# Advanced Programming
# Home Assignment 1

Carlo Rosso, Chinar Shah

## Contents

**1 Introduction** . . . . . **1**

**2 Task: Functions** . . . . . **2**

**3 Task: `try-catch`** . . . . . **2**

**4 Task: Pretty-printer** . . . . . **2**

**5 Answers to Questions** . . . . . **2**

## 1 Introduction

I have some doubts about my implementation the third tasks and I am not sure I completely understand the first question: what is the order of evaluation of the `Apply` subexpressions? And what are the variations you can implement to achieve different results?
Other than that, I am confident in the correctness of my solutions, also because I spent some time implementing tests for them. In particular, the test coverage is over 90%, which doesn't imply that the code I wrote is correct, but it is an indicator of the quality of the code.
Following, I describe how to run the tests:

1. unzipping `a1-handout.zip`;
2. entering the `a1-handout` directory through the terminal;
3. running `cabal test` in the terminal.

Note that you can run `cabal --enable-coverage test` to see the coverage of the codebase and you can also see what lines are not covered by the tests. Finally, I do not know what does "Top Level Definitions" mean in the coverage report.

## 2 Task: Functions

I have no doubt about the implementation of `eval` for the `Lambda` case, because you do not need to evaluate the body of the lambda, you only need to instantiate the lambda and so you convert the `Exp` to a `Value` wrapped in an `Either` monad.
I am also confident in the implementation of the `Apply` case, because I have tested it, not only with the provided examples, but I also created a test to check what happens when you apply a non-function to an expression and it return `Left "Non-function applied"`.

## 3 Task: `try-catch`

I think that this implementation is correct, because I have tested it with the provided examples and I also created two more tests.

## 4 Task: Pretty-printer

I think my implementation is correct, because I have tested it with throughly. The problem is that I may have not understood the question correctly so the tests may not guarantee the correctness of the implementation. Basically, I put parentheses around every expression, except for:

- `CstInt`;
- `CstBool`;
- `Var`.

## 5 Answers to Questions

**What is your observable evaluation order for `Apply`, what difference does it make, and which of your test(s) demonstrate this?**

I implemented the `Apply` case in the following way:

```
eval env (Apply fun e1) =
  do
    lam <- eval env fun
    arg <- eval env e1
```

```
    case lam of
      ValFun env' name body -> eval (envExtend name arg env') body
      _ -> Left "Non-function applied"
```

Follows the explanation of the evaluation order:

1. `fun` is evaluated first;
2. if `fun` is a `ValFun`, then `e1` is evaluated; otherwise it returns `Left "Non-function applied"`;
3. if the evaluation of `e1` is successful, then the lambda's environment is extended with the result of the evaluation of `e1`;
4. finally, the body of the lambda is evaluated in the extended environment.

I demonstrated this evaluation order: I test whether `fun` fails to evaluate before `e1` is evaluated; then I tested whether `e1` fails to evaluate after `fun` but before the lambda's body is evaluated.

I suppose that you can switch the order of evaluation of `fun` and `e1` and still adhere to the required `Apply` implementation.

**Would it be a correct implementation of `eval` for `TryCatch` to *first* call `eval` on *both* subexpressions? Why or why not? If not, under which assumptions might it be a correct implementation?**

Citing the request of task:

"[...] we start by evaluating the first expression. If it finishes successfully, then that is the result of the `TryCatch`. If it encounters a failure `(Left)`, then we evaluate the second expression [...]."

Since it is explicitly stated that we should evaluate the first expression first, and only evaluate the second expression if the first one fails, it would be incorrect to evaluate both subexpressions simultaneously. However, because Haskell is a lazy language, it is possible to write code that allows both subexpressions to be evaluated at the same time, as shown below:

```
eval env (TryCatch body except) =
  let result = eval env body
      result2 = eval env except
   in case result of
        Left _ -> result2
        _ -> result
```

In this case, we call `eval` on the `except` expression at the same time as the `result` expression, but Haskell is going to evaluate `except` only if `body` fails.
Finally, if `except` does not depend on `body`'s evaluation, it does not produce side effects and it terminates - then it is correct to evaluate both subexpressions at the same time in practice.

**Is it possible for your implementation of `eval` to go into an infinite loop for some inputs?**

Yes it is, for example the following expression goes into an infinite loop:

```
eval
  envEmpty
  ( Let
      "x"
      (Lambda "y" (Apply (Var "y") (Var "y")))
      (Apply (Var "x") (Var "x"))
  )
```

It is an infinite loop because we define a function that applies its argument to itself and then we apply this function to itself.