CSE 390P: Problem Solving with Programming   Winter 2018

Day 2: Symbolic Algebra

We begin by completely switching languages. Although SICP uses Scheme the whole way through, we will explore different languages as we make our way through the book. For Day 2, we'll use Javascript. In particular, we'll use node.js, but we will *not use any of its libraries*.

Today, you will create your own (simple) version of wolfram alpha.

Getting Familiar With Javascript

(a) In Javascript, we use the `define` procedure to define a variable and the `print` procedure to print strings.

```
var hello = "hello world";
process.stdout.write("We want to print " + hello);

> We want to print hello world
```

(b) We define procedures in a very similar way:

```
function print_greeting(greeting) {
   return process.stdout.write(greeting + " World");
}
print_greeting("Hello")
print_greeting("Ahoy")

> Hello World
> Ahoy World
```

Diving In

One of the things we'll explore in this course is *modifying* existing code rather than working from scratch. This can sometimes be more difficult, but it's a common scenario. In particular, we've started a symbolic algebra library for you, and you will be implementing missing features. Download the file from the course website, `ssh` to `attu`, and start trying reading through the code on `attu`! (These instructions are deliberately vague and useless. If nobody in your group is familiar with UNIX, you should call me over for help.)

Parsing Mathematical Expressions & The Shunting Yard Algorithm

Last time we saw *prefix* notation–this time, we will end up using *postfix* notation which is slightly more complicated. However, it has the advantage of being unambiguous without *needing any parentheses* which is very helpful. We've handled all the details of postfix notation for you, but if you're curious about more details, stop here and look it up on Wikipedia.

(a) **Reading 1.1**
Read the `shunting_yard` code and make sure you understand it before moving on.

(b) **Task 1.2**
In this task, you will modify the `shunting_yard` algorithm to deal with unary negation (for example, the expression $-x - y$ uses unary negation on x and binary subtraction on $-x$ and y.

Modifying the `shunting_yard` code involves two sub-parts:

(1) First, you will need to determine for each $-$ sign you see if it is subtraction or unary negation. To do this, consider the *previous* token (operator, parenthesis, number, or variable). In the cases where (1) there is no previous token, (2) the previous token is an operator, or (3) the previous token is a left parenthesis, then it is *necessarily* a unary minus. In all other cases, it should be interpreted as binary subtraction. You will need to distinguish between these two in your *output*; so, you should use a separate symbol (we used !) for unary negation.

(2) Second, you will need to create a new "type" of output. Take a look at the post-processing step where the code interprets the postfix expression. You will need to add a case there for a `type` 'unary operation' with `op` '!'.

Checking Equality of Expressions

In a computer algebra system, the user often wants to know if two expressions, e_1 and e_2 are equal. This is a non-trivial problem that gets very complicated, but the general idea is to take $e_1 - e_2$ and simplify it as much as possible. Then, if it's zero, we say they're equal; if not, we say "probably not equal". So, equality boils down to a good simplification function.

There are lots of parts to simplification of algebraic expressions, but some of them are pretty straightforward. In particular, if you take simple mathematical identities like $x * 0 = 0$ and $x * x = x^2$ and apply as many identities as possible, it gets you pretty far. We've given you the identities. . . but not code that checks if they apply.

(a) **Task 2.1**

The heart of our algorithm is in what we call a "transformation". A "transformation" is a general identity written out and parsed in our own algebraic system. This allows us to compare structures of the expressions to see which identities apply. We've outlined the algorithm for you in the `make_transformation` function. First, we convert the provided pieces of the identity into expressions. Then, we check if `expr` "matches" the pattern. Finally, if it does match the pattern, we make a substitution.

We take the convention that a, b, c, and d stand for *numbers*, and w, x, y, and z stand for *arbitrary expressions*. For each case of the matching algorithm, you should check if the pattern recursively matches `expr`. Keep in mind that your variable assignments should not conflict. To test equality of expressions, you will find the `exactly_equal` function useful (which we have written for you).

For the second part, where you actually replace the values, use the modified map from the previous part to recursively substitute variables in the OUTPUT pattern with their assignments.

(b) **Task 2.2**

Now that you have a transformation maker, it's time to update our code to handle unary expressions.

(c) **Task 2.3**

To simplify polynomials, we will need to group like-terms. Unfortunately, our expressions are all binary and unary rather than n-ary. To deal with this, we introduce two new types of expression nodes: $+$ and $*$ which are general addition and multiplication, respectively. Then, we recursively collect the addition expressions and the multiplication expressions. In essence, a polynomial is just a bunch of additions of multiplications/negations. Read and understand `collect_op` before continuing.

Now that we've collected together each term, we need to identify them individually. For example, we might have $2xyx + 2y + 3xxy + 5y + 3yx^2 = 8x^2y + 7y$. To identify all three of these terms as the same, we first need to "canonicalize" them. We do this by calling `constant_fold` on each product individually. This gives us $2x^2y + 2y + 3x^2y + 5y + 3yx^2$. Next, we "sort" the string representation of each part of each expression to order the product terms correctly. This gives us: $2x^2y + 2y + 3x^2y + 5y + 3x^2y$. Finally, we "sort" the terms of the sum; so, "like terms" are close together–which will then get simplified by our transformations which gives $8x^2y + 7y$ as expected. When sorting, you will want to make use of the `localeCompare` function as well as the regular expression "`^/-?\(*[0-9]*([^)]*)\)*$/`". As if this weren't complicated enough, there's a "gotcha" with how unary expressions interact with this algorithm. You will need to add an extra case. Try looking at the test outputs to see what is missing.

Simplifying Rational Polynomials

We're still not done simplifying! We do not do a good job of handling rational functions (polynomials divided by other polynomials). Look up and implement the synthetic division algorithm.