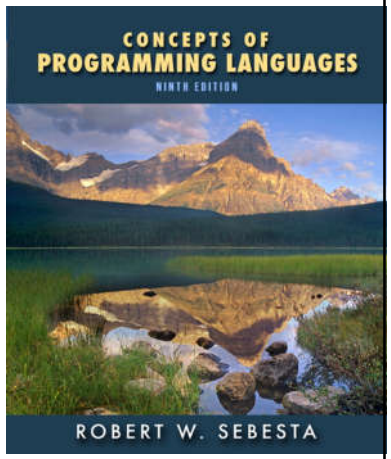
Bölüm 1

Hazırlık



Bölüm 1 Konular

- Programlama Dilleri Kavramlarının Öğrenilmesinin Nedenleri
- Programlama Alanları
- Dil Değerlendirme Kriterleri
- Dil Tasarımına Etki Eden Faktörler
- Dil Kategorileri
- Dil Tasarım Ödünleşmesi (Bir şeyi kazanmak için başka bir şeyden fedakarlık etme)
- Uygulama Yöntemleri
- Programlama Çevresi

Programlama Dilleri Kavramlarının Öğrenilmesinin Nedenleri

- Fikirlerin ifade edilmesi için artırılmış yetenek
- Uygun dillerin seçimi için geliştirilmiş geçmiş
- Yeni dilleri öğrenebilmek için artırılmış yetenek
- Uygulamanın önemini daha iyi anlama
- Bilinen dillerin daha iyi kullanımı
- Hesaplamanın etraflıca ilerletilmesi

Programlama Alanları

- Bilimsel uygulamalar (scientific)
  - Kayan noktalı hesaplamaların büyüklüğü; dizilerin kullanımı
  - Fortran
- İş uygulamaları (business)
  - Raporların üretimi, ondalık rakamların ve karakterlerin kullanımı
  - COBOL
- Yapay zeka (artificial intellegence)
  - Sayılarla uğraşmak yerine sembollerin kullanılması; yönlendirilmiş listelerin kullanımı
  - LISP
- Sistem programlaması (system programming)
  - Sürekli kullanım nedeniyle verim gerektirir
  - C
- Web yazılımı (web software)
  - Dillerin seçim koleksiyonu: biçimleme (XHTML), komut dizisi oluşturma (PHP), genel amaçlı (Java)

Dil Değerlendirme Kriterleri

- **Okunabilirlik-readability**: programın okunabilir ve anlaşılabilir kolaylığı
- **Yazılabilirlik-writability**: program oluşturmada kullanılacak dilin kolaylığı (uygunluğu)
- **Güvenilirlik-reliability**: şartnamelerin uygunluğu (örneğin, kendi şartnamelerin yapılması)
- **Maliyet-cost**: nihai toplam maliyet

Değerlendirme Kriteri: Okunabilirlik

- Bütün yönüyle kolaylık
  - Özelliklerin ve yapılandırma setinin yönetilebilirliği
  - Özellik çeşitliliğinin minimum seviyede tutulması
  - Aşırı işlem yüklenmesinin minimum seviyede tutulması
- Ortogonallik
  - İlkel yapılandırmanın göreceli küçük bir seti, göreceli küçük bir sayı usulleri ile birleştirilebilir
  - Mümkün olan her kombinasyon yasaldır
- Veri tipleri
  - Uygun ön tanımlı veri tipleri
- Sözdizimin (syntax) kurallarının göz önüne alınması
  - İşaretleyici formlar: esnek kompozisyonlar
  - Bileşim ifadelerin şekillendirilmesinin özel kelimeleri ve yöntemleri
  - Biçim ve anlamlar: kendinden tanımlı yapılar, anlamlı anahtar sözcükler

Değerlendirme Kriteri: Yazılabilirlik

- Basitlik ve Ortogonallik
  - Daha az yapılar, ilkel küçük sayıları, onları birleştirmek için küçük sayıda kurallar seti
- Soyutlamayı desteklemeli
  - Detayların ihmal edilmesine müsaade edecek biçimde kompleks yapıların tanımlanması ve kullanılması yeteneği
- Anlatımcılık
  - Belirlenmiş işlemlerin göreceli uygun usullerinin bir seti
  - Dayanıklık ve işlemlerin sayısı ve ön tanımlı fonksiyonlar

Değerlendirme Kriteri: Güvenirlirlik

- Tip kontrolü
  - Tip hatalarının test edilmesi
- Hariç tutma kullanımı
  - Run-time hata kesişmesi ve düzeltici ölçmelerin alınması
- Örtüşme
  - Aynı hafıza lokasyonu için bir veya daha fazla farklı referans verme yöntemlerinin mevcudiyeti
- Okunabilirlik ve yazılabilirlik
  - Bir algoritmayı ifade etmede «doğal» yolları desteklemeyen bir dil, «doğal olmayan» yaklaşımların kullanılmasını gerektirir ve bu yüzden de güvenliği azalır.

Değerlendirme Kriteri: Maliyet

- Dili kullanmak için programcıların eğitimi
- Programların yazılması (Belirli uygulamalara yakınlık)
- Programların derlenmesi
- Programların çalıştırılması
- Dil uygulama sistemi: ücretsiz derleyicilerin mevcudiyeti
- Güvenilirlik: Zayıf güvenilirlik maliyetlerin artmasına neden olur
- Programların bakımı

Değerlendirme Kriteri: Diğerleri

- Taşınabilirlik
  - Bir uygulamadan diğer bir uygulamaya bir programın taşınabilirliğindenki kolaylık
- Genellik
  - Uygulamaların geniş bir alana uygulanabilirliği
- İyi tanımlama
  - Dilin resmi tanımının bütünlüğü ve kesinliği

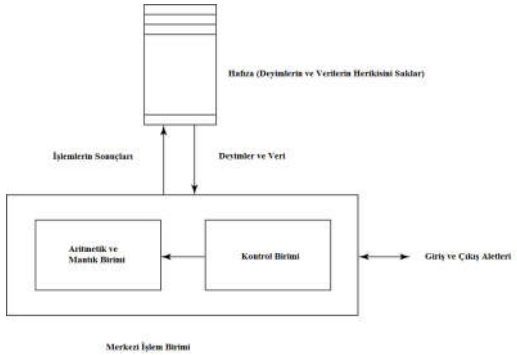
Dil Tasarımına Etki Eden Faktörler

- Bilgisayar Mimarisi
  - Diller *von Neumann* mimarisi olarak bilinen yaygın bilgisayar mimarisi üzerine geliştirilir
- Programlama metodolojileri
  - Yeni yazılım geliştirme metodolojileri (örneğin nesne tabanlı yazılım geliştirilmesi) yeni programlama paradigmalarının ve uzantılarının, yeni programlama dillerinin doğmasına neden olmaktadır

Bilgisayar Mimarisi Etkisi

- İyi bilinen bilgisayar mimarisi: Von Neumann
- von Neumann bilgisayarları nedeniyle, emir dilleri daha baskındır
  - Veri ve programlar hafızada saklanır
  - Hafıza CPU'dan ayrıdır
  - Komutlar ve veriler hafızadan CPU'ya iletilir
  - Emir dillerini esas alır
    - Değişken model hafıza hücreleri
    - Atama ifadeleri model iletme
    - İterasyon (adım adım) etkindir

von Neumann Mimarisi



von Neumann Mimarisi

- Fetch (bilgisayarda emirlerin getirilmesi)–çalıştırma–döngüsü (von Neumann mimarisi bilgisayarda)

```
Program sayıcısını başlangıç durumuna getir
repeat sureklidon
  sayac tarafindan isretlenen deyimleri getir
  sayaci artir
  komutu coz
  komutu calistir
end repeat
```


Programlama Metodolojilerinin Etkileri

- 1950li yıllar ve1960'lı yılların başı: Basit uygulamalar; makine verimliliği hakkında kaygılar
- 1960'lı yılların sonu: İnsan verimliliği önem kazandı; okunabilirlik, daha iyi kontrol yapıları
  - Yapılandırılmış programlama
  - Yukarıdan aşağıya tasarım ve adım usulü arıtma (düzeltme)
- 1970'li yılların sonu: İşlem tabanlıdan veri tabanlıya geçiş
  - Veri soyutlama
- 1980'li yılların ortaları: Nesne tabanlı programlama
  - Veri soyutlama + kalıtsallık + Çok biçimlilik

Dil Kategorileri

- Emir–Imperative
  - Merkezi özelikler değişkenlerdir, atama ifadeleri ve iterasyon
  - Nesne tabanlı programlamayı destekleyen dilleri kapsar
  - Script dillerini içerir
  - Görsel dilleri içerir
  - Örnekler: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- Fonksiyonel–Functional
  - Hesaplamaları icra ederken asıl araç, verilen parametreler için fonksiyonların uygulanmasıdır
  - Örnek: LISP, Scheme
- Mantıksal– Logical
  - Kurala dayalı (kurallar belirli bir sırada özelleştirilmez)
  - Örnek: Prolog
- Biçimleme/hibrid programlama–Markup/programming hybrid
  - Biçimleme dilleri bazı programlama dillerini desteklemek için genişletilmiştir
  - Örnekler: JSTL, XSLT

Dil Tasarım Ödünleşim (Trade-Offs)

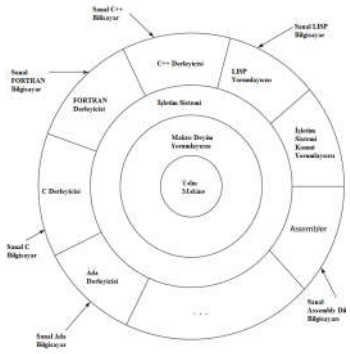
- Güvenilirlik, yürütme maliyeti
  - Örnek: Java uygun indeksleme için dizi elemanlarının tümünün kontrol edilmesini talep eder, bu durum yürütme maliyetini artırır.
- Okunabilirlik yazılabilirlik
  Örnek: APL birçok güçlü operatörü (ve büyük miktarlarda yeni sembolleri) sağlar. Bu durum kompakt bir program içinde kompleks hesaplamaların yazılmasına müsaade eder. Fakat zayıf okunabilirlik maliyetin artmasına neden olur.
- Yazılabilirlik (esneklik) ve güvenilirlik
  - Örnek: C++ işaretleyicileri güçlüdür ve çok esnektir, fakat güvenilir değildirler

Uygulama Yöntemleri

- Derleme- Compilation
  - Programlar makine diline çevrilir
- Sade Yorumlama-Pure Interpretation
  - Programlar, yorumlayıcı olarak adlandırılan diğer bir program tarafından yorumlanır
- Hibrid Yorumlama Sistemleri-Hybrid Implementation Systems
  - Derleyiciler ve sade yorumlayıcılar arasındaki uzlaşmayı sağlar

Bilgisayarın Tabakalaştırılmış Görünümü

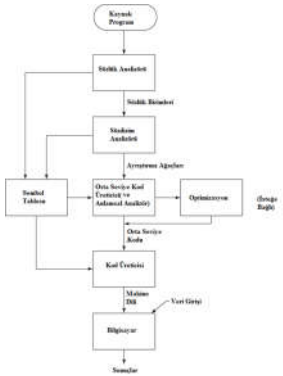
İşletim sistemi ve dil yorumlaması bir bilgisayarın makine arayüzü üzerinden tabakalaştırılır



Derleme

- Yüksek seviyedeki programı (kaynak dil) makine koduna (makine diline) çevirir
- Yavaş çevirme, hızlı yürütme
- Yürütme işleminin birkaç fazı vardır:
  - Sözcük analizi- lexical : Kaynak programdaki karakterleri sözcük birimlerine çevirir
  - Söz dizim analizi-syntax: sözcük birimlerini programın sözdizimsel yapısını temsil eden ayrıştırma ağaçlarına (parse units) dönüştürür
  - Anlamsal analiz-semantic: orta düzey kodları üretir
  - Kod üretimi: makine kodu üretilir

Derleme İşlemi



İlave Derleme Termolojileri

- **Yükleme modülü** (çalıştırılabilir imajı): kullanıcı ve sistem kodu birlikte
- **Yönlendirme ve yükleme**: sistem program birimlerinin toplanması işlemi ve onları bir kullanıcı programına yönlendirme

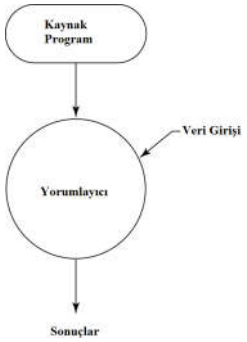
Von Neumann Darboğazı

- Bir bilgisayar hafızası ile onun işlemcisi arasındaki bağlantı hızı, bilgisayarın hızını tanımlar
- Genelde program komutları (deyimleri) bağlantı hızından daha hızlı çalışır; bu yüzden bağlantı hızı bir dar boğaza (tıkanıklığa) sebebiyet verir
- *Von Neumann darboğazı,* bilgisayar hızını öncelikli sınırlayan faktörüdür

Sade Yorumlama

- Çevirme yapılmamaktadır
- Programlar daha kolay uygulanır (çalışma zamanındaki hatalar kolayca ve anında görüntülenebilir)
- Daha yavaş çalışma (derlenen programlardan 10'dan 100'e kadar daha yavaştır)
- Genelde daha fazla yer gerektirir
- Geleneksel yüksek seviyeli diller için şimdi daha nadir kullanılır
- Bazı web skript dillerinde tekrar kullanılmaya başlanılmıştır (JavaScript, PHP)

Sade Yorumlama İşlemi



Hibrid Yorumlama Sistemleri

- Derleyici ve sade yorumlayıcılar arasında uzlaşmayı sağlar
- Bir yüksek seviye dil programı, kolay yorumlamaya müsaade eden bir orta dil seviyesine çevrilir
- Sade yorumlamadan daha hızlıdır
- Örnekler
  - Yorumlama yapmadan önce hataları bulmak için Perl programlar kısmen derlenir
  - Java'nın başlangıçtaki uygulamaları hibrid idi; orta seviye biçimi, bayt kod; bayt kod yorumlayıcı ve run-time sistemi (birlikte bunlar Java Sanal Makine olarak adlandırılır) olan herhangi bir makineye taşınabilirlik sağlar.

Hibrid Uygulama İşlemi

Tam zamanında (Just-in-Time) Uygulama Sistemleri

- Başlangıçta programları orta seviyedeki dile çevirir
- Daha sonra alt programların orta seviyedeki dilini çağrıldıklarında makine koduna derler
- Makine kod versiyonu müteakip çağrılmalar için muhafaza edilir
- Java programları için JIT sistemleri yaygın kullanılır
- .NET dilleri JIT sistemi ile uygulanır

Ön İşlemciler

- Ön işlemci makroları (deyimleri) genellikle diğer dosya kodlarını içerisine alacak şekilde belirlemek için kullanılır
- Bir ön işlemci, gömülü ön işlemci makrolarını genişletmek için program derlenmeden önce bir programı anında işler
- Çok iyi bilinen bir örnek: C önişlemci
  - `#include, #define` komutları ve benzer makrolar

Programlama Çevresi

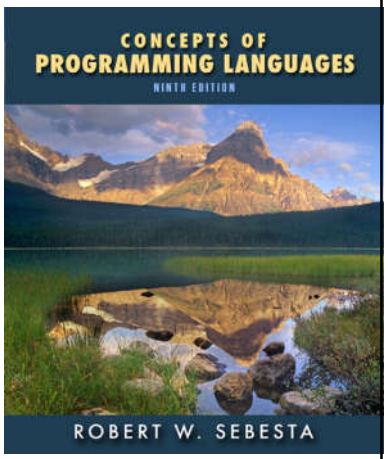
- Yazılım geliştirilmesinde kullanılan araçların bir koleksiyonu
- UNIX
  - Eski bir işletim sistemi ve araç koleksiyonu
  - Günümüzde sıkça GUI ile birlikte kullanılır (CDE, KDE, veya GNOME) Bunlar UNIX'in üst seviyesinde çalışır
- Microsoft Visual Studio.NET
  - Büyük, kompleks görsel çevre
- Web uygulamalarını ve Web olmayan herhangi bir .NET dili kullanılır
- NetBeans
  - Java'daki web uygulamaları hariç olmak üzere, Visual Studio .NET ile ilgilidir

Özet

- Programlama dillerini öğrenmek, birkaç nedenden ötürü önemlidir:
  - Farklı yapıları kullanmak için kapasitemizi artırır
  - Dilleri daha akıllıca seçmemize imkan tanır
  - Yeni dilleri öğrenmemizi kolaylaştırır
- Programlama dillerini değerlendirmek için daha önemli kriterler aşağıdakileri kapsar:
  - Okunabilirlik, Yazılabilirlik, Güvenilirlik, Maliyet
- Dil tasarımında asıl etkiler makine mimarisi ve yazılım geliştirme metodolojileridir
- Programlama dillerinin uygulamalarının asıl (temel) yöntemleri: derleme, sade yorumlama ve hibrid uygulamadır

Bölüm 2

Önemli Programlama Dillerinin Gelişimi



Bölüm 2 Konular

- Zuse'nin Plankalkül'ü (Programlama dilinin adı)
- Minimum Donanımlı Programlama: Sözde kodlar (Pseudocodes)
- IBM 704 ve Fortran
- Fonksiyonel Programlama: LISP
- Sofistikeliğe (çok yönlülüğe) Doğru İlk Adım: ALGOL 60
- Ticari kayıtları Bilgisayarlaştırma: COBOL
- Zaman Paylaşımının Başlangıcı: BASIC

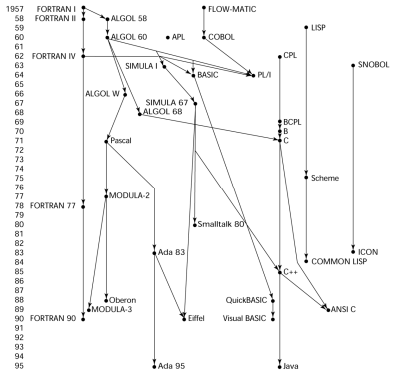
Bölüm 2 Konular (devam)

- Herkes için herşey: PL/I
- İlk İki Dinamik Dil: APL ve SNOBOL
- Veri Soyutlamanın Başlangıcı: SIMULA 67
- Ortogonal Tasarım: ALGOL 68
- Mantık Temelli Programlama: Prolog
- Tarihin en büyük tasarım çabası : Ada

Bölüm 2 Konular (devam)

- Nesne Tabanlı Programlama : Smalltalk
- Emir ve Nesne Tabanlı Özelliklerin Birleştirilmesi: C++
- Emir Temelli Nesne Tabanlı bir Dil : Java
- Metin (Script) Dilleri
- Yeni milenyum için C-temelli bir dil : C#
- Biçimlendirme / Hibrit Dillerin Programlaması

Yaygın Dillerin Şeceresi



Zuse'nin Plankalkül'ü

- 1945 yılında tasarlandı, fakat 1972 yılına kadar yayınlanmadı
- Asla uygulanmadı
- İleri veri yapıları
  - Kayan noktalı, diziler, kayıtlar
- Değişmezler

Plankalkül Sözdizimi

- A[5]'e A[4] + 1 ifadesini atamak için bir atama deyimi

```
     |   A + 1 => A
V    |   4         5          (indisler)
S    |   1.n       1.n        (veri türleri)
```


Minimum Donanım Programlama: Sözde kodlar (Pseudocodes)

- Makine kodu kullanmadaki yanlışlık neydi?
  - Zayıf okunabilirlik
  - Zayıf değiştirilebilirlik
  - İfade kodlama sıkıcı idi
  - Makine eksiklikleri – hiçbir indeksleme veya kayan nokta bulunmamaktadır

Sözde kodlar: Kısa kodlar

- Kısa kodlar 1949'da BINAC bilgisayarlar için Mauchly tarafından geliştirildi
  - İfadeler soldan sağa, kodlanmıştı
  - Örnek İşlemler:

```
01 - 06 abs value 1n (n+2)nd power
02 ) 07 +         2n (n+2)nd root
03 = 08 pause     4n if <= n
04 / 09 (         58 print and tab
```

```
X3 = ( X1 + Y1 ) / X1                    X3 03 09 X1 07 Y1 02 04 X1
```


Sözde kodlar: Hızlı kodlama

- Hızlı kodlama IBM 701 için 1954 yılında Backus tarafından geliştirilmiştir
  - Aritmetik ve matematik fonksiyonları için Sözde operatörler
  - Koşullu ve koşulsuz dallanma
  - Dizi erişimi için otomatik-artış kaydedicileri (register)
  - Yavaş!
  - Kullanıcı programı için sadece 700 kelime arta kalmaktadır

Sözde kodlar: İlgili Sistemler

- UNIVAC Sistem Derleme
  - Grace Hopper liderliğinde bir ekip tarafından geliştirildi
  - Sözdekod makine koduna genişletildi
- David J. Wheeler (Cambridge Üniversitesi)
  - Mutlak adresleme problemini çözmek için yeniden yerleştirilebilir adres blokları kullanarak bir yöntem geliştirdi.

IBM 704 ve Fortran

- Fortran 0: 1954 - uygulanmadı
- Fortran I:1957
  - Indeks kayıtlarına ve kayan nokta donanımına sahip yeni IBM 704 için tasarlanmıştır
    - Bu durum derlenmiş programlama dilleri fikrine yol açtı, çünkü yorumlama maliyetini saklayacak hiçbir yer yoktu (hiçbir kayan nokta yazılımı)
  - Gelişmenin çevresi
    - Bilgisayarlar kapasite açısından küçük ve güvenilmez idi
    - Uygulamalar bilimseldi
    - Programlama metodolojisi ve araçları yoktu
    - Makine verimliliği en önemli sorundu

Fortran Tasarım Süreci

- Fortran I tasarımında çevrenin etkisi
  - Dinamik depolamaya gerek yok
  - Iyi bir dizi işleme ve sayma döngülerine ihtiyaç duyuluyordu
  - String işleme, ondalık aritmetik, veya güçlü girdi / çıktı yoktu (iş yazılımı için)

Fortran I Genel Bakış

- Fortran'ın ilk uygulanan versiyonu
  - İsimler altı karakter uzunluğa kadar olabiliyordu
  - Post-testi sayma döngüsü (`DO`)
  - Formatlanmış I/O
  - Kullanıcı tanımlı alt programlar
  - Üç yollu seçme ifadeleri (aritmetik `IF`)
  - Veri tip ifadeleri bulunmamakta

Fortran I Genel Bakış (devam)

- FORTRAN'ın ilk uygulanan versiyonu
  - Ayrı derlemesi yoktu
  - 18 işçi-yıllık bir çaba sonrasında derleyici Nisan 1957'de çıktı
  - Genelde 704'ün kötü güvenilirliği nedeniyle 400 satırdan daha büyük programlar nadiren doğru derleniyordu.
  - Kod çok hızlı idi
  - Hızlıca yaygın olarak kullanıldı

Fortran II

- 1958'de dağıtıldı
  - Bağımsız derleme
  - Hatalar düzeltildi

Fortran IV

- 1960-62 yılları arasında gelişti
  - Açık tip bildirimleri
  - Mantıksal seçim ifadesi
  - Alt program isimleri parametreler olabiliyordu
  - 1966 yılında ANSI standardını aldı

Fortran 77

- 1978 yılında yeni standartları oldu
  - Karakter dizesi (string) işleme
  - Mantıksal döngü kontrol ifadesi
  - `IF-THEN-ELSE` ifadesi

Fortran 90

- Fortran 77'ye nazaran daha fazla önemli değişimler yapıldı
  - Modüller
  - Dinamik diziler
  - İşaretleyiciler
  - Önyineleme
  - `CASE` ifadesi
  - Parametre tür denetlemesi

Fortran'ın en son versiyonu

- Fortran 95 – nispeten ufak eklemeler, artı bazı silmeler
- Fortran 2003 – aynı

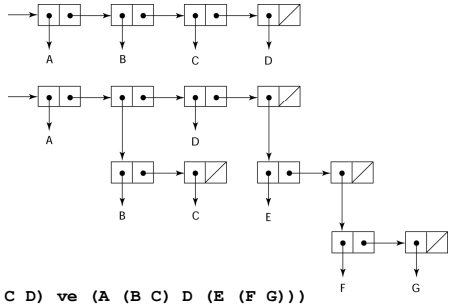
Fortran Değerlendirmesi

- Son derece optimize derleyicileri (90 öncesi tüm versiyonları)
  - Tüm değişkenlerin tip ve depolaması çalışma zamanından önce düzeltilir
- Bilgisayarların kullanıldığı yol dramatik olarak daima değişti
- Bilgisayar dünyasında geçerli dil olarak karakterize edildi

Fonksiyonel Programlama: LISP

- LISP Processing language
  - MIT'de McCarthy tarafından tasarlandı
- Yapay zeka (AI) araştırması;
  - Diziler yerine listelerde işlem verilerini,
  - Sayısal yerine sembolik hesaplamaları gerektirir
- Sadece iki veri tipi var: atomlar ve listeler
- Sözdizimi *lambda matematiğine* dayanır

İki LISP Listesinin Gösterimi



(A B C D) ve (A (B C) D (E (F G)))
Listelerinin gösterimi

LISP Değerlendirilmesi

- Öncü fonksiyonel programlama
  - Değişken ve atamaya ihtiyaç yoktur
  - Özyineleme ve koşullu ifadeler üzerinden kontrol
- AI (Artificial Intelligence) için hala baskın bir dildir
- COMMON LISP ve Scheme LISP'in çağdaş lehçeleridir

Scheme

- 1970'lerin ortalarında MIT'de geliştirildi
- Küçüktür
- Statik kapsamın yaygın kullanımı
- Birinci sınıf varlıklar olarak İşlevleri
- Basit sözdizimi (ve küçük boyutu) onu eğitim uygulamaları için ideal kılar

COMMON LISP

- LISP'in birkaç lehçesinin özelliklerini tek bir dilde birleştirme çabasıdır
- Büyük, karmaşık

Sofistikeliğe doğru İlk Adım: ALGOL 60

- Gelişme çevresi
  - FORTRAN (sadece) IBM 70x içindi
  - Diğer bir çok dil geliştirildi, tümü özel makineler içindi
  - Taşınabilir dil değillerdi; hepsi makineye bağlı idiler
  - Algoritma haberleşmesi için üniversal bir dil değildi
- ALGOL 60 üniversal bir dil tasarlama çabasının sonucu idi

Erken Tasarım Süreci

- ACM ve GAMM (27 Mayıs –1 Haziran 1958) tasarım için sadece dört gün bir araya geldi
- Dilin Hedefleri
  - Matematik notasyonlara yakın olmalı
  - Algoritmaları tanımlamak için iyi olmalı
  - Makine koduna çevrilebilmeli

ALGOL 58

- Tip kavramı resmileştirildi
- İsimler herhangi bir uzunlukta olabilirdi
- Diziler indislerin herhangi bir sayısı olabilir
- Parametreler (in & out) modu ile ayrıldı
- İndisler köşeli parantezler içine yerleştirildi
- Bileşik ifadeler (`begin ... end`)
- Bir deyimi ayırıcı olarak noktalı virgül kullanıldı
- Atama operatörü:= idi
- `If`, `else-if` cümlesi içerirdi
- I/O yok – "Bu durum makine bağımlı hale getirirdi"

ALGOL 58 Uygulaması

- Uygulanacağı anlamına gelmiyordu, fakat varyasyonları (MAD, JOVIAL) idi
- Başlangıçta IBM istekli olmasına rağmen, 1959 ortalarında tüm destek bırakıldı

ALGOL 60 Genel Bakış

- ALGOL 58 Paris'teki 6-gün süren toplantıda değiştirildi
- Yeni özellikleri
  - Blok yapı (yerel kapsam)
  - İki parametre geçişi yöntemleri
  - Özyineleme alt program
  - Yığın-dinamik diziler
  - Hala I / O yok ve herhangi bir dize (string) işleme bulunmamakta idi

ALGOL 60 Gelişimi

- Başarıları
  - 20 yılı aşkın sürede algoritmaları yayınlamak için standart yöntem idi
  - Müteakip emir dilleri bu (Algol 60) temelli idi
  - İlk makine bağımsız dil idi
  - Söz dizimi resmi tanımlanmış (BNF) ilk dil idi

ALGOL 60 Gelişimi (devam)

- Başarısızlıkları
  - Özellikle Amerika'da asla yaygın olarak kullanılmadı
  - Nedenleri
    - I/O eksikliği ve karakter seti eksikliği yüzünden programları taşınmaz yapmaktaydı
    - Çok fazla esnek idi–uygulama zordu
    - Fortran köklü (Entrenchment) idi
    - Örgün sözdizimi açıklama
    - IBM'den yeterli destek alamaması

Ticari Kayıtları Bilgisayarlaştırma: COBOL

- Gelişim çevresi
  - FLOW–MATIC'i kullanmak için UNIVAC başlangıç idi
  - AIMACO'yu kullanmak için USAF başlangıç idi
  - COMTRAN'ı IBM geliştirdi

COBOL'un Tarihsel Geçmişi

- FLOW–MATIC temellidir
- FLOW–MATIC özelliklerini taşır
  - İsimler 12 karaktere kadar olabiliyordu, gömülü tire ile birlikte
  - Aritmetik operatörler için İngilizce isimler (Aritmetik ifadeler yoktu)
  - Veri ve kod tümden birbirinden ayrı idi
  - Her ifade içerisindeki ilk kelime bir fiil idi

COBOL Tasarım Süreci

- İlk Tasarım Toplantısı (Pentagon) – May 1959
- Tasarım Hedefleri
  - Basit İngilizce gibi görünmeli
  - O az güçlü olacağı anlamına gelse bile, kullanımı kolay olmalı
  - Bilgisayar kullanıcıları tabanını genişletmeli
  - Mevcut derleyici problemleriyle kısıtlanmış olmamalı
- Tasarım komitesi üyelerinin tamamı bilgisayar üreticilerinden ve DoD (Dept. Of Defence) birimlerinden oluşuyordu
- Tasarım Problemleri: aritmetik ifadeler? indisler? Üreticileri arasında Savaşlar

COBOL Değerlendirmesi

- Katkıları
  - Yüksek düzeyli bir dilde ilk kez makro olanağı
  - Hiyerarşik veri yapıları (kayıtlar)
  - İç içe seçim ifadeleri
  - Tire ile uzun isimler (30 karaktere kadar),
  - Ayrı veri bölümü

COBOL: DoD Etkisi

- DoD tarafından ihtiyaç duyulan ilk dil
  - DoD desteği olmasaydı başarısız olacaktı
- Halen en yaygın kullanılan ticari uygulama dilidir

Zaman Paylaşım Başlangıcı: BASIC

- Dartmouth'da Kemeny ve Kurtz tarafından tasarlanmıştır
- Tasarım Hedefleri:
  - Öğrenmesi kolay ve fen bilgisi öğrencisi olmayanlar kullanabilmeli
  - "Hoş ve arkadaşça" olmalı
  - Ödevler hızlı yapılabilmeli
  - Ücretsiz olmalı ve kişisel erişim özelliği olmalı
  - Kullanıcı zamanının bilgisayar zamanından çok daha önemli olduğu unutulmamalı
- Mevcut popüler diyalekt: Visual BASIC
- Zaman paylaşımı ile birlikte kullanılan ilk yaygın dil

2.8 Herkes için Herşey : PL/I

- IBM ve SHARE tarafından tasarlandı
- 1964 yılındaki bilgisayar durumu (IBM'in bakış noktası)
  - Bilimsel hesaplama
    - IBM 1620 ve 7090 bilgisayarlar
    - FORTRAN
    - SHARE kullanıcı grubu
  - İş hesaplama
    - IBM 1401, 7080 bilgisayarlar
    - COBOL
    - GUIDE kullanıcı grubu

PL/I: Geçmişi

- 1963'de
  - COBOL'deki gibi Bilimsel kullanıcılar, I / O daha ayrıntılı ihtiyaç duymaya başladılar; iş kullanıcıları MIS için kayan nokta ve diziye ihtiyaç duymaya başladılar
  - Bilgisayarların iki çeşidi, dilleri ve teknik eleman desteği için çok satış yapacak gibi görünüyordu—Çok maliyetli idi
- Bariz çözüm
  - Her iki tür uygulamaları yapmak için yeni bir bilgisayar yapmak
  - Uygulamaların her iki çeşidini de yapabilecek yeni bir dil tasarlamak

PL/I: Tasarım Süreci

- 6 Komite üyesi tarafından beş ayda tasarlandı
  - IBM'den üç üye, SHARE üç üye
  - İlk kavram
  - Fortran IV'ün bir uzantısı
- Başlangıçta NPL olarak adlandırıldı (New Programming Language)
- 1965'de ismi PL/I oldu

PL/I: Değerlendirme

- PL/I katkıları
  - Birim seviyesi eşzamanlılıkta bir ilk
  - Harici işlemlerde bir ilk
  - Anahtar-seçmeli özyineleme
  - İşaretleyici veri türünde bir ilk
  - Dizi kesitlerinde bir ilk
- Endişeler
  - Birçok yeni özellik zayıf tasarlanmıştı
  - Çok büyük ve çok karmaşıktı

İlk iki Dinamik Dil : APL ve SNOBOL

- Dinamik yazma ve bellek tahsisi ile karakterize edilir
- Değişkenler yazılmaz
  - Bir değer atandığı zaman değişken tip edinir
- Bir değer atandığı zaman bellekte bir değişken tahsis edilir

APL: A Programlama Dili

- 1960 civarında Ken Iverson tarafından IBM'de çalışan bir donanım tanımlama dili olarak tasarlanmıştır
  - Çok anlamlıdır (birçok operatör, çeşitli boyutlarda hem skaler ve diziler için)
  - Programların okunması çok zor
- Halen kullanımdadır; Az düzeyde değişiklikler yapıldı

SNOBOL

- 1964 yılında Farber, Griswold ve Polensky tarafından Bell Laboratuvarları'nda metin (string) işleme dili olarak tasarlandı
- Metin desen eşleştirme için güçlü operatörlere sahip
- Alternatif dillerden (ve bu yüzden artık yazım editörleri tarafından kullanılmamaktadır) daha yavaş
- Halen bazı metin işleme görevleri için kullanılmaktadır

Veri Soyutlamanın Başlangıcı : SIMULA 67

- Nygaard ve Dahl tarafından öncelikle Norveç'te sistem simülasyonu için tasarlandı
- ALGOL 60 ve SIMULA I temellidir
- Birincil Katkıları
  - Eşyordamlar - bir çeşit alt program
  - Sınıflar, nesneler ve miras

Ortogonal Tasarım: ALGOL 68

- ALGOL 60'ın süregelen gelişmesindendir, fakat o dilin üstü değildir
- Birçok yeni fikirlerin kaynağıdır (dilin kendisi hiçbir zaman yaygın kullanıma ulaşamamasına rağmen)
- Tasarım ortogonal kavramına dayanmaktadır
  - Birkaç temel kavramlar, artı birkaç birleştirici mekanizma

ALGOL 68 Değerlendirme

- Katılımlar
  - Kullanıcı tanımlı veri yapıları
  - Referans türleri
  - Dinamik diziler (flex diziler olarak adlandırılır)
- Yorumlar
  - ALGOL 60 dan daha az kullanım
  - Müteakip dillerde güçlü etkisi oldu, özellikle Pascal, C ve Ada üzerinde

Pascal – 1971

- Wirth (o dönemlerde ALGOL 68 komitesi üyesi) tarafından geliştirildi
- Yapısal programlama öğretmek için tasarlandı
- Küçük, basit, gerçekte yeni bir şey yok
- En büyük etkisi programlama öğretme üzerine oldu
  - 1970'lerin ortalarından başlayarak 1990'ların sonlarına kadar, programlama öğretmek için kullanılan en yaygın dildi

C – 1972

- (Dennis Richie tarafından Bell Laboratuvarları'nda) sistem programlaması için tasarlandı
- Öncelikle BCLP, B, fakat aynı zamanda ALGOL 68'den geliştirildi
- Güçlü operatörler setine sahip, fakat zayıf tip kontrolü var
- Başlangıçta UNIX üzerinden yayıldı
- Birçok uygulama alanı var

Mantık Temelli Programlama : Prolog

- Kowalski (Edinburgh Üniversitesi) yardımıyla, Comerauer ve Roussel (Aix-Marseille Üniversitesi) tarafından geliştirildi
- Formel mantığa dayalıdır
- Prosedürel değildir
- Verilen sorguların doğruluğunu anlamak için bir sonuç çıkarma kullanan akıllı bir veritabanı sistemi olarak özetlenebilir
- Çok verimsiz, dar uygulama alanları var

Tarihin en büyük tasarım çabası: Ada

- Büyük tasarım çabası, yüzlerce insan ilgilendi, çok fazla paraya mal oldu ve sekiz yıllık bir süreyi aldı
  - Strawman gereksinimleri (Nisan 1975)
  - Woodman gereksinimleri (Ağustos 1975)
  - Tinman gereksinimleri (1976)
  - Ironman ekipmanları (1977)
  - Steelman gereksinimleri (1978)
- İlk programcı Augusta Ada Byron ismine izafeten Ada olarak adlandırıldı

Ada Değerlendirilmesi

- Katkıları
  - Paketler – veri soyutlaması için destek
  - Kural dışı durum işleme– özenle hazırlandı
  - Generik program birimleri
  - Aynı anda kullanım – Görev modeli ile
- Yorumlar
  - Rekabetçi tasarım
  - Yazılım mühendisliği ve dil tasarımı hakkında bilinen her şeyi kapsıyordu
  - İlk derleyiciler çok zordu; ilk gerçek kullanılabilir derleyici, dil tasarımı tamamlandıktan hemen hemen 5 yıl sonra ortaya çıkabildi

Ada 95

- Ada 95 (1988 yılında başladı)
  - Nesne tabanlı programlamada tip türetimi için destek sağladı
  - Paylaşılmış veriler için daha iyi kontrol mekanizmasına sahip idi
  - Yeni aynı anda kullanım özellikleri
  - Daha esnek kütüphane
- DoD artık ihtiyaç duymaması ve C++ nın popüler olması nedenleriyle popülerliği azaldı

Nesne Tabanlı Programlama: Smalltalk

- Xerox PARC'ta geliştirildi, başlangıçta Alan Kay ve sonradan Adele Goldberg tarafından geliştirildi
- Nesne tabanlı dilde (veri soyutlama, miras ve dinamik bağlama) ilk tam uygulama
- Grafik kullanıcı arayüzü tasarımına öncülük etti
- Nesne Tabanlı Programlamaya tanıtıldı

Emir ve Nesne Tabanlı Programlamanın Birleştirilmesi: C++

- 1980 yılında Bell Laboratuvarında Stroustrup tarafından geliştirildi
- C ve SIMULA 67'den geliştirildi
- Nesne Tabanlı Programlama özellikleri kısmen SIMULA 67'den alındı
- Özel durum işleme sağlıyor
- Hem prosedürel hem de Nesne Tabanlı Programlamayı desteklediği için büyük ve karmaşık bir dildir
- OOP ile birlikte hızla, popülerliğini arttırdı
- ANSI standardı Kasım 1997'de onaylandı
- Microsoft versiyonu (2002'de .NET piyasaya sürüldü): Yönetildi C++
  - delegeler, arayüzleri, çoklu kalıtım yok

İlgili Nesne Tabanlı Diller

- Eiffel (Bertrand Meyer tarafından tasarlandı – 1992)
  - Doğrudan başka dilden değil
  - C++'dan daha küçük ve basit, fakat hala güçlü
  - C++'ın popülerliği eksikti, çünkü birçok C++ hayranları aynı zamanda C programcılarıydı
- Delphi (Borland)
  - Nesne tabanlı programlama desteklemek için Pascal'ın ilave özelliklerini kullandı
  - C++'tan daha zarif ve güvenli

Emir Temelli Nesne Tabanlı Dil: Java

- 1990'lı yılların başında Sun tarafından geliştirildi
  - C ve C++ gömülü elektronik aletler için yeterli değildi
- C++ Temellidir
  - Önemli seviyede basitleştirilmiş (`struct`, `union`, `enum`, işaretçi aritmetiği ve C++'ın atama zorlamalarının yarısını kapsamıyor)
  - Sadece OOP'yi destekliyor
  - Referansları var ama işaretleyicileri yok
  - Uygulamalar için destek ve eşzamanlılık formu içerir

Java Değerlendirilmesi

- C++'ın birçok güvensiz özelliklerini bertaraf etti
- Eşzamanlılığı destekliyor
- Apletler için kütüphaneleri var, GUI, veritabanı erişimi mümkün
- Taşınabilir: Java Sanal Makine konsepti, JIT derleyicileri
- Web programlaması için çok kullanıldı
- Önceki dillere nazaran kullanımı daha hızlı arttı
- En son yeni versiyonu olan 5.0 2004 yılında piyasaya sürüldü

Web için Metin Dilleri

- Perl
  - Larry Wall tarafından tasarlandı—ilk olarak 1987'de piyasaya sürüldü
  - Değişkenler statik olarak yazılırdı, ancak dolaylı olarak deklere edilirdi
  - Üç ayırt edici ad, değişken adının ilk karakteri ile gösterilir
  - Güçlü, fakat tehlikeli
  - Web CGI programlama için yaygın kullanımı kazanmış
  - Ayrıca UNIX sistem yönetimi dil için yedekolarak kullanıldı
- JavaScript
  - Netscape ile başladı, fakat sonra Netscape ve Sun Mikro sistemleri ortak girişimi olarak devam etti
  - Genellikle dinamik HTML dökümanları oluşturmak için kullanılan bir istemci tarafı HTML içine gömülü bir betik dili şeklinde kullanıldı
  - Tamamen yorumlayıcıdır
  - Java ile olan ilişkisi sadece aynı söz dizimini kullanmasıdır
- PHP
  - PHP: Hiper metin önişlemci, Rasmus Lerdorf tarafından tasarlandı
  - Genellikle Web üzerinden form işleme ve veritabanı erişimi için kullanılan bir sunucu tarafı HTML içine gömülü bir betik dilidir
  - Tamamen yorumlayıcıdır

Web için Metin Dilleri

- Python
  - Nesne tabanlı yorumlayıcıya sahip bir metin dilidir
  - Tip kontrol edilir ama dinamik yazılır
  - CGI programlama ve form işleme için kullanılır
  - Dinamik yazılabilir, ancak tipi kontrol edilir
  - Listeleri, değişkenler gurubu ve karmaları destekler
- Lua
  - Nesne tabanlı yorumlayıcıya sahip bir metin dilidir
  - Tip kontrol edilir ama dinamik yazılır
  - CGI programlama ve form işleme için kullanılır
  - Dinamik yazılabilir, ancak tipi kontrol edilir
  - Listeleri, değişkenler gurubu ve karmaları destekler, bütün bunları onun tek veri yapısı ve tabloları üzerinden yapar
  - Kolayca genişletilebilir

Web İçin Metin Dilleri

- Ruby
  - Yukihiro Matsumoto (a.k.a, "Matz") tarafından Japonya'da tasarlandı
  - Perl ve Python için yedek bir dil olarak başladı
  - Bir saf nesne yönelimli bir (Script) dil
    - Tüm veriler nesnedir
  - Birçok operatör kullanıcı kodu tarafından yeniden tanımlanabilen metotlar olarak uygulanır
  - Sade yorumlayıcıdır

Yeni Milenyum için C Temelli bir Dil: C#

- .NET geliştirme platformunun (2000) bir parçasıdır
- C++ , Java ve Delphi temellidir
- Bileşen tabanlı yazılım geliştirme için bir dil sağlar
- Bütün .NET dilleri genel sınıf kütüphanesi sağlayan Common Type System –Ortak Tip Sistemini (CTS) kullanır
- Yaygın kullanıma ulaşacak olması muhtemeldir

İşaretleme(Markup)/Programlama Hibrit Diller

- XSLT
  - eXtensible Markup Language (XML)(genişletilebilir işaretleme dili): bir metamarkup dili
  - eXtensible Stylesheet Language Transformation (XSTL)(genişletilebilir stilsayfası dil dönüşümü) XML dökümanlarını görüntülenebilmesi için dönüştürür
  - Programlama yapıları (örn., döngüler)
- JSP
  - Java Server Pages(Java Sunucu Sayfaları): dinamik web dökümanlarını destekleyen teknolojiler koleksiyonu
  - servlet: bir Web servera ait bir Java programı; servlet'in çıktısı browserda görüntülenir

Özet

- Geliştirme (development), geliştirme platformu (development environment), ve bazı önemli programlama dillerinin değerlendirilmesi
- Dil tasarımındaki mevcut sorunlara bakış açısı

BLG339
PROGRAMLAMA DİLLERİ KAVRAMI

Hafta 2

Yrd. Doç. Dr. Melike Şah Direkoğlu

Alındığı kaynak:
Addison-Wesley's Programming Language Concepts slaytları ve Prof. Dr. Tuğrul Yılmaz' ın ders notlarından faydalanarak hazırlanmıştır.

Konular

- Sözdizim (syntax) ve Anlambilim (Semantics) Analizi
- Sözdizim(Syntax) Tanımlamanın Genel Terminolojisi
- Anlambilim (Semantics) Tanımlamanın Genel Terminolojisi

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 bir örnek, bir değişkenin kullanılmadan önce tanımlanması bir anlam kuralı örneğidir.
- Bir ya da daha çok dilin sözdizimini anlatmak amacıyla kullanılan dile metadil (metalanguage) 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 ve Anlambilim (devam)

- Sözdizimi (Syntax) ve anlamsallar/anlambilim (semantics) bir dilin tanımını sağlarlar.
  - Bir dil tanımının kullanıcıları
    - Diğer dil tasarımcıları; dilin nasıl çalıştığını anlamak için
    - Gerçekleştiriciler (implementors/compilers); programın nasıl çalıştığını anlamak için
    - Programcılar; nasıl programlanacağını anlamak için

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

- Bir başka örnek;
  "while (boolean ifade) deyim"
  → semantics (anlamı): Boolean ifade doğru ise deyime geç ve boolean ifade doğru olduğu sürece dögüye devam et.
- Ayrıca sözdizimindeki küçük farklar anlamda büyük farklılıklara neden olabilir. Bunlara dikkat etmek gerekir:

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

- İyi tasarlanmış programlama dillerinde syntax (sözdizim) ve semantics (anlam) birbiriyle bağlantılıdır.

Sözdizim (Syntax)

Sözdizim(Syntax) Tanımlamanın Genel Terminilojisi

- Dil konuşma dili (Türkçe) veya yapay bir dil (Java) olsun, dil karakterler ve cümleler/deyimlerden meydana gelmektedir.
  - Karakterlerin tümü alfabeyi oluşturur.
  - Cümleler kümesi de dili oluşturur.
- Karakterin sözdizimi kuralları o alfabede kelimelerden nasıl cümleler oluştuğunu belirler.
  - Örnek: Türkçede özne başta, fiil sonda olur. Gizli özne kuralı, devrik cümle kuralı, vs. Bu kuralların hepsi Türkçe dilinin nasıl kullanılacağını belirlediği gibi, bir programlama dilinin syntax (sözdizim) kuralları da o dilin nasıl kullanılacağını belirler.
- Bir lexeme dilin en düşük seviyeli sözdizimsel birimidir
  - örn., operatörler (+, - *, /), sayısal değerler, özel kelimeler (begin, end)

Sözdizim(Syntax) Tanımlamanın Genel Terminilojisi (Devam)

- Simge(token) da Lexeme lerin bulunduğu kategoriyi temsil eder
  - Değişken (variable), metot (method) veya sınıf (class) isimlerine identifier grubu denir.
  - Her grup da bir isim veya token (simge) ile temsil edilir (örn. identifier)

token	sample lexemes
identifier	`count, i, x2, ...`
if	`if`
add_op	`+`
semi	`;`
int_lit	`0, 10, -2.4`

Lexeme ve Token Örneği

- Java deyimi:
  - toplam = 2 * sayi + 17;

Lexeme	Token (simge)
toplam	identifier
=	equal_sign
2	int_literal
*	mult_op
sayi	identifier
+	plus_op
17	int_literal
;	semicolon

Dillerin Resmi Tanımı

- Diller iki farklı method ile; recognition (tanıyıcılar) ve generation (üreticiler) ile tanımlanabilirler.
- Dil Tanıyıcıları(Recognizers)
  - Bir tanıyıcı aygıt dilden bir karakter dizisi okur ve bu karakter dizisinin bu dile ait olup olmadığına karar verir; karakter dizisisi ya kabul olunur yada reddedilir.
  - Bir dilde birçok kelime seçeneği olabileceğinden bu zaman alır. Fakat dilin tüm olası kelimelerini yaratmak için kullanılır.
  - Örnek: bir derleyicinin (compiler) söz dizim analizcisi (sadece eldeki programın sözdizimi dile uyup uymadığını test ettiğinden hızlıdır).
- Dil Üreticileri(Generators)
  - Bir dile ait cümle üreten aygıtlar olarak düşünülebilir.
  - Belirli bir cümlenin söz diziminin doğru olup olamadığı o dili üreten üretici incelenerek tespit edilebilir.
- Bu iki metot dillerin nasıl derlendiği (compile) ve çalıştığını (interpret) etkilemiştir.

Söz Diziminin Tanımında Kullanılan Resmi Metotlar

- Serbest- İçerik Gramerleri (Context-Free Grammars) aynı zamanda Backus-Naur Form olarak da bilinir.
  - Programlama dillerinin söz dizimlerini tanımlamada kullanılan en yaygın metottur.
- Genişletilmiş BNF(Extended BNF)
  - BNF nin okunabilirliğini ve yazılabilirliğini arttırır.
- Gramerler ve Dil Tanıyıcıları

BNF ve Serbest-İçerik Gramlerleri (Context- Free Grammars)

- Serbest İçerik Gramerleri (dil bilgisi)
  - Noam Chomsky (linguist) tarafından 1950 lerin ortasında doğal dillerin söz dizimlerini tanımlamak için geliştirdi
  - Serbest-İçerik dilleri adlanılan bir diller sınıfını tanımladı.
  - Serbest-İçerik dilleri (Context-free grammars) daha sonra programlama dillerini uygulandı.

Backus-Naur Form (BNF)

- BNF, 1950'li yıllarda John Backus ve Peter Naur 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.
- Metalanguage (meta dil) bir başka dili anlatmak amacıyla kullanılan dile denir. BNF bir metadildir.

Backus-Naur Form (BNF) devam

- BNF de, soyutlamalar (abstraction) kullanılarak söz dizimsel yapıların temsil edilir.
- Örnek abstraction (soyutlama): Assign (atama) kuralı

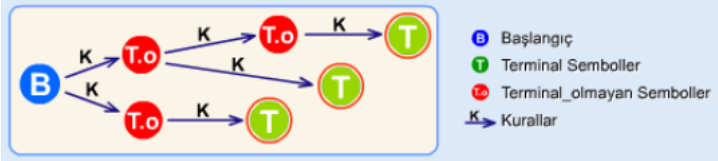
<assign> → <var> = <expression>

Left hand side (LHT) (sol taraf) Right hand side (RHT) (sağ taraf)

  - Bu bir kuraldır ve assign soyutlamasının tanımlanabilmesi için <var> ve <expression> da tanımlanması gerekir
  - Toplam = toplam1 + toplam2 buna örnektir.
  - Burada <assign> yada örnektedi toplam non-terminal symbol (son olmayan semboller) ve toplam1, toplam2 ve lexemeler (=, +) da terminal symbol (son olan sembol) olarak adlandırılır.
- BNF grameri de bir kurallar topluluğudur.

BNF Temelleri

- Son Olmayan semboller(Non-terminals): BNF soyutlamaları olarak adlandırılır (abstractions)
- Başlangıç sembolu
- Son semboller(Terminals): lexeme lar ve token lar
- Gramer: bir kurallar koleksiyonudur.
- BNF gramer yapısı aşağıdaki şekildeki gibi gösterilebilir:



BNF Kurallar

- Bir kuralda sol taraf (LHS-left hand side) ve sağ taraf(RHS-right hand side) vardır ve son olan(terminal) ve son olmayan (nonterminal) sembollerden oluşur.
- Bir gramer sonu boş olmayan kurallar kümesidir.
- Bir soyutlamanın (abstraction) (veya son olmayan sembol) birden fazla sağ tarafı olabilir.

  <stmt> → <single_stmt> | begin <stmt_list> end
- '|' işereti logical OR (veya) anlamındadır.

BNF Kurallar – Birden fazla sağ tarafın birleştirilmesi

- Java if deyimini örneği;

<if_stmt> → **if** (<logic_expr>) <stmt>
<if_stmt> → **if** (<logic_expr>) <stmt> **else** <stmt>

VEYA

<if_stmt> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <stmt> **else** <stmt>

BNF Kurallar – Listelerin Tanımlanması ve Gramer

- BNF'te listeleri tanımlamak için recursion (kendi kendini çağırma/yineleme) metodu kullanılır.
  <ident_list> → ident | ident, <ident_list>
- Bu türetme (derivation) kuralların tekrarlı bir şekilde uygulanmasıdır. Bir başlangıç sembolünden başlanır ve tüm elemanları son olan sembol (terminal) den oluşan bir cümle de biter.
- Türetme kullanılarak gramer oluşturulur.

Küçük bir Program Dili Gramer Örneği

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

Program begin ve end özel kelimeleriyle başlayıp bitmeli

Sadece bir statement (deyim) formatı var o da assignment

Her deyimin sonunda ; (noktalı virgül) var

Değişken isimleri sadece A, B veya C olabilir

Türetme ile gramer oluşturulabilir

Sadece + veya – operatörleri kullanılabilir

Örnek Dilinden Türemiş Program

Program **begin** A = B + C ; B = C **end**

Gramer

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

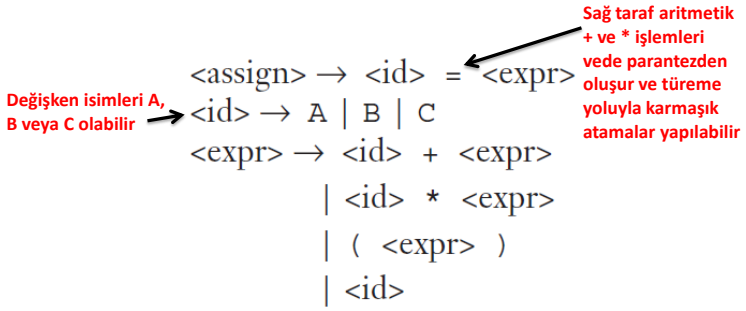
<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

Bütün türemeler program sembolu ile başlamalıdır

Basit Atama Deyimi Grameri



A = B * (A + C) deyimi bu gramere bir örnektir

Örnek Atamadan Türemiş Deyim

A = B * (A + C)

Gramer

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

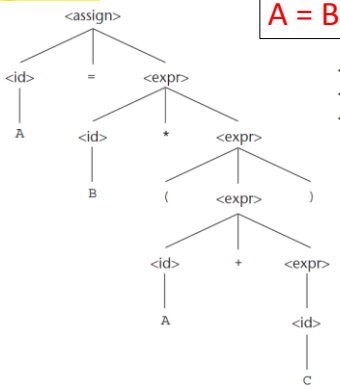
Bütün türemeler assign sembolu ile başlamalıdır

<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> A = B * (<expr>)
=> A = B * (<id> + <expr>)
=> A = B * (A + <expr>)
=> A = B * (A + <id>)
=> A = B * (A + C)

Ayrıştırma Ağacı (Parse Tree)

- Bir türetmenin hiyerarşik temsilidir.
- Bir dilin deyimlerinin (expressions) hiyerarşik olarak tanımlanması sağlar.

A = B * (A + C) deyimi



<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Gramerlerin Belirsizliği (Ambiguity)

- Bir gramerdeki bir cümlesel biçim iki veya daha fazla ayrıştırma ağacı (parse tree) oluşturuyorsa bu gramer belirsiz (ambiguous) olarak adlanır.

Bir Belirsiz Deyim(expression) Grameri

Önceki Gramer

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Belirsiz Gramer

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Belirsizlik, gramerin sözdizim kurallarında daha esnek olmasından kaynaklanır!

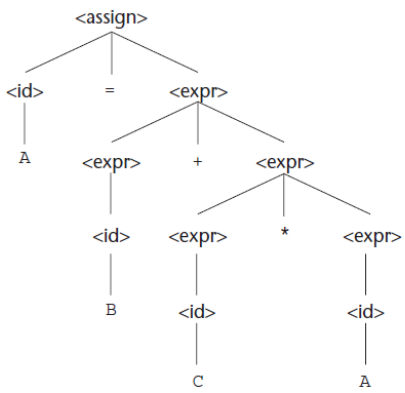
Belirsiz Deyim Ayrıştırma Ağacı (Parse Tree)

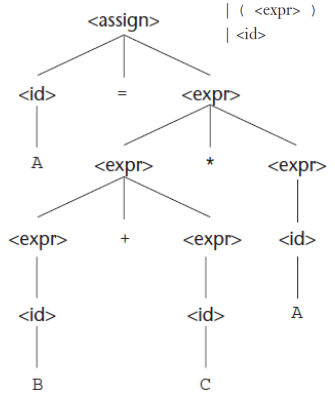
Deyim

A = B + C * A

Belirsiz Gramer

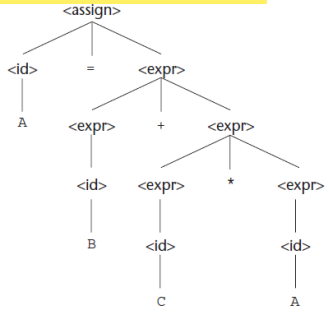
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>





Belirsiz Deyimde İşleçlerin (operator) Önceliği

- Eğer ayrıştırma ağacında işleçlerin(operators) öncelik seviyeleri gösterilirse belirsizlik olmaz.
- Bir ayrıştırma ağacında işleç (operator), ağacın (parse tree) ne kadar aşağısında yaratılıyorsa, o kadar öncelik hakkı var demektir.



Burada çarpma işlecinin öncellik hakkı var çünkü ilk bu işlem ele alınır.

Belirsiz Deyim Ayrıştırma Ağacı (Devam)

- Deyimler
  - A = A + B * C (B * C) ayrıştırma ağacında en aşağıda
  - A = A * B + C (B + C) ayrıştırma ağacında en aşağıda
    - Özetle bu gramerde operatörlerin sırası bir belirsizlik yaratıyor
- Gramerde basit deyimler kullanarak ve operatör (işleç) sırasını belirterek belirsizlik yok edilebilir.
  - Bunun için başka son olmayan semboller (non-terminals) ve kurallar gereklidir.
  - Örn. Örnekteki <expr> non-terminal (son olmayan sembol) ile birlikte çarpma (factor non-terminal)ve toplama (term non-terminal) sembolleri kullanılarak işlem sırası belirtilebilir.
  - <expr> sembolü root (ağaçta en üstte) olduğundan expr ile sadece toplama için kullanılan <term> sembolü, <expr> sembolü ile birleştirilirse, toplama her zaman ağacın üst kısmında kalmış olur. (yani çarpma önceliği eklenir)

Bir Belirsiz Olmayan (Unambigous) Deyim (Expression) Grameri

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

Toplam her zaman expr ile birlikte yapılmak zorunda olduğundan, toplam ayrıştırma ağacında üstte yer alır. Çarpma toplamaya eklendiğinden çarpma her zaman ayrıştırma ağacında en aşağıda yer alır.

Belirsiz Olmayan Gramerden Türemiş Deyim

Belirsiz Olmayan Gramer

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

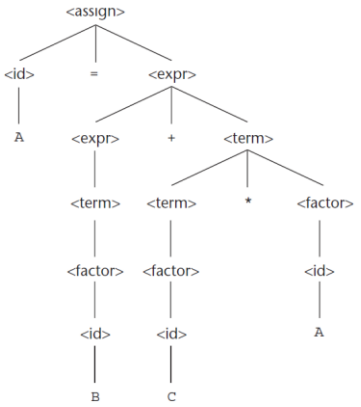
Deyim

A = B + C * A

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A

Belirsiz Olmayan Deyim Ayrıştırma Ağacı (Parse Tree)

- Ayrıştırma ağacı (parse tree) türemiş gramerden kolaylıkla çıkarılabilir.



Örnekler

- Verilen grameri kullanarak, aşağıdaki deyimler için sol taraftan türetme kullanarak, ayrıştırma ağacını (parse tree) gösteriniz.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

a. A = A * (B + (C * A))
b. B = C * (A * C + B)
c. A = A * (B + (C))

Örnekler

- Aşağıdaki gramerin oluşturduğu dili bir cümle ile açıklayınız.

<S> → <A> <B> <C>

<A> → a <A> | a

<B> → b <B> | b

<C> → c <C> | c

Örnekler

- Verile gramer şöyledir:

<S> → <A> a <B> b

<A> → <A> b | b

<B> → a <B> | a

- Hangi cümleler bu gramerden türemiştir?
  a. baab
  b. bbbab
  c. bbaaaaa
  d. bbaab

Örnekler

- Verile gramer şöyledir:

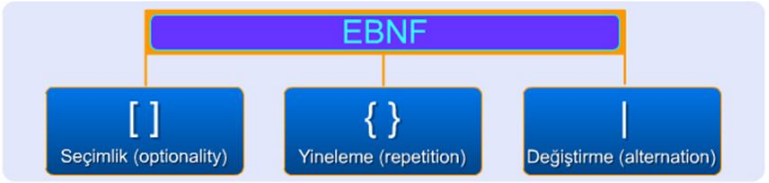
<S> → a <S> c <B> | <A> | b
<A> → c <A> | c
<B> → d | <A>

- Hangi cümleler bu gramerden türemiştir?

a. abcd
b. acccbd
c. acccbcc
d. acd
e. accc

Genişletilmiş BNF (Extended BNF)

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



EBNF – Seçim/Optionality [] ve Değiştirme/Alternation (|)

- EBNF de seçimli olan parçalar [ ] içinde tanımlanır.

<if_stmt> → **if** (<expression>) <statement> [**else** <statement>]

- BNF te ise

<if_stmt> → **if** (<expression>) <statement>
| **if** (<expression>) <statement> **else** <statement>

- Sağ tarafın(RHS) alernatif seçenekli kısımları parantez içinde dikey çubuklar ile ayrılarak gösterilir.

<term> → <term> (* | / | %) <factor>

- BNF te ise

<term> → <term> * <factor>
| <term> / <factor>
| <term> % <factor>

EBNF – Yineleme (Recursion)

- (0 ya da daha fazla) tekrarlamalar { } içinde yazılır; ya hiç yazılmaz (0) veya sonsuza kadar tekrarlanabilir.
  - <ident_list> → <identifier> {, <identifier>}
  - <ident> → letter {letter|digit}

Örnek BNF ve EBNF Grameri

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
```

Özellik Gramerleri (Attribute Grammars)

- Serbest içerik gramerleri programlama dillerinin tüm söz dizimini tanımlamazlar
  - Örneğin bir değişkenin tanımlanmadan önce kullanılamaması veya float bir sayının integer bir sayıya atanamaması
- Bu tür problemler genellikle static semantics (statik anlamı) olarak tanımlanır. Program anlamı ile ilgili bir konu olmasına rağmen, derleyici (compiler) tarafından, program derlenirken bu tür hatalar bulunmak zorundadır. Özetle sözdizimini (syntax) de etkiler. Fakat BNF ile bu tür anlamlar tanımlanamaz.
- Özellik gramerleri (attribute grammars) bu sorunu çözen bir serbest içerik grameridir (context-free grammar).
  - Programın hem sözdizimi hemde statik anlamını tanımlayarak, programın derlenmesini sağlarlar. İki özelliği vardır:
    - Statik anlamsal belirtimleri (static semantics specification)
    - Derleyici tasarımı (static semantics checking)

Özellik Gramerleri Tanımlama

- Her terminal (son olan sembol) ve non-terminal (son olmayan sembol) için özellikler (attributes) eklenmiştir.
- Attribute functions (veya semantic functions) olarak adlandırılan anlam fonksiyonları kurallar ile birlikte çalışır.
  - Bu şekilde dilin statik anlamsal bilgileri gramer kurallarına eklenmiş olur.
  - Bu anlam fonksiyonlarına predicate functions denir.

Özellik Gramerleri Tanımlama

- Bir özellik grameri aşağıdaki eklentilere sahip bir serbest içerik grameridir.
  - Her x gramer kuralı için bir A(x) özellik değerleri kümesi vardır.
  - Özellikler syntesized (ayrıştırma ağacına bilgi geçiren) veya inherited (ayrıştırma ağacında kalıtım yoluyla aşagıdaki bölgelere bilgi geçiren) özellikler olarak ikiye ayrılır.
  - Syntesized S(X) fonksiyonu ile
  - Inherited I(X) fonksiyonu ile gösterilir.

real+ int=real
real+real=real
int+int=int

Özellik Gramerlerine Örnek

- Atama gramerine değişken türü anlamının eklenmesi

<assign> → <var> = <expr>
<expr> → <var> + <var>
| <var>
<var> → A | B | C

BNF Gramer

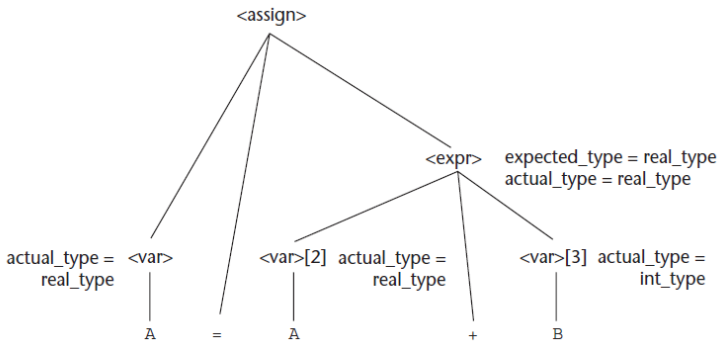
Özellik Grameri

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

Non-terminaller isimlenmiştir

Anlam fonksiyonu (predicate function)

Özellik Gramerleri Eklenmiş Ayrıştırma Ağacı (Attributed Parse Tree)



AnlamBilim / Anlamsallar (Semantics)

Anlambilim/Anlamsallar (Semantics)

- Anlamsalları tanımlamak için geçiş çapta kabul edilen tek bir notasyon yoktur.
- İşlemsel Anlamlar(Operational Semantics)
  - Bir program deyiminin veya programın, makinede çalıştırarak anlamaya çalış; ya simüle eder ya da gerçekten çalıştırır.
    - Örnegin, basit bir print metodu için yazılmış program, makinenin durumundaki (bellek-memory, yazmaçlar-registers) değişiklik ifadenin anlamını tanımlar.

İşlemsel Anlamlar (Operational Semantics)

- Bir yüksek seviye dil için işlemsel anlamları kullanmak için bir sanal makineye ihtiyaç vardır.
- Bir dönüştürücü oluştur (kaynak kodu idealleştirilmiş bilgisayar için makine koduna dönüştür) (intermediate language)
- İdealleştirilmiş bilgisayar için bir simulatör oluştur.

```
for (expr1; expr2; expr3) {
  ...
}
```

C Kodu

```
        expr1;
loop:   if expr2 == 0 goto out
        ...
        expr3;
        goto loop
out:    ...
```

İnsanlar için anlamı

```
ident = var
ident = ident + 1
ident = ident – 1
goto label
if var relop var goto label
```

Bilgisayar simülasyonu için anlamı

Aksiyomatik Anlamlar (Axiomatic Semantics)

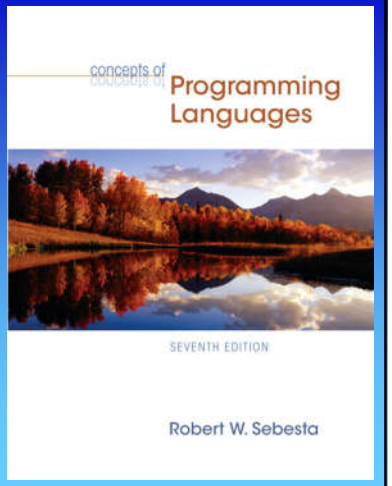
- Biçimsel mantık temellidir – matematiğe dayanır.
- Program değişkenleri ve program ifadeleri arasındaki anlamı anlamak icin kullanılır.
- Orijinal amaç: biçimsel program doğrulama – program biçimi doğru mu?
- Aksiyonlar veya girişim (inference) kuralları dildeki her ifade için tanımlanır (bu deyimlerden başka deyimlere dönüştürmeye izin verir)
- Deyimler iddialar(assertions) olarak adlandırılır.

Aksiyomatik Anlamlar (Axiomatic Semantics)

- Bir ifadeden önceki iddia (a precondition) değişkenler arasındaki ilişkileri ve kısıtları tanımlar
- Bir ifadeden sonraki iddia postcondition dır.
- En zayıf ön koşul (weakest precondition) en az kısıtlayıcı önkoşuldur ve bu koşul post condition ı garanti eder.
- Bir örnek
  - Mümkün önkoşul(precondition): {b > 10}
  - En zayıf önkoşul: {b > 0}

Bölüm 5

Adlar(Names), İlişkilendirmeler(Bindings), Tip Kontrolü(Type Checking), ve Kapsamlar(Scopes)



ISBN 0-321-33025-0

Bölüm 5 Konular

1. Giriş
2. Adlar(names)
3. Değişkenler(variables)
4. Bağlama(binding)Kavramı
5. Tip Kontrolü(Type Checking)
6. Kesin Tiplendirme(Strong Typing)
7. Tip Uyumluluğu(Type Compatibility)

Bölüm 5 Konular (devamı)

8. Kapsam(Scope) ve Ömür(Lifetime)
9. Referans(Kaynak Gösterme) Platformları(Referencing Environments)
10. Adlandırılmış sabitler(Named Constants)

5.1 Giriş

- Zorunlu Diller(Imperative Languages), von Neumann mimarisinin soyutlamalarıdır (abstractions)
  - Bellek(Memory)
  - İşlemci(Processor)
- Özelliklerle(attributes) tanımlanan değişkenler(variable)
  - Tip(Type): tasarlamak için, kapsam(Scope), ömür(Lifetime), tip kontrolü(type checking), başlatma(initialization), ve tip uyumluluğu(type compatibility) üzerinde düşünülmelidir.

5.2 Adlar(names)

- Adlar(names) için tasarım sorunları:
  - Maksimum uzunluk?
  - Bağlaç(connector) karakterleri kullanılabilir mi?
  - Adların(names) büyük-küçük harfe duyarlılığı(case sensitive) var mıdır?
  - Özel sözcükler(special words) ayrılmış sözcükler (reserved words) mi veya anahtar sözcükler midir(keywords)?

5.2 Adlar(names) (devamı)

- Uzunluk(Length)
  - Eğer çok kısa ise, anlaşılmaz
  - Dil örnekleri:
    - FORTRAN I: maksimum 6
    - COBOL: maksimum 30
    - FORTRAN 90 ve ANSI C: maksimum 31
    - Ada ve Java: limit yoktur, ve hepsi anlamlıdır(significant)
    - C++: limit yoktur, fakat konabilir

5.2 Adlar(names) (devamı)

- Bağlaçlar(Connectors)
  - Pascal, Modula-2 ve FORTRAN 77 kabul etmez
  - Diğerleri eder

5.2 Adlar(names) (devamı)

- Büyük küçük harf duyarlılığı (Case sensitivity)
  - Dezavantaj: okunabilirlik(readability) (benzer görünen adlar(names) farklıdır)
    - C++ ve Java'da daha kötüdür çünkü önceden tanımlanmış(predefined) adlar(names) karışık büyük-küçüklüktedir (örn. `IndexOutOfBoundsException`)
  - C, C++, ve Java adları(names) büyük küçük harfe duyarlıdır(case sensitive)
  - Diğer dillerdeki adlar(names) değildir

5.2 Adlar(names) (devamı)

- Özel Sözcükler(Special words)
  - Okunabilirliğe yardımcı olmak için; ifade yantümcelerini(statement clauses) sınırlamak ve ayırmak için kullanılır
  - Tanım: Bir anahtar sözcük(keyword) yalnızca belirli bir bağlamlarda(kontekstler)(contexts) özel olan sözcüktür(word) örn. Fortran'da:

    Real VarName (*Real arkasından bir ad(name) gelen bir veri tipidir(data type), bu yüzden Real bir anahtar sözcüktür (keyword))*

    Real = 3.4 (*Real bir değişkendir(variable))*
  - Dezavantaj: zayıf okunabilirlik
  - Tanım: Bir ayrılmış sözcük(reserved word) kullanıcı-tanımlı ad (a user-defined name) olarak kullanılamayan bir özel sözcüktür(word)

5.3 Değişkenler(variable)

- Bir değişken(variable) bir bellek hücresinin (memory cell) soyutlanmasıdır(abstraction)
- Değişkenler(variables), özelliklerin(attributes) bir altılısı olarak karakterize edilebilir:

  (ad(name), adres(address), değer(value), tip(type), ömür(Lifetime) ve Kapsam(Scope))
- Ad(Name) – bütün değişkenler(variables) onlara sahip değildir (adsız(anonim)(anonymous))

5.3 Değişkenler(variable) (devamı)

- Adres(Address) – ilgili olduğu bellek adresi(memory address) (/-value de denir)
  - Bir değişken(variable) çalışma süresi boyunca farklı zamanlarda farklı adreslere sahip olabilir
  - Bir değişken(variable) bir program içerisinde farklı yerlerde faklı adreslere sahip olabilir
  - Eğer iki değişken adı(variable names) aynı bellek konumuna erişmek için kullanılabiliyorsa, bunlara takma ad(diğer ad)(alias) adı verilir
  - Takma adlar(Aliases) okunabilirlik açısından zaralıdır (program okuyucuları hepsini hatırlamak zorundadır)

5.3 Değişkenler(variable) (devamı)

- Takma adlar(aliases) nasıl oluşturulabilir:
  - İşaretçiler(Pointers), referans değişkenleri(reference variables), C ve C++ bileşimleri(unions), (ve geçiş parametrelerle- Bölüm 9 da bahsedilecek)
  - Takma adlar(aliases) için orijinal gerekçelerin bazıları artık geçerli değildir; örn. FORTRAN'da belleğin yeniden kullanımı
  - Bunlar dinamik ayırma(dynamic allocation) ile değiştirilir

5.3 Değişkenler(variable) (devamı)

- Tip(Type) – değişken(variable) değerlerinin aralığını(range) ve o tipin(type) değerleri için tanımlanan işlemler kümesini belirler; kayan-nokta(floating point) olduğu durumda, tip(type) aynı zamanda duyarlılığı(precision) da belirler
- Değer(Value) – değişken(variable) ile ilişkilendirilmiş olan konumun içeriği
- Soyut bellek hücresi(Abstract memory cell) – değişken(variable) ile ilişkilendirilmiş olan fiziksel hücre veya hücreler koleksiyonu

5.4 Bağlama(binding)kavramı

- Bir değişkenin(variable) *l*–değeri(*l*–value) onun adresidir(address)
- Bir değişkenin(variable) *r*–değeri(*r*–value) onun değeridir(value)
- Tanım: Bağlama(binding) bir ilişkilendirmedir, bir özellik(attribute) ve bir varlık(entity) arasında, veya bir işlem(operation) ve bir sembol(symbol) arasındaki gibi.
- Tanım: Bağlama süresi(binding time) bir bağlamanın(binding) meydana geldiği süredir

5.4 Bağlama(binding)kavramı(devamı)

- Olası bağlama süreleri(binding times):
  - Dil tasarım süresi(design time)--örn., operatör sembollerini(operator symbols) işlemlere(operations) bağlama
  - Dil implementasyon süresi(implementation time)--örn., kayan nokta tipini(floating point type) gösterime(representation) bağlama
  - Derleme süresi(Compile time)--örn., bir değişkeni(variable) C veya Java'daki bir tipe(type) bağlama
  - Yükleme süresi(Load time)--örn., bir FORTRAN 77 değişkenini(variable) bir bellek hücresine(memory cell) bağlama (veya bir C `static` değişkenine(variable))
  - Yürütme süresi(Runtime)--örn., bir statik olmayan(nonstatic) lokal değişkeni(local variable) bir bellek hücresine(memory cell) bağlama

5.4 Bağlama(binding)kavramı(devamı)

- Tanım: Bir bağlama(binding) eğer yürütme süresinden(run time) önce meydana geliyor ve programın çalışması boyunca değişmeden kalıyorsa statiktir(static) .
- Tanım: Bir bağlama(binding) eğer yürütme süresi(run time) sırasında meydana geliyor veya programın çalışması sırasında değişebiliyorsa dinamiktir(dynamic).

5.4 Bağlama(binding)kavramı(devamı)

- Tip bağlamaları(Type bindings)
  - Bir tip nasıl belirlenir?
  - Bağlama(binding) ne zaman meydana gelir?
  - Eğer statik ise, tip ya açık(belirtik)(explicit) veya örtük(implicit) bildirim(declaration) ile belirlenebilir

5.4 Bağlama(binding)kavramı(devamı)

- Tanım: Açık bildirim(explicit declaration) değişkenlerin(variable) tiplerini tanımlamak için kullanılan bir program ifadesidir(statement)
- Tanım: Örtük bildirim(implicit declaration) değişkenlerin(variable) tiplerini belirlemek için varsayılan bir mekanizmadır (değişkenin(variable) programdaki ilk görünüşü)
- FORTRAN, PL/I, BASIC, ve Perl örtük bildirimler(implicit declarations) sağlar
  - Avantaj: yazılabilirlik(writability)
  - Dezavantaj: güvenilirlik(reliability) (Perl'de daha az problem)

5.4 Bağlama(binding)kavramı(devamı)

- Dinamik Tip Bağlama(Dynamic Type Binding)(JavaScript ve PHP)
- Bir atama ifadesiyle(assignment statement) belirlenir. örn., JavaScript

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

  - Avantaj: esneklik(flexibility) (soysal(generic) program birimleri(units))
  - Dezavantajlar:
    - Yüksek maliyet (dinamik tip kontrolü(dynamic type checking) ve yorumlama(interpretation))
    - Derleyici(compiler) ile tip hatası saptamak(Type error detection) zordur

5.4 Bağlama(binding)kavramı(devamı)

- Tip çıkarım(Type Inferencing)(ML, Miranda, ve Haskell)
  - Atama ifadesi(assignment statement) yerine, tipler(types) referansın(reference) bağlamından(context) belirlenir
- Bellek Bağlamalar(Storage bindings) & Ömür(Lifetime)
  - Ayırma(Allocation) – kullanabilir hücreler(cells) havuzundan bir hücre almak
  - Serbest Bırakma(Deallocation) – bir hücreyi(cell) havuza geri koymak
- Tanım: Bir değişkenin(variable) ömrü(Lifetime) onun belli bir bellek hücresine(memory cell) bağımlı olduğu sürece geçen zamandır

5.4 Bağlama(binding)kavramı(devamı)

- Ömürlerine(Lifetimes) göre değişkenlerin(variables) kategorileri
  - Statik(Static)—çalışma başlamadan önce bellek hücrelerine(memory cells) bağlanır ve çalışma süresince aynı bellek hücresine bağımlı kalır.
    örn. Tüm FORTRAN 77 değişkenleri(variables), C statik değişkenleri
  - Avantajlar: verimlilik(efficiency) (direk adresleme), tarih-duyarlı altprogram(history-sensitive subprogram) desteği
  - Dezavantaj: esnek (flexibility) olmaması (özyineleme(recursion) yoktur)

5.4 Bağlama(binding)kavramı(devamı)

- Ömürlerine(Lifetimes) göre değişkenlerin(variables) kategorileri
  - Yığın-dinamik(Stack-dynamic)—Bellek bağlamalar(Storage bindings) değişkenler(variables) için bildirim ifadeleri (declaration statements) incelendiği zaman oluşturulur
  - Eğer skaler(scalar) ise, adres dışındaki bütün özellikler(attributes) statik olarak bağlanmıştır
    örn. C altprogramlarındaki lokal değişkenler(local variables) ve Java metotları (methods)
  - Avantaj: özyinelemeye (recursion) izin verir; belleği korur
  - Dezavantajlar:
    - Ayırma (allocation) ve serbest bırakma(deallocation) nın getirdiği ek yük(overhead)
    - Altprogramlar(Subprograms) tarih-duyarlı olamaz (history sensitive)
    - Verimsiz referanslar (dolaylı adresleme(indirect addressing))

5.4 Bağlama(binding)kavramı(devamı)

- Ömürlerine(Lifetimes) göre değişkenlerin(variables) kategorileri
  - Açık altyığın-dinamik(Explicit heap-dynamic)—Açık(belirtik)(explicit) yönergeler(directives) tarafından ayrılır(allocated) ve serbest bırakılır(deallocated), programcı tarafından belirlenmiştir, çalışma süresi boyunca etkili olur
  - Sadece işaretçiler(pointers) veya referanslar(references) ile başvurulur
    örn. C++ 'taki dinamik nesneler (new ve delete yoluyla)
    Java daki bütün nesneler
  - Avantaj: dinamik bellek yönetimi sağlar
  - Dezavantaj: verimsiz ve güvenilmezdir

5.4 Bağlama(binding)kavramı(devamı)

- Ömürlerine(Lifetimes) göre değişkenlerin(variables) kategorileri
  - Örtük altyığın-dinamik(Implicit heap-dynamic)—atama ifadelerinin sebep olduğu ayırma(Allocation) ve serbest bırakma(deallocation)
    örn. APL 'deki bütün değişkenler(variables); Perl ve JavaScript 'deki bütün stringler ve diziler(arrays)
  - Avantaj: esneklik(flexibility)
  - Dezavantajlar:
    - verimsizdir, çünkü bütün özellikler(attributes) dinamiktir
    - Hata saptama kaybı(error detection)

5.5 Tip Kontrolü(Type checking)

- İşlenenler(Operands) ve operatörler kavramını altprogramlar(subprograms) ve atamaları(assignments) içerecek şekilde genelleştirmek
- Tip Kontrolü(Type checking) bir operatörün(operator) işlenenlerinin(operands) uyumlu tiplerde(compatible types) olmasını güvence altına alma faaliyetidir.
- Bir uyumlu tip(compatible type), ya operatör için legal olan, veya derleyicinin(compiler) ürettiği kod ile dil kuralları(rules) altında örtük(dolaylı) olarak (implicitly) legal bir tipe çevrilmesine izin verilen tiptir. Bu otomatik dönüştürmeye(conversion) zorlama(coercion) adı verilir.
- Bir tip hatası(type error), bir operatörün uygun olmayan tipteki bir işlenene(operand) uygulanmasıdır

5.5 Tip Kontrolü(Type checking) (devamı)

- Eğer bütün tip bağlamaları(type bindings) statik ise, neredeyse tüm tip kontrolü(Type checking) statik olabilir
- Eğer tip bağlamaları(type bindings) dinamik ise, tip kontrolü(Type checking) dinamik olmalıdır
- Tanım: Bir programlama dilinde eğer tip hataları(type errors) her zaman saptanıyorsa, o dil kesin/kuvvetli tiplendirilmiştir(strongly typed)

5.6 Kesin Tiplendirme(Strong Typing)

- Kesin tiplendirmenin(strong typing) avantajı: değişkenlerin(variable) yanlış kullanılmasıyla oluşan tip hatalarını engeller. Dil örnekleri:
  - FORTRAN 77 böyle değildir: parametreler, `EQUIVALENCE`
  - Pascal böyle değildir: variant records
  - C ve C++ değildir: parametre tip kontrolü(parameter type checking) önlenebilir; bileşimler(unions) tip kontrollü değildir(type checked)
  - Ada hemen hemen böyledir, (`UNCHECKED CONVERSION` kaçamak noktasıdır(loophole))

    (Java buna benzerdir)

5.6 Kesin Tiplendirme(Strong Typing) (devamı)

- Zorlama kuralları(Coercion rules) kesin tiplendirmeyi(strong typing) fazlasıya etkiler—oldukça yavaşlatabilirler (C++ 'a karşı Ada)
- Java'nın C++'ın atama zorlamalarının(assignment coercions) yalnızca yarısına sahip olmasına rağmen, onun kesin tiplendirmesi(strong typing) Ada'nınkinden çok daha az etkilidir

5.7 Tip Uyumluluğu(Type Compatibility)

- Öncelikle yapısal tiplerle(structured types) ilgileniyoruz
- Tanım: Ad tipi uyumluluğu(Name type compatibility),iki değişkenin(variable) aynı bildirimde(declaration) veya aynı tip adını(type name)kullanan bildirimlerde olması durumunda uyumlu tiplere sahip olması anlamına gelir
- Gerçekleştirilmesi kolaydır fakat çok kısıtlayıcıdır:
  - İnteger tiplerinin altaralıkları(subranges) integer tipleriyle uyumlu(compatible) değildir
  - Formal parametreler onlara karşılık gelen güncel(actual) parametreler ile aynı tipte olmalıdır (Pascal)

5.7 Tip Uyumluluğu(Type Compatibility) (devamı)

- Yapı(Structure) tipi uyumluluğu(compatibility) iki değişkenin(variable) eğer tipleri özdeş (identical) yapılara sahipse uyumlu tiplere sahip olması anlamına gelir
- Daha esnektir, fakat gerçekleştirilmesi zordur

5.7 Tip Uyumluluğu(Type Compatibility) (devamı)

- İki yapısal tipin(structured types) problemini düşünelim:
  - Eğer iki tutanak(kayıt) tipi(record types) yapısal olarak aynı ise fakat farklı alan adları(field names) kullanıyorlarsa bunlar uyumlu mudur?
  - Eğer iki dizi tipi (array types) altsimgelerinin(subscripts) farklı olması dışında aynıysa uyumlu mudur?
    (örn. [1..10] ve [0..9])
  - Eğer iki sayım tipinin(enumeration types) bileşenlerinin yazılışları farklı ise bunlar uyumlu mudur?
  - Yapısal tip(structural type) uyumluluğuyla, aynı yapıya ait farklı tipleri ayırt edemezsiniz (örn. Farklı hız birimleri, ikisi de float)

5.7 Tip Uyumluluğu(Type Compatibility) (devamı)

- Dil örnekleri:
  - Pascal: genellikle yapı(structure), fakat bazı durumlarda ad(name) kullanılır (formal parametreler)
  - C: yapı, tutanaklar(kayıtlar)(records) için hariç
  - Ada: adın(name) kısıtlanmış biçimi
    - Türetilmiş(Derived) tipler aynı yapıdaki(structure) tiplerin farklı olmasına izin verir
    - Anonim tiplerin hepsi benzersizdir(unique), hatta:

```
A, B : array (1..10) of INTEGER;
```


5.8 Kapsam(Scope)

- Bir değişkenin(variable) kapsamı(Scope) onun görünür(visible) olduğu ifadelerin(statements) aralığıdır(range)
- Bir program biriminin lokal olmayan değişkenleri (nonlocal variables) görünür fakat belirtilmemiş(declared) olan değişkenlerdir
- Bir dilin kapsam kuralları(Scope rules) adlara(names) referansların(references) değişkenlerle(variables) nasıl ilişkilendirildiğini belirler

5.8 Kapsam(Scope) (devamı)

- Statik Kapsam(Static Scope)
  - Program metnine(text) dayalıdır
  - Bir değişkene(variable) bir ad referansı(name reference) bağlamak için, siz (veya derleyici(compiler)) belirtimi (declaration) bulmalısınız
  - Arama işlemi: bildirimler(declarations) aranır, ilk önce lokal olarak, sonra gittikçe daha geniş çevreleyen (enclosing) kapsamlarda(scopes), verilen ad(name) için bir tane bulunana kadar
  - Çevreleyen statik kapsamlar(Enclosing static scopes)(belirli bir kapsama(specific scope)) onun statik ataları(static ancestors) denir; en yakın statik ataya(static ancestor) statik ebeveyn(static parent) adı verilir

5.8 Kapsam(Scope) (devamı)

- Değişkenler(variables) bir birimden aynı isimli “daha yakın” ("closer") bir değişkene(variable) sahip olarak saklanabilir
- C++ ve Ada bu “gizli”("hidden") değişkenlere(variables) erişime izin verir
  - Ada’da: `unit.name`
  - C++’da: `class_name::name`

5.8 Kapsam(Scope) (devamı)

- Bloklar(Blocks)
  - Program birimleri içinde statik kapsamlar(static scopes) oluşturmanın bir metodu-- ALGOL 60’dan
  - örnekler:

C ve C++:
```
for (...)
{
  int index;
  ...
}
```

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


5.8 Kapsam(Scope) (devamı)

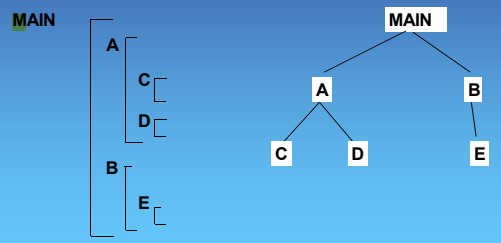
- Statik Kapsamanın(Static Scoping) değerlendirmesi
- Örneğe bakalım:

  Varsayalım ki MAIN, A ve B yi çağırır

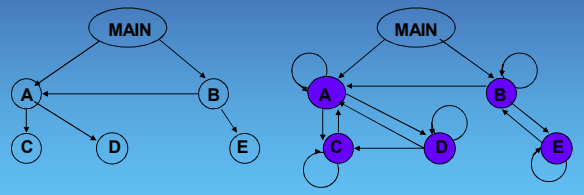
  A, C ve D yi çağırır

  B, A ve E yi çağırır

Statik Kapsam(Static Scope) Örneği



Statik Kapsam(Static Scope) Örneği



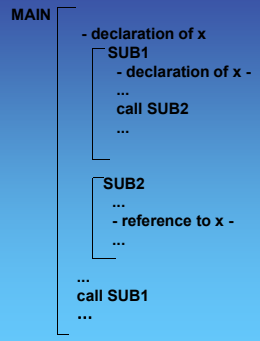
Statik Kapsam(Static Scope)

- Şartın değiştiğini varsayalım öyle ki D, B deki bazı veriye erişmek zorunda
- Çözümler:
  - D'yi B'nin içine koy (fakat o zaman C artık onu çağıramaz ve D, A'nın değişkenlerine(variables) erişemez)
  - D'nin ihtiyacı olan veriyi B'den MAIN'e taşı (fakat o zaman bütün prosedürler(procedures) onlara erişebilir)
- Prosedür erişim için aynı problem
- Sonuçta: statik kapsama(static scoping) çoğunlukla birçok globale teşvik eder

5.8 Kapsam(Scope) (devamı)

- Dinamik Kapsam(Dynamic Scope)
  - Program birimleri sıralarının çağrılmasına dayalıdır, onların metinsel düzenine(textual layout) değil, zamansala(temporal) karşı mekansal(uzaysal)(spatial))
  - Değişkenlere(variable) referanslar, bu noktaya kadar çalışmayı zorlamış altprogram çağrıları zincirinden geriye doğru arama yapma yoluyla bildirimlere(declarations) bağlıdır

Kapsam(Scope) Örneği



MAIN SUB1'i çağırır
SUB1 SUB2'yi çağırır
SUB2 x'i kullanır

Kapsam(Scope) Örneği

- Statik kapsama(Static scoping)
  - x'e referans MAIN'in x'inedir
- Dinamik Kapsama(Dynamic scoping)
  - x'e referans SUB1'in x'inedir
- Dinamik kapsamanın değerlendrilmesi:
  - Avantaj: elverişlilik
  - Dezavantaj: zayıf okunabilirlik(readability)

5.10 Referans Platformları(Referencing Environments)

- Tanım: Bir ifadenin (statement) referans platformu(referencing environment) ifadede görünen bütün adların(names) koleksiyonudur
- Bir statik kapsamlı dil(static-scoped language), lokal değişkenler(local variables) artı bütün çevreleyen (enclosing) kapsamlardaki (scopes) görünür değişkenlerin(variables) tümüdür
- Bir altprogramın(subprogram) çalıştırılması başlamışsa ama henüz bitmemişse o altprogram aktiftir
- Bir dinamik kapsamlı dilde(dynamic-scoped language), referans platformu lokal değişkenler(local variables) artı tüm aktif altprogramlardaki(subprograms) bütün görünür değişkenlerdir(variable)

5.11 Adlandırılmış sabitler (Named Constants)

- Tanım: Bir adlandırılmış sabit(named constant), sadece belleğe bağlı olduğu zaman bir değere bağlanmış olan değişkendir
- Avantajlar: okunabilirlik ve değiştirilebilirlik(modifiability)
- Programları parametrelerle ifade etmek için kullanılır
- Adlandırılmış sabitlere(named constants) değer bağlama(binding) statik (called manifest constants) veya dinamik olabilir
- Diller:
  - Pascal: sadece kalıp deyimler (literals)
  - FORTRAN 90: sabit-değerli deyimler
  - Ada, C++, ve Java: herhangi bir tipteki deyimler

Değişken başlatma (variable initialization)

- Tanım: Bir değişkeni(variable) belleğe bağlı olduğu sırada bir değere(value) bağlamaya (binding) başlatma(initialization) denir
- Başlatma(Initialization) genellikle bildirim ifadesinde(declaration statement) yapılır

örn., Java

```
int sum = 0;
```

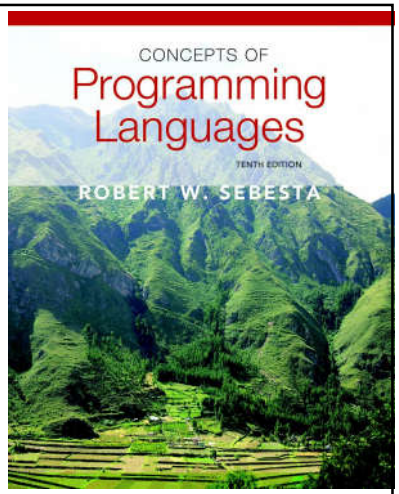

Özet

- Büyük küçük harf duyarlılığı(Case sensitivity) ve adların(names) özel sözcüklerle(special words) ilişkisi adların(names) tasarım sorunlarını ifade eder
- Değişkenler(variables) altıkatlı ile karakterize edilir: ad(name), adres(address), değer(value), tip(type), ömür(Lifetime), kapsam(Scope)
- Bağlama(binding) özelliklerle(attributes) program varlıklarının(entities) birleştirilmesidir
- Skaler(Sayısal) değişkenler(Scalar variables) şu şekilde sınıflandırılır: statik(static), yığın-dinamik(stack dynamic), açık-altyığın dinamik(explicit heap dynamic), örtük altyığın-dinamik(implicit heap dynamic)
- Kesin tiplendirme(Strong typing) bütün tip hatalarını saptamak anlamına gelir

skaler deikenler:
1. statik
2. yigin-dinamik
3.acik- altyigin dinamik
4. örtük altyigin dinamik

Bölüm 6

Veri Tipleri



6. Bölümün Başlıkları

- Giriş
- İlkel Veri Tipleri
- Karakter (String) Veri Tipleri
- Kullanıcı Tanımlı Sıra Tipleri
- Dizi Tipleri
- İlişkili Diziler
- Kayıt Tipleri
- Demet Tipler
- Liste Tipleri
- Birleşim Tipleri
- Pointer ve Referans Tipleri
- Tip Kontrolü
- Güçlü Tipleme
- Tip Eşitleme
- Teori ve Veri Tipleri

Giriş

- Bir veri tipi(data type) bir veri nesneleri(data objects) koleksiyonunu ve bu nesnelerin bir takım ön tanımlı işlemlerini (predefined operations) tanımlar.
- Bir tanımlayıcı bir değişken niteliklerinin topluluğudur.
- Bir nesne bir kullanıcı tanımlı (soyut veri) türünde bir örnek temsil eder.
- Bütün veri tipleri(data types) için bir tasarım meselesi:

  Hangi işlemler tanımlanmıştır ve nasıl belirlenir?

İlkel Veri Tipleri

- Neredeyse tüm programlama dilleri ilkel veri türleri kümesi sağlar.
- İlkel Veri Tipleri: Diğer veri tipleri cinsinden tanımlanmayan veri tipleridir.
- Bazı ilkel veri tipleri sadece donanım yansımalarıdır.
- Diğerleri bunların uygulanması için sadece küçük olmayan donanım desteği gerektirir.

İlkel Veri Tipleri: Integer

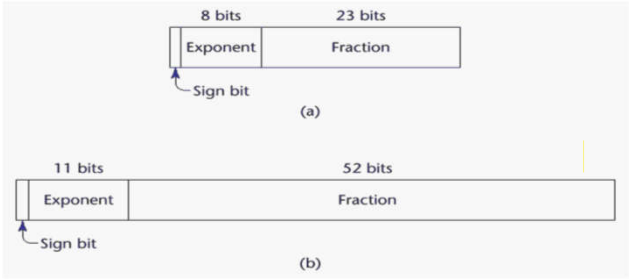
- Genellikle her zaman donanımın(hardware) tam yansımasıdır, bu yüzden eşlenme(mapping) önemsizdir.
- Bir dilde en çok sekiz farklı tamsayı (integer) tipi olabilir
- Java'nın integer tipi veri tipleri: **`byte, short, int, long`**

İlkel Veri Tipleri: Kayan Nokta

- Reel Sayıları yalnızca yaklaşım olarak modeller.
- Bilimsel kullanım için olan diller en az iki kayan nokta tipini destekler (Örneğin, **`float`** and **`double`**); Bazen daha fazla.
- Genellikle aynen donanım(hardware) gibidir, fakat her zaman değil
- IEEE 754 Kayan Nokta

Standartı

IEEE 754 Kayan Nokta Standartı



İlkel Veri Tipleri: Kompleks

- Bazı dilleri bu veri tipini destekler, Örneğin: C99, Fortran ve Python
- Her değer iki kısımdan oluşur, birisi reel değer diğer kısım ise imajiner değerdir.
- Python'daki formu aşağıdaki örnekte verilmiştir.:

  (7 + 3j), 7 sayının reel değeri, 3 ise imajiner değeridir.

İlkel Veri Tipleri: Ondalık(Decimal)

- Ticari uygulamalar için kullanılır(para)
  - COBOL temellidir.
  - C# dili decimal veri tipi sunar.
- Sabit ondalık sayıları muhafaza ederler. (BCD) (Ondalıklı sayıların ikilik kodlanması)
- *Avantaj*: Doğruluk
- *Dezavantaj*: Sınırlı aralık, belleği gereksiz harcama.

İlkel Veri Tipleri: Boolean

- En basit veri tipidir.
- Değer aralığında yalnızca iki değer bulunmaktadır. Bunlar true (doğru) ve false (yanlış)'tır.
- Bitler olarak uygulanabilir, fakat çoğu zaman byte kullanılır.
  - Avanaj: Okunabilirlik

İlkel Veri Tipleri: Karakter

- Sayısal kodlama olarak saklanırlar.
- En sık kullanılan kodlama: ASCII (8 bitlik kodlamadır.)
- 16 bitlik alternatif kodlama: Unicode (UCS-2)
  - Çoğu doğal dildeki karakterleri içerir.
  - Başlangıçta Java'da kullanıldı.
  - C# ve JavaScript'de Unicode'u destekleyen diller arasındadır.
- 32-bit Unicode (UCS-4)
  - 2003'te oluşturulmuştur ve fortran tarafından desteklenir

Karakter String Tipleri

- Değerler karakter (char) dizileridir. Örn: char dizi[10]=string deger;
- Tasarım Sorunları:
  1. Bu bir ilkel(primitive) tip midir yoksa sadece bir çeşit özel dizi midir (array)?
  2. Stringlerin uzunluğu statik mi yoksa dinamik mi olmaldır?

Karakter String Tip İşlemleri

- Tipik İşlemler:
  - Atama(Assignment) ve kopyalama
  - Karşılaştırma (Kıyaslama) (Comparison) (=, >, vs.)
  - Birleştirme(Catenation)
  - Altstring referansı (Substring reference)
  - Desen Eşleme (Pattern matching)

Belirli Dillerdeki Karakter String Tipleri

- C ve C++
  - İlkel değil
  - Char dizilerini ve işlemleri sağlayan fonksiyonların kütüphane fonksiyonları kullanır.
- SNOBOL4 (String işleme dili)
  - İlkel
  - Eşleştirme, ayrıntılı desenleme (örüntü tanıma) dahil bir çok işlemi yerine getirebilir.
- Fortran ve Python
  - Atama ve çeşitli operatörleri kullanan ilkel bir tiptir.
- Java
  - String class ile ilkel bir tiptir.
- Perl, JavaScript, Ruby ve PHP

  -Düzenli deyimler kullanılarak string tipinde desen eşleştirme sağlar.

Karakter String Uzunluk Seçenekleri

- Statik: COBOL,Fortran, Java'nın `String` sınıfı
- *Sınırlı Dinamik Uzunlık*: C and C++
  - Bu dillerde, uzunluğu temin etmekten ziyade string karakterlerin sonunu göstermek için özel bir karakter kullanılır.
- *Dinamik* : SNOBOL4, Perl, JavaScript
- Ada, tüm string uzunluk seçeneklerini destekler.

Karakter String Tip Değerlendirmesi

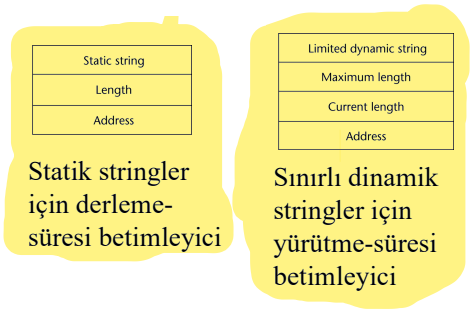
Yazılabilirliğe yardımcıdır

- Statik uzunluklu(Static length) bir ilkel(primitive) tip olarak, temin edilmesi ucuz--neden kullanmayalım ?
- Dinamik uzunluk(Dynamic length) iyidir, fakat masrafa değer mi?

Karakter String Uygulaması

- İmplementasyon (Uygulama):
  - Statik Uzunluk(Static length): Derleme-süresi betimleyicisi (compile-time descriptor)
  - Sınırlı dinamik uzunluk(Limited dynamic length): Uzunluk için bir yürütme-süresi betimleyicisine (run-time descriptor) ihtiyaç duyabilir (fakat C ve C++ da değil)
  - Dinamik Uzunluk(Dynamic length) : Yürütme-süresi betimleyicisine(run-time descriptor) ihtiyaç duyar; atama/atamayı kaldırma (allocation/deallocation) en büyük uygulama problemidir.

Derleme ve Çalışma Zamanı Tanımlayıcıları



Kullanıcı Tanımlı Sıralı Tipler

- Bir sıralı tip(ordinal type), mümkün olan değerler(values) aralığının(range) pozitif tamsayılar(integers) kümesi ile kolayca ilişkilendirilebildiği tiptir.
- Java dilindeki ilkel tip örnekleri
  - **integer**
  - **char**
  - **boolean**

Enumeration(Sayım listesi) Tipleri

- Sabit olarak isimlendirilen tüm olası değerler, tanımda sağlanır.
- C# örneği

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- Tasarım Problemi:
  - Bir sembolik sabitin(symbolic constant) birden fazla tip tanımlaması içinde yer almasına izin verilmeli midir? Eğer böyle ise o sabitin ortaya çıkmasının tipi nasıl kontrol edilir?
  - Sayım listesi değerleri tamsayıya zorlanır mı?
  - Diğer bir tip bir sayma tipine zorlanır mı?

Sayım Listesi Tip Değerlendirmesi

- Okunabilirliğe yardım, Örneğin bir sayı olarak bir renk koduna gerek yoktur.
- Güvenilirliğe yardım, Örneğin, derleyici aşağıdakileri kontrol edilebilir:
  - İşlemleri (Renk eklenmesine izin vermez)
  - Sayım listesi değişkeninin dışından bir değer atanmasına izin vermez
  - Ada, C# ve Java 5.0; C++'tan daha iyi sayım listesi desteği sağlar. Çünkü bu dillerdeki sayım listesi değişkenleri tamsayı tiplere zorlanmaz.

Altaralık (Subrange) Tipleri

- Sıralı bir tipteki bir sıralanmıi ardışık alt dizi.
  - Örnek: 12..18 tamsayı tipinin alt aralığıdır.
- Ada'nın tasarımı

```
Günler tipi (mon, tue, wed, thu, fri, sat, sun);
Günler tipinin altaralık tipi haftaiçi günler
  mon..fri;
  Index alt tipi aralığı 1..100 tamsayıdır.
Day1: Days;
Day2: Weekday;
Day2 := Day1;
```


Altaralık Tipleri Değerlendirmesi

- Okunabilirliğe yardım
  - Okuyucuların kolayca görebileceği altaralık değişkenlerini yalnızca belirli aralıkta saklayabiliriz.
- Güvenilirlik
  - Belirlenen değerler dışında altaralık değişkene farklı değerler atamak hata olarak algılanır.

Kullanıcı Tanımlı Sıralı Tiplerin Uygulanması

- Sıralı listeleme (Enumaration) tipleri, tamsayı olarak uygulanır.
- Altaralık tipleri, altaralık değişkenlerine atamaları sınırlamak için (derleyici tarafından) kod yerleştirilmiş ebeveyn(parent) tiplerdir.

Dizi Tipleri

- Bir dizi(array), homojen veri elemanların bir kümesidir, elemanlardan her biri kümedeki birinci elemana göre olan pozisyonuyla tanımlanır.

Dizi Tasarım Problemleri

- 1. Altsimgeler(indeks)(subscripts) için hangi tipler legaldir?
- 2. Eleman referansları aralığındaki altsimgeleme ifadeleri(subscripting expressions) kontrol edilmiş midir?
- 3. Altsimge aralıkları ne zaman bağlanır(bound)?
- 4. Ayırma (allocation) ne zaman olur?
- 5. Altsimgelerin(subscripts) maksimum sayısı nedir?
- 6. Dizi(array) nesneleri(objects) başlatılabilir mi (initialized)?
- 7. Herhangi bir çeşit kesite(slice) izin verilmiş midir?

Dizi İndeksleme

- İndeksleme(Dizin oluşturma)(Indexing) indislerden(indices) elementlere eşleştirme(mapping) yapmaktır

map(array_name, index_value_list) → an element

- İndeks Sentaksı
  - FORTRAN, PL/I, Ada parentezler kullanır
  - Diğer dillerin çoğu köşeli parantez (brackets) ([ ]) kullanır

Dizi İndeksleme (Altsimge) Tipleri

- FORTRAN, C,C#: Yalnızca integer veri tipini kullanır
- Ada: integer yada enumeration (Boolean ve char veri tiplerini içerir.)
- Java: Sadece integer veri tipi.
- Indeks aralık kontrolü
  - C, C++, Perl, ve Fortran özel aralık kontrolü yapmaz.
  - Java, ML, C# dillerinde özel aralık kontrolleri vardır.
  - Ada, Varsayılan aralığı gerektiren kontrolu yapar, ama bazen bu kontrol devre dışı olabilir.

İndis Bağlama ve Dizi Kategorileri

- Dizilerin(Arrays) Kategorileri(altsimge bağlamaya(subscript binding) ve belleğe(storage) bağlamaya dayalıdır)
  1. Statik – altsimgelerin(subscripts) aralığı(range) ve bellek bağlamaları(storage bindings) statiktir
     örn. FORTRAN 77, Ada 'daki bazı diziler(Arrays)
  - Avantaj: uygulama verimliliği(execution efficiency) (ayırma(atama)(allocation) veya serbest bırakma(atamayı kaldırma)(deallocation) yoktur)

İndis Bağlama ve Dizi Kategorileri (Devamı)

2. Sabit yığın dinamik (Fixed stack dynamic)– altsimgelerin (subscripts) aralığı(range) is statik olarak bağlanmıştır, fakat bellek(storage) işlenme zamanında bağlanır
   - örn. Çoğu Java lokalleri, ve `static` olmayan C lokalleri

     Avantaj: alan verimliliği(space efficiency)

İndis Bağlama ve Dizi Kategorileri (Devamı)

3. Yığın Dinamik(Stack-dynamic) – aralık (range) ve bellek(storage) dinamiktir, fakat sonra değişkenin(variable) ömrüne göre(lifetime) sabitlenir
   - örn. Ada bloklar tanımlar

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

   - Avantaj: esneklik – dizi(array) kullanılmaya başlanmadan önce boyutu(size) bilinmek zorunda değildir.

İndis Bağlama ve Dizi Kategorileri (Devamı)

4. Heap-dynamic(Dinamik Öbekler) – altsimge(subscript) aralığı (range) ve bellek bağlamalar(storage bindings) dinamiktir ve sabitlenmiş değildir
   - Avantaj: esneklik(diziler programın çalıştırılması esnasında büyültülebilir veya küçültülebilir.)
   - APL'de, Perl, ve JavaScript, diziler(Arrays) ihtiyaca göre büyüyüp küçülebilir
   - Java ve C#'ta, bütün diziler(Arrays) birer nesnedir (heap-dynamic)

İndis Bağlama ve Dizi Kategorileri (Devamı)

- C ve C++'ta statik niteleyiciler içeren diziler statiktir.
- C ve C++'ta statik niteleyiciler içermeyen diziler sabit yığın (fixed stack) –dinamiktir.
- C ve C++ sabit öbek(fixed heap) ve dinamik dizileri destekler.
- C# 2. dizi sınıfı olan ArrayList'i destekler ve sabit öbeklerle dinamik diziler oluşturulabilir.
- Perl, JavaScript, Python, ve Ruby dinamik öbek dizilerini destekler(sağlar).

Dizi Oluşturma

- Bazı diller dizi oluşturma ve oluşturulan diziye bellek alanı ayırma işlemini aynı anda yapabilir.
  - C, C++, Java, C# örneği

```
int list [] = {4, 5, 7, 83}
```

  - C ve C++'ta karakter dizisi oluşturma

```
char name [] = "Asaf Varol";
```

  - C ve C++'ta pointerlar yardımıyla string dizisi oluşturma

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java'da string nesnesi oluşturma

```
String[] names = {"Bob", "Jake", "Joe"};
```


Heterojen Diziler

- Heterojen dizi, elemanları aynı tipten olması gerekmeyen dizilerdir.
- Perl, Python, JavaScript ve Ruby tarafından desteklenir.
- Diğer dillerde heterojen diziler yerine struct(yapılar) kullanılır, fakat yapılar heterojen diziyi tam manasıyla karşılayamazlar.

Dizi Oluşturma

- C-tabanlı diller
  - `int list [] = {1, 3, 5, 7}`
  - `char *names [] = {"Mike", "Fred", "Mary Lou"};`
- Ada
  -
```
List : array (1..5) of Integer :=
    (1 => 17, 3 => 34, others => 0);
```
- Python
  - Liste Kapsamları

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
puts [0, 9, 36, 81] in list
```


Dizi İşlemleri

- APL vektörler ve matrisler için hem en güçlü dizi işleme işlemleri hem de tekli operatör desteği (örneğin, kolon elemanları tersine çevirmek için) sağlar
- Ada yalnızca dizilerde atama,birleştirme ve ilişkisel operatör işlemlerine izin verir.
- Pyton'nun dizi atamaları yalnızca referans değişikliği işlemlerini yapmasına rağmen eleman üyelik sistemiyle Ada'nın sağladığı tüm işlemleri yapabilir.
- Ruby dizilerde atama,birleştirme ve ilişkisel operatör işlemlerine izin verir
- Fortran iki dizi arasındaki element işlemlerini destekler
  - Örneğin Fortrandaki + operatörü iki dizi çiftleri arasındaki elemanları toplar.

Dikdörtgen(Düzenli) ve Tırtıklı (Düzensiz) Diziler

- Bir dikdörtgen dizi satırları tüm unsurlarının aynı sayıda ve tüm sütun elemanlarının aynı sayıya sahip olduğu çok boyutlu dizidir.
- Tırtıklı matrisler ise her satırında aynı sayıda eleman bulunmayan dizilerdir.
  - Olası çoklu-boyutlu zaman dizileri aslında dizinler olarak görünür
  - C, C++, ve Java tırtıklı(düzensiz) dizileri destekler
- Fortran, Ada, ve C# dikdörtgen dizileri destekler (C# aynı zamanda tırtıklı dizileri de destekler)

Kesitler

- Kesitler(slices)
  - Kesit(slice) , bir dizininin bir kısım altyapısıdır (substructure) ; bir referanslama mekanizmasından fazla birşey değildir
  - Kesitler(slices) sadece dizi işlemleri olan diller için kullanışlıdır.

Kesit Örnekleri

- Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

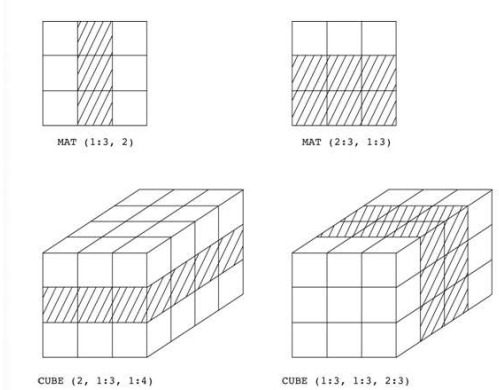
`vector (3:6)` dizinin 3 elemanını temsil eder.

`mat[0][0:2]` dizinin ilk satırındaki ilk elemandan 3. elemana kadar olanı gösterir.

- Ruby slice metot olarak kesiti destekler.

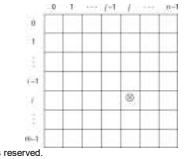
`list.slice(2, 2)` örneğinde 2. elemandan 4. elemana kadar olanları kapsar.

Kesitler



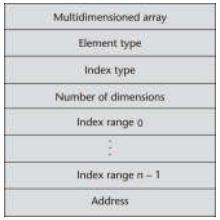
Dizi Uygulamaları

- Erişim fonksiyonları(Access function) altsimge(subscript) ifadelerini dizideki bir adrese eşler(map)
- Tek boyutlu diziler için erişim fonksiyonu:
  address(list[k]) = address (list[lower_bound]) + ((k-lower_bound) * element_size)



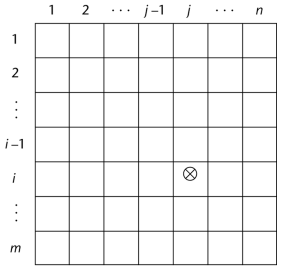
Çok Boyutlu Dizilere Erişim

- Yaygın olarak kullanılam metotlar:
  - Satıra göre sıralama – bir çok dilde kullanılan metottur
  - Sutüna göre sıralama – Fortran tarafından kullanılır
  - Çok boyutlu dizilerin derleme süreleri yan taraftaki şekilde verilmiştir.

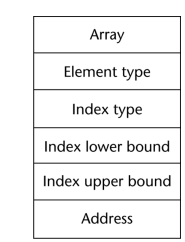


Çok Boyutlu Dizilerde Eleman Yerleştirme

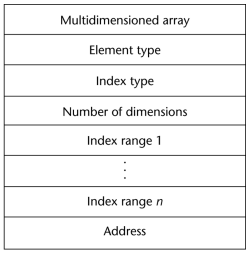
- Genel format
  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I – row_lb) * n) + (j – col_lb)) * element_size



Derleme Süresi Betimleyiciler



Tek Boyutlu Dizi



Çok Boyutlu Dizi

İlişkili Diziler

- Bir ilişkili dizi(associative array), anahtar(key) adı verilen eşit sayıda değerlerle indekslenmiş veri elemanlarının sırasız bir koleksiyonudur.
- Tasarım Problemleri:
  1. Elemanlara referansın şekli nedir?
  2. Boyut statik midir yoksa dinamik mi?

Perl'de İlişkili Diziler

- İsimler% ile başlar; değişmezleri parantez tarafından ayrılmış

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" =>
  65, …);
```

- İndisleme küme parantezi ve $ kullanılarak yapılır.

```
$hi_temps{"Wed"} = 83;
```

  - Elemanlar delete komutuyla silinir.

```
delete $hi_temps{"Tue"};
```


Kayıt Tipleri

- Bir kayıt ayrı eleman isimleri tarafından tanımlandığı bir veri elemanlarının heterojen toplamıdır.
- Tasarım problemleri:
  1. Referansların şekli nedir?
  2. Hangi birim işlemler tanımlanmıştır?

COBOL'da Kayıt Tanımlanması

- COBOL yuvalanmış kayıtları göstermek için seviye numaraları kullanır; diğerleri için özyinelemeli tanımı kullanabilirsiniz.
-
```
01 EMP-REC.
    02 EMP-NAME.
       05 FIRST PIC X(20).
       05 MID   PIC X(10).
       05 LAST  PIC X(20).
    02 HOURLY-RATE PIC 99V99.
```


Ada ile Kayıt Tanımlanması

- Kayıt yapıları ortagonol bir şekilde gösterilir.

```
type Emp_Rec_Type is record
    First: String (1..20);
    Mid: String (1..10);
    Last: String (1..20);
    Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```


Kayıt Referansları

- Kayıt alanındaki referanslar
  1. COBOL
  field_name OF record_name_1 OF ... OF record_name_n
  2. Others (dot notation)
  record_name_1.record_name_2. ... record_name_n.field_name
- Kaliteli referanslar(references) bütün kayıt adlarını(record names) içermelidir
- Eliptik referanslar(Elliptical references), referans(reference) belirsiz olmadığı(unambiguous) sürece kayıt adlarının (record names) ihmal edilmesine izin verir
  FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

Kayıt İşlemleri

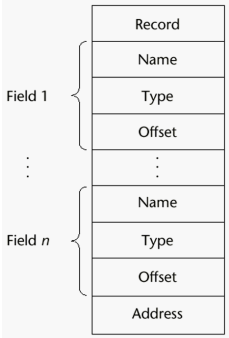
1. Atama(Assignment)
   - Pascal, Ada, ve C tipleri özdeş(identical) ise izin verir
   - Ada'da, sağ kısım(RHS) bir toplam sabit(aggregate constant) olabilir
2. Başlatma(Initialization)
   - Ada'da izin verilmiştir, toplam sabit(aggregate constant) kullanarak
3. Kıyaslama(Comparison)
   - Ada'da, = ve /=; bir operand toplam sabit(aggregate constant) olabilir
4. **MOVE CORRESPONDING**
   - COBOL'de – kaynak kayıttaki(source record) bütün alanları(fields) hedef kayıttaki(destination record) aynı ada sahip alanlara(fields) taşır

Evaluation and Comparison to Arrays

1. Dizi(array) elemanlarına erişim kayıt alanlarına (record fields) erişimden daha yavaştır, çünkü altsimgeler(subscripts) dinamiktir (alan adları(field names) statiktir)
2. Dinamik altsimgeler(subscripts) kayıt alanına(record field) erişimle kullanılabilirdi, fakat o zaman tip kontrolüne(type checking) izin vermeyecekti ve çok daha yavaş olacaktı

Kayıt Tipinin Uygulanması

Kayıtların başlangıcına göre
Ofset adresi her alanı ile ilişkilidir.



Demet (Tuple) Tipi

- Demet veri tipi kayıt veri tipine benzeyen bir veri tipidir.
- Python,ML ,F#,C#(.Net 4.0 ile birlikte)'ta kullanılır.Fonksiyonlara birden fazla değer döndürür.
  - Python
    - Listelerle yakından ilişkili ama değiştirilemez
    - Demet oluşturma

      `myTuple = (3, 5.8, 'apple')`

      İndislerini 1'den başlayaraka referanslandırır.

      + operatörünü kullanır ve del komutuyla silinir.

Demet(Tuple) Tipi (Devamı)

- ML

  ```
  val myTuple = (3, 5.8, 'apple');
  ```

  - Takipçilere erişim:

    `#1(myTuple)` demetin ilk elemanı
  - Yeni bir demet aşağıdaki gibi tanımlanır.

    ```
    type intReal = int * real;
    ```
- F#

  ```
  let tup = (3, 5, 7)
  let a, b, c = tup
  ```

Liste Tipleri

- LISP ve Şema listeleri parantez ayracıyla kullanılırlar ve elemanlar arasına virgül konulmaz.
  `(A B C D)` and `(A (B C) D)`
- Veri ve kod aynı formdadır.
  Veri, `(A B C)`
  Kod, `(A B C)` bir fonksiyonun parametreleri
- Yorumlayıcı hangi listeye ihtiyaç duyacağını bilmelidir. Burdaki karmaşıklığı ortadan kaldırmak için veri listelerinin önüne ‘ işareti konur.
  `'(A B C)` is data

Liste Tipleri (devamı)

- Şema içerisindeki Liste operatörleri
  - `CAR` listesi ilk elemanını döndürürse
    `(CAR '(A B C))` returns `A`
  - `CDR` ilk elemanı söküldükten sonra kendi listesinde parametresi kalanı verir.
    `(CDR '(A B C))` returns `(B C)`
  - `CONS` Yeni bir liste yapmak için ikinci parametre, bir liste içine ilk parametre koyar.
    `(CONS 'A (B C))` returns `(A B C)`
  - `LIST` yeni bir liste döndürür.
    `(LIST 'A 'B '(C D))` returns `(A B (C D))`

Liste Tipleri (devamı)

- ML’de Liste Operatörleri
  - Listeler parantez içinde yazılır ve elemanları virgüllerle ayrılır.
  - Liste elemanları aynı veri tipinde olmalıdır.
  - `CONS` fonksiyonu ML dilinin binary operatörüdür, `::`
    `3 :: [5, 7, 9]` dönüşür `[3, 5, 7, 9]`
  - CAR ve CDL fonksiyonları burda hd ve tl olarak adlandırılır.

Liste Tipleri (devamı)

- F# Listeler
  - ML dilindeki liste yapısına benzer, yalnızca elemanların ayrılmasıyla hd ve tl metotları List sınıfının içinde yer alır
- Python Listeler
  - Liste veri tipi genelde python dizileri olarak sunulur
  - Genelde LISP,ML,F# ve Python listeleri birbirine benzer.
  - Listedeki elemanlar değişik veri tiplerinden olabilirle
  - Liste oluşturulması aşağıdaki gibidir.

```
myList = [3, 5.8, "grape"]
```

Liste Tipleri (continued)

- Python Listeler (devamı)
  - Liste indisi "0"dan başlar ve sonradan değiştirilebilir.

```
x = myList[1]
```
Sets `x` to `5.8`
  - Liste elemanları del komutuyla silinir.

```
del myList[1]
```
  - Liste anlamları – küme gösterimiyle temsil edilebilir.

```
[x * x for x in range(6) if x % 3 == 0]
```
`range(12)` creates `[0, 1, 2, 3, 4, 5, 6]`
Constructed list: `[0, 9, 36]`

Liste Tipleri (continued)

- Haskell'in Liste Anlamları
  - Orjinal

```
[n * n | n <- [1..10]]
```
- F#'in Liste Anlamları

```
let myArray = [|for i in 1 .. 5 -> [i * i) |]
```
- C# ve Java dilleri de listeleri destekler.Kendi dinamik koleksiyonlarında List ve ArrayList adında sınıfları vardır.

Birleşim Tipi

- Bir bileşim(union), değişkenlerinin(variable) yürütme süresi sırasında farklı zamanlarda farklı tipteki değerleri tutmasına izin verildiği tiptir
- Bileşimler(unions) için tasarim problemleri :
  1. Eğer varsa hangi tip tip kontrolü yapılmalıdır?
  2. Bileşimler(unions) kayıtlar(Records) ile entegre edilmeli midir?

Serbest Birleşimleri Ayırmak

Fortran, C ve C++ – serbest bileşimler(unions) (etiketler yoktur(tags))
- kayıtlarının(Records) kısımları yoktur
- Referanslarda tip kontrolü yoktur

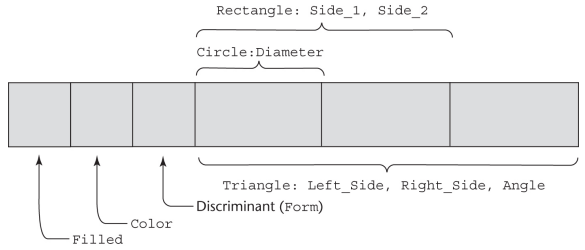
Java 'da kayıtlar(Records) da bileşimler(unions) de yoktur

- Değerlendirme – çoğu dilde güvensiz görünmektedir (Ada tek güvenilir dildir. Çünkü bileşim kontrolu yapar.)

Ada Birleşim Tipi

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
  Filled: Boolean;
  Color: Colors;
  case Form is
      when Circle => Diameter: Float;
      when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
      when Rectangle => Side1, Side2: Integer;
  end case;
end record;
```

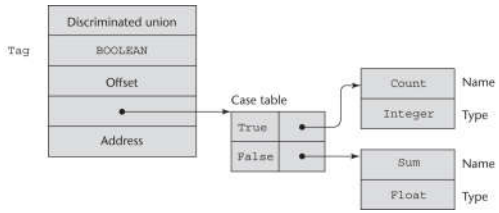

Ada Birleşim Tipi Örneği



Bir ayrılmış birleşimde 3 şeklin değeri gösterilmektedir.

Birleşimleri Uygulanması

```
type Node (Tag : Boolean) is
  record
    case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
    end case;
  end record;
```



Birleşimlerin Değerlendirilmesi

- Serbest birleşimler güvenilir değildir.
  - Tip kontrolune izin vermezler.
- Java ve C# birleşimleri desteklemez.
  - Programlama dilinin güvenilirliğini artırmak için
- Ada'nın ayrık birleşimleri güvenilirdir.

Pointer ve referans Tipleri

- Bir işaretçi(pointer) tipi değişkeni oluşturan bellek adres aralığını tutan özel bir değerdir.
- Dolaylı adresleme gücünü sağlar
- Dinamik hafıza kullanmayı sağlayan bir yoldur.
- Bir işaretçi dinamik depolama olarak oluşturulan bir konuma ulaşmak için kullanılabilir. (Genellikle dinamik bir öbek)

Pointerların Tasarım Problemleri

- Pointer'in kapsamı ve ömrü nedir ?
- Dinamik öbek değişkenlerinin ömrğ nedir?
- Işaretçiler bu konuda ortaya koyabildikleri değer türü sınırlı mı?
- Pointerlar dinamik depolama için mi kullanılıyorlar yoksa dolaylı adresleme için mi ya da her ikisi için mi kullanılıyor?
- Kullandığın dil pointer tipini mi destekliyor yoksa referans tipini mi yoksa her ikisini mi destekliyor?

Pointer Operatörleri

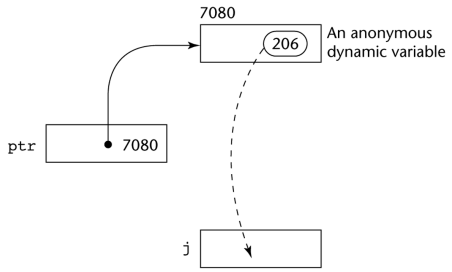
- İki temel işlemleri: atama ve başvurusu kaldırma.
- Atama bazı değişkenlerin değerini değiştirmekle birlikte bazı değişkenlerinde adres değerini değiştirir.
- Başvurusu işaretçi değeri ile temsil edilen bir konumda saklanan değeri verir.

  - C++ pointerlar için * operatorunu kullanır.

    ```
    j = *ptr
    ```

J değerini ptr'nin gosterdiği adres değerine atadı.

Pointer Assignment Illustrated



Atama işlemi j = *ptr

7080 adresindeki 206 değerini j değişkenine atadı

Pointerlarla İlgili Problemler

1. Dangling(askıdaki-boşta kalan) İşaretçiler(Pointers) (tehlikeli)
- Bir işaretçi(pointer) serbest bırakılmış(deallocate) bir heap-dinamik değişkene işaret eder
- Bir tane oluşturmak (harici(explicit deallocation)):

a. Bir heap-dinamik değişken ayırma ve pointerı bunu göstermeye ayarlama

b. Birinci pointerın değerine ikinci bir pointerı atama

c. Birinci pointerı kullanarak heap-dinamik değişkeni serbest bırakma(deallocate)

Pointerlarla İlgili Problemler(devamı)

2. Kayıp(Lost) Heap-Dinamik Değişkenler (savurgan)
- Bir program işaretçisi(program pointer) tarafından artık referans edilmeyen heap-dinamik değişken(variable)
- Bir tane oluşturmak:

a. Pointer p1 yeni oluşturulmuş bir heap-dinamik değişkeni göstermeye ayarlanır

b. p1 daha sonra diğer bir yeni oluşturulmuş heap-dinamik değişkeni göstermeye ayarlanır

- Heap-dinamik değişkenleri kaybetme işlemine memory leakage(bellek sızıntısı) denir

Ada'da Pointer Problemleri

- Boşta kalan pointerlara izin vermez çünkü dinamik nesneler otomatikman boşta kalan pointerları yok eder.
- Kayıp dinamik değişken problemi hala çözülememiştir. (Belki `UNCHECKED_DEALLOCATION`)

C ve C++'ta Pointerlar

- Dinamik Bellek Yönetimi ve adresleme için kullanılır
- Tanım Kümesi Tipi(Domain type) sabit olmak zorunda değildir (**`void *`**)
- **`void *`** – herhengi bir tipe işaret edebilir ve tip kontrolü yapılabilir (dereference yapılamaz)
- Bu dillerde pointer kullanımı esnektir.

C ve C++'ta Pointer Aritmetiği

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` eşittir. `stuff[5]` ve `p[5]`

`*(p+i)` eşittir `stuff[i]` ve `p[i]`

Referans Tipi

- C + + formal parametreleri için öncelikle kullanılan bir başvuru türü denilen işaretçi türü özel bir tür içerir.
- Avantajı: Hem referans değerini hem de veri değerini verebilir.
- Java C++'in referans değerini uzatarak pointerların sadece referans değeri tutmasını sağlar.
  - Referanslar yerine adresleri olmaktan çok, nesnelere başvurular vardır.
- C# hem Java'nın nesne modelini hemde C++'nın referans modelini kullanmaktadır.

Pointerların Değerlendirilmesi

1. Askıda İşaretçiler(Dangling Pointers) ve Askıda Nesneler (dangling objects) problemlerdir, heap yönetiminde olduğu gibi
2. İşaretçiler(Pointers) goto'lar gibidir– bir değişkenin(variable) erişebileceği hücreler(cells) aralığını(range) genişletirler
3. İşaretçiler(Pointers) veya referanslar dinamik veri yapıları için gereklidir—bu yüzden onlar olmadan bir dil tasarlayamayız

Pointerlar

- Büyük bilgisayarlar basit değerler kullanır.
- Intel mikroişlemciler(microprocessors) kesim(segment) ve ofset(göreli–konum)(offset) kullanır.

Askıda Pointer Problemi

1. Tombstone: heap-dinamik değişkene işaretçilik yapan ekstra bir heap hücresi(cell)
   - Gerçek işaretçi değişkeni(actual pointer variable) sadece tombstone'lara işaret eder
   - heap-dinamik değişken serbest bırakıldığı zaman (deallocated), tombstone kalır fakat nil'e ayarlanır

.*Kilit ve Anahtar*: Pointerlar anahtar ve adres olmak üzere iki tip değeri temsil eder.
   - Heap-dinamik değişkenleri tamsayı kilit değeri için değişken gözeler olarak temsil edilir.
   - Yığın-dinamik değişken ayrılan zaman, kilit değeri oluşturulur ve kilit hücre ve işaretçi anahtar hücresine yerleştirilir.

Yığın(Öbek) Yönetimi

- Çok karmaşık çalışma zamanı
- Tek boyutlu hücreler veya değişken boyutlu hücreler
- Çöp verileri kurtarmak için kullanılan yaklaşımlar
  - Referans sayaçları (*istekli yaklaşım*): Kurtarma işlemi kademi olarak yapılır
  - İşaretle ve Süpür (*uyuşuk yaklaşım*): değişken alan listesi boş olduğunda kurtarma başlar.

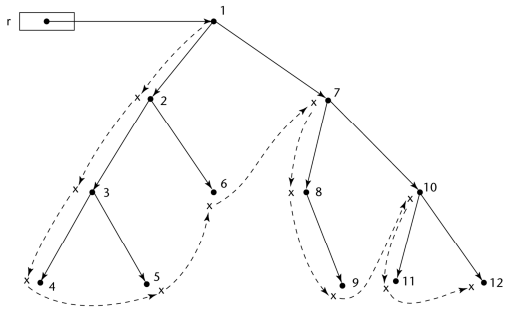
Reference Counter

Referans Sayaçlar(Reference counters): her bir hücrede o anda o hücreyi gösteren işaretçilerin sayısını tutan bir sayaç(counter) sürdürmek
- Dezavantajlar: boş alan gerekir, yürütme zamanı(execution time) gerekir, dairesel olarak bağlanmış hücreler için komplikasyonlar.
- *Avantaj*: Uygulama yürütmedeki önemli gecikmeler engellenir

İşaretle-Süpür

Eldeki bütün hücreler ayrılmış(allocated) olana kadar ayrılır(allocate) ve bağlantı kesilir (disconnect); sonra bütün atık (garbage) toplanmaya başlanır
- Her heap hücresinin(cell) collection algorithm tarafından kullanılan ekstra bir biti vardır
- Bütün hücreler başlangıçta atığa ayarlanır
- Bütün işaretçiler(Pointers) heap içine kopya edilir, ve erişilebilir hücreler atık-değil olarak işaretlenir
- Bütün atık hücreler eldeki hücreler listesine geri döndürülür
- Dezavantajlar: en çok ihtiyaç duyduğunuz zaman, en kötü çalışır (program heap deki hücrelerin çoğuna ihtiyaç duyduğunda en çok zamanı alır)

İşaretleme Algoritması



Dashed lines show the order of node_marking

Değişken Boyutlu Hücreler

- Her yönüyle tek boyutlu hücrelerden daha zordur.
- Çoğu programlama dilinde olması gerekir.
- Eğer işaretle-süpür algoritması kullanılıyorsa ek problemler meydana gelir. Bunlar;
  - Tüm hücre göstergelerinin başlangıç ayarı zordur.
  - İşaretlenen işlem ziyaret edilmemişse problem büyür.
  - Kullanılabilir alan listesini bakımı yükü artar.

Tip Kontrolü

- Altprogramlar ve atamaları içerecek şekilde işlenen ve operatörler kavramını yaygınlaştırmak.
- *Tip denetleme bir operatorun işlenen tipin uyumunu sağlama faaliyetidir.*
- Uyumlu tipin üretiminde ve denetiminde derleyicinin ürettiği kod ve kurallar göz önüne alınır.
- Bu otomatik dönüşüm bir zorlama denir.Örn: double a=15; int k=a;
- Bir tip hatası desteklenmeyen tarzdaki bir tipin o veri tipi kümesinde yorumlanmaya çalışılmasıdır

Tip Kontrolü (devamı)

- Eğer tüm tip bağlayıcıları statikse, tip denetimi de statik olarak yapılır.
- Eğer tip bağlayıcıları dinamikse tip kontrolünün dinamik yapılması zorunludur.
- Tip hataları her zaman tespit edilirse programlama dilindeki tipler güçlüdür.
- Güçlü tiplerin avantajları: Hata algılama değişkenleri tip hatası sonucunu kolayca verir.

Güçlü Tipler

Dil Örnekleri:

- C ve C++ :parametre tür denetlemesi önlenebilir; birleşimler kontrol türü değildir.
- Ada'da (`UNCHECKED CONVERSION` kodu bir gözetleme deliğidir.)

  (Java and C# Ada'ya benzer)

Güçlü Tipler (devamı)

- Zorlama kuralları güçlü tiplerde güçlü efektler oluşturabilir veya onları zayıflatabilir. (C++ ve Ada)
- Java'da sadece C++'ın yarım atama kuralları olmasına rağmen,güçlü tiplerde Ada'dan daha az etkindir.

Tip Adı Eşitleme

- Tip Adı Eşitleme'nin manası ya aynı tipi yada aynı tip adını kullanarak değişkenleri birbirine eşitlemektir. Bu metot kullanıldığında iki değişkende eşdeğer tip var demektir.
- Uygulamada basit fakat hayli kısıtlayıcıdır:
  - Alt aralıklarda tanımlanan integer tipler örn. Notlar (1,100) integer tipine eşit değildir.
  - Örgün parametreleri, karşılık gelen gerçek parametreleri aynı türde olmalıdır.

Yapı Tipi Eşitleme

- Yapı tipi eşitleme, aynı yapıları varsa iki değişken eşdeğer tip olması anlamına gelir.
- Çok esnek fakat uygulaması çok zor.

Tip Eşitleme (devamı)

- Iki yapısal tip sorununu ele alalım:
  - Yapısal olarak aynı ama farklı alan adları kullanırsanız iki kayıt türünü eşitler misiniz?
  - İki dizi türünde, simgeler farklı olması dışında aynı ise bu diziler eşdeğer midir? (e.g. `[1..10]` and `[0..9]`)
  - İki sıralama tipi onların bileşenleri farklı yazıldığından eşdeğer olur mu?

Veri Tipleri Teorisi

- Tip teorisi matematik, mantık, bilgisayar bilimleri ve felsefe çalışmasını kapsayan geniş bir disiplinler arası alandır.
- Bilgisayar bilimlerinde tip teorisi iki ana dala ayrılmıştır.
  - Pratik – Ticari dillerdeki veri türleri
  - Soyut – İleri matematiksel hesaplamalar için kullanılır.
- Bir tip sistemi tipleri ayarlayan ve kuralları yöneten programları kullanır.

Veri Tipleri Teorisi (devamı)

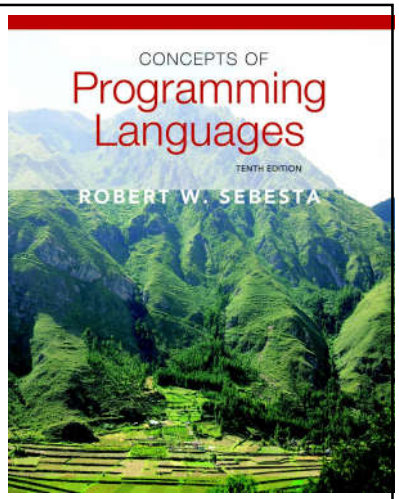
- Bir tip sisteminde biçimsel model tipleri kümesi ve tip kuralları tanımlayabilirsiniz.
  - Tipleri belirlemek için dilbilgisi kuralları veya haritalar kullanılır.
  - Sonlu haritalama – model diziler ve fonksiyonlar
  - Kartezyan ürünler – model demetler ve kayıtlar
  - Birleşimleri ayarlama – Model birleşim tipleri
  - Altayarlar – model alttipler

Summary

- Veri tipleri, bir dilin kullanışlılığını belirleyen en büyük parçasıdır.
- Çoğu dilde zorunlu olarak yer alan ilkel veri türleri sayısal, karakter, ve Boolean türlerini içerir.
- Kullanıcı tanımlı numaralandırma ve alt aralık tipleri, programların okunabilirliği ve güvenilirliğini arttırır.
- Diziler ve kayıtlar birçok dilde bulunur.
- Pointerlar esnek adresleme ve dinamik bellek kontrolü yönetiminde kullanılan veri tipleridir.

Bölüm 7

İfadeler ve
Atama İfadeleri



Bölüm 7 Konular

- Giriş
- Aritmetik İfadeler
- Operatörlerin Aşırı Yüklenmesi
- Tip Dönüşümleri
- İlişkisel ve Mantıksal İfadeler
- Kısa Devre Tespiti
- Atama İfadeleri
- Karışık-Biçim Atamaları

Giriş

- İfadeler bir programlama dilinde hesaplamaları belirtmede temel araçtır.
- İfadelerin değerlendirmesini anlamak için, operatörlerin sırası ve işlenenlerin(operant) değerlendirmesine aşina olmamız gerekir.
- Emirsel dillerin temeli atama ifadeleridir.

Aritmetik İfadeler

- Aritmetik ölçüm ilk programlama dilinin gelişiminde kullanılan motivasyonlarından biri olmuştur.
- Aritmetik ifadeler; operatörler,operantlar,parantezler ve fonksiyon çağrılarından oluşur

Aritmetik İfadeler: Tasarım Sorunları

- Aritmetik İfadeler için Tasarım Sorunları
  - Operatörlerin öncelik kuralları?
  - Operatörlerin birleşilirlik kuralları?
  - Operantların sırasının değerlendirilmesi?
  - Operant değerlendirmenin yan etkileri?
  - Operatörlere aşırı yükleme?
  - İfadelerdeki tip karıştırılması?

Aritmetik İfadeler: Operatörler

- Unary(tekli) operatorün tek operantı vardır.
- Binary(ikili) operatorün iki operantı vardır.
- Ternary(üçlü) operatorün iki operantı vardır.
- N-ary(nli) operatörün n tane operantı vardır.

Aritmetik İfadeler: Operatör Öncelik Kuralları

- Operatör öncelik kuralları farklı öncelik seviyesindeki bitişik operatörlerdeki operatörlerin işlenme sırasını belirtir.
- Klasik öncelik seviyeleri
  - Parantezler
  - Tekli operatörler
  - ** (Eğer dil destekliyorsa, üs alma)
  - *, /(çarpma,bölme)
  - +, -(toplama,çıkarma)

Aritmetik İfadeler: Operatör Birleşilirlik Kuralı

- Bu kural aynı öncelik seviyesindeki bitişik operatörlerin işlenmesi sırasını belirtir.
- Tipik birleşilirlik kuralları
  - Soldan sağa, **(hariç, burada sağdan sola),
  - Zaman zaman tekli operatörlerin birleşilirliği sağdan sola olabilir (ör., FORTRAN)
- APL dili farklıdır; bu dilde tüm operatörler eşit önceliklere sahiptir ve operatörlerin birleşilirliği sağdan soladır.
- Öncelik ve birleşilirlik kuralları parantezlerle geçersiz kılınabilir.

Ruby ve Scheme'de İfadeler

- Ruby
  - Tüm aritmetik, ilişkisel, atama operatörleri, ve hatta dizi indeksleme,kaydırma ve bit şeklindeki mantık operatörleri metotlar olarak sağlanır
  - Bunun sonuçlarından biri bu operatörlerin uygulama programları tarafından geçersiz kılınabilmesidir.
- Scheme (ve Common LISP)
  - Tüm aritmetik ve mantık işlemleri belirgin bir şekilde alt programlar tarafından çağrılır.
  - `a + b * c` ifadesi `(+ a (* b c))` olarak kodlanır.

Aritmetik İfadeler: Şartlı İfadeler

- Şartlı İfadeler
  - C-tabanlı diller (ör., C, C++)
  - Bir örnek:

```
average = (count == 0)? 0 : sum / count
```

  - Aşağıdakine eş değerdir:

```
if (count == 0)
   average = 0
else
   average = sum /count
```


Aritmetik İfadeler: operant Değerlendirme Sırası

- *operant değerlendirme sırası*
  1. Değişkenler: Bellekten değerini al
  2. Sabitler: bazen bellekten alınır bazen makine dili komutundadır.
  3. Parantezli ifadeler: İçindeki tüm operant ve operatörler ilk olarak işlenir
  4. En ilginç durum bir operantın bir fonksiyon çağrısı yapması durumudur.(İşlenme sırası çok önemli)(yan etkilerinden dolayı önemli)

Aritmetik İfadeler: Fonksiyonel Yan Etkiler

- Fonksiyonel Yan Etkiler: Bir fonksiyon iki yönlü bir parametreyi veya lokal olmayan bir değişkeni değiştirdiğinde meydana gelir.
- Örnek:
  - Bir ifadede çağrılmış bir fonksiyon ifadenin başka bir operantını değiştirdiğinde ortaya çıkar; bir parametre değişim örneği:
  -

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


Fonksiyonel Yan Etkiler

- Bu problem için 2 muhtemel çözüm:
  1. Fonksiyonel yan etkileri iptal etmek için dil tanımlaması yapılır
     - Fonksiyonlarda 2 yönlü parametre olmayacak
     - Fonksiyonlarda global değişken olmayacak
     - **Avantajı:** o çalışıyor!
     - **Dezavantajı:** tek yönlü parametrelerin kararlılığı ve global değişkenlerin olmayışı(fonksiyonların birden çok değer döndürmeleri ihtiyacından dolayı pratik değil)
  2. operantların işlem sırasını belirlemek için dil tanımlaması yapılır
     - **Dezavantajı** : Bazı derleyicilerin optimizasyonunu sınırlar
     - Java operantların soldan sağa işlenmesine izin verdiğinden bu problem oluşmaz.

İmalı Şeffaflık

- Eğer bir programdaki aynı değere sahip herhangi iki ifade programın akışını etkilemeksizin birbiri yerine kullanılabilirse bu program imalı şeffaflık özelliğine sahiptir.

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

Eğer `fun` fonksiyonu yan etkiye sahip değilse,
`result1 = result2` olacaktır.
Aksi taktirde,olmayacak , ve imalı şeffaflık bozulur.

İmalı Şeffaflık(devam)

- İmalı Şeffaflığın Avantajı
  - Eğer bir program imalı şeffaflığa sahipse programın anlamı daha kolay anlaşılır
- Onlar değişkenlere sahip olmadığı için teorik fonksiyonel dillerdeki programlar imalı şeffaftırlar.
  - Fonksiyonlar yerel değişkenler içinde saklanacak durumlara sahip olamazlar.
  - Eğer bir fonksiyon yabancı bir değer kullanırsa, o bir sabit olmalıdır(değişkenler değil).Bu yüzden bir fonksiyonun değeri sadece onun parametrelerine bağlıdır.

Aşırı Yüklenmiş Operatörler

- Bir operatörün birden fazla amaç için kullanımı *operatörün aşırı yüklenmesi* olarak adlandırılır.
- Yaygın olanlardan bazısı(örn., int ve float için '+',string ifadelerin birleştirilmesi)
- Bazısı potansiyel olarak sorunludur(örn.,C ve C++ da '*' ikili olarak çarpı,tekli olarak adresleme(pointer))
  - Derleyicinin hata belirlemesindeki kayıplar(derleyici operant eksiklerini fark etmeli)(a*b yerine *b yazmak gibi)
  - Bazı okunabilirlik kayıpları(okunabilirliği zorlaştırır)

Aşırı Yüklenmiş Operatörler(devam)

- C++, C#, ve F# kullanıcı tarafından tanımlanan operatörlere izin verir.
  - Böyle operatörler anlamlı kullanıldığında okunabilirliğe bir yardımı olabilir ,(metot çağrılarından kaçınmak, ifadeler doğal görünür)
  - Potansiyel problemler:
    - Kullanıcılar anlamsız işlemler tanımlayabilir
    - Operatörler anlamlı bile olsa okunabilirlik zarar görebilir

Tip Dönüşümleri

- **Daraltıcı dönüşüm**: Dönüştürülecek tip orijinal tipin tüm değerlerini içermiyorsa bu dönüşüme daraltıcı dönüşüm denir (örnek, `float` → `int`)
- **Genişletici dönüşüm**: Dönüştürülecek tip orijinal tipin tüm değerlerinden fazlasını içeriyorsa bu dönüşüm genişletici dönüşümdür.(örnek,`int` → `float`)

Tip Dönüşümleri: Karışık Biçim

- Eğer bir işlemin operantları farklı türden ise bu ifadeye karışık biçimli ifade denir.
- Zorlama(istemsiz) kapalı tip dönüşümdür.
- Zorlamanın Dezavantajları:
  - Derleyicinin hata bulma kabiliyetini azaltır.
- Çoğu dillerde, tüm sayısal tipler genişletici dönüşüm kullanılarak zorlanır.
- Ada'da, ifadelerde zorlama yoktur.
- ML ve F#' ta, ifadelerde zorlama yoktur.

Açık Tip Dönüşümleri

- C tabanlı dillerde *veri tipleri dönüşümü*(*casting)*
- Örnekler
  - `C: (int)angle`
  - `F#: float(sum)`

Not: F#'ın sentaksı fonksiyon çağırmaya benzer.

İfadelerdeki Hatalar

- Sebepler
  - Aritmetiğin doğal sınırları örn: sıfıra bölme
  - Bilgisayar aritmetik sınırları örn: taşma(overflow)
- Çalışma zamanında sık sık ihmal edilir.

İlişkisel ve Mantıksal İfadeler

- İlişkisel İfadeler
  - İlişkisel operatörler ve çeşitli tipteki operantların kullanımı
  - Bazı mantıksal işaretlerin ölçümü
  - Operatör sembolleri dillere göre değişiklik gösterir. (`!=`, `/=`, `~=`, `.NE.`, `<>`, `#`)
- JavaScript ve PHP 2 ek İlişkisel operatöre sahiptir, `===` and `!==`
  - Operantlarını zorlamamaları dışında kuzenlerine benzer, `==` ve `!=`,
  - Ruby eşitlik ilişki operatörü için `==` kullanır

İlişkisel ve Mantıksal İfadeler

- Mantıksal İfadeler
  - Hem operantlar hem de sonuçlar mantıksaldır
  - Örnek operatörler:
- C89 mantıksal tipe sahip değil ve bunun için int tipini kullanır.(0➔yanlış,değilse doğru)
- C ifadelerinin tuhaf bir özelliği:
- `a < b < c` doğru bir ifade, ama sonuç umduğumuz şeyi vermeyebilir:
  - Soldaki operatörler işlendiğinde,0 veya 1 üretir
  - Ölçülen sonuç o zaman 3. operant ile karşılaştırılır (ör: , `c`)

Kısa Devre Tespiti

- Bir ifadede operant/operatörlerin tüm hesaplamalarını yapmaksızın sonucun bulunmasıdır.
- Örnek: `(13 * a) * (b / 13 - 1)`
  Eğer `a==0 ise` , diğer kısmı hesaplamaya gerek yok `(b /13 - 1)`
- Kısa Devre Olmayan Problem

```
index = 0;
while (index <= length) && (LIST[index] != value)
      index++;
```

  - `index=length` **olduğunda**, `LIST[index]` **indeksleme problemi ortaya çıkaracak(**`LIST` dizisi `length - 1` uzunluğunda varsayılmış)

Kısa Devre Tespiti(devam)

- C, C++, ve Java: kısa devre tespitinin bütün mantıksal operatörler(`&&` ve `||`) için yapar, ama bit düzeyinde mantıksal operatörler(`&` and `|`) için yapmaz.
- Ruby, Perl, ML, F#, ve Python'da tüm mantık operatörleri için kısa devre tespiti yapılır.
- Ada: Programcının isteğine bağlıdır(kısa devre '**`and then`**' ve '**`or else`**' `ile belirtilir`)
- Kısa devre tespiti ifadelerdeki potansiyel yan etki problemini ortaya çıkarabilir
  örnek. `(a > b) || (b++ / 3)`
- `a>b olduğu sürece b artmayacak`

Atama İfadeleri

- Genel Sentaks
  `<target_var> <assign_operator> <expression>`
- Atama operatörü
  - `=` Fortran, BASIC, C-tabanlı diller
  - `:=` Ada
- `=` eşitlik için ilişkisel operatörler aşırı yüklendiğinde kötü olabilir(o zaman C-tabanlı diller ilişkisel operatör olarak neden '==' kullanır?)

Atama İfadeleri: Şartlı Amaçlar

- Şartlı Amaçlar(Perl)
  `($flag ? $total : $subtotal) = 0`

  Hangisi eşittir

```
if ($flag){
  $total = 0
} else {
  $subtotal = 0
}
```


Atama İfadeleri : Birleşik Atama Operatörleri

- Atama formu için yaygın olarak bir stenografi metodu belirtmek gerekir
- ALGOL'de tanımlı; C ve C-tabanlı diller de benimsemiş.
  - Örnek:

    `a = a + b`

    Aşağıdaki gibi yazılabilir.

    `a += b`

Atama İfadeleri : Tekli Atama Operatörleri

- C-tabanlı dillerdeki tekli atama operatörleri atama ile artış ve azalış işlemlerini birleştirir
- Örnekler
  - `sum = ++count` (`count` arttırıldı, daha sonra `sum`'a **`aktarıldı`**)
  - `sum = count++` (`count` `sum`'a **`aktarıldı`**, ondan sonra arttırıldı
  - `count++` (`count` arttırıldı)
  - `-count++` (`count` arttırıldı ondan sonra negatifi alındı)

Bir İfade olarak Atama

- C-tabanlı diller, Perl, ve JavaScript'te atama durumu bir sonuç üretir ve bir operant olarak kullanılabilir,

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

  `ch = getchar()` başarılı; sonuç(`ch a aktar`) while döngüsü için şartsal bir değer olarak kullanılır
- Dezavantaj: Başka tip yan etkiler.

Çoklu Atamalar

- Perl, Ruby, ve Lua çok hedefli ve çok kaynaklı atamalara izin verir

```
($first, $second, $third) = (20, 30, 40);
```

Hatta,aşağıdaki geçerli ve bir yer değiştirme uygulanır:

```
($first, $second) = ($second, $first);
```


Fonksiyonel Dillerde Atama

- Fonksiyonel dillerde tanıtıcılar(identifier) sadece değer adlarıdır.
- ML
  - İsimler **`val`** ve değer ile sınırldır İsimler

    `val` `fruit = apples + oranges;`
  - Eğer gruit için başka bir val izlenecekse, o yeni ve farklı bir isimde olmalıdır.
- F#
  - F#'s yeni bir skop(scope) yaratmanın dışında ML 'in **`val`** ile aynıdır.

Karışık Biçim Ataması

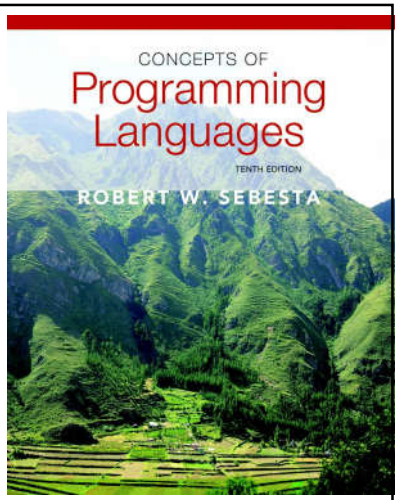
- Atama ifadeleri karışık biçimde olabilir
- Fortran, C, Perl, ve C++'ta ,her tip sayısal değer her tip sayısal değişkene atanabilir
- Java ve C#'ta, sadece genişletici atama zorlaması yapılır
- Ada'da, atama zorlaması yoktur.

Özet

- İfadeler
- Operatör önceliği ve birleşilirliği
- Operatörlerin aşırı yüklenmesi
- Karışık tipli ifadeler
- Atamaların çeşitli formları

Bölüm 8

Komut Seviyeli
Kontrol Yapıları



Bölüm 8'in Başlıkları

- Giriş
- Seçme Komutları
- Döngü (iteratif) Komutlar
- Koşulsuz Atlama Komutları
- Korunan Komutlar
- Sonuçlar

Akış Kontrolünün Seviyeleri

- İfadeler içinde(Bölüm 7)
- Program Birimleri Arasında(Bölüm 9)
- Program Komutları Arasında(Bu Bölümde)

Kontrol Komutları: 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)

Kontrol Yapısı

- Bir kontrol yapısı (control structure) bir kontrol komutu ve onunkontrolündeki komutlardan oluşur.
- Tasarım Sorunu
  - Kontrol yapısının birden çok girişi var mıdır ?

Seçme Komutları

- Bir seçme komutu (selection statement) yürümekte olan programda iki veya daha fazla yoldan birini seçmemizi sağlar.

İki sınıfa ayrılır:

- İki yollu seçiciler
- Çok yollu seçiciler

İki Yollu Seçme Komutları

Genel şekli:

if <kontrol ifadesi>

then <ifade> else <ifade>

Tasarımla ilgili hususlar:

- Kontrol ifadesinin şekli ve tipi ne olacak?
- **"then" ve "else" terimleri nasıl belirlenecek?**
- İç içe geçmiş seçicilerin anlamları nasıl belirlenecek?

Kontrol İfadeleri

- Eğer ayrılmış sözcükler (reserved word) yada diğer sentatik işaretleyiciler kullanılmamışsa, kontrol ifadeleri parantez içerisinde belirtilir.Örn: if(stmnts)
- C89, C99, Python, ve C++, kontol ifadeleri aritmetik ifadelerden oluşabilir.
- Diğer dillerin çoğunda kontrol ifadeleri Boolean veri tipi olmalıdır.

Cümle Formu

- Çoğu çağdaş dilde, then ve else bloklarının içerisinde basit veya birleşik komutlar kullanılabilir.
- Perl'de tüm cümleler ayraçlarla sınırlandırılmış olmalıdır. (Ayraçların içerisindeki kodlar birleşik olmalıdır.)
- Fortran 95, Ada, Python, ve Ruby'de cümleler kod dizilerinden oluşurlar.
- Python cümleleri tanımak için çentik(“”) kullanır

```
if x > y :
  x = y
  print " x was greater than y"
```


Yuvalama Seçiciler

- Java örneği

```
if (sum == 0)
  if (count == 0)
      result = 0;
else result = 1;
```

- **Else hangi if'e ait?**
- Java'nın statik semantik kuralı: Else kendinden bir önceki (en yakın) if'e aittir.

Yuvalama Seçiciler (devamı)

- Alternatif bir semantik elde edilmek istenirse, birleşik komutlar (iç içe komutlar) kullanılabilir.

```
if (sum == 0) {
 if (count == 0)
     result = 0;
}
else result = 1;
```

- C, C++, ve C#'da üstteki çözüm kullanılır.

Yuvalama Seçiciler (devamı)

- Ruby'deki komut dizileri ise aşağıdaki gibidir.

```
if sum == 0 then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Yuvalama Seçiciler (devamı)

- Python

```
if sum == 0 :
  if count == 0 :
    result = 0
  else :
    result = 1
```


Seçme İfadeleri

- ML, F#, ve LISP seçici bir ifadedir.
- F#

```
let y =
  if x > 0 then x
  else 2 * x
```

  - Eğer if ifadesi bir değer döndürüyorsa else ifadesi de olmalıdır.(ifade değerdense bir çıktı üretmelidir.)

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: Örnekler

- C, C++, Java ve JavaScript

```
switch (expression) {
  case const_expr_1: stmt_1;
  ...
  case const_expr_n: stmt_n;
  [default: stmt_n+1]
}
```

Çoklu Seçme Komutları: Örnekler

- Tasarım yaparken C'nin switch kodu seçilirse
  1. Kontrol ifadeleri yalnızca tamsayı olabilir.
  2. Seçilebilir segmentler komut dizileri,bloklar veya bileşik komutlar olabilir.
  3. Herhangi bir segment numarası çalıştırılabilir bir yapı olabilir.
  4. `Default` cümlesi tanımlanmayan değerler için kullanılır.

Çoklu Seçme Komutları: Örnekler

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

```
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: Örnekler

- Ruby'nin iki farklı case komutu vardır–Yalnız birinden bahsedeeğiz.

```
leap = case
    when year % 400 == 0 then true
    when year % 100 == 0 then false
    else year % 4 == 0
    end
```


Çoklu Seçicilerin Uygulanması

- Yaklaşımlar:
  - Çok koşullu dallanmalar
  - Bir tabloda durum değerleri(case değerleri) saklanır ve doğrusal arama kullanılarak değerler getirilir.
  - Eğer birden fazla case varsa Hash tablosunun case değerleri kullanılabilir.
  - If the number of cases is small and more than half of the whole range of case values are represented, an array whose indices are the case values and whose values are the case labels can be used

Çoklu Seçmede IF Kullanımı

- Çoklu seçicilerde if kullanımı aşağıdaki Python örneğindeki gibi if-else if yapısı kullanılarak yapılır.

```
if count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```


Çoklu Seçmede IF Kullanımı

- Python örneğini Ruby ile geröekleştirmek.

```
case
  when count < 10 then bag1 = true
  when count < 100 then bag2 = true
  when count < 1000 then bag3 = true
end
```


Çoklu Seçicilerin Şeması

- Genel çağırma formu ŞART:

```
(ŞART
  (YÜKLEM1 İFADE1)
  …
  (YÜKLEMn İFADEn)
  [(ELSE expressionn+1)]
)
```

  - Else deyimi isteğe bağlıdır; yukarıda kullanılan else deyimi true ile eşanlamlıdır.
  - Her yüklem-ifade çifti parametredir.
  - Anlambilim:Şart ifadesinin değeri ilk yüklem ifadesinin değeriyle ilgiliyse şart doğrudur.

Döngülü Komutlar (Iterative Statements)

- Bir komutun veya bir komut grubunun tekrarlanan yürütülmesi döngü veya özyineleme ile elde edilir.
- 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? Başlangıçta mı, sonda mı, programcı mı karar verecek?

Sayaç Kontrollü Döngüler

- Sayaç kontrollü döngülerde, başlangıç değeri ,bitiş değeri ve artış miktarı belirtilerek adım kontrolü yapılır.
- Tasarım Sorunları:

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?

Sayaç Kontrollü Döngüler: Örnekler

- Ada

```
for var in [reverse] discrete_range loop
...
end loop
```

- Tasarım Seçenekleri:
  - Döngü değişkeninin veri tipi hangi aralıkta.
  - Döngü değişkenin döngü dışına çıkmaması
  - Döngü değişkeni döngü içinde değiştirilemez (Örn: foreach döngüleri) fakat aralığı değiştirilebilir, aralığın değişmesi döngü kontrolüne etki etmez.
  - Döngü aralığı yalnızca bir defalığına değerlendirilebilir.
  - Döngü içerisinde dallanma komutları kullanılamaz.

Sayaç Kontrollü Döngüler: Örnekler

- C-tabanlı diller

```
for ([expr_1] ; [expr_2] ; [expr_3]) statement
```

  - The expressions can be whole statements, or even statement sequences, with the statements separated by commas Yukarıda C tabanlı dillerde for yapısını vermiştir. İfadelerin arasına virgül konarak ifade sayısı artırılabilir. Bloklar noktalı virgülle ayrılmıştır.
    - Birden çok komutlu ifadelerin değeri, son durumdaki ifadenin değerine eşittir.
    - İkinci ifadede herhangi bir değer yoksa döngü sonsuza dek döner.
- Tasarım Seçenekleri:
  - Belirgin bir döngü değişkeni yoktur.
  - Döngü içerisinde her şey değişebilir.
  - İlk ifade yalnızca bir kez değerlendirilirken şart ifadesi adım sayısı kadar değerlendirilir.
  - C Döngü içerisinde dallanma komutlarına izin verir (goto gibi)

Sayaç Kontrollü Döngüler: Örnekler

- C++ iki şekilde C ile farklılık gösterir :
  1. Kontrol ifadesi Boolean olabilir.
  2. Başlangıç ifadesi değişken tanımları içerebilir.
- Java ve C#
  - C++'tan farkı,kontrol ifadesinin Boolean olma zorunluluğudur.

Sayaç Kontrollü Döngüler: Örnekler

- Python

  for döngü değişkeni **in** nesne:
  - döngü gövdesi

  [**else**:
  - else cümlesi]

  - Nesne sıklıkla bir aralığı(range) temsil eder.Liste değerleri ise parantez içersinde (`[2, 4, 6]`), yada **range** fonksiyonu kullanılarak ifade edilir.(**range**`(5), 0, 1, 2, 3, 4` değerlerin döndürür.
  - The loop variable takes on the values specified in the given range, one for each iteration Döngü değişkeni aralıkta(range) gösterilen değerleri alırlar.
  - Döngü normal bir şekilde sona ererse isteğe bağlı olarak else bloğu yürütülür.

Sayaç Kontrollü Döngüler: Örnekler

- F#
  - Sayaçları değişkenler gerektiren ve fonksiyonel dillerde değişkenleri yok olduğundan, sayaç kontrollü döngü özyinelemeli fonksiyonlar ile simüle edilmelidir.

```
let rec forLoop loopBody reps =
  if reps <= 0 then ()
  else
    loopBody()
    forLoop loopBody, (reps - 1)
```

  forloop adında loopbody parametrelerini kullanan recursive bir fonsiyon tanımlanır.
  - `()` Hiçbir olay gerçekleşmiyor ve hiçbir değer dönmüyor anlamına gelmektedir.

Mantıksal-Kontrollü Döngüler

- Tekrarlamalar Boolean tabanlı ifadelerle kontrol edilir.
- Tasarım Sorunları:
  - Öntest mi yoksa sontest mi?
  - Sayaç kontrollü döngülerin özel bir halimi olacak yoksa farklı spesifik bir döngü yapısı mı?

Mantıksal-Kontrollü Döngüler: Örnekler

- C ve C++ dillerinde hem öntest hemde sontest yapısını kullanan döngü ifadeleri vardır.

```
while (control_expr)        do
  loop body                   loop body
                            while (control_expr)
```

  - C ve C++'ta parantezin içerisine mantıksal kontrol ifadesi yazılır.
- Java kontrol ifadesinin Boolean olma zorunluluğu dışında C ve C++'a benzer.(Ve döngü yapısına sadece başlangıçta girilir.- Java'da `goto` deyimi yoktur.C# ta ise döngü yapısına ortadan da girilebilir.)

Mantıksal-Kontrollü Döngüler: Örnekler

- F#
  - Sayaç kontrollü döngülerde olduğu gibi mantıksal kontrollü döngülerde de recursive fonksiyonlar kullanılır.

```
let rec whileLoop test body =
   if test() then
     body()
     whileLoop test body
   else ()
```

  - Whileloop özyinelemeli fonksiyonu test ve body parametreleriyle tanımlanır. Test parametresi mantıksal kontrolu yapar body ise kontrolün doğru olduğu durumdaki yapılacak işlemi temsil eder.

Kullanıcı Tarafından Yerleştirilen döngü Kontrol Düzenekleri

- Programcılar döngüyü farklı bir şekilde kontrol etmek için komutları döngünün farklı bölgelerine yerleştirebilirler.
- Döngüler için basit tasarımlar yapılabilir.(örn, `break`)
- Tasarım Problemleri:

  1. Koşul ve döngüden çıkış tek bir kısım mı olmalı?

  2. Kontrol birden çok döngüden dışarı çıkabilmeli mi?

Kullanıcı Tarafından Yerleştirilen döngü Kontrol Düzenekleri

- C , C++, Java, Perl ve C#'ta 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:

dongu1:

```
for(satir = 0; satir<satirsayi; satir++)
for(sutun = 0; sutun<sutunsayi; sutun++) {
toplam += mat[satir][sutun];
if(toplam > 1000.0) break dongu1;
}
```

C örneği

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

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)){ ... }**

```
for (p=root; p==NULL; traverse(p)){
  ...
}
```


Veri Yapılarına Dayalı Döngüler (devamı)

- PHP
  - `current` pointer'ın o andaki işlediği dizi elemanını temsil eder.
  - `next current` değerini bir sonraki elemana taşır.
  - `reset current` değerini dizinin ilk elemanına taşır.
- Java 5.0 (Foreach gibi davranan For kullanımı)

  Diziler ve diğer sınıflarda kullanılan döngü arayüzleri örn., `ArrayList`

```
for (String myElement : myList) { … }
```


Veri Yapılarına Dayalı Döngüler (devamı)

- C# ve F# (ve diğer .NET dilleri)'ta Java 5.0'a benzeyen kapsamlı kütüphane sınıfları vardır. (diziler,listeler,yığınlar ve kuyruklar).Bu yapılarda bulunan elemanların tümünü **`foreach`** döngüsüyle gezebiliriz.

```
List<String> names = new List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Ted");
foreach (Strings name in names)
    Console.WriteLine ("Name: {0}", name);
```


Veri Yapılarına Dayalı Döngüler (devamı)

- Ruby blokları kod dizileridir ve bloklar **`do end`** kodları arasında tanımlanmıştır
  - Bloklar döngü oluşturabilmek için metotlarla beraber kullanılabilirler.
  - Önceden tanımlanmış döngü metotları (`times`, `each`, `upto`):

```
3.times {puts "Hey!"}
list.each {|value| puts value}
```

(`list` bir dizi; `value` ise bir blok parametresi)

```
1.upto(5) {|x| print x, " "}
```

  - Ruby bir for komutuna sahiptir fakat for komutunu çalıştırabilmek için upto metoduna dönüştürmesi gerekmektedir.

Veri Yapılarına Dayalı Döngüler (devamı)

- Ada
  - Ada dilinde döngü aralığı ile dizi indisi arasında ilişki kurulabilir.

```
subtype MyRange is Integer range 0.99;
MyArray: array (MyRange) of Integer;
for Index in MyRange loop
 ...MyArray(Index) ...
end loop;
```


Koşulsuz Dallanma

- Programın akışı değiştirilebilir, her türlü kontrol komutu "goto" ve seçici ile yapılabilir. Çok etkili bir komut.
- 60 ve 70'li yılların en ateşli tartışma konusu olan goto komutu bazı programcılara göre kaos komutudur. Çünkü programı rastgele dallandırdığı için olası hatalara sebep olabiliyor.
- Temel sorun: Okunabilirlik
- Bazı dillerde goto komutu kullanılamaz. (Java)
- C#'ta `goto` komutu kullanılabilir. (switch bloguyla beraber)
- Döngü çıkış komutları (**break, last) kamufle edilmiş goto'lardır.**

Güvenlikli Komutları

- Dijkstra tarafından tasarlanmıştır.
- Yeni programlama metodolojilerini geliştirme esnasında desteklemek ve onlara kaynak sunmak.
- Eşzamanlı programlama için iki dilsel mekanizmayı temel alır. (CSP ve Ada)
- Temel Fikir: Değerlendirme sırası önemli değilse, programı tek belirtmeniz gerekir

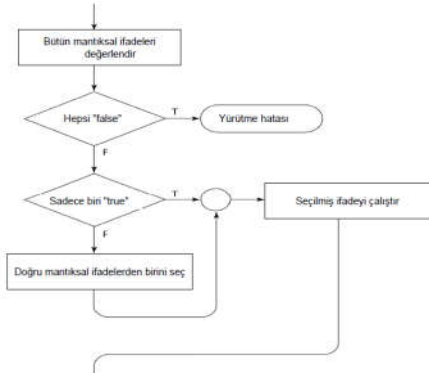
Güvenlikli Seçme Komutları

- Form

```
if <Boolean expr> -> <statement>
[] <Boolean expr> -> <statement>
 ...
[] <Boolean expr> -> <statement>
fi
```

- Anlambilim: yapıya ulaşıldığında
  - Tüm boolean ifadeleri değerlendirilir.
  - Eğer birden fazla doğru ifade varsa non-deterministik bir algoritma seçilmelidir.
  - Eğer doğru ifade yoksa çalışma zamanı hatası verilir.

Güvenlikli Seçme Komutları



Güvenlikli Döngü Komutları

- **Formu**

```
do <Boolean> -> <statement>
[] <Boolean> -> <statement>
...
[] <Boolean> -> <statement>
od
```

- Anlambilim: Her bir adım için
  - Tüm Boolean ifadeleri değerlendirilir.
  - Eğer birden fazla doğru varsa seçme komutlarındaki gibi non-deterministik bir seçim yapılır ve döngünün başına geri dönülür.
  - Eğer hepsi yanlışsa döngüden çıkılır.

Güvenlikli Komutlar: Gerekçe

- Kontrol deyimleri ve program doğrulama arasında güçlü bir bağlantı vardır.
- Goto komutlarının doğrulanması imkansızdır.
- Doğrulama seçim ve mantıksal öntest döngüleri için mümkündür.
- Güvenlikli kontrollerin doğrulanması daha basittir.

Sonuçlar

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