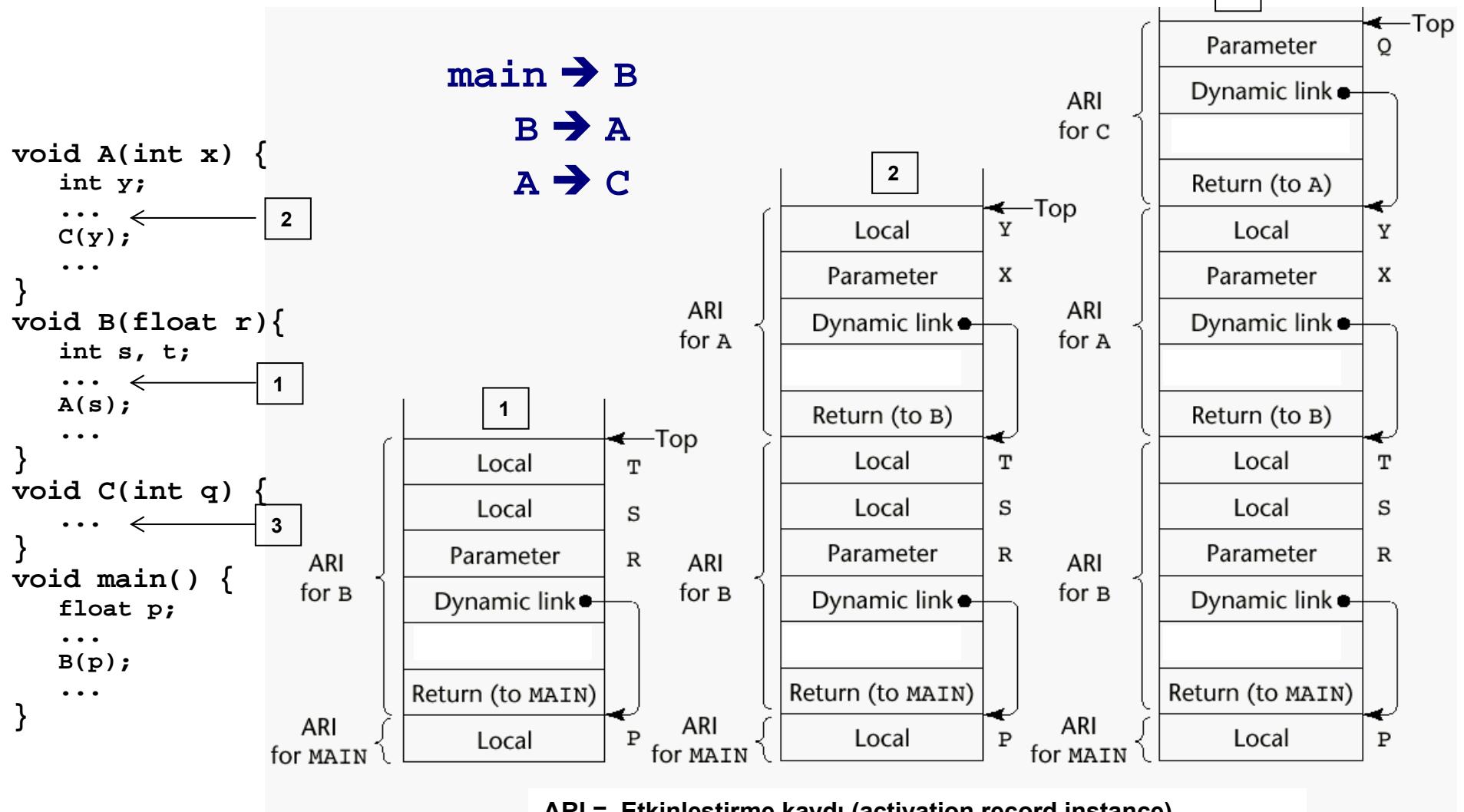
C fonksiyonu örneği

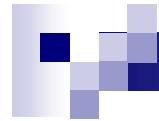
```
void sub(float total, int part)
{
    int list[5];
    float sum;

    ...
}
```



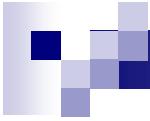
Özyinelemesiz C programı örneği – Program için yığıt bellek içeriği (Stack Contents For Program)





Altprogramların gerçekleştirmesi

- Herhangi bir anda yığittaki dinamik linklerin toplamına çağrı zinciri ([call chain](#)) denir.
- Yerel değişkenlere etkinleştirme kaydının (activation record) başlangıcına göre bağıl konumlarından (offset) erişilir. Buna yerel bağıl konum denir ([local_offset](#)).
- Yerel değişkenlerin yerel bağıl konumları derleyici tarafından belirlenir.

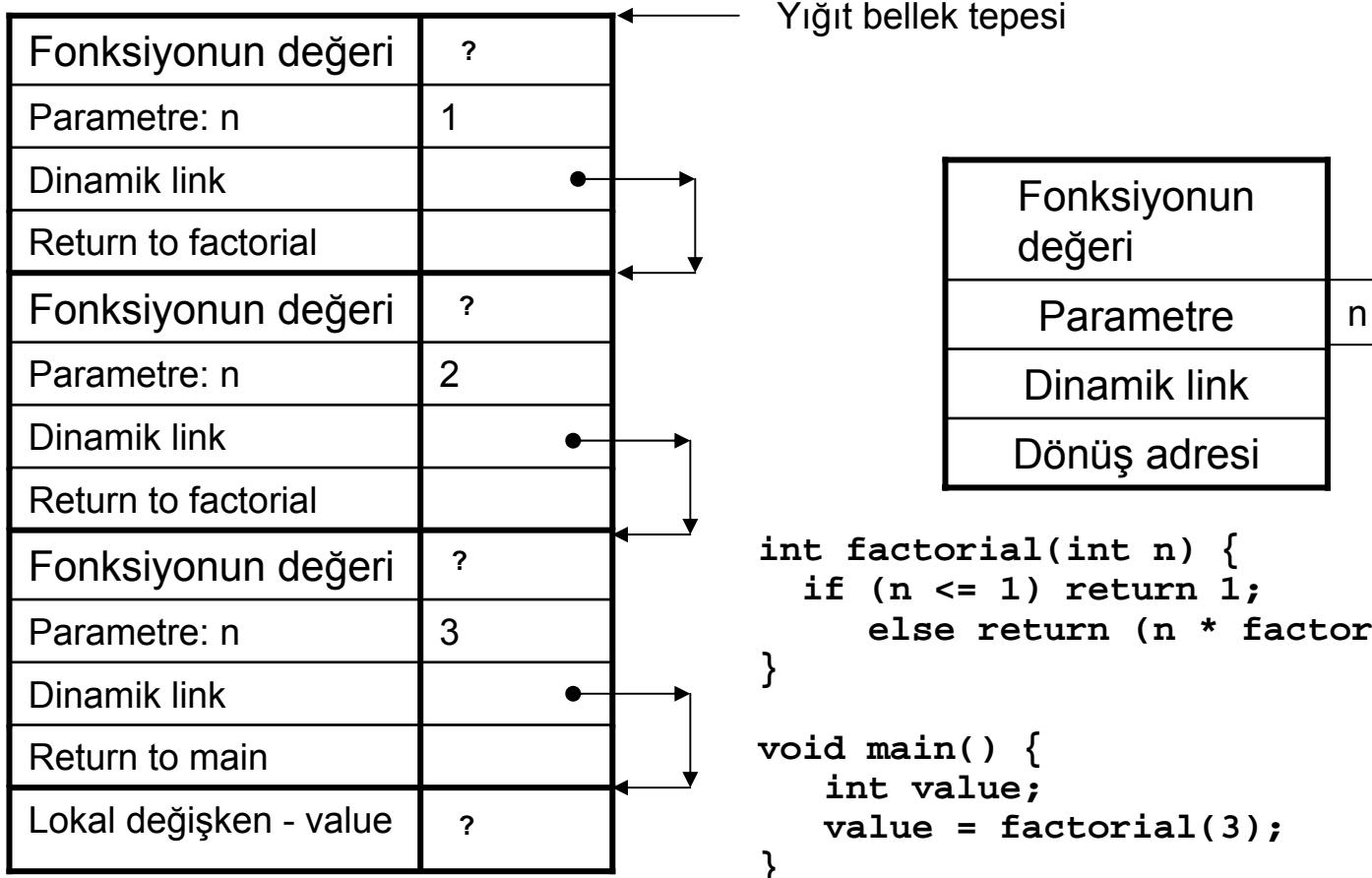


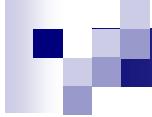
Özyineleme

- Bir önceki örnekte kullanılan etkinleştirme kaydı (activation record) özyinelemeli fonksiyonlarda da kullanılabilir:

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else return (n * factorial(n - 1));  
}  
void main() {  
    int value;  
    value = factorial(3);  
}
```

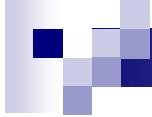
factorial için etkinleştirme kaydı





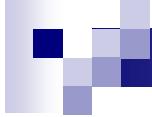
İçiçe altprogramlar(Nested subprograms)

- Bazı programlama dilleri (Fortran 95, Ada, JavaScript) yiğit dinamik yerel değişkenler kullanır (use stack-dynamic local variables) ve içiçe altprogramlara izin verirler.
- Kural: yerel olarak erişilmeyen tüm değişkenler yiğit bellekte aktif bir etkinleştirme kaydı içinde bulunurlar.
- Yerel olmayan değişken referansı bulma işlemi:
 1. Doğru etkinleştirme kaydını bul.
 2. Etkinleştirme kaydında değişkenin bağıl konumunu bul.



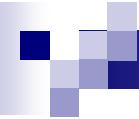
Yerel olmayan değişken referansını bulma işlemi

- Etkinleştirme kaydı içinde bağıl konumu bulmak kolay.
- Doğru etkinleştirme kaydını bulmak:
 - Statik anlam kuralları gereği yerel olarak erişilmeyen tüm değişkenler yiğit bellekte aktif bir etkinleştirme kaydı içinde bulunurlar.



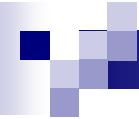
Yerel olmayan değişken referansını bulma işlemi

- **1. teknik – Statik zincir (Static Chains)**
- Belli etkinleştirme kayıtlarını birleştiren statik link zincirine statik zincir (*static chain*) denir.
- Bir altprogramın etkinleştirme kaydını ebeveyninin etkinleştirme kaydına bağlayan linke statik link (*static link*) denir.
- Statik zincir bir etkinleştirme kaydını yığıt bellekte çağrıran altprogramların etkinleştirme kayıtlarına bağlar.

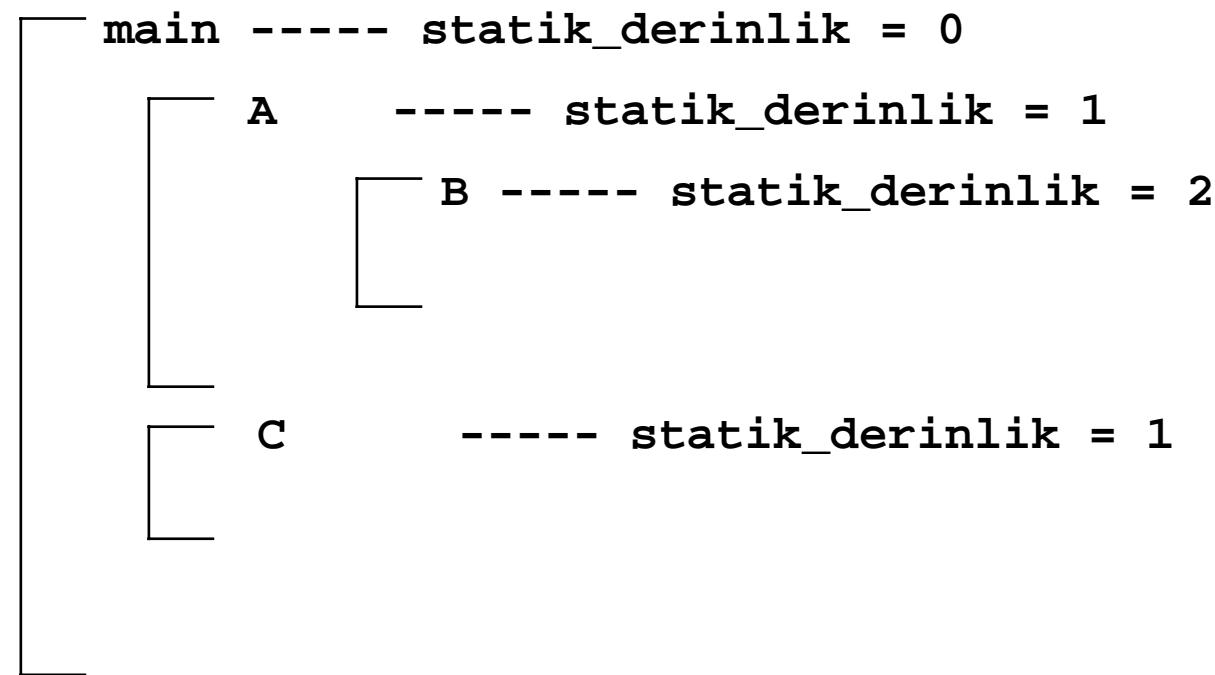


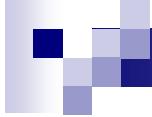
1. teknik – Statik zincir (Static Chains)

- Yerel olmayan değişkenin tanımlandığı yeri bulmak için:
 - Statik zincir üzerinden etkinleştirme kayıtlarında değişkenin adı aranır.
- Etkinleştirme kaydının statik zincirdeki derinliğine **statik_derinlik** (static_depth) denir.



1. teknik – Statik zincir (Static Chains)





1. teknik – Statik zincir (Static Chains)

- Tanım: `zincir_bağılkonum` (`chain_offset`) veya `içiçe_bağılkonum` (`nesting_depth`): Bir değişkenin referans verildiği altprogramın statik derinliği ile tanımlandığı altprogramın statik derinliği arasındaki fark.
- Bir referans aşağıdaki çift ile gösterilebilir:
(`zincir_bağılkonum`, `yerel_bağılkonum`)
burada `yerel_bağılkonum` (`local_offset`), değişkenin bulunduğu etkinleştirme kaydındaki bağıl konumudur.

1. teknik – Statik zincir (Static Chains)

Örnek Pascal Programı

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
        A := B + C; <-----1
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
    procedure SUB3;
      var C, E : integer;
      begin { SUB3 }
        SUB1;
        E := B + A; <-----2
      end; { SUB3 }
    begin { SUB2 }
      SUB3;
      A := D + E; <-----3
    end; { SUB2 }
    begin { BIGSUB }
      SUB2(7);
    end; { BIGSUB }
  begin
    BIGSUB;
  end. { MAIN_2 }
```

1. teknik – Statik zincir (Static Chains)

Örnek Pascal Programı

MAIN_2 için çağrılmış sıralaması

MAIN_2 → BIGSUB

BIGSUB → SUB2

SUB2 → SUB3

SUB3 → SUB1

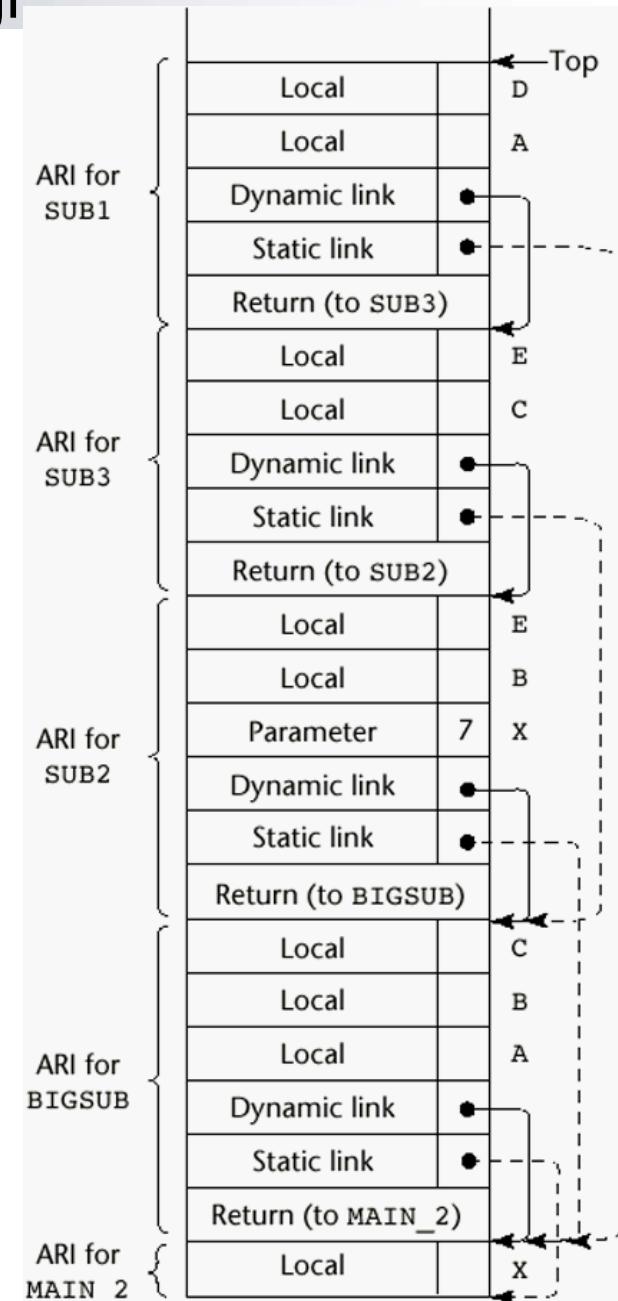
```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <-----1
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A;  <-----2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <-----3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end. { MAIN_2 }
```

1. pozisyonda yığıt içeriği

```

program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <-----1
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A;  <-----2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <-----3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
end. { MAIN_2 }

```



Örnek Pascal Programı

```
program MAIN_2;
var X : integer;
procedure BIGSUB;
  var A, B, C : integer;
procedure SUB1;
  var A, D : integer;
begin { SUB1 }
  A := B + C; <-----1
end; { SUB1 }
procedure SUB2(X : integer);
  var B, E : integer;
procedure SUB3;
  var C, E : integer;
begin { SUB3 }
  SUB1;
  E := B + A; <-----2
end; { SUB3 }
begin { SUB2 }
  SUB3;
  A := D + E; <-----3
end; { SUB2 }
begin { BIGSUB }
  SUB2(7);
end; { BIGSUB }
begin
  BIGSUB;
end. { MAIN_2 }
```

1. pozisyon SUB1:

MAIN_2->BIGSUB->SUB2->SUB3->SUB1

A - (0, 3) ($2-2=0$)

B - (1, 4) ($2-1=1$)

C - (1, 5) ($2-1=1$)

2. pozisyon SUB3:

MAIN_2->BIGSUB->SUB2->SUB3

E - (0, 4) ($3-3=0$)

B - (1, 4) ($3-2=1$)

A - (2, 3) ($3-1=2$)

3. pozisyon SUB2:

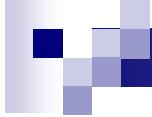
MAIN_2->BIGSUB->SUB2

A - (1, 3) ($2-1=1$)

D - an error (sub1 içinde tanımlı)

E - (0, 5) ($2-2=0$)

(zincir_bağılkonum, yerel_bağılkonum)



1. teknik – Statik zincir (Static Chains)

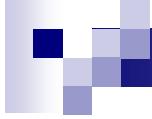
■ Statik zincir uygulaması

□ Altprogram çağrıldığı zaman (altprogram parametre ve isimle geçen parametre olmadığını varsayıarak) :

- Gerçekleştirme kaydı somutlaşan örneğini (instance) hazırla.
- Eski yığıt bellek tepesine bir dinamik link.
- Statik link, statik ebeveynin en son yaratılmış etkinleştirme kaydını gösterir.

□ İki metot:

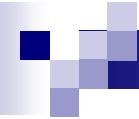
1. Statik ebeveynin ilk etkinleştirme kaydını bulmak için dinamik zincirde ara.
Kolay fakat yavaş.



1. teknik – Statik zincir (Static Chains)

2. Derleyici, derleme esnasında çağrıranla, çağrılan arasındaki derinliği (zincir bağılı konum) hesaplar ve daha sonra program yürütülürken kullanılmak üzere saklar.

- Örneğin **MAIN_2** nin yığıt içeriğine bakarsak: **SUB1**, **SUB3**'ün içinden çağrırlıken derleyici **SUB3**'ün **SUB1**'in tanımlandığı altprograma (**BigSub**) göre derinliğinin iki olduğu bilgisini programa yerleştirir. Program çalışırken bu noktada yerleştirilen bilgi kullanılarak etkinleştirme kaydına olan statik link kurulur.

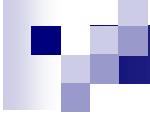


1. teknik – Statik zincir (Static Chains)

■ Statik zincir metodunun değerlendirilmesi

□ Sorunlar:

1. Yerel olmayan referanslar, özellikle derinlik fazlaysa, yavaş çalışır.
Programın performansı düşer.
2. Yerel olmayan referansların kullandığı zaman eşit olmadığından,
zamana hassas program yazmak zorlaştığı gibi bunların güncellenmesi
de bu zamanlamayı değiştirir.



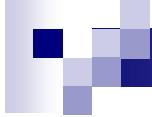
İçice altprogramlar

■ 2. Teknik - ekranlar

- Fikir: Statik linkleri "ekran" (display) denilen ayrı bir yığıta (stack) yerleştirir. Ekrandaki veriler etkinleştirme kodlarına (activation record) göstericilerdir.
- Referansları:
(ekran_bağılkonum, yerel_bağılkonum) (display_offset, local_offset)
olarak gösterebiliriz. Burada ekran_bağılkonum değeri zincir_bağılkonumla aynıdır.

■ Referansların işleyışı:

- ekran_bağılkonum'u kullanarak, aranılan değişkenin doğru etkinleştirme kaydının göstericisini bul.
- yerel_bağılkonum değerini kullanarak, etkinleştirme kaydı içinde aranılan değişkeni bul.

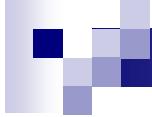


Ekranlar (Displays)

- Ekran uygulaması (altprogram parametre ve isimle geçen parametre olmadığını varsayıarak) :

Not: ekran_bağılkonum'u sadece etkinleştirme kaydı türetilmekte olan altprogramın statik derinliğine bağlıdır. Bu altprogramın statik derinliğidir.

- Belli bir anda çalışan altprogramın statik derinliği k ise, ekranda $k+1$ gösterici olur ($k=0$ ana program).

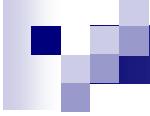


Ekranlar (Displays)

- Statik derinliği k olan P altprogramını çağırmak için:
 - a. Yeni bir etkinleştirme kaydına k pozisyonundaki ekran göstericisini kopyala.
 - b. Ekranda k pozisyonuna P altprogramının etkinleştirme kaydının göstericisini koy.
- Çıkışta, etkinleştirme kaydında saklanan ekran göstericisini yerine geri koy.

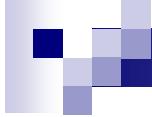
Bölüm 11

Soyut Veri Tipleri ve Kılıflama (sarmalama) Kavramı
Abstract Data Types and Encapsulation Concepts



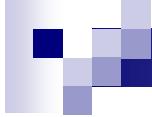
Bölüm 12 Başlıklar

- Soyutlama Kavramı
- Veri soyutlamasına giriş
- Soyut Veri Tiplerinin (SVT) (Abstract Data Types) tasarım unsurları
- Dil örnekleri
- Parametrize (jenerik) edilmiş SVT
- Kılıflama/sarmalama yapıları (encapsulation constructs)
 - Veri bütütlerini adres, hata düzeltimi, alındı gibi kontrol bilgileri ile donatmak
 - Nesneye yönelik programlamada, programcının bilmesi gerekmeyen bir sınıfın özelliklerini ayrı bir dosyaya koyma.
- Kılıflamayı isimlendirme (Naming Encapsulations)



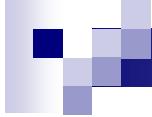
Soyutlama Kavramı

- Soyutlama, bir şeyin en önemli özellikleri ile görülmesi ve temsil edilmesidir.
- Programlama ve bilgisayar bilimlerinde soyutlama kavramı önemlidir.
- Hemen hemen bütün programlama dilleri *görev soyutlamasına* alt programlar vasıtası ile destek verirler.
- 1980 sonrası hemen hemen bütün programlama dilleri *veri soyutlamasını* desteklerler.



Veri soyutlamasına giriş

- **Soyut veri tipi** kullanıcı tanımlı veri tipidir aşağıdaki iki özelliğe sahiptir:
 1. Söz konusu tip ile elemanlarının gösterimi (betimlemesi) ve üzerlerinde yapılabilecek işlemler tek bir sözdizimsel birim ile tanımlanır.
 - Avantaj: Program organizasyonu, değiştirilebilme (veri yapısı ile ilgili herşey birlikte), parçaların ayrı derlenmesi
 2. Bu elemanların gösterimi kullanacak program birimlerinden saklıdır. Bu nedenle bunlara sadece tip tanımlarında verilen işlemler yapılabilir.
 - Avantaj: Güvenilirlik – veri gösterimlerinin saklı olması kullanıcı kodunun doğrudan veri elemanlarına erişimini engellediği gibi gösterimine bağlı olmamasını da sağlar. Kullanıcı kodu değiştirilmeden veri gösterimi değiştirilebilir.



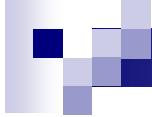
Veri soyutlamasına giriş

- Ön tanımlı tipler SVTdir

Örneğin Java `int` tipi

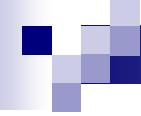
- Gösterimi saklıdır (iç yapısını bilmenize gerek yoktur)
- Yapılabilecek işlemler ön tanımlıdır
- Kullanıcı programları `int` tipinde değişkenler tanımlayabilirler

- Kullanıcı tanımlı soyut veri tiplerinin ön tanımlı SVT ile aynı özelliklere sahip olmaları gereklidir.



Tasarım Unsurları

- SVT destekleyebilmek için bir dilde olması gereklili özellikler:
 1. Tip tanımını saracak sözdizimsel birim.
 2. Tip adlarını ve alt program başlıklarını kullanıcılar açacak, buna karşılık tanımları ve gösterimleri saklayacak yöntem.
 3. Bazı temel işlemlerin dilde ön tanımlı olması gereklidir (genellikle atama, karşılaştırma)
 - Bazı işlemlere genel olarak gerek duyulur ancak tip tasarımcısı tarafından tanımlanmaları gereklidir.
 - Örneğin, döngü, “constructors”, “destructors”



Dil Örnekleri

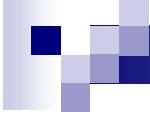
1. Ada

- Sarmalayan yapı “package”
- “Package”ler genellikle iki kısımdan oluşur:
 1. Özellikler kısmı (arayüz (the interface))
 2. Ana paket (Özellikler kısmında listelenen elemanların gerçeklestirimi)
- Bilgi saklanması (Information Hiding)
 1. Saklı tipler özellikler kısmında aşağıdaki gibi tanımlanır:

```
type NODE_TYPE is private;
```

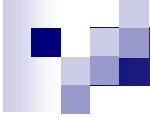
2. Paketin kullanıcılarına görünmeyen kısmında (**private** ifadesi), daha önceden açıklanan tiplerin özellikleri tanımlanır:

```
package ... is
    type NODE_TYPE is private;
    ...
    private
        type NODE_TYPE is
            record
            ...
            end record;
```



Dil Örnekleri

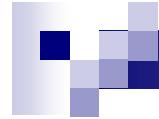
- Özellikler kısmı saklı olmayan tipleri de tanımlarının doğrudan “**private**” ifadesinden önce konulması ile tanımlayabilir.
- “**private**” tipler için atama ve karşılaştırma ön tanımlı işaretleri (= ve /=) tanımlanmıştır.
 - Limited private tipler için ön tanımlı işaretler yoktur.



Dil Örnekleri

2. C++

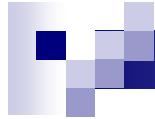
- C “**struct**” tipine ve Simula 67 sınıflarına (classes) dayanır
- “**class**” kılıflama aracıdır
- Bütün “**sınıf**” somutlaşan örnekleri (instances) sınıf fonksiyonlarının (metot) tek kopyasını paylaşırlar
- Her “**sınıf**” somutlaşan örneğinin “**sınıfin**” veri üyelerinden kendi kopyası vardır
- somutlaşan örnekler statik, yiğit dinamik veya yiğin dinamik olabilirler.



Dil Örnekleri

■ Bilgi saklama (Information Hiding):

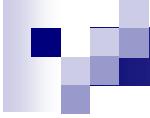
- “Private” yantümcesi saklanan elemanlar için
- “Public” yantümcesi arayüz elemanları için
- “Protected” yantümcesi kalıtım (inheritance) için (Bölüm 12)



Dil Örnekleri

■ Yapıcılar (Constructors):

- Somutlaşan örneklerin verilerini başlatan fonksiyonlar (bunlar nesne yaratamaz)
- Yığın dinamik bellek alabilir
- Nesneleri parametrize etmek için parametreleri olabilir
- “sınıfın” somutlaşan örneği yaratılırken örtülü olarak çağrılır
- Açıkça da çağrılabılır
- Adı “sınıf”的 adıyla aynı olmalıdır



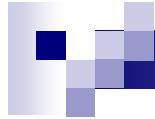
Dil Örnekleri

■ Yok ediciler (Destructors)

- Somutlaşan örneğin yaşam süresi biterken genellikle yığın bellekten alınanları geri vermek için kullanılır
- Örtülü olarak yaşam süresi sonunda çağrılır
- Açıkça da çağrılabılır
- Adı “sınıf” adının önüne tilde (~) konmuş şeklidir

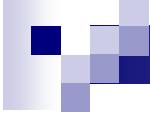
Dil Örnekleri

```
class stack {  
    private:  
        int *stackPtr, maxLen, topPtr;  
    public:  
        stack() { // a constructor  
            stackPtr = new int [100];  
            maxLen = 99;  
            topPtr = -1;  
        };  
        ~stack () {delete [] stackPtr; };  
        void push (int num) {...};  
        void pop () {...};  
        int top () {...};  
        int empty () {...};  
}
```



Dil Örnekleri

- Friend fonksiyonları ve sınıfları – bazı ilişkisiz birim veya fonksiyonların özel değerlerine ulaşmak için kullanılır.
 - C++ da gerekli
- C++ SVT desteğinin değerlendirilmesi
 - SVT açısından “sınıf”lar Ada “package” yapıları gibi
 - fark: “package” yapısı bir kılıfken “class” bir tiptir, ayrıca “class” sadece veri soyutlaması için tasarlanmamıştır.



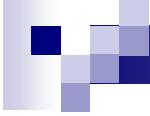
Dil Örnekleri

3. Java

- C++ gibi şu farkla ki:
 - Bütün kullanıcı tanımlı tipler “class”
 - Bütün nesneler yiğin bellekten alınıyor ve referans değişkenleri ile erişiliyor
 - “class” içindeki bütün bireysel öğelerin yantümceler yerine erişimi düzenleyen değiştiricileri (access control modifiers) (“private” veya “public”) bulunur
 - Java’da kapsamı düzenleyen ikinci bir yöntem bulunur, ”package” kapsamı
 - Bir ”package” içinde erişim düzenleyen değiştiricileri olmayan her ”class” içindeki her şey ”package” içinde görünebilir durumdadır (kapsama çevresi)

Dil Örnekleri

```
class StackClass {  
    private:  
        private int [] *stackRef;  
        private int [] maxLen, topIndex;  
        public StackClass() { // a constructor  
            stackRef = new int [100];  
            maxLen = 99;  
            topPtr = -1;  
        };  
        public void push (int num) {...};  
        public void pop () {...};  
        public int top () {...};  
        public boolean empty () {...};  
}
```



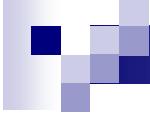
Dil Örnekleri

■ C#

- C++ ve Java'ya baz alır
- İki yeni erişim düzenleyici, “internal” ve “protected internal”
- Bütün “class” somutlaşan örnekleri yığın dinamik (Java gibi)
- Bütün sınıflar için varsayılan yapıcılar (constructor)
- Yığın bellek nesneleri için çöp toplama kullanıldığından yok ediciler (Destructors) nadiren kullanılıyor
- “structs” somutlaşan örneği (inheritance) desteklemeyen zayıf sınıflar
 - Değere göre geçilen tiplerdir
 - Yığıt bellekte tutulurlar (“class” yığın bellekte)
 - Temel veri tipleri dahil değere göre geçilen bütün tipler “structs”
- Veri üyelerine erişim için genel çözüm: Erişim metotları (veri alanlar ve veri koyanlar) (“get” edenler ve “set” edenler) (getters and setters)
- “Delphi”den esinlenmiştir
- C# alanlar ve koyanları açıkça metod çağrımadan özellik olarak düzenlemiştir.

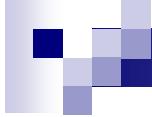
C# Property Example

```
public class Weather {  
    public int DegreeDays { /** DegreeDays özellikidir  
        get {  
            return degreeDays;  
        }  
        set {  
            degreeDays = value;  
        }  
    }  
    private int degreeDays;  
    ...  
}  
...  
Weather w = new Weather();  
int degreeDaysToday, oldDegreeDays;  
...  
w.DegreeDays = degreeDaysToday;  
...  
oldDegreeDays = w.DegreeDays;
```



Dil Örnekleri

- Alan ve koyan özelliği veriyi “public” yapmaktan aşağıdaki nedenlerle daha iyidir:
 - Sadece okuma amaçlı erişim sağlanabilir. Alan özellik tanımlanır, koyan özellik tanımlanmaya bilir.
 - Koyan özelliğe bazı sınırlamalar konulabilir. Örneğin konulacak verinin limitleri olacaksa bu tanımlanabilir.
 - Soyutlanmış veri yapısı kullananları etkilemeden kolaylıkla değiştirilebilir.



Parametrize Edilmiş SVT

- SVT parametrize etmek gerekebilir. Örneğin sayısal bir yığıt yaratılmak isteniyor olabilir. Her bir sayısal tip için ayrı ayrı SVT yaratılacağına, bu parametrize edilebilir.

1. Ada Jenerik paketleri

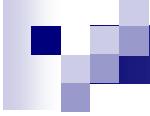
- Daha önce jenerik yordamları görmüştük; benzer şekilde jenerik paketler de olabilmektedir.
- Yığıt'ın boyunu ve tipini jenerik yaparak yığıt'ı esnek yapabiliriz.

Örnek Program

```
generic
  Max_Size : Positive; -- A generic parameter for stack
                      -- size
  type Element_Type is private; -- A generic parameter
                                -- for element type

package Generic_Stack is
  -- The visible entities, or public interface
  type Stack_Type is limited private;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
                 Element : in Element_Type);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Element_Type;
  -- The hidden part
private
  type List_Type is array (1..Max_Size) of Element_Type;
  type Stack_Type is
    record
      List : List_Type;
      Topsub : Integer range 0..Max_Size := 0;
    end record;
end Generic_Stack;
```

```
package INT_STACK is new GENERIC_STACK(100, INTEGER);
package FLOAT_STACK is new GENERIC_STACK(500, FLOAT);
```



Parametrize Edilmiş SVT

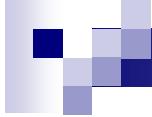
2. C++ Templatized Classes

- Sınıflar yapıcı fonksiyona parametre konulmasıyla bir miktar jenerik olabilirler
- Sınıfın “templatized class” tanımlanmasıyla jenerik yapı elde edilir.

```
stack (int size)
{
    stk_ptr = new int [size];
    max_len = size - 1;
    top = -1;
}
stack stk(100);
```

Parametrize Edilmiş SVT

```
#include <iostream.h>
template <class Type> // Type is the template parameter
class stack {
private:
    Type *stackPtr;
    int maxLen;
    int topPtr;
public:
// A constructor for 100 element stacks
    stack() {
        stackPtr = new Type [100];
        maxLen = 99;
        topPtr = -1;
    }
// A constructor for a given number of elements
    stack(int size) {
        stackPtr = new Type [size];
        maxLen = size - 1;
        topPtr = -1;
    }
~stack() {delete stackPtr;} // A destructor
void push(Type number) {
    if (topPtr == maxLen)
        cout << "Error in push-stack is full\n";
    else stackPtr[++ topPtr] = number;
}
void pop() {
    if (topPtr == -1)
        cout << "Error in pop-stack is empty\n";
    else topPtr--;
}
Type top() {return (stackPtr[topPtr]);}
int empty() {return (topPtr == -1);}
}
```



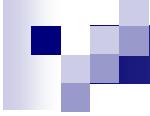
Parametrize Edilmiş SVT

■ Java 5.0

- Jenerik parametreler “class” olmalıdır.
- En genel jenerik tipler `LinkedList` ve `ArrayList` gibi toplu veri tipleridir

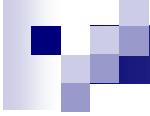
■ C#

- 2005 sürümü ile jenerik SVT desteği vermeye başlamıştır.
- Java 5.0 a benzer.



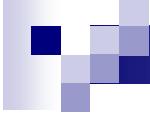
Kılıflama/Sarmalama Yapıları

- SVT tek tipli minimal kılıflama yapısıdır. Çok tipli kılıflama yapılarına büyük programlarda (birkaç bin satırı geçen) gereksinim duyulur.
- Baştaki güdüleme:
 - Büyük programların iki önemli gereksinimi bulunur:
 1. Altprogramlara bölmenin, SVT kullanmanın ötesinde bir organizasyona gidilmesi.
 2. Kısmi derleme (derlenen parçaların programın tamamından daha küçük olması, derlemenin değiştirilen parçalarla sınırlandırılması)
- Çözüm: mantıksal olarak bağlantılı altprogram grupları birlikte ve ayrı olarak derlenir (derleme birimleri)
 - Bunlara kılıflama (*encapsulations*) denir.
- Programları tasarlarken altprogramların mantıksal olarak büyük altprogramlar altında toplamak.



Kılıflama Yapıları

- Ada, Fortran 95, Python ve Ruby iç içe altprogramları desteklerler.
- C
 - Bir veya birden çok altprogram içeren dosyalar bağımsız olarak derlenir.
 - Ara yüzler **başlık (header)** dosyaları içine konur.
 - Problem: ilişkilendirici (the linker) başlık içindeki tiplerle derlenen dosyadaki tipleri karşılaştıramaz.
- C++
 - C gibi
 - Friend sınıfının özel üyelerine erişim için friend fonksiyonunun eklenmesi. İki sınıfın özel verileri arasında işlem yapılacaksa buna olanak sağlamak amacıyla kullanılır.



Kılıflama Yapıları

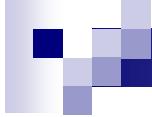
- Ada “Package” yapısı
 - İstenen sayıda veri ve altprogram tanımlaması yapılabilir, sınırlama yok.
 - İki kısımdan oluşur: özellikler ve gövde (specification and body).
 - Bu iki kısım bağımsız olarak derlenebilir.
- C# Assembly (Kurgu, montaj)
 - Tek dinamik bağlanma ile oluşmuş kütüphane veya yürütüme hazır program oluşturan dosyalar.
 - Sınıftan daha büyük bir yapı; .NET programlama dilleri tarafından kullanılır.

Adlandırma Kılıfları (Naming Encapsulations)

- Büyük programlar bir anlamda bağımsız çalışan birçok geliştirici tarafından yazılırlar. Bu mantıksal birimlerin bağımsız olmasını, bununla birlikte birlikte çalışabilmelerini gerektirir.
- Büyük programlar birçok global isim tanımlarlar, değişik grupların tanımladığı isimlerin çakışması tehlikesi vardır; bunların mantıksal gruplara bölünmesi gerekebilir.
- Isimlere yeni bir kapsam yaratmak için **Adlandırma Kılıfları** kullanılır.
- C++ Namespace (alan adı)
 - Her kütüphane kendi “namespace” içinde olabilir ve dışarıdan alan adı ile erişilir.

```
Namespace MyStack {  
    // Stack tanımları }  
    MyStack::topPtr           //başka bir paketten kullanımı...
```

- C# da aynı yapıyı kullanır



Adlandırma Kılıfları

■ Java Package

- Birden çok sınıf tanımı içerebilir; bu sınıflar kısmen “friend” (aynı package içinde ve “public” ve “protected” olanlar) olabilirler.
- “package myStack;” örneği gibi, tanımlaması dosyanın başında olmalıdır.
- Kullanıcıları pakette tanımlanan isimlere dışarıdan tam adları ile (myStack.topPtr gibi) veya diğer paket başına “import myStack.*;” komutu eklenmişse sadece değişken ismiyle erişebilirler.

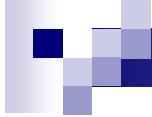
■ Ada Package

- Paketlerin tanımlama hiyerarşileri dosya hiyerarşileri ile sağlanır. subPack, Pack paketinin alt paketi ise, subPack dosyasının Pack dosyasının bulunduğu dizinin alt dizininde olması gereklidir.
- Bir program biriminde bir paketin isimlerinin görünebilmesi, o paketin adının **with** yan tümcesinde geçmesi ile elde edilir:

```
with Ada.Text_IO;
```

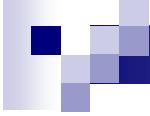
Bölüm 12

Nesneye Yönelik Programlama (NYP)
Object-Oriented Programming (OOP)



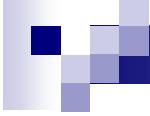
Bölüm 12 Başlıklar

- Giriş
- Nesneye Yönelik Programlama (NYP)
- Nesneye Yönelik Programlama Dillerinde Tasarım Konuları
- C++ Nesneye Yönelik Programlama
- Java Nesneye Yönelik Programlama
- C# Nesneye Yönelik Programlama



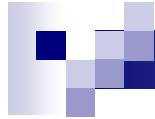
Giriş

- NYP destekleyen dillerin kategorileri:
 1. NYP desteği eklenmiş diller
 - C++ (ayrıca yordamsal ve veriye yönelik programlamayı da destekler)
 - Ada 95 (ayrıca yordamsal ve veriye yönelik programlamayı da destekler)
 - CLOS (Lisp NYP sürümü 1988) (ayrıca fonksiyonel programlamayı da destekler)
 - Scheme (ayrıca fonksiyonel programlamayı da destekler)



Giriş

2. NYP desteklenir, fakat buyurgan dillerin tipik özellikleri de bulunur
 - Eiffel (Daha önceki herhangi bir dile dayanmaz)
 - Java (C++ kaynaklı)
3. Saf NYP Dilleri
 - Smalltalk 80

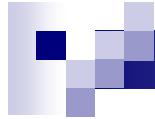


Paradigma Evrimi

■ Paradigma Evrimi (Paradigm Evolution)

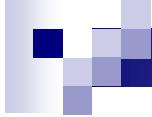
(Paradigma: Bir süreç ya da model için esas olan örnek ya da örüntü)

1. Yordamsal (Procedural) - 1950-1970 (yordamsal soyutlama)
2. Veriye Yönelik (Data-Oriented) - erken 1980ler (Veri soyutlaması)
3. NYP – geç 1980'ler (SVT, Kalıtım (inheritance) ve dinamik bağlama)



Nesneye Yönelik Programlama

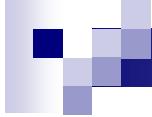
- Kalitimın başlangıcı
 - 1980lerin ortalarından gözlemler :
 - Yazılımcılar üzerinde üretkenliğin artması yönünde büyük baskı vardı.
 - Üretkenlik artışı tekrar kullanımıyla elde edilebilirdi.
 - Ne yazık ki,
 - Soyut veri tiplerinin (SVT) yeni problemlerde tekrar kullanılması zor.
 - Bütün SVT bağımsız ve aynı seviyede.
 - Kalitim bu iki problemi de çözer-SVT'yi küçük değişikliklerle tekrar kullan ve sınıfları (class) hiyerarşî ile tanımla.



Nesneye Yönelik Programlama

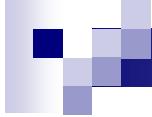
■ NYP Tanımları:

- SVT'lere **class** (sınıf) denir.
- “Class” somutlaşmış örneklerine **objects** (nesne) denir.
- Kalıtımıla başka bir sınıfın özelliklerini alan “class”a **derived class** (türetilmiş sınıf) veya **subclass** denir.
- Kalıtımıla başka bir sınıfa özelliklerini veren “class”a **parent class** (ebeveyn sınıf) veya **superclass** (süpersınıf) denir.
- Nesneler üzerinde yapılacak işlemleri tanımlayan altprogramlara **metot** denir.
- Metotların çağırılmasına **message** (mesaj) denir.
- Bir nesnenin bütün metot setine nesnenin **message protocol** (mesaj protokolü) veya **message interface** (mesaj arayüzü) denir.
- Temelde bir sınıf ebeveyninin bütün öğelerini kalıtımıla alır (In the simplest case, a class inherits all of the entities of its parent)



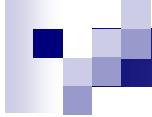
Nesneye Yönelik Programlama

- Kilitlenmiş ögelerere erişim kontrolü ile kalıtım zor olabilir.
 - Bir sınıf alt sınıflarından öğelerini saklayabilir.
 - Bir sınıf öğelerini kullanıcılarından saklayabilir.
 - Bir sınıf öğelerini kullanıcılarından saklarken alt sınıflarına açabilir.
- Metotların kalıtımıla alınması yanında, bir sınıf kalıtımıla alınan bir metodu değiştirebilir.
 - Yeni olan kalıtımıla geleni [overrides](#) cœurüterek üstüne çıkar.



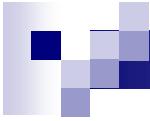
Nesneye Yönelik Programlama

- Bir sınıfta iki tip değişken bulunur:
 1. Sınıf değişkeni (**Class variables**) – bir/sınıf
 2. Somutlaşan örnek değişkeni (**Instance variables**) – bir/nesne
- Bir sınıfta iki tip metot bulunur:
 1. Sınıf metodu(**Class methods**) – sınıfa gelen mesajları kabul eder
 2. Somutlaşan örnek değişkeni (**Instance methods**) – nesnelere gelen mesajları kabul eder
- Tek'e karşı çoklu kalıtım
- Kalıtımın tekrar kullanım için dezavantajı:
 - Bakımı zorlaştıran karşılıklı bağımlılıklar yaratır.



Nesneye Yönelik Programlama

- Bir **abstract method** (**soyut metot**) protokolü (parametreleri, döndüğü değer) tanımlı ve fakat kendisi tanımlı olmayan sınıfır.
- Bir **abstract class** (**soyut sınıf**) en az bir sanal metot barındıran sınıfır.
- Soyut sınıf somutlaştırılamaz.



C++

■ Genel özellikler:

- Karışık tipli sistem: C'den türetildiğinden, onun özellikleri korunmak istenmiştir. Bu nedenle hem buyurgan dillerin tiplerini hem de sınıf yapısını destekler.
- Yapıcılar ve yok ediciler (constructors/destructors).
- Sınıf öğelerine ayrıntılı erişim kontrolü

■ Kalıtım

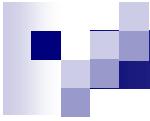
- Bir sınıf bir başka sınıfın alt sınıfı olmak zorunda değil.
- Üyelerin erişim hakları
 1. "Private" (sadece "class" ve "friends" erişebilir)
 2. "Public" (altsınıflardan ve kullanıcılarından erişilebilir)
 3. "Protected" (sınıf içinden veya altsınıflardan erişilebilir fakat kullanıcılar erişemez)

Example

```
class base_class {
    private:
        int a;
        float x;
    protected:
        int b;
        float y;
    public:
        int c;
        float z;
};

class subclass_1 : public base_class { ... };
// - burada, b ve y protected ve
//      c ve z public

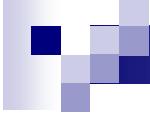
class subclass_2 : private base_class { ... };
// - burada, b, y, c, ve z private,
//      ve türetilen bir sınıf ebeveyn sınıfına erişemez.
// örneğin "c"ye erişilebilmesi için aşağıdaki yöntem kullanılır
class subclass_3 : private base_class {
    base_class :: c;
}
```



C++

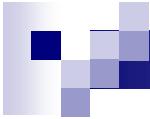
■ Değerlendirme

- gelişmiş erişim kontrolü sağlar
- çoklu kalıtım mümkün
- Programcı metodların statik veya dinamik bağlanmasına tasarım aşamasında karar vermelidir.
 - Statik bağlanma daha hızlı!



Java

- C++'a çok benzediğinden bazı farklılıklardan bahsedelim
- Genel özellikler
 - Temel tipler hariç her türlü veri nesne. Sadece boolean, character ve sayısal tipler nesne değil. Bu durum kimi zaman problemler de yaratır. Örneğin ArrayList sınıfının somutlaşmış örneği “array”dir ve eleman olarak sadece nesneler alır ve bu durumda örneğin sayılar doğrudan yerleştirilemez. Bu durum Java 5.0 a kadar`myArray.add(new Integer(10));`şeklinde çözülürken, 5.0'dan sonra örtülü olarak düzenlenmiştir`myArray.add(10);`
 - Bütün nesneler yığın dinamik, referans değişkenlerle erişilebilir ve `new` ile üretilir. Açık bir silici yok, çöp toplama ile işi biten nesneler sisteme iade ediliyor.
 - “finalize” metodu çöp toplamadan önce örtülü olarak çağrılarak bazı başka temizlik işleri yaptırılabilir. Problem ne zaman çalışacağıının belli olmaması.
 - Bütün sınıflar bir sınıfın alt sınıfı olmak zorunda. En üstteki sınıfın adı “**Object**”.



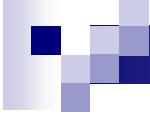
Java

■ Kalıtım (Inheritance)

- Tek kalıtım desteklenir. Ancak, “interface” isimli soyut sınıfla çoklu kalıtima destek sağlanır. Bu sınıfın tipik kullanımı ebeveyinden kimi metod ve özelliklerini alarak bunlara ebeveyne arayüz hazırlamaktır.
- Bir “interface” sadece isimli sabitler ve metod tanımlamaları içerebilir. Örneğin Java “Arrays” sınıfı dizilik sıralama metodunu içerir. Bunun için aşağıdaki gibi bir arayüzün tanımlanması gereklidir. Daha sonra “CompareTo” metodunu tanımlayan her sınıf bu metoda göre sıralama yapabilir.

```
public interface Comparable {  
    public int compareTo(Object b);  
}
```

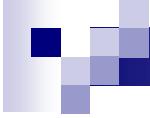
- Metotlar **final** tanımlanabilir. Bu tanım, metodun takip eden kalıtımsal takipçilerinde değiştirilemeyeceğini belirler. Eğer sınıf tanımında kullanılırsa, bu sınıfın hiçbir sınıfın ebeveyni olamayacağını gösterir.



Java

■ Dinamik bağlanma

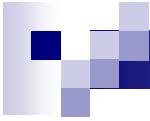
- Bütün mesajlar (metot çağrıları) metotlara dinamik olarak bağlıdır. Bunun istisnası **final** metotlardır çünkü bunlar tekrardan tanımlanamayacakları için dinamik bağlanmanın bir anlamı yoktur.



Java

■ Kilitlama

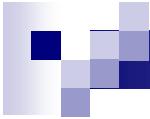
- İki şekilde yapılır, class ve package
- Package ilişkili sınıfları birleştirir (isimli veya isimsiz olabilir)
- Bir kapsamda tanımlanmayan bütün öğeler isimsiz “package” yapısı altında tanımlanmış sayılırlar, bu paket içinde görünebilirdirler.
 - Paket içindeki her sınıf paket içindeki diğer öğelerin “friend”idir.
 - Bu nedenle paket kapsamı C++ friend uygulaması ile benzeşir.



C#

■ General characteristics

- NYP desteği Java gibidir.
- class ve struct yapılarını destekler
- Sınıflar Java sınıflarına benzer



C#

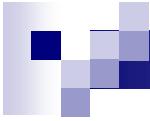
■ Kalıtım

- Sınıf tanımlamak için C++ sözdizimi kullanılır

```
public class NewClass : ParentClass { ... }
```

- Ebeveynden kalıtım kalmış metod, türetilmiş sınıfta tanımı **new** ile işaretlenerek değiştirilebilir.
- Ebeveyn sınıfı metodu hala açıkça aşağıdaki örnekte olduğu gibi çağrılabılır:

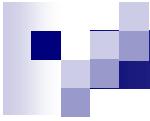
```
base.Draw();
```



C#

■ Dynamic binding

- Metotların dinamik bağlarla çağrılabilmesi için:
 - Temel sınıf metodu “**virtual**” işaretlenir.
 - Türetilmiş sınıflardaki metotlar ise “**override**” tanımlanırlar.
- Soyut metotlar “**abstract**” olarak işaretlenmeli ve bütün altsınıflarda tanımlanmalıdır.
- Bütün C# sınıfları **Object** sınıfından türetilmiştir.



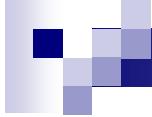
C#

■ Değerlendirme

- C# en yeni C tabanlı nesneye yönelik programlama dilidir.
- C# ve Java arasında NYP açısından çok az fark bulunur.

Bölüm 14

14 Ayrılıkların (İstisnaların) ve Olayların yönetilmesi
(Exception Handling and Event Handling)



Ayrılıkların (exception) yönetilmesi

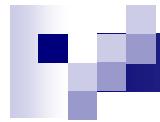
■ Tanım: Ayrılık (exception)

- bir hatadan veya değil,
- donanım veya yazılım tarafından algılanabilen (detectable)
- ve özel işlem gerektirebilen
- beklenmedik (ne zaman olacağı bilinmeyen) olaydır (event).

■ Tanım: Ayrılığın algılanmasından sonra gerekebilen özel işleme **ayrılık yönetilmesi** (exception handling) denir.

■ Tanım: Ayrılığı yöneten koda, **ayrılık yöneticisi/kotarıcısı** (exception handlers) denir.

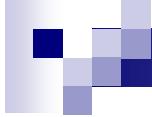
■ Tanım: **Ayrılık**, bağlı olay (event) gerçekleşince, **yürütülür**.



Ayrılıkların (exception) yönetilmesi (handling)

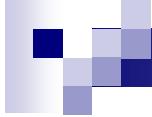
- Ayrılıkların yönetilmesi özelliği olamayan bir dilde ayrılık durumu meydana geldiğinde kontrol işletim sistemine geçer, sistem programdan gelen son mesajlara veya kendi belirlemelerine göre bir mesaj yazarak programı sonlandırır.
- Ayrılıkların yönetilmesi özelliği olan bir dilde bu tip ayrılıkların yakalanması, programının öngördüğü şekilde yönetilmesi ve programın sonlandırılmadan sürdürülmesi mümkündür.
- Hemen bütün diller girdi/çıktı'da oluşan hataları (dosya sonu dahil) yakalayabilirler.
- İlk örnek: Fortran:

```
Read (Unit=5, Fmt=1000, Err=100, End=999) Weight
```



Ayrıılıkların (exception) yönetilmesi

- Ayrılık yönetme kapasitesi olmayan bir dil bile bir ayrılığı tanımlayıp, fark edip, yürütüp, yönetebilir.
- Seçenekler:
 1. Ek bir parametre ile veya altprogramın döndüğü değer ile. Bu durumda çağrıran altprogram ayrılık durumunda kullanılacak kısma geçer.
 2. Etiket (label) parametresinin bütün alt programlara geçilmesi. Altprogram bir etiket dönerse, çağrıanda bu etiketin altındaki kod çalıştırılır. Bunun için programlama dilinin parametrelerden etiket geçilmesini desteklemesi gereklidir.
 3. Ayrıılıkları yöneten altprogramı bütün altprogramlara parametre olarak geçmek. Her altprograma, gerekip gerekmeyeceğine bakmaksızın geçilmesi zorluk yaratır. Ayrıca farklı tipli ayrıılıklar için farklı altprogramların geçilmesi gereklidir, bu da kodda anlaşılabilirliği azaltır.



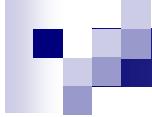
Ayrılıkların (exception) yönetilmesi

■ Yerleşik ayrılık yöneticisinin bulunmasının avantajları:

1. Hata fark edici kodların yazılması sıkıcıdır ve programı daha karışık yapar. Örneğin bir dizimedde indeksin limitler içinde olduğunu kontrol etmek istersek:

```
if (row>=0 && row < 10 && col>=0 && col<20)
    sum += mat [row] [col];
else
    System.out.println("mat indeksi küme dışı");
```

2. Ayrılık yönetiminin bulunması programcılar bütün olası olaylara karşı kod yazmada teşvik eder. Kendi başına önemsemeyebileceği olayları, programlama dilinde olduğu için dikkate alıp gerekli kodları yazmasını sağlar.
3. Bir diğer faydası ayrılıkların yayılımında (exception propagation) ortaya çıkar. Belli ayrılıklar için hazırlanan ayrılıklar yönetme kodları programın farklı kısımlarında benzer ayrılıklar için kullanılabilir. Böylelikle farklı kısımlar için ayrı kodlama yapma gereksinimi ortadan kalkar. Bu da programın karmaşıklığını, maliyetini ve boyutlarını azaltır.

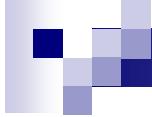


Ayrılıkların (exception) yönetilmesi

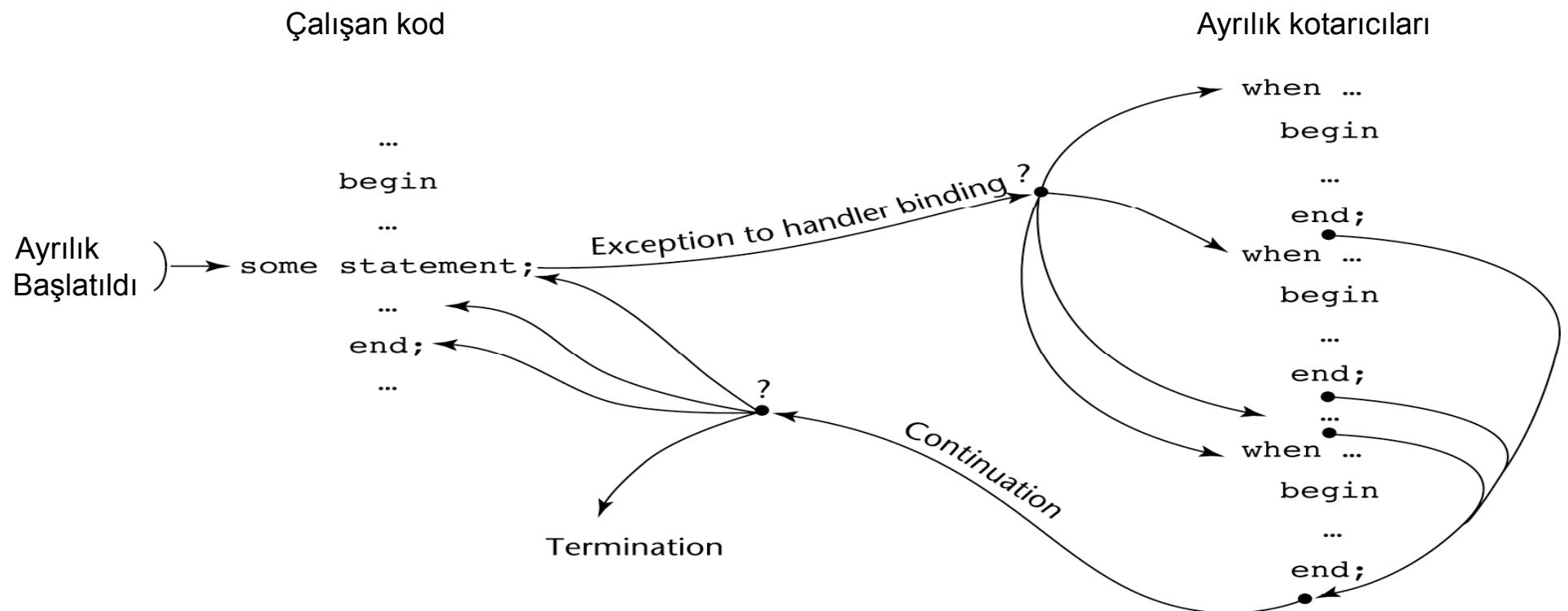
■ Ayrılıkların yönetilmesinde tasarım etmenleri

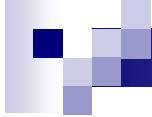
Öncelikle aşağıdaki iskelet programa bakalım:

```
procedure örnek()
begin
...
    ortalama = toplam / N;
...
    return;
/* ayrılık yönetici kısmı */
when zero_divide begin
    ortalama = 0;
    printf("sıfır ile bölün hatası\n");
end;
end;
```



Ayrılıkların (exception) yönetilmesi

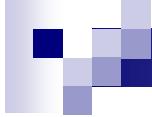




Ayrılıkların (exception) yönetilmesi

■ Ayrılıkların yönetilmesinde tasarım etmenleri

1. Ayrılık kotarıcıları nasıl ve nerede belirlenecek ve kapsamı ne olacak?
2. Ayrılık gerçekleşmesi, ayrılık kotarıcısına nasıl bağlanacak?
3. Ayrılıkla ilgili bilgi kotarıcıya geçilebilir mi?
4. Ayrılık kotarıcısı işini bitirdikten sonra program nereden devam edecek?
5. Bir şekilde sonlandırma sağlanacak mı?
6. Kullanıcı tanımlı ayrılıklar nasıl belirlenecek?
7. Ön tanımlı ayrılıklar varsa, kendi ayrılıklarını yönetme kısımları olmayan programlarda bunlar varsayılan olarak kullanılmalı mı?
8. Ön tanımlı ayrılıklar açıkça yürütülebilmeli mi?
9. Donanım tarafından algılanabilen hatalar yönetilebilecek ayrılıklar olabilir mi?
10. Ön tanımlı ayrılıklar olacak mı?
11. Ön tanımlı ayrılıklar (çok zaman aldıklarından) kapatılabilmeli mi?

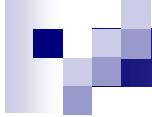


Ada'da Ayrılıkların yönetilmesi

- Ayrılık yöneticisinin çerçevesi altprogram gövdesi veya paket gövdesi veya görev veya blok'dur.
- Ayrılık yöneticileri ayrılığın olabileceği kod'la aynı kapsamda olduklarından (lokal), parametreleri olamaz.
- Yönetici şekli:

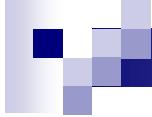
```
begin
    --blok veya birim gövdesi. Komutlar...
exception
    when exception_name { | exception_name } =>
        statement_sequence
    ...
    when ...
    ...
    [when others =>
        statement_sequence ]
end;
```

- Ayrılık yöneticisi altprogram veya birimin sonuna eklenir.



Ada'da Ayrılıkların yönetilmesi

- Ayrılıkları (exceptions) yöneticilere (handlers) bağlamak:
- Eğer ayrılık ortaya çıkan bir blokta veya birimde bu ayrılık için yönetici yoksa ayrılık yönetilmek üzere bir yere yayılır.
 1. Altprogram: çağrıvana geçir.
 2. Blok: bir üst statik kapsayıcı büyük bloğa, bloktan hemen sonra olmak üzere geçir.
 3. Paket gövdesi: paketin tanımlama kısmına geçir. Eğer paket statik ebeveyni olmayan kütüphane birimi ise program sonlanır.
 4. Görevse (task): Yayılma yok. Eğer ayrılık yöneticisi varsa onu yürüt. Her durumda "bitti" diye işaretle.
 5. Ayrılığın çıktığı birim veya blok (veya geçirildiği blok veya birim) eğer ayrılığı yönetemezse derhal sonlandırılır.



Ada'da Ayrıılıkların yönetilmesi

- Ön tanımlı ayrıılıklar (Standard paket içinde):
 - CONSTRAINT_ERROR – Kısıt hatası: indeks kısıtları, kapsam kısıtları, vs...
 - PROGRAM_ERROR – gövdesi hazırlanmamış altprograma çağrı.
 - STORAGE_ERROR – yığın bellek (heap) tükendi.
 - TASKING_ERROR – görevlerle (task) ilgili hata.
 - DATA_ERROR – veri girişi hatası.
- Kullanıcı tanımlı ayrıılıklar:
 - <ayrılık_isim_listesi> : exception;
 - raise [ayrılık_ismi]
 - Eğer bir ayrılık yöneticişi içinde kullanılacaksa ayrılık_ismi gerekmez.
 - Aynı ayrılık tekrardan başlatılır.
- Örnek sonraki sayfada

```
...
type Age_Type is range 0..125;
type Age_List_Type is array (1..4) of Age_Type;
package Age_IO is new Integer_IO (Age_Type);
use Age_IO;
Age_List : Age_List_Type;
...
begin
for Age_Count in 1..4 loop
    loop -- loop for repetition when exceptions occur
    Except_Blk:
        begin -- compound to encapsulate exception handling
            Put_Line("Enter an integer in the range 0..125");
            Get(Age_List(Age_Count));
            exit;
        exception
            when Data_Error => -- Input string is not a number
                Put_Line("Illegal numeric value");
                Put_Line("Please try again");
            when Constraint_Error => -- Input is < 0 or > 125
                Put_Line("Input number is out of range");
                Put_Line("Please try again");
        end Except_Blk;
    end loop; -- end of the infinite loop to repeat input
               -- when there is an exception
end loop; -- end of for Age_Count in 1..4 loop
...

```

```

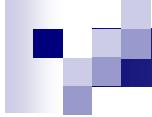
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Grade_Distribution is
    Freq: array (1..10) of Integer := (others => 0);
    New_Grade,
    Index,
    Limit_1,
    Limit_2 : Integer;
begin
    loop
        Get(New_Grade);
        Index := New_Grade / 10 + 1;
        begin -- A block for the Constraint_Error handler
            Freq(Index) := Freq(Index) + 1;
        end;
    end loop;
end Grade_Distribution;

```

```

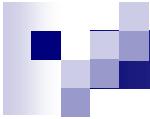
exception
when Constraint_Error =>
    if New_Grade = 100 then
        Freq(10) := Freq(10) + 1;
    else
        Put("ERROR -- new grade: ");
        Put(New_Grade);
        Put(" is out of range");
        New_Line;
    end if;
end; -- end of the Constraint_Error handler
end loop;
exception -- Handler for all final computations
when End_Error =>
    Put("Limits Frequency");
    New_Line; New_Line;
    for Index in 0..9 loop
        Limit_1 := 10 * Index;
        Limit_2 := Limit_1 + 9;
        if Index = 9 then
            Limit_2 := 100;
        end if;
        Put(Limit_1);
        Put(Limit_2);
        Put(Freq(Index + 1));
        New_Line;
    end loop; -- for Index in 0..9 ...
end Grade_Distribution;

```



Ada'da Ayrılıkların yönetilmesi

- Ayrılık yönetilmesi kontrollü olarak kapatılabilir:
 - pragma Suppress(ayrılık_listesi)
 - örneğin, Index_Check, Division_Check, vs.
- Değerlendirme
 - Ayrılıkların yönetilmesi Ada'ya 1980 yılında çok başarılı şekilde girmiştir.
 - Daha sonra C/C++'da ekleninceye kadar Ada bu konuda tekti.



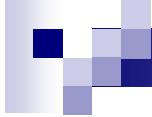
C/C++'da Ayrıılıkların yönetilmesi

- 1990'da eklendi.
- Tasarım CLU, Ada, ve ML'ye dayanır.
- Ada'dan farkı ön tanımlı ayrıılıkların olmamasıdır.
- Ayrıılık Yöneticileri (Exception Handlers)

- Şekil:

```
try {  
    -- ayrıılık yaratması beklenen kod  
}  
catch (<resmi parametre>) {  
    -- yönetici kodu  
}  
...  
catch (<resmi parametre>) {  
    -- yönetici kodu  
}
```

- "catch" bütün yöneticilerin adıdır – fazla yüklü bir isim olduğundan parametresinin özel olması gereklidir.
- Resmi parametrede bir değişken olması gerekmeyez; bir değişken tipi de (float gibi) olabilir. Bu durumda parametre yöneticiyi başkalarından ayırmaya yarar.
- Parametre yöneticiye bilgi aktarmak için kullanılabilir.
- Resmi parametre üç nokta olabilir. Bu durumda henüz yönetilmemiş ayrıılıkları alır.



C/C++'da Ayrıılıkların yönetilmesi

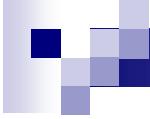
■ Ayrıılıkların yöneticilere bağlanması:

- Ayrıılıklar aşağıdaki ifade ile açıkça başlatılır (köşeli parantez içinde seçmeli parametre):

```
throw [ifade];
```
- Parametresi olmadan "throw" sadece kotarıcı içinde kullanılabilir ve kotarıcıyı tekrardan başlatır. Bu durumda kotarıcı başka bir yerde başlar (Ada'daki gibi).
- "ifade"nin tipi hangi kotarıçının kullanılacağına dair belirsizliği kaldırır.
- O seviyede yönetilmeyen ayrıılıklar çağrıran altprogram zincirinde aranır.
- Bu arama "main" fonksiyonuna kadar sürer.
- Eğer hiçbir yerde kotarıcı bulunamazsa, varsayılan kotarıcı çağrıılır.

■ Devamlılık

- Kotarıcı yürütmemeyi bitirdikten sonra kontrol son kotarıçıdan sonraki ilk ifadeye verilir.



C/C++'da Ayrıılıkların yönetilmesi

- Bütün ayrıılıklar kullanıcı tanımlıdır (veya kütüphane fonksiyonlarında).
- Varsayılan kotarıcı "unexpected" sadece programı sonlandırır.
 - Varsayılan kotarıcı bir ayrılık atılıp hiçbir kotarıcı tarafından yakalanılmamışsa çağrıılır.
 - "unexpected" kullanıcı tanımlı bir fonksiyonla değiştirilebilir. Bu fonksiyon void döner ve parametresi olmaz.
 - Adının "set_terminate" değişkenine atanması gereklidir.
- Fonksiyonların tanımlanırken hangi tip ayrıılıkları atacakları baştan belirlenebilir:
 - Örnek: int fun() throw (int, char *) { . . . }
 - Baştan belirlenmemişse her türlü ayrılık atabilir (the throw clause).
- Değerlendirme
 - Ne yazık ki ayrıılıklar isimsizdir ve ayrıca donanım ve sistem yazılımı kaynaklı ayrıılıklar yönetilemez.
 - Ayrıılıkları kotarıcılara parametrelerin tipinden bağlamak okunabilirliği zorlaştırır.

```

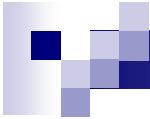
#include <iostream.h>
void main() { /* Any exception can be raised
    int new_grade,
        index,
        limit_1,
        limit_2,
        freq[10] = {0,0,0,0,0,0,0,0,0,0};
    short int eof_condition;
    try {
        while (1) {
            if (!(cin >> new_grade)) /* When cin detects eof,
                throw eof_condition; /* raise eof_condition
            index = new_grade / 10;
            {try {
                if (index < 0 || index > 9)
                    throw new_grade;
                freq[index]++;
            } /* end of inner try compound
            catch(int grade) { /* Handler for index errors
                if (grade == 100)
                    freq[9]++;
                else
                    cout << "Error -- new grade: " << grade
                        << " is out of range" << endl;
            } /* end of catch(int grade)
        } /* end of the block for the inner try-catch pair
    } /* end of while (1)
} /* end of outer try compound

```

```

catch(short int) { /* Handler for eof
    cout << "Limits    Frequency" << endl;
    for (index = 0; index < 10; index++) {
        limit_1 = 10 * index;
        limit_2 = limit_1 + 9;
        if (index == 9)
            limit_2 = 100;
        cout << limit_1 << limit_2 << freq[index] << endl;
    } /* end of for (index == 9)
} /* end of catch (short int)
} /* end of main

```

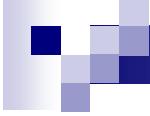


longjmp ve setjmp

- ANSI-C içinde tanımlı birçok fonksiyona ek olarak bu iki garip fonksiyon da tanımlanmıştır.
- Bunlar **longjmp** ve **setjmp**'dır.
- **longjmp** ve **setjmp** *setjmp.h* header dosyasında tanımlanmışlardır.

```
#include <setjmp.h>
```

- **Tanımları:** `int setjmp(jmp_buf env);`
`void longjmp(jmp_buf env, int val);`
- **setjmp jmp_buf** tipi değişkeni tek parametre olarak alır ve doğrudan çalıştırıldığsa 0 döner. Buna karşılık **longjmp** "env" değişkeni ile çağrırlıysa, kontrol daha önceki "env" ile çağrılan **setjmp** noktasına döner, ancak bu kez **setjmp** 0 değil "val" değerini döner.



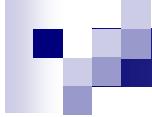
setjmp ve longjmp ile throw ve catch

```
#include <stdio.h>
#include <setjmp.h>

#define TRY    do{ jmp_buf ex_buf__; if( !setjmp(ex_buf__) ){
#define CATCH   } else {
#define ETRY    } }while(0)
#define THROW  longjmp(ex_buf__, 1)

int
main(int argc, char** argv)
{
    TRY
    {
        printf("In Try Statement\n");
        THROW;
        printf("I do not appear\n");
    }
    CATCH
    {
        printf("Got Exception!\n");
    }
    ETRY;

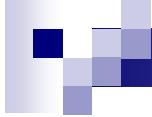
    return 0;
}
```



lisp'de örnek fonksiyon

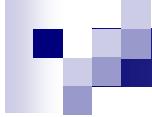
```
void zurwelt()
{
char u = 0;
struct errortrap { jmp_buf errsave;
    int stacksave;
    PALISTENT curalist;
    unsigned msgprint : 1;
    unsigned backtrace : 1; } ;

curtrap->msgprint = curtrap->backtrace = 1;
curtrap->curalist = alisttop;
curtrap->stacksave = zstackptr;
setjmp(curtrap->errsave);
if(u) zysuicide(4);
Read1();
if(!fixp(reg1)) zysuicide(2);
if(ursize != (unsigned int) intval(reg1)) zysuicide(1);
u = 1;
if(ursize)
{ for (i=0; i<ursize; i++)
  { Read1();
    urwelt[i] = reg1; }
}
curtrap = trap;
setjmp(curtrap->errsave);
if(u>1) zysuicide(5);
u++;
Read1();
if (!null(reg1)) zysuicide(3);
}
```



Java'da Ayrıılıkların yönetilmesi

- C++'a dayanır fakat daha çok Nesneye Yönelik Programlamayla (NYP) uyumludur.
- Bütün ayrıılıklar "Throwable class"ının soyundan gelen (descendants) sınıfların nesneleridir.
- Java kütüphanesi "Throwable" sınıfının iki alt sınıfını barındırır :
 1. Error
 - Java yorumlayıcısı (JVM) tarafından yiğin belleğin tükenmesi gibi bazı olaylarda atılır.
 - Kullanıcı programları tarafından atılmaz ve yakalanmazlar.
 2. Exception
 - Kullanıcı tanımlı ayrıılıklar kural gereği bu sınıfın alt sınıflarıdır (subclasses).
 - İki tane önceden tanımlı alt sınıfı bulunur: IOException ve RuntimeException. (Çok atılan ArrayIndexOutOfBoundsException ve NullPointerException, RuntimeException tarafından yakalanır).



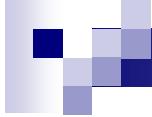
Java'da Ayrılıkların yönetilmesi

■ Java Ayrılık kotarıcıları

- C++'daki gibi , ancak ayrıca her "catch" "Throwable" sınıfının soyundan gelen bir isimli parametre ister.
- "try"ın sözdizimi C++ ile aynıdır.
- Ayrılıklar C++'daki gibi "throw" ile atılır.
- Çoğunlukla "throw" "new" ile yeni bir nesne yaratarak atar
`throw new MyException();`

■ Ayrılıkları kotarıcılara bağlamak (Binding Exceptions to Handlers)

- Ayrılıkları kotarıcılara bağlamak Java'da C++'dan daha kolaydır.
- Bir ayrılık, parametresi atılan nesneyle aynı sınıfından veya onun atası olan sınıfından olan kotarıcıya bağlanır.
- Bir ayrılık kotarıcı tarafından yakalandıktan sonra tekrar atılabilir. Tekrar atılırken farklı bir ayrılık kullanılabilir.



Java'da Ayrıılıkların yönetilmesi

- Eğer "try" yapısı içinde atılan nesne için bir kotarıcı (catch) bulunmazsa, onu kapsayan bir dış "try" yapısı aranır.
- Eğer metot içinde kotarıcı bulunmazsa, ayrılık çağrıran metoda geçer.
- Eğer "main"e kadar hiç kotarıcı bulunamaz ise program sona erer.
- Bütün ayrıılıkların yakalandığından emin olmak için en dışarıya özel bir yakalayıcı eklenebilir:

```
catch (Exception genericObject) {
```

```
    . . .
```

```
}
```

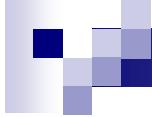
- Sınıf isimleri kendileriyle veya takip eden sınıflarla uyuştuğu için, Exception'dan üretilen tüm sınıflar bu "catch"e yakalanırlar.
- Elbette son "try" olmalıdır.

- Java "throws" komutu C++'dan sözdizimi aynı olduğu halde anlam olarak farklıdır. Java "throws"unda kullanılan ayrılık sınıf ismi, ayrılık sınıfı veya onun altındaki bir sınıfın atılabilceğini gösterir.

```
class MyException extends Exception {  
    public MyException() {}  
    public MyException(String message) {  
        super (message);  
    }  
}
```

```
throw new MyException();
```

```
MyException myExceptionObject = new MyException();  
...  
throw myExceptionObject;
```



Java'da Ayrıılıkların yönetilmesi

- Error ve RunTimeException sınıfları ve bunların altındaki bütün sınıflar kontrol edilmeyen ayrıılıklar olarak adlandırılırlar (unchecked exceptions).
- Bunların dışındakilerin hepsi kontrollü ayrıılıklardır (checked exceptions).
- Kontrollü ayrıılıklar aşağıdaki özellikleri olan metodlardan atılmalıdır (thrown)
 1. "throws" komutu içerisinde listelenmelidir, veya
 2. Bir metod içinde yönetilmelidir (Handled in the method).
- Başka bir metodun yerine geçen, onu değiştiren bir metod, throws yanıtumcesinde (clause), değiştirdiği metottan daha fazla ayrılık tanımlayamaz.
- Bir metodu çağrıran başka bir metod, eğer çağrılanın throws yanıtumcesinde listelenmiş ayrıılıklar varsa, bunlarla ilgilenmek için üç seçeneği bulunur:
 1. Yakalamak ve yönetmek.
 2. Yakalamak ve kendi listesinde bulunan başka bir ayrılık atmak.
 3. Kendiside aynı şekilde tanımlamak ve bunları yönetmemek.

```

void buildDist() throws IOException {
    // Input: A list of integer values that represent
    //         grades, followed by a negative number
    // Output: A distribution of grades, as a percentage for
    //         each of the categories 0-9, 10-19, ...,
    //         10-100.
    DataInputStream in = new DataInputStream(System.in);
    try {
        while (true) {
            System.out.println("Please input a grade");
            newGrade = Integer.parseInt(in.readLine());
            if (newGrade < 0)
                throw new NegativeInputException();
            index = newGrade / 10;
            try {
                freq[index]++;
            } //** end of inner try clause
            catch(ArrayIndexOutOfBoundsException) {
                if (newGrade == 100)
                    freq [9]++;
                else
                    System.out.println("Error - new grade: " +

```

```

import java.io.*;

// The exception definition to deal with the end of data
class NegativeInputException extends Exception {
    public NegativeInputException() {
        System.out.println("End of input data reached");
    } //** end of constructor
} //** end of NegativeInputException class

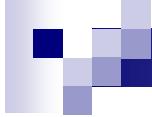
class GradeDist {
    int newGrade,
        index,
        limit_1,
        limit_2;
    int [] freq = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

```

```

        newGrade + " is out of range");
    } //** end of catch (ArrayIndex...
} //** end of while (true) ...
} //** end of outer try clause
catch(NegativeInputException) {
    System.out.println ("\nLimits      Frequency\n");
    for (index = 0; index < 10; index++) {
        limit_1 = 10 * index;
        limit_2 = limit_1 + 9;
        if (index == 9)
            limit_2 = 100;
        System.out.println(" " + limit_1 + " - " +
                           limit_2 + "      " + freq [index]);
    } //** end of for (index = 0; ...
} //** end of catch (NegativeInputException ...
} //** end of method buildDist

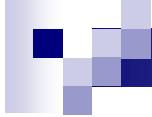
```



Java'da Ayrılıkların yönetilmesi

- finally yan tümcesi
- "try" yapısının sonunda olabilir.
- Amacı: Try içinde ne olursa olsun yürütülecek kodu belirlemek için kullanılır.
- Aşağıdaki örnekte:
 - eğer "try" içinde bir "throw" olmazsa "finally" içindeki kod da devamında çalıştırılır,
 - eğer "try" içinde "throw" olur da "catch" ile yakalanırsa, önce "catch" kodu daha sonra "finally" içindeki kod çalıştırılır,
 - eğer "try" içinde "throw" olur da "catch" ile yakalanmazsa, önce "finally" içindeki kod çalıştırılır daha sonra dışarıdaki "catch" kodu çalıştırılır.

```
try {  
    for (index = 0; index < 100; index++) {  
        ...  
        if (...) {  
            return;  
        } /** if sonu  
    } /** try yan tümcesinin sonu.  
    catch (...) {  
        ...  
    } // başka ayrılık yöneticileri de olabilir  
    finally {  
        ...  
    } /** try yapısının sonu
```



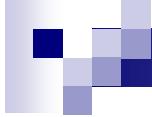
Java'da Ayrıılıkların yönetilmesi

■ Değerlendirme

- Ayrıılık tipleri C++'a göre daha anlamlı.
- "throws" yantümcesi C++'dan daha iyi (C++ "throws" yantümcesi programcıya çok az bilgi verir).
- "finally" yantümcesi kullanışlı.
- Java yorumlayıcısı programcı tarafından kullanılabilen çokok ayrıılık atar.

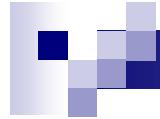
Bölüm 15

Fonksiyonel programlama dilleri
(Functional Programming Languages)



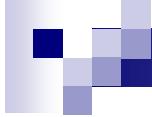
Fonksiyonel programlama dilleri

- Buyurgan dillerin tasarımı doğrudan von Neumann mimarisine (von Neumann architecture) dayanır.
- Temel ilgi yazılım geliştirmeye uygunluktan çok verimlidir.



Fonksiyonel programlama dilleri

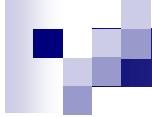
- Fonksiyonel dillerin tasarımı matematik fonksiyonlarına dayanır.
 - Üzerinde çalışılacak makinanın mimarisi ile fazla ilişkisi olmayan, kullanıcıya daha yakın sağlam bir teorik temele dayanır.



Matematiksel fonksiyonlar

- Tanım: Alan setinden değer setine olan eşlemlemeye (gönderime) fonksiyon denir. (A mathematical function is a **mapping** of members of one set, called the **domain set**, to another set, called the **range set**)
- Bir lambda ifadesi (**lambda expression**) parametreleri ve eşlemlemeyi örneğin küpü fonksiyonu ($\text{küp } (x) = x * x * x$) için aşağıdaki gibi belirler:

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



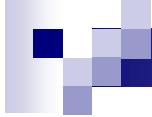
Matematiksel fonksiyonlar

- Lambda ifadeleri isimsiz fonksiyonları belirler.
- Lambda ifadeleri parametrelere, parametreleri ifadeden sonra yerleştirmek sureti ile uygulanır:

$$(\lambda(x) x * x * x)(3)$$

sonuç 27 olur.

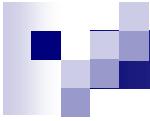
- LISP: ((lambda (x) (* x x x)) (3))
- Haskell: (\x->x*x*x) 3



Matematiksel fonksiyonlar

■ Fonksiyonel formlar

- Tanım: Yüksek sınıfından fonksiyon, fonksiyonel form, ya bir fonksiyonu parametre olarak alır, veya bir fonksiyonu sonuç olarak verir, ya da ikisini de yapar.



Fonksiyonel yapılar

1. Fonksiyon Birleşimi (Function Composition)

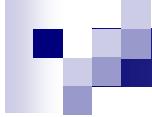
□ Şekli: $h \equiv f \circ g$

Anlamı $h(x) \equiv f(g(x))$

Örnek:

$$f(x) \equiv x * x * x \text{ ve } g(x) \equiv x + 3,$$

$$h \equiv f \circ g \rightarrow (x + 3) * (x + 3) * (x + 3)$$



Fonksiyonel yapılar

2. Yapım (Construction)

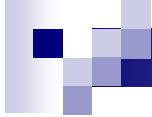
- Fonksiyonlardan oluşan listeye denir. Elemanların her biri parametreye uygulanır ve sonuçların listesi oluşturulur.

Şekil: $[f, g]$

Örnek:

$$f(x) \equiv x * x * x \text{ ve } g(x) \equiv x + 3,$$

$$[f, g](4) \rightarrow (64, 7)$$



Fonksiyonel yapılar

3. Hepsine-uygula (Apply-to-all)

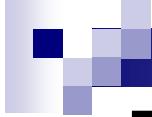
- Tek bir fonksiyonu parametre olarak alan ve bunu bir parametre listesine uygulayarak, sonuçta başka bir liste dönen yapıdır.

Şekil: α

Örnek:

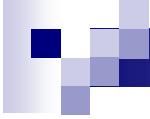
$$h(x) \equiv x * x * x$$

$$\alpha(h, (3, 2, 4)) \rightarrow (27, 8, 64)$$



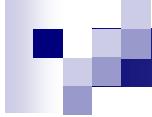
Fonksiyonel Programlama Dillerinin (FPD) esasları

- FPD tasarımının hedefi matematiksel fonksiyonları olabildiğince geniş kapsamda taklit etmektir.
- Bilgiyi işlemenin yöntemi FPD'de buyurgan dillere göre temelde farklıdır:
 - Buyurgan dillerde işlemler yapılır ve sonuçlar daha sonra kullanılmak üzere değişkenlerde saklanırlar.
 - Değişkenlerin yönetimi temel ilgi konusudur ve zorlukların kaynağıdır.
 - FPD'de değişkenler matematikte olduğu gibi gerekli değildir.
- FPD'de bir fonksiyon aynı parametrelere uygulandığında hep aynı sonucu verir, yan etki yoktur.



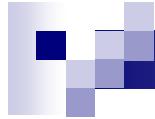
LISP

- **Veri nesne tipleri:** başlangıçta sadece atom'lar ve list'ler.
- **List formu:** Parantez içinde atomlar.
Örneğin (A B (C D) E)
- Başlangıçta, LISP'de atom ve list'den başka tip yoktu.
- Başlangıçta ilk lisp yorumlayıcısı lisp yazımının hesaplama yeteneklerini göstermek için geliştirildi.



Scheme-Lisp'in bir lehçesi

- 1970 ortalarında temiz, modern, basit ve çağdaş bir LISP lehçesi olarak tasarlandı.
- Sadece statik kapsam kullanır.
- Fonksiyonlar birinci sınıf elemanlardır:
 - Fonksiyonlar ifadelerin değeri, listelerin elemanları olabilirler.
 - Fonksiyonlar değişkenlere atanabilir, parametre olarak geçilebilirler.

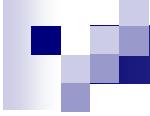


Scheme

- Basit fonksiyonlar

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

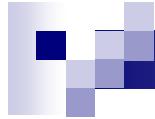
örneğin **(+ 5 2) → 7**



Scheme

2. **QUOTE** –bir parametre alır, değerlendirmeden geri döner.

- **QUOTE** gereklidir çünkü Scheme değerlendiricisi, EVAL, parametreleri fonksiyonları uygulamadan önce devamlı olarak değerlendirir. QUOTE buna engel olmak için kullanılır.
- **QUOTE** kısaca ' karakteri ile de gösterilebilir:
`'(A B) eşittir (QUOTE (A B))`



Scheme

3. Temel fonksiyonlar CAR, CDR, CONS.

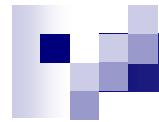
$(CAR '(A B C)) \rightarrow A$

$(CAR '((A B) C D)) \rightarrow (A B)$

$(CDR '(A B C)) \rightarrow (B C)$

$(CDR '((A B) C D)) \rightarrow (C D)$

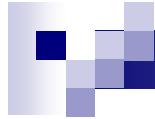
$(CONS 'A '(B C)) \rightarrow (A B C)$



Scheme

4. LIST –

$(LIST \ 'A \ 'B \ 'C \ 'D) \rightarrow (A \ B \ C \ D)$



Scheme

■ Lambda ifadeleri

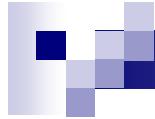
□ Form λ yazımına dayanır:

örneğin, (LAMBDA (L) (CAR (CAR L)))

L 'e bağlı değişken denir.

■ Lambda ifadeleri uygulanabilir:

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

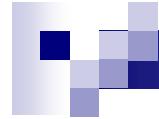


Scheme

- Fonksiyon yapılandırmak için fonksiyon **DEFINE** – İki şekli vardır:
 1. Bir sembolü bir atoma bağlamak için.

(**DEFINE** pi 3.141593)

(**DEFINE** two_pi (* 2 pi))



Scheme

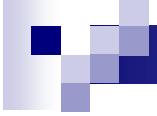
2. Bir ismi lambda ifadesine bağlamak için (clisp'de defun).

Örneğin:

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

Örnek kullanım:

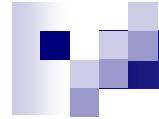
```
(cube 4)
```



Scheme

■ Değerlendirme yöntemi (normal fonksiyonlar için):

1. Belli bir sıra uygulanmadan fonksiyonların parametreleri değerlendirilir.
2. Parametrelerin değerleri fonksiyonun içinde kullanılır.
3. Fonksiyonun esas kısmında son ifadenin değeri fonksiyonun değeridir.



Scheme

- Örnekler:

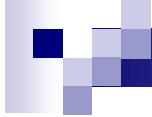
```
(DEFINE (square x) (* x x))
```

```
(DEFINE (hypotenuse side1 side1)
```

```
  (SQRT (+ (square side1)
```

```
    (square side2))))
```

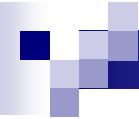
```
)
```



Örnek Scheme Fonksiyonları

1. **member** – atom listenin elemanı mı?

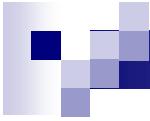
```
(DEFINE (member atm lis)
  (COND
    ( (NULL? lis) '())
    ( (EQ? atm (CAR lis)) #T)
    ( (ELSE (member atm (CDR lis))) )
  ))
```



Örnek Scheme Fonksiyonları

2. **equalsimp** – iki basit liste alır ve eşitliğini kontrol eder

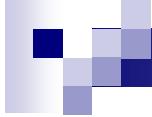
```
(DEFINE (equalsimp lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) '())
    ((EQ? (CAR lis1) (CAR lis2))
     (equalsimp (CDR lis1) (CDR lis2)))
    (ELSE '()))
  ))
```



Örnek Scheme Fonksiyonları

3. **equal** – iki genel liste alır ve eşitliğini kontrol eder

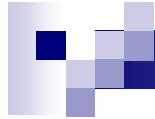
```
(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (EQ? lis1 lis2))
    ((NOT (LIST? lis2)) '())
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) '())
    ((equal (CAR lis1) (CAR lis2))
     (equal (CDR lis1) (CDR lis2)))
    (ELSE '())))
  ))
```



Örnek Scheme Fonksiyonları

4. **append** – İki listeyi peş peşe ekler

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



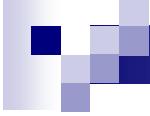
Scheme

- **LET** fonksiyonu

- Genel şekli:

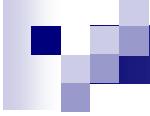
```
(LET (  
    (name_1 expression_1)  
    (name_2 expression_2)  
    ...  
    (name_n expression_n))  
  body  
)
```

- **Semantics**: bütün ifadeleri değerlendirir, isimlere atar, daha sonra body'i değerlendirir.



Scheme

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



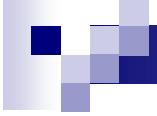
Scheme

- Yorumlama sırasında kod üretilip daha sonra değerlendirilebilir. Aşağıdaki iki örneği karşılaştırın.

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

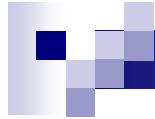
((DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis)))))

  ))
```



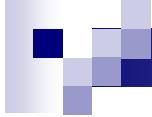
Scheme

- Scheme'in buyurgan dil özellikleri :
 1. **SET!** isimlere değer atar.
 2. **SET-CAR!** listenin **car** kısmını değiştirir.
 3. **SET-CDR!** listenin **cdr** kısmını değiştirir.



COMMON LISP

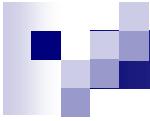
- 1980'ler sırasında popular olan LISP lehçelerinin bir birleşimi.
- Scheme'in tersine büyük ve karmaşık bir dil. Birçok buyurgan dil yapısı eklenmiştir.



COMMON LISP

■ özellikler:

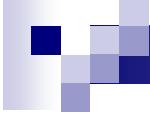
- records
- arrays
- complex numbers
- character strings
- powerful I/O capabilities
- packages with access control
- imperative features like those of Scheme
- iterative control statements



COMMON LISP

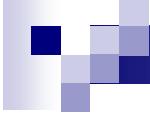
■ Örnek

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



ML

- Statik kapsamlı fonksiyonel dil. Sözdizimi Lisp'den çok Pascal'a benzer.
- Tip tanımlama kullanır ama tanımlanmamış olanların da tanımını kendi çıkarır.
- Güçlü tip tanımlama (Scheme'de tip yokken).



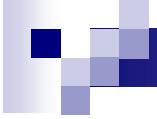
ML

- list'ler ve list işlemleri var.
- **val** ifadesi değeri isme bağlar (Scheme'de **DEFINE** gibi)
- Function tanımlama:

```
fun function_name (formal_parameters) =  
    function_body_expression;
```

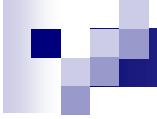
Örnek

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



Haskell

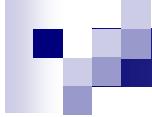
- ML gibi (sözdizimi, statik kapsamlı, güçlü tipli, tip çıkarsama)
- ML'den farkları (diğer dillerden de): tamamen fonksiyonel (sözgelişi değişken yok, atama yok, yan etki yok).



Haskell

■ En önemli özellikleri

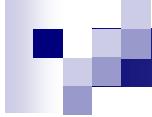
- Tembel değerlendirme (lazy evaluation) (hiçbir alt ifade değeri gerekli olmadıkça değerlendirilmez).
- Liste anlama özelliği sayesinde sonsuz listelerle işlem yapabilme özelliği.



Haskell Örnekleri

1. Fibonacci sayıları

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

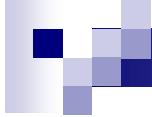


Haskell Örnekleri

2. Factorial

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

↑
guard



Haskell Örnekleri

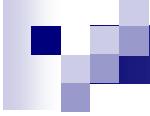
3. List İşlemleri

- List notasyonu: elemanları köşeli parantezlerin içine koy

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

- .. İşlevi ile seriler

```
[2, 4..10] → [2, 4, 6, 8, 10]
```



Haskell Örnekleri

3. List İşlemleri

- ++ ile birleştirme

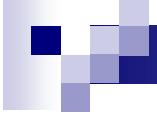
[1, 3] ++ [5, 7] → [1, 3, 5, 7]

- Üst üste nokta ile CONS, CAR, CDR (Prolog'da olduğu gibi)

1:[3, 5, 7] → [1, 3, 5, 7]

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

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



Haskell Örnekleri

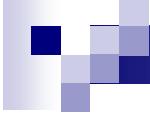
4. List yapıları: set yazımı

örneğin

```
[n * n | n <- [1..20]] → [1, 4, 9, 16, ..., 400]
```

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

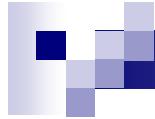
Bu fonksiyon n 'in bütün bölenlerinin listesini yapar.



Haskell Örnekleri

■ Quicksort:

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



Haskell Örnekleri

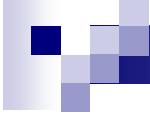
5. Geç/tembel değerlendirme (Lazy evaluation)

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

```
squares = [n * n | n ← [0..]]
```

Sadece gerekli olanları değerlendirir. Bu nedenle yukarıdaki listeler hemen hesaplanmaz. Ancak aşağıdaki örnekte görüldüğü üzere, gerektiği kadarı hesaplanır.

```
member squares 16 → True
```



Haskell Örnekleri

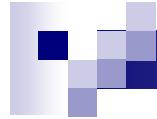
- member fonksiyonu aşağıdaki gibi yazılabilir:

```
member [] b = False
```

```
member (a:x) b = (a == b) || member x b
```

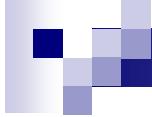
- Ancak bu bir önceki örnekte olduğu gibi tam kare olmayan girişler için sonsuza dek çalışır. Bunu önlemek için tanımlama aşağıdaki gibi yapılabilir:

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



Fonksiyonel dillerin uygulamaları

- LISP yapay zeka için kullanılır:
 - Bilgi gösternmek
 - Makine öğrenmesi
 - Doğal dil işleme
 - Konuşma ve görüşün modellenmesi
- Scheme programlama öğretilmesi amacıyla üniversitelerde.



Fonksiyonel ve Buyurgan dillerin karşılaştırılması

■ Buyurgan diller (Imperative Languages):

- Verimli çalışma
- Karmaşık anlam (Complex semantics)
- Karmaşık sözdizim (Complex syntax)
- Paralellik programcı tarafından tasarılanır (Concurrency is programmer designed)

■ Fonksiyonel diller (Functional Languages):

- Basit anlam
- Basit sözdizim
- Verimsiz çalışma (Inefficient execution)
- Programlar otomatik olarak paralel yapılabilir (Programs can automatically be made concurrent)