Instituut voor Communicatie, Media & Informatietechnologie

Naam: ……………………………………..

Studentnr: ………………………………

Klas: ……………………..……………….

OPLEIDING	:	**Informatica**
SOORT TENTAMEN	:	**Regulier 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:

$$\frac{}{x \rightarrow_\beta x}$$

Function application:

$$\frac{}{(\lambda x \rightarrow t)\ u \rightarrow_\beta t[x \mapsto u]}$$

Application

$$\frac{t \rightarrow_\beta t' \wedge u \rightarrow u' \wedge t'\ u' \rightarrow_\beta v}{t\ u \rightarrow_\beta v}$$

Exercise 1:

Given the following untyped lambda-calculus expression:

$$(\lambda x\ y \rightarrow ((\lambda z \rightarrow z)\ x))\ 0$$

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

$$\frac{}{(\lambda x \rightarrow t)\ u \rightarrow_\beta t[x \mapsto u]}$$

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

Exercise 2:

Given the following untyped lambda-calculus expression:

$$((\lambda y\ x \rightarrow y\ x)\ O)\ ((\lambda x \rightarrow x)\ K)$$

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

$$\frac{t \rightarrow_\beta t' \wedge u \rightarrow u' \wedge t'\ u' \rightarrow_\beta v}{t\ u \rightarrow_\beta v}$$

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

Exercise 3:

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

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

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 map (f : 'a -> 'b) -> (l : List<'a>) : List<'b> = ...
let (x : ___) = map (fun (x : int) -> (string x) + "1")
```

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 curry (f : 'a * 'b -> 'c) : 'a -> 'b -> 'c = ...
let add (x : int, y : int) : int = ...
let (t : ___) = curry add 5
```

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.
- You must write your answers in the official school paper.

Exercise 1:

Implement a function

```
let lineGaps (length : int) (gap : int) : string = ...
```

that returns a line with `length` asterisks. A whitespace is inserted in place of an asterisk every `gap - 1` asterisks
Example: calling `lineGaps 7 3` returns `** ** *`

Exercise 2:

Implement a function

```
let concat (l1 : List<’a>) (l2 : List<’a>) : List<’a> = ... = ...
```

that combines all the element of `l2` after all the elements of `l1` in a single list. You cannot solve this exercise by using the `@` operator built in F#.

Exercise 3:

Implement a function

```
let uncompress (l : List<CompressedElements<’a>) : List<UncompressedElements<’a>> = ...
```

Consider a list of compressed elements defined as the polymorphic type:

```
 type CompressedElement<’a> =
| Corrupted
| Compressed of ’a * int
| Uncompressed of ’a
| Nested of List<CompressedElement<’a>>
```

where a `Corrupted` indicates an error in the compression, `Compressed` indicates that the item of type `’a` is repetead for a certain amount of time, `Uncompressed` is an element without repetition, and `Nested` is a nested compressed sequence. The function `uncompres` reads a compressed sequence and decompresses a compressed sequence into a sequence of uncompressed elements defined as follows:

```
 type UncompressedElement<’a>
| Element of ’a
| Error of string
```

The decompression works as follows:

- If the element is Corruputed then you generate an `Error` with an error message (you can choose freely what you want to display).

- If the element is `Compressed` then you have to repeat the `Element` in the decompressed sequence as many time as specificed in the `Compressed` element. For example `Compressed(’x’), 3` generates a sequence `Element(’x’),Element(’x’),Element(’x’)`.
- If the element is `Uncompressed` then you simply store its content in an `Element` once.
- If the element is `Nested` then you have to recursively decompress the sub-sequence and concatenate the result with the current decompressed sequence.

Example: the compressed sequence `[Compressed(’x’,2);Nested [Compressed(’y’,2);Compressed(’z’,2)];Uncompressed(’w’)]` is decompressed into `[Element(’x’);Element(’x’);Element(’y’);Element(’y’);Element(’z’);Element(’z’),Element(’w’)]`

Template:

```
type CompressedElement<’a> =
| Corrupted
| Compressed of ’a * int
| Uncompressed of ’a
| Nested of List<CompressedElement<’a>>

type UncompressedElement<’a> =
| Element of ’a
| Error of string

let rec uncompress (l : List<CompressedElement<’a>> ) : List<UncompressedElement<’a>> =
  let rec repeat (x : ’a) (n : int) : List<’a> =
    match n with
    | 0 -> //...
    | _ -> //...
  match l with
  | [] -> []
  | x :: xs ->
      match x with
      //...
```

Exercise 4:

Implement a function

`let mapChoice (l : List<’a>) (functions: List<Either<’a -> ’b,’a -> ’c>>) : Option<List<Either<’b,’c>>> = ...`

Consider a list of generic elements of type `’a` and the polymorphic type `Either<’a,’b>` that can contain either a value of type `’a` or of type `’b`:

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

The function takes as second parameter a list of `functions` to run for each element of `l`, which must be as long as `l`. If the length is different the function returns `None`. Each function can output a result of either type `’b` or `’c`. The function `mapChoice` returns a list with the results generated by running either a function of type `’a -> ’b` or a function of type `’b -> ’c`.

Template:

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

let mapChoice
  (l : List<’a>)
  (functions : List<Either<('a -> 'b),('a -> 'c)>>) : Option<List<Either<'b,'c>>> = //...
```

Exercise 5:

Implement a function

`let search (option : SearchOption) (fs : FileSystem) : List<File> = ...`

Consider the following type describing a file:

```
 type File = {
   Name : string
   Extension : string
}
```

and a file system. A file system is a tree where each node can be a directory containing files and subdirectories or an empty node:

```
 type FileSystem =
| Directory of List<File> * List<FileSystem>
| Empty
```

A file can be searched by name, or by extension. The search by name checks if a file name contains the string given as input, and if it is the case then this is returned. You can use the method `Contains` in order to implement this. For instance `"Hello World!".Contains("orl")` returns `true`. The search by extension only checks that the search parameter exactly matches the file extension. The following is the type defintion for the search options:

```
 type SearchOption=
| ByName of string
| ByExtension of string
```

The function `search` takes as input a file system and a search option and returns all the files that match the search parameter depending on the chosen search option.

Template:

```
type File =
  {
    Name : string
    Extension : string
  }
  with
    static member Create(name, extension) =
      {
        Name = name
        Extension = extension
      }

type FileSystem =
| Directory of List<File> * List<FileSystem>
| Empty

type SearchOption =
| ByName of string
| ByExtension of string

let rec search (option : SearchOption) (fs : FileSystem)  : List<File> = //...
```