UNIVERSITY OF Southampton

COMP2209 Assignment Instructions

Module:	*Programming III*			**Examiners:**	Julian Rathke, Jian Shi, Nick Gibbins
Assignment:	*Haskell Programming Challenges*			**Effort:**	30 to 60 *hours*
Deadline:	16:00 on 14/1/2021	**Feedback:**	5/2/2021	**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 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 although you will need to at least read Part II before attempting Part III. Thus, it is recommended that you attempted 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 second 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:

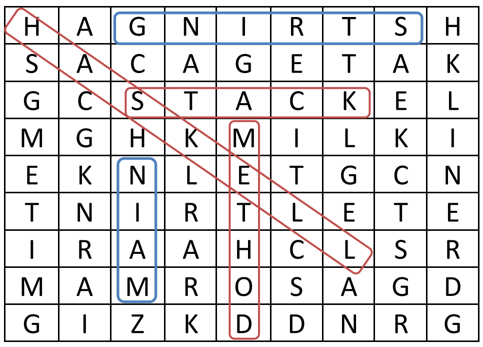
https://wiki.haskell.org/Debugging
https://www.quora.com/How-do-Haskell-programmers-debug
http://book.realworldhaskell.org/read/testing-and-quality-assurance.html

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 – Word Search Puzzles

A word search puzzle consists of an N x N non-empty square grid of characters hidden in which are a given number of words. Each word may appear in the grid as a sequence of characters in a straight line running vertically up or down, horizontally left or right, diagonally up-left or up-right or down-left or down-right. Here is an example word search puzzle with an indicated solution for the word list Haskell, String, Stack, Method, Main :



H	A	G	N	I	R	T	S	H
S	A	C	A	G	E	T	A	K
G	C	S	T	A	C	K	E	L
M	G	H	K	M	I	L	K	I
E	K	N	L	E	T	G	C	N
T	N	I	R	T	L	E	T	E
I	R	A	A	H	C	L	S	R
M	A	M	R	O	S	A	G	D
G	I	Z	K	D	D	N	R	G

This part requires you to write functions for both creating and solving word search puzzles. You will not be required to provide a graphical front-end for the puzzles as we will limit ourselves to manipulating back-end representations of them. We are going to make use of the following type synonyms:

```
type WordSearchGrid  = [ [ Char ] ]
```

representing a 2D grid of letters

```
type Placement = (Posn,Orientation)
```

representing a placement of a word hidden within the grid

```
type Posn = (Int,Int)
```

representing a (col,row)-coordinate within the 2D grid (rows and column numbers begin at zero) and

```
data Orientation = Forward | Back | Up | Down | UpForward | UpBack | DownForward | DownBack
                                                                     deriving (Eq,Ord,Show,Read)
```

representing the direction in which the word is hidden in the grid.

Challenge 1: Solve a Puzzle

The first challenge requires you to define a function

```
solveWordSearch :: [ String ] -> WordSearchGrid -> [ (String, Maybe Placement) ]
```

that, given a list of strings and a puzzle grid, returns a list of the placements of the given strings within the grid where such a placement exists. If a given string does not appear in the grid then the Nothing value of the Maybe type should be returned as its placement, otherwise the correct position and orientation of the string within the grid must be returned. You may assume that the given strings do not appear more than once in the given puzzle grid. For the example puzzle given above the function should return:

```
[("HASKELL",Just((0,0),DownForward)),("STRING",Just((7,0),Back)),("STACK",
Just((2,2),Forward)),("MAIN",Just((2,7),Up)),("METHOD",Just((4,3),Down))]
```

Challenge 2: Create a Puzzle

This challenge requires you to define a function

```
createWordSearch :: [ String ] -> Double -> IO WordSearchGrid
```

that, given a list of words, and a value representing the maximum **density** of the puzzle, returns a grid of characters in which all of the given words are hidden exactly once. Your function should choose random positions and orientations for **all** of the words hidden in the grid and you should ensure that the other characters in the grid are drawn randomly only from the set of characters appearing in the given input list. The density of a grid refers to the ratio of the number of characters within the grid that form the hidden words to the total number of characters within the grid. Therefore, the density of a grid is always between 0 and 1. The output grid must be sufficiently large so that the density of the output grid is less than the given maximum density but no larger.

N.B. for certain lists of input strings it may not be possible to meet the above requirements (e.g. if there are too few characters to draw from to guarantee uniqueness within the grid). For this challenge you may assume that this is not the case for the given inputs.

Part II – Parsing and Printing a Lambda Calculus with Macros

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.

As lambda terms can be difficult to read it is common practice to write lambda terms using an informal simple macro notation such as

λx -> λy -> G (F y x) y   where G is λx -> λy -> x and F is λx -> λy -> y

In this part we will be working with an extended version of the lambda calculus that formally supports such macros. We extend the abstract syntax of lambda calculus to macro expressions as follows

ME ::= def X = E in ME | E
E :: = x | X | *λx -> E | E E*

where x is drawn from a countably infinite set of variable names and X is drawn from a distinct countably infinite set of macro names. The example above would be written as:

def G = λx -> λy -> x in def F = λx -> λy -> y in λx -> λy -> G (F y x) y

Beta reduction in this extended lambda calculus is exactly as in the pure lambda calculus (that is)

(λx -> E) E' ----> E [E' / x]   (beta reduction)

but we also have a further form of reduction representing macro expansion:

def X = E in ME ----> ME [E / X]   (macro expansion)

in which all occurrences of X in ME are replaced by their corresponding definition E. Thus, using a leftmost innermost strategy the term above would reduce as follows:

def G = λx -> λy -> x in def F = λx -> λy -> y in λx -> λy -> G (F y x) y ----> (macro expansion)

def G = λx -> λy -> x in λx -> λy -> G ((λx -> λy -> y) y x) y ----> (beta reduction)

def G = λx -> λy -> x in λx -> λy -> G ((λy -> y) x) y ----> (beta reduction)

def G = λx -> λy -> x in λx -> λy -> G x y ----> (macro expansion)

λx -> λy -> (λx -> λy -> x) x y ----> (beta reduction)

λx -> λy -> (λy -> x) y ----> (beta reduction)

λx -> λy -> x

You will be using the following data types for the abstract syntax of this notation:

```
data LamMacroExpr = LamDef  [ (String , LamExpr) ]   LamExpr  deriving (Eq,Show,Read)
data LamExpr =  LamMacro String |  LamApp LamExpr LamExpr  |
                       LamAbs Int LamExpr  |  LamVar Int    deriving (Eq, Show, Read)
```

It is assumed here that each variable is represented as an integer using some numbering scheme. For example, it could be required that each variable is written using the letter *x* immediately followed by a natural number, such as for example *x0*, *x1*, and *x456*. These are then represented in the abstract syntax as *Var 0*, *Var 1*, and *Var 456* respectively. Likewise, for example, the lambda expression *λx1 -> x2 x1* is represented as *LamAbs* 1 (*LamApp (Var* 2*) (Var* 1)). Representing variables using integers rather than strings makes it is easier to generate a fresh variable that has not been already used elsewhere. Macro names however will be represented using a string of uppercase characters.

Challenge 3: Pretty Printing a Lambda Expression with Macros

You are asked to write a "pretty printer" to display a simple lambda expression with macros. That is, define a function

prettyPrint :: LamMacroExpr -> String

that un-parses a lambda expression with macros to a string. Use a "\" symbol (suitably escaped) rather than *λ and "->" to delimit the body within an abstraction* Your output should produce a syntactically correct expression. In addition, your solution should omit brackets where these are not required. That is to say, omit brackets when the resulting string parses to the same abstract syntax tree as the given input. Finally, your function should recognise sub-expressions within the given lambda expression that are syntactically equal to those defined in a macro. In such cases, print the corresponding macro name rather than the lambda expression. Where two such sub-expressions overlap, the larger of the two should be rendered as a macro name. Where two macro names are defined to be the same expression the one defined first in the term should be used.

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 are:

LamDef [] (LamApp (LamAbs 1 (LamVar 1)) (LamAbs 1 (LamVar 1)))	"(\x1 -> x1) \x1 -> x1"
LamDef [] (LamAbs 1 (LamApp (LamVar 1) (LamAbs 1 (LamVar 1))))	"\x1 -> x1 \x1 -> x1"
LamDef [("F", LamAbs 1 (LamVar 1))] (LamAbs 2 (LamApp (LamVar 2) (LamMacro "F")))	"def F = \x1-> x1 in \x2 -> x2 F"
LamDef [("F", LamAbs 1 (LamVar 1))] (LamAbs 2 (LamApp (LamAbs 1 (LamVar 1)) (LamVar 2)))	"def F = \x1-> x1 in \x2-> F x2"

Challenge 4: Parsing a Lambda Expression with Macros

You are asked to write a parser for lambda expressions with macros using the concrete syntax as follows:

MacroExpr ::= "def" MacroName "=" Expr "in" MacroExpr | Expr
Expr ::= *Var* | MacroName | *Expr Expr* | "\" *Var* "->" *Expr* | "(" *Expr* ")"
MacroName ::= UChar | UChar MacroName
UChar ::= "A" | "B" | ... | "Z"
Var ::= "x" *Digits*
Digits ::= *Digit* | *Digit Digits*
Digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

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*

Your challenge is to define a function:

```
parseLamMacro :: String -> Maybe LamMacroExpr
```

that returns Nothing if the given string does not parse correctly according to the rules of the grammar and a valid Abstract Syntax Tree otherwise. Moreover, your parser should enforce the following properties:

1. Macros are uniquely defined within a term - i.e. "def F = E1 in def F = E2 in E3" is not a well-formed expression
2. Macros are defined only with closed terms - i.e. for "def F = E1 in E2" then E1 has no free variables.

If either of these properties fail for the given input then the parser should return Nothing.

Example usages of the parsing function are:

"x1 (x2 x3)"	Just (LamDef [] (LamApp (LamVar 1) (LamApp (LamVar 2) (LamVar 3))))
"x1 x2 F"	Just (LamDef [] (LamApp (LamApp (LamVar 1) (LamVar 2)) (LamMacro "F")))
"def F = \x1-> x1 in \x2 -> x2 F"	Just (LamDef [("F", LamAbs 1 (LamVar 1))] (LamAbs 2 (LamApp (LamVar 2) (LamMacro "F"))))
"def F = \x1 -> x1 (def G = \x1 -> x1 in x1) in \x2 -> x2"	Nothing - *not in grammar*
"def F = \x1 -> x1 in def F = \x2 -> x2 x1 in x1"	Nothing - *repeated macro definition*
"def F = x1 in F"	Nothing - *macro body not closed*

Part III – Lambda Calculus and Continuation-Passing Style (CPS)

Continuation-Passing Style (CPS) is a form of writing lambda calculus terms in which the control of evaluation is encoded within a term as a **continuation** and these are passed explicitly within functions. Think of a continuation as a function that represents the next bit of computation to perform. You can read about this style in many online tutorials and a good starting point is the Wikipedia page at: https://en.wikipedia.org/wiki/Continuation-passing_style

A common translation <E> of a lambda calculus term to another lambda calculus term in continuation passing style is as follows:

$$< x > = \lambda\kappa.(\kappa\, x)$$

$$< \lambda x.E > = \lambda\kappa.(\kappa\, \lambda x. < E >)$$

$$< (E1\ E2) > = \lambda\kappa.(< E1 > \lambda f.(< E2 > \lambda e.(f\, e\, \kappa)))$$

For this part we will be extending this translation to the lambda calculus extended with macros as described in Part II and providing an implementation of this translation.

Challenge 5: Converting CPS Lambda Calculus to Standard Lambda Expression

We can easily extend the above translation to the lambda calculus with macros language using

$$< \text{def } X = E \text{ in } ME) > = \text{def } X = < E > \text{ in } < ME > \text{ and}$$

$$<X> = X$$

Write a function

```
cpsTransform :: LamMacroExpr -> LamMacroExpr
```

that translates a lambda expression with macros in to one in continuation-passing style according to the above translation rules. Take care not to capture any free variables in the given expression during translation.

Example usages of the cpsTransform conversion function are (pretty printed):

x1 x2	\x3 -> (\x6 -> x6 x1) (\x4 -> (\x7 -> x7 x2) (\x5 -> x4 x5 x3))
def F = \x1 -> x1 in x2	def F = \x3 -> x3 (\x1 -> (\x4 -> x4 x1)) in (\x5 -> x5 x2)
def F = \x1 -> x1 in F	def F = \x2 -> x2 (\x1 -> (\x3 -> x3 x1)) in F
def F = \x1 -> x1 in F F	def F = \x2 -> x2 (\x1 -> (\x3 -> x3 x1)) in \x4 -> F (\x5 -> F (\x6 -> x5 x6 x4)

Note that your output may use different bound variable names than shown above.

Challenge 6: Counting and Comparing Direct Lambda Calculus Reductions and CPS

For this challenge you will define functions to perform reduction of a lambda expressions with macros for both innermost and outermost reduction strategies. We will only use leftmost strategies throughout this challenge. A good starting point is to remind yourself of the definitions of innermost and outermost evaluation in Lecture 34 - Evaluation Strategies.

We say that a lambda expression with macros has terminated if it is an expression without any redexes. The famous Church-Rosser theorem (for lambda calculus) states that for different evaluation strategies any two such terminated values will be the same, or at least alpha equivalent, lambda expressions so they may differ only in how long a reduction sequence takes to reach a terminated value. Note however that lambda reduction may in fact not terminate. We are going to compare the differences between the lengths of reduction sequences to terminated values for innermost and outermost reduction for a given lambda expression with macros. We are also going to compare the same lengths for the same expression when it is converted to continuation-passing style. You may assume that there are no undefined macro usages within the given lambda expressions but you should not assume that the lambda expressions are closed.
Define functions:

```
innerRedn1 :: LamMacroExpr -> Maybe LamMacroExpr
outerRedn1 :: LamMacroExpr -> Maybe LamMacroExpr
```

that take a lambda expression with macros and perform a single reduction on that expression, if possible, by returning the reduced expression. The functions should implement the innermost and outermost reduction strategies respectively.

Define a function

```
compareInnerOuter :: LamMacroExpr -> Int -> ( Maybe Int, Maybe Int, Maybe Int, Maybe Int )
```

that takes a lambda expression with macros and a positive integer bound and returns a 4-tuple containing the length of different reduction sequences up to a given maximum length. For each reduction strategy the number returned should be the number of steps needed for the expression to terminate. If the expression does not terminate within the given bound (i.e. the number of reduction steps is strictly greater than the bound) then a Nothing value should be returned. Given an input expression E, the 4-tuple should contain lengths for:

1. Innermost reduction on E
2. Outermost reduction on E
3. Innermost reduction on the CPS translation of E applied to the identity: < E > (λx -> x)
4. Outermost reduction on the CPS translation of E applied to the identity: < E> (λx -> x)

in that order. We abuse notation here to mean def X = <E1> in (<E2> (λx -> x)) whenever we write <def X = E1 in E2 > (λx -> x).

Example usages of the compareInnerOuter function (pretty printed) are:

(\x1 -> x1 x2) using bound 10	(Just 0, Just 0, Just 6, Just 6)
def F = \x1 -> x1 in F using bound 10	(Just 1, Just 1,Just 3,Just 3)
(\x1 -> x1) (\x2 -> x2) using bound 10	(Just 1,Just 1,Just 8,Just 8)
(\x1 -> x1 x1)(\x1 -> x1 x1) using bound 100	(Nothing,Nothing,Nothing,Nothing)
def ID = \x1 -> x1 in def FST = (\x1 -> λx2 -> x1) in FST x3 (ID x4) using bound 30	(Just 4,Just 4,Just 22,Just 22)
def FST = (\x1 -> λx2 -> x1) in FST x3 ((\x1 ->x1) x4)) using bound 30	(Just 4,Just 3,Just 21,Just 21)
def ID = \x1 -> x1 in def SND = (\x1 -> λx2 -> x2) in SND ((\x1 -> x1 x1) (\x1 -> x1 x1)) ID using bound 1000	(Nothing,Just 4,Nothing,Nothing)

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 that includes a main function.* 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 stack *ghc –e "main" CW2Tests.hs,* where CW2Tests.hs 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 7.6.3). Third party libraries will not be present so do not import these. Your own test cases in the submitted file Tests.hs will also be run using a similar command line, so make sure this file defines a suitable *main* function that runs your tests and reports the results.

Appendix: Marking Scheme

Grade	*Functional Correctness*	*Readability and Coding Style*	*Development & Testing*
Excellent 5 / 5	Your code passes the supplied test cases and all the additional tests we ran; 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 the supplied test cases and almost all the additional tests we ran	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 the supplied test cases and some of the additional tests we ran	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 some of the supplied 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.