**Ağaçlar 6**

Günlük hayatımızda bir ağaca baktığımızda, kökleri genellikle toprakta ve yaprakları havada olur. Bir ağacın dalları, köklerden daha düzenli bir şekilde yayılır. "Tree" terimi, Bilgisayar Bilimi'nde verilerin nasıl düzenlenebileceğini açıklamak için kullanılır. Ağaçlar, Bölüm 4'te ele alınan "linked list" yapısına benzerlik gösterir. Bir "Tree" içinde, diğer düğümlere bağlantıları olan düğümler bulunur. Bir "linked list" içinde her düğüm yalnızca bir bağlantıya sahipken, "Tree" yapısında her düğümün iki veya daha fazla bağlantısı olabilir. "Tree" sıralı bir veri yapısı değildir. Gerçek hayattaki ağaçlardan farklı olarak, "Tree" veri yapılarında "root" en üstte, "leaves" ise en altta yer alır. Bilgisayar Bilimi'nde "Tree" birçok amaç için kullanılır. Bazen, fonksiyon çağrılarının yapısını göstermek için kullanılır. Örneğin, Şekil 6.1 Fibonacci fonksiyonunun "call tree" yapısını göstermektedir.

Şekil 6.1, "fib(5)" fonksiyonunu hesaplamak için kullanılan çağrı ağacını göstermektedir. Gerçek ağaçların aksine, "Tree" yapısının kökü üstte ve yaprakları alttadır. Bu yapıda düğümler arasında ilişkiler bulunmaktadır. "fib(5)" çağrısı, bir sol "sub-tree" ve bir sağ "sub-tree" içerir. "fib(4)" düğümü, "fib(5)" düğümünün bir "child node"udur. "fib(4)" düğümü, sağında yer alan "fib(3)" düğümüyle "sibling node"dur. "Leaf node", çocuğu olmayan düğümdür. Şekil 6.1'deki "leaf nodes", Fibonacci fonksiyonunun "base cases"ini temsil etmektedir.

Bu bölümde "Tree" yapısını keşfedecek ve bir programda ne zaman ve nasıl kullanılacağını öğreneceğiz. Her programın bir "Tree" veri yapısına ihtiyacı olmasa da, birçok farklı programda kullanılır. "Tree" yapılarının doğru şekilde kullanılması, belirli türdeki programları büyük ölçüde basitleştirebilir.

**6.1 Bölüm Hedefleri**

Bu bölüm, "Tree" yapılarını ve bu yapıların kullanıldığı bazı algoritmaları tanıtmaktadır. Bölümün sonunda, aşağıdaki sorulara cevap verebilmelisiniz:

* Ağaçlar nasıl inşa edilir?
* Bir ağacı nasıl gezebiliriz?

© Springer International Publishing Switzerland 2015 163

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,

Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9\_6

ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**Fig.6.1** The Call Tree for Computing fib(5)

* Expressions (İfadeler) ve ağaçlar nasıl ilişkilidir?
* İkili arama ağacı (binary search tree) nedir?
* Bir ikili arama ağacı ne zaman kullanışlıdır?
* Derinlik öncelikli arama (depth first search) nedir ve ağaçlar ile arama problemleriyle nasıl ilişkilidir?
* İkili ağaçlarda yapabileceğimiz üç tür ağaç gezintisi (tree traversal) nedir?
* Bir dil bilgisi (grammar) nedir ve bir dil bilgisi ile ne yapabiliriz?

Bilgisayar Bilimi'nde ağaçları ve kullanım alanlarını keşfetmek için okumaya devam edin.

**6.2 Soyut Sözdizimi Ağaçları ve İfadeler**

Ağaçların Bilgisayar Bilimi'nde birçok uygulama alanı vardır. Birçok farklı algoritma türünde kullanılırlar. Örneğin, yazdığınız her Python programı, Python yorumlayıcısı tarafından çalıştırılmadan önce, en azından bir süreliğine bir ağaca dönüştürülür. İçsel olarak, bir Python programı çalıştırılmadan önce, Soyut Sözdizimi Ağacı (Abstract Syntax Tree - AST) adı verilen ağaç benzeri bir yapıya dönüştürülür. Biz de kendi soyut sözdizimi ağaçlarımızı ifadeler için oluşturabiliriz, böylece bir ağacın nasıl değerlendirilebileceğini ve neden bir ağacı değerlendirmek isteyeceğimizi görebiliriz.

4.Bölüm’de bağlı listeler, bir listeyi organize etmenin bir yolu olarak sunulmuştu. Ağaçlar, benzer bir yapıyı kullanarak saklanabilir. Bir ağacın bir düğümü iki çocuğa sahipse, bu düğüm, sıralı bir listedeki bir sonraki düğüme bağlantı yapan bağlı listelerden farklı olarak, iki bağlantıya sahip olacaktır.

(5+4) \* 6 + 3 ifadesini düşünelim. Bu ifade için Şekil 6.2’de gösterildiği gibi bir soyut sözdizimi ağacı oluşturabiliriz. Çünkü + işlemi, bu fonksiyon değerlendirilirken son gerçekleştirilen işlem olduğundan, + düğümü ağacın kökü olacaktır. Bu düğümün iki alt ağacı vardır: +'ın solundaki ifade ve +'ın sağındaki 3.

6.2 Soyut Sözdizimi Ağaçları ve İfadeler

kırpıntı çizim, tasarım içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**Fig.6.2** The AST for (5+4) \* 6+3

Benzer şekilde, diğer operatörler ve operandlar için düğümler oluşturulabilir ve bu da Şekil 6.2'de gösterilen ağacı verir. Bunu bilgisayarda temsil etmek için, her düğüm türü için bir sınıf tanımlayabiliriz. **TimesNode**, **PlusNode** ve **NumNode** sınıflarını tanımlayacağız. Böylece soyut sözdizimi ağacını değerlendirebilmek için, ağacın her düğümünde tanımlı bir **eval** metodu olacak. 6.2.1. Bölümdeki kod, bu sınıfları, **eval** metodlarını ve Şekil 6.2'deki örnek ağacı oluşturan ana fonksiyonu tanımlar. **6.2.1 Constructing ASTs**

1. **class** **TimesNode**:
2. **def** \_\_init\_\_(self, left, right):
3. self.left = left
4. self.right = right

5

1. **def** eval(self):
2. **return** self.left.eval() \* self.right.eval()

8

1. **class** **PlusNode**:
2. **def** \_\_init\_\_(self, left, right):
3. self.left = left
4. self.right = right

13

1. **def** eval(self):
2. **return** self.left.eval() + self.right.eval()

16

1. **class** **NumNode**:
2. **def** \_\_init\_\_(self, num):
3. self.num = num

20

1. **def** eval(self):
2. **return** self.num

23

1. **def** main():
2. x = NumNode(5)
3. y = NumNode(4)
4. p = PlusNode(x,y)
5. t = TimesNode(p, NumNode(6))
6. root = PlusNode(t, NumNode(3))

30

31 **print**(root.eval())

32

1. **if** \_\_name\_\_ == "\_\_main\_\_":
2. main()

6.2.1. Bölümde, ağaç kökten (yani yapraklardan) yukarı doğru inşa edilir. Yukarıdaki kod, her düğüm için bir **eval** fonksiyonu içerir. Kök düğüm üzerinde **eval** çağrısı yapmak, ağacın her düğümünde **eval** metodunu özyinelemeli olarak çağırır ve bu, sonucu olan 57'yi ekrana yazdırır.

Bir **AST** oluşturulduktan sonra, böyle bir ağacın değerlendirilmesi, ağacın özyinelemeli olarak gezilmesiyle gerçekleştirilir. **eval** metodları, bu örnekteki özyinelemeli fonksiyonu oluşturur. **eval** metodlarının birbirini özyinelemeli çağırdığını söyleriz çünkü tüm **eval** metodları bir arada özyinelemeli fonksiyonu oluşturur.

**6.3 Prefix ve Postfix İfadeleri**

İfadeler, normalde yazdığımız şekliyle, infiks biçiminde kabul edilir. Bir infiks ifade, ikili operatörlerin operandlarının arasında yazıldığı bir ifadedir. Ancak ifadeler başka şekillerde de yazılabilir. İfadeler için bir diğer biçim postfix'tir. Bir postfix ifadede, ikili operatörler operandlarının ardından yazılır. (5 + 4) \* 6 + 3 infiks ifadesi, postfix biçiminde 5 4 + 6 \* 3 + olarak yazılabilir.

Postfix ifadeleri, bir yığın (stack) ile değerlendirme yapmak için oldukça uygundur. Bir operand ile karşılaştığımızda, değeri yığına iteriz. Bir operatör ile karşılaştığımızda, operandları yığından çıkarır, işlemi yapar ve sonucu tekrar yığına iteriz. Bu şekilde ifadeleri değerlendirmek, biraz pratikle insanlar için oldukça kolaydır. Hewlett-Packard, bu postfix değerlendirme yöntemini kullanan birçok hesap makinesi tasarlamıştır. Aslında, bilgisayarın ilk yıllarında, Hewlett-Packard birçok hesap makinesi üretmiştir.

İfadeleri değerlendirmek için aynı yöntemi kullanan bir dizi bilgisayar vardı. HP 2000 bunlardan biriydi. Daha yakın zamanlarda ise birçok sanal makine, stack makinesi olarak uygulanmıştır, bunlara örnek olarak Java Virtual Machine (JVM) ve Python virtual machine verilebilir.

Bir ağaç gezintisinin başka bir örneği olarak, bir ifadenin **string** temsilini döndüren bir metod yazmayı düşünün. **String** ağacın bir **soyut sözdizimi ağacı** (AST) gezilerek oluşturulur. Bir ifadenin infiks biçimini temsil eden bir **string** elde etmek için **AST'nin inorder gezintisi** yapılır. Postfix ifadesi elde etmek için ise ağacın **postfix gezintisi** yapılır. 6.3.1. Bölümdeki **inorder metodları**, bir **AST'nin inorder gezintisini** gerçekleştirir.

**6.3.1 AST Ağaç Dolaşımı**

1. **class** **TimesNode**:
2. **def** \_\_init\_\_(self, left, right):
3. self.left = left
4. self.right = right

6.3 Prefix and Postfix Expressions

5

1. **def** eval(self):
2. **return** self.left.eval() \* self.right.eval()

8

1. **def** inorder(self):
2. **return** "(" + self.left.inorder() + " \* " + self.right.inorder() + ")"

11

1. **class** **PlusNode**:
2. **def** \_\_init\_\_(self, left, right):
3. self.left = left
4. self.right = right

16

1. **def** eval(self):
2. **return** self.left.eval() + self.right.eval()

19

20

1. **def** inorder(self):
2. **return** "(" + self.left.inorder() + " + " + self.right.inorder() + ")"

23

1. **class** **NumNode**:
2. **def** \_\_init\_\_(self, num):
3. self.num = num

27

1. **def** eval(self):
2. **return** self.num

30

1. **def** inorder(self):
2. **return** str(self.num)

6.3.1. Bölümdeki **inorder metodları**, her **ikili operatörün** iki operandı arasına eklenmesi nedeniyle bir **inorder gezintisi** sağlar. Ağacın **postorder gezintisini** yapmak için, her **ikili operatörü** string'e ekleyecek bir **postorder metod**u yazmamız gerekir; bu, iki operandın **postorder gezintisi** yapıldıktan sonra eklenir. Postorder gezintisinin yazılma şekli nedeniyle, **postfix ifadelerinde** parantezlere hiç ihtiyaç duyulmadığını unutmayın.

Bir diğer gezinti türü ise **preorder gezintisi**dir. **Preorder gezintisinde**, her **ikili operatör** string'e, iki operandından önce eklenir. **(5 + 4) \* 6 + 3** infiks ifadesine verilecek **prefix** karşılığı **+ \* + 5 4 6 3** olur. Yine, **prefix ifadelerinin** yazılma şekli nedeniyle, **prefix ifadelerinde** de parantezlere hiç ihtiyaç duyulmaz.

**6.4 Prefix İfadelerinin Çözülmesi (Parsing)**

Soyut sözdizimi ağaçları (AST) neredeyse hiç elle oluşturulmaz. Genellikle bir yorumlayıcı (interpreter) veya derleyici (compiler) tarafından otomatik olarak inşa edilirler. Bir Python programı çalıştırıldığında, Python yorumlayıcısı programı tarar ve programın soyut sözdizimi ağacını (AST) oluşturur. Python yorumlayıcısının bu kısmı parser (çözümleyici) olarak adlandırılır. Bir parser, bir dosyayı okuyan ve ifadenin (yani bir kaynak programın) soyut sözdizimi ağacını (AST) otomatik olarak inşa eden bir program veya programın bir parçasıdır ve program veya ifade düzgün oluşturulmamışsa bir sözdizimi hatasını (syntax error) raporlar. Bunun nasıl yapıldığına dair tam detaylar, bu metnin kapsamı dışındadır. Ancak bazı basit ifadeler için, örneğin prefix ifadeleri gibi, bir parser'ı kendimiz oluşturmak oldukça kolaydır.

Ortaokulda, bir cümlenin düzgün bir şekilde oluşturulup oluşturulmadığını kontrol ederken \*\*İngilizce dil bilgisi (grammar)\*\*ni kullanmamız gerektiğini öğrenmiştik. Dil bilgisi (grammar), bir dilde bir cümlenin nasıl bir araya getirileceğini belirleyen bir kurallar setidir. Bilgisayar Bilimi'nde birçok farklı dil vardır ve her dilin kendi dil bilgisi (grammar) vardır. Prefix ifadeleri, bir dil oluşturur. Bu ifadeleri prefix ifadelerinin dili olarak adlandırırız ve kendi dil bilgilerine (grammar) sahip olup buna bağlamdan bağımsız dil bilgisi (context-free grammar) denir. Prefix ifadeleri için bağlamdan bağımsız dil bilgisi, 6.4.1. Bölümde verilmiştir.

**6.4.1 Prefix İfadeleri Dil Bilgisi**

*G* = (*N* ,*T* ,*P*,*E*) where

*N* = {*E*}

*T* = {*identifier*,*number*,+,∗}

*P* is defined by the set of productions

*E* → + *E E* | ∗ *E E* | *number*

Bir **dil bilgisi (grammar)**, G, üç kümeden oluşur: **non-terminal semboller (N)** kümesi, **terminal semboller** veya **token'lar (T)** kümesi ve **üretimler (productions)** kümesi (P). Non-terminal sembollerden biri, dil bilgisinin **başlangıç sembolü (start symbol)** olarak belirlenir. Bu dil bilgisinde, özel sembol **E**, başlangıç sembolü ve dil bilgisinin tek **non-terminal** sembolüdür. **E** sembolü, herhangi bir **prefix ifadesini** temsil eder. Bu dil bilgisinde, **prefix ifadelerinin** nasıl oluşturulacağına dair üç üretim vardır. Üretimler, herhangi bir **prefix ifadesinin** (→ sembolü "şundan oluşur" olarak okunabilir) bir **artı işareti (plus sign)** ardından iki **prefix ifadesi**, bir **çarpma sembolü (multiplication symbol)** ardından iki **prefix ifadesi** veya sadece bir **sayı** olabileceğini belirtir. Bu dil bilgisi **özyinelemeli (recursive)** olduğundan, dil bilgisinde her **E** gördüğünüzde, bu başka bir **prefix ifadesiyle** değiştirilebilir. Bu dil bilgisi, **token'lar kuyruğu** verildiğinde bir **prefix ifadesinin soyut sözdizimi ağacını (AST)** inşa edecek bir **fonksiyona** dönüştürülmesi oldukça kolaydır. **Token'ları okuyan ve bir soyut sözdizimi ağacı döndüren** bir fonksiyon, **parser (çözümleyici)** olarak adlandırılır. Dil bilgisi **özyinelemeli (recursive)** olduğu için, **parsing fonksiyonu** da özyinelemeli olacaktır. İlk olarak bir **taban durumu (base case)** bulunur, ardından **özyinelemeli durumlar (recursive cases)** gelir. 6.4.2. Bölümde bu fonksiyon sağlanmaktadır.