Bir Mantık Programlama Dili

PROLOG

(Bölüm III)

Selçuk TARAL*

Geçen bölümü *özyinelemeli* (recursive) kuralların yazılışlarına örnekler vererek kapatmıştık. Bu bölümde Prolog'un kullandığı veri yapılarını açıkladıktan sonra, *özyinelemeli* veri yapıları ve özellikle **liste** (list) üzerinde duracağız. Bu bölümü liste veri yapısı ile ilgili örnek programlar vererek bitireceğiz. Klâsik programlama dillerinin aksine Prolog'un veri tipleri basit ve esnek bir yapıya sahiptir. Klâsik programlama dillerinde bilgi önceden tanımlanmış belli formatlarda saklanır. Bu nedenle, programa yeni bir bilgi eklemek oldukça güçtür. Prolog'ta ise değişik türden bilgilerin temsili, bunları açıklayan gerçeklerin programa eklenmesiyle, kolaylıkla sağlanır.

Prolog karmaşık nesneleri birkaç basit veri tipi kullanarak sistematik bir şekilde tanımlama olanağını verir. Daha önce verdiğimiz örneklerdeki ibrahim, süleyman gibi *sabit isimlere* **atom** denir. Yüklem isimleri ve atomların küçük harfle başladıklarını biliyorsunuz. Değişkenler ise, klâsik programlama dillerinde olduğu gibi, bilgisayarın belleğinde verinin saklandığı yerin (store location) adı değil, *herhangi* ama *tek* bir *değer alabilen* bir nesnedir. Değişken isimlerin büyük harfle başlamalarının gerektiğini daha önce belirtmiştik. İlerde göreceğimiz gibi, bazen bir değişkenin değeri bizim için bir önem taşımaz. Bu durumlarda değişkeni bir alt çizgiyle ('_') göstereceğiz.

Şimdi Prolog'un tek veri yapısı olan **terimi** (term) tanımlayalım: *Atomlar ve değişkenler birer terimdir.* Ayrıca *karmaşık terimler* de birer terimdir. *Karmaşık bir terim, bir funktor ve bir veya daha fazla argümandan oluşur.* Argümanların da herbiri yine birer terimdir. Karmaşık bir terimin genel yapısını şöyle gösterebiliriz: **$f(t_1,t_2,...,t_n)$.** Burada f funktorun ismi, t_i'ler ise argümanlardır. Örneğin, **s(0), sıcak(süt), baba(ali,veli),** ve **ders(prolog,zaman(salı,9,11),hoca(selçuk,taral), yer(ana,3a))** birer karmaşık terimdir. Son örnek terim bir gerçek olarak verildiği zaman, bize şu bilgiyi verir: *Prolog dersi salı günleri saat 9 ile 11 arasında ana binanın 3a nolu odasında selçuk taral tarafından verilmektedir.* Bu örnekte birinci argüman bir atom, diğerleri ise yine karmaşık terimlerdir.

Şimdi özyinelemeli veri *tiplerine açıklayalım. Tip,* sonsuz sayıda da olabilen, terimler kümesine verilen isimdir. Şimdiye kadar basit tipleri gördük. Örneğin, daha önce gördüğünüz erkek yüklemi *erkek tipini* tanımlar. Özyinelemeli tipler (recursive types) ise özyinelemeli programlar aracılıyla tanımlanırlar. Örneğin, aşağıda verilen basit program *sonsuz bir yapı* olan *doğal sayıları* tanımlar:

doğal_sayı(0).
doğal_sayı(s(X)) :- doğal_sayı(X)

Birinci tümce 0 sembolüyle gösterilen sabitin bir doğal sayı olduğunu deklare eder. İkinci ve özyinelemeli tanımlanan kural ise, X değişkeninin bir doğal sayı olması halinde, ondan *sonra gelen doğal sayıyı* göstermek için kullanabileceğimiz **s(X)** teriminin de bir doğal sayı olduğunu belirtir. Burada X'den bir sonra gelen doğal sayıyı, s(X) yerine, örneğin **ardıl(X)** terimiyle de gösterebilirdik. Böylece kuralın yinelemeli (iterative) çalışmasıyla doğal sayılar tanımlanmış olur: 0 bir doğal sayı olduğu için s(0) da bir doğal sayıdır, s(0) bir doğal sayı olduğu için s(s(0)) de bir doğal sayıdır vb. **(0, s(0), s(s(0)), s(s(s(0))),...).** Bu programa doğal sayıları sırayla göstermesi için nasıl bir soru yöneltirdiniz?

Bu tanım bazı okuyucularımızın aklına, *'yoksa Prolog ile basit aritmetik yapmak için önce doğal sayıları tanımlamamız mı gerekecek?'* gibi bir soru getirmiş olabilir. Böyle bir şeye tabii gerek yok; Prolog ile aritmetik yaparken bildiğiniz rakamları kullanabilirsiniz. Aslında s(0), 1 rakamının, s(s(0)), 2'nin, s(s(s(0))), 3'ün vb. *sembolik olarak farklı yazılış şekillerinden* ibarettir. Bu programı vermekteki esas amacımız, sonsuz veya *sonlu ama sayısı belli olmayan* (yani potansiyel sonsuz) yapı tiplerini özyinelemeli kurallarla nasıl tanımlayabileceğimizi göstermektir.

Yukarıda bahsettiğimiz **liste** (list) iki argümanı olan (binary) bir terimdir. Listenin birinci argümanı bir nesneler kümesinin (domain) elemanıdır; Bu bir rakam, bir atom veya kendisi de bir karmaşık terim, örneğin yine bir liste olabilir. Listenin ikinci argümanı ise özyinelemeli olarak listenin *geri kalan elemanlarının oluşturduğu listedir.* Bir listenin birinci elemanına listenin **başı** (head), ikinci elemanına ise **kuyruğu** (tail) denir.

Doğal sayıların tanımında olduğu gibi, listenin tanımında da özyinelemeyi (recursion) sona erdirmek için, bir sabit sembol gereklidir. Bu sembole **boş liste** (empty list) denir ve iki ayrı şekilde gösterilebilir: **nil** veya pratikte kullanılan yazılış tarzıyla: **[ ].** Yukarıda verilen tanıma göre, örneğin tek elemanı olan bir listenin kuyruğu boş listedir. *Liste teriminin funktorunun "tarihsel" ve genel temsili bir nokta ('.') işaretidir.* Başı X, kuyruğu Y olan bir liste şöyle gösterilir: **.(X,Y).** Ancak aynı liste için pratikte, boş liste de olduğu gibi, **[X|Y]** yazılış şekli (syntax) kullanılır. *Boş liste, içinde hiçbir eleman olmadığı için, baş ve kuyruk şeklinde ikiye bölünemez.*

Örneğin, tek elemanı 1 rakamı olan bir liste, bir terim olarak yazılışıyla, **.(1,[ ]),** pratik yazılış tarzıyla ise, **[1 | [ ]]** şeklinde gösterilir. Aynı listenin en basit yazılış şekli ise şöyledir: **[1].** Birinci ve ikinci elemanları sırasıyla, a ve b olan bir listeyi, *birbirlerine eşdeğer,* ama ayrı şekillerde aşağıda verildiği gibi yazabiliriz:

* *TÜBİTAK, Bilgi İşlem Daire Başkanı.*

.(a,.(b,[ ])) [a|[b|[ ]]] [a,b]

Şimdi, doğal sayılara benzer olarak, listeyi de özyinelemeli bir programla tanımlayalım:

```
liste([ ]).
liste([X|Y]) :- liste(Y).
```

Programın birinci tümcesi olan gerçek, [ ] *ile gösterilen sabit sembolün bir liste olduğunu deklare eder.* Programın ikinci tümcesi ise, listenin yukarıda verdiğimiz tanımının bir kural olarak yazılmasından başka bir şey değildir; *Y bir liste ve X herhangi bir terimse,* [X|Y] *veya klâsik yazılışıyla* .(X,Y), yine bir listedir. Bu kural bize aynı zamanda, *mevcut bir listeden yeni listelerin nasıl türetilebileceğini* göstermektedir. *Başlangıç noktası* olarak kullanabileceğimiz bir listeden başlayarak, sonlu sayıda eleman içeren bir listeyi, kuralı yinelemeli kullanarak üretebiliriz. Bu başlangıç noktası, tahmin edebileceğiniz gibi, *boş listeden başkası değildir.* Tıpkı, doğal sayıların tanımında, 0 sembolünü ilk doğal sayı olarak kabul etmemiz ve diğer doğal sayıları onun üzerine inşa etmemiz gibi.

Aynı programa verilen bir terimin liste olup olmadığını da sorabiliriz. Örneğin, '[1,2,3] ile gösterilen yapı bir liste midir?' anlamındaki, **'?- liste([1,2,3]).'** sorusunun yanıtı 'yes' olacaktır. Prolog'un bu sonuca varmak için attığı adımları aşağıda verdik. Her adımın sağında, hedefin kuralın başıyla birleşmesini sağlayan X ve Y değerlerini göreceksiniz.

```
1. liste([1|[2,3]]) ← liste([2,3])     X = 1, Y = [2,3]
2. liste([2|[3]])   ← liste([3])       X = 2, Y = [3]
3. liste([3|[ ]])   ← liste([ ])       X = 3, Y = [ ]
4. liste([ ]).        doğru
```

Burada ':-' işareti yerine matematikte kullanılan ok işaretini kullandık. Birinci adımda Prolog, verilen hedefi önce programın gerçeğiyle birleştirmeye çalışır. Ancak, [1,2,3] ile [ ] birleşmezken, *ve programın özyinelemeli kuralı devreye girer.* Kuralın başındaki değişkenlerin yukarıda verilen değerlere bağlanması sonucu, hedef kuralın başıyla birleşir ve yeni hedef, **'?- liste([2,3]).'** olur. Diğer bir deyişle, [2,3] ile gösterilen terim bir listeyse, [1,2,3] terimi de bir listedir ve bu nedenle, hedefe varmak için [2,3] teriminin bir liste olduğunu göstermemiz yeterlidir. Aynı şekilde, İkinci adımda da gerçekle birleşme sağlanamaz ve kuralın çalışması sonucu yeni hedef, **'?- liste([3]).'** olur. Bu hedef, kuralın üçüncü kez çalışması sonucu, **'?- liste([ ]).'** sorusuna indirgenir.

Şimdi artık programın gerçeği devreye girer. Buraya kadar, ilk hedefi kural aracılıyla *adım adım eksilterek* (goal reduction), liste olduğunu bildiğimiz bir yapıya erişmiştik. (Sonunda boş listeye varacağımızdan neden emindik?) Şimdi, artık ters yönde giderek ilk sorumuza dönebiliriz: [ ] bir listedir (4. adım), [ ] liste olduğu için [3] de bir listedir (3. adım), [3] bir liste olduğu için [2,3] de bir listedir (2. adım), ve son olarak, [2,3] bir liste olduğu için [1,2,3] de bir listedir (1.adım). *Zaten biz de bunu göstermek istemiştik.*

Şekil 3'de liste ile ilgili iki temel yüklemin tanımlarını bulacaksınız. Birinci yüklem, bir elemanla bir liste arasında bulunabilecek *'üyelik'* ilişkisiyle ilgilidir. İkinci yüklem ise, iki listenin yan yana koyularak nasıl yeni bir liste elde edilebileceğini gösterir.

```
üye(X,[X|Xs]).
üye(X,[Y|Ys]) :- üye(X,Ys)

ekle([ ],Ys,Ys).
ekle([X|Xs],Ys,[X|Zs]) :- ekle(Xs,Ys,Zs).
```

Şekil 3'de verilen **üye** (member) yüklemi bir listeye üyelik ilişkisini tanımlar. Programın gerçeğini; *'X bir listenin başına eşit ise, X bu listenin bir üyesidir',* kuralını ise; *'X Ys ile gösterilen bir listenin üyesi ise, kuyruğu Ys olan herhangi bir listenin de üyesidir'* şeklinde okuyabiliriz. Aynı programın yordamsal okunuşu ise şöyledir: *'X'in bir listenin elemanı olduğunu göstermek için, önce listenin başına eşit olup olmadığına bakın, eğer eşit değilse* (eşitse hedefe varmış oluruz), *aynı işlemi bu kez listenin kuyruğu için yapın.'* Örneğin, bu programı **'?- üye(3,[1,2,3]).'** hedefini verirsek, Prolog 'yes' yanıtına aşağıda verilen adımların sonunda varır:

```
1. üye(3,[1|[2,3]]).              yanlış (3 ≠ 1)
2. üye(3,[1|[2,3]]) ← üye(3,[2,3])
3. üye(3,[2|[3]]).                yanlış (3 ≠ 2)
4. üye(3,[2|[3]]) ← üye(3,[3])
5. üye(3,[3]).                    doğru (3 = 3)
```

Sonuç, '3 sayısı [1,2,3] listesinin bir elemanıdır' anlamında, 'yes' olacaktır. Yüklemin gerçeğiyle kuralının birlikte işleyişine dikkatinizi çekmek istiyoruz. Verilen eleman listenin başına eşit değilse (1.adım), kural çalışır ve üyelik arama işi bu kez listenin kuyruğuyla tekrar edilir. Örneğin, Prolog 2. adım sonucu ortaya çıkan **'?- üye(3,[2,3]).'** hedefini sağlamak için yine programın gerçeğini dener. Yeni listenin ilk elemanı da gerçeği sağlamıyorsa (3. adım) kural yeniden devreye girer. Listenin elemanlarının sayısı sonlu olduğu için, bu eksiltme işlemi sonunda boş listeye ulaşılacaktır. (Prolog '?-üye(a,[b,c,d,e])' sorusuna nasıl 'no' ile yanıtlar?)

İki listeyi birbirine ekleyerek (append) yeni bir liste oluşturan **ekle** yükleminin yazılışı biraz daha karmaşıktır. Örneğin, **'?- ekle([a,b,c],[d,e], Sonuç)'** hedefini verdiğimizde, alacağımız yanıt **Sonuç = [a,b,c,d,e]** olacaktır. Bu programın işleyişini gelecek bölümde açıklayacağız.

(Devam edecek.)