CENG 218 Programlama Dilleri

Bölüm 9: Kontrol I - İfadeler ve Komutlar

Öğr.Gör. Şevket Umut Çakır

Pamukkale Üniversitesi

Hafta 11

Hedefler

- İfadeleri anlamak
- Koşullu ifadeleri ve muhafızları anlamak
- Döngüleri ve WHILE'daki çeşitliliği anlamak
- GOTO tartışmalarına ve döngü çıkışlarına aşina olmak
- İstisna işlemeyi anlamak
- TinyAda'da statik ifadelerin değerlerini hesaplamak

Giriş

- Bu bölüm, ifadelerin ve komutların kullanımı yoluyla temel ve yapılandırılmış kontrol soyutlamasını tartışır.
- **İfade(Expression)**: bir değer verir ve hiçbir yan etki oluşturmaz
- **Komut(Statement)**: yan etkileri için yürütülür ve hiçbir değer döndürmez
- Fonksiyonel dillerde (**ifade dilleri(expression languages)** olarak da adlandırılır), hemen hemen tüm dil yapıları ifadelerdir

Giriş

- C **ifade odaklı bir dil(expression-oriented language)** olarak adlandırılabilir
  - İfadeler, komutlardan çok daha büyük bir dil bölümünü oluşturur
- Yan etki yoksa, ifadeler görünüş olarak matematiğe en yakın olanlardır.
  - Matematiksel ifadelere benzer semantiğe sahip olur
- Yan etkilere sahip ifadelerin semantiğinin önemli bir kontrol bileşeni vardır

Giriş

- Açık kontrol yapıları ilk olarak GOTO'lar olarak ortaya çıktı
- Algol60 **yapılandırılmış kontrol(structured control)** getirdi
  - Kontrol komutları, **bloklar(blocks)** gibi **tek girişli(single-entry)**, **tek çıkışlı(single-exit)** ifadelere ve bu ifadelerden kontrol aktarır
- Bazı diller GOTO'ları tamamen ortadan kaldırır, ancak yapısal programlama bağlamında GOTO'ların faydası konusunda hala tartışmalar vardır.

İfadeler

- Temel ifadeler değişmez değerlerden(literals) ve tanımlayıcılardan(identifier) oluşur
- Karmaşık ifadeler, operatörlerin ve fonksiyonların uygulanmasıyla temel ifadelerden özyinelemeli olarak oluşturulur.
  - Parantez gibi sembollerin gruplandırılmasını içerebilir
- Örnek: 3 + 4 * 5 ifadesinde
  - + **operatörü**, iki **işlenen(operand)** 3 ve 4 * 5 alt ifadesine uygulanır
- **Tekli operatör(Unary operator)**: bir işlenen alır
- **İkili operatör(Binary operator)**: iki işlenen alır

İfadeler

- Operatörler içek(infix), sonek(postfix) veya önek(prefix) gösterimiyle yazılabilir
  - Sonek ve önek formları, operatörlerin uygulandığı sırayı ifade etmek için parantez gerektirmez
- Operatörler önceden tanımlanmıştır, özel ilişkilendirilebilirlik ve öncelik kuralları ile (ikili ise) infix biçiminde yazılmıştır.
- Fonksiyonlar, **bağımsız değişkenler(arguments)** veya **gerçek parametreler(actual parameters)** olarak görülen işlenenlerle kullanıcı tanımlıdır.
- Operatörler ve fonksiyonlar eşdeğer kavramlar olduğundan bu ayrım keyfidir

İfadeler

- Yerleşik operatörler yüksek düzeyde optimize edilmiş **satır içi kod(inline code)** olarak uygulandığı için ayırt etme önemlidir
  - Foksiyonlar **aktivasyonların(activation)** oluşturulmasını gerektirir
- Modern çevirmenler genellikle kullanıcı tanımlı fonksiyonları bile satır içi oluşturur
- Lisp, işlenenler olarak değişken sayıda argüman alabildiğinden, ifadelerin **tamamen parantez(fully parenthesized)** içine alınmasını gerektirir.
- **Uygun sıralı değerlendirme(Applicative order evaluation)** (veya **katı değerlendirme(strict evaluation)**) kuralı: önce tüm işlenenler değerlendirilir, ardından operatörler bunlara uygulanır

İfadeler

- Örnek: uygun sıralı değerlendirme
  - + ve - düğümleri 7 ve -1 olarak değerlendirilir
  - Daha sonra -7 elde etmek için * uygulanır

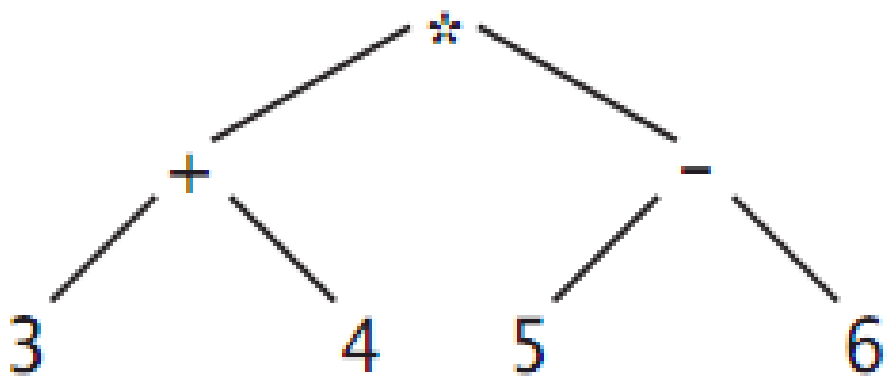


Figure 9.2 Syntax tree for the expression (3 + 4) * (5 – 6)

İfadeler

- İşlenecek doğal sıra (3 + 4) ve (5 - 6) soldan sağa, ancak birçok dil bir sıra belirtmez
  - Makinelerin prosedür ve fonksiyon çağrılarının yapısı için farklı gereksinimleri olabilir
  - Çevirmenler verimlilik için yeniden düzenlemeye çalışabilir
- Herhangi bir yan etki yoksa, alt ifadelerin değerlendirme sıralaması bir fark yaratmayacaktır.
  - Yan etkiler varsa, farklılıklar olabilir.

İfadeler

```c
#include <stdio.h>
int x = 1;
inf f(){
    x += 1;
    return x;
}
int p(int a, int b) {
    return a + b;
}
main() {
    printf("%d\n", p(x, f()));
    return 0;
}
```

Şekil: Yan etkilerle değerlendirme sırasının önemli olduğunu gösteren C programı

İfadeler

- Bazen ifadeler açıkça yan etkiler göz önünde bulundurularak oluşturulur
- C'de atama bir ifadedir
  - Örnek: C kodunda: `x = (y = z)`
    - y = z, x'e atanan bir değeri, hem atar hem de döndürür
- **Sıra operatörü(Sequence operator)**: birkaç ifadenin tek bir ifadede birleştirilmesine ve sıralı olarak değerlendirilmesine izin verir
  - Örnek: C kodunda: `x = (y += 1, x += y, x + 1)`

İfadeler

- **Kısa devre değerlendirmesi(Short-circuit evaluation)**: Boole ifadeleri, tüm ifadenin doğruluk değerinin bilindiği noktaya kadar soldan sağa değerlendirilir ve ardından değerlendirme durur
  - Örnek: Ada'da: `x or true`
    - x'in değerine bakılmaksızın her zaman doğrudur
- Kısa devre değerlendirmesinde değerlendirme sırası önemlidir
- **If ifadeleri** ve **case ifadeleri** de tam olarak değerlendirilemeyebilir

İfadeler

- If (veya if-then-else) operatörü: üç işlenenli bir **üçlü operatördür(ternary operator)**
- Mix-form: işlecin söz diziminin bölümlerini ifade boyunca dağıtır
  - Örnek: ML kodunda: `if e1 then e2 else e3`
- If-ifadeleri hiçbir zaman tüm alt ifadelerini değerlendirmez
- **Case ifadesi(expression)**: bir dizi iç içe geçmiş if ifadesine benzer
- **Gecikmeli değerlendirme(Delayed evaluation)** (veya **katı olmayan değerlendirme(nonstrict evaluation)**): operatörler işlenenlerini değerlendirmeyi geciktirdiğinde

İfadeler

- **Değiştirme kuralı(Substitution rule)** (veya **referans şeffaflığı(referential transparency)**): aynı kapsamda aynı değere sahip herhangi iki ifade birbirinin yerine kullanılabilir
  - Değerleri, değerlendirme bağlamına bakılmaksızın her zaman eşit kalır
  - Bunun ifadelerde değişkenleri yasakladığını unutmayın.
- **Normal sıralı değerlendirme(Normal order evaluation)**: her işlem, işlenenleri değerlendirilmeden önce değerlendirmeye başlar ve her işlenen, yalnızca işlemin hesaplanması için gerekliyse değerlendirilir.

İfadeler

- Örnek C kodunda:
  - `square(double(2))` ifadesini düşünün
  - `square`, `double(2)*double(2)` ile yer değiştirir
  - `double(2)` değerlendirilmeden
  - Daha sonra `2+2` ile yer değiştirilir
- Normal sıralı değerlendirme bir tür satır içi kod uygular

```
int double(int x) {
    return x + x;
}
int square(int x) {
    return x * x;
}
```

İfadeler

- Hiçbir yan etkisi olmadan, normal sıra değerlendirmesi bir programın anlamını değiştirmez
- C kodundaki yan etkilere sahip örnek:

```
int get_int() {
    int x;
    /* standart girdiden x değişkenine bir tamsayı oku */
    scanf("%d", &x);
    return x;
}
```

- `square(get_int())` İfade şu şekilde genişletilirdi: `get_int()*get_int()`
  - Bir yerine iki tamsayı değeri okurdu

İfadeler

- Normal sıralı değerlendirme:
  - Haskell fonksiyonel dilinde **tembel değerlendirme(lazy evaluation)** olarak görünür
  - Algol60'taki fonksiyonlar için **isim olarak gönderme(pass by name)** parametre geçme tekniği olarak görünür

Koşullu İfadeler ve Muhafızlar

- **if-ifadesi**: yapılandırılmış kontrolün tipik biçimi
  - Bir grup ifadenin yürütülmesi yalnızca belirli koşullar altında gerçekleşir

- **Muhafızlı(Guarded) if** ifadesi:
  - Tüm Bi'ler **muhafız(guards)** adı verilen Boole ifadelerdir
  - Tüm Si'ler komut dizileridir
  - Bir Bi doğru olarak değerlendirilirse, karşılık gelen Si yürütülür
  - Birden fazla Bi doğruysa, yalnızca bir Si çalıştırılır

```
if  B1 -> S1
|   B2 -> S2
|   B3 -> S3
        ...
|   Bn -> Sn
fi
```

Koşullu İfadeler ve Muhafızlar

- İlk gerçek Bi'nin seçildiğini söylemez
  - Bu, programlamaya belirsizliği getirir
- Tüm muhafızların değerlendirilip değerlendirilmeyeceğini belirsiz bırakır
  - Eşzamanlı programlama için kullanışlı bir özellik
- Genel uygulama, gerçek bir tane bulunana kadar tüm Bi'leri sırayla değerlendirmek, ardından karşılık gelen Si'yi çalıştırmaktır.
- If-ifadeleri ve case-ifadeleri, muhafızlı if uygulanmasının başlıca yollarıdır.

If-İfadeleri

- EBNF'deki C kodundaki if ifadesinin temel biçimi:

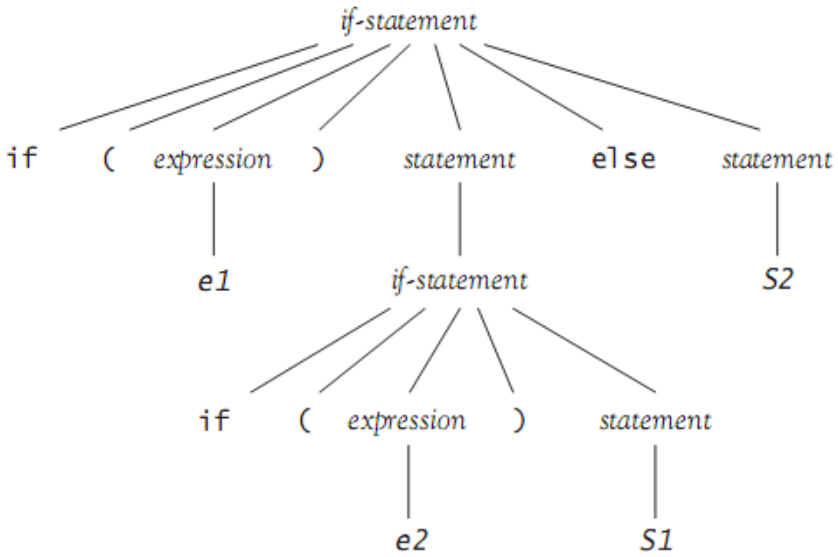
  if-statement → `if (` *expression* `)` *statement* [`else` *statement*]

  - Bir ifade, tek bir ifade veya parantez içine alınmış bir dizi ifade olabilir.
- Bu if ifadesi sorunludur, çünkü olası iki farklı ayrıştırma ağacı vardır:

```
if (e1) if (e2) S1 else S2
```

  - **Sarkan-else(Dangling-else)** problemi olarak adlandırılır

If-İfadeleri



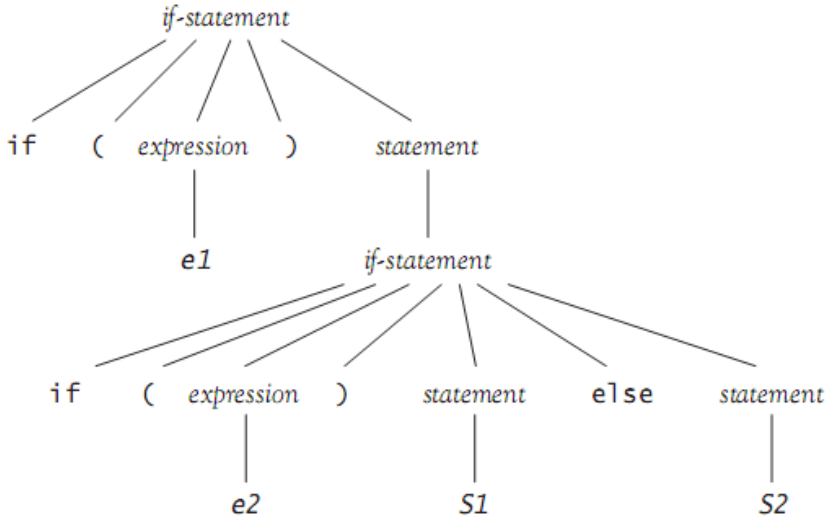


Figure 9.4 Two parse trees for the statement `if` (e1) `if` (e2) S1 `else` S2

If-İfadeleri

- C ve Pascal belirsizliği ortadan kaldıran bir kuralı uygular:
  - else, bir else parçası olmayan en yakın if ile ilişkilendirilir
  - İf-ifadeleri için **en yakın iç içe geçmiş(most closely nested)** kural olarak adlandırılır
- Sarkan-else problemini çözmenin daha iyi bir yolu, Ada kuralında olduğu gibi bir parantez(çevreleyen) anahtar kelimesi kullanmaktır:

if-statement → `if` *condition* `then` *sequence-of-statements*
`[else` *sequence-of-statements*`]` `end if ;`

  - end if, if ifadesini kapatır ve belirsizliği ortadan kaldırır

If-İfadeleri

- Bu aynı zamanda yeni bir ifade dizisi açmak için parantez kullanma zorunluluğunu da ortadan kaldırır:

```
if x > 0.0 then
   y := 1.0/x;
   done := true;
else
   x := 1.0;
   y := 1.0/z;
   done := false;
end if;
```

If-İfadeleri

- Ada'daki `elsif`, birçok alternatif varken çoklu end if'i ortadan kaldırır:

```
if e1 then S1
else if e2 then S2
else if e3 then S3
end if ; end if ; end if;
```

- Şeklinde değişir:

```
if e1 then S1
elsif e2 then S2
elsif e3 then S3
end if ;
```

Case ve Switch İfadeleri

- **Case** veya **switch ifadesi**: muhafızların sıralı bir ifade tarafından seçilen sıra değerleri olduğu durumlarda korunur
- C'de semantik:
  - Kontrol ifadesini değerlendirir
  - Denetimi, değerin listelendiği durum ifadesine aktarır
  - Listelenen iki case aynı değere sahip olamaz
  - Durum değerleri değişmez değerler veya derleme zamanı sabiti ifadeleri olabilir
  - Hiçbir değer eşleşmezse, varsayılan duruma aktarır

Case ve Switch İfadeleri

```
1   switch (x - 1) {
2       case 0:
3           y = 0;
4           z = 2;
5           break;
6       case 2:
7       case 3:
8       case 4:
9       case 5:
10          y = 3;
11          z = 1;
12          break;
13      case 7:
14      case 9:
15          z = 10;
16          break;
17      default:
18          /* bir şey yapma */
19          break;
20  }
```

Şekil: C'de switch ifadesinin kullanımına bir örnek

Case ve Switch İfadeleri

- `default` bir durum yoksa, kontrol, `switch`'den sonra gelen bir sonraki ifadeye geçer.
- Bazı yeni özellikler:
  - `case` etiketleri, sözdizimsel olarak sıralı etiketler gibi ele alınır
  - `break` ifadesi olmadan yürütme bir sonraki duruma(case) geçer
- Ada, case değerlerinin gruplanmasına izin verir ve kapsamlı olmalarını gerektirir
  - Geçerli bir değer listelenmemişse derleme zamanı hatası

Case ve Switch İfadeleri

```
1  case x - 1 is
2      when 0 =>
3          y := 0;
4          z := 2;
5      when 2 .. 5 =>
6          y := 3;
7          z := 1;
8      when 7 | 9 =>
9          z := 10;
10     when others =>
11         null;
12 end case;
```

Şekil: Bir önceki C örneğine karşılık gelen, Ada'da case ifadesinin kullanımına örnek

Case ve Switch İfadeleri

- ML'nin case yapısı, bir komut(statement) yerine bir değer döndüren bir ifadedir(expression)
  - Durumlar(case) dikey çubuklarla ayrılmıştır
  - Case ifadeleri eşleştirilecek desenlerdir
  - **Joker karakter kalıbı(Wildcard pattern)** alt çizgidir

```
fun casedemo x =
    case x - 1 of
        0 => 2 |
        2 => 1 |
        _ => 10
    ;
```

Döngüler ve WHILE'daki Çeşitlilikler

- **Muhafızlı(Guarded) do**: bir döngü yapısı için genel bir form
  - Tüm Bi'ler yanlış olana kadar ifade tekrarlanır
  - Her adımda, doğru Bi'lerden biri kesin olmayan bir şekilde(nondeterministically) seçilir ve karşılık gelen Si çalıştırılır.

```
do B1 - > S1
 | B2 - > S2
 | B3 - > S3
. . .
 | Bn -> Sn
od
```

Döngüler ve WHILE'daki Çeşitlilikler

- Temel döngü yapısı: yalnızca bir muhafızla, bir muhafızlı do
  - Belirsizliği ortadan kaldırır
- C'de: `while (e) S`
- Ada'da: `while e loop S1 ... Sn end loop;`
- Önce test ifadesi (e) değerlendirilir
  - Ada ve Java'da Boole olmalı, ancak C veya C++'da olmayabilir
  - Doğruysa (veya sıfır değilse), S yürütülür ve süreç tekrar eder

Döngüler ve WHILE'daki Çeşitlilikler

- Bazı dillerin, döngünün en az bir kez yürütülmesini sağlayan alternatif bir formu vardır.
  - C ve Java'da: `do` (veya `do-while`) ifadesi

```
do S while (e);
```

- `do` veya `while` döngüsünün sonlandırılması, yalnızca döngünün başında veya sonunda açıkça belirtilir
- C ve Java, bir döngünün içinden tamamen çıkmak için bir `break` ifadesi sağlar
  - `continue` ifadesi, döngünün gövdesinin kalanını atlar ancak bir sonraki yinelemeyle devam eder

Döngüler ve WHILE'daki Çeşitlilikler

- C/C++ ve Java'daki **for-döngüsü(for-loop)**

```
for ( e1; e2; e3 ) S;
```

  - C'deki eşdeğerdiri:
    - e1 başlatıcıdır
    - e2 testtir
    - e3 güncellemedir

```
e1;
while (e2)
{ S;
  e3;
}
```

- For-döngüsü tipik olarak, baştan sonuncuya kadar bir dizi değerin üzerinden geçmek için kullanılır.

```
for (i = 0; i < size; i++)
        sum += a[i];
```

Döngüler ve WHILE'daki Çeşitlilikler

- C ++ ve Java, döngüde bir for-loop başlatıcısının(dizin(index)) bildirilmesine izin verir:

```
for (int i = 0; i < size; i++)
        sum += a[i];
```

- Birçok dil for-döngüsü biçimini kısıtlar
- Çoğu kısıtlama, kontrol değişkeni i'yi içerir:
  - Döngünün gövdesinde i değeri değiştirilemez
  - Döngü sonlandırıldıktan sonra i değeri tanımsız
  - i kısıtlı türde olmalı ve bir prosedür parametresi veya kayıt alanı(record field) olmamalı

Döngüler ve WHILE'daki Çeşitlilikler

- Döngü davranışıyla ilgili diğer sorular şunları içerir:
  - Sınırlar yalnızca bir kez mi değerlendirilir? Eğer öyleyse, yürütme başladıktan sonra sınırlar değişmeyebilir
  - Alt sınır üst sınırdan büyükse döngü hiç yürütülüyor mu?
  - Bir `exit` veya `break` ifadesi kullanılıyorsa, kontrol değişkeni değeri tanımsız mı?
  - Döngü yapılarında hangi çevirmen kontrolleri yapılır?
- Nesne yönelimli diller, bir koleksiyonun öğeleri üzerinde döngü yapmak için bir yineleyici(iterator) nesnesi kullanır

Döngüler ve WHILE'daki Çeşitlilikler

```
Iterator iter<String> = list.iterator();
while (iter.hasNext())
    System.out.println(iter.next());

for (String s : list)
    System.out.println(s);
```

Şekil: Bir listeyi işlemek için Java'da iterator yapsının iki farklı kullanımı

GOTO Tartışması ve Döngü Çıkışları

- Goto ifadeleri, Fortran77 ve BASIC gibi ilk programlama dillerinde yoğun bir şekilde kullanıldı.
- Fortran77'deki örnek:

```
10 IF (A(I).EQ.0) GOTO 20
   !...
   I = I + 1
   GOTO 10
20 CONTINUE
```

- C kodundaki eşdeğeri:

```
while (a[i] != 0) i++;
```

GOTO Tartışması ve Döngü Çıkışları

- 1960'larda yapılandırılmış kontrol kullanımının artmasıyla birlikte, gotos'un faydası hakkında tartışmalar başladı.
  - **Spagetti koda** yol açabilir

```
    IF (X.GT.0) GOTO 10
    IF (X.LT.0) GOTO 20
    X = 1
    GOTO 30
10  X = X + 1
    GOTO 30
20  X = -X
    GOTO 10
30  CONTINUE
```

GOTO Tartışması ve Döngü Çıkışları

- 1966'da Bohm ve Jacopini, gotos'un tamamen gereksiz olduğuna dair teorik sonuç üretti.
- 1968'de Dijkstra "GOTO Statement Considered Harmful" nı yayınladı
  - Kullanımının ciddi şekilde kontrol edilmesini veya kaldırılmasını önerdi
- Birçoğu, bazı durumlarda haklı görülebilecek şekilde değerlendirildi
- Rubin 1987'de ""Goto considered harmful" considered harmful" ifadesini yayınladı

GOTO Tartışması ve Döngü Çıkışları

- Döngülerden yapılandırılmamış çıkışların uygunluğu konusunda hala bazı tartışmalar
  - Bazıları döngüde yalnızca bir çıkış olması gerektiğini savunuyor
  - Diğerleri, belirli durumlar için daha karmaşık kod gerektirebileceğini savunuyor
- Örnek: belirli bir öğe için bir dizide arama
  - Yöntem, dizinin içindeyse hedef öğenin konumunu veya aksi takdirde -1'i döndürür
- Örnek: bir dizi girdi değerini işlemek için gözcü(sentinel) tabanlı döngü
  - **Döngü ve bir buçuk problem(Loop and a half)** olarak adlandırıldı

GOTO Tartışması ve Döngü Çıkışları

```
int search(int array[], int target){
    boolean found = false;
    int index = 0;
    while(index < array.length &&
    ↪  !found)
        if(array[index] == target)
            found = true;
        else
            index++;
    if(found)
        return index;
    else
        return -1;
}
```

Şekil: Yapısal döngü çıkışlı arama

```
int search(int array[], int target){
    for(int index = 0; index <
    ↪  array.length; index++)
        if(array[index] == target)
            return index;
    return -1;
}
```

Şekil: Yapısal döngü çıkışsız arama

GOTO Tartışması ve Döngü Çıkışları

```
void processInputs(Scanner s){
    int datum = s.nextInt();
    while(datum != -1){
        process(datum);
        datum = s.nextInt();
    }
}
```

Şekil: Yapısal döngü çıkışı

```
void processInputs(Scanner s){
    while(true){
        int datum = s.nextInt();
        if(datum == -1)
            break;
        process(datum);
    }
}
```

Şekil: Yapısal olmayan döngü çıkışı

İstisna İşleme

- **Açık kontrol mekanizmaları(Explicit control mechanisms)**: Kontrol transferinin gerçekleştiği noktada, transferin sözdizimsel bir göstergesi vardır.
- **Örtülü kontrol devri(Implicit transfer of control)**: aktarım, gerçek aktarımın gerçekleştiği yerden farklı bir noktada kurulur
- **İstisna işleme(Exception handling)**: yürütme sırasında hata koşullarının veya diğer olağandışı olayların kontrolü
  - Hem istisnaların hem de istisna işleyicilerin bildirimini içerir

İstisna İşleme

- Bir istisna meydana geldiğinde, **yükseltildiği(raised)** veya **fırlatıldığı(thrown)** söylenir
- İstisna örnekleri:
  - Çalışma zamanı istisnaları(Runtime exceptions): aralık dışı dizi indisleri veya sıfıra bölme
  - Yorumlanan kod: sözdizimi veya tür hataları
- **İstisna işleyici(Exception handler)**: belirli bir istisna ortaya çıktığında yürütülecek şekilde tasarlanmış prosedür veya kod dizisi
- Bir istisna işleyicisinin bir istisnayı **işlediği(handle)** veya **yakaladığı(catch)** söylenir

İstisna İşleme

- Hemen hemen tüm büyük güncel diller yerleşik istisna işleme mekanizmalarına sahiptir
  - Bu mekanizmaları olmayan diller bazen bunu sağlayan kitaplıklara sahiptir.
- İstisna işleme, bir donanım kesmesinin(hardware interrupt) veya hata tuzağının(error trap) özelliklerini taklit etmeye çalışır
  - Temel makine veya işletim sistemi hatayı işlemeye bırakılırsa, program genellikle durdurulacak veya çökecektir.
- Kilitlenen programlar **sağlamlık(robustness)** testinde başarısız olur

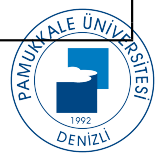
İstisna İşleme

- Bir programın meydana gelebilecek her olası hatayı işleyebilmesi beklenemez
  - Donanım dahil çok fazla olası arıza
- **Eşzamansız istisnalar(Asynchronous exceptions)**: temeldeki işletim sistemi bir sorun tespit ettiğinde ve bir programı sonlandırması gerektiğinde
  - Yürütülen program koduna yanıt olarak değil
- **Eşzamanlı istisnalar(Synchronous exceptions)**: programın eylemlerine doğrudan yanıt olarak ortaya çıkan istisnalar

İstisna İşleme

- Kullanıcı tanımlı istisnalar yalnızca eşzamanlı olabilir
- Önceden tanımlanmış veya kitaplık istisnaları, bazı eşzamansız istisnalar içerebilir
- İstisna işleme, dildeki istisnaları test etmenin mümkün olduğunu varsayar
- Hatayı oluştuğu yerde halledebilir:

```
if (y == 0)
    handleError("bölen sıfır");
else
    ratio = x / y;
```



İstisna İşleme

- Bir hata durumunu çağıran prosedüre geri iletebilir

```
enum ErrorKind {OutOfInput, BadChar, Normal};
//...
ErrorKind getNumber(unsigned* result){
    int ch = fgetc(input);
    if (ch == EOF) return OutOfInput;
    else if(!isDigit(ch)) return BadChar;
    /* hesaplamaya devam et*/
    //...
    *result=...;
    return Normal;
}
```

İstisna İşleme

- Ayrıca çağrı için ayrı bir istisna işleme prosedürü oluşturabilir
- Hata işlemeyi kolaylaştırmak için, istisnaları oluşmadan önce bildirmek ve yapılacak eylemleri belirtmek istenilmektedir.
- Bunu yapmak için aşağıdakilerle ilgili konular dikkate alınmalıdır:
  - İstisnalar
  - İstisna işleyiciler
  - Kontrol

İstisnalar

- İstisna, genellikle önceden tanımlanmış veya kullanıcı tanımlı bir veri nesnesiyle temsil edilir.
  - İşlevsel bir dilde bir değer olacak
  - Yapılandırılmış veya nesne yönelimli bir dilde, bir değişken veya bazı yapılandırılmış tipte bir nesne olacaktır.
- Örnek: ML veya Ada'da:

```
exception Trouble; (* kullanıcı tanımlı bir istisna *)
exception Big_Trouble; (* bir başka kullanıcı tanımlı
↪  istisna *)
```

- Örnek: C++ 'da:

```
struct Trouble {} trouble;
struct Big_Trouble {} big_trouble;
```

İstisnalar

- Genellikle, hata mesajı veya ilgili verilerin özeti gibi istisnalar dışında ek bilgiler eklenmek istenir

```
struct Trouble{
    string error_message;
    int wrong_value;
} trouble;
```

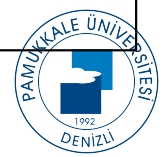
- İstisna bildirimleri genellikle diğer bildirimlerle aynı kapsam kurallarına uyar
  - Erişilebilir olduklarından emin olmak için global olarak kullanıcı tanımlı istisnaların bildirilmesi istenebilir
- Çoğu dil, doğrudan veya standart kitaplık modüllerinde önceden tanımlanmış bazı istisna değerleri veya türleri sağlar

İstisna İşleyiciler

- C ++ 'da, özel durum işleyicileri, **try-catch** bloklarıyla ilişkilendirilir
  - Herhangi bir sayıda yakalama bloğu dahil edilebilir
  - Her catch bloğu, istisna tipini bir parametre olarak alır ve yapılacak işlemlerin bileşik bir ifadesini içerir.
  - **...** parametresine sahip son yakalama bloğu, önceki catch bloklarında ele alınmayan istisnaları yakalamaktır.

İstisna İşleyiciler

```
try
{
    //...
}
catch(Trouble t)
{ //mümkünse trouble ile başa çık
    displayMessage(t.error_message);
    //...
}
catch(Big_Trouble b)
{ //mümkünse big trouble ile başa çık
    //...
}
catch (...) //üç nokta sözdiziminde
{ // kalan yakalanmamış istisnaları ele al
}
```



Şekil: Bir C++ try-catch bloğu

İstisna İşleyiciler

```
begin
    -- bir işlem yapmayı dene
    --...
exception
    when Trouble =>
        -- münkünde trouble ile başa çık
        displayMessage("trouble here!");
        --...
    when Big_Trouble =>
        -- mümkünde big trouble ile başa çık
        --...
    when others =>
        --kalan yakalanmamış istisnaları ele al
end;
```

Şekil: Bir Ada try-catch bloğu

İstisna İşleyiciler

```
val try_to_stay_out_of_trouble =
(* bir hesaplama yapmaya çalış *)
handle
    Trouble (message,value) =>
        (displayMessage(message); ...)|
    Big_Trouble => ... |
    _ =>
        (* kalan yakalanmamış istisnaları ele al *)
        ...
;
```

Şekil: Bir ML istisna işlemesi

İstisna İşleyiciler

- Önceden tanımlanmış işleyiciler tipik olarak, istisna türünü ve muhtemelen bazı ek bilgileri gösteren minimal bir hata mesajı yazdırır ve ardından programı sonlandırır.
- Ada ve ML'de, varsayılan işleyicilerin davranışını değiştirmenin bir yolu yoktur
  - Ada'da devre dışı bırakabilir
- C++ 'da, varsayılan işleyiciyi <exceptions> standart kitaplık modülünü kullanarak kullanıcı tanımlı bir işleyiciyle değiştirebilir

Kontrol

- Önceden tanımlanmış veya yerleşik istisnalar, çalışma zamanı sistemi tarafından otomatik olarak oluşturulur veya program tarafından manuel olarak fırlatılabilir
- Kullanıcı tanımlı istisnalar yalnızca program tarafından fırlatılabilir
- C++ 'da, `throw` ayrılmış sözcüğü ile bir istisna oluşturulabilir
- Ada ve ML, her ikisi de `raise` ayrılmış sözcüğünü kullanır

Kontrol

- Örnek: C++ kodunda:

```
if (/* bir şeyler ters giderse */)
{
    Trouble t;//Bilgiyi saklaması için yeni bir Trouble
    ↪ değişkeni oluştur
    t.error_message = "Kötü haber!";
    t.wrong_value = mecvut_eleman;
    throw t;
}
else if(/* daha kötü bir şey olursa */)
    throw big_trouble;//bilgi olmadığından global değişken
    ↪ kullanılabilir
```

Kontrol

```
if -- bir şeyler ters giderse
then
    raise Trouble;--Trouble'ı sabit olarak kullan
elsif -- daha kötü bir şey olursa
then
    raise Big_Trouble;--Big_Trouble'ı sabit olarak kullan
end if;
```

Şekil: Ada dilinde kullanıcı tanımlı istisna işleme

Kontrol

```
if (* bir şeyler ters giderse *)
then (* Trouble değerini inşa et *)
    raise Trouble("Kötü haber!", mevcut_eleman)
else if (* daha kötü bir şey olursa *)
    raise Big_Trouble (* Big_Trouble bir sabit *)
else ... ;
```

Şekil: ML dilinde kullanıcı tanımlı istisna işleme

Kontrol

- Bir istisna ortaya çıktığında, mevcut hesaplama terk edilir ve çalışma zamanı sistemi bir işleyici aramaya başlar.
- Ada ve C ++ 'da, önce mevcut blok, ardından çevreleyen blok vb. aranır.
  - Buna **istisnayı yaymak(propogating the exception)** denir
- Bir işleyici bulunmadan bir fonksiyonun veya prosedürün en dıştaki bloğuna ulaşılırsa, çağrıdan çıkar ve çağıranda istisna oluşur
- İşlem, bir işleyici bulunana veya ana programdan çıkılana kadar devam eder ve varsayılan işleyiciyi çağırır

Kontrol

- **Çağrı çözme(Call unwinding)** (veya **yığıt çözme(stack unwinding)**): bir işleyici arama sırasında çağırana fonksiyon çağrıları yoluyla geri dönme işlemi
- Bir işleyici bulunup yürütüldüğünde, yürütme nerede devam etmelidir?
  - **Devam ettirme modeli(Resumption)**: istisnanın ilk ortaya çıktığı noktada devam edin ve aynı ifadeyi veya komutu yeniden yapın
  - **Sonlandırma modeli(Termination model)**: çalıştırılan işleyicinin bulunduğu blok veya ifadenin hemen ardından gelen kodla devam edin

Kontrol

- Çoğu modern dil, sonlandırma modelini kullanır
  - Genelde uygulanması daha kolaydır ve yapılandırılmış programlama tekniklerine daha iyi uyar
  - Gerektiğinde devam ettirme modelini simüle edebilir
- Sıradan kontrol durumlarını uygulamak için istisnaları aşırı kullanmaktan kaçının çünkü:
  - İstisna işleme genellikle önemli miktarda çalışma süresi yükü taşır
  - İstisnalar, çok yapılandırılmış olmayan bir kontrol alternatifini temsil eder
- Bunun yerine basit testler kullanın

Kontrol

- Örnek: C ++ 'da, yetersiz bellek nedeniyle bir **new** çağrısı başarısız olduğunda devam ettirme modelini simüle etmek

```
while (true)
{
    try
    {
        x = new X;//tahsis etmeye çalış
        break; //buraya gelirsek, başarılı!
    }
    catch(bad_alloc)
    {
        collect_garbage();//henüz çıkamayız!
        if(/* eğer hala yeterli bellek yoksa */)
            //sonsuz döngünün önüne geçmek için vazgeçmeliyiz
            throw bad_alloc;
    }

}
```

Kontrol

```
void fnd(Tree* p, int i)//yardımcı prosedür
{
    if(p != 0)
        if(i == p->data) throw p;
        else if (i < p->data) fnd(p->left, i);
        else fnd(p->right, i);
}
Tree * find(Tree* p, int i)
{
    try
    {
        fnd(p, i);
    }
    catch(Tree* q)
    {
        return q;
    }
    return 0;
}
```

Şekil: C++'da istisnaları kullanan, ikili arama ağaçlarında kullanılan arama(find) fonksiyonu

Örnek Olay: TinyAda'da Statik İfadelerin Değerlerini Hesaplama

- Pascal, sembolik sabitlerinin değişmez değerler olarak tanımlanmasını gerektirir
- Ada, sabit ifadelere sabit olarak izin verir
- **Statik ifade(Static expression)**: değeri derleme zamanında belirlenebilen, değişken veya fonksiyon çağrısı içermeyen herhangi bir ifade

Statik İfadelerin Sözdizimi ve Anlambilimi

- Statik ifadeler iki tür TinyAda bildiriminde görünebilir:
  - Sembolik bir sabiti tanımlayan bir sayı bildirimi
  - Yeni bir alt aralık türünü tanımlayan bir aralık türü tanımı
- Örnek:

```
ROW_MAX : constant := 10;
COLUMN_MAX : constant := ROW_MAX * 2;
type MATRIX_TYPE is range 1..ROW_MAX, range 1..COLUMN_MAX of BOOLEAN;
```

Statik İfadelerin Sözdizimi ve Anlambilimi

- Sözdizimsel olarak, statik ifadeler tıpkı diğer ifadeler gibi görünür
- Anlamsal olarak da benzerler
- Sonuçların derleme zamanında hesaplanabilmesini sağlamak için statik ifadeler değişkenleri veya parametre referanslarını içeremez

Sembolik Sabitlerin Değerlerini Girme

- Her sembolik sabit, sembol giriş kaydında bir değer niteliğine sahiptir.
- Daha önceki bir bölümde sunulan ayrıştırma yöntemi ifadesi yeniden kullanılamaz çünkü:
  - Tüm ifade yöntemleri, tür denetimi için ve sabit tanımlayıcıların ve alt aralık türlerinin tür özniteliklerini ayarlamak için hala gerekli olan bir tür tanımlayıcı döndürür.
  - Bu yöntemler, değişkenlere ve parametre adlarına izin verir
  - Tüm ifadeler statik değildir

Statik İfadelerin Değerlerine Bakmak

- Yeni metot `staticPrimary`, statik ifadenin en basit biçiminin değerini verecektir
  - Belirteç akışında bir tamsayı veya karakter değişmez değeri veya sabit bir tanımlayıcının değeri olacaktır
- Yapısal olarak statik olmayan emsaline benzer, ancak şunlar hariç:
  - Yeni yöntem bir sembol girişi döndürür
  - Yeni yöntem, yöntem adını çağırmak yerine bir tanımlayıcı arar

Statik İfadelerin Değerlerinin Hesaplanması

- Operatörlerle karşılaşılırsa, her biri bir işlenen ifadesinin ayrıştırılmasının sonucu olan iki veya daha fazla sembol girişini ele almalıyız.