Student Number: _____

Full Name: _____

Signature: _____

The University of New South Wales

COMP3141

Software System Design and Implementation

Sample Final Exam

Term 2, 2019

Time Allowed: 2 hours, excluding reading time.

Reading Time: 10 minutes.

Total Number of Questions: 4

Answer **all** questions.

The questions **are** of equal value.

You are permitted **two** hand-written, single-sided A4 sheets of notes.

Only write your answers on the provided booklets.

Answers must be written in ink, with the exception of diagrams.

Drawing instruments or rules may be used.

Excessively verbose answers may lose marks.

There is a 3% penalty if your name and student number are not filled in correctly.

Papers and written notes may not be retained by students.

Question 1 (25 marks)

Answer the following questions in a couple of short sentences. No need to be verbose.

(a) (3 marks) What is the difference between a *partial function* and *partial application*?

(b) (3 marks) Name *two* methods of measuring program coverage of a test suite, and explain how they differ.

(c) (3 marks) How are multi-argument functions typically modelled in Haskell?

(d) (3 marks) Is the type of *getChar* below a pure function? Why or why not?

$$getChar :: \texttt{IO Char}$$

(e) (3 marks) What is a *functional correctness* specification?

(f) (3 marks) Under what circumstances is performance important for an abstract model?

(g) (3 marks) What is the relevance of termination for the Curry-Howard correspondence?

(h) (4 marks) Imagine you are working on some price tracking software for some company stocks. You have already got a list of stocks to track pre-defined.

$$\textbf{data } Stock = \mathsf{GOOG} \mid \mathsf{MSFT} \mid \mathsf{APPL}$$
$$stocks = [\mathsf{GOOG}, \mathsf{MSFT}, \mathsf{APPL}]$$

Your software is required to produce regular reports of the stock prices of these companies. Your co-worker proposes modelling reports simply as a list of prices:

$$\textbf{type } Report = [Price]$$

Where each price in the list is the stock price of the company in the corresponding position of the *stocks* list. How is this approach potentially unsafe? What would be a safer representation?

Question 2 (25 marks)

The following questions pertain to the given Haskell code:

$$\begin{array}{lcll} foldr :: (a \to b \to b) \to b \to [a] \to b & & & \\ foldr\ f\ z\ (x : xs) & = & f\ x\ (foldr\ f\ z\ xs) & \texttt{-- (1)} \\ foldr\ f\ z\ [] & = & z & \texttt{-- (2)} \end{array}$$

(a) (3 marks) State the type, if one exists, of the expression $foldr\ (:)\ ([] :: [\mathsf{Bool}])$.

(b) (4 marks) Show the evaluation of $foldr\ (:)\ []\ [1, 2]$ via equational reasoning.

(c) (2 marks) In your own words, describe what the function $foldr\ (:)\ []$ does.

(d) (12 marks) We shall prove by induction on lists that, for all lists xs and ys:

$$foldr\ (:)\ xs\ ys = ys\ \texttt{++}\ xs$$

i. (3 marks) First show this for the base case where $ys = []$ using equational reasoning. You may assume the left identity property for ++, that is, for all ls:

$$ls = []\ \texttt{++}\ ls$$

ii. (9 marks) Next, we have the case where $ys = (k : ks)$ for some item k and list ks.

α) (3 marks) What is the *inductive hypothesis* about ks?

β) (6 marks) Using this inductive hypothesis, prove the above theorem for the inductive case using equational reasoning.

(e) (2 marks) What is the time complexity of the function call $foldr\ (:)\ []\ xs$, where xs is of size n?

(f) (2 marks) What is the time complexity of the function call $foldr\ (\lambda a\ as \to as\ \texttt{++}\ [a])\ []\ xs$, where xs is of size n?

Question 3 (25 marks)

A *sparse vector* is a vector where a lot of the values in the vector are zero. We represent a sparse vector as a list of position-value pairs, as well as an `Int` to represent the overall length of the vector:

$$\textbf{data}\ \mathit{SVec} = \mathsf{SV}\ \mathsf{Int}\ [(\mathsf{Int}, \mathsf{Float})]$$

We can convert a sparse vector back into a dense representation with this *expand* function:

$$\begin{array}{l} \mathit{expand} :: \mathit{SVec} \to [\mathsf{Float}] \\ \mathit{expand}\ (\mathsf{SV}\ n\ \mathit{vs}) = \mathit{map}\ \mathit{check}\ [0..n-1] \\ \quad \textbf{where} \\ \quad\quad \mathit{check}\ x = \textbf{case}\ \mathit{lookup}\ x\ \mathit{vs}\ \textbf{of} \\ \quad\quad\quad\quad\quad \mathsf{Nothing} \to 0 \\ \quad\quad\quad\quad\quad \mathsf{Just}\ v \to v \end{array}$$

For example, the *SVec* value $\mathsf{SV}\ 5\ [(0, 2.1), (4, 10.2)]$ is expanded into $[2.1, 0, 0, 0, 10.2]$

(a) (16 marks) There are a number of *SVec* values that do not correspond to a meaningful vector — they are invalid.

i. (6 marks) Which two *data invariants* must be maintained to ensure validity of an *SVec* value? Describe the invariants in informal English.

ii. (4 marks) Give two examples of *SVec* values which violate these invariants.

iii. (6 marks) Define a Haskell function $\mathit{wellformed} :: \mathit{SVec} \to \mathit{Bool}$ which returns `True` iff the data invariants hold for the input *SVec* value. Your Haskell doesn't have to be syntactically perfect, so long as the intention is clear. You may find the function $\mathit{nub} :: (\mathsf{Eq}\ a) \Rightarrow [a] \to [a]$ useful, which removes duplicates from a list.

(b) (9 marks) Here is a function to multiply a *SVec* vector by a scalar:

$$\begin{array}{l} \mathit{vsm} :: \mathit{SVec} \to \mathit{Float} \to \mathit{SVec} \\ \mathit{vsm}\ (\mathsf{SV}\ n\ \mathit{vs})\ s = \mathsf{SV}\ n\ (\mathit{map}\ (\lambda(p, v) \to (p, v * s))\ \mathit{vs}) \end{array}$$

i. (3 marks) Define a function *vsmA* that performs the same operation, but for dense vectors (i.e. lists of Float).

ii. (6 marks) Write a set of properties to specify *functional correctness* of the function *vsm*.

Hint: All the other functions you need to define the properties have already been mentioned on this page. It should maintain data invariants as well as refinement from the abstract model of dense vectors.

Question 4 (25 marks)

(a) (10 marks) Imagine you are working for a company that maintains this library for a database of personal records, about their customers, their staff, and their suppliers.

```
newtype Person = ...

name :: Person → String
salary :: Person → Maybe String
fire :: Person → IO ()
company :: Person → Maybe String
```

The *salary* function returns Nothing if given a person who is not a member of company staff. The *fire* function will also perform no-op unless the given person is a member of company staff. The *company* function will return Nothing unless the given person is a supplier.

Rewrite the above type signatures to enforce the distinction between the different types of person statically, within Haskell's type system. The function *name* must work with all kinds of people as input.

Hint: Attach *phantom* type parameters to the *Person* type.

(b) (15 marks) Consider the following two types in Haskell:

```
data List a where
  Nil   :: List a
  Cons  :: a → List a → List a

data Nat = Z | S Nat
data Vec (n :: Nat) a where
  VNil  :: Vec Z a
  VCons :: a → Vec n a → Vec (S n) a
```

i. (5 marks) What is the difference between these types? In which circumstances would *Vec* be the better choice, and in which *List*?

ii. (5 marks) Here is a simple list function:

```
zip :: List a → List b → List (a, b)
zip  Nil          ys          =  Nil
zip  xs           Nil         =  Nil
zip  (Cons x xs)  (Cons y ys) =  Cons (x, y) (zip xs ys)
```

Define a new version of *zip* which operates on *Vec* instead of *List* wherever possible. You can constrain the lengths of the input.

iii. (5 marks) Here is another list function:

```
filter :: (a → Bool) → List a → List a
filter p Nil = Nil
filter p (Cons x xs)
      | p x          =  Cons x (filter p xs)
      | otherwise    =  filter p xs
```

Define a new version of *filter* which operates on *Vec* instead of *List* wherever possible.

END OF EXAM