8. İfade Düzeyinde Kontrol Yapıları

Bir programda akış kontrolü veya çalıştırma sırası farklı seviyelerde sağlanabilir. Bir önceki ünitede ifadelerin kendi içindeki çalıştırma sırasının en alt düzeyde, işleç önceliği ve birleşme kuralları gibi kurallarla belirlendiğini görmüştük. Bir de en üst düzeyde, program birimleri arasındaki akış kontrolünden bahsedeceğiz ilerleyen ünitelerde. Bu ünitenin konusu ise ifadeler arasındaki akış kontrolüdr.

8.1. Giriş

Buyurgan dillerde hesaplamalar, ifadelerin işletilmesi ve sonuçların değişkenlere atanması ile gerçekleştirilir. Fakat sadece atama ifadelerinden oluşan bir program nadiren kullanışlı olabilir. En azından iki tane dil mekanizmasının kullanımı ile hesaplamalar daha esnek ve güçlü yapılabilir: alternatif akış yollarından birini seçmek ve ifadelerin veya ifade gruplarının tekrarlı şekilde çalıştırılması.

Bu türden olanakları sağlayan ifadelere kontrol ifadeleri diyoruz. Bir kontrol yapısı, bir kontrol ifadesi ve ona bağlı olarak çalışan diğer ifadeleri kapsar.

8.2. Seçim İfadeleri

Bir seçim ifadesi bir programda iki veya daha fazla işletim yolu arasından seçim yapmak için kullanılır. Seçim ifadeleri iki genel gruba ayrılır: İki yollu (two -way) ve çoklu (n-way) seçim.

8.2.1. İki Yollu Seçim İfadeleri

if control_expression
then clause
else clause

then veya else' ten sonra tek veya çoklu ifadeler işletilebilir. Perl dilinde işletilen tek ifade de olsa birleşik ifade olmalıdır. Birleşik ifadeler çoğu dilde {} arasına yazılır.

Fortran 95, Ada, Python, Ruby' de then ve else sonrası birleşik ifade yerine sıralı ifadeler kullanılır. Örneğin Python'da girintili yazım ile birleşik ifadeler tanımlanır:

```
if x>y :
        x=y
        print “case1”
```

8.2.1.1. İç İçe Seçimler

3. ünitede Belirsiz bir gramer görmüştük.

<if_stmt> → if <logic_expr> then <stmt>
| if <logic_expr> then <stmt> else <stmt>

<stmt> yerine <if_stmt> yazılabildiği durumlarda hangi if ile hangi else' in eşleşeceği problemi oluşmaktaydı.

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

Yukarıdaki ifade iki şekilde yorumlanabilir. Python ve F#' ta girintili yazım ile hangi else' in hangi if' e ait olduğu belirtilir. Perl {} kullanımı ile bu ayrımı gösterir.

Ruby' de,

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

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

yukarıdaki ifadenin iki şekilde yorumlanmasının gösterimidir.

8.2.1.2. Seçici İfadeler

ML, F# ve LISP gibi fonksiyonel dillerde seçici bir deyim değil, bir sonuç üreten ifadedir. Dolayısıyla program içinde diğer ifadelerin kullanıldığı her yerde kullanılabilirler. Örneğin F#' ta:

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

8.2.2. Çoklu Seçim Deyimleri

Çoklu seçim deyimi herhangi bir sayıdaki deyim veya deyim grubundan bir tanesini seçmek için kullanılan bir deyimdir. C, C++, Java, JavaScript dillerinde çoklu seçici deyimi switch'tir.

Switch deyiminin genel formu:

```
switch ( expression ) {
        case constant _ expression 1 : statement 1 ;
        . . .
        case constant n : statement_n ;
        [default: statement n + 1 ]
}
```

Örn.

```
switch (index) {
	case 1:
	case 3: odd += 1;
		sumodd += index;
		break;
	case 2:
	case 4: even += 1;
		sumeven += index;
		break;
	default: printf("Error in switch, index = %d\n", index);
}
```

break ifadesinin kullanımı ile ilk break görüldükten sonraki diğer case' ler işletilmez. Break kullanılmadığında, ilk işletilen case' ten sonra diğer tüm case' ler kontrol edilmeksizin işletilir. Break ifadesi kısıtlı bir goto deyimidir. Dallanmayı switch gövdesinden sonrasına yapar. Break yazmayı unutursak bu bir güvenilirlik problemi oluşturur. Break kullanmamak bize esneklik sağlayabilir ancak güvenilirlik de azalmış olur.

C#' taki switch deyimi C-tabanlı diğer dillerden farklılık gösterir. Her case bölümü mutlaka break veya goto ile bitmelidir:

```
switch (value) {
	case -1:
		Negatives++;
		break;
	case 0:
		Zeros++;
		goto case 1;
	case 1:
		Positives++;
	default:
	Console.WriteLine("Error in switch \n");
}
```

Ayrıca C#' ta value ve case' ler string olabilmektedir.

Ruby' de case :

```
case
when Boolean_expression then expression
. . .
when Boolean_expression then expression
[else expression ]
end
```

Yukarıdan aşağı boolean ifadeler birer birer işletilir. Eğer TRUE ise o ifadenin değeri case ifadesinin değeri olur.

Örn: Ruby' de bir yılın artık yıl olup olmadığını bulma

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

8.2.2.1. If ile Çoklu Seçim

Ruby' nin case'i hariç çoklu seçim için switch veya case yetersiz kalmaktadır. Mantıksal kontroller tam olarak switch ile ifade edilemez.

Python'da else if için elif deyimi kullanılır.

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

Bu ifade aşağıdaki ifadenin eş değeridir.

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

Ruby case deyimi ile yukarıdaki ifade:

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

8.3. Tekrarlamalı Deyimler (Iterative Statements)

Bir yinelemeli deyim, bir ifade veya ifadeler grubunun sıfır, bir veya daha fazla defa işletilmesine sebep olan deyimdir.

Yinelemeli bir deyimin gövdesi (body), işletimi yinelemeli deyim tarafından kontrol edilen ifadeler topluluğudur. Öntest (pretest) kavramı, döngünün sonlanması için gerekli koşulun gövde işletilmeden önce kontrol edilmesini, posttest ise önce gövdenin işletilip sonra koşulun kontrol edilmesini ifade eder.

8.3.1. Sayaç Kontrollü Döngüler

Bu tip döngülerde döngü değişkeni denen bir değişken ile sayma işlemi gerçekleştirilir. Döngü değişkeninin ilk ve son değerlerinin ne olacağı ve ardışık sayaç değerleri arasındaki fark (artış miktarı) da belirtilir. Döngü değişkeninin ilk ve son değerleri ile artış miktarının tümüne döngü parametreleri denir.

8.3.1.1. Ada'da for Deyimi

Ada'da for deyimin genel yapısı şöyledir:

```
for variable in [reverse] discrete_range loop
. . .
end loop;
```

Ada' da döngü değişkeni tanımlanmaz. Dolaylı olarak yani üstü kapalı tanımlanır ve döngü bittiğinde yaşam süresi de biter.

```
Count : Float := 1.35;
for Count in 1..10 loop
        Sum := Sum + Count;
end loop;
```

Yukarıdaki örnekte public olarak tanımlı olan Count değişkeninin değeri döngüden etkilenmez, çünkü döngü değişkeni olan Count integer tipinde bir sayaçtır ve döngü bitince belleğe geri bırakılır. Ada' da döngü değişkenine değer atanamaz.

8.3.1.2. C-tabanlı dillerde for Deyimi

Genel yapı:

```
for ( expression_1 ; expression_2 ; expression_3 )
loop body
```

Burada döngü gövdesi (loop body), tek ifade, çoklu ifade olabilir veya hiçbir şey olmayabilir. İlk expression bir kere işletilir ve döngü değişkenine ilk değeri atanır. İkinci expression döngü kontrol ifadesidir, gövdenin her işletiminden önce kontrol edilir. C' de sıfır değeri FALSE anlamına gelir. İkinci ifadenin sonucu sıfır olunca for döngüsü sonlanır, aksi taktirde gövde işletilmeye devam eder.

8.3.1.3. Python' da for Deyimi

Genel yapı:

```
for loop_variable in object:
- loop body
[ else:
- else clause]
```

Döngü değişkenine nesne içindeki değer atanır ve bu genelde bir değer aralığıdır. Her bir işletimde döngü değişkenleri o aralıktan bir değer alır. Else

kısmı döngü normal olarak bitince işletilen kısımdır.

```
for count in [2, 4, 6]:
        print count
```

çalıştığında aşağıdaki sonuç üretilir:

```
2
4
6
```

For döngü aralığı range komutu ile de verilebilir for count in range(....)

range(5) ile [0,1,2,3,4] üretilir.
range(2,7) ile [2,3,4,5,6] üretilir. İkiden başlar 7'de (7 hariç) biter.
range(0,8,2) ile [0,2,4,6] üretilir.

8.3.2. Mantıksal Kontrollü Döngüler

Bazı durumlarda bir ifade grubu tekrarlı şekilde işletilmek istenebilir ancak tekrar kontrolü bir sayaç yerine mantıksal bir ifade ile gerçekleşirilir. Mantıksal kontrollü döngüler daha genel yapılardır. Her sayaç kontrollü döngü, mantıksal döngülerle gerçekleştirilebilir ancak tersi doğru değildir.

C-tabanlı diller hem pretest hem de posttest türde mantıksal kontrollü döngülere sahiptir.

```
while ( control_expression )
        loop body
```

ve

```
do
        loop body
while ( control_expression );
```

C# örneği:

```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
        sum += indat;
        indat = Int32.Parse(Console.ReadLine());
}
```

```
value = Int32.Parse(Console.ReadLine());
do {
        value /= 10;
        digits ++;
} while (value > 0);
```

Do- while ile while arasındaki fark, do kullanıldığında koşulun ne olduğundan bağımsız olarak döngü gövdesinin en az bir kere çalıştırılmasıdır.

Java'daki while ve do deyimleri C ve C++ ile aynıdır. Sadece kontrol ifadesi boolean tipinde olmak zorundadır. Java'da go to kullanımı yoktur, döngü gövdesine herhangi bir yerinden girilemez.

Testi sonradan yapılan (posttest) döngüler çok sık kullanılmaz ve aynı zamanda tehlikeli olabilirler. Çünkü yazılımcılar do döngü gövdesinin en az bir kere çalışacağını unutabilirler.

8.3.3. Kullanıcı Tarafından Konumlandırılmış Döngü Kontrol Mekanizmaları

Bazı durumlarda döngü kontrolünün gövdenin başında ya da sonunda yer alması yerine, yazılımcı kendi bir yer belirleme ihtiyacı duyabilir. Bazı diller bu imkanı sunmaktadır. Bu tip döngüler sonsuz döngüler olabilir ve kullanıcı tarafından yerleştirilen döngüden çıkma mekanizmaları kullanılır.

C, C++, Python, Ruby ve C#' taki **break** ile döngüden koşulsuz ve etiketsiz bir şekilde çıkılabilir. Java' da break ve Perl' de last deyimleri ile koşulsuz ancak etiketli çıkışlar yapılabilir.

İç içe Java döngüsü:

```
outerLoop:
for (row = 0; row < numRows; row++)
        for (col = 0; col < numCols; col++) {
                sum += mat[row][col];
                if (sum > 1000.0)
                        break outerLoop;
        }
```

C, C++ ve Python'da etiketsiz bir kontrol deyimi olan **continue** ile kontrol, en içteki kapsayan döngünün kontrol mekanizmasına geçer. Bu bir çıkış deyimi değildir, daha ziyade döngüdeki ifadelerin kalan kısmının o adım için işlenmemesini sağlar, fakat döngüyü sonlandırmaz.

```
while (sum < 1000) {
        getnext(value);
        if (value < 0) continue;
                sum += value; // negatif sayı gelirse atama ifadesi işlenmeyecek
}
```

Hem last hem de break, döngüden çoklu çıkış yapmak anlamına gelir ve okunabilirliği tehtit eden bir durum gibi gözükebilir. Fakat döngüleri sonlandırmayı gerektiren koşullarla çok sık karşılaşılır ve dolayısıyla bu kullanım kabul görmüştür. Bunlar go to ihtiyacını kısıtlı bir dallanma ile karşılar.

Go to deyimi ile programın herhangi bir yerine atlayabiliriz. Bu yer go to deyiminden yukarıda veya aşağıda olabilir. Ancak kullanıcının yerleştirdiği

döngü çıkışları, çıkış deyiminin altında biryerlere (döngü gövdesinden sonraki satıra) yapılmalıdır.

8.3.4. Veri Yapılarına Dayalı Tekrarlar

Fortran' da do kullanımı

Do count = 1, 9, 2
burada 1 ilk değer, 9 son değer, 2 de adım büyüklüğü

Ada' da dizi kullanımı:

```
Subtype MyRange is Integer range 0..99;
MyArray : array (MyRange) of Integer;
for index in MyRange loop
...
end loop;
```

C-tabanlı dillerde for deyimi yüksek esnekliği sayesinde kullanıcı tanımlı tekrar deyimi yerine kullanılabilir. Bir ikili ağacın düğümleri işlenecek ve ağacın kökü root değişkeni tarafından işaret ediliyor olsun. Traverse fonksiyonu da parametresini bir sonraki düğümü gösterecek şekilde ayarlasın.

```
for(ptr=root; ptr==null; ptr=traverse(ptr)) {
...
}
```

Bu for yapısı ile ağaçtaki tüm düğümler işlenmiş olur.

Java 5.0' dan sonra gelen gelişmiş for deyimi Iterable interface' ini implement etmiş.
Örneğin adı myList olan bir ArrayList tanımlansın ve string tipinde veriler tutsun.

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

Burada for, for each (herbiri için) anlamı taşımaktadır; listedeki tüm elemanlar sırasıyla myElement'e atanır.

C#'ta **foreach** deyimi bulunmaktadır.

```
List <String> names = new List <String>();
names.add(“ahmet”);
names.add(“ayse”);
names.add(“can”);
...
foreach(String name in names)
        Console.WriteLine(name);
```

8.3.5. Koşulsuz Dallanma

Program akışını, programın içinde başka bir yere yönlendiren deyimlere koşulsuz dallanma deyimleri diyoruz.

Go to, programdaki deyimlerin çalışma akışını kontrol etmek için kullanılan en güçlü deyimdir. Tabi bu durum go to' yu tehlikeli de yapar. Go to kullanımı, programları okunması çok zor hale getirir. Dolayısıyla güvenilirliği azaltır ve bakım maliyetini artırır.

Popüler dillerin çok azında go to yoktur: Java, Python, Ruby.