Stacks and Trees

Optional assignment 10
stacks, postfix, recursive traversal

Optional assignment

Simple postfix calculator ... with a twist

```
$ python3 symcalc.py -
3 5 + 7 * 4 5 *
Stack: ((3+5)*7); (4*5);
#
Stack: ((3+5)*7); 20;
+
Stack: (((3+5)*7)+20);
#
Stack: 76;
quit
Thank you for your patience. Sorry for any bugs.
```

Postfix

Our customary notation is **infix**: 

$ (3 + 5) * 7 \text{ (operators + and * between operands)}$

**Postfix** places operators after operands:

$3 5 + 7 *$

**Prefix** places operators before operands

$* + 3 5 7$

Postfix and prefix don’t require parentheses or operator precedence. Evaluate postfix on a stack.

Stack

A stack is a sequence that supports insertion and deletion at one end

**push(x)** adds to top

**pop()** takes element from top

Python lists can be stacks:

```
stack.append(x) is push(x)
stack.pop() is pop()
```
A stack is a sequence that supports insertion and deletion at one end.

```python
s = []
s.append(5.3)
```

A stack is a sequence that supports insertion and deletion at one end.

```python
top
s = []
s.append(5.3)
s.append(3.7)
```

A stack is a sequence that supports insertion and deletion at one end.

```python
top
x = s.pop()
s = []
s.append(5.3)
s.append(3.7)
s.append(4.0)
```

A stack is a sequence that supports insertion and deletion at one end.

```python
top
x:
```
Calculating with a Stack

to calculate
5.3 * ( 3.7 + 4.0)

rearrange as postfix:
5.3 3.7 4.0 + *

Stack for Calculation

Expression represented as tree

From stack to tree ...

stack = []
stack.append(5.3)
stack.append(3.7)
stack.append(4.0)
From stack to tree ...

```
5.3  3.7  4.0  +  ^
right = stack.pop();
left = stack.pop();
stack.append([ '+', left, right ]);  
```

Trees and recursion: pretty-printing

The “expstr()” function will be recursive:
```
(" + expstr(left) + op + expstr(right) + ")
```

Two cases:
- A “leaf” (number or variable) is just printed
- An “internal node” (operation with two sub-expressions) is constructed recursively as above

Add pretty printing

*Starter code builds trees on the stack, and prints them as lists (s-expressions):*

```
$ python3 symcalc.py -
3 5 + 7 *
Stack: ["*", ["+", 3, 5], 7];
```

*You fix expstr(e) to print in infix:*

```
$ python3 symcalc.py -
3 5 + 7 *
Stack: ((3+5)*7);
```

Add evaluation

*Starter code as given; # operation does nothing:*

```
3 5 + 7 *
Stack: ["*", ["+", 3, 5], 7];
#
Stack: ["*", ["+", 3, 5], 7];
```

*You make # into “evaluate top item” operation:*

```
3 5 + 7 *
Stack: ((3+5)*7);
#
Stack: 56;
```
Recursive evaluation

The evaluate(e) function is also recursive:
   If it’s a leaf, just return the value
   If it has left and right operands, evaluatethe left operand evaluatethe right operand if left and right are both numbers apply the operation to get a number otherwise construct s-expression [op, left, right]

Base case and progress case

As always, we have base case and progress case for recursion, but they are now in different places

Base case is in evaluation of leaves (and we know we will eventually reach them)

Progress (recursive) case in in evaluation of internal nodes

Trees are sometimes called “recursive data structures”

Recursive definition of an expression:

An expression can be a constant like “5”.
An expression can be an operation applied to a left operand and a right operand.

Recursive definition of a (complete, binary) tree:

A tree can be a single leaf node.
A tree can be an interior node with two children, each of which is a tree.
recursion in evaluation

```
*  .  *
   / \
5.3  +  4.0
  /   |
3.7  4.0
```

evaluate → 3.7
evaluate → 4.0

```
5.3  +  4.0
  /   |
3.7  4.0
```

evaluate → 3.7
evaluate → 4.0

evaluate → 7.7

```
*  .  *
   / \
5.3  +  4.0
  /   |
3.7  4.0
```

evaluate → 3.7
evaluate → 4.0

```
5.3  +  4.0
  /   |
3.7  4.0
```

evaluate → 3.7
evaluate → 4.0

evaluate → 7.7

evaluate → 40.81
Expressions can contain variables

\[3 \times 5 + 7 \times x + \star\]
Stack: \((3 \times 5)*(7+x))\);
#  
Stack: \((8\times(7+x))\);

Optional Assignment

If you do it, score replaces worst project score  
(pro-rated by possible score)

You’ll see trees and stacks again, many times  
(often with different classes for internal nodes and leaf nodes)
Recursive data structures go with recursive algorithms!