CS 3430: SciComp with Py
Assignment 05
Symbolic Math: Function Representations

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 08, 2020

Learning Objectives

1. Function Representations
2. Parsing Polynomial Strings into Function Representations
3. Converting Function Representations into Python Functions

Introduction

In this assignment, we'll play with function representations, figure out how to parse strings with polynomials into their function representations, and implement an algorithm to convert polynomial function representations into Py functions. You'll save your coding solutions in `parser.py` and `tof.py` included in the zip. Code up your solutions for reuse, because we'll need them in the next assignment when we implement a simple differentiation engine. Included in the zip is `cs3430_s20_hw05_uts.py` where I've written 10 unit tests you can use to test your code with as you implement it. I'll most likely write a few more tests for the grader to stress test your solutions some more, but if your code passes all these unit tests, it'll probably pass my additional tests.

Let's dive right in. Recall from Lecture 09 that a function representation is a data structure that represents a function. We need these representations to do differentiation. The zip for this assignment contains the files `var.py`, `const.py`, `pwr.py`, `plus.py`, `prod.py`, `pwr.py`, and `maker.py` with the classes we discussed in lecture 09. Fire up Python on your laptop (or whatever device you do your homework on), grab your favorite beverage, make yourself comfortable, and let's play with these classes.

First off, we need to import them.

```
>>> from const import const
>>> from pwr import pwr
>>> from var import var
>>> from plus import plus
>>> from maker import maker
```

We can use the static methods of the maker class in `maker.py` to create a few variables.

```
>>> x = maker.make_var('x')
>>> y = maker.make_var('y')
>>> z = maker.make_var('z')
>>> isinstance(x, var)
True
>>> isinstance(y, var)
True
>>> isinstance(z, var)
True
```

Let's create a few constants.

```
>>> c1 = maker.make_const(val=1.0)
>>> c2 = maker.make_const(val=3.0)
>>> import math
>>> c3 = maker.make_const(val=math.sqrt(10))
>>> isinstance(c1, const)
True
>>> isinstance(c2, const)
True
>>> isinstance(c3, const)
True
>>> assert c1.get_val() == 1.0
>>> assert c2.get_val() == 3.0
>>> assert c3.get_val() == math.sqrt(10)
```

Let's create a few power expressions. Recall from our discussion in Lecture 09 that a power expression is a variable raised to a real power. In our representation, a power expression is an object whose base is a variable (i.e., a `var` object) and whose exponent (i.e., degree) is a constant (i.e., a `const` object).

```
>>> p1 = maker.make_pwr('x', 2.0)
>>> p2 = maker.make_pwr('y', 2.0)
>>> p3 = maker.make_pwr('z', 5.0)
>>> isinstance(p1, pwr)
True
>>> isinstance(p2, pwr)
True
>>> isinstance(p3, pwr)
True
>>> isinstance(p1.get_base(), var)
True
>>> isinstance(p1.get_deg(), const)
True
>>> print(p1)
(x^2.0)
>>> print(p2)
(y^2.0)
>>> print(p3)
(z^5.0)
>>> p1.get_deg().get_val() == 2.0
True
>>> p2.get_deg().get_val() == 2.0
```

```
True
>>> p3.get_deg().get_val() == 5.0
True
```

A sum is a `plus` object that consists of two elements. The class that allows us to represent sums is in `plus.py`. By the way, we don't use the name `sum`, because it's the name of a Python function. Let's create a few binary sums.

```
>>> sum1 = maker.make_plus(maker.make_const(1.0), maker.make_const(2.0))
>>> print(sum1)
(1.0+2.0)
>>> sum2 = maker.make_plus(maker.make_pwr('x', 2.0), maker.make_const(10.0))
>>> print(sum1)
(1.0+2.0)
>>> print(sum2)
((x^2.0)+10.0)
>>> isinstance(sum1.get_elt1(), const)
True
>>> isinstance(sum1.get_elt2(), const)
True
>>> isinstance(sum2.get_elt1(), pwr)
True
>>> isinstance(sum2.get_elt2(), const)
True
```

We can represent sums of arbirarily many elements as binary sums. Here's how to represent $x^2 + x + 10$.

```
>>> sum2 = maker.make_plus(maker.make_plus(maker.make_pwr('x', 2.0),
                                           maker.make_pwr('x', 1.0)),
                           maker.make_const(10.0))
>>> print(sum2)
(((x^2.0)+(x^1.0))+10.0)
```

A product is an object that consists of two multiplicands. The class that allows us to represent products is defined in `prod.py`. Let's create $5x^2$ and $5x^2 - 10x^5$.

```
>>> p1 = maker.make_prod(maker.make_const(5), maker.make_pwr('x', 2.0))
>>> print(p1)
(5*(x^2.0))
>>> p2 = maker.make_prod(maker.make_prod(maker.make_const(5), maker.make_pwr('x', 2.0)),
                         maker.make_prod(maker.make_const(-10), maker.make_pwr('x', 5.0)))
>>> print(p2)
((5*(x^2.0))*(-10*(x^5.0)))
>>> print(p1.get_mult1())
5
>>> print(p2.get_mult2())
(-10*(x^5.0))
>>> print(p1)
(5*(x^2.0))
>>> print(p1.get_mult2())
(x^2.0)
```

```
>>> print(p2.get_mult1())
(5*(x^2.0))
>>> print(p2.get_mult2())
(-10*(x^5.0))
>>> print(p2.get_mult2().get_mult1())
-10
>>> print(p2.get_mult2().get_mult2())
(x^5.0)
>>> print(p2.get_mult2().get_mult2().get_deg())
5.0
```

We can use `plus` and `prod` objects to represent polynomials. Let's represent $5x^3 + 10x^2 + x + 100$.

```
elt1 = maker.make_prod(maker.make_const(5),
                       maker.make_pwr('x', 3))
elt2 = maker.make_prod(maker.make_const(10),
                       maker.make_pwr('x', 2))
elt3 = maker.make_prod(maker.make_const(1),
                       maker.make_pwr('x', 1))
elt4 = maker.make_const(100)
poly_sum = maker.make_plus(maker.make_plus(maker.make_plus(elt1, elt2),
                                           elt3),
                           elt4)
>>> print(poly_sum)
((((5*(x^3))+(10*(x^2)))+(1*(x^1)))+100)
>>> print(poly_sum.get_elt1())
(((5*(x^3))+(10*(x^2)))+(1*(x^1)))
>>> print(poly_sum.get_elt2())
100
>>> print(poly_sum.get_elt1().get_elt1())
((5*(x^3))+(10*(x^2)))
>>> print(poly_sum.get_elt1().get_elt1().get_elt2().get_mult2())
(x^2)
```

Problem 1: (1 point)

Let's implement a simple parser that takes a string that represents a polynomial and converts it into a function representation. We won't implement a full blown symbolic math parser required for a real differentiation engine, which would be a semester long project in and of itself. Rather, we'll confine ourselves to polynomials of the following form $a_n x^{r_n} + a_{n-1} x^{r_{n-1}} + ... + a_1 x^{r_1} + a_0 x^{r_o}$, where a_i and r_i are reals. Of course, we can use variables other than x in our polynomials. Thus, $5x^2$, $10.5y^{-3} + 5y^2 - 10y^{15} + 100$, $5.312z^{10} - 20.7z^{-2.5} + 4z^5 - 5.14z^3 + 10$ are valid polynomials.

Let's make a few simplifying assumptions about string representations of polynomials. We'll represent each polynomial element with the caret sign. For example, $5x^2$ is represented as `'5x^2'`, $10.5y^{-3}$ as `'10.5y^-3'`, $5.14z^3$ as `'5.14z^3'`. To make our representation uniform, let's represent constants as products whose second multiplicand is the variable raized to the 0^{th} power. For example, 100 is represented as `'100x^0'`. Let's also represent variables raised to the power of 1 as products of 1 and the variable raised to the power of 1. Thus, x is represented as `'1x^1'`, y as `'1y^1'`, etc. Here are a few examples.

1. $5x^2$ is written as `'5x^2'`;
2. $10.5y^{-3} + 5y^2 - 10y^{15} + 100$ is written as `'10.5y^-3 + 5y^2 - 10y^15 + 100x^0'`;

3. $5.312z^{10} - 20.7z^{-2.5} + 4z^5 - 5.14z^3 + z + 10$ is written as
   `'5.312z^10 - 20.7z^-2.5 + 4z^5 - 5.14z^3 + 1z^1 + 10z^0'`.

Let's further agree not to have double minuses. In other words, we wont' have polynomials written as $5x^2 - -3x^3$. However, it's fine to have a coefficient minus follow a plus: $5x^2 + -3x^3$.

Implement the static method `parser.parse_elt(elt)` that takes a string representation of a polynomial element and converts it into a `prod` object. Here's a unit test (see `test_hw05_prob01_ut01()` in `cs3430_s20_hw05_uts.py`).

```
s1 = '5x^10'
e1 = parser.parse_elt(s1)
```

If our implementation is correct, all the assertions below pass.

```
err = 0.0001
assert str(e1) == '(5.0*(x^10.0))'
assert isinstance(e1, prod)
assert isinstance(e1.get_mult1(), const)
assert abs(e1.get_mult1().get_val() - 5.0) <= err
assert isinstance(e1.get_mult2(), pwr)
assert isinstance(e1.get_mult2().get_base(), var)
assert e1.get_mult2().get_base().get_name() == 'x'
assert isinstance(e1.get_mult2().get_deg(), const)
assert abs(e1.get_mult2().get_deg().get_val() - 10.0) <= err
```

Here's another unit test (see `test_hw05_prob01_ut04()` in `cs3430_s20_hw05_uts.py`).

```
s1 = '1001.7341z^-11.57'
e1 = parser.parse_elt(s1)
```

All the assertions below pass.

```
err = 0.0001
assert isinstance(e1, prod)
assert isinstance(e1.get_mult1(), const)
assert abs(e1.get_mult1().get_val() - 1001.7341) <= err
assert isinstance(e1.get_mult2(), pwr)
assert isinstance(e1.get_mult2().get_base(), var)
assert e1.get_mult2().get_base().get_name() == 'z'
assert isinstance(e1.get_mult2().get_deg(), const)
assert abs(e1.get_mult2().get_deg().get_val() + 11.57) <= err
assert str(e1) == '(1001.7341*(z^-11.57))'
```

On to sums. Implement the static method `parse_sum(poly_str)` in `parser.py` that takes a string representation of a polynomial written according to the above conventions and returns a `plus` object representing this polynomial.

Here's a unit test (see `test_hw05_prob01_ut05()` in `cs3430_s20_hw05_uts.py`).

```
s1  = '5x^-10 + 3x^5'
sum_ex  = parser.parse_sum(s1)
```

All the assertions below pass.

```
err = 0.0001
assert isinstance(sum_ex, plus)
assert isinstance(sum_ex.get_elt1(), prod)
assert isinstance(sum_ex.get_elt2(), prod)
e1 = sum_ex.get_elt1()
assert abs(e1.get_mult1().get_val() - 5.0) <= err
assert isinstance(e1.get_mult2(), pwr)
assert isinstance(e1.get_mult2().get_base(), var)
assert e1.get_mult2().get_base().get_name() == 'x'
assert isinstance(e1.get_mult2().get_deg(), const)
assert abs(e1.get_mult2().get_deg().get_val() + 10.0) <= err
assert str(e1) == '(5.0*(x^-10.0))'
e2 = sum_ex.get_elt2()
assert abs(e2.get_mult1().get_val() - 3.0) <= err
assert isinstance(e2.get_mult2(), pwr)
assert isinstance(e2.get_mult2().get_base(), var)
assert e2.get_mult2().get_base().get_name() == 'x'
assert isinstance(e1.get_mult2().get_deg(), const)
assert abs(e2.get_mult2().get_deg().get_val() - 5.0) <= err
assert str(e2) == '(3.0*(x^5.0))'
```

More unit tests are in `cs3430_s20_hw05_uts.py`.

Problem 2: (2 points)

Our function representations are not that useful unless we can convert them into real Python functions that we can evaluate on specific values. Let's address this issue and implement the static method `tof.tof(ex)` in `tof.py` (`tof` stands for "to function") that takes a function representation `fr` returned by `parser.parse_sum()` or `parser.parse_elt()` and returns a Python function that corresponds to that representation. This method should handle `const`, `var`, `prod`, `pwr`, and `plus` objects. If `fr` is not any of these, this static method throws an exception. Here's a unit test (see `test_hw05_prob02_ut01()` in `cs3430_s20_hw05_uts.py`).

```
ex = '5x^2'
my_fun = tof.tof(parser.parse_sum(ex))
```

Let's define a ground truth function (`gt_fun`) and compare its output with the output of the function returned by `tof.tof()`.

```
gt_fun = lambda x: 5.0*(x**2.0)
err = 0.0001
for x in range(-1000000, 1000000):
    assert abs(my_fun(x) - gt_fun(x)) <= err
```

Here's another unit test (see `test_hw05_prob02_ut03()` in `cs3430_s20_hw05_uts.py`).

```
ex = '5x^-2 - 3x^5 + 4.5x^7.342'
my_fun = tof.tof(parser.parse_sum(ex))
```

Again, we define a ground truth function (`gt_fun`) and compare its output with the output of the function returned by `tof.tof()`.

```
gt_fun = lambda x: 5.0*(x**-2.0) - 3.0*(x**5.0) + 4.5*(x**7.342)
err = 0.0001
for x in range(-1000000, 0):
    assert abs(my_fun(x) - gt_fun(x)) <= err
for x in range(1, 1000000):
    assert abs(my_fun(x) - gt_fun(x)) <= err
```

What To Submit

Submit your code in `parser.py` and `tof.py`. It'll be easiest for us to grade your code if you place all the files (i.e., `var.py`, `const.py`, `pwr.py`, `plus.py`, `prod.py`, `pwr.py`, `maker.py`, `parser.py`, and `tof.py`) into one directory, zip it into `hw05.zip`, and upload your zip in Canvas.

Happy Hacking!