CSC C24 Winter 2023 Final Examination

Duration: 3 hours

Aids allowed: none

Do ***not*** *turn this page until you have received the signal to start.*

(Please fill out the identification section above and read the instructions below.)

Good Luck!

- Please, make sure to **NOT write anything in the QR code areas**.
- Please, use a **black** or **blue pen** or a **thick in diameter pencil** to answer **all questions** in this booklet.
- Comments are **not** required except where indicated, although they may help us mark your answers.
- No error checking is required: assume all user input and all argument values are **valid**.
- Please, make sure to **indent** your code properly.
- If you use any space for rough work, indicate clearly what you want marked.

Question 1. [10 MARKS]

A Haskell compiler needs to process the following input:

```
42 : foo (x + 1) xs
```

Describe the first two steps in the translation process.

Part (a) [3 MARKS]

What is the result of lexical analysis of this input? Be as specific in your answer as you can.

Part (b) [2 MARKS]

What tools/concepts that we have studied in this course were involved in this stage of translation?

Part (c) [3 MARKS]

What is the result of the syntactic analysis that follows? Be as specific in your answer as you can.

Part (d) [2 marks]

What tools/concepts that we have studied in this course were involved in this stage of translation?

Question 2. [5 marks]

Consider the following two expressions:

1. In Python:

   ```
   len([42^1234, 42^2345, 42^3456])
   ```

2. In Haskell:

   ```
   length([42^1234, 42^2345, 42^3456])
   ```

What is the most significant difference in the way the two languages handle these expressions? Explain in detail.

Question 3. [10 MARKS]

Define a Haskell function `cycle` that takes a list `xs` as input and returns a list in which the elements of `xs` are repeated infinitely many times. For example,

```
take 10 (cycle [1,2,3])   should return   [1,2,3,1,2,3,1,2,3,1]
```

Show a step-by-step evaluation of the expression `take 3 (cycle [1, 2])`. The definitions of `take` and `++` (operator append) are provided for your convenience.

```
take 0 _ = []
take n (x:xs) = x : take (n - 1) xs
take _ xs = xs

[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Question 4. [25 MARKS]

In this question we review recursion, higher-order procedures `map` and `fold`, tail recursion, and continuation-passing style.

Consider the specification of function `deep-map` below:

```
;; (deep-map f xs) -> list?
;; f: unary function, applicable to every element of xs
;; xs: list?
;; returns the result of applying f to each element of xs, on any nesting level

(check-expect (deep-map abs '((-1 (-2 -3) ((-4 (-5)))))) '((1 (2 3) ((4 (5))))))
```

Part (a) [5 MARKS]

Implement `deep-map` recursively, not using any built-in higher-order procedures.

```
(define (deep-map f xs)
```

Part (b) [5 MARKS]

Re-implement `deep-map` using `map`.

Consider the specification of the function `filter-fold` below:

```
;; (filter-fold f g id xs) -> any
;; f: unary boolean-valued function applicable to every element of xs
;; g: binary function, as in foldr
;; id: any, identity as in foldr
;; xs: list?
;; returns the result of folding g onto those elements of xs, in order, on which f returns true

(check-expect (filter-fold positive? + 0 '(1 -2 3 -4 5 -6)) 9)
```

Part (c) [5 MARKS]

Provide a recursive implementation of `filter-fold`, not using any built-in higher order procedures.

```
(define (filter-fold f g id xs)
```

Part (d) [5 MARKS]

Re-implement `filter-fold` using a single call to `foldr`:

```
(define (filter-fold f g id xs)

  (foldr
```

Part (e) [5 MARKS]

Finally, provide a tail-recursive implementation of `filter-fold` that uses continuation-passing style:

```
(define (filter-fold f g id xs)

  (local [(define (ff xs k)
```

Question 5. [30 MARKS]

Recall the `Tree a` data type from our exercise and the function `tfold` defined to work with it:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)

tfold:: (a -> b) -> (a -> b -> b -> b) -> Tree a -> b
tfold f g (Leaf x) = f x
tfold f g (Node x left right) = g x (tfold f g left) (tfold f g right)
```

In this question we modify the data type `Tree`, and the corresponding function `tfold` as follows:

```
data Tree a = Tree a [Tree a]

tfold :: (a -> b) -> ([b] -> b) -> Tree a -> b
tfold f g (Tree x trees) = g (f x : map (tfold f g) trees)
```

You need to provide simple, short solutions that use higher order procedures in good Haskell style in order to earn full marks.

Part (a) [6 MARKS]

Implement the following functions **without** using `tfold`:

```
-- |countNodes t
-- return the number of all nodes in tree t
countNodes:: Tree a -> Int
```

```
-- |forallNodes p t
-- return whether p is true of every node in tree t
forallNodes:: (a -> Bool) -> Tree a -> Bool
```

Part (b) [6 MARKS]

Re-implement the functions using `tfold`:

```
-- |countNodes' t
-- return the number of all nodes in tree t
countNodes':: Tree a -> Int
```

```
-- |forallNodes' p t
-- return whether p is true of every node in tree t
forallNodes':: (a -> Bool) -> Tree a -> Bool
```

Part (c) [6 MARKS]

Your next task is to make `Tree a` an instance `Show`, such that a `Tree` is displayed as follows:

- the word ”Tree” does not appear
- values of type α appear based on `show` defined on that type
- empty lists become empty strings (not shown)
- non-empty lists are treated in the same way as built-in lists

For example:

```
> t1 = Tree 1 [Tree 2 [], Tree 3 [], Tree 4 []]
> t1
1 [2,3,4]
> t2 = Tree "one" [Tree "two" [Tree "three" [Tree "four" []]]]
> t2
"one" ["two" ["three" ["four"]]]
```

Part (c) (CONTINUED)

Part (d) [6 MARKS]

Your next task is to define the following function:

```
-- |collect t
-- return a list of values in t
collect :: Tree a -> [a]
```

Part (e) [6 MARKS]

Finally, we make `Tree a` an instance of `Eq`, such that two `Tree`s are equal iff they contain the same elements, in any order, and ignoring possible duplicate values. For example,

```
> t1' = Tree 2 [Tree 1 [], Tree 4 [Tree 3 [], Tree 2 []]]
> t1' == t1
True
> t3 = Tree 1 [Tree 2 [], Tree 3 []]
> t1 == t3
False
```

Question 6. [10 MARKS]

For each of the following definitions, specify the type of `func`. If you think Haskell's type inference algorithm would fail, write `ERROR`.

```
func (x,y,z) = x y z
```

```
func x = foldr x []
```

```
func x = x x
```

```
func x y _ = x:y
```

```
func = map
```

Question 7. [28 marks]

Consider the following Prolog program:

```
covers(csca48,[recursion]).
covers(cscb07,[java,polymorphism,types]).
covers(cscb36,[cfg,logic]).
covers(cscc24,[cfg,haskell,java,logic,polymorphism,prolog,recursion,types]).

took(anya,cscb36).
took(anya,cscc24).
```

Part (a) [3 marks]

Your first task is to implement the predicate `isCoveredIn/2`. Hint: use built-in `member/2`.

```
% isCoveredIn(?Topic,?Course) iff topic Topic is covered in course Course.
```

Part (b) [3 marks]

Next consider my attempt at implementing the predicate `studied/2`:

```
% studied(?Student,?Topic) iff student Student took a course that covers topic Topic.
studied(Student,Topic) :- took(Student,Course), isCoveredIn(Topic,Course).
```

List all answers, in order, to the following query:

```
?- studied(anya,What).
```

Part (c) [12 MARKS]

Your next task is to impement the following predicates:

```
% removeDups(+L1,?L2) iff list L2 contains the same elements as list L1 but with no duplicates.
```

```
% flatten(+L,?Flat) iff Flat is a flattened version of L
% For example, the following should all succeed:
% ?- flatten([1,2,3],[1,2,3]).
% ?- flatten([[[[1,2]]]], [1,2]).
% ?- flatten([1, [[2, [3]], 4], [[5, 6]]], [1,2,3,4,5,6]).
```

Part (d) [6 MARKS]

You are now ready to re-implement the predicate `studied/2` so that it does not produce duplicate answers. You may use the predicate `topics/1` below.

```
% topics(Topics) iff Topics is a list of all topics that appear in our database (in covers/2).
topics(Topics) :- findall(T, covers(_,T), Ts), flatten(Ts,Flat), removeDups(Flat,Topics).

% studied(?Student,?Topic) iff student Student took a course that covers topic Topic.
```

Part (e) [4 MARKS]

I translated my (original) definition of `studied/2` into first-order logic as follows:

$$\forall x, y \cdot took(x, z) \wedge isCoveredIn(y, z) \Rightarrow \forall x, y \cdot studied(x, y)$$

Explain what my mistake(s) is/are and provide a correct translation.

Question 8. Multiple Choice Questions [10 MARKS]

Use the bubble sheet on the last page to answer the following multiple choice questions.

1. Which of the following statements are true? Select all the apply.
   (a) All programming languages implement tail-call optimisation.
   (b) Tail-call optimisation makes a function run in linear time.
   (c) Tail-call optimisation is used to save runtime stack space.
   (d) Tail-call optimisation can only be used on recursive functions if they are tail recursive.
   (e) None of the above.

2. Select all that apply. A context free grammar that is used to define syntax of a programming language must:
   (a) be less than 5-7 pages long, so that it can be implemented effectively.
   (b) be unambiguous.
   (c) contain precedence and associativity rules for each function.
   (d) contain all required regular expressions.
   (e) None of the above.

3. Which of the following statements are true? Select all the apply.
   (a) Racket is a statically typed programming language.
   (b) Haskell is a statically typed programming language.
   (c) Prolog is a statically typed programming language.
   (d) Python is a statically typed programming language.
   (e) C is a statically typed programming language.

4. Which of the following statements are true? Select all the apply.
   (a) Static typing makes compiler optimisation easier.
   (b) Static typing makes tail-call optimisation possible.
   (c) Static typing saves runtime stack space by removing unnecessary stack frames.
   (d) Polymorphism is only possible with static typing.
   (e) Inheritance is only possible with static typing.

5. Which of the following statements are true? Select all the apply.
   (a) Python supports multiple inheritance.
   (b) Java supports multiple inheritance.
   (c) C++ supports multiple inheritance.
   (d) Haskell supports multiple inheritance.
   (e) None of the above.

Question 9. Bonus [5 MARKS]

1. Tell me three interesting and important things about Alan Turing.

2. Tell me three interesting and important things about Haskell Curry.

3. Tell me three interesting and important things about Donald Knuth.

4. Tell me three interesting and important things about Noam Chomsky.

5. Order the following in order (**from most to least**) of how much you learned from them this term: (a) lectures, (b) labs, (c) textbooks and other assigned readings, (d) office hours, (e) various non-assigned Internet sources.

Corrections and additions:

1. Question 1
   The input should be `42 : foo (x + 1) xs`
2. Question 7
   The only built-in predicates you are allowed to use are `member/2` and `append/3`.
3. Question 8
   "maked" should be "makes"

Built-ins you will find helpful:

- Haskell
  - `sum :: Num a => [a] -> a`
    The `sum` function computes the sum of a finite list of numbers.
  - `all :: (a -> Bool) -> [a] -> Bool`
    Applied to a predicate and a list, `all` determines if all elements of the list satisfy the predicate.
  - `and :: [Bool] -> Bool`
    `and` returns the conjunction of a Boolean list.
  - `elem :: Eq a => a -> [a] -> Bool`
    `elem` is the list membership predicate, usually written in infix form, e.g., ``x `elem` xs``.
- Prolog
  - `append(?List1, ?List2, ?List12)` succeeds if the concatenation of the list `List1` and the list `List2` is the list `List12`.
  - `member(?Element, ?List)` succeeds if `Element` belongs to the `List`.