Homework 1: Calculator

1 Mechanics and due dates

This assignment consists of five parts, each of which can be entered into a problem on the tutor. The first two parts are due **one week** from the start of the assignment; the other parts are due **a week after that**. See the tutor for the due dates. Your code will be tested for correctness by the tutor, but also graded for organization and style by a human. We will deduct points for repeated and/or excessively complex code. You should include some comments that explain the key points in your code. The last problem asks you to enter a text answer to a 'check yourself' question; it will be graded by a human.

Do your work in the file `6.01/hw1/hw1Work.py`.

You can discuss this problem, at a high level, with other students, but the programs and text you submit must be your own work.

2 Symbolic Calculator

We will construct a simple symbolic calculator that reads, evaluates, and prints arithmetic expressions that contain variables as well as numeric values. It is similar, in structure and operation, to the Python interpreter.

To make the parsing simple, we assume that the expressions are *fully parenthesized*: so, instead of `a = 3 + 4`, we will need to write `(a = (3 + 4))`. In other words, in any expression which involves subexpressions that are not simple elements, those subexpressions will recursively be enclosed within parentheses. Thus any complex expression contains an expression, an operator and another expression, and each of these expressions, if not a number or a variable, is itself contained within parentheses.

The following is a transcript of an interaction with our calculator, where the `%` character is the prompt. After the prompt, the user types in an expression, the calculator evaluates the expression, possibly changing the environment, and prints out both the value of the expression and the new environment.

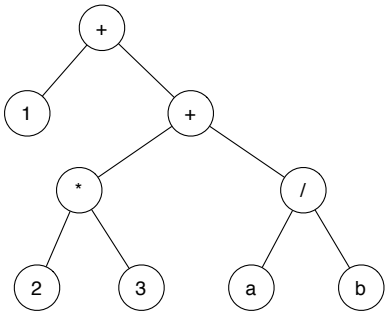
```
>>> calc()
% (a = 3)
None
   env = {'a': 3.0}
% (b = (a + 2))
None
   env = {'a': 3.0, 'b': 5.0}
% b
5.0
   env = {'a': 3.0, 'b': 5.0}
% (c = (a + (b * b)))
None
   env = {'a': 3.0, 'c': 28.0, 'b': 5.0}
```

3 Syntax Trees

The calculator operates in two phases. It

- *Parse*s the input string of characters to generate a *syntax tree*; and then
- *Evaluate*s the syntax tree to generate a value, if possible, and does any required assignments.

A syntax tree is a data structure that represents the structure of the expression. The nodes at the bottom are called *leaf* nodes and represent actual primitive components (numbers and variables) in the expression. Other nodes are called *internal* nodes. They represent an operation (such as addition or subtraction), and contain instances of *child* nodes that represent the arguments of the operation. The following tree represents the expression `(1 + ((2 * 3) + (a / b)))`. Note the use of parentheses to separate each subexpression:



We can represent syntax trees in Python using instances of the following collection of classes. These definitions are incomplete: it will be your job to fill them in.

```
class BinaryOp:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return self.opStr + '(' + \
               str(self.left) + ', ' +\
               str(self.right) + ')'
    __repr__ = __str__
class Sum(BinaryOp):
    opStr = 'Sum'
class Prod(BinaryOp):
    opStr = 'Prod'
class Quot(BinaryOp):
    opStr = 'Quot'
class Diff(BinaryOp):
    opStr = 'Diff'
class Assign(BinaryOp):
    opStr = 'Assign'
class Number:
    def __init__(self, val):
        self.value = val
    def __str__(self):
        return 'Num('+str(self.value)+')'
class Variable:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Var('+self.name+')'
```

Leaf nodes are represented by instances of `Number` and `Variable`. Internal nodes are represented by instances of `Sum, Prod, Quot`, and `Diff`. The superclass `BinaryOp` is meant to be a place

to put aspects of the binary operators that are the same for each operator, in order to minimize repetition in coding.

We could create a Python representation of `(1 + ((2 * 3) + (a / b)))` with

```
Sum(Number(1.0), Sum(Prod(Number(2.0), Number(3.0)), Quot(Variable('a'), Variable('b'))))
```

Note that we will be converting all numbers to floating point to avoid problems with division later on.

In addition to numerical expressions, the language of our calculator includes assignment 'statements', which we can represent as instances of an assignment class. They differ from the other expressions in that they are not compositional: an assignment statement has a variable on the left of the equality and an expression on the right, and it cannot itself be part of any further expressions. Because assignments share the same initialization and string methods, we have made `Assign` a subclass of `BinaryOp`, but they will require very different handling for evaluation.

4 Parsing

Parsing is the process of taking a string of characters and returning a syntax tree. We'll assume that we parse a single line, which corresponds to a single expression or assignment statement. The processing happens in two phases: tokenization and then parsing a token sequence.

4.1 Tokenization

A *tokenizer* takes a sequence of characters as input and returns a sequence of tokens, which might be words or numbers or special, meaningful characters. For instance, we might break up the string:

```
'((fred + george) / (voldemort + 666))'
```

into the list of tokens (each of which is, itself, a string):

```
['(', '(', 'fred', '+', 'george', ')', '/', '(', 'voldemort', '+', '666', ')', ')']
```

We would like our tokenizer to work the same way, even if the spaces are deleted from the input:

```
'((fred+george)/(voldemort+666))'
```

Our special, single-character tokens will be:

```
seps = ['(', ')', '+', '-', '*', '/', '=']
```

Step 1. Write a procedure `tokenize(inputString)` that takes a string of characters as input and returns a list of tokens as output. The output of `tokenize('(fred + george)')` should be `['(', 'fred', '+', 'george', ')']`. There are other test cases in the `hw1Work.py` file.

> **Wk.2.4.1** After you have debugged your code in Idle, submit it via this tutor problem. Include comments in your code.

4.2 Parsing a token sequence

The job of the parser is to take as input a list of tokens, produced by the tokenizer, and to return a syntax tree as output. Parsing Python and other programming languages can be fairly difficult,

and parsing natural language is an open research problem. But parsing our simple language is not too hard, because every expression is either:

- a number, or
- a variable name, or
- an expression of the form

  ```
  ( expression op expression )
  ```

  where op is one of `+, -, *, /, =`.

This language can be parsed using a simple *recursive descent* parser. A good way to structure your parser is as follows:

```
def parse(tokens):
    def parseExp(index):
        <your code here>
    (parsedExp, nextIndex) = parseExp(0)
    return parsedExp
```

The procedure `parseExp` is a recursive procedure that takes an integer `index` into the `tokens` list. This procedure returns a pair of values:

- the expression found starting at location `index`. This is an instance of one of the syntax tree classes: `Number`, `Variable`, `Sum`, etc.
- the index beyond where this expression ends. If the expression ends at the token with index 6, then the returned value would be 7.

In the definition of this procedure we make sure that we call it with the value `index` corresponding to the start of an expression. So, we need to handle only three cases. Let `token` be the token at location `index`. The cases are:

- If `token` represents a number, then make it into a `Number` instance and return that, paired with `index+1` . Note that the value attribute of a `Number` instance should be a Python floating point number.
- If `token` represents a variable name, then make it into a `Variable` instance and return that, paired with `index+1`. Note that the value attribute of a `Variable` instance should be a Python string.
- Otherwise, the sequence of tokens starting at `index` must be of the form:

  ```
  ( expression op expression )
  ```

  Therefore, `token` must be `'('` . We need to:
  - Parse an expression (using `parseExp`), getting a syntax tree that we'll call `leftTree` and the index for the token beyond the end of the expression.
  - The token beyond `leftTree` should be a single-character operator token; call it `op`.
  - Parse an expression (using `parseExp`) starting beyond `op`, getting a syntax tree that we'll call `rightTree`.
  - Use `op` to determine what kind of internal syntax tree instance to make: construct it using `leftTree` and `rightTree` and return it as the result of this procedure, paired with the index of the token beyond the final right paren.

We will give you two useful procedures:

- `numberTok` takes a token as an argument and returns `True` if the token represents a number and `False` otherwise.

- `variableTok` takes a token as an argument and returns `True` if the token represents a variable name and `False` otherwise.

It is also useful to know that if `token` is a string representing a legal Python number, then `float(token)` will convert it into a floating-point number.

We have implemented `__str__` methods for the syntax-tree classes. The expressions print out similarly to the Python expression that you would use to create the syntax tree:

```
>>> parse(tokenize('(1 + ((2 * 3) + (a / b)))'))
Sum(Num(1.0), Sum(Prod(Num(2.0), Num(3.0)), Quot(Var(a), Var(b))))
```

It is **very important** to remember that this is simply the string representation of what is actually an instance of the syntax tree class `Sum`.

Here are some examples:

```
>>> parse(['888'])
Num(888.0)
>>> print parse(['(', 'fred', '+', 'george', ')'])
Sum(Var(fred),  Var(george))
>>> print parse(['(', '(', 'a', '*', 'b', ')', '/', '(', 'cee', '-', 'doh', ')' ,')'])
Quot(Prod(Var(a), Var(b)), Diff(Var(cee), Var(doh)))
>>> print parse(tokenize('((a * b) / (cee - doh))'))
Quot(Prod(Var(a), Var(b)), Diff(Var(cee), Var(doh)))
```

Step 2. Implement `parse` and test it on the examples in the work file, or other strings of tokens you make up, or on the output of the tokenizer. Start by making sure it handles single numbers and variable names correctly, then work up to more complex nested expressions.

> **Wk.2.4.2** After you have debugged your code in Idle, submit it via this tutor problem. You should include only the code you wrote for `parse`. Include comments in your code.

5 Evaluation

Once we have an expression represented as a syntax tree, we can evaluate it. We will start by considering the case in which every expression can be evaluated fully to get a number; then we'll extend it to the case where expressions may remain symbolic, if the variables have not yet been defined.

For our calculator, just as for Python, expressions are evaluated with respect to an *environment*. We will represent environments using Python dictionaries (which you should read about in the Python documentation at http://docs.python.org/tutorial/datastructures.html#dictionaries), where the keys are variable names and the values are the values of those variables.

5.1 Eager evaluation

Here are the operation rules of the basic calculator, which tries to completely evaluate every expression it sees. The value of every expression is a number. The evaluation of *expr* in *env* works as follows:

- If *expr* is a `Number`, then return its value.
- If *expr* is a `Variable`, then return the value associated with the *name* of the variable in *env*.
- If *expr* is an arithmetic operation, then return the value resulting from applying the operation to the result of evaluating the left-hand tree and the result of evaluating the right-hand tree.
- If *expr* is an assignment, then evaluate the expression in the right-hand tree and find the name of the variable on the left-hand side of the expression; change the dictionary *env* so that the variable *name* is associated with the value of the expression from the right-hand side. Note that all the values in the environment should be floating point numbers.

Optional: You can make your program more beautiful and compact, using functional programming style, by storing the procedures associated with each operator in the subclass. The Python module `operator` provides definitions of the procedures for the arithmetic operators. Here is an example of using operators.

```
import operator
>>> myOp = operator.add
>>> myOp(3, 4)
7
```

Step 3. Write an `eval` method for each of the expression classes that might be returned by the parser. It should take the environment as an argument and return a number. In real life, we would worry a lot about error checking; for now, just assume that you are only ever given perfect expressions to evaluate.

Test your program incrementally, using expressions like:

```
>>> env = {}
>>> Number(6.0).eval(env)
6.0
>>> env['a'] = 5.0
>>> Variable('a').eval(env)
5.0
>>> Assign(Variable('c'), Number(10.0)).eval(env)
>>> env
{'a': 5.0, 'c': 10.0}
>>> Variable('c').eval(env)
10.0
```

You may find the `testEval` procedure useful for testing your code.

5.2 Putting it all together

Now, it's time to put all your pieces together and test your calculator. The work file defines `calc`, a procedure that will prompt the user with a `'%'` character, then read in the next line of input that the user types into a string called `inp`. On the following line, you should make whatever calls are necessary to tokenize, parse, and evaluate that input. The procedure will print the result of the evaluation, as well as the state of the environment after that evaluation.

For debugging, it can be easier to type in all the expressions at once. The `calcTest` procedure in the work file takes a list of strings as input, and processes them one by one (much the way Idle works when you ask it to 'run' a Python file). You can use `testExprs` in the work file, as input to this procedure for testing. And feel free to make up test cases of your own.

Step 4. Fill in the `calcTest` procedure, so that it calls your code, and make sure it works on the examples. Here is the desired behavior of the evaluator on `testExprs`:

```
>>> calcTest(testExprs)
% (2 + 5)
7.0
   env = {}
% (z = 6)
None
   env = {'z': 6.0}
% z
6.0
   env = {'z': 6.0}
% (w = (z + 1))
None
   env = {'z': 6.0, 'w': 7.0}
% w
7.0
   env = {'z': 6.0, 'w': 7.0}
```

Wk.2.4.3 **Note that this is due at a later date than the first two problems.** After you have debugged your code in Idle, submit it via this tutor problem. You should include the class definitions for `Sum`, `Prod`, `Quot`, `Diff`, `Assign`, `Number`, `Variable` and any other class or procedure definitions that they depend on. Include comments in your code.

6 Extensions

You should do **one** of the extensions to the calculator described below: tokenizing by state machines or lazy partial evaluation. Submit your program (in Wk.2.4.4) and your answer to the corresponding Check Yourself question (in Wk.2.4.5).

6.1 Tokenizing by State Machine

Step 5. Write a state machine class, called `Tokenizer`, whose input on each time step is a single character and whose output on each time step is a either a token (a string of 1 or more characters) or the empty string, '', if no token is ready. `Tokenizer` should be a subclass of `sm.SM`. Remember that the state of a state machine can be a string.

Here are some examples. Note that there **must be** a space at the end of the string.

```
Tokenizer().transduce('fred ')
['', '', '', '', 'fred']
Tokenizer().transduce('777 ')
['', '', '', '777']
Tokenizer().transduce('777 hi 33 ')
['', '', '', '777', '', '', 'hi', '', '', '33']
Tokenizer().transduce('**-)( ')
['', '*', '*', '-', ')', '(']
Tokenizer().transduce('(hi*ho) ')
['', '(', '', 'hi', '*', '', 'ho', ')']
Tokenizer().transduce('(fred + george) ')
['', '(', '', '', '', 'fred', '', '+', '', '', '', '', '', '', 'george', ')']
```

Step 6. Now, write a procedure `tokenize(inputString)` that takes a string of characters as input and returns a list of tokens as output. The output of `tokenize('(fred + george) ')` should be `['(', 'fred', '+', 'george', ')']`. To do this, your procedure should:

- Make an instance of your `Tokenizer` state machine.
- Call its transduce method on the input string, **with a space character appended to the end of it.** An important thing to understand about Python is that almost any construct that iterates over lists or tuples will also iterate over strings. So, even though `transduce` was designed to operate on lists, it also operates on strings: if we feed a string into the `transduce` method of a state machine, it will call the `step` method with each individual character in the string.
- Remove the empty strings to return a list of good tokens.

Wk.2.4.4 **Note that this is due at a later date than the first two problems.**
After you have debugged your code in Idle, submit it via this tutor problem. Enter the `Tokenizer` state machine class and the `tokenize` procedure, as specified above. Also enter any helper procedures that you might have written; do not enter any procedures or classes that we gave you. Include comments in your code.

Check Yourself 1.

- Explain precisely why you need a space character appended to the end of the input to the `Tokenizer` input.
- Compare and contrast your two tokenizer implementations.

Wk.2.4.5 **Note that this is due at a later date than the first two problems.**
Enter your answer to Check Yourself 1 question into this Tutor problem.

6.2 Lazy partial evaluation

Or, you can do this extension. You should do **one** of these extensions to the calculator.

To make the calculator flexible, we will allow you to define an expression, like `(d = (b + c))`, even before `b` and `c` are defined. Later, if `b` and `c` are defined to have numeric values, then evaluating `d` will result in a number.

Step 7. Change your `eval` methods, so that they are lazy, and can handle symbolic expressions for which we do not have values of all the symbols.

- If the expression is a `Variable`, test to see if it is in the dictionary. If it is in the dictionary, return the result or evaluate the value for the variable in the environment, otherwise, simply return the variable. (The Python expression `'a' in d` returns `True` if the string `'a'` is a key in dictionary `d`).
- When you evaluate an assignment *do not* evaluate the right hand side; simply assign the value of the variable in the environment to be the unevaluated syntax tree. Notice this means that the values in the environment will always be syntax trees and not numbers as in eager evaluation. This is called *lazy evaluation*, because we don't evaluate expressions until we need their values.
- If your expression is an arithmetic operation, evaluate both the left and right subtrees. If they are both actual numbers, then return a new number computed using the appropriate operator, as before. If not, then make a new instance of the operator class, whose left and right children are the results of having evaluated the left and right children of the original expression (because

the evaluation process may have simplified one or the other of the arguments) and return the operator node. This is called *partial evaluation* because we only evaluate the expression to the degree allowed by the variable bindings.

- When you look a variable up in the environment, evaluate the result before returning it, because it might be a symbolic expression.

If you want to check whether something is an actual number (float or int), you can use the `isNum` procedure defined in the work file.

Note that, if you are writing your `eval` method in the `BinaryOp` class, you will need to be able to make a new instance of the subclass that `self` belongs to (e.g. `Sum`). Python provides a `__class__` method for all objects, so that `self.__class__` can be called to create a new instance of that same class.

Here are some ideas for testing eval by itself:

```
>>> env = {}
>>> Assign(Variable('a'), Sum(Variable('b'), Variable('c'))).eval(env)
>>> Variable('a').eval(env)
Sum(Var(b), Var(c))
>>> env['b'] = Number(2.0)
>>> Variable('a').eval(env)
Sum(2.0, Var(c))
>>> env['c'] = Number(4.0)
>>> Variable('a').eval(env)
6.0
```

```
>>> calcTest(lazyTestExprs)
% (a = (b + c))
None
   env = {'a': Sum(Var(b), Var(c))}
% (b = ((d * e) / 2))
None
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0))}
% a
Sum(Quot(Prod(Var(d), Var(e)), 2.0), Var(c))
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0))}
% (d = 6)
None
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'd': Num(6.0)}
% (e = 5)
None
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd': Num(6.0)}
% a
Sum(15.0, Var(c))
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd': Num(6.0)}
% (c = 9)
None
   env = {'a': Sum(Var(b), Var(c)), 'c': Num(9.0), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd':
Num(6.0)}
% a
24.0
   env = {'a': Sum(Var(b), Var(c)), 'c': Num(9.0), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd':
Num(6.0)}
% (d = 2)
None
   env = {'a': Sum(Var(b), Var(c)), 'c': Num(9.0), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd':
Num(2.0)}
% a
14.0
   env = {'a': Sum(Var(b), Var(c)), 'c': Num(9.0), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd':
Num(2.0)}
```

```
>>> calcTest(partialTestExprs)
% (z = (y + w))
None
```

```
   env = {'z': Sum(Var(y), Var(w))}
% z
Sum(Var(y), Var(w))
   env = {'z': Sum(Var(y), Var(w))}
% (y = 2)
None
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w))}
% z
Sum(2.0, Var(w))
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w))}
% (w = 4)
None
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w)), 'w': Num(4.0)}
% z
6.0
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w)), 'w': Num(4.0)}
% (w = 100)
None
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w)), 'w': Num(100.0)}
% z
102.0
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w)), 'w': Num(100.0)}
```

Wk.2.4.4 **Note that this is due at a later date than the first two problems.**
After you have debugged your code in Idle, submit it via this tutor problem. You should include the class definitions for `BinaryOp`, `Sum`, `Prod`, `Quot`, `Diff`, `Assign`, `Number`, `Variable` and any other class or procedure definitions that they depend on. Include comments in your code.

Check Yourself 2. What happens if you evaluate

```
(a = 5)
(a = (a + 1))
a
```

- Using eager evaluation?
- Using lazy evaluation?

Explain why.

Wk.2.4.5 **Note that this is due at a later date than the first two problems.**
Enter your answer to Check Yourself 2 into this Tutor problem.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011