Recitation: Symbolic Algebra

In today’s recitation, we will develop a Python framework for symbolic algebra. In such a system, algebraic expressions including variables and numbers are not immediately evaluated but rather are stored in symbolic form.

```python
>>> x = Var('x')
>>> y = Var('y')
>>> print(x + y)
(x + y)
>>> z = x + 2*x*y + x
>>> print(z)
((x + ((2 * x) * y)) + x)
>>> print(zderiv('x'))  # derivative of z with respect to x
((1 + (((2 * x) * 0) + (y * ((2 * 1) + (x * 0)))))) + 1)
>>> print(zderiv('x').simplify())
((1 + (y * 2)) + 1)
>>> print(zderiv('y'))  # derivative of z with respect to y
((0 + (((2 * x) * 1) + (y * ((2 * 0) + (x * 0)))))) + 0)
>>> print(zderiv('y').simplify())
(2 * x)
>>> z.eval({'x': 3, 'y': 7})  # evaluate an expression with particular values for the variables
48
```

We'll start with several basic classes: a base class from which all symbols will inherit, a class to represent variables (such as \(x\) and \(y\)), and a class to represent numbers:

```python
class Symbol:
    pass

class Var(Symbol):
    def __init__(self, n):
        self.name = n

    def __str__(self):
        return self.name

    def __repr__(self):
        return "Var(" + repr(self.name) + ")"

class Num(Symbol):
    def __init__(self, n):
        self.n = n

    def __str__(self):
        return str(self.n)

    def __repr__(self):
        return "Num(" + repr(self.n) + ")"
```
Symbolic Algebra: Combinations

And then we will add support for more complicated pieces with additional classes. For example, the following class represents the sum of two symbolic expressions:

class Add:
    def __init__(self, left, right):
        self.left = left
        self.right = right
    def __str__(self):
        return f'({self.left} + {self.right})'
    def __repr__(self):
        return f'Add({self.left!r}, {self.right!r})'

Check Yourself:

Think about adding a class Sub to represent subtraction rather than addition. What would such a class look like? How could we organize things so we don't have a bunch of duplicated code?

What if we wanted to expand further, supporting additional operations?

- Add, to represent an addition
- Sub, to represent a subtraction
- Mul, to represent a multiplication
- Div, to represent a division
Symbolic Algebra: Operations

So far the classes we've written are nice but they don't do much...let's add some more functionality to them.

Using Python Operators

Entering expressions of the form \( \text{Add}(\text{Var('x')}, \text{Add}(\text{Num}(3), \text{Num}(2))) \) can get a little bit tedious. It would be much nicer, for example, to be able to enter that expression as \( \text{Var('x')} + \text{Num}(3) + \text{Num}(2) \).

We would like to support the following operations:

- \( E_1 + E_2 \) results in an instance \( \text{Add}(E_1, E_2) \)
  (note: you can override the behavior of + with the \_add\_ and \_radd\_ "dunder" methods)
- \( E_1 - E_2 \) results in an instance \( \text{Sub}(E_1, E_2) \)
  (note: you can override the behavior of - with the \_sub\_ and \_rsub\_ "dunder" methods)
- \( E_1 * E_2 \) results in an instance \( \text{Mul}(E_1, E_2) \)
  (note: you can override the behavior of * with the \_mul\_ and \_rmul\_ "dunder" methods)
- \( E_1 / E_2 \) results in an instance \( \text{Div}(E_1, E_2) \)
  (note: you can override the behavior of / with the \_truediv\_ and \_rtruediv\_ "dunder" methods)

Derivatives

Next, we'll make the computer do our 18.01 homework for us. Well, not quite. But we’ll implement support for symbolic differentiation. In particular, we would like to implement the following rules for partial derivatives (where \( x \) is an arbitrary variable, \( c \) is a constant or a variable other than \( x \), and \( u \) and \( v \) are arbitrary expressions):

\[
\begin{align*}
\frac{\partial}{\partial x} c &= 0 \\
\frac{\partial}{\partial x} x &= 1 \\
\frac{\partial}{\partial x} (u + v) &= \frac{\partial}{\partial x} u + \frac{\partial}{\partial x} v \\
\frac{\partial}{\partial x} (u - v) &= \frac{\partial}{\partial x} u - \frac{\partial}{\partial x} v \\
\frac{\partial}{\partial x} (u \cdot v) &= u \left( \frac{\partial}{\partial x} v \right) + v \left( \frac{\partial}{\partial x} u \right) \\
\frac{\partial}{\partial x} \left( \frac{u}{v} \right) &= \frac{v \left( \frac{\partial}{\partial x} u \right) - u \left( \frac{\partial}{\partial x} v \right)}{v^2}
\end{align*}
\]

This should be implemented as a function \( e.\text{deriv}(\text{var}) \), where \( e \) is an expression and \( \text{var} \) is a string representing the name of the variable with respect to which we want to differentiate.
Simplification

As demonstrated on the first page of the handout, even if we've implemented the deriv code correctly, it often produces output that is difficult to read. To help with this, we would like to be able to call e.simplify() to produce an equivalent but simpler form of e according to the following rules:

- Any binary operation on two numbers should simplify to a single number containing the result.
- Adding 0 to (or subtracting 0 from) any expression E should simplify to E.
- Multiplying or dividing any expression E by 1 should simplify to E.
- Multiplying any expression E by 0 should simplify to 0.
- Dividing 0 by any expression E should simplify to 0.
- A single number or variable always simplifies to itself.

Evaluation

Finally, we would like to support evaluation of symbolic expressions, where we can convert them back to numbers. For any symbolic expression sym, sym.eval(mapping) should find a numerical value for the given expression. mapping should be a dictionary mapping variable names to values. For example:

```python
>>> z = Add(Var('x'), Sub(Var('y'), Mul(Var('z'), Num(2))))
>>> z.eval({'x': 7, 'y': 3, 'z': 9})
-8
>>> z.eval({'x': 3, 'y': 10, 'z': 2})
9
```

For now, we will only think about the case where all necessary variables are included in the given dictionary.

Thinking About Implementation

Check Yourself:

For each of the pieces above (operators, differentiation, and simplification), where should they be implemented in terms of a class hierarchy?
Symbolic Algebra: Parsing Symbolic Expressions

Finally, we would like to support parsing strings into symbolic expressions (to provide yet another means of input). For example, we would like to do something like:

```python
>>> expression('(x * (2 + 3))')
Mul(Var('x'), Add(Num(2), Num(3)))
```

To this end, we will define a stand-alone function called `expression`, which takes a single string as input. This string should contain either:

- a single variable name,
- a single number, or
- a fully parenthesized expression of the form `(E1 op E2)`, representing a binary operation (where `E1` and `E2` are themselves strings representing expressions, and `op` is one of `+`, `-`, `*`, or `/`).

We'll assume that the string is always well-formed and fully parenthesized (for now, we do not need to handle erroneous input), but it should work for arbitrarily deep nesting of expressions.

This process is often broken down into two pieces: tokenizing (to break the input string into meaningful units) and parsing (to build our internal representation from those units).

All-in-all, then, our `expression` function might look something like:

```python
def expression(inp):
    return parse(tokenize(inp))
```

Tokenization

We'll start by taking a string as input and outputting a list of meaningful tokens (parentheses, variable names, numbers, or operands).

For our purposes, you may assume that variables are always single-character alphabetic characters and that all numbers are integers, and you may also assume that there are spaces separating operands and operators.

As an example:

```python
>>> tokenize("(x * (2 + 3))")
['(', 'x', '*', '(', '2', '+', '3', ')', ')']
```

We won't worry about this part for now (you'll get to write something similar in the week 11 lab), but it's important to understand this structure for the next part, `parsing`. 
Parsing

Next, we'll take that list of tokens and convert it into an actual symbolic expression. The approach we'll take here is known as a recursive-descent parser. One way to structure parse is to use a helper function:

```python
def parse(tokens):
    def parse_expression(index):
        pass  # your code here
    parsed_expression, next_index = parse_expression(0)
    return parsed_expression
```

The function parse_expression is a recursive function that takes as an argument an integer indexing into the tokens list and returns a pair of values:

- the expression found starting at the location given by index (an instance of one of the Symbol subclasses), and
- the index beyond where this expression ends (e.g., if the expression ends at the token with index 6 in the tokens list, then the returned value should 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 three cases. Let token be the token at location index; the cases are:

- **Number**: If token represents an integer, then make a corresponding Num instance and return that, paired with index + 1 (since a number is represented by a single token).

- **Variable**: If token represents a variable name (a single alphabetic character), then make a corresponding Var instance and return that, paired with index + 1 (since a variable is represented by a single token).

- **Operation**: Otherwise, the sequence of tokens starting at index must be of the form: (E1 op E2). Therefore, token must be (i.e., in this case, we need to recursively parse the two subexpressions, combine them into an appropriate instance of a subclass of BinOp (determined by op), and return that instance, along with the index of the token beyond the final right parenthesis.
Try It!

Fill in the code for `parse` below:

```python
def parse_expression(index):
```