UNIVERSITY OF Southampton

COMP2209 Assignment Instructions

Module:	*Programming III*			**Examiner:**	*amg@ecs*
Assignment:	*Haskell Programming Challenges*			**Effort:**	30 to 60 *hours*
Deadline:	16:00 on 05/12/2019	**Feedback:**	07/01/2020	**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 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 a page or two, but more thought is required than with previous programming tasks you have worked on. Each challenge is independent so that if you find one of them difficult you can move on to the next one.

For ease of comprehension, the examples in these instructions are given in a human readable format. To assist with your implementation a file called *Challenges2019tests.hs* is provided with the sample test cases but in machine readable format so that you do not need to re-type them. This test file imports the *Challenges.hs* code you are asked to develop and submit. A dummy version of *Challenges.hs* is also provided which you should 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 from *Challenges.hs* for testing, nor their defined types. 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.

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 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 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 test server will be deemed to have failed that exercise, and will only be eligible for partial credit. Any

[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.

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.

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 here the same way as in Haskell, with the function followed by its actual argument, which is also known as juxtaposition. Lambda expressions may be nested, for example: *λx -> λy -> e.*

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

```
  data LamExpr =
     LamApp LamExpr LamExpr  |  LamAbs Int LamExpr  |  LamVar 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, 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.

The let notation can also be used to define a function but without the need for any Greek letter or arrow. For example *let f x = e,* and *let f x y = e* are both let expressions that define a function f. A let expression can also be applied, for example *let f x = e in f y.* Multiple let expressions can also be defined simultaneously by separating them with a semi-colon, for example *let f x = e; y = d in f y*. To make this notation seem more concrete, numeric literals are allowed in these expressions, albeit these will also interpreted as just another shorthand for a lambda expression.

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

> *Expr* ::= *Num* | *Var* | *Fun* | *Expr Expr* | "let" *EqnList* "in" *Expr* | "(" *Expr* ")"
> *EqnList ::= Eqn* | *EqnList* ";" *Eqn*
> *Eqn* ::= *Fun VarList* "=" *Expr*
> *Fun* ::= "f" *Digits*
> *VarList* ::= *Var* | *Var VarList*
> *Var* ::= "x" *Digits*
> *Num* ::= *Digits*
> *Digits* ::= *Digit* | *Digits Digit*
> *Digit* ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

As with the lambda expressions above, variable and function names in these let expressions are represented numerically (x0, x1, x2, … and f0, f1, f2, … respectively) to simplify processing.

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

```
  data LetExpr =
     LetApp LetExpr LetExpr  |  LetDef [([Int], LetExpr)] LetExpr |  LetFun Int | LetVar Int |
        LetNum Int deriving (Show, Eq)
```

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

Functional Programming Challenges

Challenge 1: Converting a Lambda Calculus Expression to Alpha Normal Form

The first challenge requires you to write a function to convert a lambda expression to an alpha equivalent one and return this value. In general, there would be many possible answers. For example, *λx0 -> λx1 -> x0* is alpha equivalent to *λx1 -> λx -> y.* Your function however must construct an answer in which each bound variable uses the first available identifier. Note that variable numbering starts at 0. The first lambda expression above is therefore in alpha normal form, according to the definition given here, whereas the second is not. Care is required to avoid hiding a variable unintentionally. For example, in the expression *λx0 -> λx1 -> λx2 -> x0*, you cannot rename *x1* to *x0* as this would bind the later occurrence of *x0* to the wrong lambda abstraction. On the other hand, you can safely rename *x2* to *x1* as *x1*. Your implementation should have the property that two alpha equivalent expressions have the same alpha normal form. Here are some examples of this alpha normal form:

Lambda term	Alpha Normal Form
x1 x0	*x1 x0*
λx3 -> x2	*λx0 -> x2*
λx0 -> λx1 -> x0	*λx0 -> λx1 -> x0*
λx1 -> λx0 -> x1	*λx0 -> λx1 -> x0*
λx1 -> λx0 -> x0	*λx0 -> λx0 -> x0*
λx0 -> λx1 -> λx2 -> x0	*λx0 -> λx1 -> λx1 -> x0*

Challenge 2: Counting Lambda Calculus Reductions

For this challenge you will program beta reduction of a lambda expression using all possible strategies. In general a lambda expression may contain more than reducible (sub-)expression (redex). There are therefore many different possible reduction sequences. We are interested here only in reduction sequences that lead to an expression without any redexes, that is to say in beta normal form. The famous Church-Rosser theorem states that any two such end results will be the same, or at least alpha equivalent, lambda expressions. Note however that lambda reduction may not terminate, and moreover that some reduction sequences will be longer than others.

For example, the lambda expression whose concrete syntax is shown below has two reducible expressions, which are indicated by the underlined text: (λx -> λy -> x) z ((λt -> t) u). One reduction sequence is therefore:

    (λx -> λy -> x) z ((λt -> t) u)
    (λy -> z) ((λt -> t) u)
    z

and another one is:

    (λx -> λy -> x) z ((λt -> t) u))
    (λx -> λy -> x) z u
    (λy -> z) u
    z

You are asked to program a function which counts the number of different reduction sequences up to a given maximum length, starting with the given lambda expression and ending with a beta normal form that cannot be reduced further, and returns this count. Examples of this function are:

Lambda expression	*Maximum*	*Count of All Different Reduction Sequences*
λx -> (λy -> y)	0	0 – *as no reduction sequence is this short*
λx -> (λy -> y)	1	1 – *one singleton sequence containing this expression*
λx -> (λy -> y)	2	1 – *no other reduction sequences exist*
(λx -> x)(λy -> y)	0	0
(λx -> x)(λy -> y)	1	0 – *this expression is not in beta normal form yet*
(λx -> x)(λy -> y)	2	1 – *one reduction step gives a sequence of length 2*
(λx -> λy -> x) z ((λt -> t) u)	2	0
(λx -> λy -> x) z ((λt -> t) u)	3	1
(λx -> λy -> x) z ((λt -> t) u)	4	3 – *three reduction sequences of length up to 4 exist*

Challenge 3: Pretty Printing a Lambda Expression

You are asked to write a "pretty printer" to display a simple lambda expression. 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 represents the same abstract expression as the original one. Finally, your function should recognise a lambda expression or sub-expression alpha equivalent to the Scott encoding of a natural number, and print the corresponding numeral rather than the lambda expression. Use the encoding in which[2]

0 *is represented by λx -> λy -> x*
1 *is represented by λx -> λy -> y (λx -> λy -> x)*
2 *is represented by λx -> λy -> y (λx -> λy -> y (λx -> λy -> x)), and so on.*

[2] You may find slight variations elsewhere in the literature, since Scott did not formally publish his encoding. See the article *Programming in the λ-calculus: from Church to Scott and back* (Jansen 2013, The Beauty of Functional Code, Springer) for further explanations and discussion of this encoding and others.

In your output, use the backslash character to stand for the λ symbol. 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. Some examples of pretty printing are:

LamApp (LamVar 2) (LamVar 1)	"x2 x1"
LamApp (LamAbs 1 (LamVar 1)) (LamAbs 1 (LamVar 1))	"(λx1 -> x1) λx1 -> x1"
LamAbs 1 (LamApp (LamVar 1) (LamAbs 1 (LamVar 1)))	"λx1 -> x1 λx1 -> x1"
LamAbs 1 (LamAbs 2 (LamVar 1))	"0"
LamAbs 1 (LamAbs 1 (LamApp (LamVar 1) (LamAbs 1 (LamAbs 2 (LamVar 1)))))	"1"

Challenge 4: Parsing Let Expressions

You are asked to write a parser for the let expressions defined by the BNF grammar above. You are recommended to adapt 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 LetExpr and will return Nothing if the given string does not parse correctly according to the rules of the grammar. The grammar and LetExpr data type are defined on page 3 of these instructions. Note that you will need to transform the grammar into an equivalent form without left recursion, as this grammar construct results in a non-terminating parser. Some examples of the parsing function are:

"let x1 = x2"	Nothing -- *syntactically invalid wrt the given grammar*
"x1 (x2 x3)"	Just (LetApp (LetVar 1) (LetApp (LetVar 2) (LetVar 3)))
"x1 x2 x3"	Just (LetApp (LetApp (LetVar 1) (LetVar 2)) (LetVar 3))
"let f1 x1 = x2 in f1 x1"	Just (LetDef [([1,1], LetVar 2)] (LetApp (LetFun 1) (LetVar 1)))
"let f1 x2 = x2; f2 x1 = x1 in f1 x1"	Just (LetDef [([1,2],LetVar 2),([2,1],LetVar 1)] (LetApp (LetFun 1) (LetVar 1)))

Challenge 5: Converting a Let Expression to Lambda Calculus

The challenge requires you to convert a let expression into a lambda calculus expression. The lambda expression will be represented using the following data type defined on page 2 of these instructions. This challenge concerns parallel and recursive let definitions, so conversion can be quite complex. The description below builds up step by step to the full algorithm.

First consider a simple let expression without any use of recursion or parameters

let f = e in d

Here any occurrence of f in d is to be replaced by e, which can be represented using the notation

d [f := e]

This substitution can also be expressed directly in the lambda calculus as

(λf -> d) e

The next step, involves a let expression including a parameter or parameters, for example

let f x y = e in d

This can be expressed as a simple let by defining *f* as an abbreviation for a lambda expression

let f = (λx -> λy -> e) in d

Hence by the simple let conversion rule above, the equivalent lambda expression is

(λf -> d) (λx -> λy -> e)

Note that this is equivalent, using the substitution notation above, to

```
 d [ f := λx -> λy -> e ]
```

Now consider a recursive definition. Here it is recommended to use the trick described by Jansen (2013, op. cit.) to replace the recursion with self-application[3]

```
   let f x = e in d
```

Assume that f occurs free in d and e. This can nonetheless be reduced to a lambda expression via

```
   let f' = λf -> λx -> e[ f := f f] in
      d [ f := f' f']
```

Using the approach described previously and eliminating the resulting substitutions will result in a lambda calculus expression.

Finally, consider mutually recursive definitions of f and g.

```
   let f x = e; g y = h; in d
```

These can be converted to

```
   let f' = λf -> λg -> λx -> e [ f := f f g || g := g f g ]
       g' = λf -> λg -> λy -> h [ f := f f g || g := g f g ]
   in
      d [ f := f' f' g' || g := g' f' g']
```

where || is a parallel substitution[4]. Note that with care parallel substitution can be implemented by a sequence of simple substitutions, resulting once more in a lambda calculus expression, or you may choose to implement it directly. Note also that three or more mutually recursive functions can be converted in a similar way.

Here are some examples of the letToLambda conversion function:

let f0 = f0 in f0	(λx0 -> x0 x0) λx0 -> x0 x0
let f1 x2 = x2 in f1	(λx0 λx0 -> x0) λx0 λx0 -> x0
let f1 x2 x3 = x3 x2 in f1	λx0 -> λx1 -> x1 x0
let f0 x0 = f1; f1 x1 = x1 in f0	λx0 -> λx1 -> x1
let f0 x0 x1 = x0; f1 x1 = f0 x1 f1 in f1	λx0 -> x0

Note that there are multiple possible answers, and that this table gives only one possibility in each case. Your solution may produce different results and they will be marked correct so long as they are beta equivalent to the lambda expressions above. (Some of the results above have been beta reduced for brevity.)

Challenge 6: Converting Lambda Calculus to Let Expressions

This final challenge asks you to perform the reverse conversion by writing a function which takes a lambda expression and returns an equivalent let expression. This quite easy to do if each lambda abstraction can be converted locally. An additional constraint, however, is that the returned let expression can have only one let definition block, which must occur at the topmost level. Hence all functions defined in this block are global. There are various techniques you might use, such as the ones found in the lambda calculus section of the Wikipedia article covering lambda lifting in different

[3] An alternative is to use the Y fixed point combinator, which is also shown in Jansen's paper. You may use this solution if you prefer; it is more widely known and written about.

[4] Parallel substitutions occur simultaneously. For example (a b)[a := b || b := a] simplifies to (b a). This simplification can occur through a sequence of standard substitutions: (a b)[a := c][b := a][c := b].

notations: https://en.wikipedia.org/wiki/Lambda_lifting. Take care with your choice of individual transformation steps to ensure that the overall conversion process works correctly.

Some examples of the lambdaToLet conversion function are:

λx0 -> x0	let f0 x0 = x0 in f0
x1 λx0 -> x0	let f0 x0 = x0 in x1 f0
(λx0 -> x0) x1	let f0 x0 = x0 in f0 x1
(λx0 -> x0) (λx0 -> x0)	let f0 x0 = x0; f1 x0 = x0 in f0 f1
λx0 -> x0 λx1 -> x0 x1	let f0 x0 x1 = x0 x1; f1 x0 = x0 (f0 x0) in f1

As with the previous challenge, there are multiple correct answers. Your solution will be accepted as passing the test provided its results are equivalent to those given above.

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. Your *Tests.hs* file may incorporate and enhance the published *Challenges2019tests.hs* test cases and main function.

Your 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 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 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. Your code will be run using a command line such as *ghci –e "main" Challenges2019tests.hs*, and 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, but not third party ones. Note that the comp2209 virtual machine is running the Glorious Glasgow Haskell Compilation System version 7.6.3. Your own test cases will also be run using a similar command line, so make sure these define a suitable *main* function which 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.

Andy Gravell, *December* 2019