

THE AUSTRALIAN NATIONAL UNIVERSITY

First Semester 2006

COMP1100 (Introduction Programming and Algorithms)

Writing Period: 3 hours duration

Study Period: 15 minutes duration

Permitted Materials: One A4 sheet with hand-written notes on both sides.

Answer ALL questions. Do not spend too much time on any one question.

Total marks: 100

The questions are followed by labelled blank spaces into which your answers are to be written.

Additional answer panels are provided (at the end of the paper) should you wish to use more space for an answer than is provided in the associated labelled panels. If you use an additional panel, be sure to indicate clearly the question and part to which it is linked.

Name (family name first):

Student Number:

The following are for use by the examiners.

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Total

QUESTION 1 [8 marks]

Are each of the following valid Haskell expressions? If so, what are their values? If not, why are they invalid?

(a) `[1] : [[1, 2, 3, 4], [5], []]`

QUESTION 1(a)	[2 marks]
---------------	-----------

(b) `[1] ++ [[1, 2, 3, 4], [5], []]`

QUESTION 1(b)	[2 marks]
---------------	-----------

(c) `map sum [[1, 2, 3, 4], [5], []]`

QUESTION 1(c)	[2 marks]
---------------	-----------

(d) `length ["The", "quick", "brown", "fox"]`

QUESTION 1(d)	[2 marks]
---------------	-----------

QUESTION 2 [6 marks]

(a) The following function definition is missing its type signature.

```
check [] = False
check (b:bs) = if b
                then True
                else check bs
```

What is the type of check?

QUESTION 2(a)	[3 marks]

(b) The following two function definitions are missing their type signatures.

```
which (x:xs) = worker x xs
worker x [] = x
worker x (y:ys)
  | x > y      = worker y ys
  | otherwise = worker x ys
```

What are the types of the functions which and worker?

QUESTION 2(b)	[3 marks]

QUESTION 3 [12 marks]

An obvious way to represent a point in the xy plane is as a pair of Floats, being the cartesian coordinates of the point:

```
type Point = (Float,Float)
```

- (a) For each pair of points there is exactly one straight line passing through both points. The *gradient* of the line passing through (x_1, y_1) and (x_2, y_2) is given by the formula $(y_2 - y_1)/(x_2 - x_1)$.

Define a Haskell function

```
gradient :: Point -> Point -> Float
```

to calculate the gradient of the line passing through two points, using this formula.

QUESTION 3(a)	[4 marks]

- (b) Points are said to be *colinear* if they lie on the same straight line. A simple way to test whether 3 points p_1, p_2 and p_3 are colinear is to compare the gradients of the line passing through p_1, p_2 and the line passing through p_2, p_3 . If their gradients are the same, the points are colinear (since p_2 is on both lines).

Define a Haskell function

```
colinear :: Point -> Point -> Point -> Bool
```

which uses this technique to determine if 3 points are colinear. (For the purposes of this question, you may ignore issues of the imprecise representation of real numbers as Float.)

QUESTION 3(b)	[4 marks]

- (c) We can extend the technique used in the `colinear` function to check whether an arbitrary number of points fall on the same straight line. Define a Haskell function

```
allColinear :: [Point] -> Bool
```

which determines if *all* the points in the input list are colinear.

QUESTION 3(c)	[4 marks]
---------------	-----------

QUESTION 4 [9 marks]

The following functions have been disguised by giving them nonsensical names. Describe the behaviour of each function and suggest more meaningful names.

- (a)

```
 sponge :: Int -> [a] -> [a]
 sponge 0 xs      = xs
 sponge n []      = []
 sponge n (x:xs) = sponge (n-1) xs
```

QUESTION 4(a)

[3 marks]

- (b)

```
 muffin :: Int -> a -> [a]
 muffin n x
   | n < 0      = error "Not enough muffins."
   | n == 0     = []
   | otherwise  = x: muffin (n-1) x
```

QUESTION 4(b)

[3 marks]

(c) `bun :: Eq a => [a] -> [a]`
`bun [] = []`
`bun (x:xs) = x : bun (tart x xs)`
 where `mud :: Eq a => a -> a -> Bool`
 `mud x y = (x /= y)`
 `tart :: Eq a => a -> [a] -> [a]`
 `tart x ys = filter (mud x) ys`

QUESTION 4(c)

[3 marks]

- (d) In some card games (such as Bridge) players often assess how good their hand is before play commences. One aspect of this assessment is to calculate a number of points for the collection of court cards in the hand. Each *Ace* is worth 4 points, each *King* is worth 3 points, each *Queen* is 2 points, and each *Jack* is 1 point. The spot cards are worth no points.

Define a function `courtPoints :: Hand -> Int` which returns the total number of points for court cards in a hand.

QUESTION 5(d)

[4 marks]

- (e) Briefly outline how you might represent the abstract data types *Suit*, *Rank*, *Card* and *Hand* in Java.

QUESTION 5(e)

[4 marks]

QUESTION 6 [15 marks]

Suppose that we are developing a software package that is to play *Noughts and Crosses* (otherwise known as *Tic-Tac-Toe*) against a human sitting at a computer.

In the unlikely event that you do not already know this game, here is a brief description. The game is played on a 3×3 board by two players who take turns in putting a mark on the board. One player uses a cross (X) and the other player uses a nought (O). The winner is the first player to get three of their marks in a row, either vertically, horizontally or diagonally. If every space on the board is occupied and there is no winner, the game is a draw. For example, the following board configuration represents a win for noughts:

X		O
	X	X
O	O	O

During play, the human user of the software package indicates the position of their move and the computer responds with its own move, or by reporting that the game is over and who won, or if it is a draw.

- (a) There are numerous abstract data types appropriate to the development of this software package. Among the most obvious is the current state of play (i.e. the board configuration). The players' marks can be conveniently represented by the following algebraic data type:

```
data Player = Nought | Cross
```

The board consists of nine squares, each is either vacant or contains a player's mark:

```
data Square = Vacant | Occupied Player
```

Build on these types to define a representation of the board during play. Explain your representation, pointing out any limitations or constraints. How is the example board shown diagrammatically above represented as a value of this type?

QUESTION 6(a)

[5 marks]

- (b) One obvious basic operation on the board abstract data type is to add a player's mark at some specified position, provided it is vacant. Give a type declaration for this operation. Either define a Haskell function to implement the operation, or give a careful description of *how* you would implement it.

QUESTION 6(b)	[4 marks]
---------------	-----------

- (c) What other basic operations on the board can you identify? Give type declarations and brief descriptions but *do not* attempt to define them as Haskell functions.

QUESTION 6(c)	[6 marks]
---------------	-----------

QUESTION 7 [10 marks]

The standard functions `length` and `map` can be defined as follows:

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + length xs
```

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

(a) Use the above definitions to prove, using induction, the following identity:

$$\text{length } (\text{map } f \text{ } xs) = \text{length } xs$$

You first need to prove the base case:

QUESTION 7(a)	[1 mark]
---------------	----------

Now state what the step case is:

QUESTION 7(a)	[2 marks]
---------------	-----------

Lastly, prove this step case.

QUESTION 7(a)	[2 marks]
---------------	-----------

(b) The function `sum2L` performs pairwise addition of elements of two lists:

```
sum2L :: [Int] -> [Int] -> [Int]
sum2L []      ys      = []
sum2L xs     []      = []
sum2L (x:xs) (y:ys) = x+y : sum2L xs ys
```

Use this definition and that of `length` to prove the following predicate:

```
length (sum2L xs ys) <= (length (xs))
```

You first need to state and prove the base case.

QUESTION 7(b)	[2 marks]
---------------	-----------

Lastly, prove the step case.

QUESTION 7(b)	[3 marks]
---------------	-----------

QUESTION 8 [16 marks]

The calculation of grades in a particular course depends on a number of assessment items, including tutorial participation, assignment marks and exam marks. If we were writing a Java program to help with the collection and storage of marks and the calculation of grades, a `Student` class such as is shown on page 19 will be an important component.

The fields in the `Student` class represent the following information about a student:

- `String name` is the student's name;
- `ArrayList<Integer> attendance` is a sequence of marks for tutorial attendance. For each tutorial students are given a mark of 1 if they were present, or 0 if they were absent. There are 10 tutorials.
- `Integer assignment` is a mark (out of 20) for the student's assignment. The course only has one assignment.
- `Integer exam` is a mark (out of 70) for the student's final exam.
- `Integer total` is the student's total mark for the course. It is calculated by adding all 10 of their tutorial attendance marks, plus their assignment, plus their exam mark. The maximum total mark is therefore 100.
- `Grade grade` is the grade awarded to the student, based on their `total` mark.

The `Student` class has a constructor which sets the name, creates an empty `attendance` list, and initialises the other `Integer` fields to 0. There are methods provided to update tutorial attendance, assignment marks and exam marks. The code for calculating the student's total mark (as described above) and assigning a grade are missing.

- (a) Write the `public void calcTotal()` method, to set the `total` field to be the sum of the `attendance` marks, plus the `assignment` mark, plus the `exam` mark.

QUESTION 8(a)

[8 marks]

(b) Write the public void calcGrade() method, to set the Grade grade field, depending on the value of the total field.

If the total mark is 80 or more, the grade is HD. If the total mark is 70–79, the grade is D; 60–69 is CR; 50–59 is P; and less than 50 is N.

QUESTION 8(b)

[8 marks]

QUESTION 9 [10 marks]

- (a) In object-oriented programming languages such as Java, two of the fundamental constructs are *classes* and *objects*. What is a class? What is an object?

QUESTION 9(a)	[5 marks]
---------------	-----------

- (b) How are the concepts of *class* and *object* useful in designing programs?

QUESTION 9(b)	[5 marks]
---------------	-----------

QUESTION __ (__)

QUESTION __ (__)

QUESTION __ (__)

QUESTION __ (__)

[Java Code for Question 8]

```
import java.util.ArrayList;

enum Grade { N, P, CR, D, HD};

class Student {
    private String name;
    private ArrayList<Integer> attendance;
    private Integer assignment;
    private Integer exam;
    private Integer total;
    private Grade grade;

    public Student(String who) {
        name = who;
        attendance = new ArrayList<Integer>();
        assignment = 0;
        exam = 0;
        total = 0;
    }
    public String getName() {
        return name;
    }
    public Integer getAsst() {
        return assignment;
    }
    public Integer getExam() {
        return exam;
    }
    public Integer getTotal() {
        return total;
    }
    public Grade getGrade() {
        return grade;
    }
    public void assMark(Integer mark) {
        assignment = mark;
    }
    public void attend(Integer presence) {
        attendance.add(presence);
    }
    public void examMark(Integer mark) {
        exam = mark;
    }
    public void calcTotal() {
        // your code here
    }
    public void calcGrade() {
        // your code here
    }
}
```

