COMP2209 Assignment Instructions

UNIVERSITY OF Southampton

Module:	*Programming III*			**Examiner:**	*amg@ecs*
Assignment:	*Haskell Programming Challenges*			**Effort:**	30 to 55 *hours*
Deadline:	16:00 on 13/12/2018	**Feedback:**	10/01/2019	**Weighting:**	35%

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 associated with interpreting, translating, analysing and parsing variations of the lambda calculus, a language which is known to be Turing complete[1]. 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 those you wrote for the previous exercises, but it is likely more thought is required. Each challenge is independent so that if you find one of them difficult you can move on to the next one.

To assist with your implementation a file is provided with the sample test cases given in these instructions so that you do not need to re-type or cut and paste these. This imports your *Challenges.hs* code, and a dummy version of this file is also provided which you can edit to incorporate the code you have developed. 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 which are exported for testing, nor their defined types. Likewise, you may not add any third party Import statements, hence you may only Import modules from the standard Haskell distribution. As well as the published test cases an additional test suite will be applied to your code during marking. To prevent anyone from gaining advantage from special case code, this second test suite will only be published after 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 exercise, 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 the tests 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 30 seconds on the ECS login 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.

[1] Church showed how to encode integer arithmetic and Booleans as pure lambda expressions; combinators which enable recursive definitions to be encoded were invented by Turing and Curry, among others.

Depending on your proficiency with functional programming, the time required for you to implement and test each solution 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.

Lambda Calculus and Let Expressions

You should start by reviewing the material on the lambda calculus given in lectures 12, 13 and 14. 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.

The simplest lambda notation uses a syntax such as *λx -> e* where *x* is a variable and *e* another lambda expression. In this notation, the lambda expression has only one formal argument or parameter. Application of a lambda function is written as in Haskell, however, with the function followed by its actual argument. Lambda expressions may be nested, for example: *λx -> λy -> e.*

The let notation can also be used to define a function without using any Greek letter or arrow, for example *let f x = e,* and *let f x y = e*. A let expression can also be applied, for example *let f x = e in y.* We will only consider the latter of these in this assignment.

The let expressions used in this assignment have the following concrete syntax (expressed in BNF):

> *Expr* ::= *Var* | *Expr Expr* | "let" *VarList* "=" *Expr* "in" *Expr* | (*Expr*)
> *VarList* ::= *Var* | *Var VarList*
> *Var* ::= "x" *Digits*
> *Digits* ::= *Digit* | *Digits Digit*
> *Digit* ::= "0" | "1" | … | "9"

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

```
   data Expr = App Expr Expr  |  Let [Int] Expr Expr |  Var Int  deriving (Show, Eq)
```

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, *let x1 = x2 in x1* is then represented as *Let [1] (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.

It is also assumed in these challenges that application associates to the left and binds tighter than any other operator in each of the notations used here.

Functional Programming Challenges

Challenge 1: Converting a Let Expression to Lambda Calculus

The first challenge requires you to convert a let expression into an equivalent lambda calculus expression using, for example, the translation given below. The lambda expression will be represented using the following data type:

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

which represent an application, a lambda abstraction, and a variable such as *x1* respectively.

Conversion constructs a lambda abstraction and applies this as described below.

First consider a simple case *let x1 = e in d*, where *e* and *d* are also let expressions. Here the required result is *d′* but with any free occurrence of *x1* replaced by *e′*, where *d′* is the lambda expression resulting from converting *d*, and *e′* is the lambda expression resulting from converting e. This substitution is expressed in lambda calculus using the abstraction and application *(λx1 -> d′) e′*.

Now consider a more complex let expression such as *let x1 x2 = e in d.* Here *x1* is a function with parameter *x2* whose body is the let expression *e* (or after conversion, the lambda expression *e′*). Hence the required result is the lambda expression *d′* but with *x1* replaced by the abstraction *λx2 -> e′*. The required lambda expression is therefore *(λx1 -> d′) (λx2 -> e′).* And so on for let expressions with 2, 3 or more arguments.

(Note that there are other let conversion rules, which give different results in some cases[2] so be sure to implement the version given here. This version is based on Dana Scott's simple let expression. Specifically, for *let x1 = e in d,* then variable *x1* is bound in *d* but is not bound in *e*.

Here are some examples of let conversions:

let x1 = x1 in x2	*(λx1 -> x2) x1*
let x1 x2 = x2 in x1	*(λx1 -> x1) (λx2 -> x2)*
let x1 x2 x3 = x3 x2 in x1 x4	*(λx1 -> x1 x4) (λx2 -> λx3 -> x3 x2)*
let x1 = x2 in let x3 = x4 in x1 x3	*(λx1 -> (λx3 -> x1 x3) x4) x2*

Challenge 2: Pretty Printing a Let Expression

You are asked to write a "pretty printer" to display a simple let expression. Your output should produce a syntactically correct let expression. In addition, your solution should omit brackets where these are not required. That is to say, you can omit brackets when the resulting string represents the same abstract expression as the original one. 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.

[2] Haskell let expressions are recursive rather than simple. This means that evaluation of the Haskell expression *let a = a in a* does not terminate, and that *let zs = 'z':zs in take 5 zs* results in the string *"zzzzz"*.

Some examples of pretty printing are:

Let [1] (Var 2) (Var 1)	"let x1 = x2 in x1"
Let [1,2] (Var 2) (App (Var 3) (Var 1))	"let x1 x2 = x2 in x3 x1"
App (Let [1,2] (Var 2) (Var 3)) (Var 1)	"(let x1 x2 = x2 in x3) x1"
App (Var 1) (App (Var 2) (Var 3))	“x1 (x2 x3)”
App (App (Var 1) (Var 2)) (Var 3)	“x1 x2 x3”

Challenge 3: Parsing Let Expressions

You are asked to write a parser for simple let expressions. You should do this by adapting the monadic parser examples published by Hutton and Meijer. You should start by reading the monadic parser tutorial by Hutton in Chapter 13 of his Haskell textbook, or the on-line tutorial below:

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

Your parser will have the type signature String -> Maybe Expr and will return Nothing if the given string does not parse correctly according to the rules of the grammar. Note that you will need to transform the grammar into an equivalent form in order to avoid left recursion, which results in a non-terminating parser. Some examples of parsing are:

"let x1 = x2"	Nothing -- *syntactically invalid*
"let x1 = x2 in x1"	Just (Let [1] (Var 2) (Var 1))
"let x1 x2 = x2 in x1 x1"	Just (Let [1,2] (Var 2) (App (Var 1) (Var 1)))
"x1 (x2 x3)"	Just (App (Var 1) (App (Var 2) (Var 3)))
"x1 x2 x3"	Just (App (App (Var 1) (Var 2)) (Var 3))

Challenge 4: Counting Lambda Calculus Reductions

For this challenge, you will program beta reduction of a lambda expression using two different strategies. The first one repeatedly takes the innermost leftmost reducible expression (redex) and applies beta conversion to that. The second repeatedly takes the innermost rightmost redex and applies beta conversion to that. Note that in the lambda calculus, reduction under lambda is allowed. Your *countReds* function should take a lambda calculus expression and return a pair of values. The first component of this pair gives the number of reduction steps required before the first strategy converges, and the number of reduction steps required before the second strategy converges. Your function has a second parameter which limits the maximum number of reductions to attempt. If strictly more than this number are required by one strategy or other, the *countReds* function returns *Nothing* instead of a number for the corresponding component of the pair.

For example, the lambda expression whose concrete syntax is shown below has two reducible expressions, which are indicated by the underlined text above:

(λx -> λy -> x) z ((λt -> t) u)

This expression beta reduces using innermost leftmost reduction as follows:

(λx -> λy -> x) z ((λt -> t) u)

(λy -> z) ((λt -> t) u)

z

This expression reduces using innermost rightmost reduction as follows:

(λx -> λy -> x) z ((λt -> t) u))

(λx -> λy -> x) z u

(λy -> z) u

z

As the first sequence has 2 reductions, and the second has 3, *countReds* returns (Just 2, Just 3) in this case, provided the limit is 3 or more. Some examples of this function are:

Lambda expression	*Limit*	*Result*
λx -> (λy -> y)	0	(Just 0, Just 0)
(λx -> x)(λy -> y)	1	(Just 1, Just 1)
(λx -> λy -> x) z ((λt -> t) u)	10	(Just 2, Just 3)
(λx -> λy -> x) z ((λt -> t) u)	2	(Just 2, Nothing) -- *limit exceeded for rightmost*
(λx -> λy -> x) z ((λt -> t) u)	1	(Nothing, Nothing) -- *limit exceeded for both*

Challenge 5: Compiling Arithmetic Expressions to Lambda Calculus

As noted above, the lambda calculus can encode arithmetic. For this challenge, you are asked to write a function which translates or compiles an expression of natural number arithmetic to an equivalent lambda calculus expression. For example, it should compile "0" to the lambda calculus expression that encodes zero, "1" to the lambda calculus expression that encodes one, and "(+ 1)" to the lambda calculus expression that encodes the successor function.

The concrete syntax of these expressions is given in BNF as:

ArithmeticExpression ::= *Value* | *Section*
Section ::= "(" "+" *Value* ")"
Value ::= Section Value | *Natural* | *Value* "+" *Value* | (*Value*)
Natural ::= *Digit* | *Natural Digit*
Digit ::= "0" | "1" | … | "9"

The same operator precedence and associativity applies as in Haskell. You may use monadic parsing, recursive descent parsing, or any other means of parsing the input string. You are recommended to parse this string into some abstract data type, or types, before then translating this to a lambda calculus expression. (Although it is possible to parse and compile at the same time, this approach is usually more difficult to implement and maintain.)

It is required the lambda expressions your compiler generates will conform to the usual rules of arithmetic, but using beta equivalence as the "equality" relation. For example, compiling "(+ 1)" should result in a lambda expression which when applied to the result of compiling "2" returns a lambda expression which is beta equivalent to the result of compiling "3". You are advised to use the Church encoding described in the Wikipedia article cited above, and indicated in the table below, although others exist.

Some possible examples are as follows. Note that your solutions may differ from these depending on your chosen encoding but they ought to be beta-equivalent.

"0++"	Nothing -- *syntactically invalid*
"0"	Just (λx -> (λy -> y))
"1"	Just (λx -> (λy -> x y))
"2"	Just (λx -> (λy -> x x y))
"(+1)"	Just ((λx -> (λy -> x y)) (λx -> λy -> λz -> y (x y z)))

The final result can be simplified, although you are not required to implement this simplification.

Implementation and Test Report

In addition to your solutions to these programming challenges, you are asked to submit a report. The report should include an explanation of how you implemented and tested your solutions, which 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 code itself together with any comments you have included where necessary and appropriate. Instead you should cover the development tools and techniques you used, and comment on their effectiveness.

You are expected to test your code carefully before submitting it. Please include any test functions and constants you have written in with your *Challenges.hs* file, and mention these in your report.

Your report should include a second page with a bibliography listing the source(s) for any fragments of Haskell 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*. When you have put together your submission, please ensure that it compiles and runs satisfactorily using the supplied test cases (even if these do not all pass).

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 35 marks for this assignment, which is worth 35% of the module.

Your solutions to these challenges will be subjected to automated testing, so it is important you adhere to the type definitions and type signatures given in the supplied dummy code file, and that you do not change the list of functions and types exported by this file.

Andy Gravell, November 2018

Appendix: Marking Scheme

Grade	*Functional Correctness*	*Readability and Coding Style*	*Development Process*
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	You used a range of development 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	You made very good use of a number of development 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 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 tools and techniques but not showing 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	Poor use of development tools and techniques 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.
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.