Exercise 9: Basic Type Checking

In lecture, we have seen how Haskell's static type checking detects and reports type errors *before* evaluating any code. In this exercise, you'll write a small type-checker in the same style as your interpreter from earlier in this course: a type-checker is yet another program that operates on an *abstract syntax tree* representation of source code. But while an interpreter takes an expression and calculates its *value*, a type-checker takes an expression and calculates its *type*.

As a meta note, this exercise is really the first step for your Assignment 2, in which you'll take what you do here an extend its functionality. Pay particular attention to the types we use to represent expressions and types, as this will come up over and over again in the next week few weeks.

Starter code

- Ex9.hs

Note about imports: while we've provided the minimum required imports to complete this exercise, because it's a bit more open-ended we're allowing you to add your own imports from the `Data.*` and `Control.*` libraries (the latter is quite advanced, so don't use it unless you're more experienced with Haskell).

Task 0: Understanding the types

As is typical with Haskell code, studying the types used in a program offers great insight into its design. In the starter code, read through the provided types and main type-checking functions. There's quite a bit to process here, since you'll need to understand how to represent both *expressions* and *types* in the code—take your time!

Note that we've done the three primitive cases in the `typeCheck` function for you; study them carefully as well, as they illustrate the different types and a use of Haskell pattern-matching you might want to take advantage of in your own code.

Task 1: Simple constraints, working recursively

Your first task is to implement `typeCheck` for the `If` constructor, enforcing the following constraints. Beside each constraint, we've written the corresponding error message variable that your code should use to report a violation of the constraint.

- The condition of an `If` must have type `Bool_`. (`errorIfCondition`)
- The two branches must have the same type. (`errorIfBranches`)

If these two constraints are satisfied, the returned type of the `If` expression is the same as the type of its branches.

Note that you are not evaluating the expressions! It doesn't matter whether the condition is true or false—the value returned by `typeCheck` represents the *type* of the branches, without knowing which branch is actually evaluated. This is why we (and Haskell) enforce the constraint that the two branches have the same type, unlike Racket.

Finally, note that since the three arguments to `If` are arbitrary `Expr` values, you should type-check each one recursively to determine its type. If any three of the type-checks fail (i.e., return a `Left`), then simply return that `Left` value, rather than checking the above two constraints. If more than one of these has an error, you may return any of the `Left`s. That is, you should **propagate type-**

check errors in a subexpression upwards; this is exactly analogous to a raised exception propagating up through the call stack, e.g. on Assignment 1. The difference here is we want you to experience doing it manually, for reasons we'll discuss in lecture. In the mean time, see if you can define a useful helper function to propagate `Left`s upwards without lots of code duplication.

Task 2: Function calls

Next, enforce the following constraints for `Call` expressions:

- The type of the first expression of the call is a `Function` type. (`errorCallNotAFunction`)
- The number of arguments in the call is equal to the number of parameters in the function type. (`errorCallWrongArgNumber`)
- The type of each argument is equal to the corresponding parameter type. (`errorCallWrongArgType`)

If these conditions are met, the type of the entire function call expression is the *return type* of the function.

As with `If`, if type-checking an argument subexpression fails, don't check the function call itself; instead, return the `Left` value corresponding to the type-check failure from the subexpression. If there are multiple type errors from a subexpression, return any one of them.

Also, another reminder that *defining key helper functions* is highly recommended to prevent your code from quickly becoming unreadable. Here's a hint of some potentially-helpful type signatures to get your imagination running:

```
(a -> Either String b) -> Either String a -> Either String b
(b -> a -> Either String b) -> b -> [a] -> Either String b
```

Task 3: Handling definitions

Your final task is to complete `buildTypeEnv`, which is analogous to `build-env` in a previous exercise. *Remember that you aren't evaluating expressions here, only determining their types*.

Identifiers can be used only *after* they have been defined, so that you can resolve all types with a single traversal of the list (e.g., with a single call to `foldl`). Report an `errorUnboundIdentifier` if you encounter an unbound identifier; note that definition order matters, so this error should be reported for an unbound identifier that is defined in a later definition.

You may assume that recursive definitions are not allowed (this complicates the type-checking).

Your implementation should handle the case when a definition's expression does not type-check. As you should expect, in this case the error should propagate upwards and be returned by `runTypeCheck`, and no subsequent expressions should be type-checked.

For course-related questions, please contact **david at cs.toronto.edu**.