7. İfadeler ve Atama Deyimleri

İfadeler bir programlama dilinde hesaplamaların tanımlandığı temel yapılardır. Kullanılan dildeki ifadelerin sözdizimsel ve anlamsal olarak anlaşılması bir programcı için çok önemlidir.

Buyurgan dillerde atama ifadelerinin baskın rolü vardır. Bu ifadelerin amacı değişkenlerin veya durumların değerini değiştirmek gibi yan etkiler yaratmaktır. Dolayısıyla programlama dillerinde yapıtaşı, program boyunca değerleri değişen değişkenlerdir.

Fonksiyonel dillerde fonksiyon parametreleri gibi farklı türde değişkenler kullanılır. Bu diller ayrıca isimleri değğerlere bağlayan deyimlere sahiptirler. Bu deyimler atama ifadelerine benzese de yan etkileri bulunmaz.

7.1. Aritmetik ifadeler

Programlama dillerinde aritmetik ifadeler işleçler (operators), işlenenler (operands), parantezler ve fonksiyon çağrılarından oluşur. Bir işlecin tekli (unary) olması o işlecin tek bir işleneninin olduğu anlamına gelmektedir. Ayrıcabir işleç ikili (binary) ve üçlü (ternary) de olabilir ve sırasıyla iki veya üç işlenenin olduğu anlamına gelmektedirler.

Çoğu programlama dilinde ikili işleçler infix'tir, yani işleç iki işlenenin arasında bulunmaktadır. Buna istisna olarak Perl dilinde işleçler prefixtir yani işlenenlerden önce yazılmaktadır.

Aritmetik ifadenin amacı aritmetik bir hesaplama belirtmektir. Böyle bir hesaplama yapılırken iki olay oluşmaktadır:

1. İşlenenleri bellekten almak
2. Bu işlenenler üzerinde aritmetik işlemi gerçekleştirmek

7.1.1. İşleç Hesaplama Sırası

7.1.1.1. Öncelik

Bir ifadenin değeri işleç hesaplama sırasına bağlıdır. Örneğin a+b*c ifadesinde a,b ve c sırasıyla 3,4,5 değerlerine sahip olsun. Eğer işlemleri soldan sağa yaparsak sonuç 35, sağdan sola yaparsak da 23 çıkar. Sadece soldan sağa veya sağdan sola işlem yapmak yerine matematikçilerin çok uzun zaman önce geliştirdikleri işleç önceliğini kullanırız. Dolayısıyla matematikte çarpım işleminin önceliği toplamaya göre daha yüksek olduğundan, çoğu dilde önce çarpma işlemi gerçekleştirilir.

İşleç öncelik kuralları, bir ifadede farklı öncelikleri olan işleçlerin hangi sırada işleme gireceğini kısmen tanımlayan kurallardır. Yaygın olarak kullanılan buyurgan dillerde işleç öncelik kuralları neredeyse

aynıdır çünkü matematik kurallarına dayanırlar. Bu dillerde, eğer tanımlıysa, üs alma işlemi en yüksek önceliğe sahiptir. Daha sonra öncelikleri aynı düzeyde olan çarpma ve bölme, en son da yine aynı öncelik düzeyine sahip toplama ve çıkarma işlemleri gelmektedir.
Çoğu dilde toplama ve çıkarma için tekli operatör kullanımı da vardır. Tekli toplamaya özdeşlik işleci de denir, çünkü işleneni üzerinde herhangi bir etkiye sahip değildir. Bu yüzden gereksizdir. Tekli eksi işareti ise bir işlenenin işaretini değiştirdiği için farklıdır. Java ve C#' taki eksi işareti, short ve byte işlenenleri üstü kapalı olarak int' e çevirmektedir.

Tüm yaygın buyurgan dillerde - işareti bir ifadenin başında veya içinde bir yerde kullanılabilir. Ancak, başka bir işleçle yan yana gelmemesi için parantez içinde kullanılması gerekir. A + (-B) * C ifadesi doğru iken A+ -B * C genellikle yanlıştır.

Örn: -a/b -a*b -a**b ifadelerini inceleyelim. İlk iki ifadede eksi işlecinin öncelik sırası önemsizdir çünkü sonucun ne olduğunu etkilemez. Fakat son ifade için durum farklıdır. Yaygın kullanılan programlama dillerinden sadece Fortran, Ruby, Visual Basic ve Ada'da üs alma işleci bulunmaktadır. Bu dillerin tümünde üs alma eksi'ye göre önceliklidir. Dolayısıyla -a**b ifadesi -(a**b) ifadesine eşittir.

Ruby ve C-tabanlı diller için işleç öncelikleri şöyledir:

	Ruby	C-tabanlı diller
En yüksek	**	postfix ++, --
	unary +,-	prefix ++,--, unary +,-
	*, /, %	*, /, %
En düşük	binary +,-	binary +,-

7.1.1.2. Birleşme

a - b + c - d ifadesini ele alalım. Eksi ve artı işleçleri aynı öncelik düzeyindeyse (programlama dillerinde olduğu gibi), öncelik kuralı hangi işlemin önce yapılacağını belirtmez. Bir ifadede aynı öncelik düzeyine sahip işleçler kullanılmışsa, hangi işlecin önce işlem göreceği sorusuna dilin birleşme kuralları ile cevap bulunur. Bir işleç sağdan veya soldan birleşmeli olabilir. Bunun anlamı, aynı öncelik düzeyine sahip işleçlerin sağdan sola veya soldan sağa doğru işleme girebilmesidir.

Birleşme işlemi çoğu dilde soldan sağa doğrudur. Üs alma işlemi bu durumun dışında kalmaktadır ve sağdan sola doğru işlenir. A ** B ** C işlemi Fortran ve Ruby'da sağdan sola doğru işlenir. Ada'da üs alma işlemi birleşmeli değildir. Dolaısıyla paranteslerle öncelik

belirtilmelidir: A ** (B ** C) veya (A ** B) ** C gibi. Visual Basic dilindeki üs alma işleci ^ soldan birleşimlidir.

APL dilinde işleç önceliği yoktur. Birleşme kuralı sağdan sola doğru uygulanır. Örneğin A x B + C işleminde önce + işareti işleme girer. A, B, C' nin değerleri 3, 4, 5 ise sonuç 27'dir.

A + B + C + D işleminde A ve C'nin çok büyük pozitif sayılar, B ve D' nin de mutlak değerleri çok büyük negatif tamsayılar olduğunu düşünelim. Bu durumda A ve B' yi toplamak bir sorun oluşturmazken A ve C' yi toplamak taşmaya (overflow) sebep olabilir. Bilgisayar aritmetiğinin kısıtları sebebiyle toplama işlemi bu durumda birleşmeli değildir. Demek oluyor ki derleyici bu toplama işlemlerini yeniden bir sıraya sokarsa sonuç değeri etkilenir. Bu problem, eğer değişkenlerin alabileceği değerler önceden tahmin edilirse, programcı tarafından önlenebilir.

7.1.1.3. Parantezler

Programcılar ifadelerin içinde parantezler kullanarak işlem önceliği sırasını değiştirebilirler. Örneğin (A + B) * C ifadesinde *, +'ya göre daha öncelikli olmasına rağmen ilk önce + işleme girer. Programcı işleç öncelik sıralarını hatırlamak zorunda kalmadan parantezleri kullanarak istenen öncelik sırasını belirler. Bu kullanımın dezavantajı ifade yazımının yorucu hale gelmesi ve kodun okunabilirliğinden taviz verilmesidir.

7.1.1.4. LISP ifadeleri

LISP' teki tüm aritmetik ve mantıksal işlemler fonksiyonlarla gerçekleştirilmektedir. C' deki a + b * c ifadesi LISP'te şu şekilde yazılır: (+ a (* b c)). Burada + ve * fonksiyon isimleridir.

7.1.1.5. Koşullu ifadeler

If-then-else deyimi ile koşullu atama ifadeleri gerçekleştirilebilir.

```
Örn: if (count==0)
         average = 0;
     else
         average = sum/count;
```

C tabanlı dillerde bu kod, koşul ifadesi atama ifadesi içinde kullanılarak daha uygun bir şekilde ifade edilebilir. Genel biçimi:

expression_1 ? expression_2 : expression_3

Burada expression_1 boolean ifade olarak yorumlanır. Eğer sonuç doğru ise expression_2'nin, değilse expression_3' ün değeri geçerli olur.

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

Burada ? then : ise else anlamında düşünülebilir. Bu kullanımda hem then hem de else kısımlarının bulunması zorunludur. ? işlecinin üçlü işleç olduğunu görmekteyiz.

7.1.2. İşlenen Hesaplama Sırası

İfadelerin tasarımında daha az konuşulan bir konu da işlenenlerin hesaplanma sırasıdır. İfadedeki değişkenler, değerlerinin bellekten getirilmesi ile işleme alınırlar. Sabitler de bazen aynı şekilde değerlendirilir. Onun dışında bir sabit bir makine dili komutunun bir parçası olabilir ve bellekten getirilmesi gerekmez. Bir işlenen eğer parantez içinde bir ifade ise değeri bir işlenen olarak kullanılmadan önce içerdiği tüm işleçler hesaplanmalıdır.

Bir işlecin işlenenleri yan etkiye sahip değilse işlenenlerin işleme alınma sırası önemsizdir. Dolayısıyla sadece yan etki söz konusu olduğunda ortaya ilginç durumlar çıkabilir.

7.1.2.1. Yan etkiler

Bir fonksiyonun yan etkisi (fonksiyonel yan etki), fonksiyon bir parametresini veya global bir değişkeni değiştirdiğinde oluşur. Global değişken bir fonksiyonun erişebildiği ancak fonksiyonun dışında tanımlanmış olan değişkendir.

a + fun(a) ifadesini ele alalım. Eğer fun(a), a'nın değerini değiştirecek bir etkiye sahip değilse, iki işlenenin işlenme sırasının bu işlemin sonucuna bir etkisi yoktur. Fakat fun(a), a' yı değiştiriyorsa bir yan etki söz konusu olmaktadır. Örnek olarak, fun fonksiyonunun 10 değerini döndürdüğünü ve parameresinin değerini de 20 yaptığını düşünelim.

```
a = 10;
b = a + fun(a);
```

Yukarıdaki ifadelerde eğer a'nın değerinin bellekten daha önce getirildiğini düşünürsek, b = 10 + 10 = 20 olur. Fakat önce ikinci işlenen işlem görürse (yani fonksiyon sonucu) b'nin değeri 30 olmaktadır.

Aşağıdaki C kodu bir fonksiyonun global değişkeni değiştirmesiyle oluşan benzer bir problemi göstermektedir.

```
int a = 5;
int fun1(){
    a=17;
    return 3;
}
void main(){
    a = a + fun1();
}
```

Main içindeki a’ nın değeri a + fun1() ifadesinde hangi işlenenin daha önce işlendiğine bağlı olarak 8 veya 20 olur.

Matematiksel fonksiyonların yan etkilerinin olmadığını belirtmeliyiz. Çünkü matematikte değişken kavramı yoktur. Aynı şey fonksiyonel diller için de geçerlidir. Fonksiyonel dillerde ve matematikte fonksiyonları anlamak, buyurgan dillerde olduğundan çok daha kolaydır.

İşlenen hesaplama sırasının ve yan etkilerin oluşturduğu soruna iki şekilde çözüm bulmak mümkündür. İlki dil tasarımcılarının fonksiyonların yan etki yaratmasına izin vermeyecek şekilde tasarım yapmasıdır. İkincisi de dil tanımında, ifadede bulunan işlenenlerin bir hesaplama sırası içinde işlenmeye zorlanmalarıdır.

Fonksiyonların yan etki oluşturmasını engellemek buyurgan diller için zordur ve programcı için esneklikten ödün vermek anlamına gelir. Örneğin sadece fonksiyonlar bulunan C ve C++ kodunda her fonksiyonun bir değer döndürdüğü durumda yan etkileri yok etmek için birden fazla değer döndürebilmek için değerler bir struct içinde toplanır ve struct döndürülür. Fonksiyonların globallere erişimine de izin verilmemelidir. Fakat etkinliğin önemli olduğu noktada fonksiyonlara parametre göndermek yerine globallerin kullanımı hızı artırmak için kullanılan önemli bir yöntemdir. Java dilinde bu probleme çözüm olrak, işlenenler soldan sağa doğru işlenir.

7.1.2.2. Gösterimsel Şeffaflık ve Yan Etkiler

Eğer programdaki aynı değere sahip iki ifade çalışmayı etkilemeyecek şekilde programın herhangi bir yerinde birbiri yerine kullanılabiliyorsa o programda gösterimsel şeffaflık olduğunu söyleyebiliriz. Gösterimsel şeffaflığı olan bir fonksiyonun değeri tamamen parametrelerine bağlıdır. Gösterimsel şeffaflık ve yan etkiler arasındaki ilişki şu örnekle gösterilebilir:

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

Yan etkisi olmayan bir fonksiyon için result1 ve result2’ nin değerleri birbirine eşit olacaktır. Fakat fun’ın b veya c’ ye 1 eklemek gibi bir yan etkisi olsaydı sonuçlar eşit olmazdı ve gösterimsel şeffaflık ihlal edilmiş olurdu.

Gösterimsel şeffaflığın birçok avantajı bulunmaktadır. En önemlisi, gösterimsel şeffaflığa sahip programlarda anlamsallığın çok daha kolay anlaşılmasıdır. Değişken kullanmamaları sebebiyle saf fonksiyonel diller de gösterimsel olarak şeffaftırlar.

7.2. İşleç Aşırı Yüklemesi (operator overloading)

Aritmetik işleçler genellikle birden fazla amaç için kullanılırlar. Örneğin + işleci genellikle integer veya floating point toplaması için kullanılır ancak Java’da aynı zamanda stringleri birleştirir (catenation). İşleçlerin bu tip çoklu kullanımlarına işleç aşırı yüklemesi diyoruz.

Aşırı yüklemenin tehlikeli olabileceğini göstermek için C’de kullanılan & işlecini ele alalım. İkili işleç olarak &, mantıksal bit işlemlerinde AND işlemi için kullanılır. Fakat tekli bir işleç olarak kullanıldığında anlamı tamamen farklıdır. Birlikte kullanıldığı işlenenin adresini belirtmektedir. Burada karşımıza iki sorun çıkmaktadır: Birbiri ile ilgisi olmayan işlemler için aynı işleci kullanmak okunabilirliğe zarar verir. Bir de örneğin, AND işlemi yapacakken yanlış yazım sonucu &’i bir değişkenin önüne yazarsak bu hata derleyici tarafından tespit edilemez çünkü sözdizimsel olarak bir hata değildir.

Tüm programlama dillerinde aynı ciddiyette olmasa da - işlecinin ikili ve tekli kullanımıyla ilgili benzer bir sorun bulunmaktadr. Bu durumda da - işlecinin yanlışlıkla yazılması derleyici tarafından hata olarak görülmez. Ancak - işlecinin her iki kullanımı da birbiri ile alakalı olduğundan okunabilirlik açısından sakıncalı bir durum yaratmaz.

7.3. Tip Dönüşümleri

Tip dönüşümü ya genişletme ya da daraltma şeklinde gerçekleştirilir. Bir daraltma dönüşümü bir değeri orijinal tipinin tüm değerlerini tutamayacak bir tipe dönüştürmektir. Java’ da double bir değeri float’a dönüştürmek bir daraltma dönüşümüdür. Çünkü double’ ın değer aralığı float’ ınkinden çok daha geniştir. Genişletme dönüşümü de bir değeri orijinal tipinin alabileceği tüm değerleri kapsayan bir tipe dönüştürme işlemidir. Java’ da int’ i float’ a çevirmek böyle bir işlemdir. Genişletme işlemleri neredeyse her zaman güvenlidir. Ancak daraltma dönüşümleri için aynısı söylenemez.

Genişletme dönüşümü güvenli olsa da kesinlik kaybına yol açabilir. Integer ve float 32-bit olmalarına rağmen, Integer için en az 9 float için ise 7 bit kesinlik için ayrılmıştır. Bu durumda int'ten float'a dönüşüm genişletme olmasına rağmen 2 digitlik bir kesinlik kaybına yol açmaktadır.

7.3.1. Zorlama (Coertion)

Bir aritmetik ifadede işlecin farklı tipte işlenen kullanıp kullanmayacağı bir tasarım kararıdır. Karışık mod ifadeleri denen bu tarz ifadelerin bulunabildiği diller, tip dönüşümünü kapalı bir şekilde gerçekleştirmek zorundadır çünkü bilgisayarlar iki farklı işlenen üzerinde işlem yapamazlar.

Örn: Java'da

```
int a;
float b, c, d;
...
d = b * a; <--- Burada yanlışlıkla c yerine a yazdığımızı varsayalım.
```

Java' da karışık mod ifadeler geçerli olduğu için derleyici bunu hata olarak görmez ve int olan a' nın değerini ifloat' a dönüştürür. Bu ifadeler Java'da geçerli olmasaydı bu ifade derleyici tarafından hata olarak yakalanırdı.

7.3.2. Açık Tip Dönüşümü

Çoğu dil hem genişletme hem de daraltma için açık tip dönüşümü yapma imkanı sunar. Bazı durumlarda daraltma dönüşümü dönüştürülen nesnenin değerinde çok büyük bir değişiklik yaratacaksa uyarı mesajları üretilir.

C-tabanlı dillerde açık tip dönüşümüne "cast" denmektedir. Örneğin, (int) angle ifadesinde angle adlı değişkenin tipi int'e çevrilmek isteniyor. Dönüştürülmek istenen tip parantez içinde değişken adının veya ifadenin önünde yazılır. Parantez kullanımının sebebi C' de iki kelimelik tiplerin bulunmasıdır, long int gibi.

7.3.3. İfadelerdeki Hatalar

Bir ifade işlenirken birçok hata oluşabilir. Eğer dil, tip kontrolü gerektiriyorsa tip hataları ayıklanır. En genel hatalardan biri, bir ifade sonucunun yüklenmesi gereken bellek hücresi ile uyumsuzluğudur. Bu durumlarda taşma (overflow) veya küçük kalma (underflow) denilen, değerin çok büyük veya çok küçük olması gibi hatalar oluşur. Sıfıra bölme hatası da bir ifade hatasıdır. Bu tip

hatalar çalışma zamanı hatalarıdır ve istisnalar (exceptions) olarak adlandırılırlar.

7.4. İlişkisel ve Mantıksal İfadeler

7.4.1. İlişkisel işleç

İki işlenenin değerlerini karşılaştırmak için kullanılan işlece ilişkisel işleç denir. Bir ilişkisel ifade, bir ilişkisel işleç ve iki işlenenden oluşmaktadır. İlişkisel ifadenin değeri, eğer o dilde tanımlı ise Boolean bir değerdir.

İlişkisel ifadenin doğruluğuna veya yanlışlığına karar verilen işlem işlenenlerin tipine bağlıdır. İşlenenler integer gibi basit veya string gibi karmaşık tipler olabilir.
Eşitlik veya farklılık için kullanılan işleçler programlama dilleri arasında farklılık gösterebilir. İki işlenenin birbirinden farklı olduğu C tabanlı dillerde != ile kontrol edilir. Ada dilinde bu işleç /=, Lua için ~=, Fortran 95+ için .NE. veya <> işleçleridir. JavaScript ve PHP' de iki ek ilişkisel operatör bulunur. Bunlar == ve != ile benzer anlam taşırlar Ancak === ve !== ile tip zorlaması ortadan kalkar. Örneğin JavaScript' te “7” == 7 true iken “7” === 7 false olur.

7.4.2. Boolean İfadeler

Boolean ifadeler, boolean değişkenler, boolean sabitler, ilişkisel ifadeler ve boolean işleçlerden oluşurlar. İşleçler genelde AND, OR, NOT işlemlerini tanımlarlar. Boolean işleçler sadece boolean işlenenler alırlar ve boolean değer üretirler.

Boolean cebirde OR ve AND eşit önceliğe sahip olmak zorundadır. Bunu temel alan Ada dilinde AND ve OR eşit öncelik düzeyine sahiptir. Fakat C tabanlı dillerde AND için OR' a göre daha yüksek bir öncelik değeri atanmıştır.

C tabanlı dillerde aritmetik ve Boolean işleç öncelikleri:

En yüksek	postfix ++, --
	prefix ++,--, unary +,-, !
	*, /, %
	binary +,-
	<, >, <=, >=
	==, !=
	&&
En düşük	||

C dilinde a>b>c ifadesi geçerli bir ifadedir. Burada işleçler soldan sağa doğru işlendiği için önce a>b ifadesinin sonucu bulunur ve o sonuç c ile karşılaştırılır. Burada b>c ifadesi hiçbir zaman kontrol edilmez.

Okunabilirlik açısından bir dilde boolean tipinin bulunması gereklidir. Aksi taktirde boolean yerine nümerik tipler kullanılacaktır. Boolean işlenenler yerine nümerik tipler kullanıldığında bazı hatalar yakalanamayabilir.

7.5. Kısa Devre

Bir ifadede sonuç tüm işleçleri ve işlenenleri hesaplamadan elde edilebiliyorsa buna o ifadenin kısa devre hesaplaması diyoruz.

(13 * a) * (b / 13 – 1) ifadesinde eğer a = 0 ise bu ifadenin sonucu, (b / 13 – 1) ifadesinin sonucunun ne olduğundan bağımsızdır. Dolayısıyla a'nın sıfır olduğu biliniyorsa (b / 13 – 1) ifadesini hesaplamaya gerek yoktur. Ancak aritmetik işlemlerde bunu farketmek kolay değildir. O yüzden uygulanmaz.

(a >= 0) && (b < 10) boolean ifadesinin değeri, eğer a < 0 ise ikinci bölümün sonucundan bağımsızdır. Çünkü FALSE && (b < 10), b'nin ne olduğundan bağımsız olarak FALSE değerine sahiptir. Dolayısıyla a < 0 iken, && (b < 10) işlemlerinden hiçbirini yapmaya gerek yoktur. Aritmetik ifadelerin aksine burada kısa devre kolayca tespit edilir.

Kısa devrenin hesaplanamadığı durumlardaki potansiyel sorunu bir örnekle görelim:

Örn. Java' da kısa devre olmadığını varsayalım.

```
index = 0;
while ((index < listlen) && (list[index] != key))
    index = index + 1;
```

Döngüde liste sonuna ulaşıldığını ve key'in listede olmadığını düşünelim. İndex artışı yapıldı. Eğer kısa devre yoksa ilk ifade kontrol edilir ve sonra ikinci ifade de kontrol edilir. İkinci ifadedeki list[index] ifadesinde liste boyutları dışına çıkılmış olur ve program subscript out of range hatası ile sonlanır.

7.6. Atama ifadeleri

= veya := genellikle atama için kullanılan işleçlerdir.

7.6.1. Koşullu Hedef

Perl dilinde,

($flag ? $count1 : $count2) = 0 ifadesi aşağıdaki if deyiminin kısaltılmış şeklidir.

```
İf ($flag) {
    $count1 = 0;
} else {
    $count2 = 0;
}
```

7.6.2. Unary Atama

sum = ++ count; => count = count +1; sum = count;
sum = count ++; => sum = count; count = count + 1;

--count++ => iki tane unary işleç var, sağdan sola doğru işlenir.