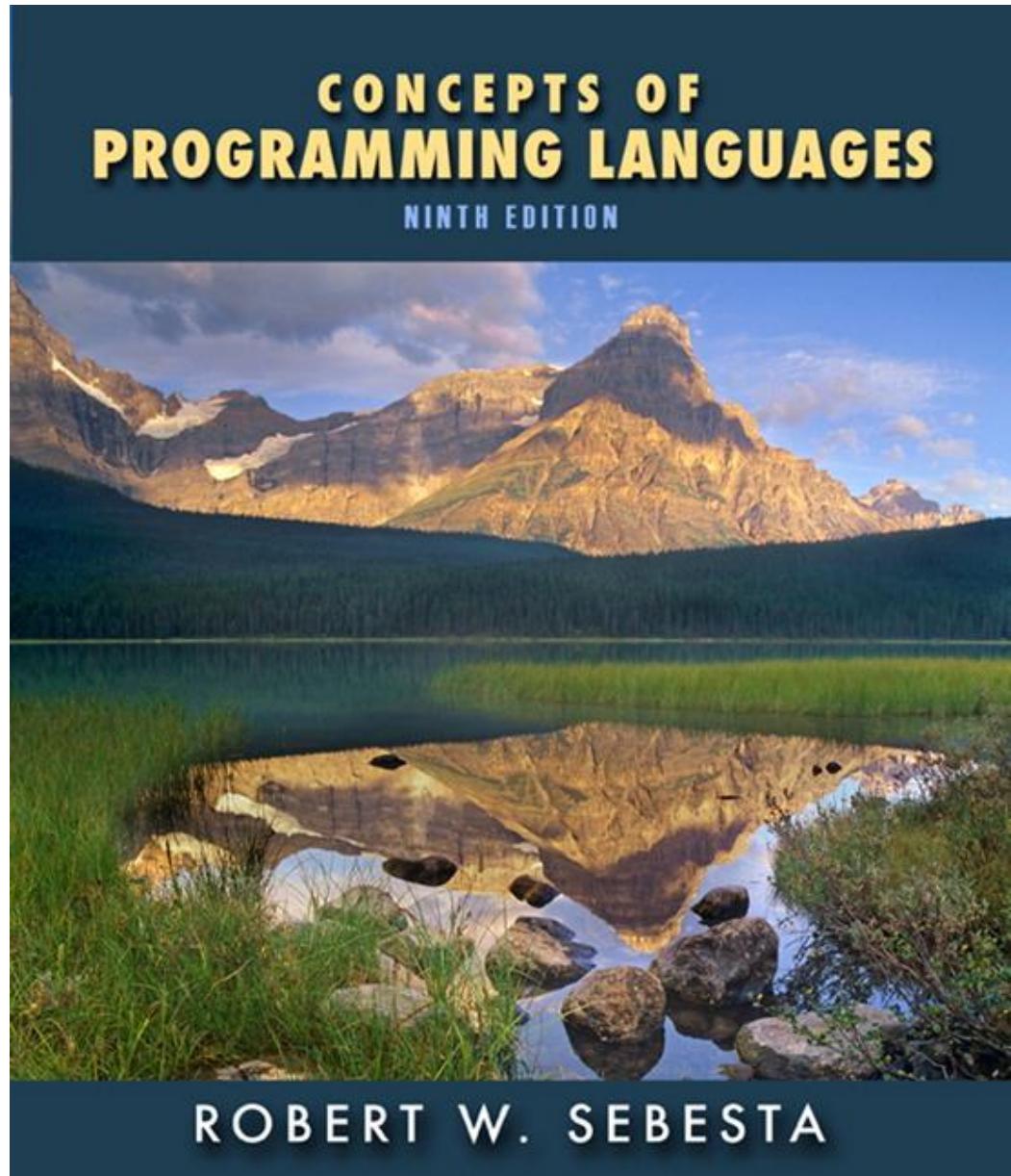


Bölüm 1

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 etrafıca ilerletilmesi

Programlama Alanları

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

Dil Değerlendirme Kriterleri

- **Okunabilirlik:** programın okunabilir ve anlaşılabilir kolaylığı
- **Yazılabilirlik:** program oluşturmada kullanılacak dilin kolaylığı (uygunluğu)
- **Güvenilirlik:** şartnamelerin uygunluğu (örneğin, kendi şartnamelerin yapılması)
- **Maliyet:** 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
- Ortogonalilik
 - 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 Ortogonalilik
 - 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: Güvenirlilik

- 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» yaklaşımının 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

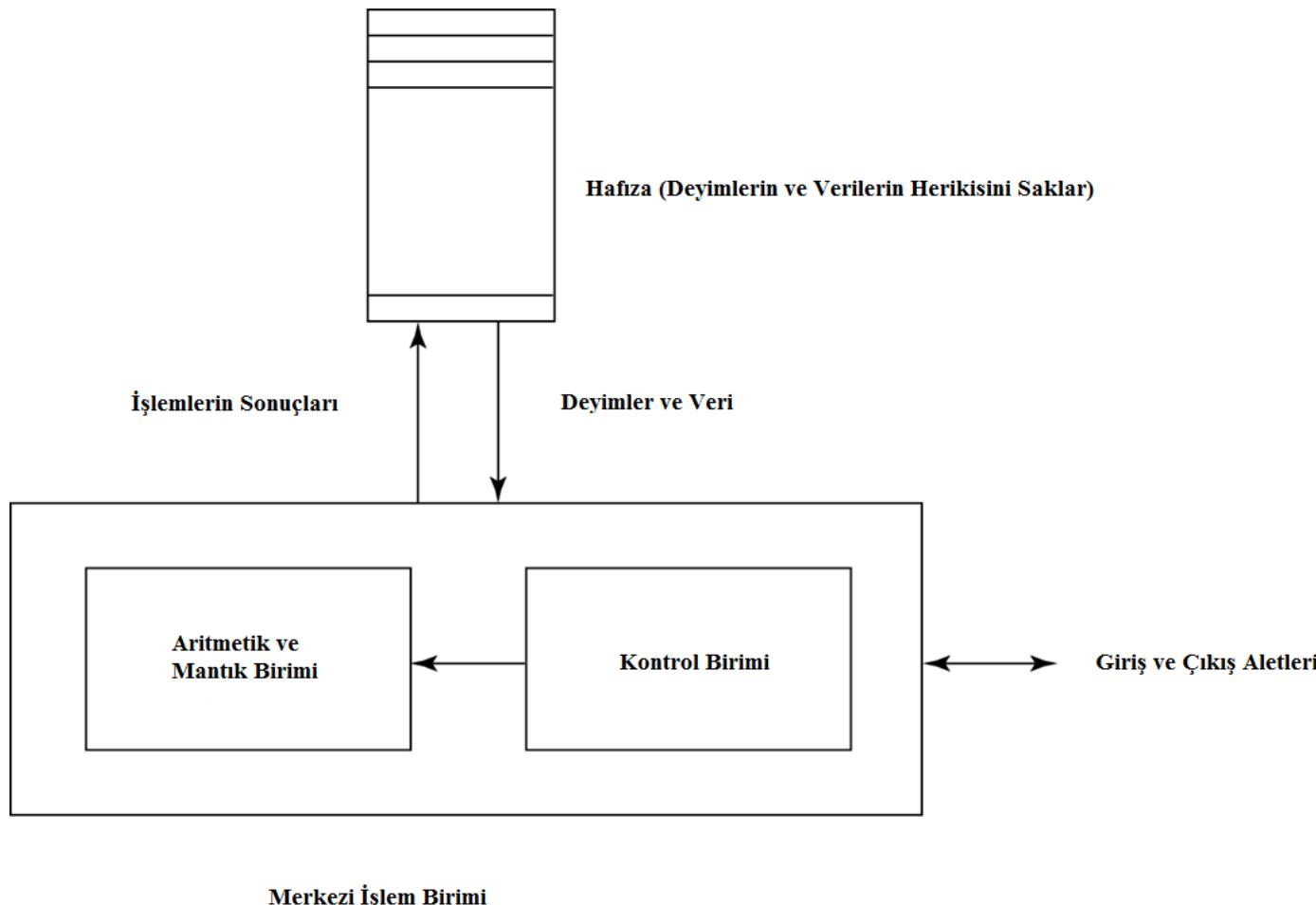
Dil Tasarımına Etki Eden Faktörler

- 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 iletılır
 - 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 sureklidön

sayac tarafından işaretlenen deyimleri getir

sayaci artır

komutu coz

komutu calistir

end repeat

Programlama Metodolojilerinin Etkileri

- 1950li yıllar ve 1960'lı yılların başı: Basit uygulamalar; makine verimliliği hakkında kaygılar
- 1960'lı yılların sonu: Halk verimliliği önem kazandı; okunabilirlik, daha iyi kontrol yapıları
 - Yapılandırılmış programlama
 - Büyüktен küçüğe 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
 - 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
 - Hesaplamaları icra ederken asıl araç, verilen parametreler için fonksiyonların uygulanmasıdır
 - Örnek: LISP, Scheme
- Mantıksal
 - Kurala dayalı (kurallar belirli bir sırada özelleştirilmmez)
 - Örnek: Prolog
- Biçimleme/hibrid programlama
 - 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 yazılırlılık**

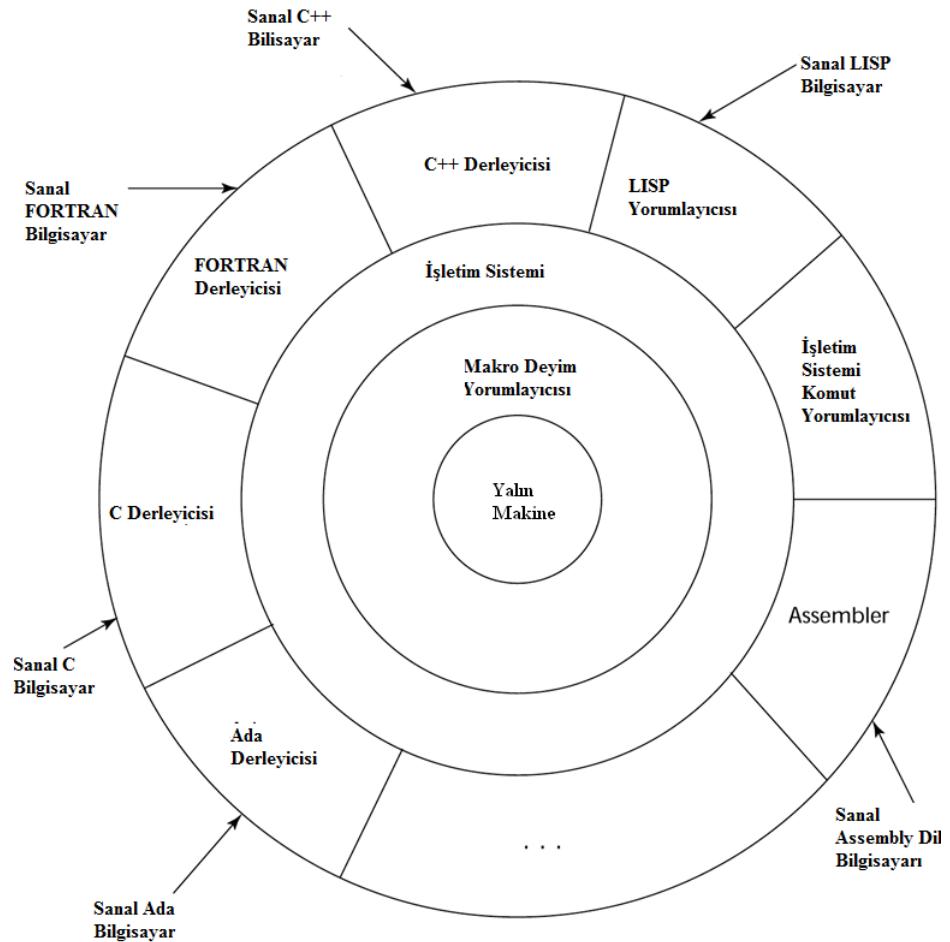
Örnek: APL birçok güçlü operatörü (ve büyük miktarlarda yeni semboller) sağlar. Bu durum kompakt bir program içinde kompleks hesaplamaların yazılmasına müsaade eder. Fakat zayıf okunabilirlik maliyetin artmasına neden olur.
- **Yazılabilirlik (esneklik) ve güvenilirlik**
 - Örnek: C++ işaretleyicileri güçlündür ve çok esnektir, fakat güvenilir değildirler

Uygulama Yöntemleri

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

Bilgisayarın Tabakalaştırılmış Görünümü

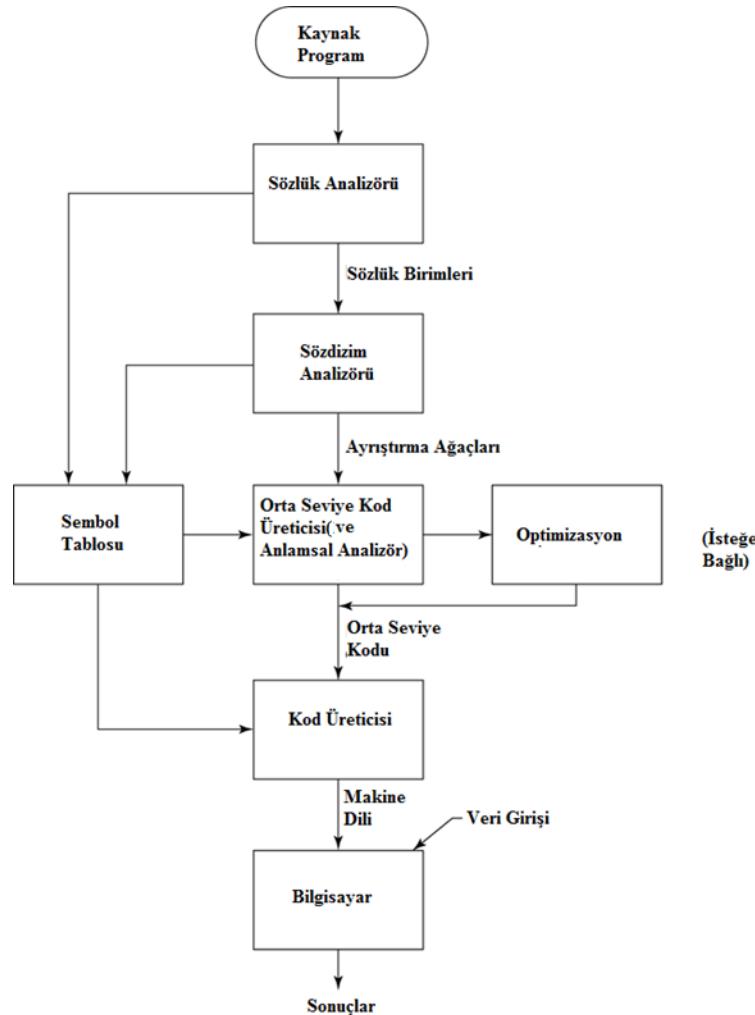
İşletim sistemi ve dil yorumlaması bir bilgisayarın makine arayüzü üzerinden tabakalaştırılır



Derleme

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

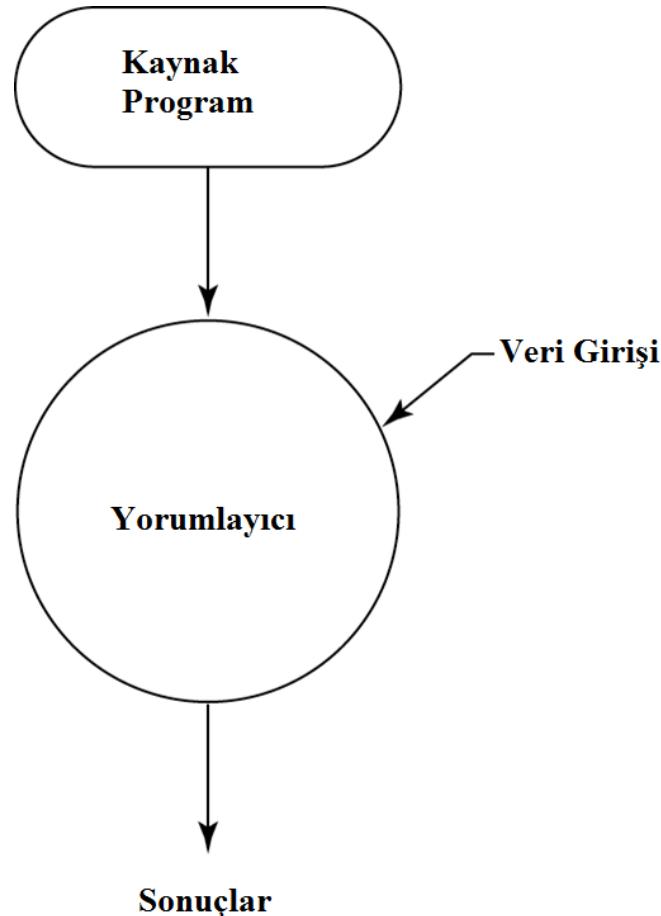
Von Neumann Darboğazı

- 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 (tikanıklığa) 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)

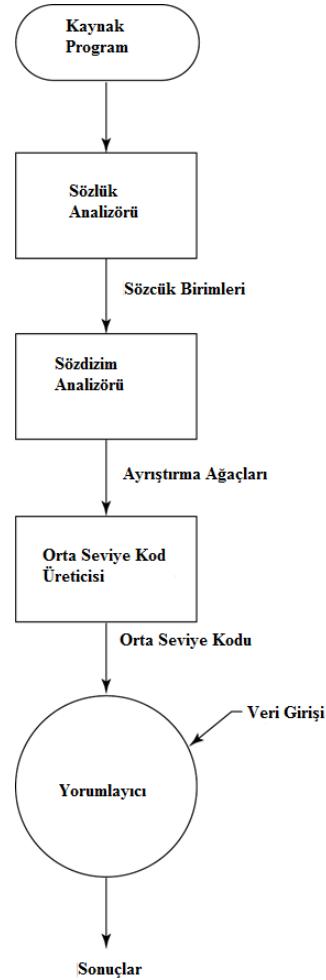
Sade Yorumlama İşlemi



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 Metin (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

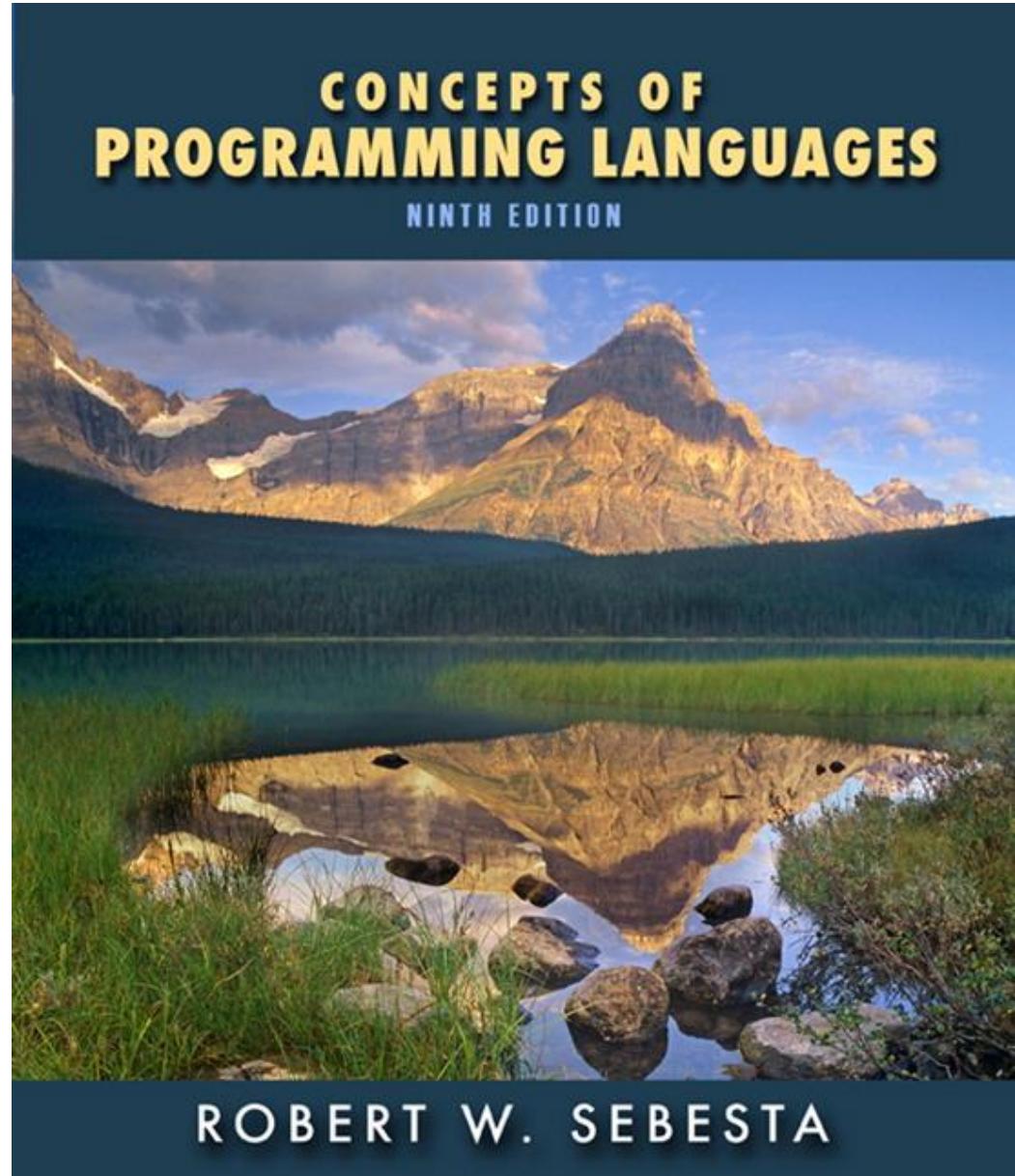
- Yazılım geliştirilmesinde kullanılan araçların bir koleksiyonu
- **UNIX**
 - Eski bir işletim sistemi ve araç koleksiyonu
 - Günümüzde sıkça GUI ile birlikte kullanılır (CDE, KDE, veya GNOME) Bunlar UNIX'in üst seviyesinde çalışır
- **Microsoft Visual Studio.NET**
 - Büyük, kompleks görsel çevre
- 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 tasarımindan 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

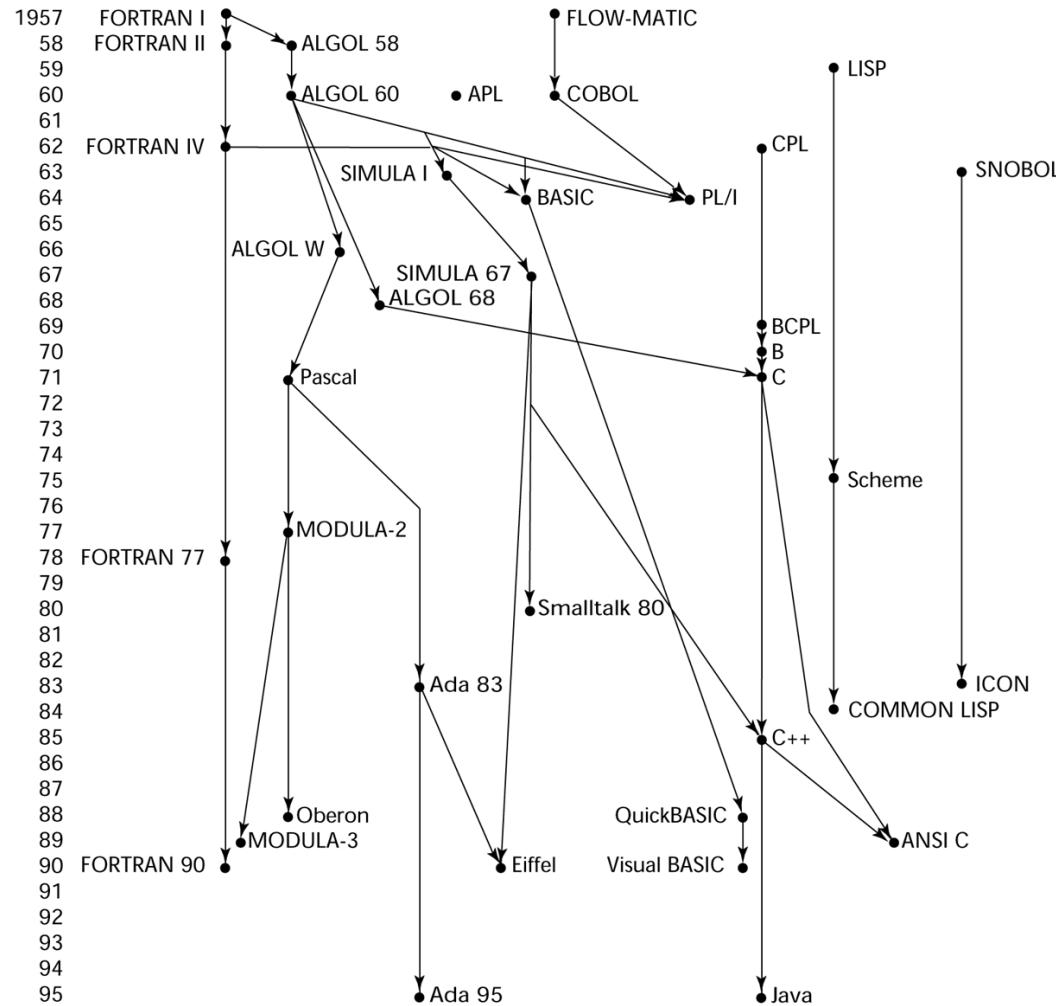
Bölüm 2 Konular (devam)

- Herkes için hersey: PL/I
- İlk İki Dinamik Dil: APL ve SNOBOL
- Veri Soyutlamanın Başlangıcı: SIMULA 67
- Ortogonal Tasarım: ALGOL 68
- ALGOL'erin ilk torunlarından bazıları
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 Söz dizimi

- 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?
 - Kötü okunabilirlik
 - Kötü 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	

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 arada kalmaktadır

Sözde kodlar: İlgili Sistemler

- UNIVAC Sistem Derleme
 - Grace Hopper liderliğinde bir ekip tarafından geliştirildi
 - Sözdekode 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
 - Programlama metodolojisi ve araçları yoktu
 - Makine verimliliği en önemli sorundu

Fortran Tasarım Süreci

- Fortran I tasarımında çevrenin etkisi
 - Dinamik depolamaya gerek yok
 - İyi 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 uzunluğuna 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

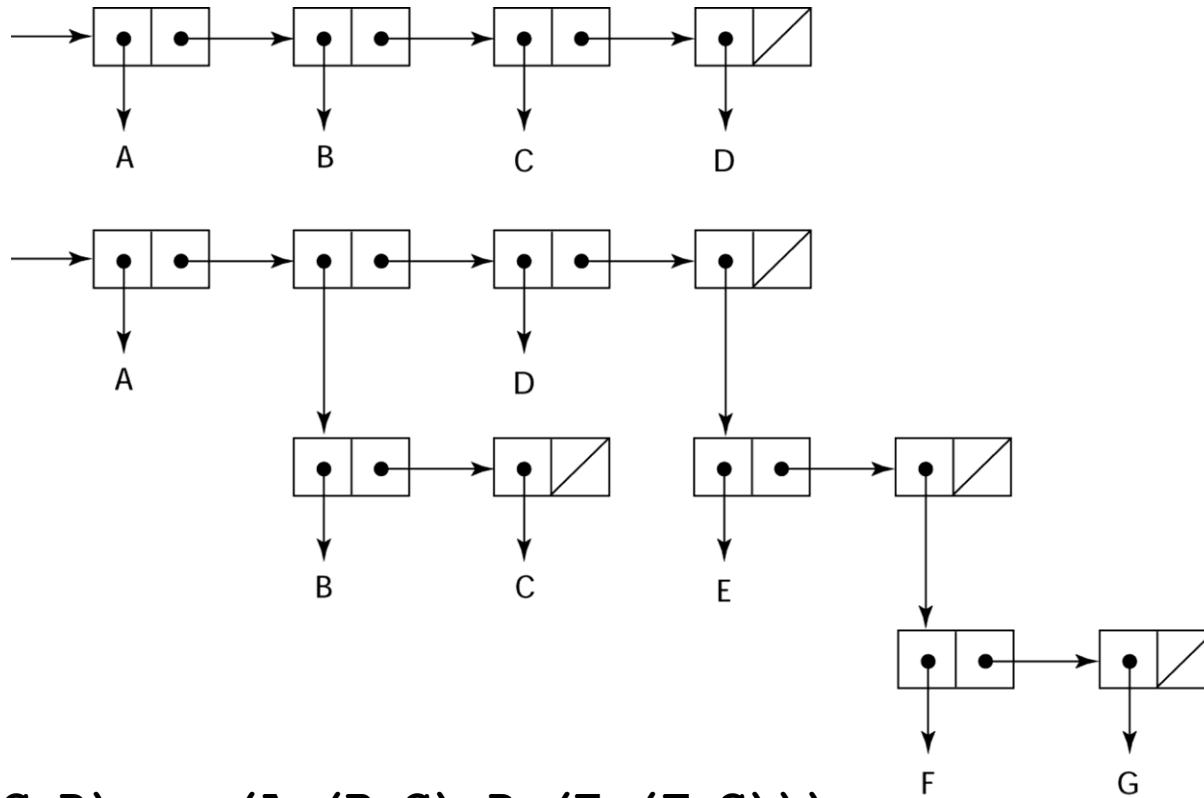
Fortran Değerlendirmesi

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

Fonksiyonel Programlama: LISP

- LISt 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
- ML, Miranda ve Haskell ilgili dillerdir

Scheme

- 1970'lerin ortalarında MIT'de geliştirildi
- Küçüktür
- Statik kapsamın yaygın kullanımı
- Birinci sınıf varlıklar olarak işlevleri
- 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 değildilerdi; hepsi makineye bağlı idiler
 - Algoritma haberleşmesi için universal bir dil değildi
- ALGOL 60 universal 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
- Simgeler köşeli parantezler içine yerleştirildi
- Bileşik ifadeler (**begin** . . . **end**)
- Bir deyimi ayırcı 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 özelliklerি
 - Blok yapı (yerel kapsam)
 - İki parametre geçişи 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 yayınılamak 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)

- Başarısızlıklar
 - Özellikle US'de asla yaygın olarak kullanılmadı
 - Nedenleri
 - I/O eksikliği ve karakter seti eksikliği yüzünden programları taşınmaz yapmaktaydı
 - Çok fazla esnek idi-ugulama zordu
 - Fortran köklü (Entrenchment) idi
 - Örgün sözdizimi açıklaması
 - 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 temelliidir
- 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 önemini de yapabilecek yeni bir dil tasarlama

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 tasarılmıştı
 - Çok büyük ve çok karmaşıktı

İlk iki Dinamik Dil : APL ve SNOBOL

- Dinamik yazma ve bellek tahsis 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 süreğelen gelişmesindendir, fakat o dilin üstü değildir
- Birçok yeni fikirlerin kaynağıdır (dilin kendisi hiçbir zaman yaygın kullanıma ulaşamamasına rağmen)
- Tasarım ortogonal kavramına dayanmaktadır
 - Birkaç temel kavamlar, 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
 - Genéric 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 çıktı

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++ nin 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 standarı 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 eksiki, 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++ Temelliidir
 - 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 ayrıt edici ad, değişken adının ilk karakteri ile gösterilir
 - Güçlü, fakat tehlikeli
 - Web CGI programlama için yaygın kullanımı kazanmış
 - Ayrıca UNIX sistem yönetimi dil için yedek olarak 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 yazılabılır, 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 yazılabılır, 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 metodlar olarak uygulanır
 - Sade yorumlayıcıdır

Yeni Milenyum için C Temelli bir Dil: C#

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

İşaretleme(Markup)/Programlama Hibrit Diller

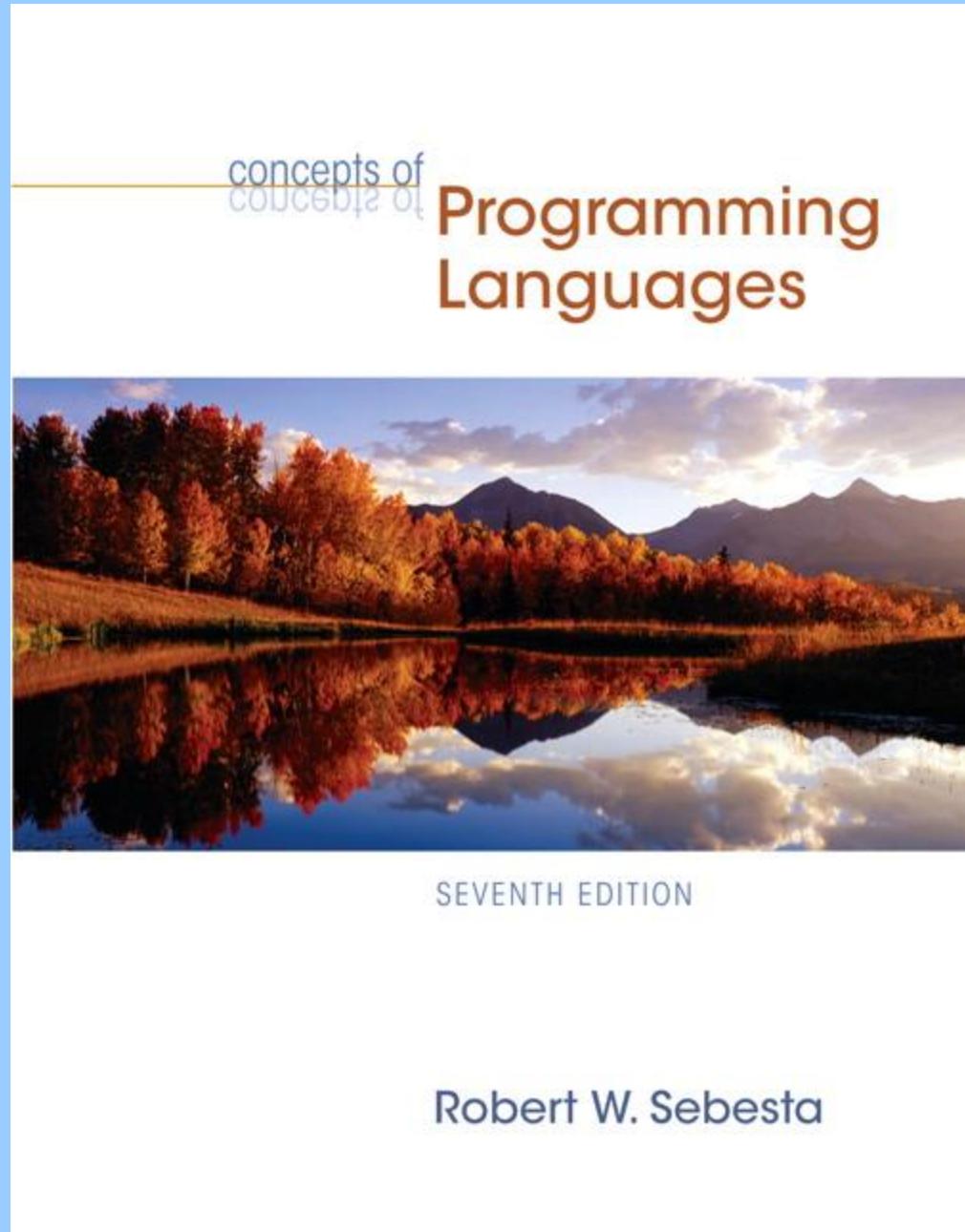
- XSLT
 - eXtensible Markup Language (XML)(genişletilebilir işaretleme dili): bir metamarkup dili
 - eXtensible Stylesheet Language Transformation (XSLT)(genişletilebilir stilsayfası dil dönüşümü) XML dökümanlarını görüntülenebilmesi için dönüştürür
 - Programlama yapıları (örn., döngüler)
- JSP
 - Java 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 tasarımindaki mevcut sorunlara bakış açısı

Bölüm 3

Sentaks ve
Semantiği
tanımlama



SEVENTH EDITION

Robert W. Sebesta

ISBN 0-321-33025-0

Bölüm 3 Konuları

1. Giriş
2. Genel Sentaks tanımlama problemi
3. Sentaks tanımlamanın biçimsel metotları
4. Özellik(Attribute) Gramerleri
5. Programların anlamlarını açıklamak:
Dinamik Semantik

3.1 Giriş

- **Sentaks(Syntax)**: ifadelerin(statements), deyimlerin(expressions), ve program birimlerinin biçimini veya yapısını
- **Semantik(Semantics)**: deyimlerin, ifadelerin, ve program birimlerinin anlamını
- Sentaks ve semantik bir dilin tanımı sağlar
 - Bir dil tanımının kullanıcıları
 - Diğer dil tasarımcıları
 - **Uygulamacılar(Implementers)**
 - Programcılar (dilin kullanıcıları)

3.2 Genel Sentaks tanımlama problemi : Terminoloji

- Bir *cümle*(*sentence*) herhangi bir alfabe'de karakterlerden oluşan bir stringdir
- Bir *dil*(*language*) cümlelerden oluşan bir kümedir
- Bir *lexeme* bir dilin en alt seviyedeki sentaktik(*syntactic*) birimidir (örn., *, sum, begin)
- Bir *simge* (*token*) lexemelerin bir kategorisidir (örn., **tanıtıçı**(*identifier*))

Dillerin formal tanımları

- **Tanıycılar(Recognizers)**

- Bir tanıma aygıtı bir dilin girdi stringlerini okur ve girdi stringinin dile ait olup olmadığına karar verir
 - Örnek: bir derleyicinin sentaks analizi kısmı
 - Bölüm 4'te daha detaylı anlatılacak

- **Üreteçler(Generators)**

- Bir dilin cümlelerini(sentences) üreten aygıttır
 - Belli bir cümlein(sentence) sentaksının doğru olup olmadığı, üreticin(generator) yapısıyla karşılaştırılarak anlaşılabilir

3.3 Sentaks tanımlamanın biçimsel metotları

- Backus–Naur Form ve **Bağlam–Duyarsız** (Context-Free) Gramerler
 - Programlama dili sentaksını tanımlamayan en çok bilinen metottur.
- (Genişletilmiş)Extended BNF
 - BNF'un okunabilirliği ve yazılabilirliğini arttırmır
- Gramerler (grammars) ve tanıyıcılar (recognizers)

BNF ve Bağlam-Duyarsız (Context-Free) Gramerler

- Bağlam-Duyarsız(Context-Free) Gramerler
 - Noam Chomsky tarafından 1950lerin ortalarında geliştirildi
 - Dil **üreteçleri**(generators), doğal dillerin sentaksını tanımlama amacıyla
 - Bağlam-Duyarsız(Context-Free) diller adı verilen bir diller sınıfı tanımlandı

Backus–Naur Form (BNF)

- Backus–Naur Form (1959)
 - John Backus tarafından Algol 58'i belirlemek için icat edildi
 - BNF bağlam-duyarsız (context-free) gramerlerin eşdeğeridir
 - BNF başka bir dili tanımlamak için kullanılan bir *metadilidir*(*metalinguage*)
 - BNF'de, soyutlamalar(abstractions) sentaktik(syntactic) yapı sınıflarını temsil etmek için kullanılır--sentaktik değişkenler gibi davranışlarılar (*nonterminal semboller* adı da verilir)

BNF'un Temelleri

- Non-terminaller(uçbirim olmayanlar): BNF soyutlamaları
- Terminaller(uçbirimler): lexemeler ve simgeler(tokens)
- Gramer: kurallar(rules) koleksiyonu
 - BNF kurallarından(rules) örnekler:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

BNF Kuralları(Rules)

- Bir kuralın bir sol-tarafı (left-hand side (LHS)) ve bir sağ tarafı (right-hand side (RHS)) vardır, ve *terminal* ve *nonterminal* sembollerden oluşur
- Bir gramer, kurallarının(rules) boş olmayan sonlu bir kümesidir
- Bir soyutlama (abstraction)(veya nonterminal simbol) birden fazla RHS'ye sahip olabilir

```
<stmt> → <single_stmt>
      | begin <stmt_list> end
```

Listeleri(Lists) Tanımlama

- Sentaktik listeler özyineleme(recursion) kullanılarak tanımlanır

```
<ident_list> → ident  
          | ident, <ident_list>
```

- Bir türev(derivation), başlangıç sembolüyle başlayan ve bir cümleyle(sentence)(tüm terminal semboller) biten kuralların(rules) tekrarlamalı bir uygulamasıdır.

Bir Gramer Örneği

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

Bir türev(derivation) örneği

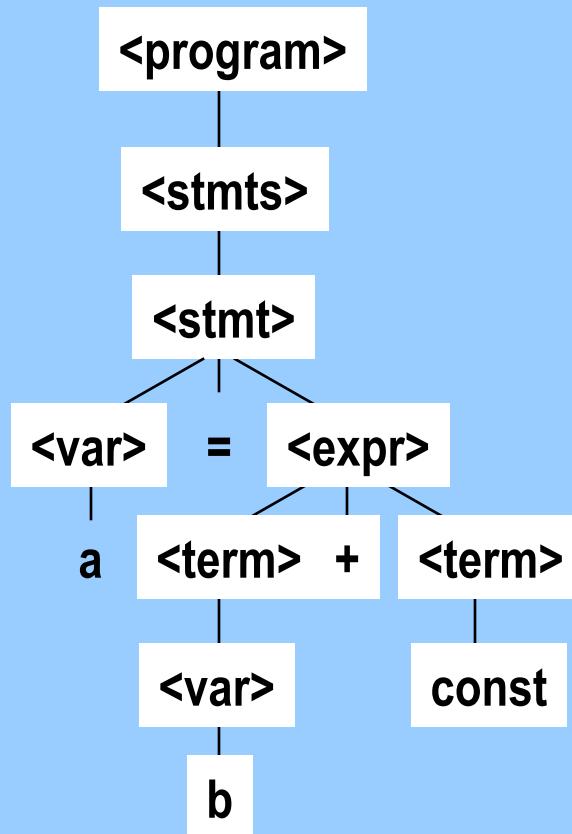
```
<program> => <stmts> => <stmt>
                  => <var> = <expr> => a = <expr>
                  => a = <term> + <term>
                  => a = <var> + <term>
                  => a = b + <term>
                  => a = b + const
```

Türev(Derivation)

- Bir türevde yar alan bütün sembol stringleri cümlesel biçimdedir(sentential form)
- Bir cümle(sentence) sadece terminal semboller içeren cümlesel bir biçimdir
- Bir ensol türev(leftmost derivation), içindeki her bir cümlesel biçimdeki ensol nonterminalin genişletilmiş olmadığı türevdir(is one in which the leftmost nonterminal in each sentential form is the one that is expanded)
- Bir türev(derivation) ensol(leftmost) veya ensağ (rightmost) dan her ikisi de olmayabilir

Ayrıştırma Ağacı (Parse Tree)

- Bir türevin(derivation) hiyerarşik gösterimi



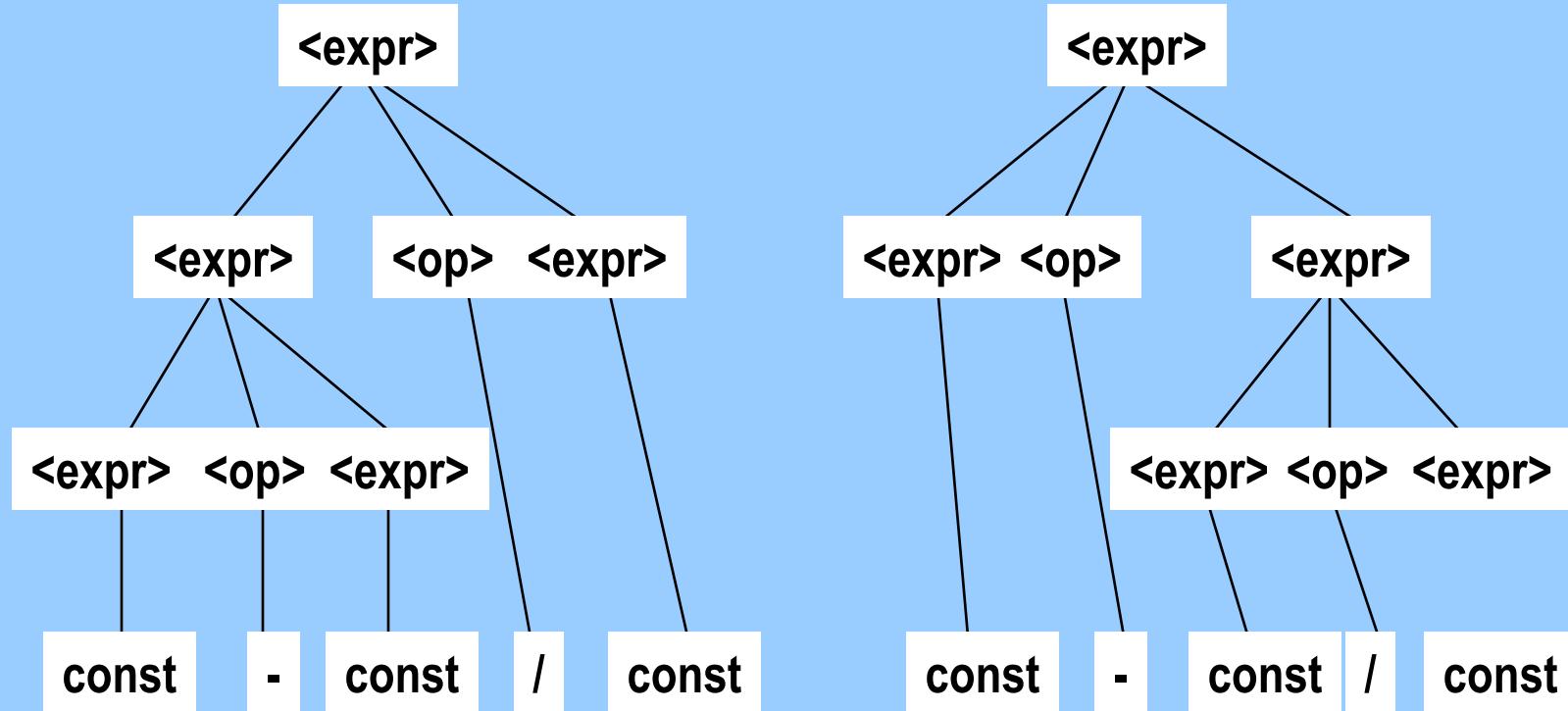
Gramerlerde Belirsizlik(Ambiguity)

- Bir grammer ancak ve ancak(iff) iki veya daha fazla farklı ayrıştırma ağacı(parse trees) olan bir cümlesel biçim(sentential form) üretiyorsa *belirsizdir(ambiguous)*

Bir Belirsiz Deyim Grameri

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \quad | \quad \text{const}$

$\langle \text{op} \rangle \rightarrow / \quad | \quad -$

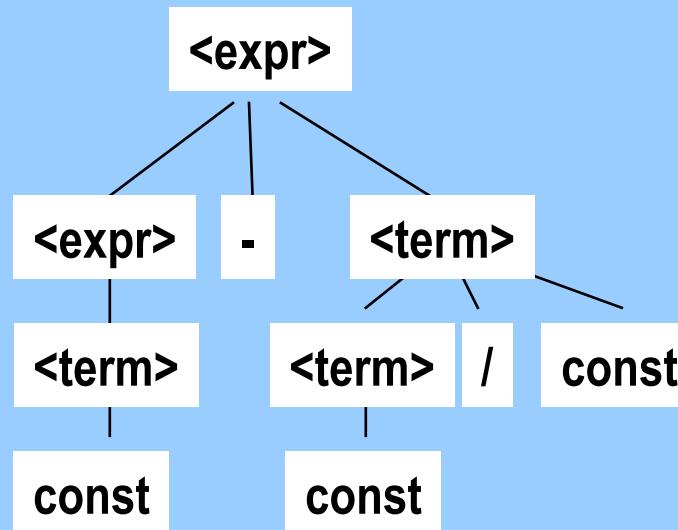


Bir belirsiz olmayan(Unambiguous) deyim Grameri

- Eğer ayrıştırma ağacını(parse tree) operatörlerin öncelik(precedence) seviyelerini göstermek için kullanırsak, belirsizlik elde edemeyiz

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle - \langle \text{term} \rangle \mid \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle / \text{const} \mid \text{const}$

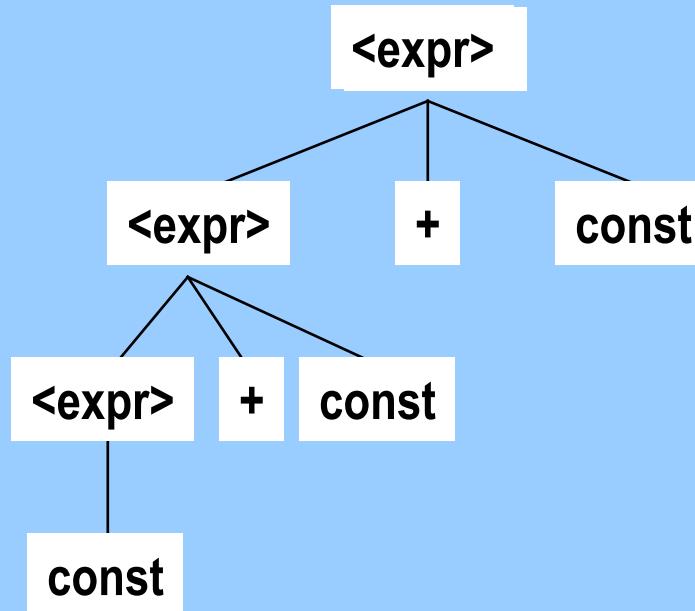


Operatörlerin Birleşirliği(Associativity)

- Operatör birleşirliği(associativity) de gramerle gösterilebilir

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \text{const}$ (ambiguous)

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \text{const} \mid \text{const}$ (unambiguous)



Extended BNF(Genişletilmiş BNF)

- Opsiyonel kısımlar köşeli parantez içine yerleştirilir ([])

`<proc_call> -> ident [(<expr_list>)]`

- RHS lerin(sağ-taraf) alternatif kısımları parantezler içine yerleştirilir ve dikey çizgilerle ayrılır

`<term> → <term> (+|-) const`

- Tekrarlamalar(Repetitions) (0 veya daha fazla) süslü parantez ({ }) içine yerleştirilir

`<ident> → letter {letter|digit} brace`

BNF ve EBNF

- **BNF**

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \\&\quad | \quad \langle \text{expr} \rangle - \langle \text{term} \rangle \\&\quad | \quad \langle \text{term} \rangle \\ \langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \\&\quad | \quad \langle \text{term} \rangle / \langle \text{factor} \rangle \\&\quad | \quad \langle \text{factor} \rangle\end{aligned}$$

- **EBNF**

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle \} \\ \langle \text{term} \rangle &\rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle \}\end{aligned}$$

3.4 Özellik(Attribute) Gramerleri

- Bağlam-duyarsız(context-free) gramerler (CFGs) bir programlama dilinin bütün sentaksını tanımlayamazlar
- Ayristirma ağaçlarıyla(parse trees) birlikte bazı semantik bilgiyi taşıması için CFG'lere eklemeler
- Özellik(attribute) gramerlerinin(AGs) birincil değerleri :
 - Statik semantik belirtimi
 - Derleyici(Compiler) tasarımı (statik semantik kontrolü)

Özellik(Attribute) Gramerleri: Tanım

- Bir özellik grameri $G = (S, N, T, P)$ aşağıdaki eklemelerle birlikte bir bağlam-duyarsız gramerdir :
 - Her bir x gramer sembolü için özellik değerlerinden(attribute values) oluşan bir $A(x)$ kümesi vardır
 - Her kural(rule), içindeki nonterminallerin belirli özelliklerini(attributes) tanımlayan bir fonksiyonlar kümesine sahiptir
 - Her kural, özellik tutarlılığını(consistency) kontrol etmek için karşılaştırma belirtimlerinden(predicate) oluşan (boş olabilir) bir kümeye sahiptir

Özellik Gramerleri: Tanım

- $X_0 \rightarrow X_1 \dots X_n$ bir kural olsun
- $S(X_0) = f(A(X_1), \dots, A(X_n))$ biçimindeki fonksiyonlar *sentezlenmiş özellikler* (*synthesized attributes*) tanımlar
- $I(X_j) = f(A(X_0), \dots, A(X_n)), i \leq j \leq n$ için, şeklindeki fonksiyonlar *miras alınmış özellikler* (*inherited attributes*) tanımlar
- Başlangıçta, yapraklarda *yerleşik özellikler* (*intrinsic attributes*) vardır

Özellik Gramerleri : Örnek

- **Sentaks**

`<assign> -> <var> = <expr>`

`<expr> -> <var> + <var> | <var>`

`<var> A | B | C`

- **actual_type:** `<var>` ve `<expr>` ile sentezlenmiştir
- **expected_type:** `<expr>` ile miras bırakılmıştır

3.4 Özellik Gramerleri (örn.)

- Sentaks kuralı: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[1] + \langle \text{var} \rangle[2]$
Semantik kurallar:

$\langle \text{expr} \rangle.\text{actual_type} \leftarrow \langle \text{var} \rangle[1].\text{actual_type}$

Karşılaştırma belirtimi (Predicate):

$\langle \text{var} \rangle[1].\text{actual_type} == \langle \text{var} \rangle[2].\text{actual_type}$

$\langle \text{expr} \rangle.\text{expected_type} == \langle \text{expr} \rangle.\text{actual_type}$

- Sentaks kuralı: $\langle \text{var} \rangle \rightarrow \text{id}$

Semantik kuralı:

$\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{lookup } (\langle \text{var} \rangle.\text{string})$

3.4 Özellik Gramerleri (örn.)

- Özellik(attribute) değerleri nasıl hesaplnır?
 - Eğer bütün özellikler miras alınmışsa, ağaç yukarıdan- aşağıya(top-down order) şekilde düzenlenir
 - Eğer özellikler sentezlenmişse, ağaç aşağıdan- yukarıya(bottom-up order) şekilde düzenlenir.
 - Çoku kez, bu iki çeşit özelliğin her ikisi de kullanılır, ve aşağıdan-yukarıya ve yukarıdan- aşağıya düzenlerin kombinasyonu kullanılmalıdır.

3.4 Özellik Gramerleri (örn.)

<expr>.expected_type ← ebeveyninden miras almıştır

<var>[1].actual_type ← lookup (A)

<var>[2].actual_type ← lookup (B)

<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type

<expr>.actual_type =? <expr>.expected_type

3.5 Semantik

- Semantiği tanımlamak için yaygın kabul edilmiş tek bir gösterim veya formalizm yoktur
- İşlemsel(Operational) Semantik
 - Bir programı simulasyon veya gerçek olarak makine üzerinde çalıştırarak anlamını açıklamaktır. Makinenin durumundaki(state) değişme (bellek, saklayıcılar(registers), vs.) ifadenin anlamını tanımlar

3.5 Semantik (devamı)

- Yüksek-düzenli bir dil için işlemsel semantiği kullanmak için, bir sanal (virtual) makine gereklidir
- **Donanım** saf yorumlayıcı(pure interpreter) çok pahalı olacaktır
- **Yazılım** saf yorumlayıcının bazı problemleri:
 - Bilgisayara özgü ayrıntılı özellikler faaliyetlerin anlaşılmasını zorlaştırır
 - Böyle bir semantik tanımı makine-bağımlı olurdu

İşlemsel(Operational) Semantik

- Daha iyi bir alternatif: Tam bir bilgisayar simülasyonu
- İşlem:
 - Bir çevirmen(translator) oluştur (kaynak kodunu, idealleştirilen bir bilgisayarın makine koduna çevirir)
 - İdealleştirilen bilgisayar için bir simülator oluştur
- İşlemsel semantiğin değerlendirmesi:
 - Informal olarak kullanılırsa iyidir(dil el kitapları, vs.)
 - Formal olarak kullanılırsa aşırı derecede karmaşıktır(örn., VDL), PL/I'ın semantiğini tanımlamak için kullanılıyordu.

3.5 Semantik

- Aksiyomatik(Axiomatic) Semantik
 - Biçimsel mantığa(formal logic) dayalıdır (predicate calculus)
 - Orjinal amaç: biçimsel program doğrulaması(verification)
 - Yaklaşım: Dildeki her bir ifade tipi için aksiyomlar veya çıkarsama kuralları(inference rules) tanımlamak(deyimlerin(expressions) diğer deyimlere dönüştürülmesine imkan sağlamak için)
 - Deyimlere **iddia**(assertions) adı verilir

Aksiyomatic Semantik

- Bir ifadenin önündeki bir iddia(assertion) (bir önsart(precondition)), çalıştırıldığı zaman değişkenler arasında true olan ilişki ve kısıtları(constraints) belirtir
- Bir ifadenin arkasından gelen iddiaya sonşart(postcondition) denir
- En zayıf önsart(weakest precondition), sonşartı garanti eden asgari kısıtlayıcı önsarttır

Aksiyomatik Semantik

- Ön-son(Pre-post) biçim :
 $\{P\}$ ifade $\{Q\}$
- Bir örnek: $a = b + 1 \quad \{a > 1\}$
Mümkün bir önşart: $\{b > 10\}$
En zayıf önşart: $\{b > 0\}$

Aksiyomatik Semantik

- Program ispat(proof) işlemi: Bütün program için sonşart istenen sonuçtur. Programda ilk ifadeye kadar geriye doğru çalışılır. Birinci ifadedeki önsart program şartnamesiyle(spec.) aynıysa, program doğrudur.

Aksiyomatik Semantik

- Atama ifadeleri için bir aksiyomdur
 $(x = E)$:

$$\{Q_{x \rightarrow E}\} x = E \{Q\}$$

- Sonuç(Consequence) kuralı:

$$\frac{\{P\} S \{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\} S \{Q'\}}$$

Aksiyomatik Semantik

- Sıralar (sequences) için
çıkarsama(inference)
kuralı

Bir $S_1;S_2$ sırası için:

$\{P_1\} S_1 \{P_2\}$

$\{P_2\} S_2 \{P_3\}$

Çıkarsama kuralı:

$$\frac{\{P_1\}S_1\{P_2\}, \{P_2\}S_2\{P_3\}}{\{P_1\}S_1;S_2\{P_3\}}$$

Aksiyomatik Semantik

- Mantıksal öntest döngüleri için bir çıkışsama kuralı

Döngü yapısı için:

{P} while B do S end {Q}

çıkarsama kuralı:

$$\frac{(I \text{ and } B) S \{ I \}}{\{ I \} \text{ while } B \text{ do } S \{ I \text{ and } (\text{not } B) \}}$$

I döngü sabiti(invariant) ise.
(tümevarımsal hipotez(inductive hypothesis))

Aksiyomatik Semantik

- Döngü sabitinin(invariant) özellikleri
I , aşağıdaki şartları sağlamalıdır:
 - $P \Rightarrow I$ (döngü değişkeni başlangıçta true olmalı)
 - $\{I\} B \{I\}$ (Boolean hesabı I'nin doğruluğunu(geçerliliğini) değiştirmemelidir)
 - $\{I \text{ and } B\} S \{I\}$ (Döngü gövdesinin çalıştırılmasıyla I değişmez)
 - $(I \text{ and } (\text{not } B)) \Rightarrow Q$ (I true ise ve B false ise, Q bulunur(implied))
 - Döngü sonlanır (ispatlamak zor olabilir)

Aksiyomatik Semantik

- Döngü sabiti I , döngü sonşartının zayıflatılmış bir sürümür, ve aynı zamanda bir önsarttır.
- I , döngünün başlamasından önce yerine getirilebilecek kadar zayıf olmalıdır, fakat döngü çıkış şartıyla birleştirildiğinde, sonşartın doğru olmasını zorlayacak kadar güçlü olmalıdır

3.5 Semantik (devamı)

- Aksiyomatik Semantiğin değerlendirilmesi:
 - Bir dildeki bütün ifadeler için aksiyom ve çıkışsama(inference) kuralları geliştirmek zordur
 - İspatların doğruluğu için iyi bir araçtır, ve programlama mantığı için mükemmel bir çatıdır framework for reasoning about programs, fakat dil kullanıcıları ve derleyici yazarlar için kullanışlı değildir
 - Programlama dilinin anlamını tanımlamadaki yararlılığı dil kullanıcıları veya derleyici yazarları için sınırlanmıştır

3.5 Semantik (devamı)

- Denotasyonel Semantik
 - Özyinelemeli(recursive) fonksiyon teorisine dayalıdır
 - En soyut semantik tanımlama metodudur
 - Scott ve Strachey (1970) tarafından geliştirilmiştir

Denotasyonel(Denotational) Semantik

- Dil için denotasyonel şartnameler(spec) oluşturma işlemidir (kolay sayılmaz):
 - Her dil varlığı(entity) için matematiksel bir nesne tanımlanır
 - Dil varlıklarının(entity) örneklerini(instances) karşılık gelen matematiksel nesnelerin örneklerine eşleştirilen bir fonksiyon tanımlanır
- Dil yapılarının(construct) anımları sadece programın değişkenlerinin değerleriyle tanımlanır

3.5 Semantik (devamı)

- Denotasyonel and işlemsel(operational) semantik arasındaki fark: İşlemsel semantikte, durum(state) değişimleri kodlanmış algoritmalarla tanımlanır ; denotasyonel semantikte, sıkı matematiksel fonksiyonlarla tanımlanır

Denotasyonel Semantik

- Programın durumu(state) bütün güncel değişkenlerinin değerleridir

$$s = \{ \langle i_1, v_1 \rangle, \langle i_2, v_2 \rangle, \dots, \langle i_n, v_n \rangle \}$$

- **VARMAP** öyle bir fonksiyon olsun ki, bir değişkenin adı ve durumu verildiğinde, o değişkenin güncel değerini döndürsün

$$\text{VARMAP}(i_j, s) = v_j$$

3.5 Semantik (devamı)

- Ondalık sayılar
 - Şu denotasyonel semantik tanımı, string sembollerden oluşan ondalık sayıları sayısal değerlere eşleştirir

3.5 Semantik (devamı)

$$\begin{aligned} <\text{dec_num}> \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ & \mid <\text{dec_num}> (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \\ & \quad 5 \mid 6 \mid 7 \mid 8 \mid 9) \end{aligned}$$
$$M_{\text{dec}}('0') = 0, M_{\text{dec}}('1') = 1, \dots, M_{\text{dec}}('9') = 9$$
$$M_{\text{dec}}(<\text{dec_num}> '0') = 10 * M_{\text{dec}}(<\text{dec_num}>)$$
$$M_{\text{dec}}(<\text{dec_num}> '1') = 10 * M_{\text{dec}}(<\text{dec_num}>) + 1$$

...

$$M_{\text{dec}}(<\text{dec_num}> '9') = 10 * M_{\text{dec}}(<\text{dec_num}>) + 9$$

3.5 Semantik (devamı)

- Deyimler(Expressions)
 - Deyimleri $Z \cup \{\text{error}\}$ üzerine eşleştirir
 - Deyimleri, ondalık sayılar, değişkenler, veya bir aritmetik operatör ve her biri bir deyim olabilen iki operanda sahip ikili(binary) deyimler olarak varsayıyoruz.

We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

3.5 Semantik (devamı)

```
Me(<expr>, s) Δ=
  case <expr> of
    <dec_num> => Mdec(<dec_num>, s)
    <var> =>
      if VARMAP(<var>, s) == undef
          then error
          else VARMAP(<var>, s)
    <binary_expr> =>
      if (Me(<binary_expr>.<left_expr>, s) == undef
          OR Me(<binary_expr>.<right_expr>, s) =
              undef)
          then error
      else
        if (<binary_expr>.<operator> == '+' then
            Me(<binary_expr>.<left_expr>, s) +
            Me(<binary_expr>.<right_expr>, s)
        else Me(<binary_expr>.<left_expr>, s) *
            Me(<binary_expr>.<right_expr>, s)
```

...

3.5 Semantik (devamı)

- Atama ifadeleri
 - Durum kümelerini durum kümelerine eşleştirir

```
Ma (x := E, s) Δ=
    if Me(E, s) == error
        then error
    else s' =
{<i1', v1'>, <i2', v2'>, . . . , <in', vn'>},
    where for j = 1, 2, . . . , n,
          vj' = VARMAP(ij, s) if ij <> x
          = Me(E, s) if ij == x
```

3.5 Semantik (devamı)

- Mantıksal Öntest Döngüleri(Logical Pretest Loops)
 - Durum kümelerini durum kümelerine eşleştirir

```
Ml (while B do L, s) Δ=
  if Mb(B, s) == undef
    then error
  else if Mb(B, s) == false
    then s
  else if Ms1(L, s) == error
    then error
  else Ml (while B do L, Ms1(L, s))
```

3.5 Semantik (devamı)

- Döngünün anlamı; program değişkenlerinin, döngüdeki ifadelerin belirtilen sayıda ve hata olmadığını varsayıarak çalıştırılmasından sonra aldığı değerleridir
- Esasında döngü, iterasyondan özyinelemeye(recursion) dönüştürülmüştür, reküratif kontrol(recursive control) matematiksel olarak diğer reküratif durum eşleştirme fonksiyonlarıyla tanımlanır
- Özyineleme(Recursion), iterasyonla(iteration) karşılaştırıldığında, matematiksel **kesinliklerle güçlüklerle (rigor)** açıklaması daha kolaydır

3.5 Semantik (devamı)

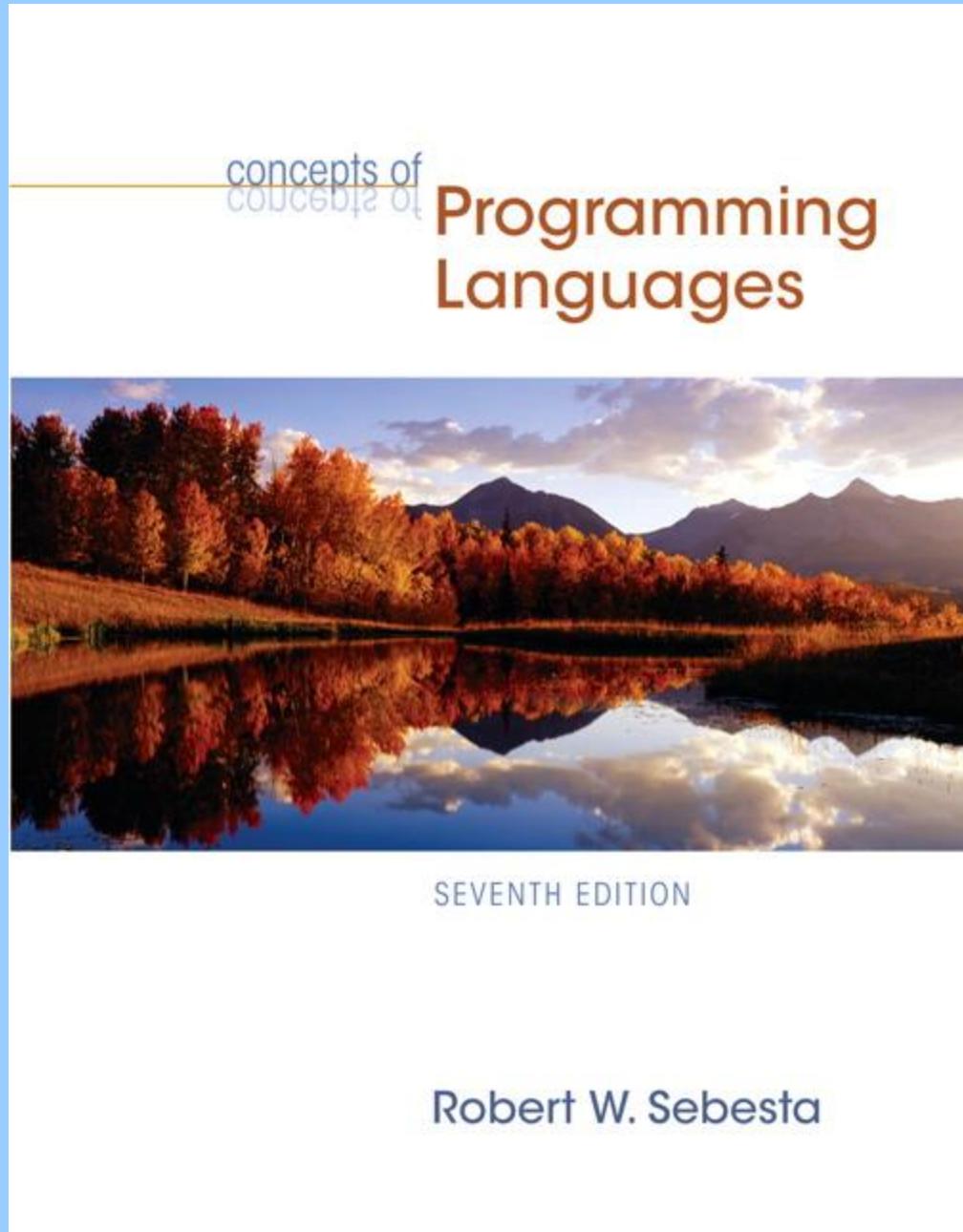
- Denotasyonel semantiğin değerlendirilmesi
 - Programların doğruluğunu ispatlama için kullanılabilir
 - Programlar hakkında düşünmek için sıkı(kesin)(rigorous) bir yol sağlar
 - Dil tasarımindan yardımcı olabilir
 - Derleyici üretme sistemlerinde kullanılmıştır
 - Karmaşıklığı yüzünden, dil kullanıcıları tarafından çok az kullanılmıştır

Özet

- BNF ve bağlam-duyarsız gramerler eşdeğer meta-dillerdir
 - Programlama dillerinin sentaksını tanımlayabilmek için uygundur
- Özellik grameri bir dilin hem sentaksını hem de semantiğini tanımlayabilen tanımlayıcı bir formalizmdir
- Semantik tanımının birincil metotları
 - İşlem(Operation), aksiyomatik, denotasyonel

Bölüm 4

Sözcüksel(Lexical)
ve Sentaks Analiz



SEVENTH EDITION

Robert W. Sebesta

ISBN 0-321-33025-0

Bölüm 4 Konular

1. Giriş
2. Sözcüksel Analiz(Lexical Analysis)
3. Ayrıştırma(Parsing) Problemi
4. Özyineli–azalan Ayrıştırma (Recursive–Descent Parsing)
5. Aşağıdan–yukarıya Ayrıştırma (Bottom–Up Parsing)

4.1 Giriş

- Dil(language) implementasyon sistemleri, belirli(specific) implementasyon yaklaşımına aldırmadan kaynak kodu(source code) analiz etmelidir
- Hemen hemen bütün sentaks analizi kaynak(source) dilin sentaksının biçimsel tanımlamasına(formal description) dayalıdır (BNF)

4.1 Giriş (Devamı)

- Bir dil işlemcisinin (language processor) sentaks(syntax) analizi bölümü genellikle iki kısımdan oluşur:
 - Bir düşük-düzenli(low-level) kısım, sözcüksel analizci (lexical analyzer) (matematiksel olarak, kurallı bir gramere(regular grammar) dayalı bir sonlu otomasyon(finite automaton))
 - Bir yüksek-düzenli(high-level) kısım, sentaks analizci(syntax analyzer), veya ayıristırıcı(parser) (matematiksel olarak, bağlam duyarsız gramere(context-free grammar) dayalı bir aşağı-itme otomasyonu(push-down automaton), veya BNF)

4.1 Giriş (Devamı)

- Sentaksi(syntax) tanımlamak için BNF kullanmanın nedenleri :
 - Net ve özlü bir sentaks tanımı(syntax description) sağlar
 - Ayrıştırıcı(parser) doğrudan BNF ye dayalı olabilir
 - BNF ye dayalı ayrıştırıcıların(parsers) bakımı daha kolaydır

4.1 Giriş (Devamı)

- Sözcüksel(lexical) ve sentaks(syntax) analizini ayırmayı nedenleri:
 - Basitlik(Simplicity) – sözcüksel analiz (lexical analysis) için daha az karmaşık yaklaşımlar kullanılabilir; bunları ayırmak ayrıştırıcıyı(parser) basitleştirir
 - Verimlilik(Efficiency) – ayırmak sözcüksel analizcinin(lexical analyzer) optimizasyonuna imkan verir
 - Taşınabilirlik(Portability) – sözcüksel analizcinin(lexical analyzer) bölümleri taşınabilir olmayabilir, fakat ayrıştırıcı(parser) her zaman taşınabilirdir

4.2 Sözcüksel(Lexical) Analiz

- Sözcüksel analizci (lexical analyzer), karakter stringleri (character strings) için desen eşleştiricidir(pattern matcher)
- Sözcüksel analizci ayırtıcı(parser) için bir “ön-uç”tur (“front-end”)
- Kaynak programın(source program) birbirine ait olan altstringlerini(substrings) tanımlar – **lexeme’ler**
 - **Lexemeler**, jeton(token) adı verilen sözcüksel(lexical) bir kategoriyle ilişkilendirilmiş olan bir karakter desenini eşleştirir
 - **sum** bir **lexemedir**; jetonu(token) **IDENT** olabilir

4.2 Sözcüksel(Lexical) Analiz (Devamı)

- Sözcüksel analizci(lexical analyzer), genellikle ayrıştırıcının sonraki jetona(token) ihtiyaç duyduğunda çağrırdığı fonksiyondur. Sözcüksel analizci(lexical analyzer) oluşturmaya üç yaklaşım:
 - Jetonların biçimsel tanımı(formal description) yazılır ve bu tanıma göre tablo-sürümlü(table-driven) sözcüksel analizcisi oluşturulan yazılım aracı(software tool) kullanılır
 - Jetonları(tokens) tanımlayan bir durum diyagramı(state diagram) tasarılanır ve durum diyagramını implement eden bir program yazılır
 - Jetonları(tokens) tanımlayan bir durum diyagramı(state diagram) tasarılanır ve el ile durum diyagramının(state diagram) tablo-sürümlü(table-driven) bir implementasyonu yapılır
- Sadece ikinci yaklaşımından bahsedeceğiz

4.2 Sözcüksel(Lexical) Analiz (Devamı)

- Durum diyagramı(State diagram) tasarıımı:
 - Saf(Naive) bir durum diyagramı(state diagram) kaynak(source) dildeki her karakterde her durumdan(state) bir geçişe(transition) sahip olacaktı – böyle bir diyagram çok büyük olurdu!

4.2 Sözcüksel(Lexical) Analiz (Devamı)

- Çoğu kez, durum diyagramı basitleştirmek için geçişler(transitions) birleştirilebilir
 - Bir tanıtıcıyı(identifier) tanırken, bütün büyük(uppercase) ve küçük(lowercase) harfler eşittir
 - Bütün harfleri içeren bir karakter sınıfı(character class) kullanılır
 - Bir sabit tamsayıyı (integer literal) tanırken, bütün rakamlar(digits) eşittir – bir rakam sınıfı(digit class) kullanılır

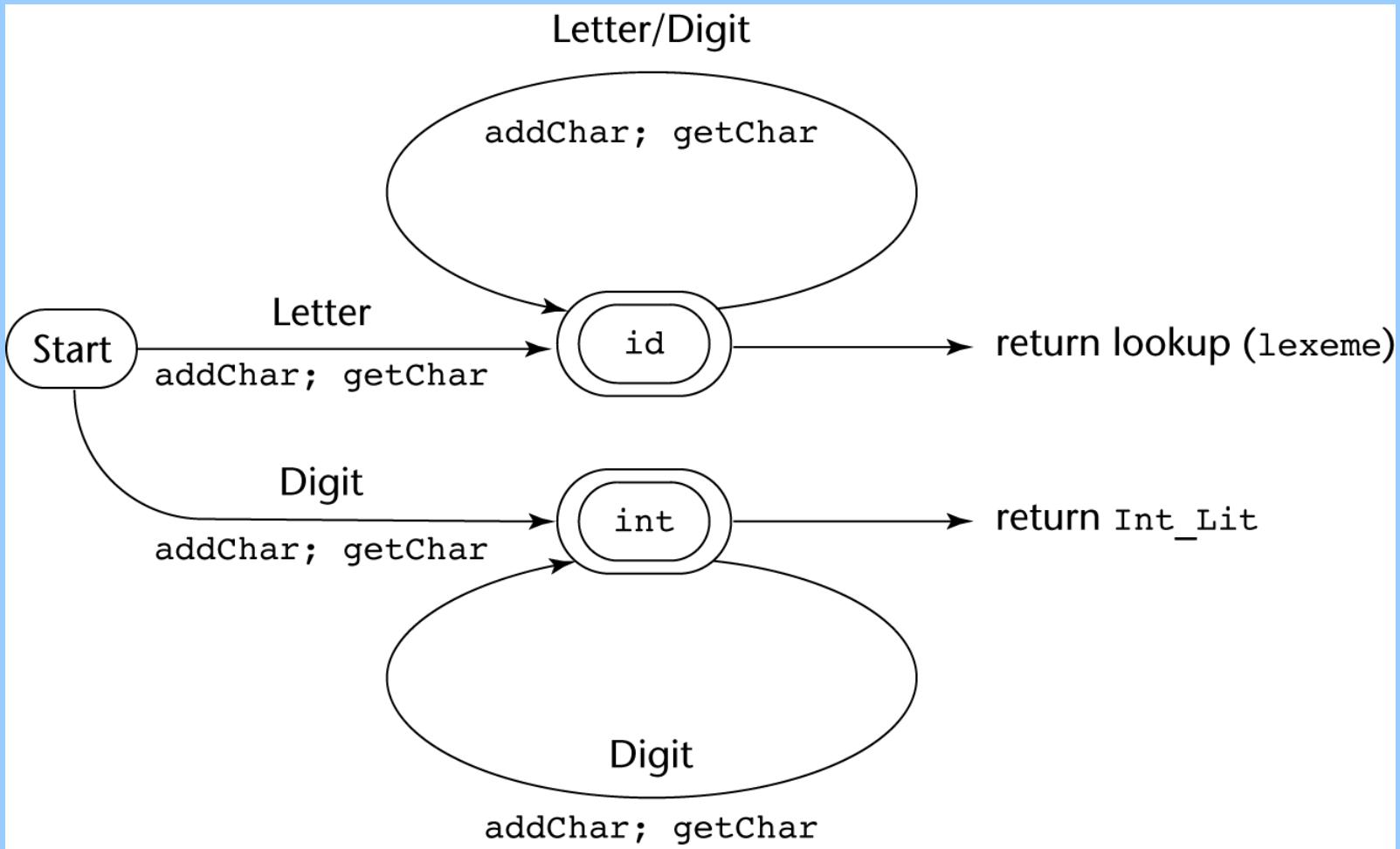
4.2 Sözcüksel(Lexical) Analiz (Devamı)

- Özgül sözcükler(reserved words) ve tanıtıcılar(identifiers) birlikte tanımlanabilir (her bir özgül sözcük(reserved word) için programın bir parçasını almak yerine)
 - Olası bir tanıtıcının(identifier) aslında özgül sözcük(reserved word) olup olmadığına karar vermek için tabloya başvurma(table lookup) kullanılır

4.2 Sözcüksel(Lexical) Analiz (Devamı)

- Kullanışlı yardımcı altprogramlar (utility subprograms):
 - **getChar** – girdinin(input) sonraki karakterini alır, bunu **nextChar** içine koyar, sınıfını (class) belirler ve sınıfı(class) **charClass** içine koyar
 - **addChar** – **nextChar** dan gelen karakteri **lexemenin** biriktirildiği yere koyar, **lexeme**
 - arama(lookup) – **lexeme** deki stringin özgül sözcük(reserved word) olup olmadığını belirler (bir kod döndürür)

Durum Diyagramı(State Diagram)



4.2 Sözcüksel(Lexical) Analiz (Devamı)

implementasyon (başlatma(initialization) varsayılmı):

```
int lex() {  
    getChar();  
    switch (charClass) {  
        case LETTER:  
            addChar();  
            getChar();  
            while (charClass == LETTER || charClass == DIGIT)  
            {  
                addChar();  
                getChar();  
            }  
            return lookup(lexeme);  
        break;  
  
        ...  
    }  
}
```

4.2 Sözcüksel(Lexical) Analiz(Devamı)

```
...
case DIGIT:
    addChar();
    getChar();
    while (charClass == DIGIT) {
        addChar();
        getChar();
    }
    return INT_LIT;
    break;
} /* switch'in sonu */
} /* lex fonksiyonunun sonu */
```

4.3 Ayrıştırma(Parsing) Problemi

- Ayrıştırıcıının amaçları, bir girdi(input) programı verildiğinde :
 - Bütün sentaks hatalarını(syntax errors) bulur; her birisi için, uygun bir tanılayıcı(diagnostic) mesaj üretir, ve hemen eski haline döndürür (recover)
 - Ayrıştırma ağacını(parse tree) üretir, veya en azından program için ayrıştırma ağacının izini(dökümünü)(trace) üretir

4.3 Ayrıştırma(Parsing) Problemi (Devamı)

- Ayrıştırıcıların(parser) iki kategorisi:
 - Aşağıdan-yukarıya(Top down) – ayrıştırma ağacını(parse tree) kökten(root) başlayarak oluşturur
 - Sıra, ensol türevindir (leftmost derivation)
 - Ayrıştırma ağacını(parse tree) preorderda izler veya oluşturur
 - Yukarıdan-aşağıya(Bottom up) – ayrıştırma ağacını(parse tree), yapraklardan(leaves) başlayarak oluşturur
 - Sıra, ensağ türevin (rightmost derivation) tersidir
- Ayrıştırıcılar(parser) girdide(input) sadece bir jeton(token) ileriye bakar

4.3 Ayrıştırma(Parsing) Problemi (Devamı)

- Yukarıdan-aşağıya ayrıstırıcılar(Top-down parsers)
 - Bir $xA\alpha$ sağ cümlesel formu (right sentential form) verildiğinde , ayrıstırıcı(parser), sadece A nin ürettiği ilk jetonu(token) kullanarak, ensol türevdeki(leftmost derivation) sonraki cümlesel formu(sentential form) elde etmek için doğru olan A-kuralını(A-rule) seçmelidir
- En yaygın yukarıdan-aşağıya ayrıştırma (top-down parsing) algoritmaları:
 - Özyineli azalan(recursive-descent)- kodlanmış bir implementasyon
 - LL ayrıstırıcılar(parser) – tablo sürümlü(table driven)implementasyon

4.3 Ayrıştırma(Parsing) Problemi (Devamı)

- Aşağıdan-yukarıya ayrıstırıcılar(bottom-up parsers)
 - Bir α sağ cümlesel formu(right sentential form) verildiğinde, α nın sağ türevde önceki cümlesel formu(sentential form) üretmesi için azaltılması gerekli olan, gramerde kuralın sağ tarafınd(right-hand side) olan altstringinin(substring) ne olduğuna karar verir
(determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation)
 - En yaygın aşağıdan-yukarıya ayrıştırma(bottom-up parsing) algoritmaları LR ailesindedir

4.3 Ayrıştırma(Parsing) Problemi (Devamı)

- Ayrıştırmanın Karmaşıklığı(Complexity of Parsing)
 - Herhangi bir belirsiz-olmayan gramer(unambiguous grammar) için çalışan ayrıştırıcılar(parsers) karmaşık(complex) ve belirsizdir(inefficient) ($O(n^3)$, n girdinin(input) uzunluğu(length) olmak üzere)
 - Derleyiciler(compilers), sadece bütün belirsiz-olmayan gramerlerin(unambiguous grammars) bir altkümesi(subset) için çalışan ayrıştırıcıları(parser) kullanır, fakat bunu lineer sürede(linear time) yapar ($O(n)$, n girdinin(input) uzunluğu(length) olmak üzere)

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)

- Özyineli-azalan işlem(Recursive-descent Process)
 - Gramerde(grammar) her bir nonterminal için o nonterminal tarafından üretilen cümleleri(sentences) ayırtırabilen(parse) bir altprogram(subprogram) vardır
 - EBNF, özyineli-azalan ayrıştırıcıya (recursive-descent parser) temel oluşturmak için idealdir, çünkü EBNF nonterminal sayısını minimize eder

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing) (Devamı)

- Basit deyimler(expressions) için bir gramer(grammar) :

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

4.4 Özyineli–azalan Ayrıştırma (Recursive–Descent Parsing) (Devamı)

- **Lex** isimli, sonraki jeton(token) kodunu **nextToken** içine koyan bir anlamsal analizci(lexical analyzer) olduğunu varsayıalım
- Sadece bir sağdaki kısım(RHS) olduğunda kodlama işlemi:
 - Sağdaki kısında(RHS) olan her bir terminal sembol(symbol) için, onu bir sonraki girdi jetonuyla(token) karşılaştır; eğer eşleşiyorsa, devam et, değilse hata(error) vardır
 - Sağdaki kısında(RHS) her bir nonterminal sembol(symbol) için, onunla ilgili ayırtıcı alt programını(parsing subprogram) çağırır

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

```
/* expr fonksiyonu
   Dilde kural(rule) tarafından üretilen
   stringleri ayristirir(parses) :
   <expr> → <term> { (+ | -) <term>}
*/
void expr() {
    /* İlk terimi(first term) ayristir(parse) */
    term();
    ...
}
```

4.4 Özyineli-azalan Ayrıştırma(Recursive-Descent Parsing)(Devamı)

```
/* Sonraki jeton(token) + veya - olduğu sürece, sonraki
   jetonu(token) almak için lex'i çağır, ve sonraki
   terimi(next term) ayırtır(parse)
*/
while (nextToken == PLUS_CODE ||
       nextToken == MINUS_CODE) {
    lex();
    term();
}
• Bu özel rutin hataları(errors) bulmaz
• Kural: Her ayrıştırma rutini(parsing routine) sonraki
  jetonu(next token) nextToken 'da bırakır
```

4.4 Özyineli–azalan Ayrıştırma (Recursive– Descent Parsing)(Devamı)

- Birden fazla sağdaki kısmı(RHS) olan bir nonterminal, hangi sağdaki kısmı(RHS) ayrıştıracağına(parse) karar vermek için bir başlangıç işlemeye(initial process) gerek duyar
 - Doğru sağdaki kısım(RHS), girdinin(input) sonraki jetonunu(token) temel alarak seçilir (lookahead)
 - Bir eşlenik bulana kadar sonraki jeton(next token) her bir sağdaki kısım(RHS) tarafından üretilen ilk jetonla(first token) karşılaştırılır
 - Eğer eşlenik bulunmazsa, bu bir sentaks hatasıdır (syntax error)

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

```
/* factor fonksiyonu dilde şu kuralın(rule)
   ürettiği stringleri ayırtırır(parse) :
   <factor> -> id | (<expr>) */  
  
void factor() {  
  
    /* Hangi RHS oldugunu belirle*/  
  
    if (nextToken) == ID_CODE)  
  
        /* RHS id si için, lex 'i çağır*/  
  
        lex();
```

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

```
/* Eğer sağdaki kısım(RHS) (<expr>) ise - sol
parantezi ihmal ederek lex 'i çağır, expr 'yi
çağır, ve sağ parantezi kontrol et */

else if (nextToken == LEFT_PAREN_CODE) {
    lex();
    expr();
    if (nextToken == RIGHT_PAREN_CODE)
        lex();
    else
        error();
} /* End of else if (nextToken == ... */

else error(); /* Hiçbir RHS eşleşmedi*/
}
```

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

- LL Gramer Sınıfı (LL Grammar Class)
 - Sol Özyineleme(Left Recursion) Problemi
 - Eğer bir gramerin(grammar) sol özyinelemesi(left recursion) varsa , doğrudan(direct) veya dolaylı(indirect), yukarıdan-aşağıya(Top-down) ayrıştırıcının(parser) temeli olamaz
 - Bir gramer(grammar) sol özyinelemeyi(left recursion) yoketmek için değiştirilebilir

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

- Yukarıdan-aşağıya ayrıştırmaya (top-down parsing) izin vermeyen gramerlerin diğer bir özelliği pairwise disjointness (**çiftli ayrıklık**) eksikliğidir
 - Doğru olan sağ kısmı (RHS) lookaheadın bir jetonuna (token) dayanarak belirleyememesi
 - Def: $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$
(Eğer $\alpha \Rightarrow^* \varepsilon$ ise, $\varepsilon \in \text{FIRST}(\alpha)$ içindedir)

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

- **Pairwise Disjointness Testi:**
 - Her bir nonterminal A için, birden fazla sağ kısmı(RHS) olan gramerde(grammar), her bir kural(rule) çifti $A \rightarrow \alpha_i$ ve $A \rightarrow \alpha_j$ için, şu doğru olmalıdır:

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$$

- Örnekler:

$$A \rightarrow a \quad | \quad bB \quad | \quad cAb$$

$$A \rightarrow a \quad | \quad aB$$

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

- Sol çarpan alma(Left factoring) problemi çözebilir
Şu ifadeyi:

$\langle \text{variable} \rangle \rightarrow \text{identifier} \mid \text{identifier } [\langle \text{expression} \rangle]$
aşağıdakilerden biriyle değiştirin:

$\langle \text{variable} \rangle \rightarrow \text{identifier } \langle \text{new} \rangle$

$\langle \text{new} \rangle \rightarrow \varepsilon \mid [\langle \text{expression} \rangle]$

veya

$\langle \text{variable} \rangle \rightarrow \text{identifier } [[\langle \text{expression} \rangle]]$

(dıştaki köşeli parantezler EBNF ‘nin
metasembolleridir(metasymbols))

4.5 Aşağıdan–Yukarıya Ayrıştırma (Bottom-up Parsing)

- Ayrıştırma(parsing) problemi bir sağ-cümlesel formda(right-sentential form) türevde önceki sağ-cümlesel(right-sentential) formu elde etmek için azaltılacak doğru sağ kısmı(RHS) bulmaktır

4.5 Aşağıdan–Yukarıya Ayrıştırma (Bottom-up Parsing) (Devamı)

- İşleyiciler(tanıtıcı değer)(handles) hakkında:
 - Tanım: β , sağ cümlesel formun(right sentential form) işleyicisidir(tanıtıcı değeridir)(handle)
 $\gamma = \alpha\beta w$ ancak ve ancak (if and only if)
 $S \Rightarrow^* rm \alpha Aw \Rightarrow rm \alpha\beta w$
 - Tanım: β , sağ cümlesel formun(right sentential form) tümceciğidir(phrase)
 γ ancak ve ancak(if and only if)
 $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
 - Tanım: β , sağ cümlesel form(right sentential form) γ nin basit tümceciğidir (simple phrase)
ancak ve ancak(if and only if)
 $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$

4.5 Aşağıdan–Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

- İşleyiciler(**tanıtıcı değer**)(handles) hakkında :
 - Bir sağ cümlesel formun(right sentential form) işleyicisi(handle) onun en soldaki(leftmost) basit tümceciğidir(simple phrase)
 - Verilen bir ayrıştırma ağacında(parse tree), şimdi işleyiciyi(handle) bulmak kolaydır
 - Ayrıştırma(parsing) , işleyici budama(handle pruning) olarak düşünülebilir

4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom-up Parsing)(Devamı)

- **Kaydırma–İndirgeme Algoritmaları**(Shift–Reduce Algorithms)
 - İndirgeme(Reduce), ayırtırma yiğininin(parse stack) üstündeki işleyici(handle) ile ona ilişkin LHS nin yerini değiştirmeye işlemidir
 - Kaydırma(Shift), bir sonraki jetonu(next token) ayırtırma yiğininin(parse stack) üzerine koyma işlemidir

4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom-up Parsing)(Devamı)

- LR ayırtıcılarının(LR parsers) avantajları:
 - Programlama dillerini tanımlayan gramerlerin hemen hemen tümü için çalışır.
 - Diğer aşağıdan–yukarıya algoritmaların (bottom-up algorithms) daha geniş bir gramerler sınıfı için çalışır, fakat diğer aşağıdan–yukarıya ayırtıcılarının(bottom-up parser) herhangi biri kadar da verimlidir
 - Mümkün olan en kısa zamanda sentaks hatalarını(syntax errors) saptayabilir
 - LR gramerler sınıfı(LR class of grammars), LL ayırtıcıları (LL parsers) tarafından ayırtılabilen sınıfının(class) üstkümesidir(superset)

4.5 Aşağıdan–Yukarıya Ayrıştırma (Bottom-up Parsing) (Devamı)

- LR ayrıştırıcıları(parsers) bir araç(tool) ile oluşturulmalıdır
- Knuth'un görüşü(Knuth's insight): Bir aşağıdan–yukarıya ayrıştırıcı(bottom-up parser), ayrıştırma(parsing) kararları almak için, ayrıştırmanın o ana kadar olan bütün geçmişini(history) kullanabilirdi
 - Sonlu ve nispeten az sayıda farklı ayrıştırma durumu oluşabilirdi, bu yüzden geçmiş(history) ayrıştırma yiğini(parse stack) üzerinde bir ayrıştırıcı durumunda(parser state) saklanabilirdi

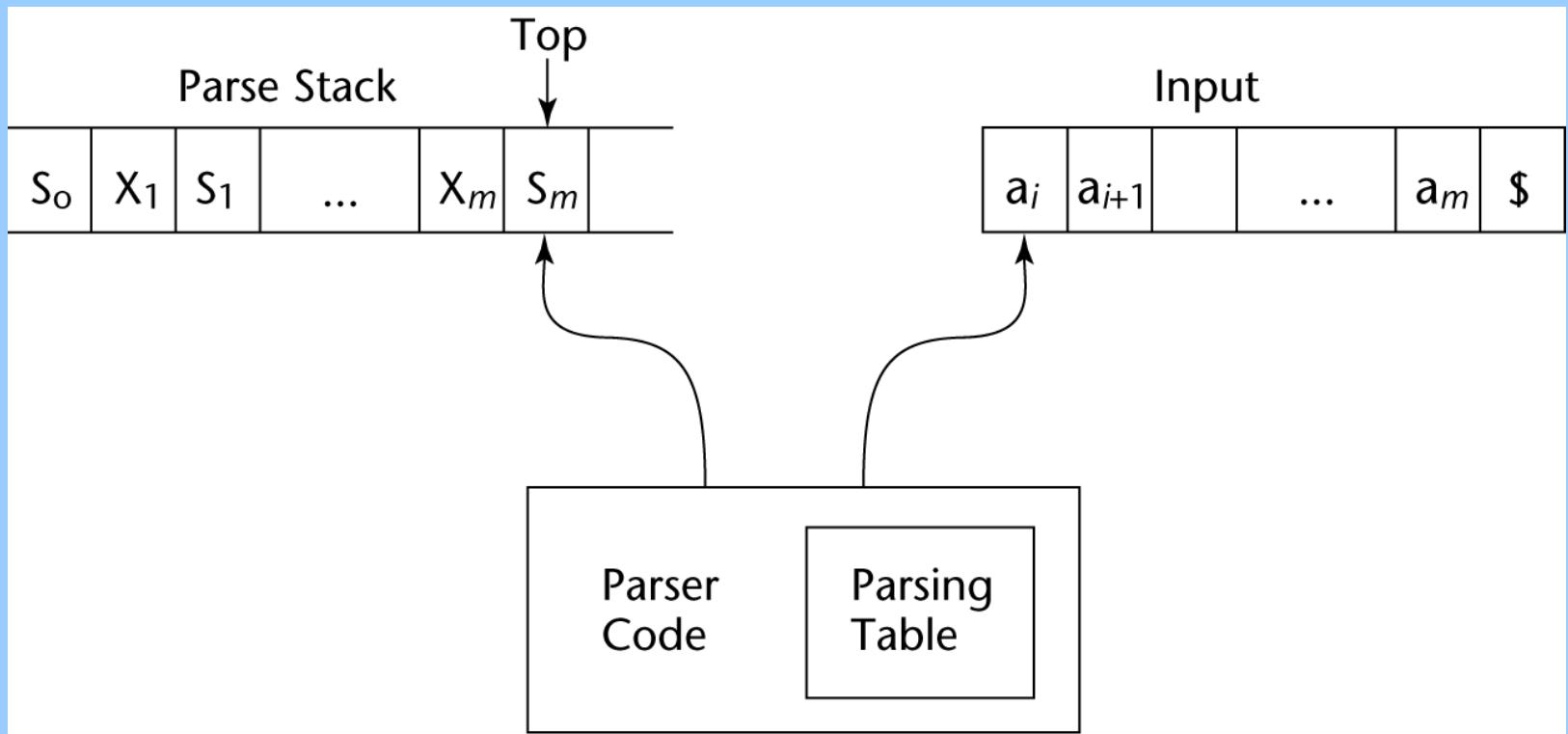
4.5 Aşağıdan–Yukarıya Ayrıştırma (Bottom-up Parsing) (Devamı)

- Bir LR yapılandırması(configuration) bir LR ayrıştırıcının(LR parser) durumunu(state) saklar
- $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_1 a_1 + 1 \dots a_n \$)$

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devamı)

- LR ayrıştırıcılar(LR parsers) tablo sürümlüdür(table driven), bu tablonun iki bileşeni vardır, bir ACTION tablosu ve bir GOTO tablosu
 - ACTION tablosu, verilen bir ayrıştırıcı durumu(parser state) ve sonraki jeton(next token) için, ayrıştırıcının(parser) hareketini(action) belirler
 - Satırlar(rows) durum(state) adlarıdır; sütunlar(columns) terminallerdir
 - The GOTO tablosu bir indirgeme hareketi(reduction action) yapıldıktan sonra ayrıştırma yiğini(parse stack) üzerine hangi durumun(state) konulacağını belirler
 - Satırlar(rows) durum(state) adlarıdır; sütunlar(columns) nonterminallerdir

Bir LR Ayrıştırıcısının(Parser) Yapısı



4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devamı)

- Başlangıç yapılandırması: $(S_0, a_1 \dots a_n \$)$
- Ayrıştırıcı hareketleri(parser actions):
 - If $\text{ACTION}[S_m, a_i] = \text{Shift } S$, sonraki yapılandırma:
 $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$
 - If $\text{ACTION}[S_m, a_i] = \text{Reduce } A \rightarrow \beta$ and $S = \text{GOTO}[S_{m-r}, A]$, $r = \beta$ nin uzunluğu olmak üzere , sonraki yapılandırma:
 $(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} A S, a_i a_{i+1} \dots a_n \$)$

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devamı)

- Ayrıştırıcı hareketleri(parser actions)
(devamı):
 - If ACTION[Sm, ai] = Accept, ayrıştırma(parse) tamamlanmıştır ve hata(error) bulunmamıştır
 - If ACTION[Sm, ai] = Error, ayrıştırıcı(ayrıştırıcı) bir hata-işleme rutini(error-handling routine) çağırır

LR Ayrıştırma Tablosu(Parsing Table)

State	Action							Goto		
	id	+	*	()	\$	E	T	F	
0	S5		S4				1	2	3	
1		S6				accept				
2		R2	S7		R2	R2				
3		R4	R4		R4	R4				
4	S5			S4			8	2	3	
5		R6	R6		R6	R6				
6	S5			S4				9	3	
7	S5			S4					10	
8		S6			S11					
9		R1	S7		R1	R1				
10		R3	R3		R3	R3				
11		R5	R5		R5	R5				

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devamı)

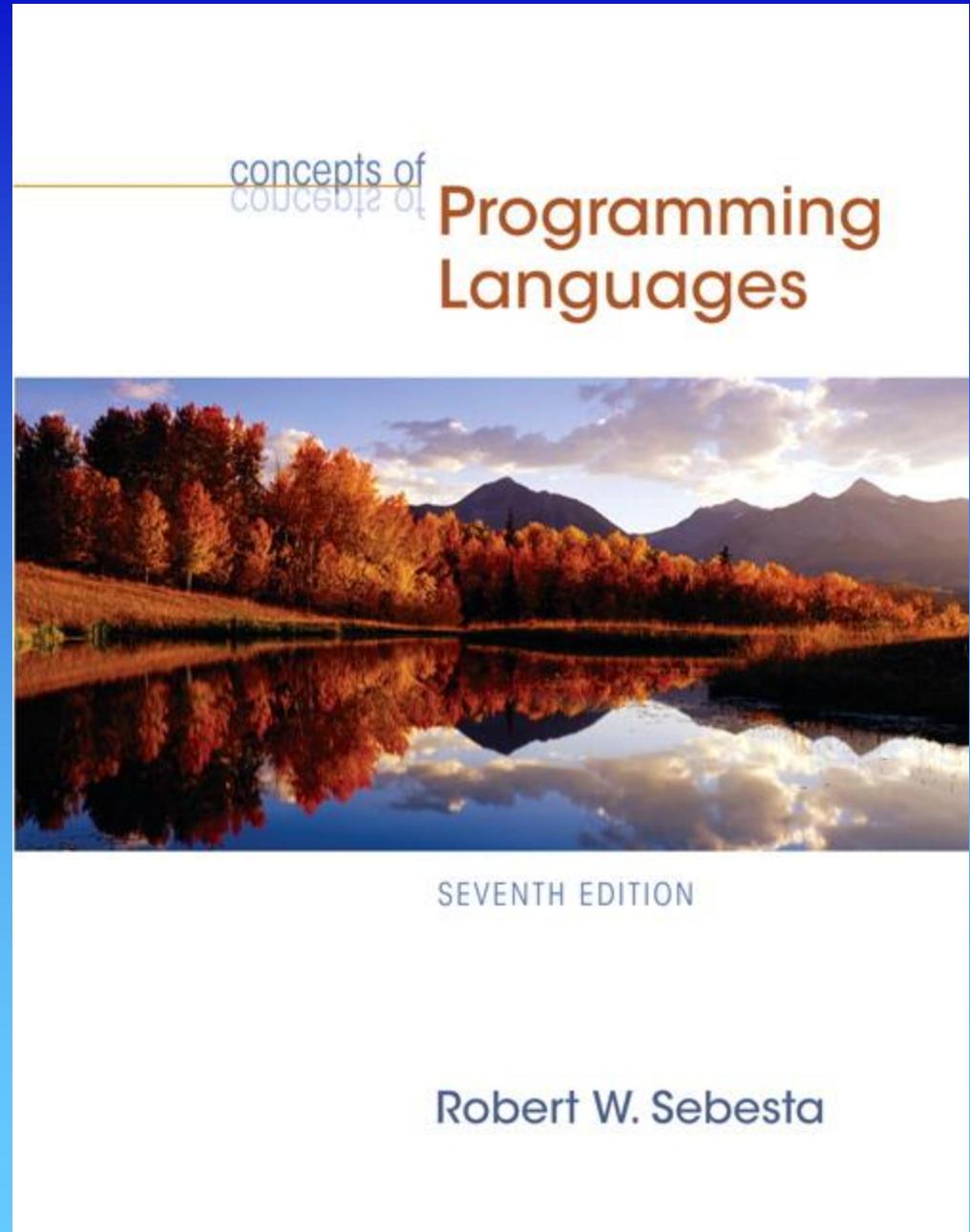
- Verilen bir gramerden(grammar) bir araç (tool) ile bir ayrıştırıcı tablosu(parser table) üretilebilir , örn., yacc

Özet

- Sentaks analizi(Syntax analysis) dil implementasyonunun ortak kısmıdır
- Bir sözcüksel analizci(lexical analyzer) bir programın küçük-ölçekli parçalarını ayıran bir desen eşleştiricidir(pattern matcher)
 - Sentaks hatalarını(syntax errors) saptar
 - Bir ayrıştırma ağacı(parse tree) üretir
- Bir özyineli-iniş ayrıştırıcı(recursive-descent parser) bir LL ayrıştırıcıdır(LL parser)
 - EBNF
- Aşağıdan-yukarıya ayrıştırıcıların(bottom-up parsers) ayrıştırma problemi: o anki cümleSEL formun altstringini(substring) bulma
- LR ailesi kaydırma-indirgeme ayrıştırıcıları(shift-reduce parsers) en yaygın olan aşağıdan-yukarıya ayrıştırma (bottom-up parsing) yaklaşımıdır

Bölüm 5

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



SEVENTH EDITION

Robert W. Sebesta

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 görününen 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 bağamlarda(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 farklı adreslere sahip olabilir
 - Eğer iki değişken adı(variable names) aynı bellek konumuna erişmek için kullanılabiliyorsa, bunlara takma ad(düger 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 parametreleri – 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 bağlanmanın(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 hücresına(memory cell) bağlama (veya bir C **static** değişkenine(variable))
 - Yürütme süresi(Runtime)--örn., bir statik olmayan(nonstatic) lokal değişkeni(local variable) bir bellek hücresına(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 bildirimleri(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 çıkarımı(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 hücresına(memory cell) bağlılığı olduğu sürece geçen zamanıdır

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 bağlı 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 ayrıılır(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 işlenenine(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) fazlasıyla etkiler—oldukça yavaşlatılabilirler (C++ ‘a karşı Ada)
- Java’nın C++’ın atama zorlamalarının(assignment coercions) yalnızca yarısına sahip olmasına rağmen, onun kesin tiplendirmesi(strong typing) Ada’nıñkinden 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:
 - Integer 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) kullanıiyorlarsa 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 sayı tipinin(enumeration types) bileşenlerinin yazılışları farklı ise bunlar uyumlu mudur?
 - Yapısal tip(structural type) uyumluluğuyla, aynı yapıya ait farklı tipleri ayırt edemezsınız (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) referanslarının(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;  
    ...  
}
```

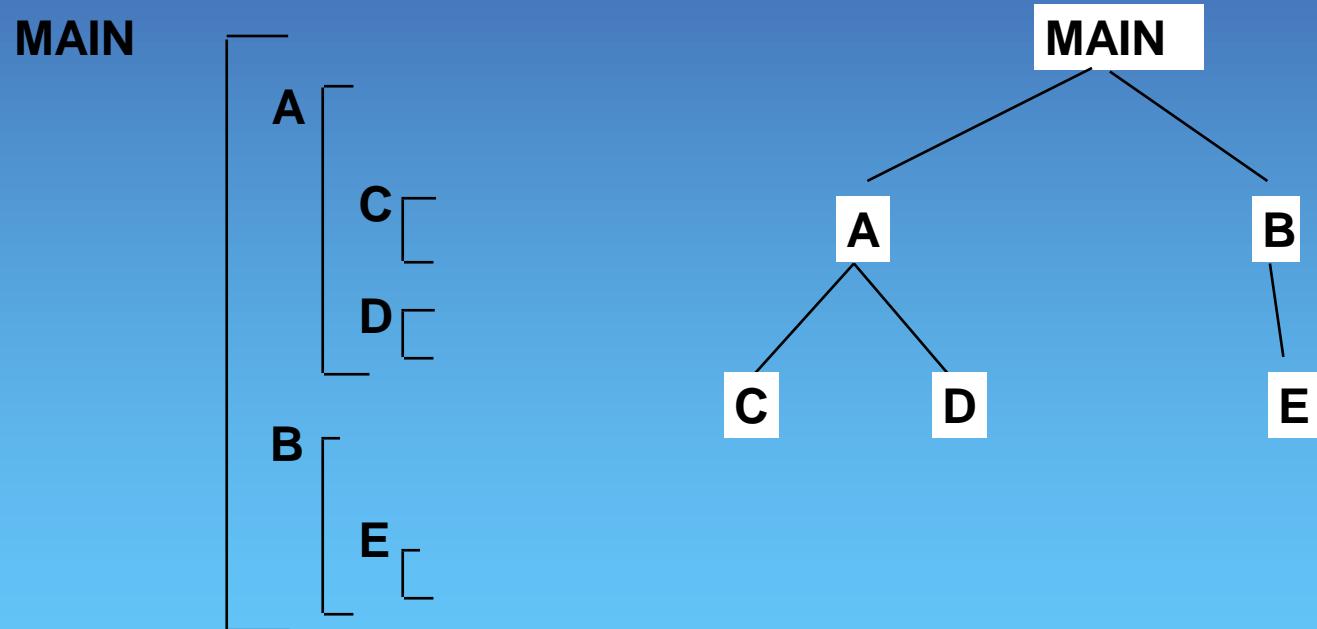
Ada: **declare** LCL : FLOAT;
begin
 ...
end

5.8 Kapsam(Scope) (devamı)

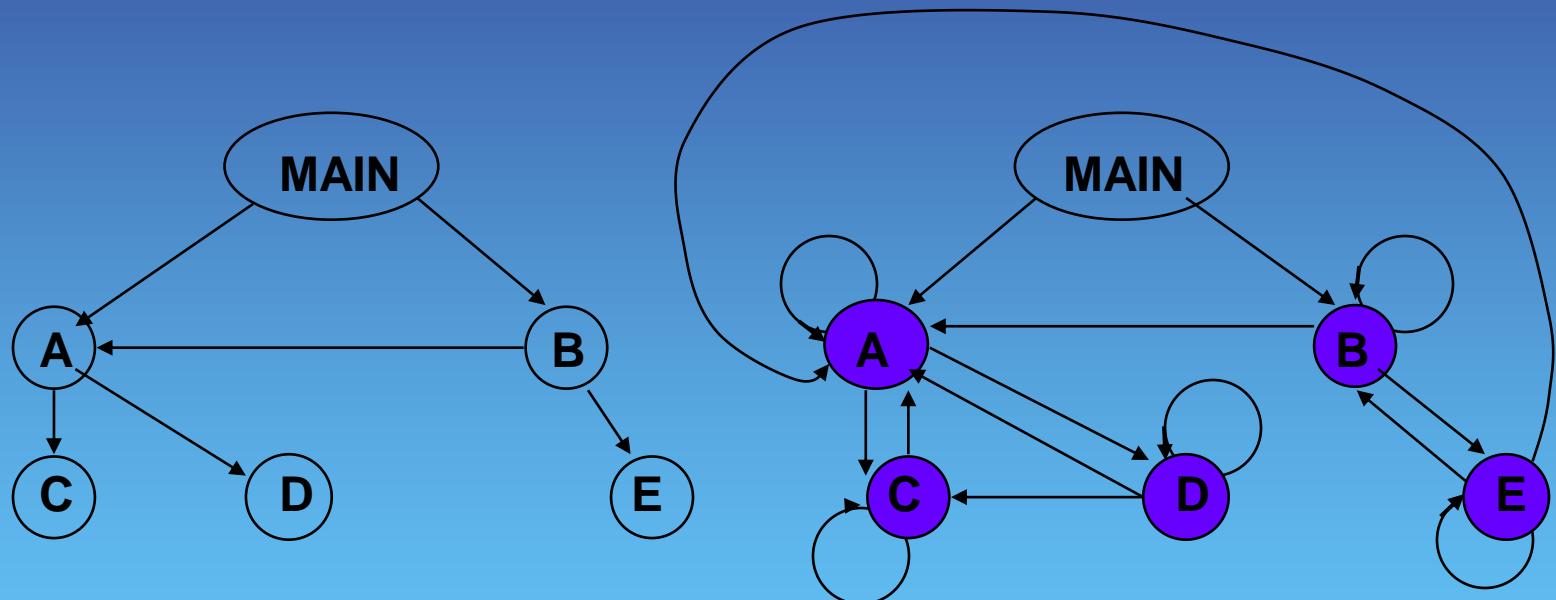
- Statik Kapsamanın(Static Scoping) değerlendirmesi
- Örneğe bakalım:

Varsayıyalım 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 varsayıalım öyle ki D, B deki bazı veriye erişmek zorunda
- Çözümler:
 - D'yi B'nin içine koy (fakat o zaman C artık onu çağrıramaz 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 çağrımasına 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

- declaration of x
SUB1
- declaration of x -
...

call SUB2

...

SUB2

...

- reference to x -

...

call SUB1

...

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 değerlendirilmesi:
 - Avantaj: elverişlilik
 - Dezavantaj: zayıf okunabilirlik(readability)

5.9 Kapsam(Scope) ve Ömür(Lifetime)

- Kapsam(Scope) ve Ömür(Lifetime) bazen yakından ilişkilidir, fakat farklı kavramlardır
- Bir C veya C++ fonksiyonundaki bir **static** değişkeni(variable) düşünelim

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 değişkenler(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

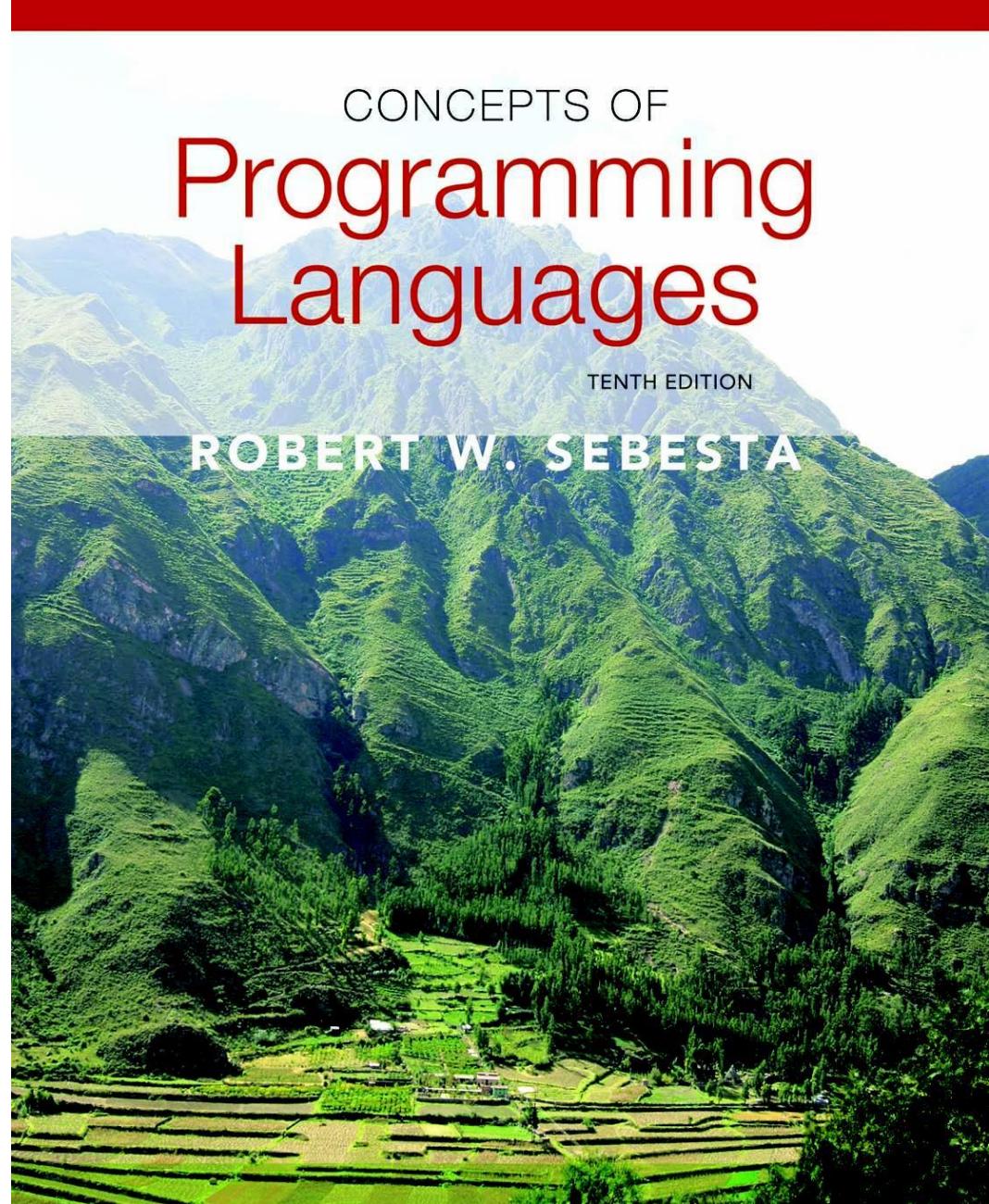
```
int sum = 0;
```

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), yiğin-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

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

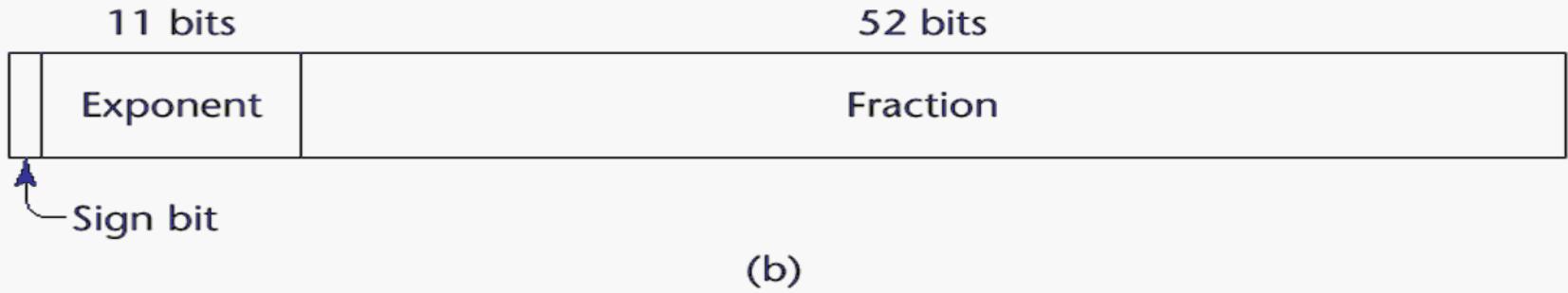
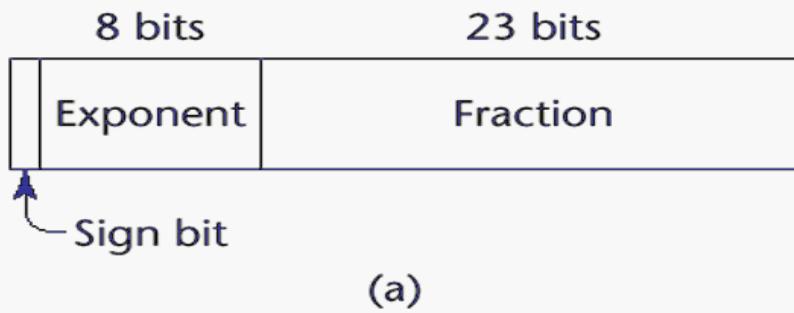
- Genellikle her zaman donanımın(hardware) tam yansımasıdır, bu yüzden eşlenme(mapping) önemsizdir.
- Bir dilde en çok sekiz farklı tamsayı (integer) tipi olabilir
- Java'nın integer tipi veri tipleri: **byte**, **short**, **int**, **long**

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 kısmı 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 temelliidir.
 - 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 olmalıdır?

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 (örbüntü 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

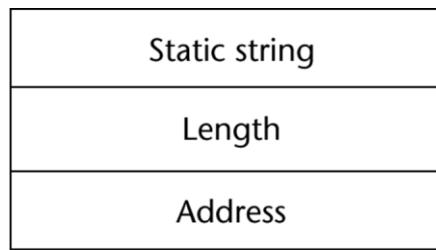
Yazılabilirliğe yardımcıdır

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

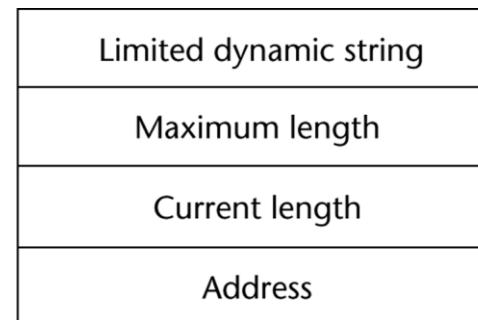
Karakter String Uygulaması

- **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



Statik stringler
için derleme-
süresi betimleyici



Sınırlı dinamik
stringler için
yürütmeye-süresi
betimleyici

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 alınmasına izin verilmeli midir? Eğer böyle ise o sabitin ortaya çıkışının 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 sıralanmılı ardışık alt dizi.
 - Örnek: 12..18 tamsayı tipinin alt aralığıdır.
- Ada'nın tasarıımı
 - 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 (Enumeration) 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 elemanlarının bir küməsidir, elemanlardan her biri kümedeki birinci eleməna görə olan pozisyonuyla tanımlanır.

Dizi Tasarım Problemleri

- 1. Altısimgeler(indeks)(subscripts) için hangi tipler legaldir?
- 2. Eleman referansları aralığındaki altısimge ifadeleri(subscripting expressions) kontrol edilmiş midir?
- 3. Altısimge aralıkları ne zaman bağlanır(**bound**)?
- 4. Ayırma (allocation) ne zaman olur?
- 5. Altısimgelerin(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) yapmaktadır
 $\text{map(array_name, index_value_list)} \rightarrow \text{an element}$
- 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(rrays) 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(rrays)
 - 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 büyütülebilir veya küçültülebilir.)
- APL'de, Perl, ve JavaScript, diziler(Arrays)
ihtiyaca göre büyüp 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.
- Python'un 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]]
```

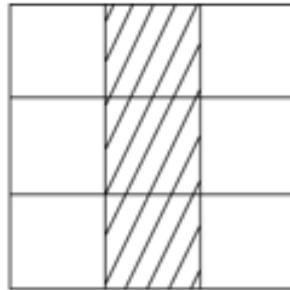
vector (3:6) dizinin 3 elemanını temsil eder.

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

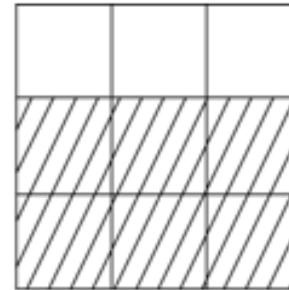
- **Ruby slice metot olarak kesiti destekler.**

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

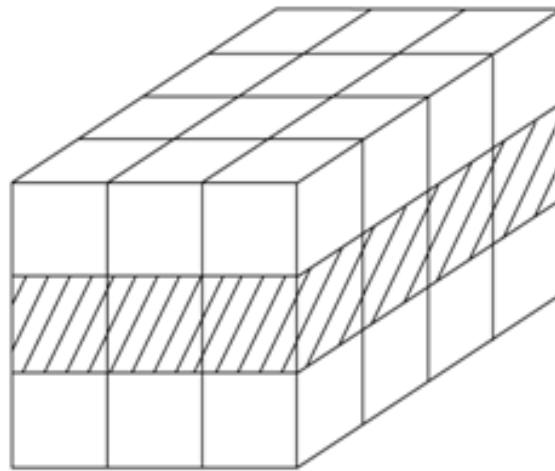
Kesitler



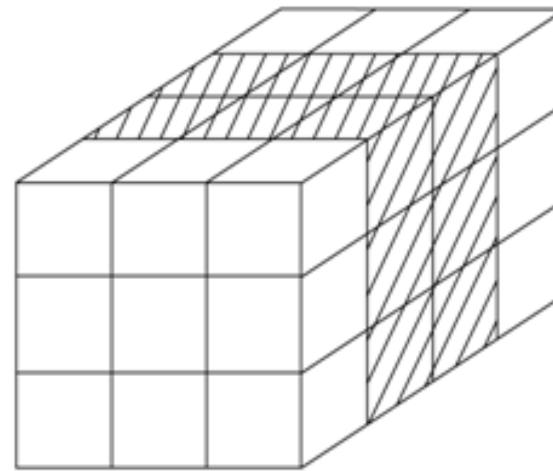
MAT $(1:3, 2)$



MAT $(2:3, 1:3)$



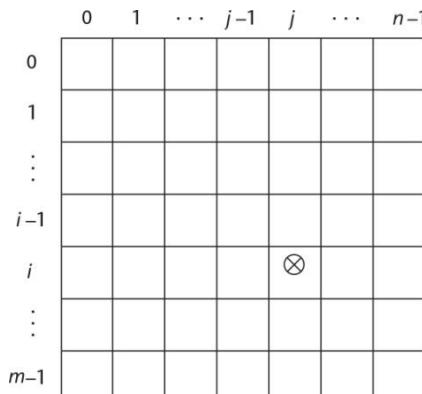
CUBE $(2, 1:3, 1:4)$



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

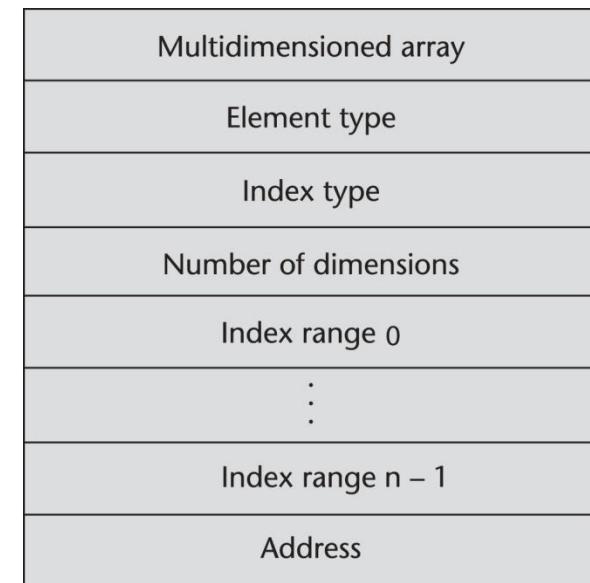
Dizi Uygulamaları

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



Çok Boyutlu Dizilere Erişim

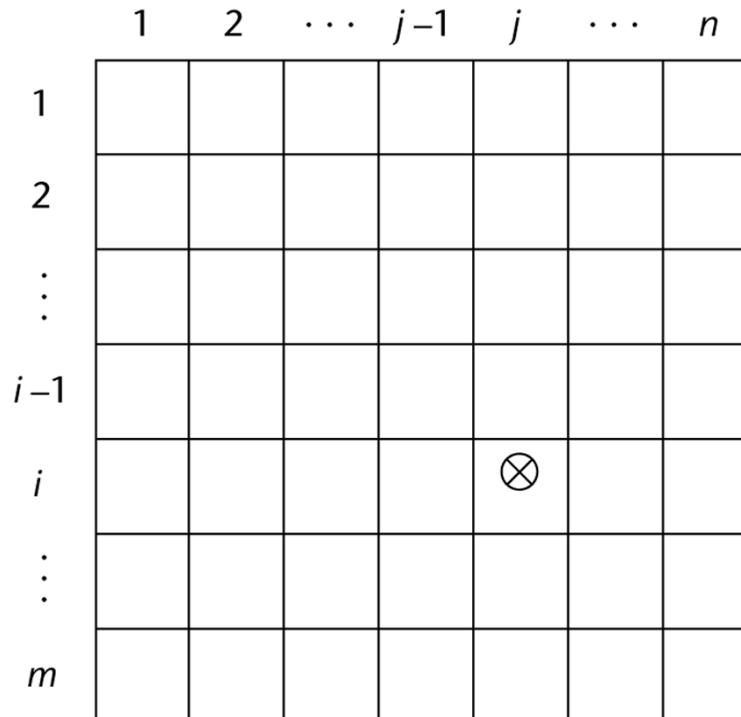
- Yaygın olarak kullanılam metotlar:
 - Satıra göre sıralama – bir çok dilde kullanılan metottur
 - Sütüna göre sıralama – Fortran tarafından kullanılır
 - Çok boyutlu dizilerin derleme süreleri yan taraftaki şekilde verilmiştir.



Çok Boyutlu Dizilerde Eleman Yerleştirme

- Genel format

Location ($a[l,j]$) = address of a [row_lb,col_lb] +
 $((l - \text{row_lb}) * n) + (j - \text{col_lb}) * \text{element_size}$



Derleme Süresi Betimleyiciler

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Tek Boyutlu Dizi

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
:
Index range n
Address

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 ortagonal 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

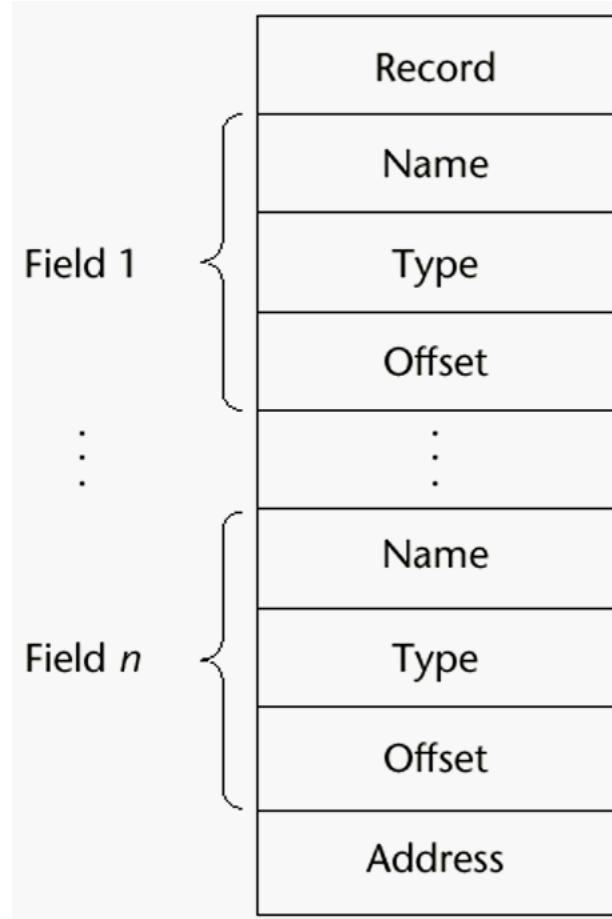
- COBOL'de - kaynak kayıttaki(source record) bütün alanları(fields) hedef kayıttaki(destination record) aynı ada sahip alanlara(fields) taşıır

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 kullanılabılırdı, fakat o zaman tip kontrolüne(type checking) izin vermeyecekti ve çok daha yavaş olacaktı

Kayıt Tipinin Uygulanması

Kayıtların başlangıcına göre
Offset 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 başlayarak referanslandırır.
 - + operatörünü kullanır ve del komutıyla 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 ayrıcıyla 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 ‘ işaretini 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 virgülerle 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 indisleri “0”dan başlar ve sonradan değiştirilebilir.
`x = myList[1] Sets x to 5.8`
 - Liste elemanları `del` komutuyla silinir.
`del myList[1]`
 - Liste anımları – 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 tasarım 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

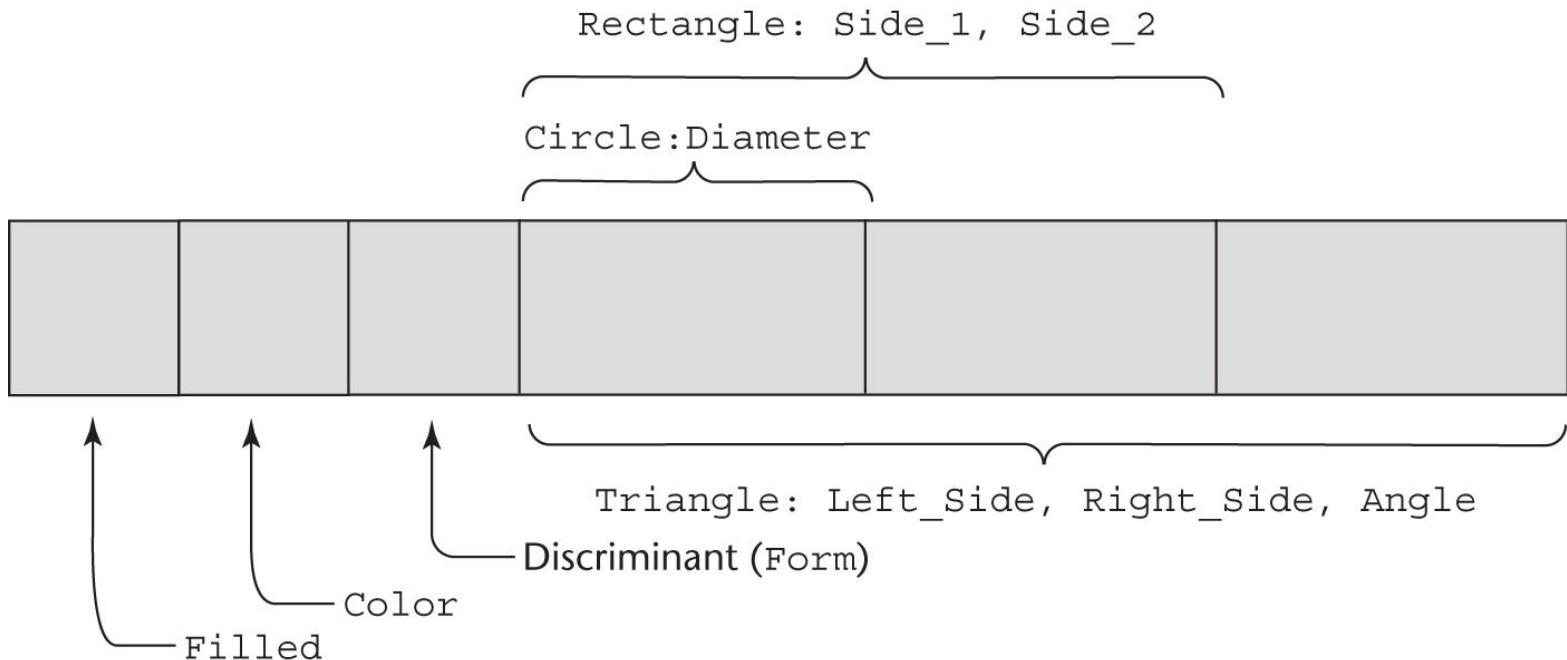
Java ‘da kayıtlar(Records) da bileşimler(unions)
de yoktur

- 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;
```

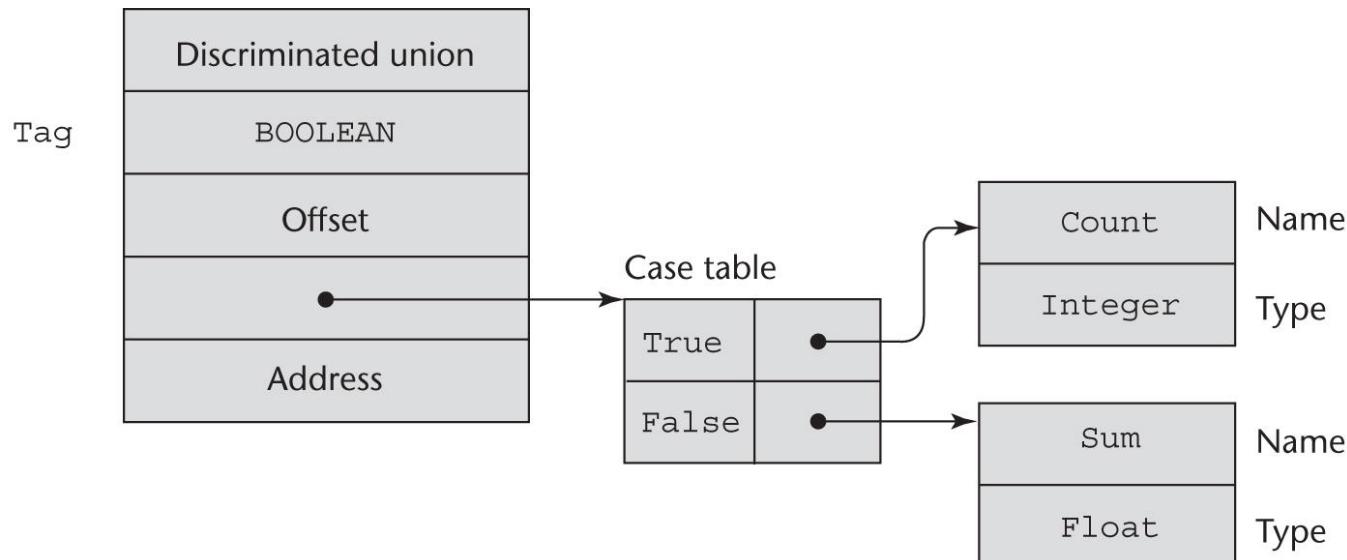
Ada Birleşim Tipi Örneği



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 ayrik 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)

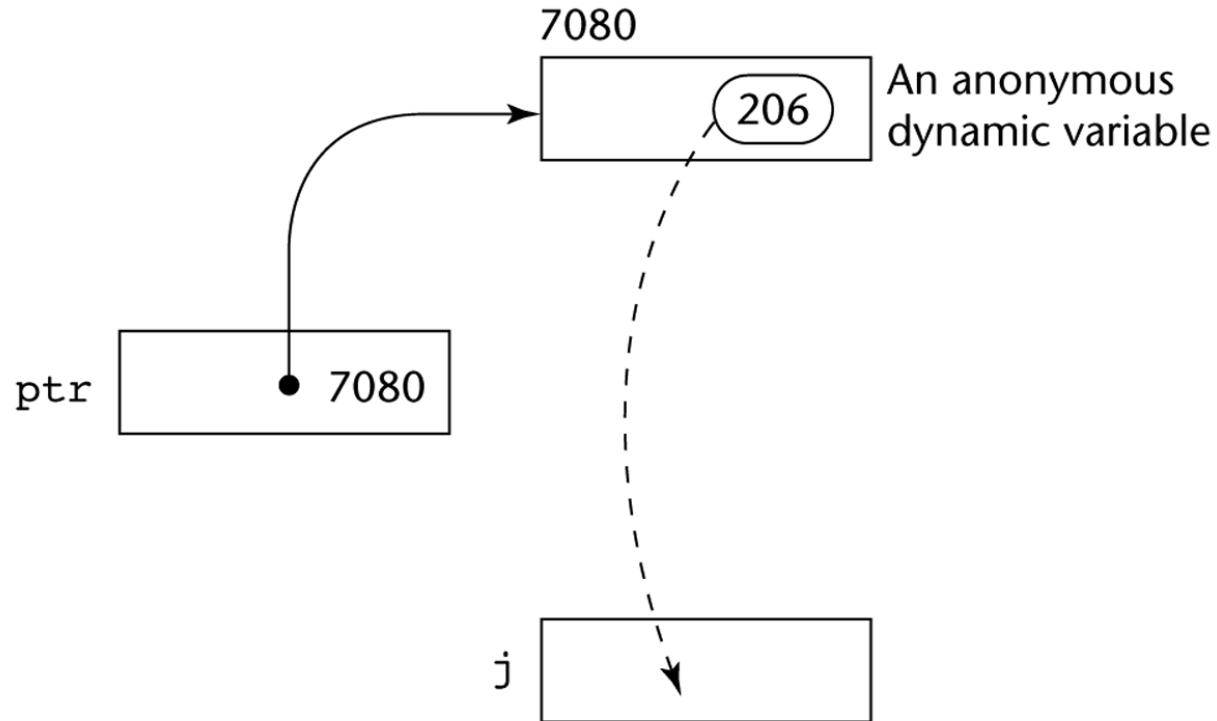
Pointerların Tasarım Problemleri

- Pointer'in kapsamı ve ömrü nedir ?
- Dinamik öbek değişkenlerinin ömrüğü nedir?
- İşaretçiler bu konuda ortaya koyabildikleri değer türü sınırlı mı?
- Pointerlar dinamik depolama için mi kullanılıyor 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 gösterdiği 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 pointerin değerine ikinci bir pointeri atama
 - c. Birinci pointeri 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 *** – herhangi bir tipe işaret edebilir ve tip kontrolü yapılabılır (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 tasarılayamayız

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 sayıları (*istikli 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

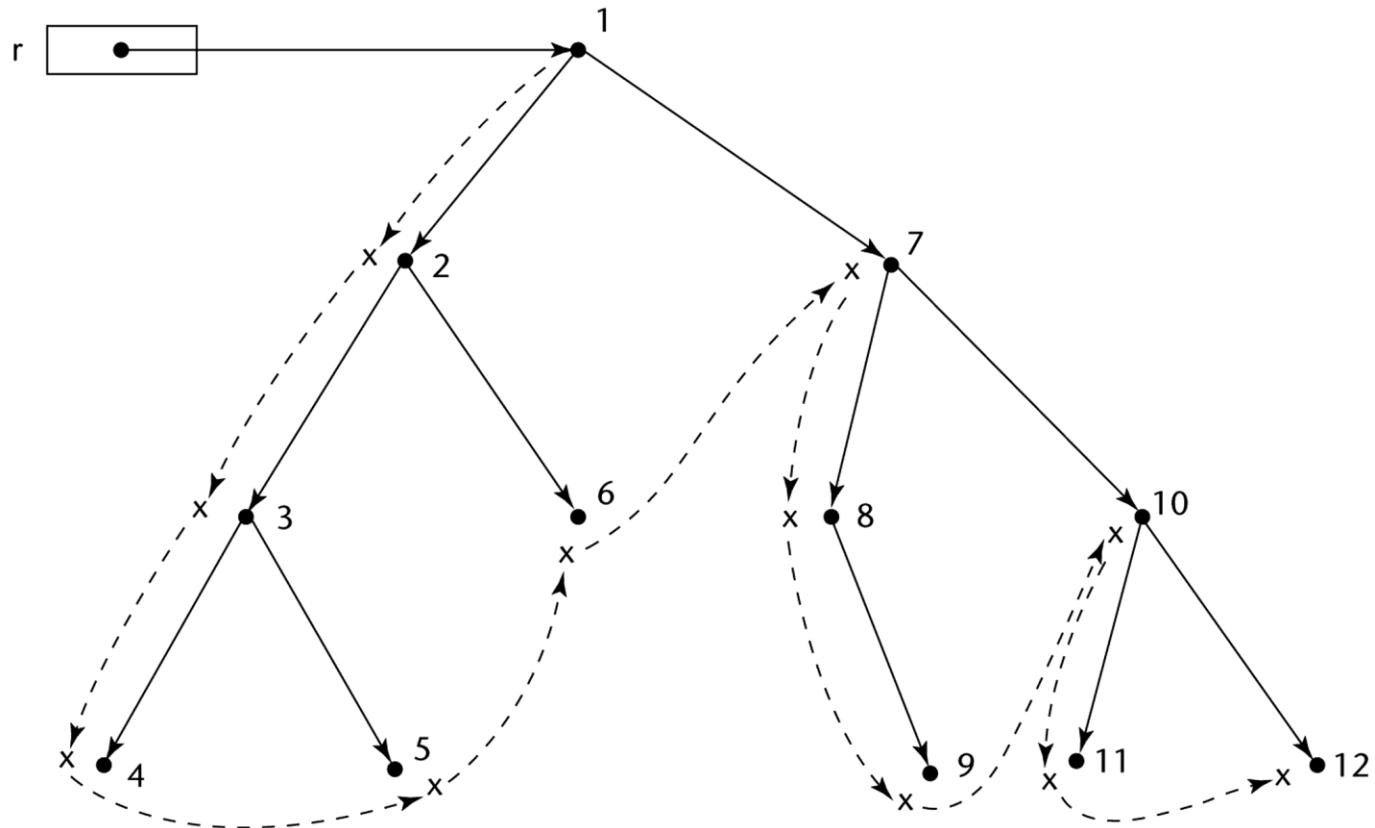
- Dezavantajlar: boş alan gereklidir, yürütme zamanı(execution time) gereklidir, dairesel olarak bağlanmış hücreler için komplikasyonlar.
- *Avantaj:* Uygulama yürütmedeki önemli gecikmeler engellenir

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 gereklidir.
- 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 yaygınlaştmak.
- *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 güçlündür.
- 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 deligidir.)
(Java and C# Ada'ya benzer)

Güçlü Tipler (devamı)

- Zorlama kuralları güçlü tiplerde güçlü efektler oluşturabilir veya onları zayıflatır. (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 hayatı 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)

- İki 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 çalışmalarını 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)

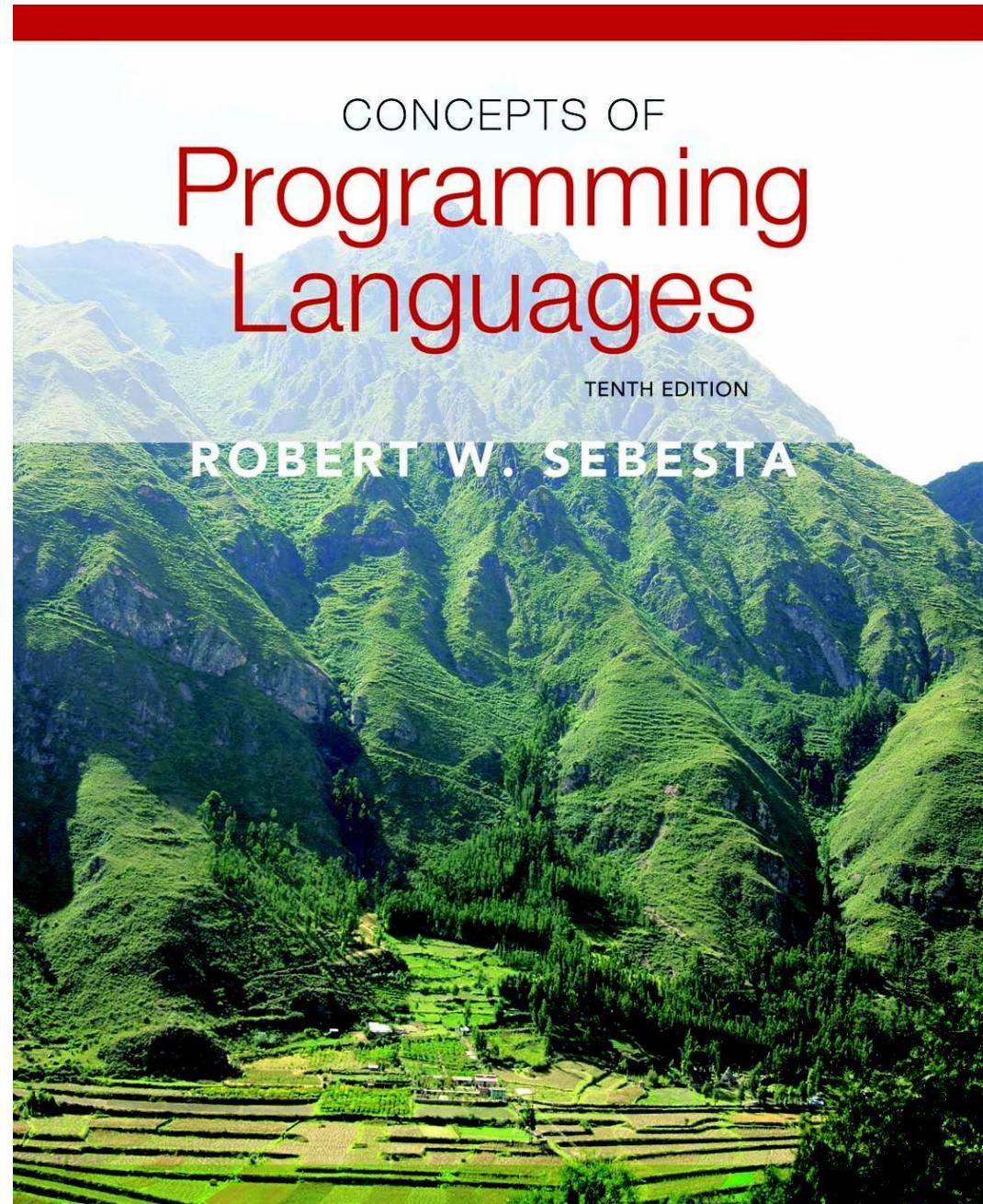
- Bir tip sisteminde biçimsel model tipleri kümesi ve tip kuralları tanımlayabilirsiniz.
 - Tipleri belirlemek için dilbilgisi kuralları veya haritalar kullanılır.
 - Sonlu haritalama – model diziler ve fonksiyonlar
 - Kartezyan ürünler – model demetler ve kayıtlar
 - Birleşimleri ayarlama – Model birleşim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

Summary

- Veri tipleri, bir dilin kullanışılığını 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 arttırmır.
- 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 gereklidir.
- 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 birleşimlilik 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) operatörünün n tane operantı vardır.

Aritmetik İfadeler: Operatör Öncelik Kuralları

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

Aritmetik İfadeler: Operatör Birleşimlilik Kuralı

- Bu kural aynı öncelik seviyesindeki bitişik operatörlerin işlenmesi sırasını belirtir.
- Tipik birleşimlilik kuralları
 - Soldan sağa, **(hariç, burada sağdan sola),
 - Zaman zaman tekli operatörlerin birleşimliliği sağdan sola olabilir (ör., FORTRAN)
- APL dili farklıdır; bu dilde tüm operatörler eşit önceliklere sahiptir ve operatörlerin birleşimliliği sağdan soladır.
- Öncelik ve birleşimlilik 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 metodlar 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 çağrıları 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 olmayacağı(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 verdiğiinden 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, olmayacağı, 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 çarşı, tekli olarak adresleme(pointer))
  - Derleyicinin hata belirlemesindeki kayıplar(derleyici operant eksiklerini fark etmeli)( $a*b$  yerine  $*b$  yazmak gibi)
  - Bazı okunabilirlik kayıpları(okunabilirliği zorlastırır)

# Aşırı Yüklenmiş Operatörler(devam)

---

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

# Tip Dönüşümleri

---

- **Daraltıcı dönüşüm:** Dönüşürülecek tip orijinal tipin tüm değerlerini içermiyorsa bu dönüşüme daraltıcı dönüşüm denir (örnek, `float → int`)
- **Genişletici dönüşüm:** Dönüşürülecek 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 dönüşümüdür.
- 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#'nın sentaksı fonksiyon çağrırmaya 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 sembollerini 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 Tespitı

---

- 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  $length - 1$  uzunlığında 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 yüklenliğinde 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 gereklidir.
- ALGOL'de tanımlı; C ve C-tabanlı diller de benimsemış.
  - 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 **arttırdı**, daha sonra sum' a **aktarıldı** )

sum = count++ (count sum' a **aktarıldı**, ondan **sonra arttırdı**)

count++ (count **arttırdı**)

-count++ (count **arttırdı** ondan sonra negatif **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 gibi 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 sınırlıdır İsimler  
**val** fruit = apples + oranges;
  - Eğer fruit 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.

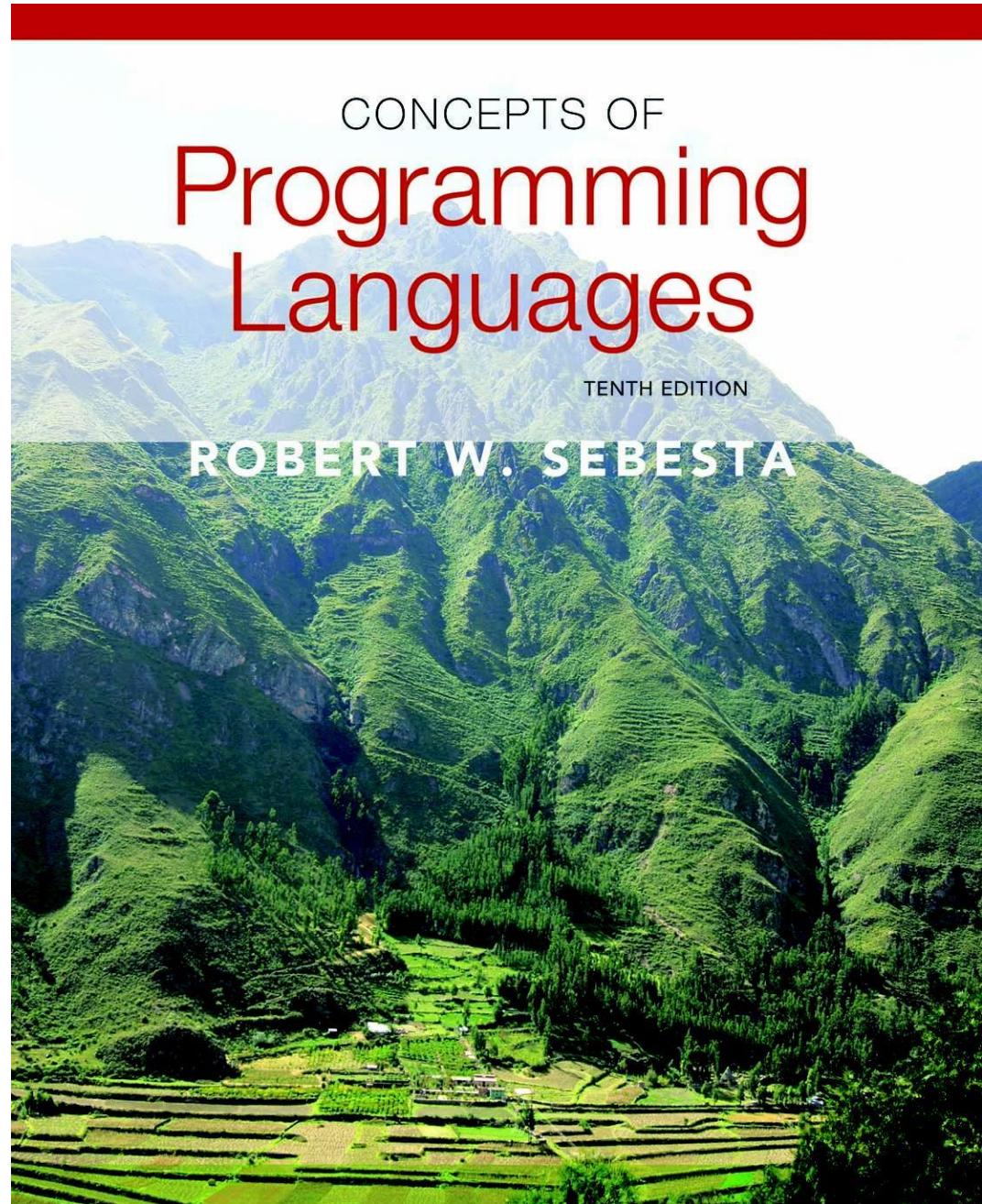
# Özet

---

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

# Bölüm 8

Komut Seviyeli  
Kontrol Yapıları



# Bölüm 8'in Başlıklarısı

---

- 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 sınıfaya 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 seçeneklerin anamları nasıl belirlenecek?

# Kontrol ifadeleri

---

- 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 tanıtmak 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 değerlendense 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_expr1: stmt1;
 ...
 case const_exprn: stmtn;
 [default: stmtn+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 gereklidir.
- ```
• switch (indeks) {  
•     case 1: goto case 3;  
•     case 3: tek +=1;  
•     toplamtek += indeks;  
•     break;  
•     case 2: goto case 4;  
•     case 4: cift += 1;  
•     toplamcift += indeks;  
•     break;  
•     default: Console.WriteLine("switch içinde hata, indeks = %d\n",  
•         indeks);  
• }
```

Çoklu Seçme Komutları: 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 seçeneklerde 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 gerölekştirmek.

case

when count < 10 **then** bag1 = **true**

when count < 100 **then** bag2 = **true**

when count < 1000 **then** bag3 = **true**

end

Çoklu Seçicilerin Şeması

- Genel çağrıma formu ŞART:

(ŞART

(YÜKLEM₁ İFADE₁)

...

(YÜKLEM_n İFADE_n)

[(ELSE expression_{n+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 değişimdir.
- 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 içerisinde ([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 istege 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 simülle 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 son test 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, yiğinlar 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 metodlarla beraber kullanılabilirler.
 - Önceden tanımlanmış döngü metodları (`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 indisleri 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**) kamuflه edilmiş goto'lardır.

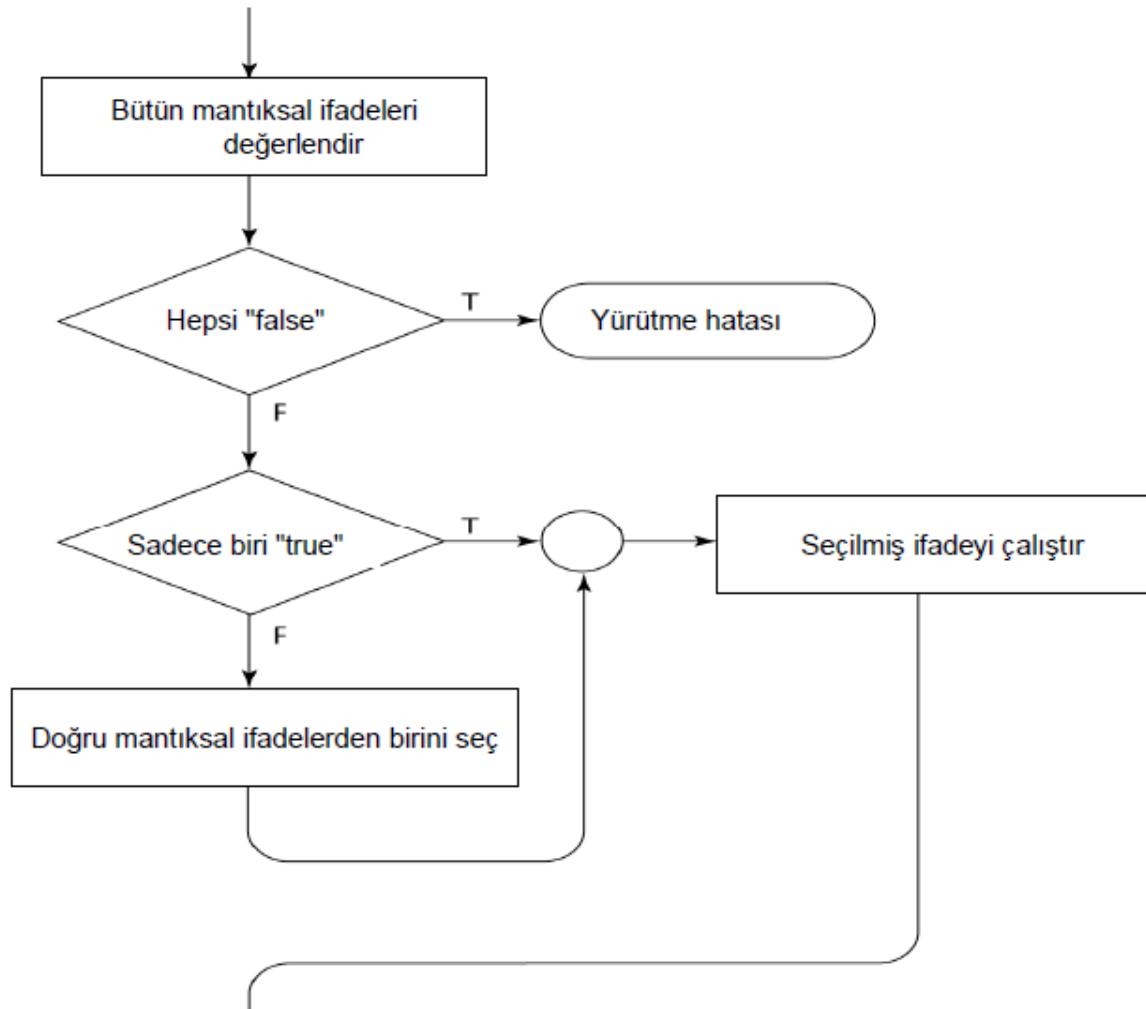
Güvenlikli Komutları

- 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 gereklidir.

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 çıkarılır.

Güvenlikli Komutlar: Gerekçe

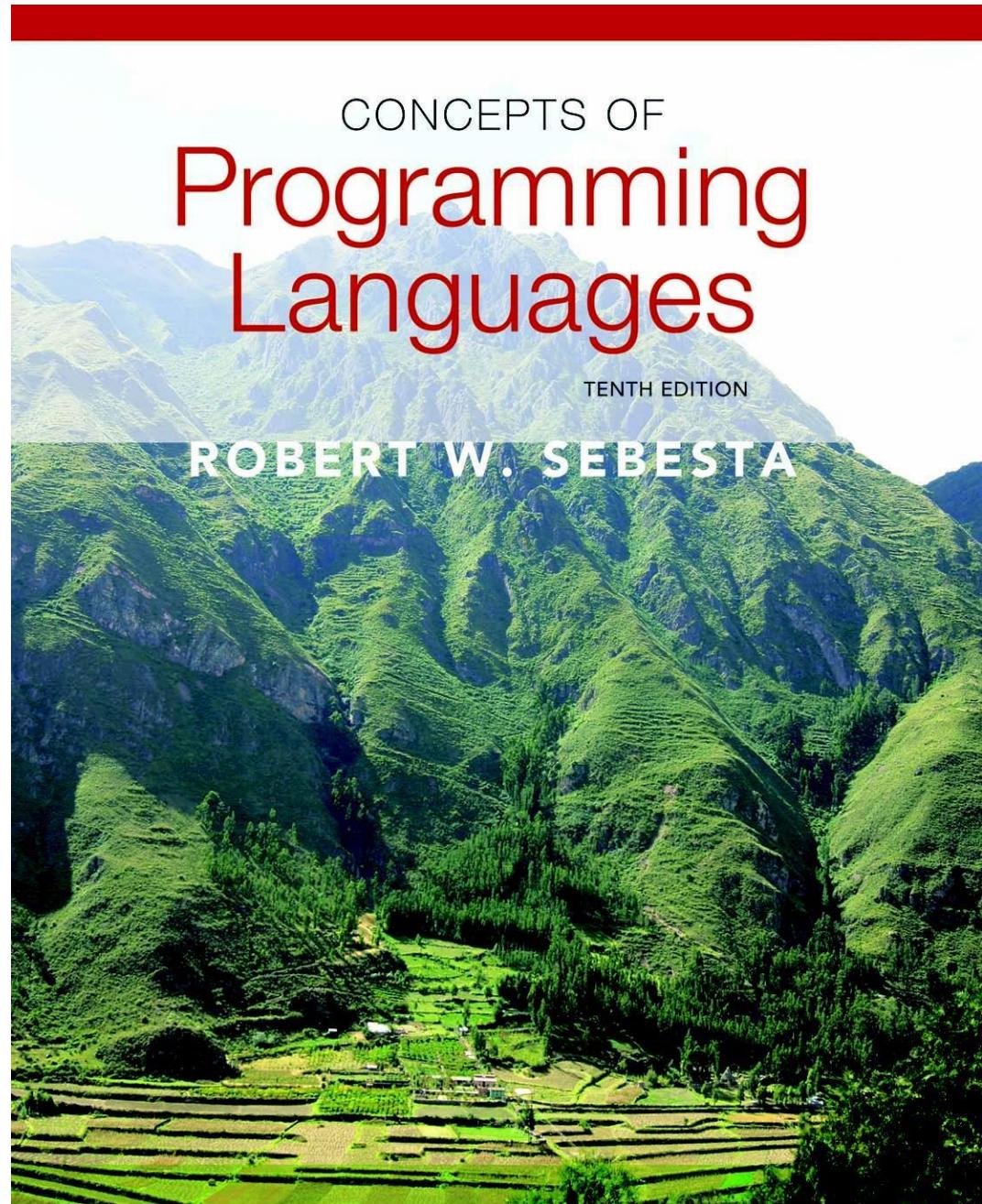
- Kontrol deyimleri ve program doğrulama arasında güçlü bir bağlantı vardır.
- Goto komutlarının doğrulanması imkansızdır.
- Doğrulama seçim ve mantıksal öntest döngüleri için mümkündür.
- Güvenlikli kontrollerin doğrulanması daha basittir.

Sonuçlar

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

9.bölüm

Alt Programlar



9.Bölüm Konuları

- Giriş
- Alt Programların Temelleri
- Alt Programların Tasarım Problemleri
- Yerel Referans Platformları
- Parametre–Geçirme Metodları
- Altprogram Adı Olan Parametrelər
- Altprogramları dolaylı Olarak Çağırma
- Aşırı Yüklenmiş Programlar
- Soysal Programlar
- Fonksiyonların Tasarım Modelleri
- Kullanıcı Tanımlı Aşırı Yüklenmiş Operatörler
- Kapatmalar
- Eşyordamlar

Giriş

- İki Temel Soyutlama Olanağı
 - İşlem Soyutlama
 - Erken Vurgulanmışlardır
 - Bu bölümde tartışıldı
 - Veri Soyutlama
 - 1980'lerde vurgulanmıştır
 - Bölüm 11'de uzunca tartışıldı

Altprogramların Temelleri

- Bir altprogramın tekbir giriş noktası vardır
- Çağrılan altprogramın yürütülmesi sırasında çağrılan askıya alınır
- Çağrılan altprogramın yürütülmesi sona erince kontrol daima çağrıvana döner

Temel Tanımlar

- Bir altprogramın
 - Python'da,fonksiyon tanımları yürütülebilir,diğer bütün dillerde fonksiyon tanımları yürütülemez
 - Ruby'de işlev tanımlarında veya sınıf tanımlarının dışındada görünebilir. Eğer dışarda ise, Object metodlarıdır. Nesne olmadan fonksiyon gibi çağrırlabilirler
 - Lua'da bütün fonksiyonlar anonimdir
- Bir altprogram çağrısı altprogramın çalışması için belirtik bir istektir
 - Bir altprogram başlığı(subprogram header) tanımının (definition) ilk satırıdır; adı(name), altprogramın tipini, ve formal parametreleri içerir
- Bir altprogramın parametre profili o parametrenin sayısı,sırası ve türündür.
 - Bir altprogramın(subprogram) protokolü(protocol) onun parametre profili artı, eğer bir fonksiyon ise, döndürdüğü tiptir (return type)

Temel Tanımlar(Devam)

- *C ve C++*' da fonksiyon bildirimlerine prototipler denir.
- Bir altprogram bildirimi(subprogram declaration) altprogramın protokolünü sağlar, ama gövdesini değil
- Bir formal parametre(formal parameter) altprogram başlığında (subprogram header) gösterilen ve altprogramda(subprogram) kullanılan bir kukla değişkendir (dummy variable)
- Bir etkin parametre (actual parameter) altprogram çağrı ifadesinde(subprogram call statement) kullanılan değer(value) veya adresi gösterir

Etkin/Biçimsel Parametrelerin Uygunluğu

- Konumsal(Pozisyonel)
 - Etkin parametrelerin biçimsel parametrelere bağlılığı gereği: Birinci etkin parametre,birinci biçimsel parametreye bağlıdır,diğerleri de aynı şekilde
 - Güvenli ve Etkili
- Anahtar Sözcük
 - Etkin parametreye bağlı olan biçimsel parametrenin adı:etkin parametre ile belirtilir
 - *Avantaj:Advantage:* Parametreler herhangi bir sırada ortaya çıkabilir, böylece uygunluk hatalarından kaçınılmış olunur
 - *Dezavantaj:* Kullanıcı biçimsel parametrelerin ismini bilmek zorundandır

Biçimsel(Formal) Parametrelerin Varsayılan(Default) Değerleri

- Belirli dillerde (örn., C++, Python, Ruby, Ada, PHP), biçimsel parametreler varsayılan değerlere sahip olabilirler (Eğer hiçbir etkin parametre geçerli değil ise)
 - C++’da, varsayılan parametreler en son ortaya çıkar çünkü parametreler konumsal olarak birleştirilmiştir (anahtar sözcük parametre yok)
- Parametrelerin Değişken Sayıları
 - C# yöntemleri parametrelerin değişken sayıları olarak kabul edilebilir, uygun biçimsel parametrelerin `params` öncesinde bir dizi ile aynı olduğu sürece.
 - Ruby’de, etkin parametreler anahtar(hash) kalıp elemanı olarak gönderilir ve uygun etkin parametre yıldız işaretinden(asterisk) önce gelir.
 - Python’da, gerçek değerlerin listesi ve uygun etkin parametre asterisk(yıldız işaretü) olarak isimlendirilir.
 - Lua’da, parametrelerin değişken sayıları biçimsel parametre olarak üç perıottur; onlara `for` deyi̇miyle yada çoklu atama ile üç perıottan ulaşılır

Ruby Blokları

- Ruby sık sıkla dizlerin eleman özelliklerini kullanan yinelyici fonksiyonlarının sayılarını içerir.
- Yineleyiciler tanımlı uygulamalar tarafından bloklarla sağlanır.
- Bloklar ilişik yöntem çağrımlarıdır; parametreleri olabilir (dikey çubuklarda); Yöntem `yield` deyi̇miyle uygulandığında onlarda uygulanabilir

```
def fibonacci(last)
    first, second = 1, 1
    while first <= last
        yield first
        first, second = second, first + second
    end
end

puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "}
puts
```

Prosedürler ve Fonksiyonlar

- Alt programların iki kategorisi vardır:
 - *Prosedürler* tanımlı parametreli hesaplamaların deyimlerinin toplamıdır
 - *Fonksiyonlar* yapısal olarak prosedürlere benzerler fakat yapı bakımından matematiksel fonksiyonlar üzerine modellenmişlerdir
 - Yan etki yaratmamaları beklenir
 - Pratikte, programın yan etkisi vardır

Altprogramların Tasarım Modelleri

- Yerel değişkenler(local variables) statik midir dinamik midir?
- Altprogram tanımları(subprogram definitions) diğer altprogram tanımlarında görünebilir mi?
- Hangi Parametre–Geçirme(Parameter–Passing) metodları sağlanmıştır?
- Parametre tipleri kontrol edilmiş midir?
- Geçen(passed) bir altprogramın referans platformu(referencing environment) nedir?
- Altprogramlar(Subprograms) aşırı–yüklenebilir mi(overloaded)?
- Altprogramların(Subprograms) soysal(generic) olmasına izin verilebilir mi?
- Eğer dil altprogramın yuvalanmasına izin verirse,kapatma bunu desteklermi?

Yerel Referans Platformları

- Eğer yerel değişkenler(local variables) yığın-dinamik(stack-dynamic) ise
 - Avantajlar
 - Özyineleme(recursion) desteği
 - Yereller(locals) için bellek bazı altprogramlar(subprograms) arasında paylaşılır
 - Dezavantajlar
 - Ayırma/Serbest Bırakma(Allocation/deallocation) süresi
 - Dolaylı Adresleme(Indirect addressing)
 - Altprogramlar(Subprograms) tarih duyarlı(history sensitive) olamaz
- Yerel Değişkenler statik olabilir
 - Avantajlar ve dezavantajlar statik-dinamik yerel değişkenlerin tam tersidir

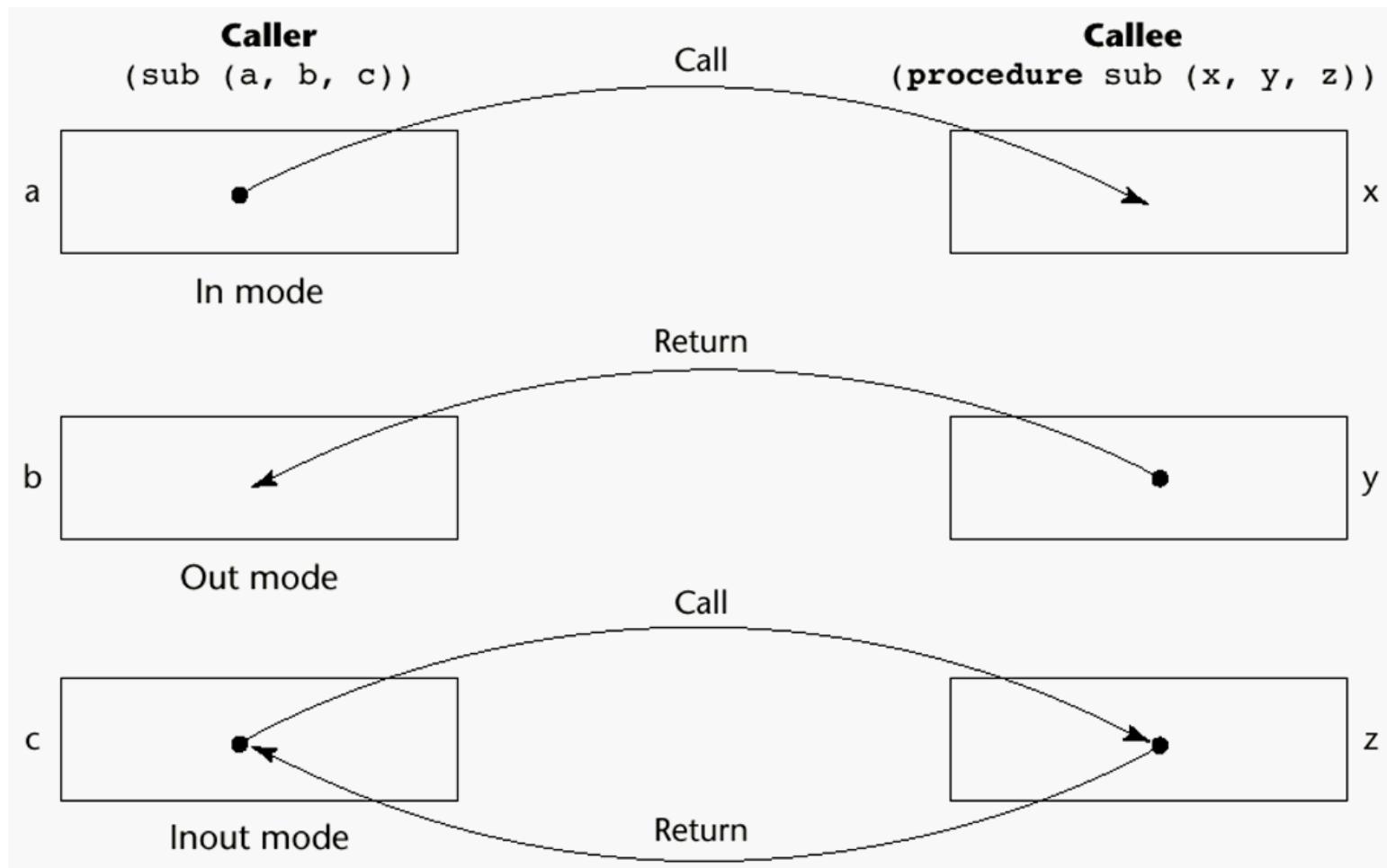
Yerel Referans Platformları: Örnekler

- Bir çok çağdaş dilde, yereller yiğin dinamiktir
- C – her ikisi (değişkenler **static** olarak tanımlanan değişkenler)
- C++, Java, Python, C#'de sadece yiğin dinamik yerel vardır
- Lua'da, bütün dolaylı bildirimli değişkenler küreseldir; yerel değişkenler **local** ile bildirilmiştir ve yiğin dinamiktir

Parametre Geçişlerinde Semantik Modeller

- In mode
- Out mode
- Inout mode

Parametre Geçişlerinin Modelleri



Kavramsal Modellerin Transferi

- Değeri fiziksel taşıma
- Erişim yolu ile taşıma

Değeriyle Geçirme (In Mode)

- Etkin parametrenin değeri uygulanabilir biçimsel parametreyi sıfırlardı
 - Normalde kopyalama tarafından uygulanan
 - Erişim yoluyla uygulanabilir fakat tavsiye edilmez (zorlama yazı koruması kolay değil)
 - *Dezavantajları* (eğer fiziksel yolla taşınmışsa): Daha çok belleğe ihtiyaç duyar ve taşımaların maliyeti
 - *Dezvantajları* (eğer erişim yoluyla taşınmışsa): Çağrılmış altprogramda yazmaya korumalı olmalıdır ve maliyeti çoktur

Sonucuyla Geçirme (Out Mode)

- Yerelin değeri(local's value) çağrıvana(caller) geri gönderilir
- Genellikle fiziksel taşıma(Physical move) kullanılır
- Dezavantajlar:
 - Eğer değer(value) passed ise, zaman ve alan
 - Her iki durumda da, sıraya bağımlılık problem olabilir
- Potansiyel problemler:
 - `sub(p1, p1);` hangisi biçimsel parametre ise tekrar kopyalanıp `p1` tarafından temsil edilecek
 - `sub(list[sub], sub);` Programın balında yada sonunda adres listesini[sub] hesapla

Sonucuya–Değeriyle Geçirme (inout Mode)

- Pass-by-value ve pass-by-result kombinasyonu
- Aynı zamanda pass-by-copy de denir
- Biçimsel parametrelerin yerel bellekleri vardır
- Dezavantajlar:
 - Pass-by-result'inkiler
 - Pass-by-value'inkiler

Referansıyla Geçirme (Inout Mode)

- Bir erişim yolunu(access path) geçme
- Aynı zamanda pass-by-sharing de denir
- Avantaj: geçiş(passing) işlemi verimlidir (kopyalama veya ikiye katlanmış bellek yoktur)
- Dezavantaj:
 - Erişimler daha yavaştır
 - Potansiyel istenmeyen yan etkiler (çarpışmalar)
 - İstenmeyen takma adlar (genişletilmiş erişim)

```
fun(total, total);  fun(list[i], list[j];  fun(list[i], i);
```

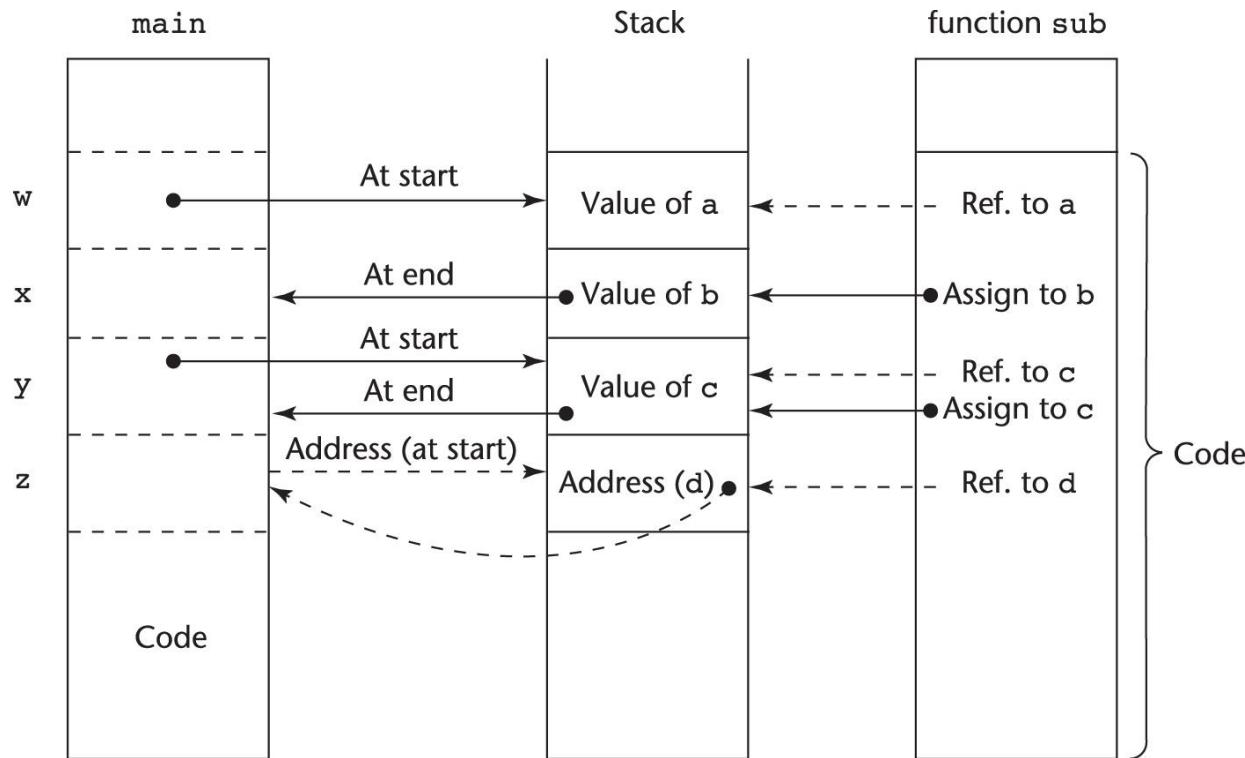
Adıyla Geçirme (Inout Mode)

- Metinsel ornatım(yerdeğiştirme) ile(textual substitution)
- Formaller çağrı(call) sırasında bir erişim metoduna (access method) bağlanır, fakat bir değere veya adrese asıl bağlama referans(reference) veya atama(assignment) sırasında olur
- İzinler geç bağamlarda esnetilebilir
- Uygulamalar referansal platformun arayan parametresini geçirmesini gerektirir, böylece etkin parametre adresi hesaplanabilir

Parametre geçirmenin(Parameter Passing) implementasyonu

- Birçok dilde parametre传递ini yiğin süresinin yerini alıyor
- Referansıyla geçirmede uygulaması çok basit; sadece adres yiğinının yerini alır

Parametre-Geçirmenin (Parameter-Passing) Yığıн(Stack) Implementasyonu



Fonksiyon başlığı: `void sub(int a, int b, int c, int d)`

Fonksiyon ana arama: `sub(w, x, y, z)`

(**w** 'değeri tarafından geç, **x** tarafından sonuç **y** tarafından sonuç değeri, referans tarafından **z**)

Başlıca Dillerde Parametre Geçirme Metodları

- C
 - Değeriyile geçirme
 - Referansıyla geçirme işaretçileri parametre olarak kullanarak ulaşma
- C++
 - Referansıyla geçirmede referansal tip denilen özel bir işaretçi türü
- Java
 - C++ gibidir, farkı sadece referanslar
- Ada
 - Üç semantik model de mümkündür: `in`, `out`, `in out`; `in` varsayılan moddur
 - “`out`” ise referans edilemez; “`in`” ise referans edilebilir fakat atanamaz; `in out` parametreleri referan edilebilir ve atanabilir

Başlıca Dillerde Parametre Geçirme Metodları(devamı)

- Fortran 95+
 - Parametreler in, out, yada inout modlarında etkinleştirilebilir
- C#
 - Varsayılan metod: değeriyle geçirme
 - Referansıyla geçirmede öncelikle biçimsel parametre ve etkin parametre `ref` ile belirtilmiştir
- PHP:C#'ye çok benzerdir, ikisinin de etkin yada biçimsel parametreleri `ref` ile belirtilmesi dışında
- Perl: bütün etkin parametreler dolaylı olarak önceden yerleştirilmiş ve tanımlanmış dizi ismi `@_`
- Python ve Ruby atamaya geçirmeyi kullanır (burun data değerleri nesnelerdir); etkin biçimsele atanmıştır

Tip Kontrol Parametreleri(Type checking parameters)

- Güvenilirlik için çok önemli olduğu düşünülmektedir
- FORTRAN 77 ve orjinal C: yok
- Pascal, FORTRAN 90+, Java, ve Ada: daima gerektirir
- ANSI C and C++: kullanıcıya bırakılmıştır
 - Prototipler
- Nispeten yeni diller Perl, JavaScript, ve PHP tip kontrolünü gerektirmez
- In Python ve Ruby, değişkenlerin tipleri yoktur (nesnelerin vardır), yani parametre tip kontrolü mümkün değildir

Parametre olarak Çok boyutlu Diziler (Multidimensional Arrays)

- Eğer bir çok boyutlu dizi(multidimensional array) bir altprograma geçirilirse ve altprogram(subprogram) ayrı olarak derlenirse (compiled), derleyici(compiler) bellek eşleme fonksiyonunu(storage mapping function) oluşturmak(build) için o dizinin bildirilmiş boyutunu(declared size) bilmesi gereklidir

Parametre olarak Çok boyutlu Diziler : C ve C++

- Programcı, etkin(actual) parametredeki ilk altsimge(subscript) dışında bütün belirtilmiş boyutları (declared sizes) dahil etmek zorundadır
- Bu esnek altprogramlar(Subprograms) yazmaya izin vermez
- Bu esnek altprogramlar(Subprograms) yazmaya izin vermez

Parametre olarak Çok boyutlu Diziler : Ada

- Ada – problem değildir
 - Kısıtlı Diziler – boyut dizi tipinin bir parçasıdır
 - Kısıtlı olamayan diziler – bildirilmiş boyut nesne bildiriminin(object declaration) parçasıdır

Parametre olarak Çok boyutlu Diziler : Fortran

- Biçimsel parametreler başlıktan sonra deklarasyonu(bildirim) olan dizilerdir
 - Tek için dizi boyutunda, indis alakasız(??)
 - Parametre olarak çoklu diziler için, biçimsel parametre bildirimleri geçirilmiş parametreler içerebilir, böylece bu değişkenler kullanılıp gönderim fonksiyonu olarak depolanabilir

Parametre olarak Çok boyutlu Diziler : Java and C#

- Ada ile benzerdir
- Diziler nesnelerdir; hepsi tek boyutlandırılmıştır fakat elemanlar diziler olabilir
- Her dizi adlandırılmış bir sabiti devralır (`length` Java'da, `Length` C#'de) dizi nesnesi yaratıldığında dizi uzunluğu ayarlanmış olur

Tasarımda Düşünülmesi Gerekenler

- İki önemli düşünce
 - Verimlilik
 - Tek yönlü veya çift yönlü
- Fakat bu iki düşüncede çelişkidir
 - İyi programlama => değişkenlere sınırlı erişim, yani mümkünse tek yönlü kullanım
 - Fakat referansıyla geçirme büyük yapıların geçirilmesinden daha verimlidir

Alt Programlar

- Altprogramların adının parametre olarak geçmesi bazen uygundur
- Problemler:
 1. Parametrelerde tip kontrolü yapıldı mı?
 2. Parametre olarak gönderilmiş olan bir altprogramın doğru referans çevresi nedir?

Altprogram Parametre İsimleri

- *Yüzeysel Bağlama*: Platformu yasalaştıran (enacted) altprogram
 - Dinamik-kapsamlı dil için en doğaldır
- *Derin Bağlama*: Platformu tanımlayan alt program
 - Dinamik-kapsamlı dil için en doğaldır
- *Ad hoc bağlama*: Platformda çağrı deyiminin geçtiği altprogram

Altprogramları Dolaylı olarak Çağırma

- Genellikle çağrılmak birkaç olası alt program vardır ve doğru olanın çalışma süresi uygulanan kadar bilinmez (örn., olay giderme ve GUIs)
- C ve C++’da, bu tür çağrılar fonksiyon işaretçileriyle yapılır

Alt Programları Dolaylı Olarak Çağırma

(devam)

- C#'de yöntem işaretçileri nesne olarak implementasyon(uygulandığında) edildiğinde buna *temsilciler(delegate)* denir

- Bir temsilci(delegate) deklarasyonu:

```
public delegate int Change(int x);
```

- Bu temsilci tipinde Change adında bir parametre, int alan bir parametre ile örneklendirilebilir ve int değerine döner

Yöntem: static int fun1(int x) { ... }

Instantiate: Change chgfun1 = new Change(fun1);

Denilebilir: chgfun1(12);

- Bir temsilci bir adresden fazlasını depolayabilir,buna *çoğa gönderim temsilci (multicast delegate)*denir

Aşırı Yüklenmiş Altprogramlar

- Bir overloaded altprogram(subprogram) aynı referans çevresinde(referencing environment) bulunan diğer bir altprogramla aynı ada sahip olan altprogramdır
 - Aşırı yüklenmiş altprogramların bütün versiyonlarında benzersiz bir protokol vardır
- C++, Java, C#, ve Ada yerleşik aşırı yüklenmiş programlardır
- Ada'da, aşırı yüklü fonksiyonun dönüş türü belirsizliği gideren aramalar için kullanılabilir (böylece iki aşırı yüklenmiş fonksiyon aynı parameterlere sahip olabilir)
- Ada, Java, C++, ve C# altprogramlarının aynı ismi ile çoklu versiyonlarının yazımına izin verirler

Soysal Altprogramlar(Generic Subprograms)

- Bir soysal(generic) veya polimorfik altprogram(polymorphic subprogram) farklı etkinleştirmelerde farklı tipten parametreler alandır
- Aşırı-yüklenmiş altprogramlar(Overloaded Subprograms) ad hoc polymorphism sağlar
- *Subtype polymorphism* T türünde bir değişken tipi T herhangi bir nesne ya da T elde edilen herhangi bir tipe erişebilir (OOP dilleri) anlamına gelir
- Altprogramın parametrelerinin tiplerini tanımlayan bir tip ifadesinde kullanılan soysal bir parametre alan bir altprogram (subprogram) parametrik polimorfizm sağlar
 - Ucuz derleme-zaman yerine için dinamik bağlama

Soysal Altprogramlar(Generic Subprograms)(devam)

- C++
 - Fonksiyon bir çağrıda(call) adlandırıldığından veya adresi & operatörü ile çağrıldığında, C++ şablon(template) fonksiyonları örtük olarak(implicitly) başlatılır
 - Soysal altprogramlar **template** tarafından takip edilir soysal değişkenleri maddeler halinde tür isimleri ve sınıf isimlerine göre listelenir

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

Soysal Altprogramlar(devam)

- Java 5.0
 - Java 5.0 ve C++'arasındaki soysal farklar:
 1. Java 5.0'de soysal parametreler sınıflandırılmış olmalıdır
 2. Java 5.0 'de soysal yöntemler sadece bir kere doğru olarak soysal yöntemler olarak örneklendirilmelidir
 3. Sınırlamalar sınıf aralıklarına göre belirtilmelidir ki soysal yöntemler soysal parametreler olarak geçebilsin
 4. Soysal parametrelerin joker türleri

Soysal Altprogramlar(devam)

- Java 5.0 (devam)

```
public static <T> T doIt(T[] list) { ... }
```

- parametreler soysal elemanların dizileridir (T türün ismidir)

- Bir çağrı:

```
doIt<String>(myList);
```

Soysal parametrelerin sınırları olabilir:

```
public static <T extends Comparable> T  
    doIt(T[] list) { ... }
```

Soysal tür Comparable arabirimini uygulayan bir sınıf olmalıdır

Soysal Altprogramlar(devam)

- Java 5.0 (continued)
 - Joker türleri

Collection<?> Derleme sınıfları için bir joker türüdür

```
void printCollection(Collection<?> c) {  
    for (Object e: c) {  
        System.out.println(e);  
    }  
}
```

– Herhangi bir derleme sınıfı için çalışabilir

Soysal Altprogramlar(devam)

- C# 2005
 - Java 5.0 ile benzer soysal yöntemleri destekler
 - Bir fark: etkin tür parametreler derleyici tanımlanmamış türüne ulaşabilirse,bir çağrı atlanabilir
 - Diğer – C# 2005 jokerleri(wildcard) desteklemez

Soysal Altprogramlar(devam)

- F#
 - Soysal tür dışında bir parametre türü veya bir işlemin dönüş türü saptanamıyorsa – *otomatik genelleştirme*
 - Böyle türler bir kesme işaretti veya bir har ile belirtilir, örn., 'a
 - Fonksiyonlar genel parametreleri tanımlayabilir

```
let printPair (x: 'a) (y: 'a) =
    printfn "%A %A" x y
    – %A her tür için format koddur
    – Bu parametreler kısıtlı tür degillerdir
```

Soysal Altprogramlar(devam)

- F# (devam)
 - Eğer parametrelerin fonksiyonları aritmetik operatörler ile kullanılıyorlarsa hatta parametreler soysal olarak belirtilse bile
 - Tür sonuç çıkarmaları(inferencing) ve tür zorlamalarının(coercions) eksikliği yüzünden, F# soysal fonksiyonları C++, Java 5.0+, ve C# 2005+'dan çok daha az kullanışlıdır

Fonksiyonların Dizayn Problemleri

- Yan etkilere izin verilir mi?
 - Çift-yönlü parametreler (Ada izin vermez)
- Hangi return tiplerine izin verilir?
 - Çoğu zorunlu diller return türlerini kısıtlar
 - C diziler ve fonksiyonlar türleri haricinde hiçbirine izin vermez
 - C++ ,C gibidir ancak sadece kullanıcı tanımlı türlere izin verir
 - Ada altprogramları herhangi bir türüne dönabilir(return) (fakat Ada altprogramları tür değildir, yani dönemezler(return olamazlar))
 - Java ve C# yöntemleri herhangi bir türüne return olabilir(dönüştürülebilir) (çünkü yöntemler tür değildir,döndürülemezler(return olamazlar))
 - Python ve Ruby yöntemleri birinci sınıf nesne gibi işleyebilir,yani diğer sınıfları kadar iyi dönabilirler(return olabilirler)
 - Lua fonksiyonlarının çoklu değerlere dönmesine(return olmasına) izin verir

Kullanıcı tanımlı Aşırı yüklü Operatörler

- Ada, C++, Python, ve Ruby'de kullanıcılar daha fazla operatör overload yapabilirler
- Python örneği:

```
def __add__(self, second) :  
    return Complex(self.real + second.real,  
                  self.imag + second.imag)
```

Kullanım: hesaplama $x + y$, $x.__add__(y)$

Kapatmalar

- Kapatma(closure): bir altprogramın ve referansal platformun nerde tanımlandığıdır
 - Altprogram rastgele bir yerden çağrılabılır değilse referansal platform gereklidir
 - Statik-kapsamlı dil yuvalanmış altprogramlara izin vermez kapatmalara ihtiyacı yoktur
 - Kapatmalara sadece eğer altprogram yuvalanmış kapsamının içindeki değişkene erişebildiği zaman ihtiyaç duyulur ve herhangi bir yerden çağrılabılır
 - Kapatmaları desteklemek için, bir implemantasyonda(uygulamada)bazı değişkenlere sınırsız ölçüde değer verilebilir (çünkü bir altprogram normalde hayatı olmayan bir değişkene erişebilir)

Kapatmalar (devam)

- Bir JavaScript kapatması:

```
function makeAdder(x) {  
    return function(y) {return x + y; }  
}  
  
...  
  
var add10 = makeAdder(10);  
var add5 = makeAdder(5);  
  
document.write("add 10 to 20: " + add10(20) +  
    "<br />");  
  
document.write("add 5 to 20: " + add5(20) +  
    "<br />");
```

- Kapatılması `makeAdder` tarafından döndürülen adsız işlem

Kapatmalar (devam)

- C#

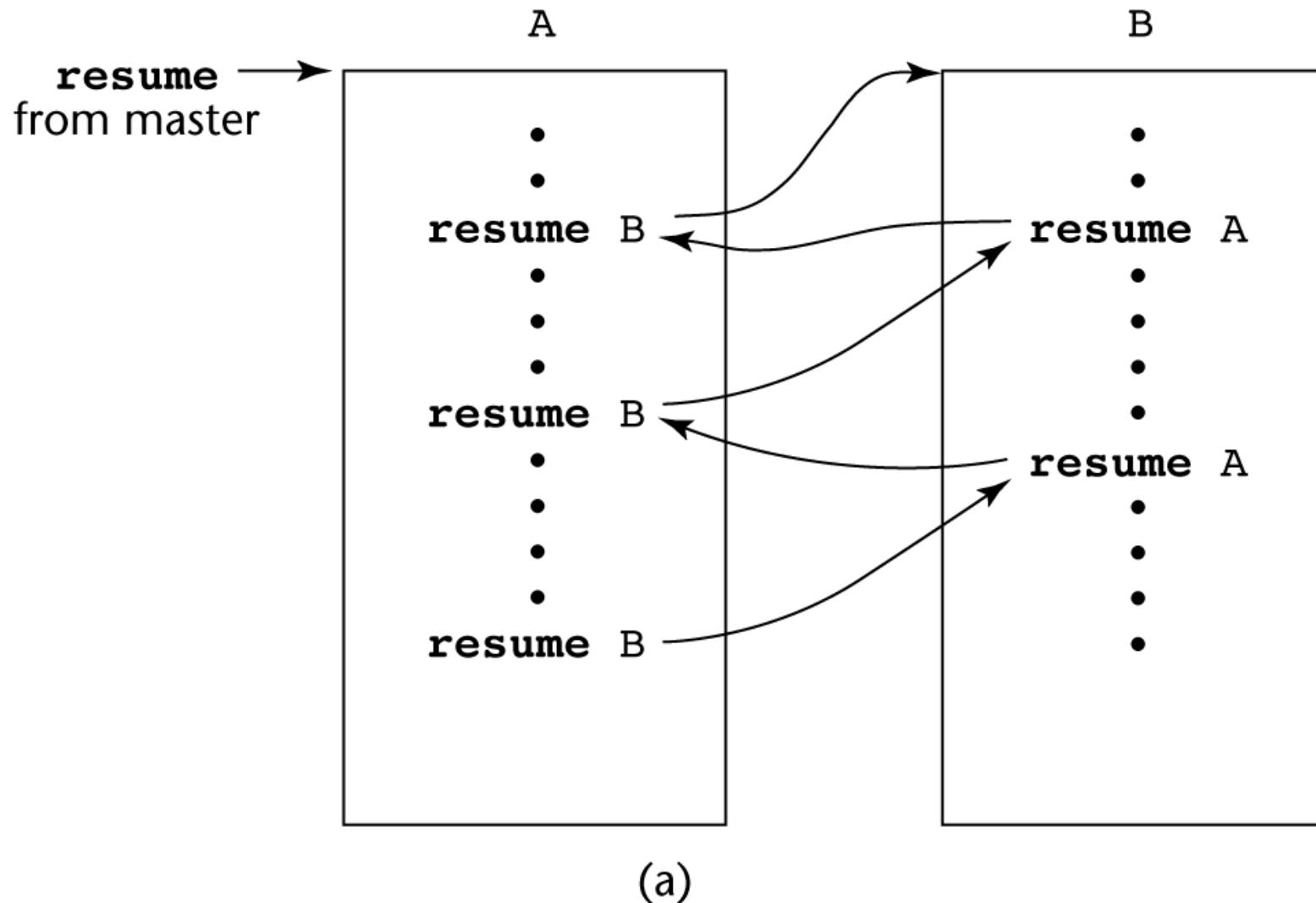
- Yuvalanmış bir temsilci kullanarak C# ‘de aynı kapatmayı yazabiliriz
- `Func<int, int>` (return türü) specifies a delegate that takes an `int` as a parameter and returns an `int`

```
static Func<int, int> makeAdder(int x) {  
    return delegate(int y) { return x + y; };  
}  
  
...  
Func<int, int> Add10 = makeAdder(10);  
Func<int, int> Add5 = makeAdder(5);  
Console.WriteLine("Add 10 to 20: {0}", Add10(20));  
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

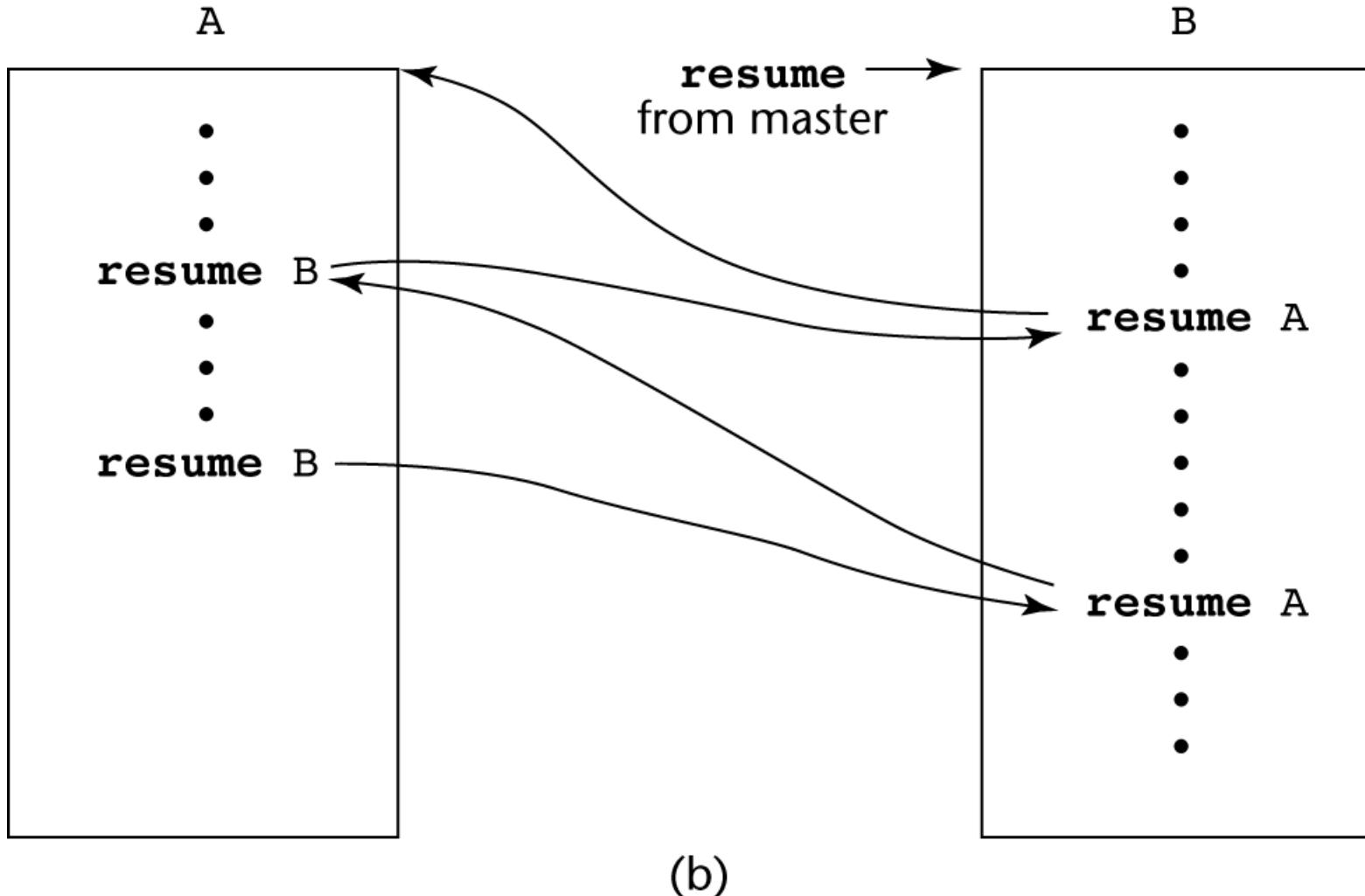
Eşyordamlar

- Bir eşyordam(c coroutine) birden çok girişi (entries) olan ve bunları kendi başına kontrol eden bir altprogramdır(subprogram) Lua'da
- Simetrik Kontrol(symmetric control) de denir
- Bir eşyordamın ilk sürdürmesi(resume) onun başlangıcınınadır, fakat sonra gelen çağrılar(calls) eşyordamın en son çalıştırılan ifadesinden hemen sonraki noktadan girer
- Genellikle, eşyordamlar(c coroutines) tekrar tekrar birbirini sürdürür(resume), belki sonsuza kadar
- Eşyordamlar(Coroutines) program birimlerinin(program units--(eşyordamlar)) eşzamanlıymış gibi yürütülmesini(quasi-concurrent execution) sağlar

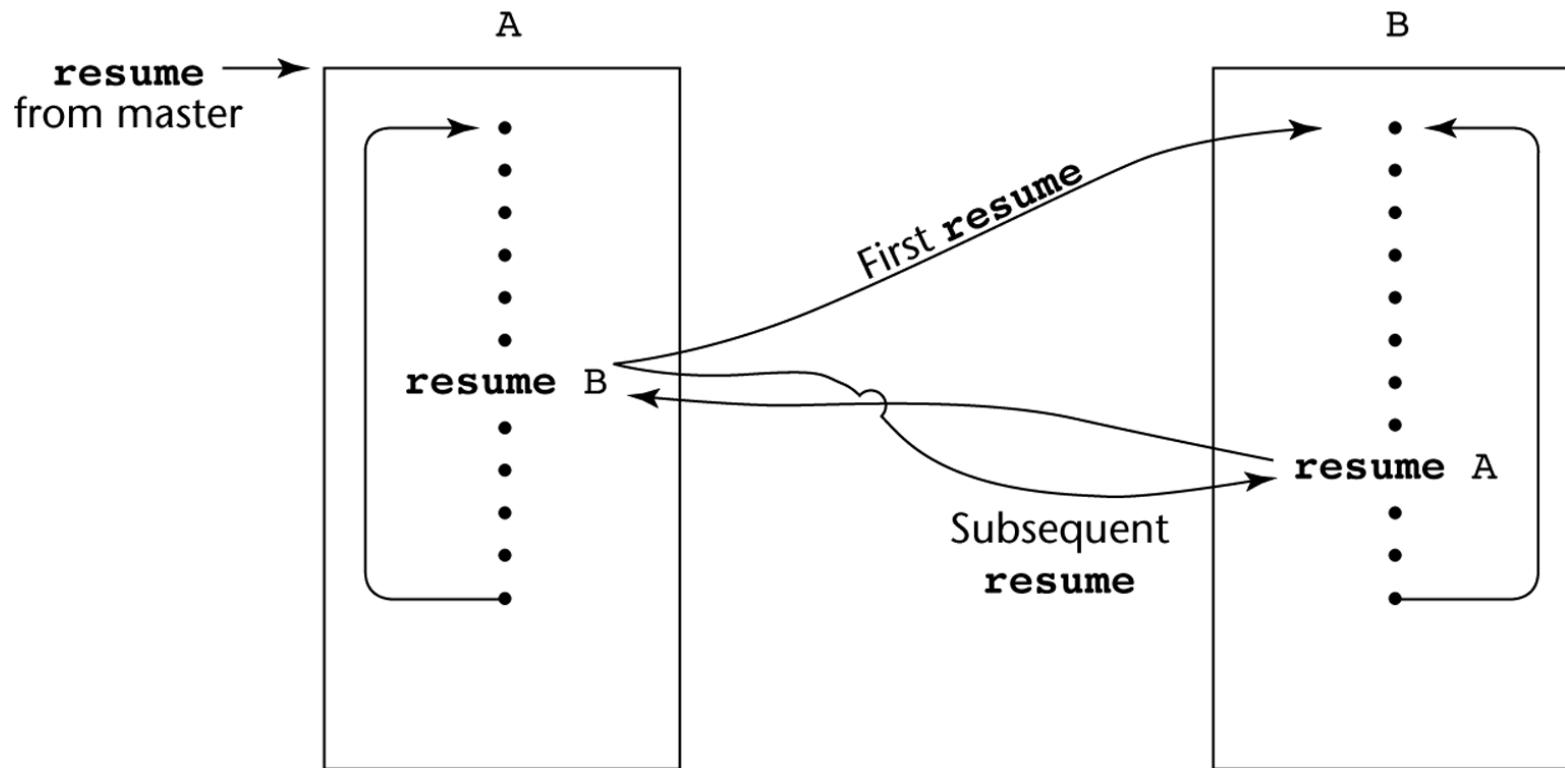
Eşyordamlar



Eşyordamlar



Eşyordamlar

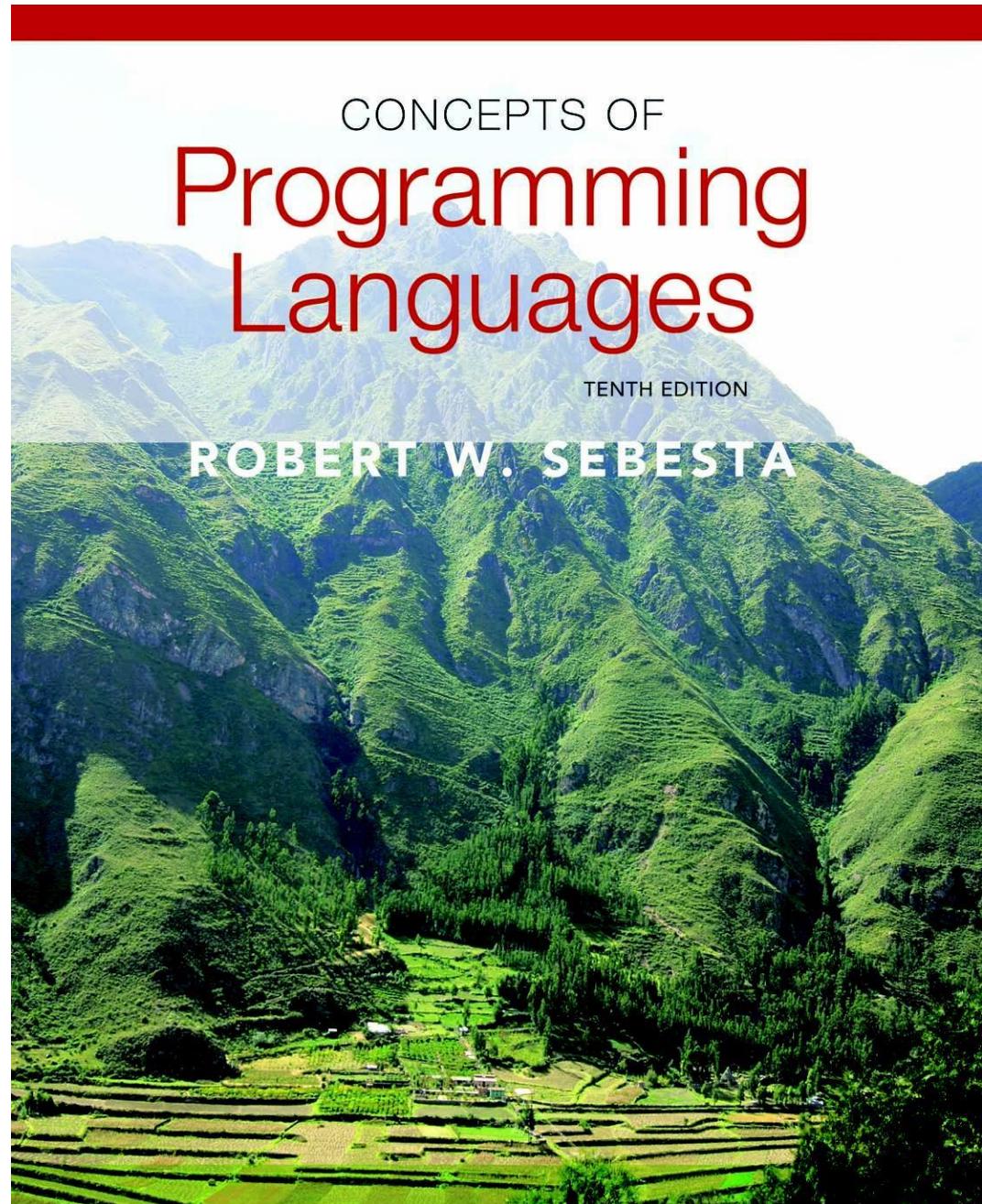


Özet

- Bir altprogram tanımlarken olay yine bir altprogram tarafından temsil edilir
- Altprogramlar fonksiyon veya prosedürde olabilir
- Altprogramlardaki yerel değişkenler statik yada dinamik olarak yiğilabilirler
- Parametre geçmelerinde üç model vardır: in mode, out mode, ve inout mode
- Bazı diller operatörlerin aşırı yüklenmesine izin verir
- Altprogramlar soysal olabilir
- Bir kapatma, altprogram ve onun referansal platformudur
- Bir eşyordam çoklu girişlerde bir altprogramdır

Bölüm 10

Altprogramların Uygulanması



10. Bölüm’ün Başlıkları

- Arama ve geri dönüşlerin genel anlamı
- “Basit” alt programların uygulaması
- Alt programların yiğin dinamiği bölgesel değişkenleri ile uygulanması
- İçiçe alt programlar
- Bloklar
- Dinamik kapsam uygulaması

Arama ve geri dönüşlerin genel anlamı

- Bir dilin alt program arama ve geri dönüş işlemleri hep birlikte onun *alt program* bağlantısı olarak adlandırılır.
- Bir Alt Programı Aramanın Genel Anlamı
 - * parametre geçiş metotları
 - * bölgesel değişkenlerin yiğin dinamiği paylaşımı
 - * arama programının uygulama durumunun kaydedilmesi
 - * kontrol transferi ve geri dönüşün düzenlenmesi
 - * içiçe programlar destekleniyorsa, yerel olmayan değişkenlere erişim yeniden düzenlenmelidir.

Arama ve Geri Dönüşlerin Genel Anlamı

Alt program geri dönüşlerinin genel anamları:

- Giriş ve çıkış modu parametreleri kendi değerlerine geri döndürülmelidir.
- – Yığın dinamiği bölgelerinin atamalarının kaldırılması
- – Uygulama durumlarının eski haline getirilmesi (sıfırlanması)
- – Arayıcının geri dönüş kontrolü

“Basit” Alt Programların Uygulanması

- Arama anımları:
 - Arayanın uygulama durumunu kaydet
 - Parametreleri girin
 - Aranana yapılan geri dönüş adresini girin
 - Aranana geri dönüş kontrolü

“Basit” Alt Programların Uygulanması (devamı)

- Geri dönüş anımları
- Eğer giriş değer sonuçları ya da mod dışı parametreler kullanılırsa, bu parametrelerin mevcut değerlerini onların gerçek eş parametrelerine taşı.
- Eğer bu bir fonksiyon ise, fonksiyonel değerleri arayıcının bulabileceği bir yere taşı.
- Arayıcının uygulamam durumunu sıfırla
- Arayıcıya tekrar kontrol transferi

Gereken Alan:

- Durum bilgisi, parametreler, geri dönüş adresi, fonksiyonların geri dönüş değeri, geçici öğeler

'Basit' Alt Programların Uygulanması (devamı)

- *İki ayrı kısım: gerçek ve gerçek olmayan kod kısımları (yerel değişkenler ve değişimebilen veriler)*
- *Uygulamadaki bir alt programın kodsız kısmının taslağına ya da formatına aktivasyon kaydı denir.*
- *Aktivasyon kaydının bir örneği, aktivasyon kaydının somut bir kanıtıdır (özel bir alt program aktivasyonu için bilgi toplanması)*

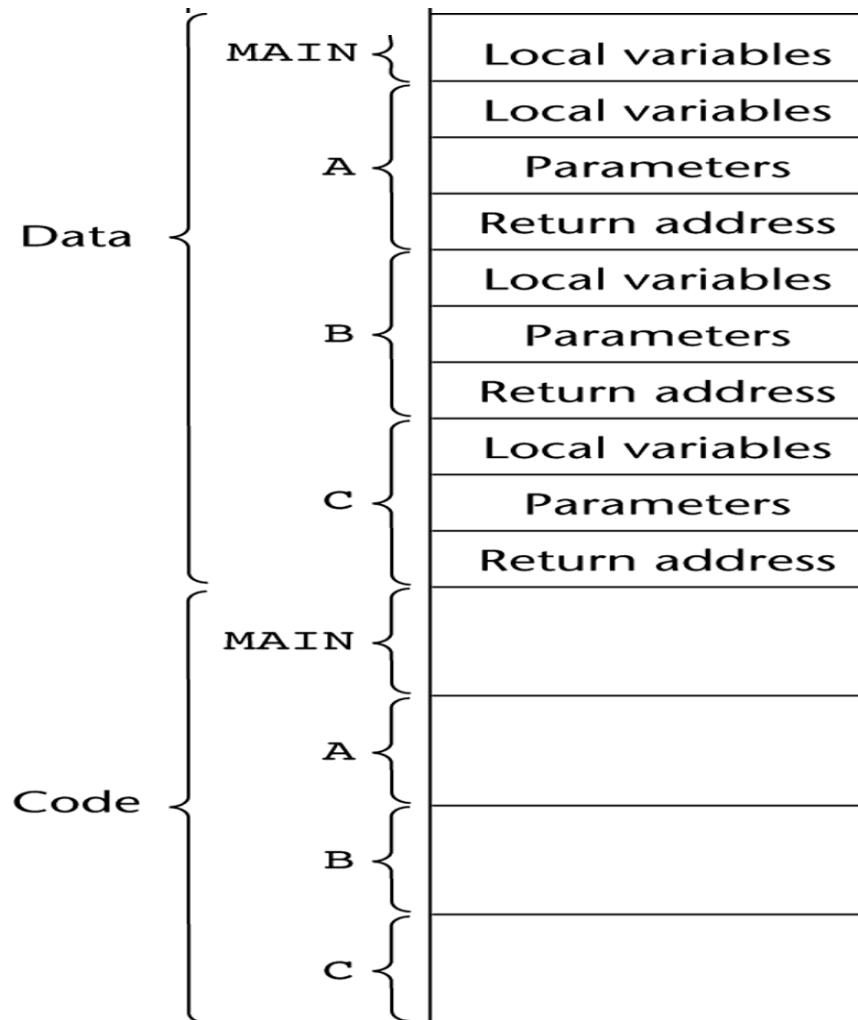
Basit Altprogramlar İçin Bir Aktivasyon Kaydı

Local variables

Parameters

Return address

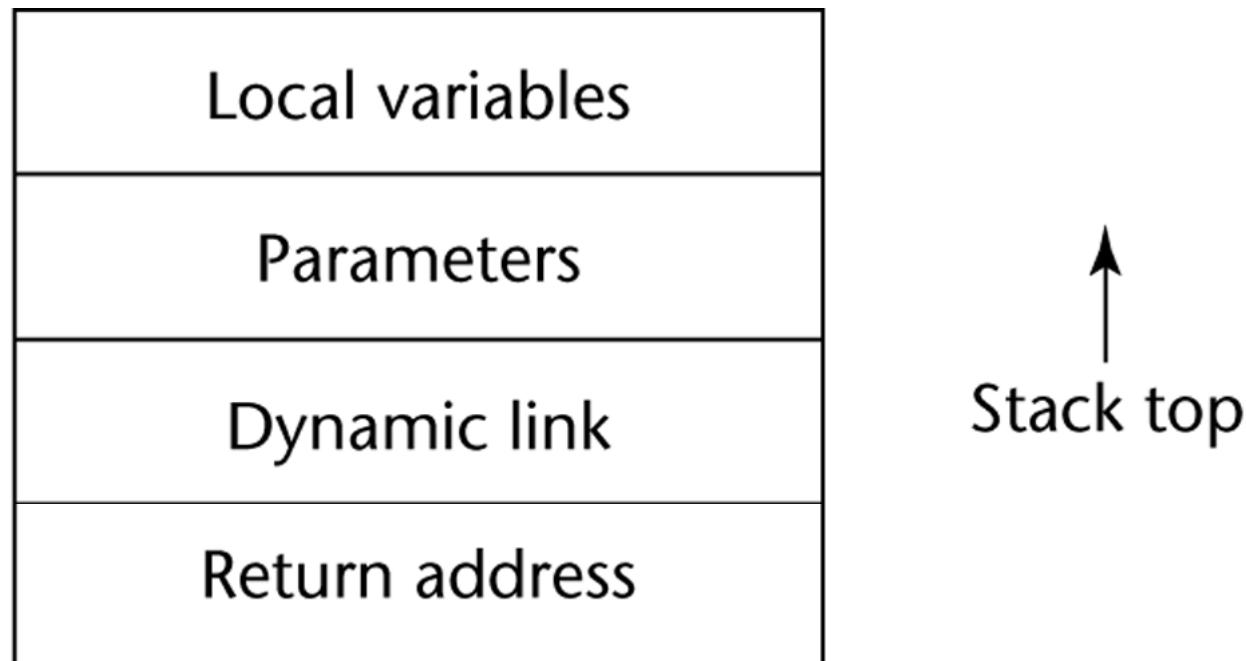
“Basit” alt programlı bir programın kod ve aktivasyon kayıtları



Yığın-dinamik yerel değişkenli alt programların uygulanışı

- Daha karmaşık aktivasyon kaydı
 - Derleyici yerel değişkenlerin atamasının kaldırılması ve gizli atamalara neden olan kodları oluşturmalıdır.
 - Tekrarlamalar desteklenmelidir (bir alt programın çoklu eş zamanlı aktivasyonlarının olasılığını ekler)

Yığın Dinamikte Yerel Değişkenler İçin Tipik Bir Aktivasyon Kaydı

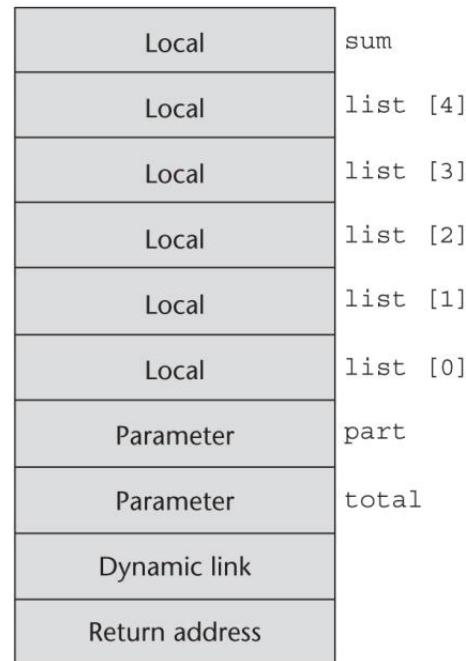


Yığın–dinamik yerel değişkenli alt programların uygulanması: aktivasyon kaydı

- Aktivasyon kayıt formatı statiktir, fakat boyutu dinamik olabilir.
- Dinamik bağlantı arayanın aktivasyon kaydının tepesini gösterir.
- Bir aktivasyon kayıt örneği, bir alt program arandığında dinamik olarak oluşturulur.
- Aktivasyon kayıt örnekleri yürütme süresi yığınında durur.
- Run-time sistemi çevre işaretçisinin devamlılığını sağlamalıdır. Çevre işaretçisi daima mevcut çalışan programın aktivasyon kayıt örneğinin merkezini gösterir.

C Fonksiyon Örneği

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



Gözden geçirilmiş anlam arama /geridönüş işlemleri

Arayıcı işlemleri

- Aktivasyon kayıt örneği oluştur
- Mevcut program biriminin işletim durumunu kaydet
- Parametreleri hesapla ve gir
- Aranana geri dönüş adresini girin
- Aranana transfer kontrol
- Arananın Giriş İşlemleri
- yiğindaki eski çevre işaretçisini (EP) dinamik bağlantı olarak kaydedin ve yeni değerler belirleyin
- Yeni değerler atayın

Gözden geçirilmiş anlam arama /geridönüş işlemleri (devamı)

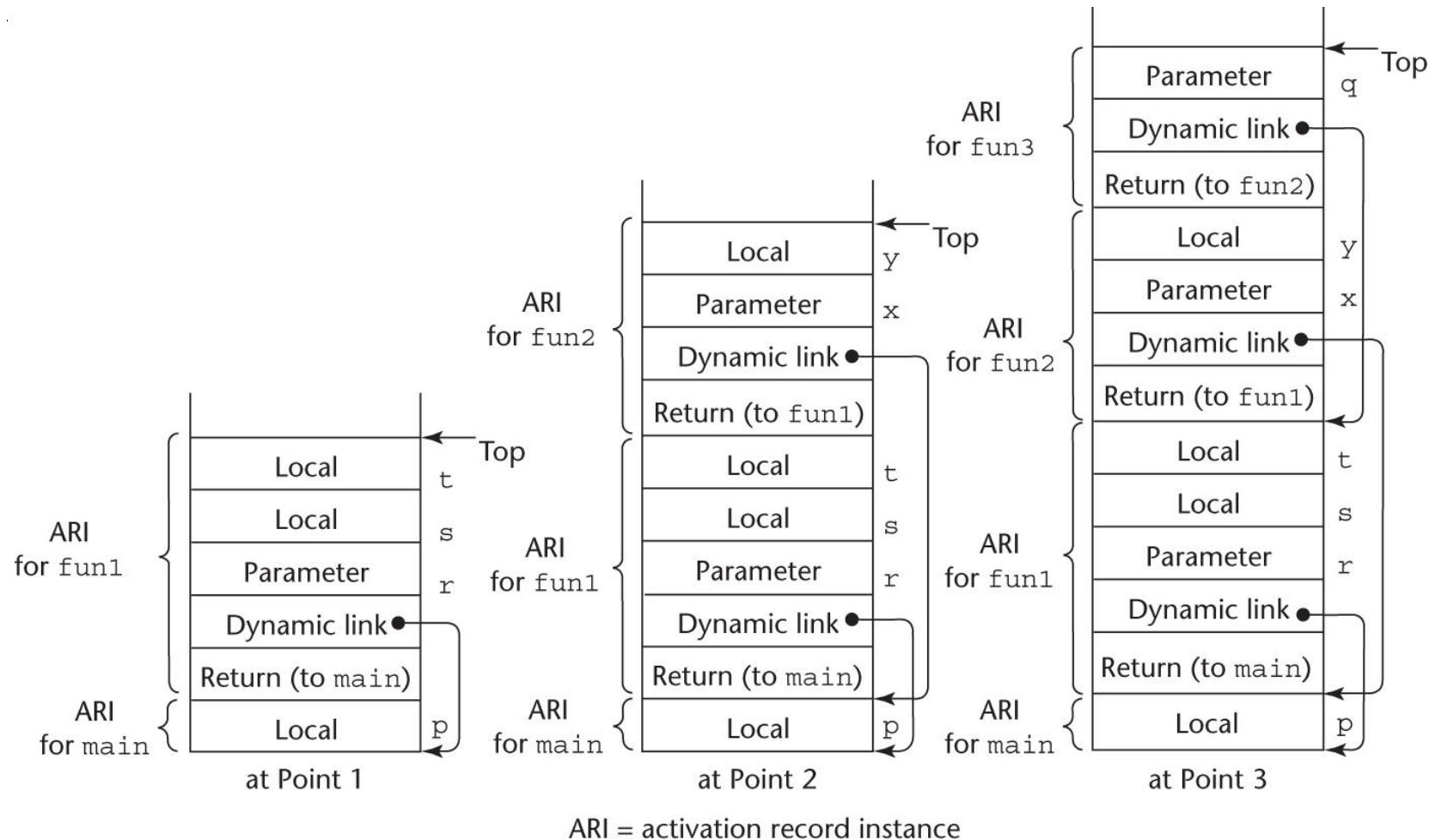
- Arananın son işlemleri
 - Giriş değer sonuçları ya da mod dışı parametreler varsa, bu parametrelerin mevcut değerleri eşdeğer olan gerçek parametrelere taşınır.
 - Eğer alt program çalışıyorsa, değeri ayananın ulaşabileceği bir yere taşınır.
 - Yığın işaretçisini mevcut EP-1 değerine getirerek sıfırlayın ve EP'yi eski dinamik bağlantı değerine getirin.
 - Arayanın işlem durumunu sıfırlayın
 - Arayana tekrar transfer kontrolü

Rekürsif Olmayan Bir Altprogram Örneği

```
void fun1(float r) {  
    int s, t;  
    ...  
    fun2(s);  
    ...  
}  
void fun2(int x) {  
    int y;  
    ...  
    fun3(y);  
    ...  
}  
void fun3(int q) {  
    ...  
}  
void main() {  
    float p;  
    ...  
    fun1(p);  
    ...  
}
```

main **calls** fun1
fun1 **calls** fun2
fun2 **calls** fun3

Rekürsif Olmayan Bir Altprogram Örneği



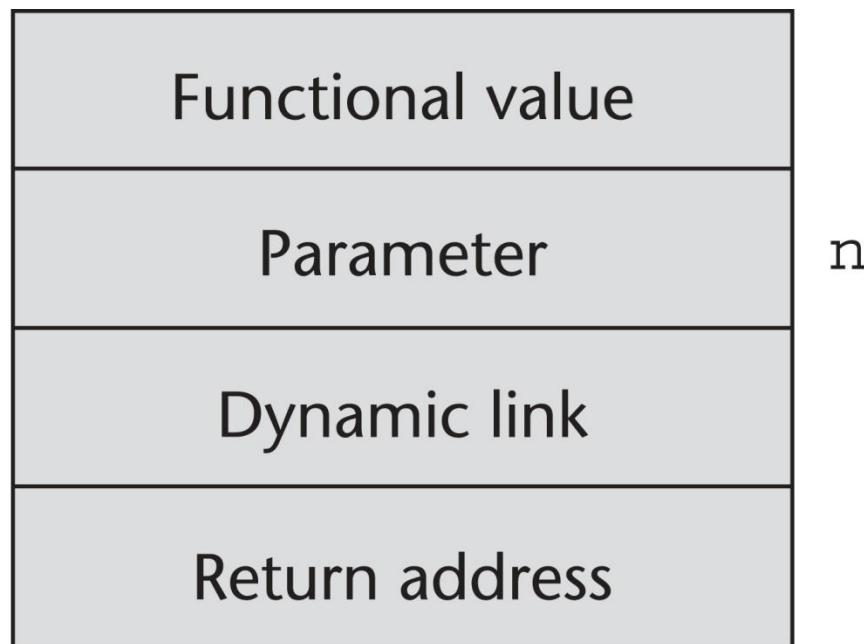
Dinamik zincirler ve yerel offset

- Belirli bir zamanda yığındaki dinamik bağlantılar kümesine *dinamik zincir* ya da *arama zinciri* denir.
- Yerel değişkenlere, adresi EP de bulunan aktivasyon kaydının başlangıcından, offset'leri tarafından erişim mümkündür.
- yerel değişkenin local_offset'i derleyici tarafından derleme anında belirlenebilir.

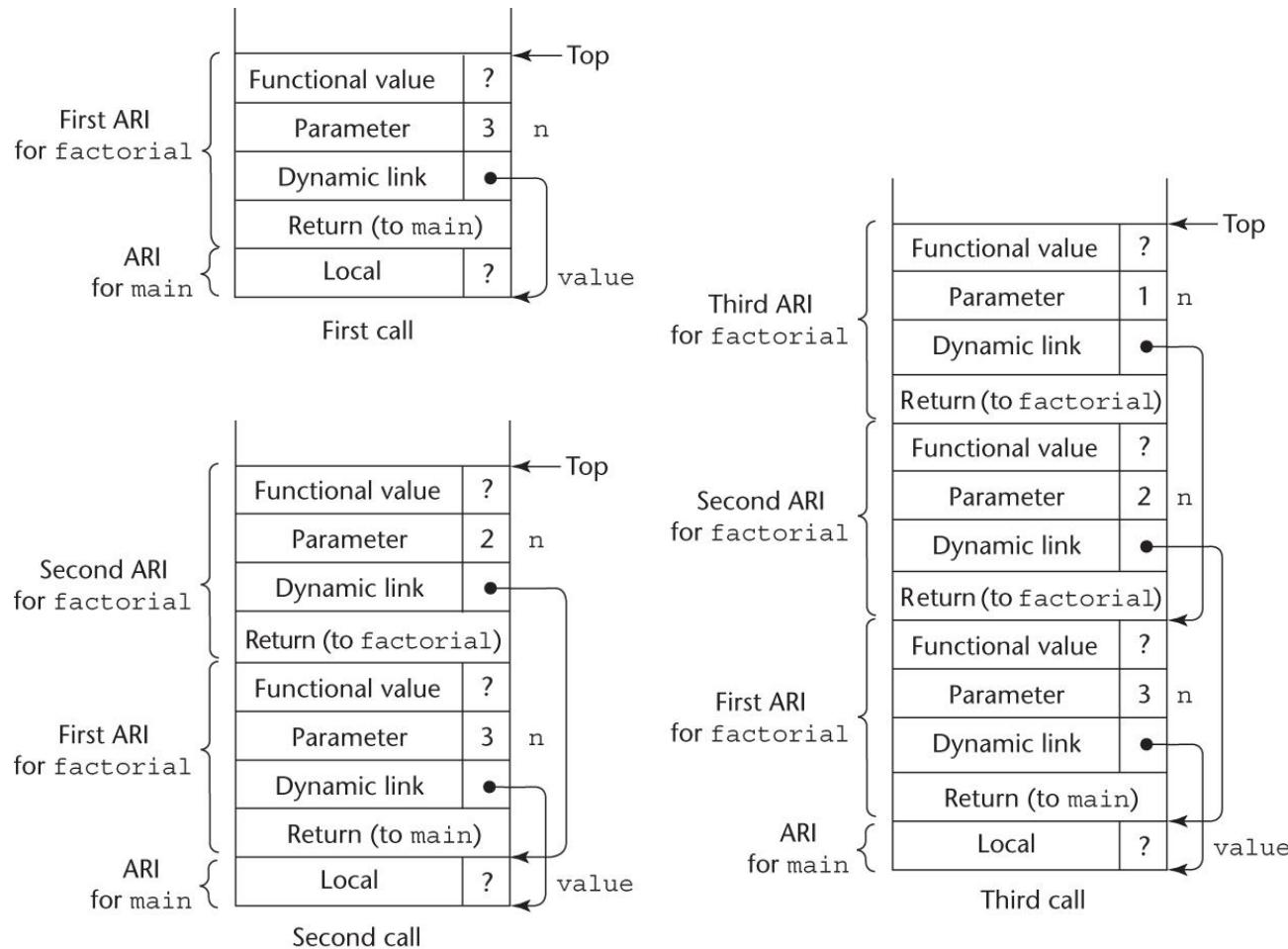
Tekrarlamalı örnek

- Önceki örnekte kullanılan aktivasyon kaydı, tekrarı destekler.

Faktoriyelin Aktivasyon Kaydı

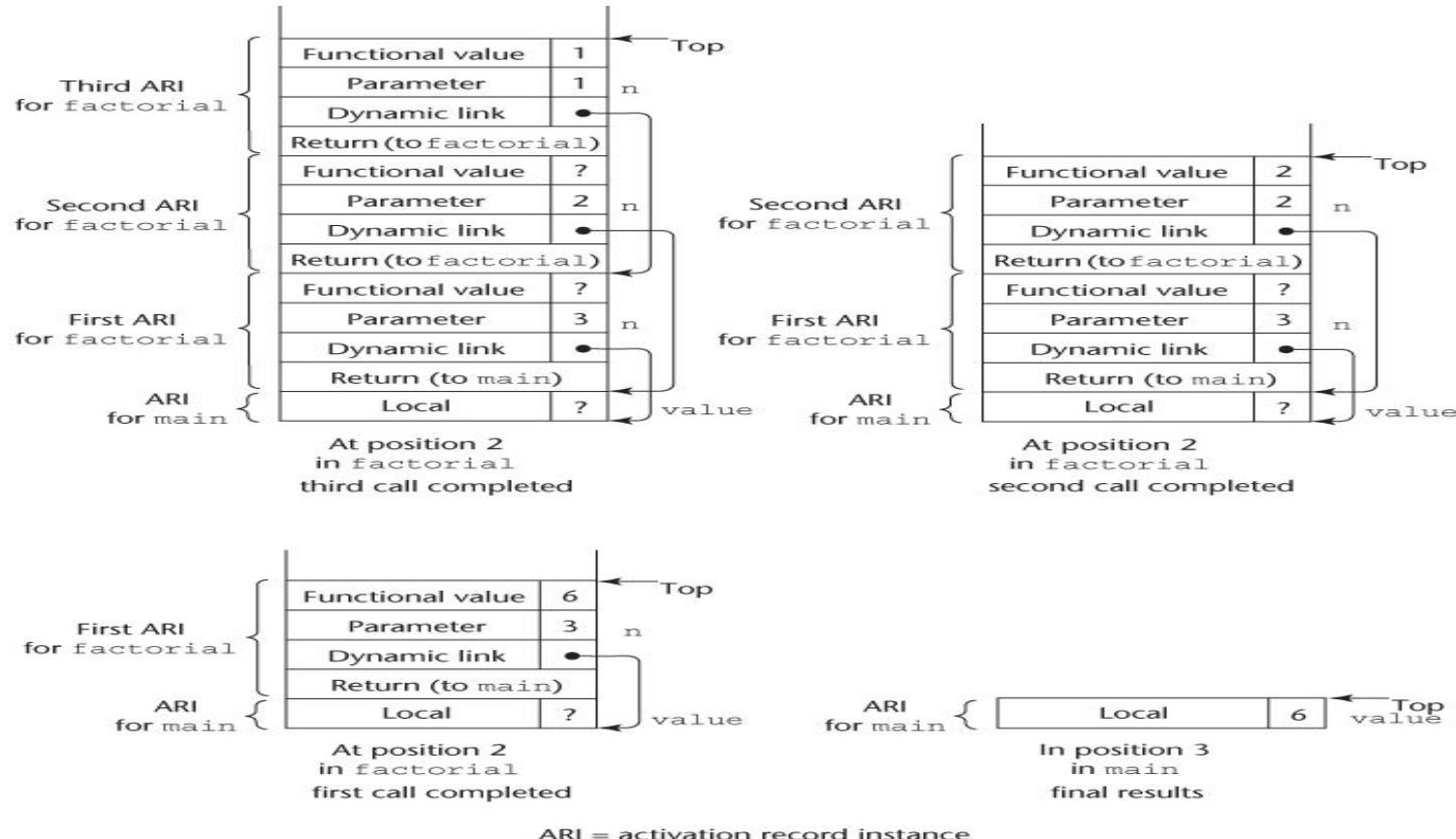


Yığının Faktoriyeli Çağırması



ARI = activation record instance

Faktoriyelin Yığından Dönmesi



İçinde alt programlar

- C- tabanlı olmayan statik bazı diller (Fortran 95+, Ada, Python, JavaScript, Ruby, and Lua. Vb.) yığın dinamik yerel değişkenleri kullanır ve alt programların içe içe geçmesine olağan sağırlar.
- Yerel olmasa da erişilebilen bütün değişkenler yığın içerisinde aktivasyon kayıt örneklerinde durur.
- Yerel olmayan referansları bulma işlemi
 1. doğru aktivasyon kayıtörneğini bulun
 2. ve o aktivasyon kayıtörneğinin içindeki doğru offset'i saptayın.

Yerel olmayan bir referansın yerinin belirlenmesi

- Offsetin bulunması gayet basittir
- Doğru aktivasyon kayıtörneğinin bulunması
- Statik anlam kuralları şunu garanti etmektedir: referanslı ve yerel olmayan bütün değişkenlerin referans yapıldığı anda yiğindaki bazı aktivasyon kayıt örnekleri içine atar.

Statik Ölçek

- Bir statik zincir; belirli aktivasyon kayıt örneklerini bağlayan bir statik bağlantı zinciridir.
- A alt program aktivasyon kayıtörneğindeki statik link, A'nın statik üst ögesinin aktivasyon kayıt örneklerinden birini gösterir.
- Bir aktivasyon kayıtörneğindeki statik zincir onu kendinden önce gelmiş statik değerlere bağlar.
- *Statik_Depth*, değeri o ölçeğin iç içe geçmiş derinliği olan bir statik ölçekle ilişkili tam sayıdır.

Statik Ölçek (devamı)

- Yerel olmayan bir referansın offset zinciri ya da iç derinliği, referansın statik derinliği ile bildirildiğinde ölçeğin derinliği arasındaki farktır.
- Değişkene yapılan bir referans Aşağıdaki şekilde temsil edilebilir:
(zincir ofset ve yerel ofset), bu yerler referans yapılan değişken kayıt aktivasyonundaki offsettir.

Ada Programı Örneği

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
        A := B + C;  <-----1
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
          Sub1;
          E := B + A;  <-----2
        end;  -- of Sub3
      begin -- of Sub2
        Sub3;
        A := D + E;  <-----3
      end;  -- of Sub2 }
    begin -- of Bigsub
      Sub2(7);
    end;  -- of Bigsub
  begin
    Bigsub;
  end; of Main_2 }
```

Ada Programı Örneği (devamı)

- Call sequence for `Main_2`

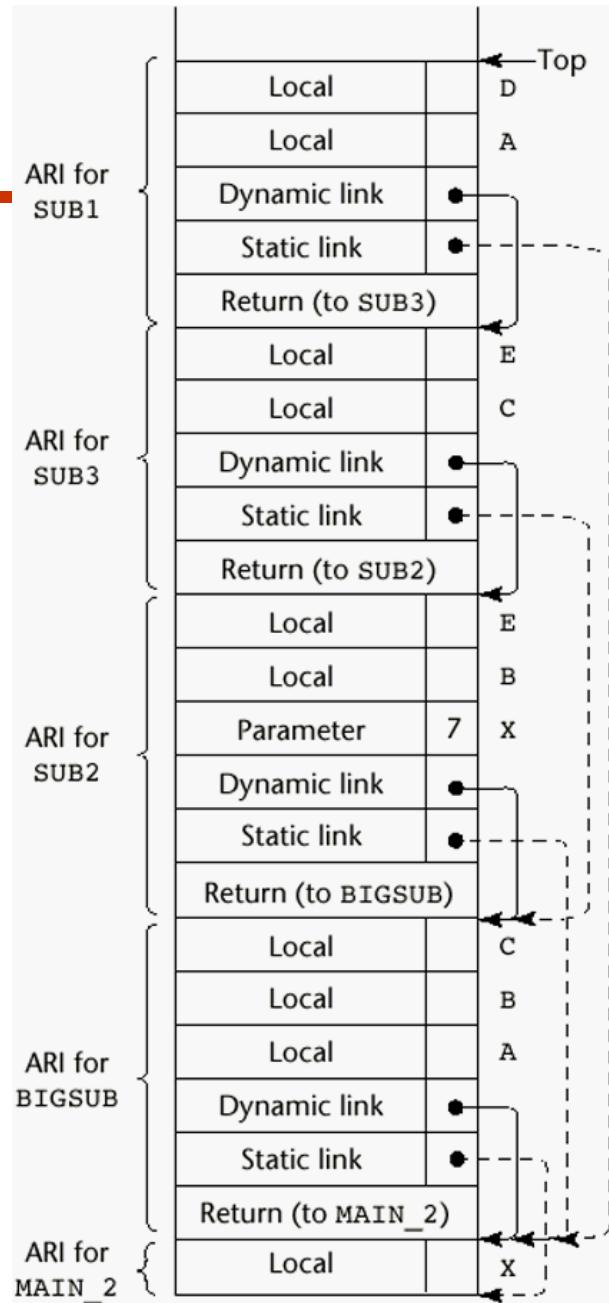
`Main_2` **calls** `Bigsub`

`Bigsub` **calls** `Sub2`

`Sub2` **calls** `Sub3`

`Sub3` **calls** `Sub1`

Pozisyon-1'deki Yığın İçerikleri



Statik zincir bakımı

- Aramada,
 - Aktivasyon kayıt örneği oluşturulmalı
 - Dinamik bağlantı eski yiğin tepe işaretçisidir.
 - Statik bağlantı üst ögenin en son arısını işaret etmelidir.
- İKİ YÖNTEM
 - Dinamik zinciri araştır.
 - Alt program aramaları ve tanımlar ve değişken referanslar gibi tanımları işleme koy.

Statik Zincirlerin Değerlendirilmesi

- Sorunlar:
 1. Yerel olmayan bir referans iç derinlik büyük olursa yavaş olur.
 2. Kritik zaman kodu zordur:
 - a- yerel olmayan referansların maliyetlerinin belirlenmesi zordur.
 - b- kod değişiklikleri iç derinliği ve dolayısı ile maliyetleride değiştirebilir.

Bloklar

- Bloklar, değişkenler için kullanıcı odaklı yerel ölçeklerdir.
- C deki bir örnek

```
{int temp;  
    temp = list [upper];  
    list [upper] = list [lower];  
    list [lower] = temp  
}
```

- Temp in yukarıdaki örnekteki ömrü kontrolün bloku girmesi ile beraber başlar.
- Temp gibi bir yerel değişkeni kullanmanın bir avantajı şudur: aynı isimli başka bir değişkenle çakışmaz

Blokların Uygulanması

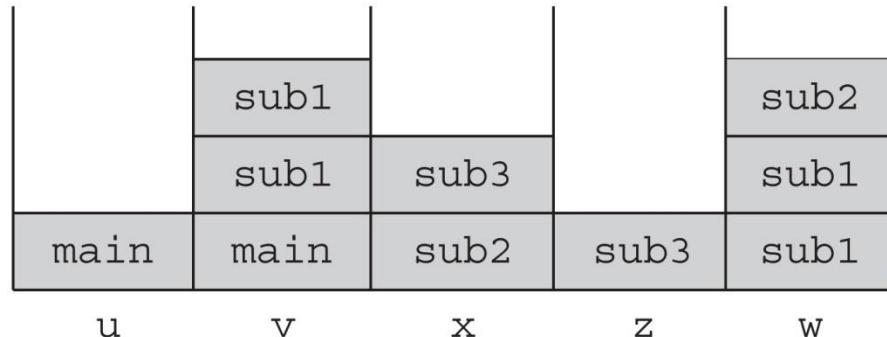
- İKİ YÖNTEM
1. Bloklara daima aynı bölgeden çağrılan alt programlardan ziyade bir parametre olarak davranışın
 - her blokun bir aktivasyon kaydı vardır: blok her çalıştığında bunun bir örneğini oluşturur.
 2. Bir blok için gereken maximum depo statik olarak tespit edilebildiği için, bu gereken alan aktivasyon kaydındaki yerel değişkenlerden sonra ayrılır.

Dinamik ölçeğin Uygulanması

- * *Derin Erişim:* yerel olmayan referanslar dinamik zincirdeki aktivasyon kayıt örneklerinin araştırılmasıyla bulunur.
 - zincirin uzunluğu statik olarak belirlenemez.
 - her aktivasyon kayıt örneği değişken isimlere sahip olmalıdır.
- * *Sığ Erişim:* yerelleri merkezi bir yere koy
 - her bir değişken için bir yığın
 - her bir değişken isim için içinde girdi bulunan merkezi bir çizelge

Dinamik ölçek uygulamasında Sığ erişimin kullanımı

```
void sub3() {  
    int x, z;  
    x = u + v;  
    ...  
}  
  
void sub2() {  
    int w, x;  
    ...  
}  
  
void sub1() {  
    int v, w;  
    ...  
}  
  
void main() {  
    int v, u;  
    ...  
}
```



(The names in the stack cells indicate the program units of the variable declaration.)

Yığın hücrelerindeki isimler
Değişken bildirimlerin
program birimlerini gösterir

Özet

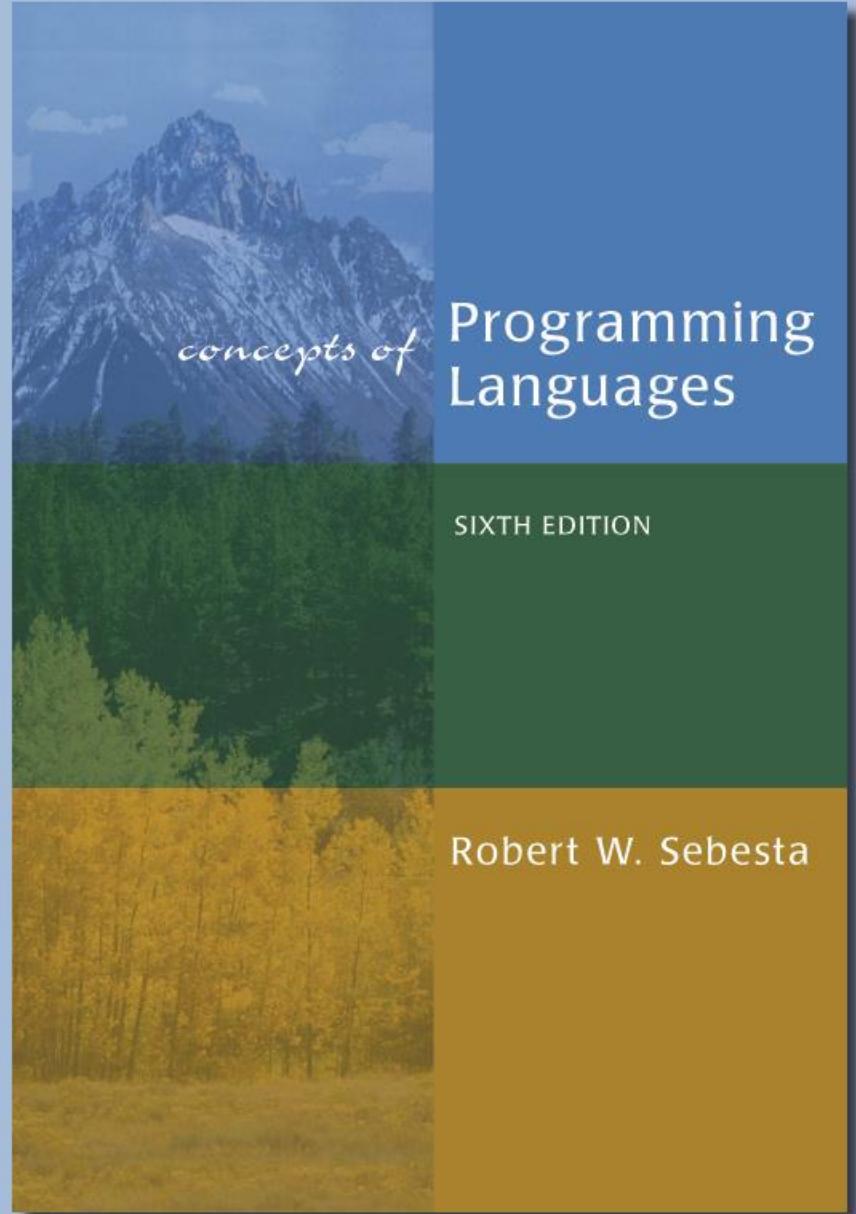
- Alt program bağlantı anımları uygulamalı bir çok eylem gerektirir.
- Basit alt programlar nispeten daha basit eylemler gerektirir.
- Yığın dinamik dilleri daha karmaşıktır.
- Yığın dinamik değişkenli alt programlar ve iç içe geçmiş alt programlar iki bileşene sahiptir:
 - gerçek kod
 - aktivasyon kaydı

Özet (devamı)

- Aktivasyon kayıt örnekleri biçimsel parametreler ve diğerleri arasındaki yerel değişkenleri içerir.
- Statik zincirler iç içe alt programlı statik ölçekli dillerdeki yerel olmayan değişkenlere erişim uygulamasının birincil yöntemidir.
- Dinamik ölçekli dillerdeki yerel olmayan değişkenlere erişim dinamik zincir kullanımı yada bazı merkezi çizelge metodları yolu ile uygulanır

Bölüm 15

Fonksiyonel Programlama Dilleri



Bölüm 15 Topics

- Giriş
- Matematiksel Fonksiyonlar
- Fonksiyonel Programlama Dillerinin (Functional Programming Languages) Temelleri
- İlk Fonksiyonel Programlama Dili: LISP
- Scheme'e Giriş
- COMMON LISP
- ML
- Haskell
- Fonksiyonel dillerin uygulamaları
- Fonksiyonel(Functional) ve Buyurgan(zorunlu-Imperative) Dillerin Karşılaştırılması

Giriş

- Buyurgan(imperative) dillerin tasarıımı doğrudan doğruya von Neumann mimarisine dayanır
 - İlgilenilen başlıca konu dilin yazılım geliştirme için uygunluğudan ziyade verimliliğidir

Giriş

- Fonksiyonel dillerin tasarımı **Matematiksel Fonksiyonlara** dayalıdır
 - Kullanıcıya da yakın olan sağlam bir teorik temel, fakat nispeten programların koşacağı makinelerin mimarileriyle ilgisizdir

Matematiksel Fonksiyonlar

- Tanım: Bir matematiksel fonksiyon, **tanım kümesi(domain set)** adı verilen bir kümənin(set) üyelerinin (members), **değer kümesi(range set)** adı verilen diğer bir kümə ile eşlenmesidir(mapping)
- Bir **lambda ifadesi(lambda expression)** bir fonksiyonun parametresini/parametrelerini ve eşlenmesini(mapping) belirler

$$\lambda(x) \ x * x * x$$

$\text{cube}(x) = x * x * x$ fonksiyonu için.

Matematiksel Fonksiyonlar

- Lambda ifadeleri adsız fonksiyonları tanımlar
- Lambda ifadelerinin parametreye(lere) uygulanması, parametrenin(lerin) ifadenin sonuna getirilmesiyle olur
örn. $(\lambda(x) x * x * x)(3)$
sonuç 27

Fonksiyon Biçimleri

1. Fonksiyon Bileşimi(Function Composition)

- Parametre olarak iki fonksiyon alan ve sonuç olarak, değeri ilk gerçek(actual) parametre fonksiyonun ikincisine uygulanması olan bir fonksiyon veren fonksiyonel form

Form: $h \equiv f \circ g$

su anlama gelir $h(x) \equiv f(g(x))$

$f(x) \equiv x * x * x$ ve $g(x) \equiv x + 3$ için,

$h \equiv f \circ g$ su sonucu verir:

$$(x + 3)^* (x + 3)^* (x + 3)$$

Fonksiyon Biçimleri

2. Yapım(Construction)

- Parametre olarak fonksiyonlardan oluşan bir liste alan ve sonuç olarak her bir parametre fonksiyonunu verilen bir parametreye uygulama sonuçlarının listesini veren fonksiyonel form

Form: $[f, g]$

$f(x) \equiv x * x * x$ ve $g(x) \equiv x + 3$ için,

$[f, g](4)$ in sonucu: $(64, 7)$ dür

Fonksiyon Biçimleri

3. Tümüne uygula(Apply-to-all)

- Parametre olarak bir tek fonksiyon alan ve sonuç olarak parametrelerden oluşan bir listenin her bir elemanına verilen fonksiyonun uygulanmasıyla elde edilen değerlerden oluşan bir liste döndüren fonksiyonel form

Form: α

$$h(x) \equiv x * x * x \text{ için}$$

$\alpha(h, (3, 2, 4))$ in sonucu: (27, 8, 64)



Fonksiyonel Programlama Dillerinin Temelleri

- Bir FPL tasarlamanın amacı matematiksel fonksiyonları mümkün olduğunca taklit etmektir
- Bir FPL deki temel hesaplama işlemi buyurgan(imperative) dildekinden farklıdır
 - Bir buyurgan(imperative) dilde, işlemler yapılır ve sonuçlar daha sonra kullanım için değişkenlerde(variables) tutulur
 - Sürekli ilgilenilen ve buyurgan(imperative) programlamanın karmaşıklık kaynağı olan şey değişkenlerin(variables) yönetimidir
- Bir FPL de, değişkenler(variables), matematikte olduğu gibi gerekli değildir



Fonksiyonel Programlama

Dillerinin Temelleri

- Bir FPL de, bir fonksiyon aynı parametreler verildiğinde daima aynı sonucu üretir
 - Buna *referential transparency* adı verilir

LISP

- Veri nesnesi tipleri(Data object types): atomlar(atoms) ve listeler(lists)
- Liste biçimi(List form): altliste(sublist) ve/veya atomların paranteze alınmış koleksiyonları
örn., (A B (C D) E)
- LISP, tipsiz(typeless) bir dildir
- LISP listeleri, tek-bağılı liste(single-linked lists) olarak saklanır

LISP

- Lambda gösterimi fonksiyonları ve fonksiyon tanımlarını belirtmek için kullanılır. Fonksiyon uygulamaları ve veri aynı biçimde sahiptir.
örn., Eğer (A B C) listesi, veri(data) olarak yorumlanırsa, A, B, ve C diye üç atomdan oluşan basit bir listedir
Eğer bir fonksiyon uygulaması olarak yorumlanırsa, A adındaki fonksiyonun B ve C adındaki iki parametreye uygulanması anlamına gelir
- İlk LISP yorumlayıcısı(interpreter), sadece gösterimin(notation) sayısal hesaplama yeteneklerinin evrenselliğinin ispatı olarak ortaya çıkmıştır

Scheme’ e Giriş

- 1970lerin ortalarında çıkmış bir LISP diyalektidir, çağdaş LISP diyalektlerinin daha temiz, daha modern, ve daha basit bir versiyonudur
- Sadece statik kapsama(static scoping) kullanır
- Fonksiyonlar birinci-sınıf varlıklardır(entities)
 - İfadelerin(expressions) değerleri ve listelerin elemanları olabilirler
 - Değişkenlere(variables) atanabilirler ve parametrelere geçilebilirler

Scheme' e Giriş

- İlkel(Primitive) Fonksiyonlar
 1. Aritmetik: **+**, **-**, *****, **/**, **ABS**,
SQRT, **REMAINDER**, **MIN**, **MAX**
örn., $(+ 5 2)$ nin sonucu: 7

Scheme' e Giriş

2. **QUOTE** -bir parametreyi alır; onu değerlendirmeden geri döndürür
 - **QUOTE** gereklidir çünkü Scheme yorumlayıcısı(interpreter) olan **EVAL**, her zaman fonksiyonu uygulamadan önce parametreleri fonksiyon uygulamalarında değerlendirir.
QUOTE, parametreler uygun olmadığı zaman onların değerlendirilmesini önlemek için kullanılır
 - **QUOTE** , kesme işareti(apostrophe) önek(prefix) operatörüyle kısaltılabilir
örn., '(A B) şuna eşittir: (**QUOTE** (A B))

Scheme' e Giriş

3. **CAR** bir parametre listesini alır; o listenin ilk elemanını döndürür
 - örn., (CAR '(A B C)) sonucu: A
 - (CAR '((A B) C D)) sonucu: (A B)
4. **CDR** bir parametre listesini alır; ilk elemanını kopardıktan sonra listenin kalanını döndürür
 - örn., (CDR '(A B C)) sonucu: (B C)
 - (CDR '((A B) C D)) sonucu: (C D)

Scheme' e Giriş

5. **CONS** iki parametre alır, bunların birincisi bir atom veya bir liste olabilir ve ikincisi bir listedir; sonuç olarak ilk elemanı olarak birinci parametreyi, sonucunun kalanı olarak da ikinci parametreyi içeren yeni bir liste döndürür
örn., (CONS 'A '(B C)) sonuç: (A B C)

Scheme' e Giriş

6. **LIST** – herhangi bir sayıda parametre alır; elemanları bu parametreler olan bir liste döndürür

Scheme' e Giriş

- Lambda İfadeleri(Expressions)
 - Biçimi λ gösterimine(notation) dayalıdır
örn., (LAMBDA (L) (CAR (CAR L)))
L ‘ye bağlı değişken(bound variable) denir
- Lambda ifadeleri şu şekilde uygulanabilir
örn.,
((LAMBDA (L) (CAR (CAR L))) '((A B) C D))

Scheme' e Giriş

- Fonksiyonları oluşturmak için bir fonksiyon:
DEFINE – İki biçimdedir:
 1. Bir sembolü(symbol) bir ifadeye(expression) bağlama
örn.,
(**DEFINE** pi 3.141593)
(**DEFINE** two_pi (* 2 pi))

Scheme' e Giriş

2. Adları(names) lambda ifadelerine bağlama
örn.,

(DEFINE (cube x) (* x x x))

örnek kullanım:

(cube 4)

Scheme' e Giriş

- Değerlendirme işlemi (normal fonksiyonlar için):
 1. Parametreler belli bir sıraya bağlı olmadan değerlendirilir
 2. Parametrelerin değerleri fonksiyon gövdesinde yerine konur
 3. Fonksiyon gövdesi değerlendirilir
 4. Gövdedeki son ifadenin değeri fonksiyonun değeridir
(Özel biçimler farklı bir değerlendirme işlemi kullanır)

Scheme' e Giriş

- örnekler:

```
(DEFINE (square x) (* x x))
```

```
(DEFINE (hypotenuse side1 side2)
        (SQRT (+ (square side1)
                  (square side2))))
```

)

Scheme' e Giriş

- Predicate(**Hüküm**) Fonksiyonlar: (#T true ve () false)
 1. **EQ?** İki sembolik parametre alır; eğer her ikisi de atomsa ve aynı ise #T döndürür
örn., (**EQ?** 'A 'A) sonuç: #T
(**EQ?** 'A ' (A B)) sonuç: ()
 - Dikkat edilmelidir ki eğer **EQ?** liste parametrelerle çağrılsa, sonuç güvenilir olmaz
 - Bir de, **EQ?** sayısal(numeric) atomlar için çalışmaz

Scheme' e Giriş

- Predicate Fonksiyonlar:
 2. **LIST?** Bir parametre alır; parametre bir liste ise **#T**; değilse () döndürür
 3. **NULL?** Bir parametre alır; parametre bir boş(empty) liste ise **#T**; değilse () döndürür
- Dikkat edilmelidir ki **NULL?** , eğer parametre () ise **#T** döndürür
- 4. Sayısal Hüküm(Numeric Predicate) Fonksiyonları
=, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?, NEGATIVE?

Scheme' e Giriş

- Çıktı Yardımcı(Output Utility) Fonksiyonları:
(DISPLAY expression)
(NEWLINE)

Scheme' e Giriş

- Kontrol akışı

1. Seçim(Selection)- özel biçim, **IF**

**(IF predicate then_exp
else_exp)**

örn.,

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

Scheme' e Giriş

- Kontrol akışı
 - 2. Çoklu Seçim – özel biçim, **COND**
 - Genel biçim:
 - (COND
 - (**predicate_1** **expr** {**expr**})
 - (**predicate_2** **expr** {**expr**})
 - ...
 - (**predicate_n** **expr** {**expr**})
 - (**ELSE** **expr** {**expr**})
 -)

En sondaki expr nin değerini hükmü(predicate) true değeri veren birinci çiftte(pair) döndürür

COND Örneği

```
(DEFINE (compare x y)
  (COND
    ((> x y) (DISPLAY "x, y' den
                         büyüktür"))
    ((< x y) (DISPLAY "y, x' ten
                         büyüktür"))
    (ELSE (DISPLAY "x ve y
                     eşittir")) )
  )
```

Örnek Scheme Fonksiyonları

1. **member** - bir atom ve bir basit liste alır; eğer atom listede varsa #T; yoksa () döndürür

```
(DEFINE (member atm lis)
  (COND
    ((NULL? lis) '())
    ((EQ? atm (CAR lis)) #T)
    ((ELSE (member atm (CDR lis))))
  ))
```

Example Scheme Fonksiyonlar

2. **equalsimp** – parametre olarak iki basit liste alır; iki liste birbirine eşitse #T; değilde () döndürür

```
(DEFINE (equalsimp lis1 lis2)
  (COND
    ( (NULL? lis1) (NULL? lis2) )
    ( (NULL? lis2) '())
    ( (EQ? (CAR lis1) (CAR lis2))
      (equalsimp (CDR lis1) (CDR lis2)) )
    (ELSE '())
  )))

```

Example Scheme Fonksiyonlar

3. **equal** – parametre olarak iki genel liste alır; iki liste birbirine eşitse #T; değilse () döndürür

```
(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (EQ? lis1 lis2))
    ((NOT (LIST? lis2)) '())
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) '())
    ((equal (CAR lis1) (CAR lis2))
     (equal (CDR lis1) (CDR lis2)))
    (ELSE '()))
  ))
```

Example Scheme Fonksiyonlar

4. **append** - parametre olarak iki liste alır; birinci parametre listesinin sonuna ikinci parametre listesinin elemanlarını ekleyerek geri döndürür

```
(DEFINE (append lis1 lis2)
  (COND
    ( (NULL? lis1) lis2)
    (ELSE (CONS (CAR lis1)
                (append (CDR lis1) lis2))))
```

))

Scheme' e Giriş

- **LET** fonksiyonu

- Genel biçim:

```
(LET (  
      (name_1 expression_1)  
      (name_2 expression_2)  
      ...  
      (name_n expression_n))  
  body  
)
```

- Semantik(Semantics): bütün ifadeleri değerlendirir, sonra değerleri(values) adlara(names) bağla; gövdeyi(body) değerlendir

Scheme' e Giriş

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
      (/ (SQRT (- (* b b) (* 4 a c)))
          (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))))

  (DISPLAY (+ minus_b_over_2a
              root_part_over_2a))
  (NEWLINE)
  (DISPLAY (- minus_b_over_2a
              root_part_over_2a))
  ))
```

Scheme' e Giriş

- Fonksiyonel Biçimler
 - 1. Bileşim(Composition)
 - Önceki örnekler bunu kullandı
 - 2. Tümüne Uygula(Apply to All) - Scheme de bir biçim **mapcar**
 - Verilen fonksiyonu verilen bir listenin bütün elemanlarına uygular; sonuç, sonuçlardan oluşan bir listedir

```
(DEFINE (mapcar fun lis)
  (COND
    ( (NULL? lis) '())
    (ELSE (CONS (fun (CAR lis))
                (mapcar fun (CDR lis))))))
  ))
```

Scheme' e Giriş

- Scheme'de, Scheme kodunu oluşturan ve yorumlanması(interpretation) isteyen bir fonksiyon tanımlamak mümkündür
- Bu mümkündür çünkü yorumlayıcı(interpreter) kullanıcı tarafından erişilebilen(user-available) bir fonksiyondur, **EVAL**
- örn., farzedelim ki birbiriyle toplanması gereken sayılardan oluşan bir listemiz var

```
(DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (+ (CAR lis)
              (adder (CDR lis )))))
  ))
```

Bir Listedeki Sayıları Toplama

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

- Parametre, eklenmesi gereken sayılardan oluşan bir listedir; **adder** araya bir + operatörü ekler ve sonuçtaki listeyi yorumlar

Scheme' e Giriş

- Scheme bazı buyurgan(zorunlu-imperative) özellikler içerir:
 1. **SET!** bir değeri(value) bir ada(name) bağlar(bind) veya yeniden bağlar(rebind)
 2. **SET-CAR!** bir listenin **car** 'ını değiştirir
 3. **SET-CDR!** bir listenin **cdr** kısmını değiştirir

COMMON LISP

- LISP'in popüler diyalektlerine ait çoğu özelliklerin 1980lerin başlarında ortaya çıkmış bir birleşimidir
- Büyük ve karmaşık bir dildir– Scheme'nin tersi

COMMON LISP

- İçeriği:
 - kayıtlar(records)
 - diziler(arrays)
 - karmaşık sayılar(complex numbers)
 - karakter stringleri(character strings)
 - güçlü I/O yetenekleri
 - erişim kontrollü(access control) paketler
 - Scheme'deki gibi buyurgan(imperative) özellikler
 - yinelenen(iterative) kontrol ifadeleri

COMMON LISP

- Örnek (yinelenen küme üyeliği, member (üye))

```
(DEFUN iterative_member (atm lst)
  (PROG ()
    loop_1
    (COND
      ((NULL lst) (RETURN NIL))
      ((EQUAL atm (CAR lst)) (RETURN T))
      )
    (SETQ lst (CDR lst))
    (GO loop_1)
  ))
```

ML

- Sentaksı LISP'den daha çok Pascala yakın olan bir statik-kapsamlı(static-scoped) fonksiyonel dildir
- Tip tanımlamaları(type declarations) kullanır, fakat aynı zamanda tanımsız(undeclared) değişkenlerin tiplerini belirlemek için **tip çıkarma(type inferencing)** kullanır (Bölüm 5 de anlatılacak)
- **Strongly typed**'tır (Scheme temelde tipsizdir(typeless)) ve tip baskısı(type coercions) yoktur
- İstisna yakalama(exception handling) ve soyut veri tipleri(abstract data types) oluşturmak için bir modül özelliği içerir

ML

- Listeler ve liste işlemleri içerir
- **val** ifadesi bir adı bir değere(value) bağlar (Scheme'deki **DEFINE**'a çok benzer)
- Fonksiyon tanımlama biçimi:
fun fonksiyon_adı (formal_parametreler) =
fonksiyon_govdesi_ifadesi;
örn.,
fun cube (x : int) = x * x * x;

Haskell

- ML'e benzer (sentaks, statik kapsamlı(static scoped), strongly typed, type inferencing)
- ML den(ve çoğu diğer fonksiyonel dillerden) farklıdır çünkü **tamamen(purely)** fonksiyoneldir (örn., değişkenler yoktur, atama ifadeleri yoktur, hiçbir çeşit yan etki yoktur)

Haskell

- En önemli özellikler
 - Tembel değerlendirme([lazy evaluation](#)) kullanır (değer gerekmediği sürece hiçbir alt-ifadeyi değerlendirmeye)
 - Liste kapsamları([list comprehensions](#)), sonsuz listelerle çalışabilmeye izin verir

Haskell örnekleri

1. Fibonacci sayıları (fonksiyon tanımlarını farklı parametre biçimleriyle gösterir)

```
fib 0 = 1
```

```
fib 1 = 1
```

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

Haskell örnekleri

2. Faktoriyel(Factorial) (guardları gösterir)

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

Özel kelime **otherwise** guard olarak
görünebilir

Haskell örnekleri

3. Liste işlemleri

- Liste gösterimi(notation): elemanları köşeli parantez içine yaz

örn., **directions** = **[“north”, “south”, “east”, “west”]**

- Uzunluk(Length): #

örn., **#directions is 4**

- .. operatorü ile aritmetik seriler

örn., **[2, 4 .. 10] is [2, 4, 6, 8, 10]**

Haskell örnekleri

3. Liste işlemleri(devam)

- ++ ile zincirleme(Catenation)

örn., [1, 3] ++ [5, 7] şu sonucu verir:

[1, 3, 5, 7]

- İki nokta(colon) operatörü yoluyla CONS, CAR, CDR (Prolog'daki gibi)

örn., 1 : [3, 5, 7] nin sonucu:

[1, 3, 5, 7]

Haskell örnekleri

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

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

Haskell örnekleri

4. Liste kapsamları(comprehensions): küme gösterimi(set notation)

örn., `[n * n | n ← [1..20]]`

ilk 20 pozitif tamsayının karelerinden oluşan bir liste tanımlar

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

Bu fonksiyon verilen parametrenin bütün faktörlerini(çarpan) hesaplar

Haskell örnekleri

- Quicksort(Hızlı Sıralama):

```
sort [] = []
sort (a:x) = sort [b | b ← x; b
                     <= a]
              ++
            [a] ++
sort [b | b ← x; b > a]
```

Haskell örnekleri

5. Tembel değerlendirme(Lazy evaluation)

örn.,

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

```
squares = [n * n | n ← [0..]]
```

(sadece gerekli olanları hesapla)

örn., **member squares 16**

True döndürür

Haskell örnekleri

- Üye(member) şu şekilde de yazılabilirdi:

```
member [] b = False
```

```
member (a:x) b=(a == b) || member x b
```

- Ancak, bu sadece kare(square) olan parametre tamkare olduğu zaman çalışacaktı; eğer değilse, sonsuza kadar üretmeye devam edecekti. Şu versiyon her zaman çalışır:

```
member2 (m:x) n
```

```
  | m < n= member2 x n
```

```
  | m == n      = True
```

```
  | otherwise = False
```

Fonksiyonel Dillerin uygulamaları

- APL atılan(throw-away) programlar için kullanılır
- LISP yapay zeka(artificial intelligence) için kullanılır
 - Bilgi gösterimi
 - Makine öğrenmesi(Machine learning)
 - Doğal Dil İşleme(Natural language processing)
 - Konuşma ve görmeyi modelleme
- Scheme birçok üniversitede programlamayı öğretmeye giriş için kullanılır

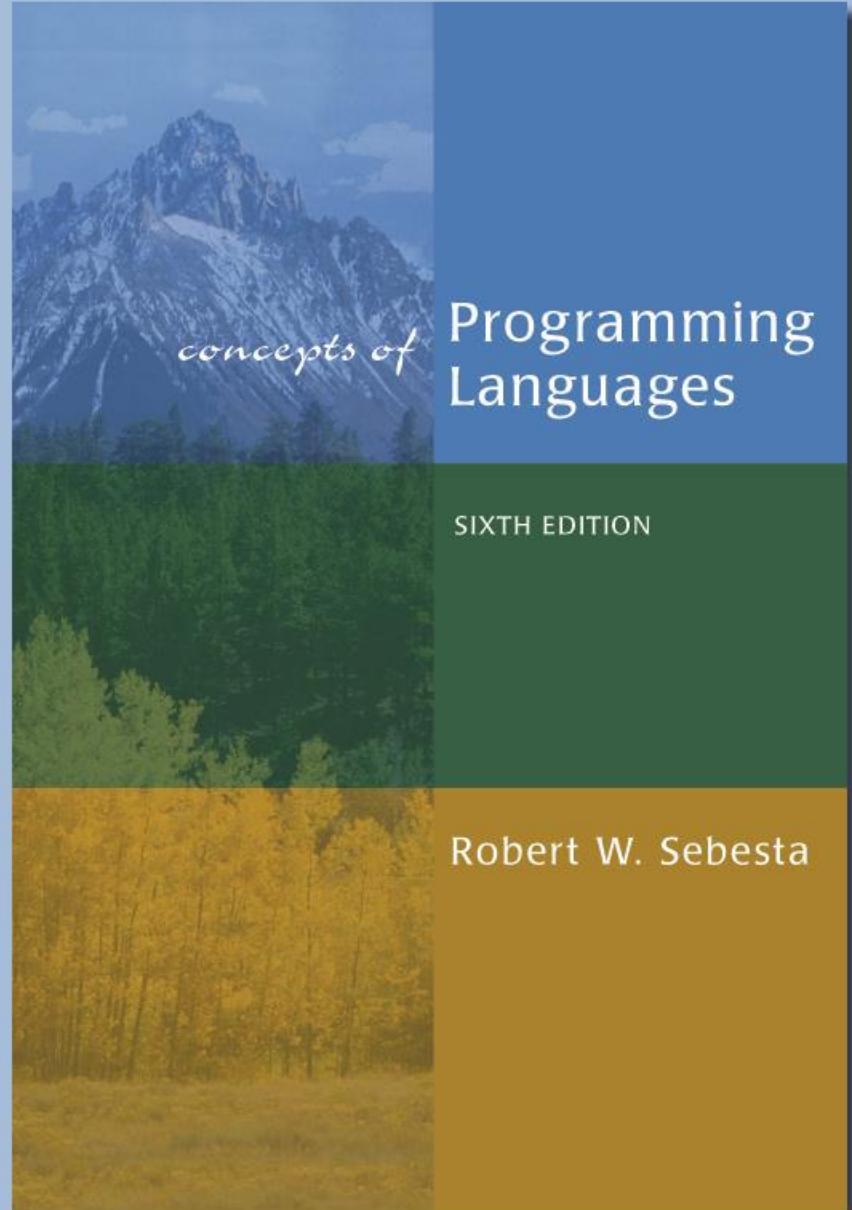


Fonksiyonel ve Buyurgan(imperative) Dillerin Karşılaştırılması

- buyurgan(imperative) diller:
 - Verimli çalışma
 - Karmaşık semantik(semantics)
 - Karmaşık sentaks(syntax)
 - Eşzamanlılık(Concurrency) programcı tarafından tasarlanır
- fonksiyonel diller:
 - Basit semantik(semantics)
 - Basit sentaks(syntax)
 - Verimsiz çalışma
 - Programlar otomatik olarak eşzamanlı(concurrent) yapılabilir

Bölüm 16

Mantık(Lojik-Logic)
Programlama Dilleri



Bölüm 16 Topics

- Giriş
- **Hüküm Hesabına(Predicate Calculus)** Kısa bir Giriş
- **Hüküm Hesabı(Predicate Calculus)** ve Teorem İspatlama
- Mantık Programlamaya Genel Bakış
- Prolog'un Kökenleri
- Prolog'un temel elemanları
- Prolog'un eksikleri
- Mantık programlama uygulamaları

Giriş

- Mantık(Logic) programlama dili veya tanıtıcı(declarative) programlama dili
- Programları sembolik mantık biçiminde ifade etme
- Sonuçları üretmek için mantıksal çıkarsama (logical inferencing) işlemi kullanma
- Yordamsal(procedural) yerine Tanıtıcı(Declarative) :
 - Sadece sonuçların(results) özelliği belirtilir (onları üreten detaylı prosedürlerin(yordamların)(procedures) değil)

Predicate Calculus 'a Giriş

- Önerme(Proposition): doğru olan veya olmayan bir mantıksal ifade
 - Nesneler(objects) ve bunların birbiri arasındaki ilişkilerini(relationships) içerir

Predicate Calculus 'a Giriş

- Sembolik Mantık(Symbolic logic) biçimsel mantığın(formal logic) temel ihtiyaçları kullanılabılır :
 - Önermeleri(propositions) ifade etme
 - Önermeler arasındaki ilişkileri(relationships) ifade etme
 - Diğer önermelerden nasıl yeni önermeler çıkarılabileceğini anlatma
- Sembolik mantığın mantık programlama için kullanılan belli bir biçimine predicate calculus adı verilir

Önermeler(Propositions)

- Önermelerdeki nesneler basit terimlerle ifade edilir: sabitler(constants) veya değişkenler(variables)
- **Sabit(Constant)**: bir nesneyi gösteren bir sembol(symbol)
- **Değişken(Variable)**: farklı zamanlarda farklı nesneleri gösterebilen bir sembol(symbol)
 - Buyurgan dillerdeki(imperative languages) değişkenlerden(variables) farklıdır

Önermeler(Propositions)

- Atomik Önermeler(Atomic propositions) bileşik terimlerden(compound terms) oluşur
- Bileşik Terim(Compound term): matematiksel fonksiyon gibi yazılan, matematiksel ilişkinin bir elemanı
 - Matematiksel fonksiyon bir eşlemedir(mapping)
 - Bir tablo olarak yazılabılır

Önermeler(Propositions)

- Bileşik terim(Compound term) iki kısımdan oluşur
 - Functor: ilişkiyi(relationship) adlandıran fonksiyon sembolü(symbol)
 - Parametrelerin sıralı listesi (tuple)
- Örnekler:
 - student(jon)
 - like(seth, OSX)
 - like(nick, windows)
 - like(jim, linux)

Önermeler(Propositions)

- Önermeler iki biçimde belirtilir:
 - Gerçek(Fact): doğru olduğu varsayılan önerme
 - Sorgu(Query): önermenin doğruluğuna karar verilir
- Bileşik Önerme(Compound proposition):
 - İki veya daha fazla atomik önerme içerir
 - Önermeler operatörlerle bağlanır

Mantıksal Operatörler

Adı	Sembol	Örnek	Anlamı
Negation (olumsuzluk)	\neg	$\neg a$	a'nın değil
Conjunction (birleşme ve ile)	\cap	$a \cap b$	a ve b
disjunction (ayırılma veya ile)	\cup	$a \cup b$	a veya b
equivalence (eşitlik)	\equiv	$a \equiv b$	a eşittir b
Implication (icerme)	\supset	$a \supset b$ $a \subset b$	a ,b yi içerir b ,a yi içerir

Niceleyiciler(Quantifiers)

adı	örnek	anlamı
Universal (evrensel)	$\forall X.P$	her X için, P doğrudur
Existential (varoluşsal)	$\exists X.P$	P nin doğru değeri için bir X değeri vardır

Cümlesel Biçim(Clausal Form)

- Aynı şeyi belirtmek için çok fazla yol
- Önermeler için standart bir form kullan
- Cümlesel Biçim(Clausal form):

$$B_1 \cup B_2 \cup \dots \cup B_n \subset A_1 \cap A_2 \cap \dots \cap A_m$$

şu anlama gelir : eğer bütün A lar doğru ise, o zaman en az bir B doğrudur

- Önceki(Antecedent): sağ taraf
- Sonuç(Consequent): sol taraf



Predicate Calculus ve Teorem(theorems) ispatlama

- Önermelerin (propositions) bir kullanımı bilinen aksiyomlardan(axioms) ve teoremlerden(theorems) çıkarılabilen yeni teoremler keşfetmektir
- Çözüm(Resolution): verilen önermelerden çıkarılmış önermelerin(inferred propositions) hesaplanmasına imkan veren bir çıkarım prensibi (inference principle)

Çözüm(Resolution)

- Birleştirme(Unification): eşlenme(matching) işleminin başarılı olması için önermelerdeki(propositions) değişkenler(variables) için değerler(values) bulma
- Başlatma(Instantiation): birleştirmenin(unification) başarılı olması için değişkenlere(variables) geçici değerler atama
- Bir değişkeni(variable) bir değerle başlattıktan sonra, eğer eşlenme(matching) başarısız olursa, geri-izleme(backtrack) ve farklı bir değerle yeniden başlatma yapmaya gereksinim duyabilir

Teorem İspatlama(Theorem Proving)

- Çelişki(contradiction) ile kanıt (proof) kullanır
- Hipotez(Hypotheses): bir geçerli önermeler (pertinent propositions) kümesi
- Hedef(Goal): teoremin(theorem) olumsuzluğu(negation) önerme(proposition) olarak belirtilir
- Bir tutarsızlık(inconsistency) bulunarak teorem (theorem) ispatlanır

Teorem İspatlama

- Mantık(logic) programlamanın temeli
- Önermeler(propositions) çözüm(resolution) için kullanıldığı zaman, sadece kısıtlanmış(restricted) biçim kullanılabilir
- Horn clause – sadece iki biçimini olabilir
 - Headed: sol kısmında basit atomik önerme
 - Headless: boş sol kısm (gerçek(fact)leri belirtmek için kullanılır)
- Çoğu önermeler Horn clause olarak belirtilebilir



Mantık programlamaya genel bakış

- Tanıtıcı semantik(Declarative semantics)
 - Her bir ifadenin anlamını belirlemek için basit bir yol vardır
 - Buyurgan dillerin sematiğinden daha basittir
- Programlama yordamsal değildir(nonprocedural)
 - Programlar hesaplanan bir sonuç belirtmez, fakat sonucun biçimini belirtir

Örnek: bir listeyi sıralama

- Bir sıralı listenin özelliğini tanımlamak, listeyi yeniden düzenleme işlemi değildir

$\text{sort}(\text{old_list}, \text{new_list}) \subset \text{permute } (\text{old_list}, \text{new_list}) \cap \text{sorted } (\text{new_list})$

$\text{sorted } (\text{list}) \subset \forall_j \text{ such that } 1 \leq j < n, \text{list}(j) \leq \text{list } (j+1)$

Prolog'un esasları

- University of Aix-Marseille
 - Doğal Dil İşleme(Natural language processing)
- University of Edinburgh
 - Otomatik Teorem İspatlama(Automated theorem proving)

Prolog'un temel elemanları

- Edinburgh Syntax
- Terim(Term): bir sabit(constant), değişken(variable), veya yapı(structure)
- Sabit(Constant): bir atom veya bir tamsayı(integer)
- Atom: Prolog'un sembolik değeri
- Atom şunlardan birinden oluşur:
 - Küçük harfle başlayan harfler(letters), rakamlar(digits), ve alt-tirelerden(underscores) oluşan bir string
 - Kesme işaretleriyle (apostrophes) yazdırılabilir ASCII karakterlerinden oluşan bir string

Prolog'un temel elemanları

- Değişken(Variable): büyük harfle başlayan, harfler(letters), rakamlar(digits), ve alt-tirelerden (underscores) oluşan herhangi bir string
- Başlatma(Instantiation): bir değişkenin bir değere bağlanması
 - Sadece bir hedefe tamamen ulaşana kadar sürer
- Yapı(Structure): atomik önerme functor(parametre listesi)'ı gösterir

Gerçek İfadeleri(Fact Statements)

- Hipotezler(hypotheses) için kullanılır
- Headless Horn cümleleri

student(jonathan).

sophomore(ben).

brother(tyler, cj).

Kural ifadeleri(Rule Statements)

- Hipotezler(hypotheses) için kullanılır
- Headed Horn cümlesi
- Sağ kısım: önceki(antecedent) (*if* kısmı)
 - Basit terim veya birleşme(conjunction) olabilir
- Sol kısım: sonuç(consequent) (*then* kısmı)
 - Basit terim olmalıdır
- Birleşme(Conjunction): mantıksal AND işlemleriyle ayrılmış çoklu terimler(multiple terms)

Kural ifadeleri(Rule Statements)

```
parent(kim,kathy) :- mother(kim,kathy).
```

- Anlamı genelleştirmek için değişkenler (evrensel nesneler-universal objects) kullanabilir:

```
parent(X,Y) :- mother(X,Y).  
sibling(X,Y) :- mother(M,X),  
                mother(M,Y),  
                father(F,X),  
                father(F,Y).
```

Hedef İfadeleri(Goal Statements)

- Teorem ispatlama için, teorem sistemin ispat etmesini veya etmemesini istediğimiz önermenin biçimindedir – hedef ifadesi(goal statement)
- headless Horn daki aynı biçim
student(james)
- Bileşik önermeler(Conjunctive propositions) ve değişkenli önermeler de geçerli hedeflerdir
father(x, joe)

Prolog'un Çıkarsama İşlemi(Inferring Process)

- Sorgulara(Queries) hedef(goals) denir
- Eğer bir hedef(goal) bir bileşik ifade ise(compound proposition), her bir gerçek(facts) bir alt-hedefdir (subgoal)
- bir hedefin(goal) doğruluğunu(true) ispatlamak için, çıkarım kuralları(inference rules) ve/veya gerçeklerden(facts) oluşan bir zincir bulmalıdır.

Hedef(goal) Q için:

B :- A

C :- B

...

Q :- P

- Althedefi ispatlama işlemine eşleme(matching), sağlama(satisfying), veya çözüm(resolution) adı verilir

Çıkarsama işlemi(Inferring Process)

- Aşağıdan-yukarıya çözüm, ileri zincirleme(Bottom-up resolution, forward chaining)
 - Gerçekler(facts) ve veritabanı(database) kurallarıyla (rules) başlar ve hedefe(goal) ulaştıracak sırayı bulmaya çalışır
 - Geniş bir olası doğru cevaplar kümesiyle iyi çalışır
- Yukarıdan-aşağıya çözüm, geri zincirleme(Top-down resolution, backward chaining)
 - Hedef ile başlar ve veritabanındaki gerçekler(facts) kümesine ulaşırıan sırayı(sequence) bulmaya çalışır
 - Küçük bir olası doğru cevaplar kümesiyle iyi çalışır
- Prolog implementasyonları geri zincirleme(backward chaining) kullanır

Çıkarsama işlemi(Inferring Process)

- Hedef birden fazla alt hedefe sahipse, şunlardan birini kullanır
 - Depth-first arama: diğerleriyle çalışmadan önce ilk althedefin tamamen ispatını bulmak
 - Breadth-first arama: bütün alt hedefler üzerinde paralel çalışma
- Prolog depth-first arama kullanır
 - Daha az bilgisayar kaynağıyla yapılabilir

Çıkarsama işlemi(Inferring Process)

- Birden çok alt-hedefi(subgoal) bulunan bir hedef(goal) ile, eğer althedeflerden birinin doğruluğunu göstermekte başarısız olunursa, alternatif bir çözüm için bir önceki althedef yeniden ele alınır: geri-izleme(backtracking)
- Bir önceki aramanın bıraktığı yerden aramaya başlanır
- Çok fazla zaman ve alan alabilir çünkü her bir alt-hedefin(subgoal) olası bütün ispatlarını bulabilir

Basit Aritmetik

- Prolog tamsayı değişkenlerini(integer variables) ve tamsayı aritmetiğini destekler
- *is* operatörü: sağ işlenen(operand) olarak bir aritmetik ifadeyi ve sol işlenen(operand) olarak da değişkeni(variable) alır
- $A \text{ is } B / 10 + C$
- Atama ifadesi(assignment statement) ile aynı değildir!

Örnek

```
speed(ford,100).  
speed(chevy,105).  
speed(dodge,95).  
speed(volvo,80).  
time(ford,20).  
time(chevy,21).  
time(dodge,24).  
time(volvo,24).  
distance(X,Y) :- speed(X,Speed),  
                  time(X,Time),  
                  Y is Speed * Time.
```

İzleme(Trace)

- Her adımdaki başlatmaları(instantiations) gösteren yerleşik yapı(built-in structure)
- Yürütmeyen **İzleme Modeli**(Tracing model of execution)- dört olay:
 - Çağırma(Call) (hedefi(goal) gerçekleştirme çabasının başlangıcı)
 - Çıkış(Exit) (hedef gerçekleştirilmiş olunca)
 - Yinele(Redo) (geriizleme(backtrack) olur)
 - Başarısız(Fail) (hedef başarısız olduğunda)

Örnek

likes(jake, chocolate).

likes(jake, apricots).

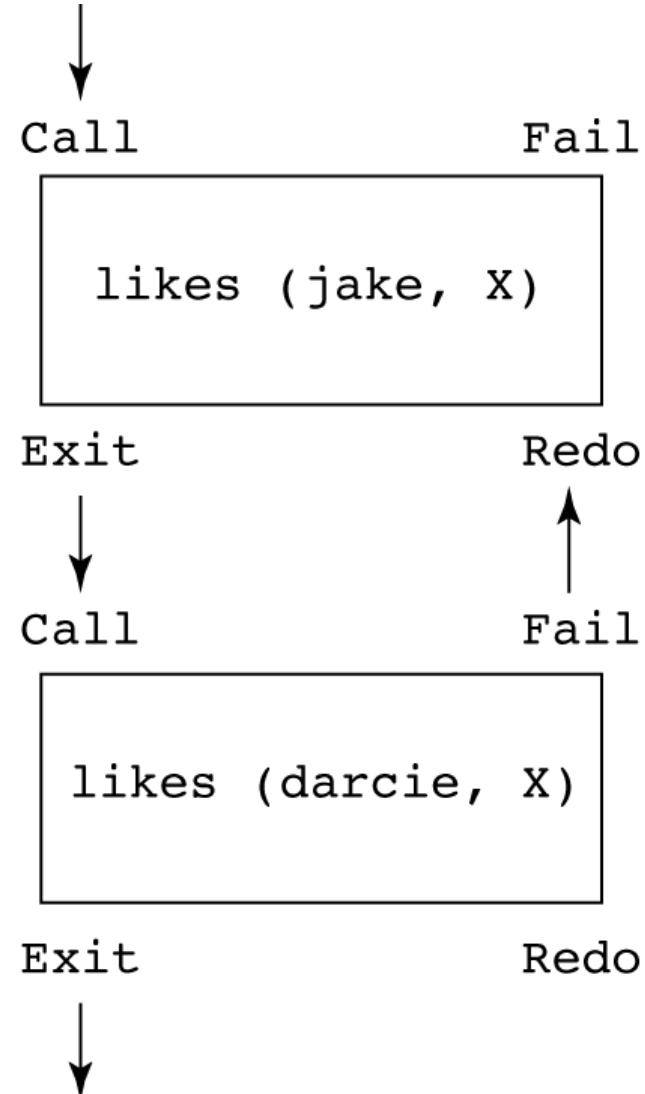
likes(darcie, licorice).

likes(darcie, apricots).

trace.

likes(jake, X),

likes(darcie, X).



Liste yapıları(List Structures)

- Diğer temel veri yapısı(data structure) (daha önce gördüğümüz atomik önermelere(propositions) ek olarak): liste
- Liste herhangi bir sayıdaki elemanlar(elements) sırasıdır
- Elemanlar atomlar, atomik önermeler(propositions), veya diğer terimler (diğer listeler de dahil) olabilir

[apple, prune, grape, kumquat]

[] (boş liste)

[x | Y] (**baş** (head) X ve kuyruk(tail) Y)

Örnek

- append fonksiyonunun tanımı:

```
append([], List, List).
```

```
append([Head | List_1], List_2, [Head  
| List_3]) :-
```

```
    append(List_1, List_2, List_3).
```

Örnek

- reverse fonksiyonunun tanımı:

```
reverse([], []).
```

```
reverse([Head | Tail], List) :-  
    reverse (Tail, Result) ,  
    append (Result, [Head] , List) .
```

Prolog'un eksiklikleri(Deficiencies)

- Çözüm (Resolution) sırası kontrolü
- Kapalı-çevre varsayıımı (closed-world assumption)
- Değilini alma (negation) problemi
- Yerleşik kısıtlamalar(Intrinsic limitations)

Mantık programlama uygulamaları

- İlişkisel veritabanı yönetim sistemleri(Relational database management systems)
- Exper Sistemleri(Expert systems)
- Doğal Dil işleme(Natural language processing)
- Eğitim(Education)

Sonuçlar

- Avantajlar:
 - Prolog programlar mantığa dayalıdır, bu yüzden daha mantıksal düzenlenebilir ve yazılabilir
 - İşleme doğal olarak paraleldir, bu yüzden Prolog yorumlayıcıları(interpreters) çoklu-işlemcili makine avantajını kullanabilirler
 - Programla kısa ve özdür, bu yüzden geliştirme süresi azalmıştır– prototipleme(ilk-ürün oluşturma-prototyping) için iyidir