Module Code
COM00025I

BEng, BSc, MEng and MMath Degree Examinations 2020-21

Department Computer Science

Title Software 3 - Formative 2

Time Allowed 48 Hours

Note: This it is a formative assessment and submission is voluntary, and this assessment does not go towards your final grade for this module.

Time Recommended TWO hours

Word Limit `Not Applicable.`

Allocation of Marks:

This paper has one question and it is worth 40 marks.

Instructions:

Candidates should answer **all** questions using Haskell. Failing to do so will result in a mark of 0%. For every question, a template has been provided (*Q<qnumber><roman-numeral>[<partnumber>].hs*) which corresponds to the question number, and that is the **only** file to modify and submit. Each **.hs** file has the question description, declaration – implementation as undefined and a testing function.

You may use any values defined in the standard 'Prelude', but you may not import any other module unless explicitly permitted. You may use Neil Mitchell's $[Hoog\lambda e]$ to discover useful functions in the *Prelude*.

Where tests accompany the English description of the problem, it is not necesarily enough for your implementation to pass the tests to be correct. Partially functional solutions with *undefined*, may be considered for part of the full marks.

Download the paper and the required source files from the VLE, in the "Assessment" section. Once downloaded, unzip the file. You **must** save all your code in the `FormativeTwo` folder and the related *Q<qnumber><roman-numeral>[<partnumber>].hs* file provided. **Do not** save your code anywhere else other than this folder.

Submit your answers to the Department's Teaching Portal as a single **.zip** file containing the

`FormativeTwo` folder and its files. Failing to include the `FormativeTwo` folder will result in a mark of 0%. Other archival format such as .rar and .tar files will not be accepted.

If a question is unclear, answer the question as best as you can, and note the assumptions you have made to allow you to proceed. Please inform ⟨`cs-exams@york.ac.uk`⟩ about any suspected errors on the paper immediately after you submit.

A Note on Academic Integrity

We are treating this online examination as a time-limited open assessment, and you are therefore permitted to refer to written and online materials to aid you in your answers.

However, you must ensure that the work you submit is entirely your own, and for the whole time the assessment is live you must not:

- communicate with departmental staff on the topic of the assessment
- communicate with other students on the topic of this assessment
- seek assistance with the assignment from the academic and/or disability support services, such as the Writing and Language Skills Centre, Maths Skills Centre and/or Disability Services. (The only exception to this will be for those students who have been recommended an exam support worker in a Student Support Plan. If this applies to you, you are advised to contact Disability Services as soon as possible to discuss the necessary arrangements.)
- seek advice or contribution from any third party, including proofreaders, online fora, friends, or family members.

We expect, and trust, that all our students will seek to maintain the integrity of the assessment, and of their award, through ensuring that these instructions are strictly followed. Failure to adhere to these requirements will be considered a breach of the Academic Misconduct regulations, where the offences of plagiarism, breach/cheating, collusion and commissioning are relevant: see AM1.2.1 *(Note this supercedes Section 7.3 of the Guide to Assessment)*.

1 (40 marks) Extended example

This formative exercise asks you to implement components for playing: The Royal Game of Ur

The Royal Game of Ur is the oldest known board game. The oldest known board and pieces are around 4,500 years old. The only rule book we have, which is for versions of the game rather than the original game, is a youthful 2,500 years or so old. This rule book is written in cuneiform letters on a clay tablet, and currently resides in the British Museum. It was translated by one of the British Museum's curators, Irving Finkel. There are YouTube videos of him discussing and playing the game with Tom Scott (Tom Scott is a linguistics graduate from the University of York).

The rules are not completely known, but this exercise uses rules based on the Finkel-Scott match.

The game is a race game, such as ludo or backgammon for two players, that we will call *'Red'* and *'Green'*.

```
data Player = Red | Green deriving (Eq, Show)
```

(i) [2 marks] Implement a function that returns a player's opponent.

```
opponent :: Player -> Player
opponent = undefined
```

A board contains 14 *logical* squares, arranged as 20 *physical* squares.

```
data Square = Sq_1 | Sq_2 | Sq_3 | Sq_4
| Sq_5 | Sq_6 | Sq_7 | Sq_8 | Sq_9 | Sq10 | Sq11 | Sq12
| Sq13 | Sq14
deriving (Eq, Enum, Show)
```

The red player's pieces move along the top and middle rows, in numerical order, while the green player's pieces move similarly along the bottom and middle rows. Squares 1-4 and 13-14 are private, but squares 5-12 are *shared* and where, in Irving Finkel's words, the two players are "at war".

There are special squares: the private squares 4 and 14, and the shared square 8. On a real board these are decorated with a rosette, indicated here by an asterisk ('*').

Each player has seven (7) identical pieces. Each piece has a position:

- waiting to enter the board (at the *'Start'*),
- on a square on the board, or
- having reached *'Home'*.

```
data Position = Start | OnBoard Square | Home deriving (Eq, Show)
```

A game state consists of:

- a placing of pieces in positions, and
- the identity of the next player.

```
type Placing = () -- UNDEFINED TYPE
placing :: Placing -> Position -> Player -> Int
placing = undefined
data GameState = GameState Placing Player
```

(ii) [10 marks] Implement the type *'Placing'* and the function *'placing'*.

- When implementing the type you may change its declaration keyword to *'newtype'* or *'data'* if appropriate.
- You may derive type classes as part of the type definition, if *'newtype'* or *'data'*.
- The expression *'placing b p q'* should return the number of pieces player *'q'* has at position *'p'* in placing *'b'*.

Initially all of a player's pieces are at the *'Start'* position, waiting to enter the board. The first move is by the red player.

(iii) [2 marks] Implement the initial game state, *'initGS'*.

```
initGS :: GameState
initGS = undefined
```

A dice roll is, essentially, the number of heads in four fair coin tosses. This gives the following probabilities.

Roll	Probability
0	1/16
1	4/16 = 1/4
2	6/16 = 3/8
3	4/16 = 1/4
4	1/16

You are **not** asked to implement a dice; instead the value will be given by an external oracle.

A player may move any one of their pieces by the value of the dice roll, subject to many conditions.

A move of 'n' steps from position *'s'* by player *'p'* is valid if:

1. The number of steps is in the range 0 up to 4 inclusive.
2. Position *'s'* is not *'Home'*.
3. There is a piece belonging to the player in position *'s'*.
4. There is not already a piece belonging to 'p' in the new position, unless the new position is *'Home'*.
5. The new position is not the shared rosette occupied by the opponent.
6. A move "beyond" *'Home'* is valid (for example, a piece on Square 13 may be moved with rolls of 3 and 4, as well as 1 and 2).

(iv) [6 marks] Implement the function which, given a game state and a dice roll, returns a list of squares from which a move is valid.

```
possibleMoves :: GameState -> Int -> [Position]
possibleMoves = undefined
```

1. If there are valid moves with the current dice roll:
   a) The current player chooses one.
   b) The player's token is moved from the chosen position to the new position.
   c) If the new position is a shared square, and it is occupied by the other player then the other player's piece returns to the start.
   d) If the new position is "beyond" *'Home'*, it is taken to be *'Home'*. (For example, a piece on Square 13 may be moved to home with a roll of 2, 3 or 4.)
   e) The next player is the opponent, unless the new position is a rosette, in which case the current player has another roll.
2. If the dice roll has no valid moves, the next player is the opponent.

(v) [8 marks] Implement the function which, given a game state and a dice roll/position pair, returns the new game state.

```
move :: GameState -> (Int, Position) -> GameState
move = undefined
```

(vi) [4 marks] The game is over when one player gets all its tokens to "home".

We will model this by a function that, given a game state, returns

- *'Nothing'* when neither player has won, and
- *'Just p'* when player *'p'* has won (note: draws are not possible).

```
gameOver :: GameState                -> Maybe Player
gameOver = undefined
```

(vii) [8 marks] Given a list of dice roll/position pairs representing moves, implement a function, *'winner'*, that returns the winner, if there is one, of that sequence of moves.

```
winner :: [(Int, Position)] -> Maybe Player
winner = undefined
```

If you wish, you can load the file *'PlayRGU.hs'* into *'ghci'*. It imports your file, *'Formative2.hs'* and provides a function *'play'* that allows you to have an interactive match. You must provide your own dice.

WARNING the interface does no error checking, and will crash or otherwise behave badly, if you enter an unexpected value.

End of examination paper