Bilgisayar Bilimi Köşesi

Chris Stephenson ve Ali Nesin*
cs@cs.bilgi.edu.tr, anesin@bilgi.edu.tr

Özyinelemeli Fonksiyonlarla
Gerçekten Sıralayalım

Karışık bir liste halinde verilmiş şehir nüfusları, öğrenci notları gibi sayısal verileri olabilecek en kısa zamanda sıralayacağız. Ayrıca, verilen bir sayı listesini küçükten büyüğe doğru **gerçekten** sıralayan bir bilgisayar programı yazacağız. Böylece bu köşede ilk kez program örnekleri göreceğiz. Bizim yaptıklarımızı siz de evinizde aynen yapabilirsiniz, hiçbir tehlikesi yoktur.

Çok açık ve anlaşılır programlar yazacağız, programlama görmemiş birinin bile anlayacağını umuyoruz.

Scheme Dili. Programımızı **Scheme** diye adlandırılan bir programlama dilinde yazacağız. Bu dilin sözdizim kuralları hem çok az hem de çok basittir; ayrıca ünlü mantıkçı Alonzo Church'un geliştirdiği Lambda Calculus'ün birebir uygulanmasıdır. Dilin dayanağı, her türlü işlemi özyinelemeli (tümevarımsal, recursive) fonksiyonları kullanarak yapmak düşüncesidir.

Alonzo Church
(1903-1995)

Bu yazıdaki programları bilgisayarlarında gerçekten denemek isteyenler güzel bir Scheme ortamını http://www.drscheme.org/ adresinden bilgisayarlarına yükleyebilirler. Windows ve Linux versiyonları mevcut. Ama bu yazıyı anlamak için illa programı yüklemek gerekmez.

Scheme, girilen bir ifadenin değerini yanıt olarak verme temeline dayanır. Programa girdi olarak, sözgelimi, $a^2 + 4$ gibi bir ifade ve a değeri verilir; program, verilen bu ifadeye karşılık gelen değeri hesaplayıp bunu çıktı olarak verir. Bizim sıralama programlarımızda, girilen ifade sıralanmamış herhangi bir liste, alınan çıktı ise bu listenin sıralanmış hali olacak.

Basit İşlemler. Scheme, ">" simgesinden sonra bir ifade girmenizi bekler. Örneğin:

```
> 42
42
```

Bu denemeden, tamsayıların kendi değerlerine sahip olduklarını anladık, olması gerektiği gibi, sonuç yine girdinin aynısı oldu.

Scheme dilinde bazı sabitlere ad verebiliriz. Örneğin 3 sayısını bundan böyle *a* simgesiyle göstermek isteyelim. Bunun için

```
> (define a 3)
```

komutu yeterli. "Define" İngilizce "tanımla" demek. Birçok programlama dilinde olduğu gibi bu dilde de komutlar İngilizce. Şimdi programdan *a* değerini istersek, yanıt olarak 3 değerini alırız.

```
> a
3
```

Aynı şekilde *b*'yi 42 olarak tanımlayıp programımızdan b değerini isteyelim:

```
> (define b 42)
> b
42
```

Simgelerimiz illa tek harf olmak zorunda değil, yeter ki harfler arasında mesafe bırakmayalım:

```
> (define çşt 42)
> çşt
42
```

Şimdi bazı aritmetik işlemler yapalım:

```
> (+ 27 15)
42
```

Buradaki + 27 15 aslında 27 + 15 demektir. Scheme dilinde **Polonya notasyonu** kullanılır, yani işlemin simgesi en başa, işlemin uygulanacağı terimler işlem simgesinin hemen ardına yazılır. Bu sayede parantezlerden kurtulmuş oluruz. Örneğin Polonya yazılımında "– + 4 5 2" yazılımı (4 + 5) – 2'ye tekabül

* İstanbul Bilgi Üniversitesi öğretim üyesi.

eder. 4 + (5 – 2) yazmak için ise + 4 – 5 2 yazarız.

Alıştırma olarak 14'le *a*'yı, yani 3'ü çarpalım:

```
> (* 14 a)
42
```

Ardından *b*'yi *a*'ya bölelim:

```
> (/ b a)
14
```

Scheme İfadeleri. Scheme ifadeleri, basit ifadelerden zora doğru tümevarımla tanımlanır. Genel olarak, Scheme ifadeleri parantez içine alınmış daha küçük ifadelerden oluşur. Bu ifadeler, "atom"lar (tamsayılar ve aşağıda göreceğimiz üzere "define" sözcüğüyle tanımlanmış simgeler) ve yine parantezli Scheme ifadeleri olabilir.

Scheme dilinde yeni bir işlemin nasıl tanımlanacağını görelim. Örnek olarak makinaya bir sayının karesini almayı öğretelim :

```
> (define kare (lambda (x) (* x x)))
```

Buradaki terimleri teker teker gözden geçirelim: "define" yazarak programa yeni bir fonksiyon tanımlayacağımızı söylüyoruz. "kare" adını biz uydurduk, bambaşka bir ad da olabilirdi, tanımlayacağımız fonksiyonun adını söylüyor. lambda (x), tanımlayacağımız kare fonksiyonunun değişkeninin *x* olacağını söylüyor. En sondaki "* x x" terimi ise Polonya notasyonunda *x*'le *x*'in çarpılacağını söylüyor. Birkaç deneme yapalım:

```
>(kare 7)
49
>(kare (kare 3))
81
>(* (kare 3) (kare 2))
36
```

Koşullu İfadeler. Kullanışlı programlar yazabilmek için öğrenmemiz gereken bir konu daha var. "cond" simgesiyle, bir ifadeye farklı koşullarda farklı değerler atayabiliriz. Aşağıdaki satır, eğer $u > 4$ ise 5 değerini, eğer $u = 1$ ise 99 değerini, diğer durumlarda da 6 değerini verir:

```
> (define u 2)
> (cond ((> u 4) 5) ((= u 1) 99) (else 6))
```

Biz *u*'ya 2 değeri verdiğimizden, yukardaki program yanıt olarak 6 değerini verecektir. Aşağıdaki program ise $c = 1$ için 99 değerini verecektir.

```
> (define c 1)
> (cond ((> c 4) 5) ((= c 1) 99) (else 6))
99
```

"cond" komutu, iki terimden oluşan ifadelerin bir listesini alır, sonra sırayla bu listelerin (> c 4) gibi ilk ifadelerine bakar, eğer buradaki koşul gerçekleşmişse, sonuç bu ifadenin ikinci terimidir (örnekte 5). Eğer koşul gerçeklenmemişse, bir sonraki ifadenin koşuluna bakar. Koşulların hiçbiri gerçeklenmediğinde, en sondaki "else" ifadesi sonuç olarak alınır. Böylece bir yanıt alınması garanti edilir. Ayrıca, bir sonraki örnekte olduğu gibi, koşullar örtüşebilir de, öncelik listedeki ilk koşula verilir.

```
>(define d 8)
>(cond
    ((> d 4) 5)
    ((= d 8) 7)
    ((= d 1) 99)
    (else 6))
5
```

Artık *n*! (faktoriyel) hesaplayan bir program yazabiliriz. *n*!'in tümevarımsal tanımını anımsatalım:

$$n! = \begin{cases} 1 & \text{eğer } n = 1 \text{ ise} \\ n \times (n-1)! & \text{eğer } n > 1 \text{ ise} \end{cases}$$

Bu tanımı aynen programa aktaralım:

```
> (define faktoriyel
    (lambda (n)
    (cond ((<= n 1) 1)
    (else (* n (faktoriyel (- n 1)))))))
> (faktoriyel 7)
5040
```

Yukardaki program, faktoriyel fonksiyonunu tanımlıyor. Eğer $n \leq 1$ ise 1 yanıtını veriyor, değilse *n* ile tümevarımla hesapladığı $(n - 1)!$ sayısını çarpıyor.

Liste. Listeler üzerinde çalışarak sıralama programları yazmaya başlamak için yeterince bilgiye sahibiz. Scheme dilinde, bir **liste** ya boş listedir ve "( )" ile gösterilir, ya da bir terim (listemizin birinci terimi) ve bir başka listeden (listemizin geri kalan terimleri, yani kuyruğundan) oluşan sıralı bir ikili gruptur. Örneğin, (1 3 8 7) listesini Scheme (1 (3 8 7)) ikilisi olarak algılar. (3 8 7) listesini de (3 (8 7)) ikilisi olarak algılar. Bu böyle devam eder. Sonunda,

(1 3 8 7) = (1 (3 (8 (7 ()))))

bulunur. Scheme programına göre (1 3 8 7) listesinin birinci terimi 1 sayısı, ikinci terimi (3 8 7) dizisidir.

Scheme programında '(1 3 8 7) yazıldığında (başındaki tırnak önemli), Scheme bunun aslında

(1 (3 (8 (7 '())))) listesi olduğunu anlar ve bu bize hatırı sayılır bir kolaylık sağlar:

```
> '(1 3 8 7)
(1 3 8 7)
```

Bu listenin başına bir terim eklemek için "cons" komutunu kullanırız:

```
> (cons 8 '(1 3 6 4 2))
(8 1 3 6 4 2)
```

Dileseydik, yukardaki listeyi

```
> (cons 8 (cons 1 (cons 3 (cons 6 (cons 4 (cons 2 '()))))))
```

olarak da yazabilirdik, ama hamallık yapmış olurduk.

Listelere adlar verebiliriz:

```
>(define e (cons 1 '(7 5 3)))
>e
(1 7 5 3)
>(define f (cons 1 '()))
>f
(1)
```

Bizim listelerimizin terimleri tamsayılar olacak. "cons", yukarda gördüğümüz üzere ikili grupları oluşturur, "car" bu ikilinin ilk terimini, "cdr" ise ikinci terimini ifade eder:

```
>(car '(1 4 5))
1
>(cdr '(5 6 4))
(6 4)
>(define g (cons 1 '(7 5 3)))
>(car g)
1
>(cdr g)
(7 5 3)
```

Listeyi İkiye Bölmek. Şimdi daha ilginç bir şey yapalım. Bir listenin birinci, üçüncü, beşinci, ..., yani tek endisli terimlerinden yeni bir dizi yaratalım. Örneğin (8 3 2 5 7 2 3 1 2) girildiğinde, program bize (8 2 7 3 2) versin.

Bu (8 2 7 3 2) listesi, (8 3 2 5 7 2 3 1 2) listesinin birinci terimi olan 8'le başlar ve bu ilk terimi attığımızda geri kalan (3 2 5 7 2 3 1 2) listesinin çift endisli terimleriyle devam eder. Genel olarak, tek endisli terimlerin listesi, listenin birinci terimiyle başlar ve listenin geri kalanının çift endisli terimleriyle devam eder. Demek ki, ayrıca, bir listenin çift endisli terimlerini veren bir programa da ihtiyacımız var. Bu program, (8 3 2 5 7 2 3 1 2) girildiğinde, bize (3 5 2 1) listesini vermeli. Elbette buradaki (3 5 2 1) listesi, (8 3 2 5 7 2 3 1 2) listesinden 8'i attığımızda geri kalan listenin tek endisli terimlerinden oluşan listedir.

Artık "tekler" ve "çiftler" adını vereceğimiz iki program yazabiliriz. Önce tekler programını yazalım:

```
>(define tekler
    (lambda (x) (cons (car x) (çiftler (cdr x)))))
```

Yukardaki program, x listesinin birinci terimi olan car x'le başlar ve x'in geri kalanının çift endisli terimleriyle devam eder.

Henüz "çiftler" programını yazmadık. Birazdan yazacağız. Daha tekler'le işimiz bitmedi, çünkü yukardaki tekler programı ne zaman biteceğini bilmiyor. Ana koşulu eklememiz gerek. Boş bir listedeki tek endisli terimlerin listesi yine boştur. "null?" ifadesi, listenin boş olup olmadığını kontrol eder.

```
> (define tekler
(lambda (x) (cond ((null? x) '( ))
    (else (cons (car x) (çiftler (cdr x)))))))
```

Böylece eğer x boş listeyse sonuç gene boş liste olur.

Çiftler programınına gelince:

```
> (define çiftler
    (lambda (x) (cond ((null? x) '( ))
            (else (tekler (cdr x))))))
```

Deneyelim:

```
> (tekler '(1 2 3 4 5 6 7 9))
(1 3 5 7)
> (çiftler '(1 2 3 4 5 6 7 9))
(2 4 6 9)
```

Böylece bir listeyi uzunlukları hemen hemen eşit iki ayrık parçaya ayırmış olduk. Listede tek sayıda terim olduğunda parçaların birinde bir terim fazla olacak ama bu bir sorun çıkarmayacak.

Listeyi ikiye bölmek sıralama konusunda ne işimize yarar? Bunu irdelemeden önce kendi içinde sıralı iki listeyi nasıl birleştirip sıralı bir liste haline getiririz, onu görelim.

Listeleri Birleştirmek. İki sıralı listeden bir sıralı liste yapacağız. Bu iş gerçekten çok kolay. Hesaplama süresi de çok hızlı. Tek yapmamız gereken, her iki listenin de ilk terimlerine bakıp hangisinin birleşik listenin ilk terimi olacağına karar vermek ve bu işlemi özyinelemeli olarak listelerin geri kalan terimlerine uygulamak:

```
>(define birleştir
        (lambda (liste1 liste2) (cond
                ((< (car liste1) (car liste2)
```

```
(cons (car liste1) (birleştir (cdr liste1) liste2)))
                    (else
(cons (car liste2) (birleştir (liste1 (cdr liste2)))))))))
```

Bu "birleştir" adını vediğimiz program ne yapıyor? Eğer liste1 adı verilen birinci listenin ilk terimi liste2 adı verilen ikinci listenin ilk teriminden daha küçükse (< (car liste1) (car liste2) komutuyla sorulmuş soru), birinci listenin ilk terimini en başa koyuyor; sonra, o terimi birinci listeden çıkarıp aynı işlemi, elde ettiği yeni listelere tekrar uygulayıp ilk terimin ardına koyuyor. Örneğin listelerimiz (3 8 1) ve (6 1 2 3 6) ise, birinci listenin ilk terimi olan 3'ü en başa koyup aynı işlemi (8 1) ve (6 1 2 3 6) listelerine uygulayıp 3'ün ardına diziyor. Bir sonraki adımda ilk terimlerden küçüğü olan 6 seçilecek ve program (8 1) ve (1 2 3 6) listelerine uygulanacak.

Fikir güzel ve basit ama yine ana koşulumuz eksik. Eğer birleştirmek istediğimiz listelerden biri boşsa, birleşmiş liste diğer listenin ta kendisidir.

```
> (define birleştir
    (lambda (liste1 liste2)
        (cond ((null? liste1) liste2)
              ((null? liste2) liste1)
              ((< (car liste1) (car liste2)) (cons (car
liste1) (birleştir (cdr liste1) liste2)))
              (else (cons (car liste2) (birleştir liste1
(cdr liste2)))))))
```

Sınayalım:

```
>(birleştir '( 2 4 6 9) '(1 3 5 7))
(1 2 3 4 5 6 7 9)
```

Gayet iyi gidiyoruz. Şimdi toparlayalım ve bu işlemler sıralama konusunu nasıl kolaylaştıracak görelim.

Listeyi Sıralamak. Küçük listeleri sıralamak çok daha kolay olduğundan, önce listemizi küçük parçalara böleceğiz, küçük parçaları sıralayıp bu sıralanmış küçük parçaları birleştirme işlemine sokacağız. Küçük listeleri nasıl sıralayacağızı gördük. Oluşturduğumuz birleştirme algoritmasını özyinelemeli olarak çalıştıracağız, hepsi bu:

```
> (define sırala
(lambda (x)
(birleştir (sırala (çiftler x)) (sırala (tekler x)))))
```

Basit bir fikir ve kısa bir program. Ancak her zamanki gibi dikkatli olmamız gereken bir nokta var: Programımız boş veya tek terimli listeleri ikiye bölmeye çalışacak ve bu nedenle hiç bitmeyecek. Bir terimli veya boş listeler zaten sıralıdır. O halde bu ana koşulumuzu programa ekleyelim:

```
> (define sırala
(lambda (liste)
    (cond
    ((null? liste) liste)
    ((null? (cdr liste)) liste)
    (else (birleştir (sırala (tekler liste)) (sırala
(çiftler liste)))))))
```

Bakalım programımız doğru çalışıyor mu?

```
>(sırala '(9 4 2 1 3 8 6))
(1 2 3 4 6 8 9)
```

İşte programımızın son hali:

```
(define tekler
 (lambda (liste)
  (cond ((null? liste) '())
   (else (cons (car liste) (çiftler (cdr liste)))))))
(define çiftler
 (lambda (liste)
  (cond ((null? liste) '())
   (else (tekler (cdr liste))))))
(define birleştir
 (lambda (liste1 liste2)
  (cond
   ((null? liste1) liste2)
   ((null? liste2) liste1)
   ((< (car liste1) (car liste2)) (cons (car liste1)
    (birleştir (cdr liste1) liste2)))
   (else (cons (car liste2)
    (birleştir liste1 (cdr liste2)))))))
(define sırala
 (lambda (liste)
  (cond ((null? liste) liste)
   ((null? (cdr liste)) liste)
   (else (birleştir (sırala (tekler liste))
    (sırala (çiftler liste)))))))
```

O(n) zamanda iki listeyi birleştiren bir programımız var artık. n uzunluğundaki bir listeyi ikiye bölmek için kullandığımız yöntem O(n) zaman alıyor. Listemizi ikiye böle böle tek elemanlı listelere varmak için de en fazla O(log n) adıma ihtiyacımız var. Dolayısıyla sıralama işlemimizi en fazla O(n log n) zamanda biterebiliriz

İşin güzel tarafı tamamen özyinelemeli bir fonksiyonla liste sıralama işlemi yapılabildiğini görmek. Bu da ilginç bir şekilde sonsuz listelerin sıralanmasına yol açıyor. ♥