Getting the matching and rewriting to work

This is the description page for the substitution/matching/proving part of the **final project (https://canvas.umn.edu/courses/391238/pages/final-project-description)** .

Note that if you're unhappy about your parser, or the data-structures you picked, you're allowed to just use my implementation instead, **found here (https://canvas.umn.edu/courses/391238/files/39579893?wrap=1)** ↓ **(https://canvas.umn.edu/courses/391238/files/39579893/download?download_frd=1)** (this is the 80% solution to the previous assignment).

Scope

This time around the homework is all about getting a simple proof to go through. You can take a look at the **gettingstarted.ml (https://canvas.umn.edu/courses/391238/files/39179461?wrap=1)** ↓ **(https://canvas.umn.edu/courses/391238/files/39179461/download?download_frd=1)** file to see what you should be able to read in, and the type of proofs you are supposed to produce. Note that an example of such a proof is given in the comments of gettingstarted.ml, but the files on which your implementation will be tested will of course have no such comment. However, there are some features that don't play a role for this assignment yet:

- No comma-separated values in constructors (like: Cons (x,y))
- No definition statements (no 'let rec ...')
- No occurrences of 'match' in expressions
- No 'induction' hints (and hence no induction proofs are required)

Some features that are not in gettingstarted.ml should be handled:

- function applications with multiple arguments
- constructors with one or zero arguments
- proofs where the right hand side needs to be simplified as well

An example file that does include the above, is **given here (https://canvas.umn.edu/courses/391238/files/39579045?wrap=1)** ↓ **(https://canvas.umn.edu/courses/391238/files/39579045/download?download_frd=1)** . (Note that although the two proofs would form an induction proof, there is in fact no induction from the tool's perspective. This input file is faking it).

You'll call your executable using the --simple switch (for: simple proofs only). Take a look at **my solutions to the parser (https://canvas.umn.edu/courses/391238/files/39579893?wrap=1)** ↓ **(https://canvas.umn.edu/courses/391238/files/39579893/download?download_frd=1)** for how to pass along a function when creating a new switch (look in bin/main.ml).

```
./_build/default/bin/main.exe --simple moreproofs.ml
```

Then the output could read something like this:

```
Proof of fooxy:
  foo x y
 = ??? Could not determine a next proof step ???
  x

Proof of append_assoc_base:
  append (append Nil xs) ys
```

```
  = {append_nil}
   append xs ys
  = {append_nil}
   append Nil (append xs ys)

Proof of append_assoc_inductive_step:
   append (append (cons h tl) xs) ys
  = {append_cons}
   append (cons h (append tl xs)) ys
  = {append_cons}
   cons h (append (append tl xs) ys)
  = {ih_append_assoc}
   cons h (append tl (append xs ys))
  = {append_cons}
   append (cons h tl) (append xs ys)
```

Approach

To find a proof by calculation, we take an expression of the form 'L = R', and apply rewrite steps to both 'L' and 'R' until there is some point where they produce the same value. This means we have to write a function that applies a rewrite step. Constructing a rewrite step takes an expression and a rewrite rule, and then attempts to apply the rewrite rule to the expression. For example, if `append (append (Cons (h,t)) (Cons (y,lst2))) z` is our expression, and `append (Cons (h,t)) lst = Cons (h,append t lst)` is our rewrite rule, then there is precisely one way to carry out the rewrite step: it is to apply the rule to the sub term `append (Cons (h,t)) (Cons (y,lst2))`. To see that we can indeed apply the rule to this term, we have to find a *matching*.

A matching of a pattern x to an expression e is a *substitution* such that the pattern under that substitution becomes equal to the expression. In our example, the substitution would be: `h:=h, t:=t, lst:=Cons (y,lst2)`. It might be useful to make an abstraction of substitutions.

We'll go over the separate parts in what follows:

Subsitutions

Substitutions can be tested fully on their own, without needing to be a part of the larger project, because for the implementation it essentially does not matter that we are substituting variables by expressions. Make sure you have at least these functions:

- empty: the empty substitution
- singleton k v: a substitution that substitutes k for v
- merge s1 s2: merge substitutions s1 and s2. Note that this might fail. Choose a type to deal with this!

We'll also want to apply a substitution to an expression.

- find k s1: find variable k and return what it needs to be substituted by according to s1. It's good practice to have something meaningful happen (like a meaningful error message) in case k is not assigned to any value in s1.
- then a function that cannot be defined in isolation: substitute
  substitute s1 expr, or substitute vars s1 expr: find all the variables in expr (the argument 'vars' can be used to indicate whether an identifier is a variable or not), and substitute them according to the substitution.

Make sure to test your substitutions in utop, but they shouldn't be too complicated. You can use the module 'Map.Make(String)' to implement substitutions, but just representing them with lists will be fine too.

Matching

The most complicated function you'll implement is the one that matches a pattern *pat* to a goal *goal* by finding a substitution s such that 'substitute vars s pat = goal' (or without the 'vars' if your substitution function does not need it.). That is, if such an s exists, 'matching vars pat goal = Some s', and 'matching vars pat goal = None' otherwise.

You write this function by following the structure of the pattern recursively. Since you'll immediately want to look at the structure of 'goal', it makes sense to match on both:
matching vars pat goal = match (pat, goal) with ...

The cases you'll need are discussed in the **slides on matchings (https://canvas.umn.edu/courses/391238/files/39384734?wrap=1)** **(https://canvas.umn.edu/courses/391238/files/39384734/download?download_frd=1)** .

For matchings, there's a lot of opportunity to test: see if the matching successfully returns None if two things cannot be matched, and see if it returns the right substitution otherwise. You'll be writing a ton of expressions to do this testing, so setting up a function 'parseExpr : string -> expression' (or you can simply call it 'parse') might be a useful way to help with this. See the **halloween lab (https://canvas.umn.edu/courses/391238/pages/lab-instructions-october-31st)** on how to write a quick parser that takes a string.

Applying a single step

This is where things get exciting! Get yourself some equalities (simply pairs of expressions) and see if you can apply them to get the 'next' term in a proof. Only apply equalities left to right.

Start by only applying equalities if the outer term matches the rule. Then proceed to also trying to match sub-terms.

Let me know if this is where you're stuck (I need to write more here). After doing so, take a look at the slides of last week.