COMP2209 Assignment Instructions

UNIVERSITY OF Southampton

Module:	*Programming III*			**Examiners:**	Julian Rathke, Nick Gibbins
Assignment:	*Haskell Programming Challenges*			**Effort:**	30 to 60 *hours*
Deadline:	16:00 on 11/1/2024	**Feedback:**	2/2/2024	**Weighting:**	40%

Learning Outcomes (LOs)

- Understand the concept of functional programming and be able to write programs in this style,
- Reason about evaluation mechanisms.

Introduction

This assignment asks you to tackle some functional programming challenges to further improve your functional programming skills. Four of these challenges are associated with interpreting, translating, analysing and parsing a variation of the lambda calculus. It is hoped these challenges will give you additional insights into the mechanisms used to implement functional languages, as well as practising some advanced functional programming techniques such as pattern matching over recursive data types, complex recursion, and the use of monads in parsing. Your solutions need not be much longer than a page or two, but more thought is required than with previous Haskell programming tasks you have worked on. There are three parts to this coursework and each of them has two challenges. Each part can be solved independently of the others and they are of varying difficulty, thus, it is recommended that you attempt them in the order that you find easiest.

For ease of comprehension, the examples in these instructions are given in a human readable format you may wish to code these as tests in Haskell. To assist with a semi-automated assessment of this coursework we will provide a file called Challenges.hs. This contains Haskell code with signatures for the methods that you are asked to develop and submit. You should edit and submit this file to incorporate the code you have developed as solutions. However, feel free to take advantage of Haskell development tools such as Stack or Cabal as you wish. You may and indeed should define auxiliary or helper functions to ensure your code is easy to read and understand. You must not, however, change the signatures of the functions that are exported for testing in Challenges.hs. Likewise, you may not add any **third-party** import statements, so that you may only import modules from the standard Haskell distribution. If you make such changes, your code may fail to compile on the server used for automatic marking, and you will lose a significant number of marks.

There will be no published test cases for this coursework beyond the simple examples given here - as part of the development we expect you to develop your own test cases and report on them. We will apply our own testing code as part of the marking process. To prevent anyone from gaining advantage from special case code, this test suite will only be published after all marking has been completed.

It is your responsibility to adhere to the instructions specifying the behaviour of each function, and your work will not receive full marks if you fail to do so. Your code will be tested only on values satisfying the assumptions stated in the description of each challenge, so you can implement any error handling you wish, including none at all. Where the specification allows more than one possible result, any such result will be accepted. When applying our tests for marking it is possible

your code will run out of space or time. A solution which fails to complete a test suite for one exercise within 15 seconds on the test server will be deemed to have failed that exercise and will only be eligible for partial credit. Any reasonably efficient solution should take significantly less time than this to terminate on the actual test data that will be supplied.

Depending on your proficiency with functional programming, the time required for you to implement and test your code is expected to be 5 to 10 hours per challenge. If you are spending much longer than this, you are advised to consult the teaching team for advice on coding practices.

Note that this assignment involves slightly more challenging programming compared to the previous functional programming exercises. You may benefit, therefore, from the following advice on debugging and testing Haskell code:

> https://wiki.haskell.org/Debugging
> https://www.quora.com/How-do-Haskell-programmers-debug
> http://book.realworldhaskell.org/read/testing-and-quality-assurance.html

It is possible you will find samples of code on the web providing similar behaviour to these challenges. Within reason, you may incorporate, adapt and extend such code in your own implementation. Warning: where you make use of code from elsewhere, you *must* acknowledge and cite the source(s) both in your code and in the bibliography of your report. Note also that you cannot expect to gain full credit for code you did not write yourself, and that it remains your responsibility to ensure the correctness of your solution with respect to these instructions.

The Challenges

Part I – Circuit Puzzles

In these two challenges we will introduce a type of circuit puzzle in which the solver is presented with a grid of "tiles" each with "wires" printed on them. The solver is then expected to rotate each tile in the grid so that all of the wires connect together to form a complete circuit. Moreover, each puzzle will contain at least one tile that is a "source" tile for the circuit, and at least one tile that is a "sink" tile. A completed circuit will ensure that every sink is reachable from a source and vice-versa. There may however be multiple sources and multiple sinks.

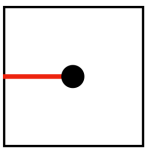
The grid may be of any rectangular size and will be given as a list of non-empty lists of Tile values. A Tile value is value of the data type given by:

```
data Edge = North | East | South | West  deriving (Eq,Ord,Show,Read)
data Tile = Source [ Edge ] | Sink [ Edge ] | Wire [ Edge ]  deriving (Eq,Show,Read)
type Puzzle = [ [ Tile ] ]
```

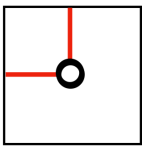
where a Tile simply lists which of its edges offer connection of wires. The Source and Sink tiles must contain at least one connector edge and Wire tiles must contain either zero (an empty Tile) or at least two connector edges. Duplicate entries in the edges list are ignored and order does not matter. Connector edges are considered to connect across two Tiles if they share a connector edge. For example, a Tile offering a West connector placed to the right of a Tile offering an East connector would have a connecting wire. A Wire Tile is connected if all of its connector edges are connected. Similarly Source and Sink tiles are connected if, all of their connector edges are connected.

Example tiles are as follows :

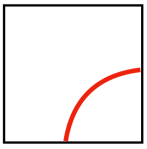
Source [West] could be represented visually as



Sink [North, West] could be represented visually as

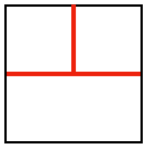


Wire [East, South] could be represented visually as



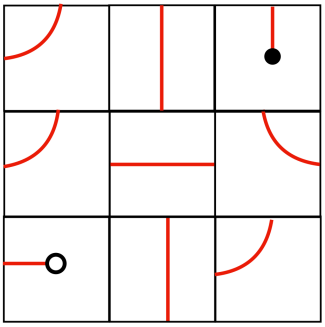
and finally

Wire [North, East , West] could be represented visually as

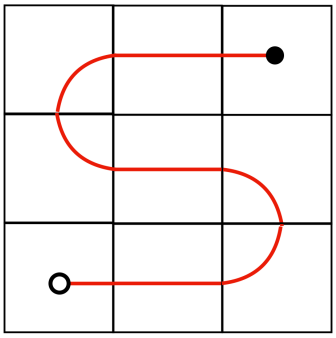


An example 3x3 puzzle is given below followed by a visual representation of the puzzle:

[[Wire [North,West] , Wire [North,South] , Source [North]],
[Wire [North,West], Wire [East,West], Wire [North,East]],
[Sink [West] , Wire [North,South] , Wire [North,West]]]



The following image shows a solution to the above puzzle obtained by rotating each of the Tiles. Note the completed circuit in the solution.



Challenge 1: Completedness of circuits.

The first challenge requires you to define a function

isPuzzleComplete :: Puzzle -> Bool

that, given a list of list of tiles, simply returns whether the puzzle is completed. That is, this function returns True if and only if all Tiles are connected, for every Source tile, there exists a path following the wires to at least one Sink tile and for every Sink tile, there is a path following the wires to at least one Source tile.

Challenge 2: Solve a Circuit Puzzle

This challenge requires you to define a function

```
solveCircuit :: Puzzle -> Maybe [ [ Rotation ] ]
```

where data Rotation = R0 | R90 | R180 | R270 deriving (Eq,Show,Read)

This function should, given a circuit puzzle, return Just of a grid of rotations such that, if the rotations were applied to the corresponding Tile in the input grid, the resulting Puzzle will be a completed circuit. Where this is not possible, the function should return the Nothing value.

The values of Rotation represent

- R0 no rotation
- R90 rotate Tile clockwise 90 degrees around the centre of the tile
- R180 rotate Tile clockwise 180 degrees around the centre of the tile
- R270 rotate Tile clockwise 270 degrees around the centre of the tile

For example,

solveCircuit [[Wire [North,West] , Wire [North,South] , Source [North]], [Wire [North,West], Wire [East,West], Wire [North,East]], [Sink [West] , Wire [North,South] , Wire [North,West]]]

could return

Just [[R180,R90,R270],[R90,R0,R180],[R180,R90,R0]]

note that this solution is not unique.

Part II – Parsing and Printing

You should start by reviewing the material on the lambda calculus given in the lectures. You may also review the Wikipedia article, https://en.wikipedia.org/wiki/Lambda_calculus, or Selinger's notes http://www.mscs.dal.ca/~selinger/papers/papers/lambdanotes.pdf, or both.

For the remaining challenges we will be working with a variant of the Lambda calculus that support let-blocks, discard binders and pairing. We call this variant Let_x and the BNF grammar for this language is as follows

```
Expr ::= Var | Expr Expr | "let" Eqn "in" Expr | "(" Expr ")"
        | "(" Expr "," Expr ")" | "fst" "("Expr")" | "snd" "("Expr")" | "\" VarList "->" Expr
Eqn ::= VarList "=" Expr
VarList ::= VarB | VarB VarList
VarB ::= "x" Digits | "_"
Var ::= "x" Digits
Digits ::= Digit | Digit Digits
Digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6 " | "7" | "8" | "9"
```

The syntax "let x1 x2 ... xN = e1 in e2" is syntactic sugar for "let x1 = \x2 -> ... -> \xN -> e1 in e2" and the syntax "\x1 x2 ... xN → e" is syntactic sugar for "\x1 -> \x2 -> ... -> xN -> e".
We can use the underscore "_" character to represent a discard binder that can be used in place of a variable where no binding is required. Pairing of expressions is represented as "(e1,e2)" and there is no pattern matching in this language so we use "fst" and "snd" to extract the respective components of a paired expression. For the purposes of this coursework we limit the use of variable names in the lambda calculus to those drawn from the set "x0 , x1, x2, x3, ... ", that is "x" followed by a natural number. An example expression in the language is

```
let x2  x3 _ =  x0 in fst ((x2 x4 x5 , x2 x5 x4))  snd ((x2 x4 x5 , x2 x5 x4))
```

Application binds tightly here and is left associative so "let x = e1 in e2 e3 e4" is to be understood as "let x = e1 in ((e2 e3) e4)".

Challenge 3: Pretty Printing a Let_x Expression

Consider the datatypes

```
data LExpr = Var Int | App LExpr LExpr | Let Bind  LExpr LExpr | Pair LExpr LExpr | Fst LExpr | Snd LExpr  | Abs Bind LExpr
  deriving (Eq,Show,Read)
data Bind = Discard | V Int
  deriving (Eq,Show,Read)
```

We use LExpr to represent Abstract Syntax Trees (AST) of the Let_x language.

This challenge requires you to write a function that takes the AST of a Let_x expression and to "pretty print" it by returning a string representation the expression. That is, define a function

prettyPrint :: LExpr -> String

that outputs a syntactically correct expression of Let_x. Your solution should omit brackets where these are not required and the output string should parse to the same abstract syntax tree as the given input. Finally, your solution should print using sugared syntax where possible. For example, an expression given as Let (V 1) (Abs (V 2) (Abs Discard e1)) e2 should be printed as "let x1 x2 _ = <e1> in <e2>" where e1 and e2 are expressions and <e1> and <e2> are their pretty print strings.

Beyond that you are free to format and lay out the expression as you choose in order to make it shorter or easier to read or both.

Example usages of pretty printing (showing the single \ escaped using \\) are:

App (Abs (V 1) (Var 1)) (Abs (V 1) (Var 1))	"(\\x1 -> x1) \\x1 -> x1"
Let Discard (Var 0) (Abs (V 1) (App (Var 1) (Abs (V 1) (Var 1))))	"let _ = x0 in \\x1 -> x1 \\x1 -> x1"
Abs (V 1) (Abs Discard (Abs (V 2) (App (Var 2) (Var 1))))	"\\x1 _ x2 -> x2 x1"
App (Var 2) (Abs (V 1) (Abs Discard (Var 1)))	"x2 \\x1 _ -> x1"

Challenge 4: Parsing Let_x Expressions

In this Challenge we will write a parser for the Let_x language using the datatype LExpr given above.

Your challenge is to define a function:

parseLetx :: String -> Maybe LExpr

that returns Nothing if the given string does not parse correctly according to the rules of the concrete grammar for Let_x and returns a valid Abstract Syntax Tree otherwise.
You are recommended to adapt the monadic parser examples published by Hutton and Meijer. You should start by following the COMP2209 lecture on Parsing, reading the monadic parser tutorial by Hutton in Chapter 13 of his Haskell textbook, and/or the on-line tutorial below:

http://www.cs.nott.ac.uk/~pszgmh/pearl.pdf *on-line tutorial*

Example usages of the parsing function are:

parseLetx "x1 (x2 x3)"	Just (App (Var 1) (App (Var 2) (Var 3)))
parseLetx "x1 x2 x3"	Just (App (App (Var 1) (Var 2)) (Var 3))
parseLetx "let x1 x3 = x2 in x1 x2"	Just (Let (V 1) (Abs (V 3) (Var 2)) (App (Var 1) (Var 2)))
parseLetx "let x1 _ x3 = x3 in \\x3 -> x1 x3 x3"	Just (Let (V 1) (Abs Discard (Abs (V 3) (Var 3))) (Abs (V 3) (App (App (Var 1) (Var 3)) (Var 3))))

Part III – Encoding Let_x in Lambda Calculus

It is well known that the Lambda Calculus can be used to encode many programming constructs. In particular, to encode a let blocks we simply use application

<let x0 = e1 in e2> is encoded as (\x0 -> <e2>) <e1> where <e1> and <e2> are the encodings of e1 and e2 respectively.

To encode the discard binder we simply need to choose a suitable variable with which to replace it:

<\ _ -> e1 > is encoded as (\xN -> <e1>) where xN is chosen so as to not interfere with <e1>

Finally, pairing can be encoded as follows:

< (e1 , e2)> is encoded as (\xN- > xN <e1> <e2>) where xN does not interfere with <e1> and <e2>

and

<fst e> is encoded as <e> (\x0 - > \x1 - > x0)
<snd e> is encoded as <e> (\x0 - > \x1 - > x1)

Challenge 5: Converting Arithmetic Expressions to Lambda Calculus

Given the datatype

```
data LamExpr = LamVar Int | LamApp LamExpr LamExpr | LamAbs Int LamExpr
   deriving (Eq,Show,Read)
```

Write a function

```
letEnc :: LExpr -> LamExpr
```

that translates an arithmetic expression in to a lambda calculus expression according to the above translation rules. The lambda expression returned by your function may use any naming of the bound variables provided the given expression is alpha-equivalent to the intended output.

Usage of the letEnc function on the examples show above is as follows:

letEnc (Let Discard (Abs (V 1) (Var 1)) (Abs (V 1) (Var 1))	LamApp (LamAbs 0 (LamAbs 2 (LamVar 2))) (LamAbs 2 (LamVar 2))
letEnc (Fst (Pair (Abs (V 1) (Var 1)) (Abs Discard (Var 2))))	LamApp (LamAbs 0 (LamApp (LamApp (LamVar 0) (LamAbs 2 (LamVar 2))) (LamAbs 0 (LamVar 2)))) (LamAbs 0 (LamAbs 1 (LamVar 0)))

Challenge 6: Counting and Comparing Let_x Reductions

For this challenge you will define functions to perform reduction of Let_x expressions. We will implement both a call-by-value and a call-by-name reduction strategy. A good starting point is to remind yourself of the definitions of call-by-value and call-by-name evaluation in Lecture 34 - Evaluation Strategies.

We are going to compare the differences between the lengths of reduction sequences to terminated for both call-by-value and call-by-name reduction for a given Let_x expression and the lambda expression obtained by converting the Let_x expression to a lambda expression as defined in Challenge 5. For the purposes of this coursework, we will consider an expression to have terminated for a given strategy if it simply has no further reduction steps according to that strategy. For example, blocked terms such as "x1 x2" are considered as terminated.

In order to understand evaluation in the language of Let_x expressions, we need to identify the redexes of that language. The relevant reduction rules are as follows:

- (\x -> e1) e2 -----> e1 [e2 / x]
- (_ -> e1) e2 -----> e1
- let _ = e1 in e2 -----> e2
- let x = e1 in e2 -----> e2 [e1 / x]
- fst (e1 , e2) -----> e1
- snd (e1 , e2) -----> e2

also note that, in the expressions "let x1 = e1 in e2" or "let _ = e1 in e2" the expression "e2" occurs underneath a binding operation and therefore, similarly to "\x1 -> e2", according to both call-by-value and call-by-name strategies, reduction in "e2" is suspended until the binder is resolved.

Define a function:

compareRedn :: LExpr -> Int -> (Int, Int , Int, Int)

that takes a Let_x expression and upper bound for the number of steps to be counted and returns a 4-tuple containing the length of four reduction sequences. In each case the number returned should be the minimum of the upper bound and the number of steps needed for the expression to terminate. Given an input Let_x expression E, the pair should contain lengths of reduction sequences for (in this order) :

1. termination using call-by-value reduction on E
2. termination using call-by-value reduction on the lambda calculus translation of E
3. termination using call-by-name reduction on E
4. termination using call-by-name reduction on the lambda calculus translation of E

Example usages of the compareRedn function are:

compareRedn (Let (V 3) (Pair (App (Abs (V 1) (App (Var 1) (Var 1))) (Abs (V 2) (Var 2))) (App (Abs (V 1) (App (Var 1) (Var 1))) (Abs (V 2) (Var 2)))) (Fst (Var 3))) 10	`(6,8,4,6)`

compareRedn (Let Discard (App (Abs (V 1) (Var 1)) (App (Abs (V 1) (Var 1)) (Abs (V 1) (Var 1)))) (Snd (Pair (App (Abs (V 1) (Var 1)) (Abs (V 1) (Var 1))) (Abs (V 1) (Var 1))))) 10	`(5,7,2,4)`
compareRedn (Let (V 2) (Let (V 1) (Abs (V 0) (App (Var 0) (Var 0))) (App (Var 1) (Var 1))) (Snd (Pair (Var 2) (Abs (V 1) (Var 1))))) 100	`(100,100,2,4)`

Implementation, Test File and Report

In addition to your solutions to these programming challenges, you are asked to submit an additional *Tests.hs* file with your own tests, and a report:

You are expected to test your code carefully before submitting it and we ask that you write a report on your development strategy. Your report should include an explanation of how you implemented **and** tested your solutions. Your report should be up to 1 page (400 words). Note that this report is not expected to explain how your code works, as this should be evident from your commented code itself. Instead you should cover the development and testing tools and techniques you used, and comment on their effectiveness.

Your report should include a second page with a bibliography listing the source(s) for any fragments of code written by other people that you have adapted or included directly in your submission.

Submission and Marking

Your Haskell solutions should be submitted as a single plain text file *Challenges.hs*. Your tests should also be submitted as a plain text file *Tests.hs.* Finally, your report should be submitted as a PDF file, *Report.pdf*.

The marking scheme is given in the appendix below. There are up to 5 marks for your solution to each of the programming challenges, up to 5 for your explanation of how you implemented and tested these, and up to 5 for your coding style. This gives a maximum of 40 marks for this assignment, which is worth 40% of the module.

Your solutions to these challenges will be subject to automated testing so it is important that you adhere to the type definitions and type signatures given in the supplied dummy code file Challenges.hs. **Do not change** the list of functions and types exported by this file. Your code will be run using a command line such as *ghc –e "main" CW2TestSuite.hs,* where CW2TestSuite.hs is **my** test harness that imports Challenge.hs. You should check before you submit that your solution compiles and runs as expected.
The supplied Parsing.hs file will be present so it is safe to import this and any library included in the standard Haskell distribution (Version 8.10.7). Third party libraries will not be present so do not import these. *We **will not** compile and execute your Tests.hs file when marking.*

Appendix: Marking Scheme

Grade	*Functional Correctness*	*Readability and Coding Style*	*Development & Testing*
Excellent 5 / 5	Your code passes all of our test cases; anomalous inputs are detected and handled appropriately	You have clearly mastered this programming language, libraries and paradigm; your code is very easy to read and understand; excellent coding style	Proficient use of a range of development & testing tools and techniques correctly and effectively to design, build and test your software
Very Good 4 / 5	Your code passes almost all of our test cases.	Very good use of the language and libraries; code is easy to understand with very good programming style, with only minor flaws	Very good use of a number of development & testing tools and techniques to design, build and test your software
Good 3 / 5	Your code passes some of our tests cases.	Good use of the language and libraries; code is mostly easy to understand with good programming style, some improvements possible	Good use of development & testing tools and techniques to design, build and test your software
Acceptable 2 / 5	Your code passes a few of our test cases; an acceptable / borderline attempt	Acceptable use of the language and libraries; programming style and readability are borderline	Adequate use of development & testing tools and techniques but not showing full professional competence
Poor 1 / 5	Your code compiles but does not run; you have attempted to code the required functionality	Poor use of the language and libraries; coding style and readability need significant improvement	Some use of development & testing tools and techniques but lacking professional competence
Inadequate 0 / 5	You have not submitted code which compiles and runs; not a serious attempt	Language and libraries have not been used properly; expected coding style is not used; code is difficult to read	Inadequate use of development tools and techniques; far from professional competence

Guidance on Coding Style and Readability

Authorship	You should include a comment in a standard format at the start of your code identifying you as the author, and stating that this is copyright of the University of Southampton. Where you include any fragments from another source, for example an on-line tutorial, you should identify where each of these starts and ends using a similar style of comment.
Comments	If any of your code is not self-documenting you should include an appropriate comment. Comments should be clear, concise and easy to understand and follow a common commenting convention. They should add to rather than repeat what is already clear from reading your code.
Variable and Function Names	Names in your code should be carefully chosen to be clear and concise. Consider adopting the naming conventions given in professional programming guidelines and adhering to these.
Ease of Understanding and Readability	It should be easy to read your program from top to bottom. This should be organised so that there is a logical sequence of functions. Declarations should be placed where it is clear where and why they are needed. Local definitions using *let* and *where* improve comprehensibility.
Logical clarity	Functions should be coherent and clear. If it is possible to improve the re-usability of your code by breaking a long block of code into smaller pieces, you should do so. On the other hand, if your code consists of blocks which are too small, you may be able to improve its clarity by combining some of these.
Maintainability	Ensure that your code can easily be maintained. Adopt a standard convention for the layout and format of your code so that it is clear where each statement and block begins and ends, and likewise each comment. Where the programming language provides a standard way to implement some feature, adopt this rather than a non-standard technique which is likely to be misunderstood and more difficult to maintain. Avoid “magic numbers” by using named constants for these instead.