1 Exercise

I started by creating a data type for interpreting *Stlc* terms that rightfully covers the type checking patterns of lambda-calculus.
The `evaluation` function takes a *Stlc* term, an integer (or gas) and returns a new type: `String | (Stlc, Int)`. To this effect, I instantiated the following data type:

```
data MaybeTerm = Message String | Term (Stlc, Int)
```

1.1 Type Checking

The type checking is done over the `eval` function that has the following type:

```
eval :: [Context] -> Stlc -> Maybe Type
```

The *typing contexts* Γ is captured as a list of CONTEXTs, where each of the latter corresponds to a character being associated with a TYPE.

```
data Context = Env Char Type
```

To implement the types described in the assignment, which are needed for typed abstractions, I defined the following data type:

```
data Type = TypeInt | TypeFloat | TypeUnit | TypeFunction Type Type
```

1.2 Reduction

The most valued function in *Stlc* reduction is the β-reduction, that tries to replace a variable x from an abstraction for the term under a TAPP.

1.3 Notes

To implement floating values, I added an *Stlc* term to refer to those: the TFLOAT constructor, which requires a value of type float. Adding (T-ADD) directly a TFLOAT to a TINT will cause an exception (*Maybe* type); however, I created additional type constructors, that allows the conversion between TFLOAT and TINT:

```
TFromFloat Stlc | TFromInt Stlc
```

The *call-by-value* for these terms use functions to allow conversion between types INT and FLOAT, that are then wrapped in either the TINT or TFLOAT constructor, respectively. For instance, the type checking for TFROMINT is as follows:

```
eval c (TFromInt term) =
    case eval c term of
        Just TypeInt -> Just TypeFloat
        _            -> Nothing
```

The most intriguing task was to understand how to use the *call-by-value* within CONTEXT. I was not quite sure on how T-APP and T-ADD would work on such situations.