SAMPLE EXAM

INFDEV02-8 - Advanced programming

Exam procedure

- The students begin with the theoretical part.
- For this part, the students can only use pen and paper.
- The theoretical part lasts 30 minutes.
- The students deliver their answers for the theoretical part on the official school paper when the time expires.
- After the theoretical part is over, the students can start with the practical part.
- The practical part lasts 120 minutes.
- In this part the students are allowed to use their laptop, but not the internet.
- The students can download the exam template from the github repository `Dev81718First` and then they must switch their connection off. After 5 minutes from the start all the students should have their connection disabled, unless specific permission is given by the supervisor.
- If a student is caught with a connection enabled, the supervisor stops his exam and reports him for suspicion of fraud.
- It is also not possible to use any help other than the exam template. If a student is caught using other material the same consequences apply.

Theoretical part

Instructions

- For this part you can only use your pen and paper. No computer is allowed. You are also not allowed to use any notes. The relevant formulas are given in the exam if needed.
- Your answers must be written on the official school paper. Everything that is not on the paper will not be graded.
- Each question awards you with 2 points. The final grade for this part is the sum of the points.

Beta-reduction rules:

Variables:

$$\frac{}{x \to_\beta x}$$

Function application:

$$\frac{}{(\lambda x \to t)\ u \to_\beta t[x \mapsto u]}$$

Application

$$\frac{t \to_\beta t' \wedge u \to u' \wedge t'\ u' \to_\beta v}{t\ u \to_\beta v}$$

Exercise 1:

Given the following untyped lambda-calculus expression:

$$(\lambda x\ y\ x \to ((y\ z)\ x))\ 5$$

replace the requested terms with the elements from the expression in the following lambda-calculus rule that evaluates it:

$$\frac{}{(\lambda x \to t)\ u \to_\beta t[x \mapsto u]}$$

$$\begin{cases} x = \dots \\ t = \dots \\ u = \dots \\ t[x \mapsto u] = \dots \end{cases}$$

Exercise 2:

Given the following untyped lambda-calculus expression:

$$(((\lambda x\ y \to x\ y)\ (\lambda z \to z + 1))\ 5)$$

replace the requested terms with the elements from the expression in the following lambda-calculus rule that evaluates it:

$$\frac{t \to_\beta t' \wedge u \to u' \wedge t'\ u' \to_\beta v}{t\ u \to_\beta v}$$

$$\begin{cases} t = \dots \\ u = \dots \\ t' = \dots \\ u' = \dots \\ v = \dots \end{cases}$$

Exercise 3:

Complete the missing types (denoted with ___) in the following code:

```
let boo = (x:  number) => (z :  number, y:  number):  string
let x:___= boo(8)
```

Exercise 4:

Complete the missing types (denoted with ___) in the following code:

```
let filter = <a, b>(p:  (x:  a) => bool) => (l:  List<a>) => List<a>
let x:___= filter((x:  number) => x > 5)
```

Exercise 5:

Complete the missing types (denoted with ___) in the following code:

```
type List<___> = { kind:___, value:___, next:List<___> } | { kind:___}
let fold = <___,___>(l:List<___>) => (f:(a:___) => (b:___) => ___) => (z:___)) :  ___=>  ...
let length = (l:List<number>):___=> fold (l) ((elem:___) => (state:___) => state + 1) (0)
```

Practical part

Instructions

- You can only use the data structures that are defined in the exam templates.
- You are not allowed to use the Internet except for downloading the exam.
- You cannot use the course materials during the exam.
- You cannot use ANY imperative statement except printing to the standard output. Imperative constructs include (but are not limited to) variables, loops, classes, records with mutable fields.
- You must copy your answers into the official school paper. Everything that is not on the paper will not be graded.
- Each exercise awards you with 2 points. The final grade for this part is the sum of the points.

Exercise 1:

Implement a function

```
let eval = (expr1:  Expr):  Expr
```

The function allows to evaluate arithmetic expressions. An arithmetic expression is defined as a discriminated union with the following cases:

- An atomic value. This is just a number
- The sum of two expressions.
- The difference of two expressions.
- The product of two expressions.
- The division of two expressions.

The function `eval` for an atomic value simply returns the atomic value. For the other cases it simply evaluates recursively the arguments of the operator and then combine the two results together. For example, evaluating the sum of two expressions `expr1 + expr2` recursively evaluates `expr1` and `expr2` and then sum ups the two atomic values together. The other operators behave analogously.

Exercise 2:

Implement a function

```
let mapFold = <a, b>(f:  (x:  a) => b) => (l:  List<a>):  List<b>
```

that implements `map` only using `fold`

Exercise 3:

Implement a function

```
let curry = <a, b, c>(f:  Tuple<a, b> => c) => (x:  a) => (y:  b):  c
```

that applies function ‘f‘ using as input elements ‘x‘ and ‘y‘ stored as a tuple.

Exercise 4:

Implement a function

```
let allEvenRange = (lower:  number) => (upper:  number):  string
```

that returns a string containing all even numbers between `lower` and `upper`. Separate the numbers with a white space.

Exercise 5:

Implement a function

```
let tryFind = <a>(value:  a) => (tree:  BinaryTree<a>):  Option<BinaryTree<a>>
```

that looks up for an element in a binary search tree. If the element is not found the function returns `None`.