Solutions to *Mathematica* Homework
CS 6104: Symbolic Computation

Craig Struble

The solutions to the homework are included in the LaTeX source for this file. When compiling this source file, the two solution files, `natural.m` and `monoid.m`, are created. These solutions are *Mathematica* packages that are read in with the `<<` operator.

Problem 1

The first problem was to represent the natural numbers, $\mathbb{N} = \{0, 1, 2, ...\}$ using the successor function represented with, `succ` in *Mathematica*. The example from the problem statement was to represent 5 as `succ[succ[succ[succ[succ[0]]]]]`.

The first part of the problem is to implement addition using *Mathematica* patterns, so that `add[x,0]` returns `x`, `add[0,y]` returns `y`, and `add[x,y]` returns the successor representation for the sum of `x` and `y`. For example,

```
add[succ[succ[succ[0]]], succ[succ[0]]]
```

returns the representation for 5.

The solution for this first part of the problem is,

```
add[x_,0] := x
add[succ[x_], y_] := succ[add[x, y]]
add[x_, y_] := add[y, x]
```

The first rule matches anything, `x`, added to 0 and returns `x`. The second rule "peels" off the `succ` function from the first argument and moves it outside of the `add` function. The third rule makes addition commutative.

Some test cases to show this works are,

```
In[1]:= << natural.m

In[2]:= add[succ[succ[succ[0]]], succ[succ[0]]]

Out[2]= succ[succ[succ[succ[succ[0]]]]]

In[3]:= add[0,succ[succ[0]]]

Out[3]= succ[succ[0]]

In[4]:= add[succ[succ[0]], 0]

Out[4]= succ[succ[0]]
```

We read in the solution to make use of it. The first test is to add 3 and 2. This returns our representation of 5. The next two cases show that adding 0 and 2 in either order returns 2.

The next part of the problem was to implement negative integers with the function `neg`. For example, -2 is represented as `neg[succ[succ[0]]]`. The additional rules to implement negative integers are

```
neg[0] := 0
neg[neg[x_]] := x
succ[neg[succ[x_]]] := neg[x]
add[neg[x_], neg[y_]] := neg[add[x, y]]
```

The first rule in this set says that the negative of 0 is 0. The second rule says that the negative of a negative integer is just the integer. The third rule says that the successor of a negative number "peels" off a successor from within the negative number, i.e., we add 1 to the negative number. The final rule says that if we are adding two negative numbers, then it is just the negative of the sums of the positive parts of the numbers.

Some test cases to show that this works are

```
In[5]:= neg[0]

Out[5]= 0

In[6]:= neg[neg[succ[succ[0]]]]

Out[6]= succ[succ[0]]

In[7]:= neg[succ[neg[succ[0]]]]

Out[7]= 0

In[8]:= add[neg[succ[succ[0]]], neg[succ[0]]]

Out[8]= neg[succ[succ[succ[0]]]]

In[9]:= add[neg[succ[succ[0]]], succ[0]]

Out[9]= neg[succ[0]]
```

The first test shows that the negative of 0 is 0. The second test shows that $-(-2)$ is 2. The third test shows that $-2 + -1 = -3$. The fourth test shows that $-2 + 1 = -1$. For each test, the input and output is in our representation of natural numbers.

The optional part of this problem was to implement multiplication using this representation of natural numbers. Multiplication is representated as the `mult` function. The solution to this part is

```
mult[0,x_] := 0
mult[succ[0], x_] := x
mult[x_, y_] := mult[y, x]
```

```
mult[neg[x_], neg[y_]] := mult[x,y]
mult[succ[x_], y_] := add[mult[x,y], y]
mult[neg[x_], y_] := neg[mult[x, y]]
```

The first rule represents 0 times anything as zero. The second rule represents 1 times anything, x, is x. The third rule makes `mult` commutative. The fourth rule represents multiplying two negative integers as the same as multiplying the two positive representations of the integers. The fifth rule represents multiplication as the same as adding the second parameter to the result of multiplying one less of the first parameter. The sixth and final rule represents multiplying a negative and a positive number as the negative of multiplying the positive presentation of both integers.

Some test cases for this solution are,

```
In[10]:= mult[0, succ[0]]

Out[10]= 0

In[11]:= mult[succ[succ[0]], succ[0]]

Out[11]= succ[succ[0]]

In[12]:= mult[succ[succ[succ[0]]], succ[succ[0]]]

Out[12]= succ[succ[succ[succ[succ[succ[0]]]]]]

In[13]:= mult[neg[succ[succ[0]]], succ[succ[0]]]

Out[13]= neg[succ[succ[succ[succ[0]]]]]

In[14]:= mult[neg[succ[succ[0]]], neg[succ[succ[succ[0]]]]]

Out[14]= succ[succ[succ[succ[succ[succ[0]]]]]]
```

The first test case shows $0 * 1 = 0$. The second test case shows $2 * 1 = 2$. The third test case shows $3 * 2 = 6$. The fourth test case shows that $-2 * 2 = -4$. The fifth test case shows that $-2 * -3 = 6$. All integers are represented in our representation of integers.

Problem 2

For this problem, we need to represent a monoid M in *Mathematica*. In this representation, we do not worry about representing the generators Σ, but only the relations $R \subset \Sigma^*$.

In order to represent a monoid, we want to coerce the `Times` and `Power` functions to act correctly. *Mathematica* implements `Times` as being commutative. Since a monoid uses a non-commutative multiplication operator, we must tell *Mathematica* to make `Times` non-commutative. Also, `Power` expands $(a\ b)^2$ to a^2b^2. This is incorrect behavior for a monoid. Our representation also adds a constant string `lambda` to represent λ.

Our rules for doing this in *Mathematica* are

```
Unprotect[Power]
ClearAll[Power]
Attributes[Power] := {Listable, OneIdentity}
Power[lambda,x_] := lambda
Power[x_, 0] := lambda
Power[a_ * b_, y_?Positive] := (a * b) Power[a * b, y - 1]

Unprotect[Times]
ClearAll[Times]
Attributes[Times] := {Flat, Listable, OneIdentity}
Default[Times] := lambda
Times[lambda, a_] := a
Times[a_, lambda] := a
```

First, for both `Power` and `Times` we unprotect the definitions and clear any user-defined definition for the functions. Then, we set appropriate attributes for the functions, making sure not to include `Orderless` for `Times`. `Times` is still a flat operator, i.e., `Times[Times[x,y],z]` is the same as `Times[x,y,z]`.

Next, we define rules for `lambda`, which are representations of $\lambda^x = \lambda$, $x^0 = \lambda$, and $\lambda\ a = a\ \lambda = a$. We also override the `Power` function so that $(a\ b)^y = a\ b(a\ b)^{y-1}$ if $y > 0$.

That is the easy part. Now we need rules to perform the pattern matching on the various strings. In general, if we have a relation $r \in R$, then r will look like $a^x\ b\ c^y$ for some $x, y \geq 1$, $a, c \in \Sigma^+$, and $b \in \Sigma^*$. When we do the pattern matching, any string looking like $a^z\ b\ c^w$ will get reduced to $a^{z-x}\ c^{w-y}$ if $z \geq x, w \geq y$. In *Mathematica*, a rule that implements this looks something like

```
Times[Power[a,z_?(#>=x&)], b, Power[c, w_?(#>=y&)]] :=
      Times[Power[a, z-x], Power[c,w-y]]
```

We use this idea as the basis of our implementation.

Since *Mathematica* does not match a single character a with a^1, we need to handle patterns with single characters as special cases. This in effect causes us to generate two rules for each single character on the end of a pattern. One looks like the above rule with x replaced by a 1, so as to catch patters with exponents. The other rule matches the single character exactly. A rule that would match the single character for a relation pattern that looks like $a\ b\ c^y$ is

```
Times[a, b, Power[c, w_?(#>=y&)]] ] := Power[c, w - y]
```

We also have the cases where a relation has a single component, a^x or a. These are basically the same as the cases we have already discussed, just by removing all of the information for b and c.

Overall, there are six distinct cases. So we need six rules that will generate the necessary multiplication and power rules to perform the pattern matching. To do this, we introduce the function `createMonoidRule` which takes a relation pattern and generates the pattern matching rules for implementing the monoid with those relations. Also, we need to `Evaluate` the left hand side of the rule to make sure that the expression has been expanded completely so that *Mathematica* will perform the matching. The rules implementing this are,

```
ClearAll[createMonoidRule]
(* a^x b c^y *)
createMonoidRule[Times[Power[a_,x_Integer], b___, Power[c_,y_Integer]]] :=
        Evaluate[ Times[Power[a,z_?(#>=x&)], b, Power[c, w_?(#>=y&)]] ] :=
                Times[Power[a, z-x], Power[c,w-y]]

(* a^x b c *)
createMonoidRule[Times[Power[a_,x_Integer], b___, c_]] := (
        Evaluate[ Times[Power[a,z_?(#>=x&)], b, c] ] :=
                Power[a, z-x] ;
        Evaluate[ Times[Power[a,z_?(#>=x&)], b, Power[c, w_?(#>=1&)]] ] :=
                Times[Power[a, z-x], Power[c, w-1]]
        )

(* a b c^y *)
createMonoidRule[Times[a_, b___, Power[c_,y_Integer]]] := (
        Evaluate[ Times[Power[a,z_?(#>=1&)], b, Power[c, w_?(#>=y&)]] ] :=
                Times[Power[a, z-1], Power[c,w-y]] ;
        Evaluate[ Times[a, b, Power[c, w_?(#>=y&)]] ] :=
                Power[c, w - y]
        )

(* a b c *)
createMonoidRule[Times[a_, b___, c_]] := (
        Evaluate[Times[a, b, c]] := lambda ;
        Evaluate[ Times[Power[a, z_?(#>=1&)], b, Power[c, w_?(#>=1&)]] ] :=
                Times[Power[a, z-1], Power[c,w-1]] ;
        Evaluate[ Times[a, b, Power[c, w_?(#>=1&)]] ] :=
                Power[c,w-1] ;
        Evaluate[ Times[Power[a, z_?(#>=1&)], b, c] ] :=
                Power[a, z-1]
        )

(* a *)
createMonoidRule[a_] := (
        Evaluate[a] := lambda;
        Evaluate[Power[a, z_?(#>=1&)]] :=
                Power[a,z - 1]
        )

(* a^x *)
createMonoidRule[Power[a_,x_]] := (
        Evaluate[Power[a, z_?(#>=x&)]] :=
                Power[a,z - x]
        )
```

The final thing we do is implement a `Monoid` function that takes a list of relations and maps the `createMonoidRule` to each relation in the list. The implementation is

```
Monoid[relations_] :=
        Map[createMonoidRule, relations]
```

Now to represent a monoid, a *Mathematica* user loads the package and uses the `Monoid` function. For example,

```
n[1]:= << monoid.m

In[2]:= Monoid[{a^2 b c^2, c d}]

Out[2]= {Null, Null}
```

represents the monoid M with the relations $R = \{a^2\ b\ c^2, c\ d\}$. This is the example from the homework statement.

To check that this works, we give the string $a^4\ b\ c^3\ d$ to *Mathematica*. The result should be a^2. This is the example from the homework assignment.

```
In[3]:= a^4 b c^3 d

         2
Out[3]= a
```

Some more tests are

```
In[4]:= a^2 b c^3

Out[4]= c

In[5]:= c d

Out[5]= lambda

In[6]:= (d c)^5

Out[6]= d c
```

There is one problem with this implementation that was unable to be resolved. Expressions of the form $(a\ b\ a)^x$ get expanded incorrectly. *Mathematica* performs commutative operations when simplifying the expression, so that the above will look like $a^{2x}\ b^x$. Basically, anything that causes *Mathematica* to collapse powers more than once will cause an incorrect expansion. I was unable to prevent *Mathematica* from performing the simplification.