Instituut voor Communicatie, Media & Informatietechnologie

Naam: .....

Studentnr: .....

Klas: .....

OPLEIDING	:	**Informatica**
SOORT TENTAMEN	:	**Herkansing Theorie/Praktijk**
VOLTIJD/DEELTIJD	:	**Voltijd**
CURSUSCODE	:	**INFDEV02-8**
PERIODE	:	**4**
GROEPEN	:	**INF2A-F, herkansers, BO**
TIJDSDUUR	:	**30 (Theorie) + 120 (Praktijk) min**
CURSUSHOUDER(S)	:	**F. Di Giacomo**
AUTEUR(S)	:	**F. Di Giacomo**
TWEEDE LEZER	:	**M. Abbadi**

DIT TENTAMEN BESTAAT UIT VOORBLAD MET **7** GENUMMERDE PAGINA'S

- ☐ __ MEERKEUZEVRAGEN
- ☐ __ CASUS
- ☒ __ OPEN VRAGEN
- ☐ DIGITALE TOETS

TOEGESTANE HULPMIDDELEN**:**

- ☒ woordenboek EN-NL en/of NL-EN
- ☐ boek
- ☐ eenvoudige rekenmachine (niet programmeerbaar)
- ☐ laptop
- ☐ ___
- ☐ ___

SCHRIJF JE ANTWOORDEN EN/OF BEREKENINGEN:

- ☒ op het uitgereikte uitwerkingenpapier, dus niet op het tentamen
- ☐ in het tentamen (zoals aangegeven)
- ☐ op het bijgeleverde antwoordformulier

BIJZONDERHEDEN**: Tentamen weer inleveren!**

SCHRIJF DUIDELIJK! Wat niet duidelijk leesbaar is, wordt niet beoordeeld.

Veel succes!

INFDEV02-8 - Advanced programming

Exam procedure

- The students begin with the theoretical part.
- 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.
- It is not possible to use any help other than what can be found in the exam. If a student is caught using other material, the student's exam is immediately stopped and he/she will be reported to the exam commission.

Theoretical part

Instructions

- For this part you are 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:

$$(\lambda f\ g \to f)\ (\lambda f\ g \to g)$$

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:

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

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. The dots denote missing code implementation **that is omitted for brevity and you do not have to complete**:

```
let foo (x :  string) (y :  int -> int) :  string = ...
let (f :  ___) = foo((fun (x :  int) -> string x) 5)
```

Exercise 4:

Complete the missing types (denoted with ___) in the following code. The dots denote missing code implementation **that is omitted for brevity and you do not have to complete**:

```
let fold (f :  'state -> 'a -> 'state) (init :  'state) (l :  List<'a>) :  'state = ...
let (x :  ___) = fold (fun (state :  string) (x :  string) -> state + x)
```

Exercise 5:

Complete the missing types (denoted with ___) in the following code. The dots denote missing code implementation **that is omitted for brevity and you do not have to complete**:

```
let map (f :  'a -> a1) (g :  'b -> 'b1) (e :  Either<'a,'b>) :  Either<'a1,'b1> = ...
let (foo :  string -> float) = ...
let (t :  ___) = map foo
```

Practical part

Instructions

- You can only use the data structures that are defined in the exam templates.
- 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 are not allowed to use library functions that provide an immediate answer to the question. For instance, if a question asks the implementation of `map2` you are not allowed to simply call the function `List.map2`, which is already built in the F# standard library.
- If the question contains a code template that you have to complete, you must follow the structure of the snippet. This means that you cannot just ignore it and write everything from scratch. The parts that you have to complete are marked with the comment `\\...`
- 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 allNumbersMod (n : int) (div : int) : string = ...`

that returns all numbers, concatenated in a string and separated by a space, between 0 and `n` that divisible by `div` (hint: use modulus).

Example: calling `allNumbersMod 10 3` returns `"0 3 6 9"`

Exercise 2:

Implement a function

`let evenOdd (l : List<int>) : List<string> = ...`

that returns a list containing the string `"even"` or `"odd"`, depending on whether the element of the list is even or odd.

Example: `evenOdd [3;5;4;5;6;7] = ["odd";"odd";"even";"odd";"even";"odd"]`

Template:

```
let rec evenOdd (l : List<int>) : List<string> = //...
```

Exercise 3:

Implement a function

`let overage (l : List<Person>) : List<Person> = ...`

Consider the record `Person` representing information about a citizen. Return a list containing people that are overage. An overage person is 18 years old or older.

Template:

```
type Person =
  {
    Name : string
    LastName : string
    Age : int
  }

let rec overage (l : List<Person>) : List<Person> = //...
```

Exercise 4:

Implement a function

`let functionChain (functions : List<FunctionWithConversion<'a,'b,'c>>) (input : 'a) : 'a = ...`

Consider a list of elements of type `FunctionWithConversion<'a,'b,'c>` as defined in the template below, containing the following elements:

- A function from a generic type `'a` to `Either<'b,'c>`.
- A conversion function that is able to convert from `'b` to `'a`.
- A conversion function that is able to convert from `'c` to `'a`.

The function `functionChain` starts by applying `input` to `Function` (having type `'a -> Either<'b,'c>`). If this function returns the left case of `Either` then it uses `Conversion1` function (having type `'b -> 'c`). Otherwise it uses `Conversion2` function (having type `'b -> 'c`). The procedure is then reapplied by passing as `input` the result of one of the two conversions until all the functions have been used. When all the functions have been applied, the result of the last conversion is returned.

Template:

```
type Either<'a,'b> =
| Left of 'a
| Right of 'b

type FunctionWithConversion<'a,'b,'c> =
  {
    Function : 'a -> Either<'b,'c>
    Conversion1 : 'b -> 'a
    Conversion2 : 'c -> 'a
  }


let rec functionChain
  (functions : List<FunctionWithConversion<'a,'b,'c>>) (input : 'a) : 'a =

  functions |>
  List.fold (fun value f -> //... ) input
```

Exercise 5:

Implement a function

`let treeFoldDepth (f : 'state -> 'a -> 'state) (state : 'state) (t : Tree<'a>): 'state = ...`

that first recursively applies `treeFoldDepth` to the subtrees of the current tree `t`, and then takes the resulting `state` and applies `f` to the root of the current tree `t`.

Example: `treeFoldDepth (fun s x -> s + (string x)) "" tree` where

```
let tree =
  Tree(3,
      [
        Tree(1,[Empty])
        Tree(2,
          [
            Tree(1,[Empty])
            Tree(2,[Empty])
            Tree(3,[Empty])
          ])
        Tree(5,
          [
            Tree(4,[Empty])
            Tree(6,[Empty])
```

```
            ])
        ]
```

returns `"112324653"`.

Template:

```
type Tree<'a> =
| Empty
| Tree of 'a * List<Tree<'a>>

let rec treeFoldDepth (f :  'state -> 'a -> 'state) (state : 'state) (t : Tree<'a>): '
    state = //...
```