This exercise ties together concepts that we have been discussing since the first lecture to solve a real-life problem. One important learning goal in this exercise is to develop experience breaking programming languages problems into parts.

A. Start with the ideas

To type check the entire spreadsheet data structure, we'll break the problem into two parts:

1. Type check a single formula.
2. Combine the results of type checking each formula.

B. Type checking a single formula

Type checking a single formula means:

1. Checking that the return of the formula aligns with the annotated return type
2. Checking that each argument in the formula aligns with what is expected

The second step will be recursive, since the argument of a formula could also be a formula!

In order to type check a single formula, we will need the following pieces of information:

- The formula itself (whose Haskell type is `Formula`)
- The formula's annotated return type (whose Haskell type is `Type`), which should be either `StrCol` or `NumCol`
- The mapping between column names and their types. The reason is that our formula could mention other column by name, so we need a way to look up their value.

These three components are the parameters to the helper function `typeCheckFormula`, which has the following Haskell type annotation:

```
typeCheckFormula :: Formula -> Type -> TypeEnv -> Bool
```

This function will perform steps 1 and 2 at the top of Section B. Since these two steps will differ depending on the kind of Formula we have, we can pattern match on the formulas. Let's start with `(Length x)`:

Formula pattern: (Length s)

```
typecheckFormula (Length s) ctype tenv = ...
```

In order for this type of formula to type check, two things must be true: (a) the annotated return type `ctype` needs to be a `NumCol`, *and* (b) the argument formula `s` needs to be a `StrCol`. So, we need these conditions to be true:

```
typecheckFormula (Length s) ctype tenv =
    and [ctype == NumCol,                -- condition (a)
         typecheckFormula s StrCol tenv]  -- condition (b)
```

The code for condition (a) checks whether the return type is `NumCol`. The code for condition (b) recursively checks whether `s` also passes type checking.

The code for `(NumToString s)` follows the same logic.

Formula pattern: (Plus s t)

The code to type check `(Plus s t)` is actually not that different. The only difference is that we have *two* parameters to type check:

```
typecheckFormula (Plus s t) ctype tenv =
    and [ctype == NumCol,                -- type check return type
         typecheckFormula s NumCol tenv,  -- type check argument s
         typecheckFormula t NumCol tenv]  -- type check argument t
```

The code for `(Concat s t)` follows the same logic.

Formula pattern: (Column s)

For this final pattern, we need to check that the output column type is consistent with the input column type. That is, we need to check that `ctype` is consistent with the type of `k` in side the map `tenv`.

We would perform the map lookup using the expression `(Map.lookup k tenv)`. This will give us a value of type `Maybe Type`, with the possible values being `Just StrCol`, `Just NumCol` and `Nothing`. We want to make sure that the value is the same as `Just ctype`:

```
typecheckFormula (Column k) ctype tenv =
    (Map.lookup k tenv) == Just ctype
```

C. Combining the results of type checking each formula

We would like to use the helper function to complete this function:

```
typeCheck :: TypeEnv -> Formulas -> Bool
typeCheck tenv formulas = ...
```

I'll present two ways to do this problem: one by "folding" over a Haskell Map, and another by "mapping" over that Map. Since we're not familiar with Haskell Maps, the natural thing to do is to look at the functions available here:

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Foldl

The intended solution is to use `Map.foldlWithKey` to check each formula. This function has the following type signature, and works similarly to `foldl`. Our accumulator will be a boolean that represents whether type checking is successful so far.

```
Map.foldlWithKey :: (a -> k -> b -> a) -> a -> Map k b -> a
```

So, at the top level, our solution will look like this:

```
typeCheck tenv formulas =
    let typeCheckHelper acc key value = ...
    in (Map.foldlWithKey typeCheckHelper True formulas)
```

Where the `typeCheckHelper` will check type check one formula, and combine the result with the existing accumulator.

Let's get some of the easy cases in the `typeCheckHelper` out of the way. First, if the accumulator is already `False`, then it really doesn't matter whether any other formulas type check! We already found one thing that fails. So we can use value-based pattern matching to handle that case:

```
typeCheck tenv formulas =
    let typeCheckHelper False _   _     = False
        typeCheckHelper True  key value = ...
    in (Map.foldlWithKey typeCheckHelper True formulas)
```

Now, the `value` here is either `Nothing`, or `(Just formula)`. We can again use pattern matching to handle these cases. In the case that we don't have a formula, then there is no formula to type check and we can automatically go on to the next formula:

```
typeCheck tenv formulas =
    let typeCheckHelper False _   _              = False
        typeCheckHelper True  key Nothing        = True
        typeCheckHelper True  key (Just formula) = ...
    in (Map.foldlWithKey typeCheckHelper True formulas)
```

In that last case, we will want to call `typeCheckFormula`. But first, we need to find the annotated type of `key` in `tenv`. Since `Map.lookup key tenv` returns the Haskell type `Maybe Type`, we need to do some more pattern matching:

```
typeCheck tenv formulas =
    let typeCheckHelper False _   _              = False
        typeCheckHelper True  key Nothing        = True
```

```
        typeCheckHelper True  key (Just formula) =
            case (Map.lookup key tenv) of
                Just ctype -> typeCheckFormula formula ctype tenv
                Nothing    -> False -- this shouldn't happen
    in (Map.foldlWithKey typeCheckHelper True formulas)
```

And we're done!

Map

One way is to use `Map.mapWithKeys` to check each formula. This function has the following type signature, meaning that its return value will be a Map of some kind. We would like the values in this returned map to be booleans representing whether type checking succeeded or not:

```
Map.mapWithKey :: (k -> a -> b) -> Map k a -> Map k b
```

We'll need a piece of code like this:

```
boolMap = Map.mapWithKey func formulas -- we still need to define "func".
```

The type signature of `func` needs to be

```
func :: String -> Formula -> Bool
```

Now, this function needs to call `typecheckFormula` somewhere inside. In order to call `typecheckFormula`, we need as arguments

1. the formula — which we almost have
2. the annotated return type in `tenv` – which we will need to look up.
3. the `tenv` itself — which means that `func` needs to be defined where `tenv` is in scope.

Here is an attempt that will *not* work:

```
WRONG_typeCheck tenv formulas =
    let func key formula = typeCheckFormula formula (Map.lookup key tenv) tenv
        boolMap = Map.mapWithKey func formulas
    in ...todo...
```

The reason is that `typeCheckFormula` expects an argument with the Haskell type `Type` as the second argument, not the Haskell type `Maybe Type`. The fix is a combination of pattern matching, like in the FAQ https://piazza.com/class/k0cr2ylpbxq3me?cid=374

```
typeCheck tenv formulas =
    let func key Nothing = True         -- there is no formula to type check
        func key (Just formula) =       -- there is a formula to check
            case (Map.lookup key tenv) of
                Just ctype -> typeCheckFormula formula ctype tenv
                Nothing    -> False     -- if the key doesn't exist in tenv
        boolMap = Map.mapWithKey func formulas
        boolValues = Map.elems boolMap -- extract the values
    in and boolValues                  -- check if all booleans are True
```