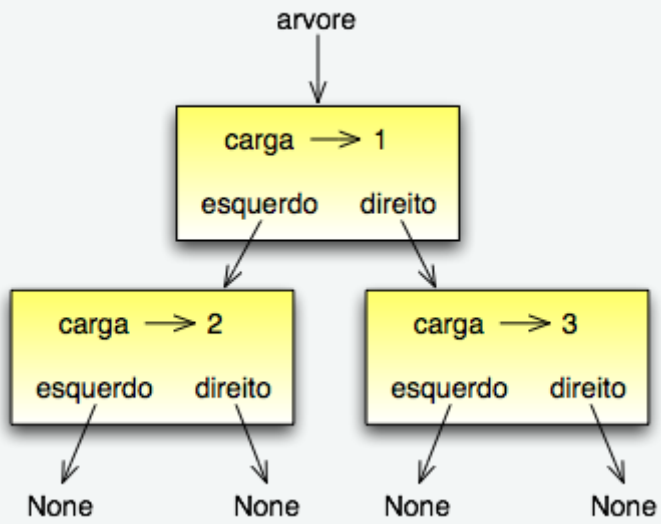
ÁRVORE BINÁRIA

Como listas ligadas, árvores são constituídas de células. Uma espécie comum de árvores é a **árvore binária**, em que cada célula contém referências a duas outras células (possivelmente nulas). Tais referências são chamadas de subárvore esquerda e direita. Como as células de listas ligadas, as células de árvores também contém uma carga. Um diagrama de estados para uma árvore pode aparecer assim:

Figura 1



Para evitar a sobrecarga da figura, nós frequentemente omitimos os `Nones`.

O topo da árvore (a célula à qual o apontador `tree` se refere) é chamada de **raiz**. Seguindo a metáfora das árvores, as outras células são chamadas de galhos e as células nas pontas contendo as referências vazia são chamadas de **folhas**. Pode parecer estranho que desenhamos a figura com a raiz em cima e as folhas em baixo, mas isto nem será a coisa mais estranha.

Para piorar as coisas, cientistas da computação misturam outra metáfora além da metáfora biológica - a árvore genealógica. Uma célula superior pode ser chamada de **pai** e as células a que ela se refere são chamadas de seus **filhos**. Células com o mesmo pai são chamadas de **irmãos**.

Finalmente, existe também o vocabulário geométrico para falar de árvores. Já mencionamos esquerda e direita, mas existem também as direções “para cima” (na direção da raiz) e “para baixo” (na direção dos filhos/folhas). Ainda nesta terminologia, todas as células situadas à mesma distância da raiz constituem um **nível** da árvore.

Provavelmente não precisamos de três metáforas para falar de árvores, mas aí elas estão.

Como listas ligadas, árvores são estruturas de dados recursivas já que elas são definidas recursivamente:

Uma árvore é

- a árvore vazia, representada por `None`, ou
- uma célula que contém uma referência a um objeto (a carga da célula) e duas referências a árvores.

Construindo árvores

O processo de montar uma árvore é similar ao processo de montar uma lista ligada. cada invocação do construtor cria uma célula.

```
class Tree :
  def __init__(self, cargo, left=None, right=None) :
    self.cargo = cargo
    self.left  = left
    self.right = right

  def __str__(self) :
    return str(self.cargo)
```

A `carga` pode ser de qualquer tipo, mas os parâmetros `left` e `right` devem ser células. `left` e `right` são opcionais; o valor default é `None`.

Para imprimir uma célula, imprimimos apenas a sua carga.

Uma forma de construir uma árvore é de baixo para cima. Aloque os filhos primeiro:

```
left = Tree(2)
right = Tree(3)
```

Em seguida crie a célula pai e ligue ela a seus filhos:

```
tree = Tree(1, left, right);
```

Podemos escrever este código mais concisamente encaixando as invocações do construtor:

```
>>> tree = Tree(1, Tree(2), Tree(3))
```

De qualquer forma, o resultado é a árvore que apareceu no início do capítulo.

Percorrendo árvores

Cada vez que Você vê uma nova estrutura de dados, sua primeira pergunta deveria ser “Como eu percorro esta estrutura?” A forma mais natural de percorrer uma árvore é fazer o percurso recursivamente. Por exemplo, se a árvore contém inteiros na carga, a função abaixo retorna a soma das cargas:

```
def total(tree) :
  if tree == None : return 0
  return total(tree.left) + total(tree.right) + tree.cargo
```

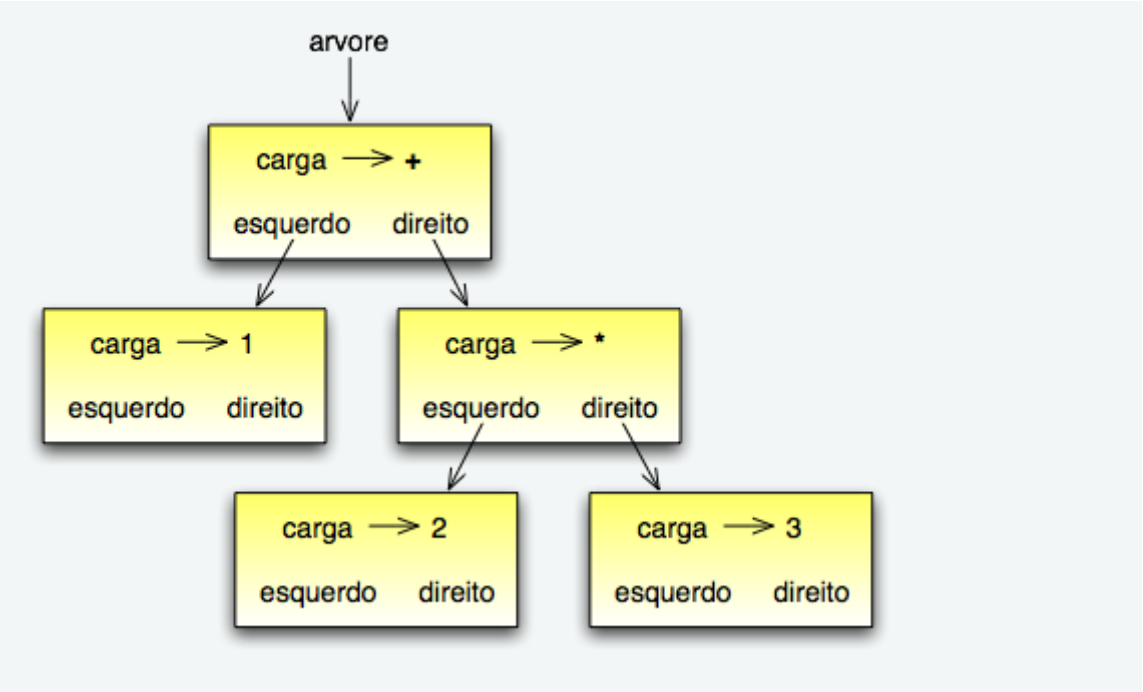
O caso base é a árvore vazia, que não contém nenhuma carga, logo a soma das cargas é 0. O passo recursivo faz duas chamadas recursivas para achar a soma das cargas das subárvores dos filhos. Ao finalizar a chamada recursiva, adicionamos a carga do pai e devolvemos o valor total.

Árvores de expressões

Uma árvore é uma forma natural para representar a estrutura de uma expressão. Ao contrário de outras notações, a árvore pode representar a computação de forma não ambígua. Por exemplo, a expressão infixa `1 + 2 * 3` é ambígua, a menos que saibamos que a multiplicação é feita antes da adição.

A árvore de expressão seguinte representa a mesma computação:

Figura 2



As células de uma árvore de expressão podem ser operandos como `1` e `2` ou operações como `+` e `*`. As células contendo operandos são folhas; aquelas contendo operações devem ter referências aos seus operandos. (Todos os nossos operandos são **binários**, significando que eles tem exatamente dois operandos.)

Podemos construir árvores assim:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
```

Examinando a figura, não há dúvida quanto à ordem das operações; a multiplicação é feita primeiro para calcular o segundo operando da adição.

Árvores de expressão tem muitos usos. O exemplo neste capítulo usa árvores para traduzir expressões para as notações pósfixa, prefixa e infixa. Árvores similares são usadas em compiladores para analisar sintaticamente, otimizar e traduzir programas.

Percurso de árvores

Podemos percorrer uma árvore de expressão e imprimir o seu conteúdo como segue:

```
def printTree(tree) :
  if tree == None : return
  print (tree.cargo, end="")
  printTree(tree.left)
```

```
  printTree(tree.right)
```

Em outras palavras, para imprimir uma árvore, imprima primeiro o conteúdo da raiz, em seguida imprima toda a subárvore esquerda e finalmente imprima toda a subárvore direita. Esta forma de percorrer uma árvore é chamada de **préordem**, porque o conteúdo da raiz aparece *antes* dos conteúdos dos filhos. Para o exemplo anterior, a saída é:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
>>> printTree(tree)
+ 1 * 2 3
```

Esta notação é diferente tanto da notação pósfixa quanto da infixa; é uma notação chamada de **prefixa**, em que os operadores aparecem antes dos seus operandos.

Você pode suspeitar que se Você percorre a árvore numa ordem diferente, Você produzirá expressões numa notação diferente. Por exemplo, se Você imprime subárvores primeiro e depois a raiz, Você terá:

```
def printTreePostorder(tree) :
  if tree == None : return
  printTreePostorder(tree.left)
  printTreePostorder(tree.right)
  print (tree.cargo, end="")
```

O resultado, `1 2 3 * +`, está na notação pósfixa! Esta ordem de percurso é chamada de pósordem.

Finalmente, para percorrer uma árvore em **inordem**, Você imprime a subárvore esquerda, depois a raiz e depois a subárvore direita:

```
def printTreeInorder(tree) :
  if tree == None : return
  printTreeInorder(tree.left)
  print (tree.cargo, end="")
  printTreeInorder(tree.right)
```

O resultado é `1 + 2 * 3`, que é a expressão na notação infixa.

Para sermos justos, devemos lembrar que acabamos de omitir uma complicação importante. algumas vezes quando escrevemos expressões na notação infixa devemos usar parêntesis para prescrever a ordem das operações. Ou seja, um percurso em inordem não é suficiente para gerar a expressão infixa.

Ainda assim, com alguns aperfeiçoamentos, a árvore de expressão e os três modos recursivos de percurso resultam em algoritmos para transformar expressões de uma notação para outra.

Como um exercício, modifique `printTreeInorder` de modo que ele coloque parêntesis em volta de cada operador e par de operandos. A saída é correta e não ambígua? Os parêntesis são sempre necessários?

Se percorrermos uma árvore em inordem e acompanharmos em qual nível na árvore estamos, podemos gerar uma representação gráfica da árvore:

```
def printTreeIndented(tree, level=0) :
  if tree == None : return
  printTreeIndented(tree.right, level+1)
  print ('  '*level + str(tree.cargo))
  printTreeIndented(tree.left, level+1)
```

O parâmetro `level` registra aonde estamos na árvore. Por 'default', o nível inicialmente é zero. A cada chamada recursiva repassamos `level+1` porque o nível do filho é sempre um a mais do que o nível do pai. Cada item é indentado dois espaços por nível. Para o nosso exemplo obtemos:

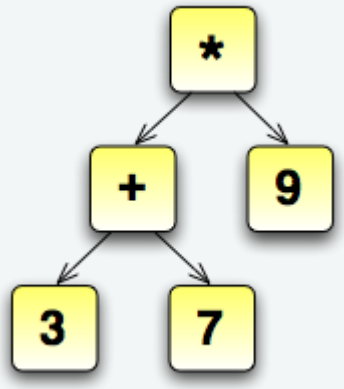
```
>>> printTreeIndented(tree)
    3
  *
    2
+
  1
```

Se Você deitar a saída acima Você enxerga uma versão simplificada da figura original.

Construindo uma árvore de expressão

Nesta seção analisamos expressões infixas e construímos as árvores de expressã correspondentes. Por exemplo, para a expressão `(3+7)*9` resultará a seguinte árvore:

Figura 3



Note que simplificamos o diagrama omitindo os nomes dos campos.

O analisador que escreveremos aceitará expressões que incluam números, parêntesis e as operações `+` e `*`. Vamos supor que a cadeia de entrada já foi tokenizada numa lista do Python. A lista de tokens para a expressão `(3+7)*9` é:

```
['(', 3, '+', 7, ')', '*', 9, 'end']
```

O token final `end` é prático para prevenir que o analisador tente buscar mais dados após o término da lista.

A título de um exercício, escreva uma função que recebe uma expressão na forma de uma cadeia e devolve a lista de tokens.

A primeira função que escreveremos é `getToken` que recebe como parâmetros uma lista de tokens e um token esperado. Ela compara o token esperado com o o primeiro token da lista: se eles batem a função remove o token da lista e devolve um valor verdadeiro, caso contrário a função devolve um valor falso:

```
def getToken(tokenList, expected) :
  if tokenList[0] == expected :
    tokenList[0:1] = []   # remove the token
    return 1
  else :
    return 0
```

Já que `tokenList` refere a um objeto mutável, as alterações feitas aqui são visíveis para qualquer outra variável que se refira ao mesmo objeto.

A próxima função, `getNumber`, trata de operandos. Se o primeiro token na `tokenList` for um número então `getNumber` o remove da lista e devolve uma célula folha contendo o número; caso contrário ele devolve `None`.

```
def getNumber(tokenList) :
  x = tokenList[0]
  if type(x) != type(0) : return None
  del tokenList[0]
  return Tree(x, None, None)
```

Antes de continuar, convém testar `getNumber` isoladamente. Atribuímos uma lista de números a `tokenList`, extraímos o primeiro, imprimimos o resultado e imprimimos o que resta na lista de tokens:

```
>>> tokenList = [9, 11, 'end']
>>> x = getNumber(tokenList)
>>> printTreePostorder(x)
9
>>> print (tokenList)
[11, 'end']
```

Em seguida precisaremos da função `getProduct`, que constrói uma árvore de expressão para produtos. Os dois operandos de um produto simples são números, como em `3 * 7`.

Segue uma versão de `getProduct` que trata de produtos simples.

```
def getProduct(tokenList) :
  a = getNumber(tokenList)
  if getToken(tokenList, '*') :
    b = getNumber(tokenList)
    return Tree('*', a, b)
  else :
    return a
```

Supondo que a chamada de `getNumber` seja bem sucedida e devolva uma árvore de uma só célula atribuímos o primeiro operando a *à*`. Se o próximo caractere for `*`, vamos buscar o segundo número e construir a árvore com `a`, `b` e o operador.

Se o caractere seguinte for qualquer outra coisa, então simplesmente devolvemos uma célula folha com `a`. Seguem dois exemplos:

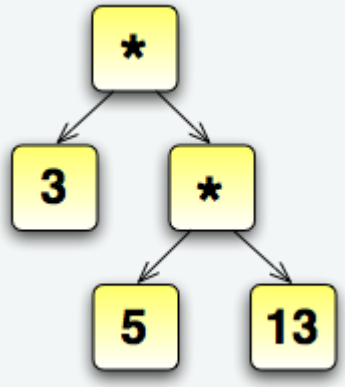
```
>>> tokenList = [9, '*', 11, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
9 11 *


>>> tokenList = [9, '+', 11, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
9
```

O segundo exemplo sugere que nós consideramos um operando unitário como uma espécie de produto. Esta definição de “produto” talvez não seja intuitiva, mas ela será útil.

Agora tratamos produtos compostos, como `3 * 5 * 13`. Encaramos esta expressão como um produto de produtos, mais precisamente como `3 * (5 * 13)`. A árvore resultante é:

Figura 4



Com uma pequena alteração em `getProduct`, podemos acomodar produtos arbitrariamente longos:

```
def getProduct(tokenList) :
  a = getNumber(tokenList)
  if getToken(tokenList, '*') :
    b = getProduct(tokenList)        # this line changed
    return Tree('*', a, b)
  else :
    return a
```

Em outras palavras, um produto pode ser um singleton ou uma árvore com `*` na raiz, que tem um número como filho esquerdo e um produto como filho direito. Este tipo de definição recursiva devia começar a ficar familiar.

Testemos a nova versão com um produto composto:

```
>>> tokenList = [2, '*', 3, '*', 5 , '*', 7, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
2 3 5 7 * * *
```

A seguir adicionamos o tratamento de somas. De novo, usamos uma definição de “soma” que é ligeiramente não intuitiva. Para nós, uma soma pode ser uma

árvore com `+` na raiz, que tem um produto como filho esquerdo e uma soma como filho direito. Ou, uma soma pode ser simplesmente um produto.

Se Você está disposto a brincar com esta definição, ela tem uma propriedade interessante: podemos representar qualquer expressão (sem parêntesis) como uma soma de produtos. Esta propriedade é a base do nosso algoritmo de análise sintática.

`getSum` tenta construir a árvore com um produto à esquerda e uma soma à direita. Mas, se ele não encontra uma `+`, ele simplesmente constrói um produto.

```
def getSum(tokenList) :
  a = getProduct(tokenList)
  if getToken(tokenList, '+') :
    b = getSum(tokenList)
    return Tree('+', a, b)
  else :
    return a
```

Vamos testar o algoritmo com `9 * 11 + 5 * 7`:

```
>>> tokenList = [9, '*', 11, '+', 5, '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 * 5 7 * +
```

Quase terminamos, mas ainda temos que tratar dos parêntesis. Em qualquer lugar numa expressão onde podemos ter um número, podemos também ter uma soma inteira envolvida entre parêntesis. Precisamos, apenas, modificar `getNumber` para que ela possa tratar de **subexpressões**:

```
def getNumber(tokenList) :
  if getToken(tokenList, '(') :
    x = getSum(tokenList)      # get subexpression
    getToken(tokenList, ')')    # eat the closing parenthesis
    return x
  else :
    x = tokenList[0]
    if type(x) != type(0) : return None
    tokenList[0:1] = []   # remove the token
    return Tree(x, None, None)    # return a leaf with the number
```

Testemos este código com `9 * (11 + 5) * 7`:

```
>>> tokenList = [9, '*', '(', 11, '+', 5, ')', '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 5 + 7 * *
```

O analisador tratou os parêntesis corretamente; a adição é feita antes da multiplicação.

Na versão final do programa, seria uma boa ideia dar a `getNumber` um nome mais descritivo do seu novo papel.

Manipulando erros

Ao longo do analisador sintático tínhamos suposto que as expressões (de entrada) são bem formadas. Por exemplo, quando atingimos o fim de uma subexpressão, supomos que o próximo caractere é uma facha parêntesis. Caso haja um erro e o próximo caractere seja algo diferente, devemos tratar disto.

::

```
class BadExpressionError(Exception):
    pass

def getNumber(tokenList) :
    if getToken(tokenList, '(') :
        x = getSum(tokenList) if not getToken(tokenList, ')'):
            raise BadExpressionError('missing parenthesis')
        return x
    else :
        # the rest of the function omitted
```

O comando `raise` cria uma exceção; neste caso criamos uma classe de exceção, chamada de `BadExpressionError`. Se a função que chamou `getNumber`, ou uma das outras funções no traceback, manipular a exceção, então o programa pode continuar. Caso contrário Python vai imprimir uma mensagem de erro e terminará o processamento em seguida.

A título de exercício, encontre outros locais nas funções criadas onde erros possam ocorrer e adiciona comandos ``raise`` apropriados. Teste seu código com expressões mal formadas.