Exercise 03

Task 1: Taxonomy of Functions

1. What is the difference between first-order functions and higher-order functions? Give an example for a first-order function and for a higher-order function!

Answer: A first-order function is a function that *takes* a non-function as argument (e.g. an Int) and *returns* a non-function. Examples in Scala: +, -, String.length

A higher-order function is a function that *takes* a (first-class) function as argument or *returns* a (first-class) function. Examples in Scala: map, filter, fold on List.

2. What does it mean that a language has first-class functions? Does a language with first-class functions support higher-order functions?

Answer: A language that supports first-class functions treats functions as values (= first-class members). A first-class function is a function expression that is itself an expression in the language and that can, for example, be passed to another function as argument. That means that a language that supports first-class functions also supports higher-order functions.

Example for a first-class function in Scala: `val f = (x : Int) => x + 5`

3. Are there first-class/higher-order functions in F1LAE? Justify your answer to this question by pointing out relevant parts in the F1LAE interpreter code!

Answer: F1LAE contains neither first-class functions, nor higher-order functions: In the F1LAE language definition, FunDef does not extend F1LAE and hence a function definition is not by itself a valid expression in the language. Thus, it is also not possible to define a function in F1LAE that takes another function as argument or returns a function.

Task 2: F1LAE with immediate substitution - examples

The first interpreter for F1LAE that we saw used immediate substitution for evaluating let-expressions. Let us look at some example expressions to understand what that means.

1. Define the following mathematical formula “as is” in F1LAE (i.e., not using any let-expressions!):

$$f(x) + g(y), \text{ where } f(x) = x + c \text{ and } g(x) = x + 1 + c$$

Answer:

```
Map(
  'f -> FunDef('x, Add(Id('x), Id('c))),
  'g -> FunDef('x, Add(Id('x), Add(Num(1), Id('c))))
)

val expr = Add(App('f, Id('x)), App('g, Id('y)))
```

2. What happens if you directly execute this expression? What do you have to do in order to execute the expression with, e.g. x=5, y=10, c=2 ? Can you make the expression executable without modifying your original expressions for f(x) and g(x)? If not, what is the problem?

Answer: Direct execution of expr with fundefs results in an exception ("found unbound id 'x"). In order to execute expr with concrete values, we have to bind the identifiers using Let-expressions. For example, we may try:

```
interp(
    Let('x, Num(5), Let('y, Num(10), Let('c, Num(2), expr))),
    fundefs
)
```

However, in our F1LAE interpreter with immediate substitution, this results in an exception "found unbound id 'c". In order to make expr executable with these values, we would have to bind c directly within the definitions of the functions (twice, once in every function body!).

Task 3: Immediate substitution vs deferred substitution

For the language F1LAE, we introduced environments as an alternative for substitutions.

1. Why did we do this?

Answer: There are multiple answers to this question.

Answer 1: To reduce the complexity of the interpreter! If we have many let-expressions in our program, executing immediate substitution is really slow, because the interpreter has to traverse parts of the program multiple times in order to apply substitution. Also have a look to the beginning of Chapter 5 in the PLAI book for some answers (http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-04-26.pdf).

Answer 2: If we want to evaluate programs like the one from Task 2 with dynamic scoping, we have to be able to know about the substitution for a variable like c after fetching the function body from the function definition map.

2. Discuss the advantages and disadvantages of substitution versus environments.

Answer: Again, multiple answers are possible here. One could say that an interpreter with substitution is more intuitive, because in each step, it is clear what happens and which exact expressions will be evaluated. Also, when using immediate substitution, we don't need an extra data-structure (which could potentially get very large). However, as we have seen before, environments make our interpreter a lot more efficient in the presence of programs with a lot of let-expressions.

3. Come up with more F1LAE examples that notably behave differently when run with an interpreter that uses substitution versus environments.

Answer: The examples are all along the lines of task 2 and the example in the lecture slides: If you have a function definition with a free identifier that you try to bind *outside* of the function definition via a let, this will throw an error during substitution, but evaluate when using environments and dynamic scoping (assuming that the program expression does not contain any other errors!).

Task 4: Scoping

1. What is the difference between dynamic scoping and static (lexical) scoping?

Answer: In a language with dynamic scoping, the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect. In a language with static scoping, the scope of an identifier's binding is a *syntactically* delimited region. In the interpreter code, we can see the difference when interpreting App-nodes: For interpreting the function's body, we throw away the previous environment and only bind the arguments.

2. What does 'static' stand for in static scoping?

Answer: In static scoping, we can decide *statically* (by looking at the source text of an expression) whether an identifier is bound or not.

3. What are potential problems with dynamic scoping?

Answer: Again, multiple answers are possible. In general, dynamic scoping can result in unexpected behavior: Since free identifiers can be dynamically bound during program execution, it is not immediately clear before execution how or whether a free identifier in a function body is bound. If the programmer forgets to bind an identifier, the interpreter will dynamically fail.