Introduction

This assignment asks you to tackle some functional programming challenges associated with interpreting, translating, analysing and parsing the lambda calculus and the closely related notation of combinatory logic. These notations are both known to be Turing complete[1], even though they are not practical programming languages in themselves. It is hoped these challenges will give you additional insights into the mechanisms used to implement functional languages, as well as practising some more advanced functional programming techniques such as pattern matching and the use of monads in parsing, for example.

As with the previous exercises your submitted solutions must be pure functional programs with no use of side effects such as printing, and no main program. You are of course welcome to use such features while developing your code, but remember to remove them before you upload it. Your code may import only such modules as are required for the monadic parser (challenge 4) below.

Note that this assignment involve 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://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. Within reason, you may incorporate, adapt and extend such code. It is your responsibility to check that your solution matches the behaviour specified in these instructions. Warning: where you make use of code from elsewhere, you *must* acknowledge and cite the source(s) in your report.

Lambda Calculus

You should start by reviewing the relevant material on the lecture slides. You may also benefit from some further reading such as the wikipedia article, https://en.wikipedia.org/wiki/Lambda_calculus, or Selinger's notes http://www.mscs.dal.ca/~selinger/papers/papers/lambdanotes.pdf.

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. This gives the following concrete syntax (expressed in BNF):

Expr ::= *Var* | *Expr Expr* | λ *var* -> *Expr* | (*Expr*)

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

```
data Expr =
    App Expr Expr
  | Lam Int Expr
  | Var Int
    deriving (Show)
```

[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.

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 non-negative integer, 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, λx1->x1 is then represented as Lam 1 (Var 1). Representing variables as integers rather than strings somewhat simplifies the solutions of the problems given below, in particular where it is required to generate a fresh variable that occurs nowhere else.

Functional Programming Challenges

Challenge 1: Lambda Calculus Utility Functions

The first part of this assignment asks you to write five utility functions that would be useful in developing an interpreter for the lambda calculus. These should be as follows

a) a function to rename a bound variable in a given lambda calculus expression. For example, renaming the bound variable x1 in λx1->x1 gives the new lambda expression λx2->x2.

b) a function to check that two lambda expressions are alpha-equivalent: one can be converted to the other by renaming its bound variables. For example, the lambda expressions λx1->x1 and λx2->x2 are alpha-equivalent since the first can be converted to the second by renaming as shown above.

c) a function to return the list of free variables in a given lambda calculus expression. For example, the expression λx1->x1 has no free variables, so your function should return [], whereas the expression λx1->(x1x2) has one free variable, x2, so the returned list should be [2].

d) a function to substitute an expression for a variable in the pure lambda calculus using for example the algorithm given in the Wikipedia article on the lambda calculus:

   x1[x1 := r] = r
   x2[x1 := r] = r *where x1 and x2 are different variables*
   (ts)[x := r] = (t[x := r])(s[x := r])
   (λx1->t)[x1 := r] = λx1->t *note that x1 is a bound variable here*
   (λx2->t)[x1 := r] = λx2->(t[x1 := r])
   where x1 and x2 are different variables, and x2 does not occur free in r

   Note that the final rule here may require the bound variable to be renamed to one which is fresh before it be applied. You may therefore need to implement some further auxiliary functions, for example to give the next fresh variable.

e) a function to test whether the given lambda expression contains any redex, meaning that beta-conversion would be possible. For example, λx1->(x1x2) has no redex and so result should be False, whereas λx1->((λx2->x2x2)x3) includes the redex (λx2->x2x2)x3, so the function should return True.

Your five utility functions must have the following signatures:

a) rename :: Expr -> Int -> Int -> Expr
   where the first integer is the identifier to be renamed, and the second integer is the replacement identifier

b) alphaEquivalent :: Expr -> Expr -> Bool

c) freeVariables :: Expr -> [Int]

d) substitute :: Expr -> Int -> Expr -> Expr

   where the integer is the identifier to be replaced by the third parameter in the expression given as parameter one

e) isNormal :: Expr -> Bool

Challenge 2: Transforming Lambda Calculus to Combinatory Logic

Combinatory logic is a closely related formalism. In this formalism there are no lambda expressions, but instead there are two pre-defined functions S and K where K x1 x2 = x2 and S x1 x2 x3 = x1 x2 (x1 x3). As in Haskell, function application in combinatory logic associates to the left. For example, the combinatory logic expression K x1 x2 is equivalent to ((K x1) x2). Surprisingly, any lambda calculus expression can be translated into combinatory logic as you are asked to demonstrate in this challenge. You will be using the following datatype for the abstract syntax of this notation:

```
 data ExprCL =
   AppCL ExprCL ExprCL
  | CombinatorK
  | CombinatorS
  | VarCL Int
   deriving (Show)
```

You are asked to write a function which converts a lambda expression to SK combinatory logic using the T[ ] transformation defined below[2]. It is likely that you will also need to define, informally or formally, a type for expressions which are halfway between pure lambda expressions and pure combinatory logic expressions.

1. $T[x] \Rightarrow x$
2. $T[(E_1\ E_2)] \Rightarrow (T[E_1]\ T[E_2])$
3. $T[\lambda x\text{->}E] \Rightarrow (\mathrm{K}\ T[E])$ — *if x does not occur free in E*
4. $T[\lambda x\text{->}x] \Rightarrow \mathrm{SKK}$
5. $T[\lambda x\text{->}\lambda y\text{->}E] \Rightarrow T[\lambda x\text{->}T[\lambda y\text{->}E]]$ — *if x occurs free in E*
6. $T[\lambda x\text{->}(E_1\ E_2)] \Rightarrow (\mathrm{S}\ T[\lambda x\text{->}E_1]\ T[\lambda x\text{->}E_2])$ — *if x occurs free in E_1 or E_2*

Your translation function must have the following signature:

```
translate :: Expr -> ExprCL
```

Challenge 3: Pretty Printing a Lambda Expression

You are asked to write a "pretty printer" to print a lambda expression. Your output should use the following conventions:

a) λx1->λx2->E is written λx1x2->E and likewise for higher levels of currying.

b) brackets are omitted where possible, using the convention that lambda expressions extend to the right as far as possible. Your pretty printer should generate λx1x2->x1x2x3 rather than, say, (λx1x2->((x1x2)x3)).

c) Beyond these requirements you are free to format and lay out the lambda expression as you choose in order to make it easier to read and understand.

[2] See https://en.wikipedia.org/wiki/Combinatory_logic

Challenge 4: Parsing Lambda Expressions

You are asked to write a lambda calculus parser. You should do this by extending 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, and paper:

http://www.cs.nott.ac.uk/~pszgmh/pearl.pdf, or
http://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf

The concrete syntax for the parser should be as described in challenge 3. In addition, include an optional where expression as in Haskell with the concrete syntax:

Expr ::= *Var* | *Expr Expr* | λ *Vars* -> *Expr* | (*Expr*) | *Expr* where *Var* = *Expr*
Vars ::= *Var* | *Var Vars*

Define the extended lambda notation as

```
data ExtExpr =
    ExtApp ExtExpr ExtExpr
  | ExtLam [Int] ExtExpr
  | Where ExtExpr Int ExtExpr
  | ExtVar Int
    deriving (Show)
```

For example the extended λx1->x2 where x2 = x1x1 should be parsed as

```
Where ExtExpr (ExtLam [ExtVar 1, ExtVar 2] ExtExpr (ExtVar 2)) 2
   (ExtExpr (ExtApp (ExtVar 1) (ExtVar 2)))
```

As well as the parser you are also asked to write a function to transform an extended lambda expression into the original abstract syntax given above. You will need to make use of the rule

E2 where x1 = E2 → [x1 := E2] E1 *where* x1 *does not occur free in* E2

Your functions should have the signature:

```
parse :: String -> ExtExpr
transform :: ExtExpr -> Expr
```

Challenge 5:

Finally you are asked to carry out the following exercise in equational reasoning: show that the Maybe type satisfies the applicative laws covered on pages 162 & 163 of Hutton's textbook and in the Learn you a Haskell page:

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Your proofs should be included in your final report.

Submission and Marking

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), together with your proofs (challenge 5) as a separate appendix.

Your report should be submitted as a PDF file, Report.pdf. Your Haskell solutions should be submitted as a single file Challenges.hs.

Each of the five lambda calculus utility functions will be marked out of 2 using the same marks scheme as the previous assignment:

2 / 2: the submitted solution passed all automatically applied tests.
1 / 2: the submitted solution fails one or more of the automatic tests, but is substantially correct.
0 / 2: no solution was provided, or the submitted solution is incorrect.

The marking scheme then awards up to 5 marks to your solution to each of the remaining three programming challenges, up to 5 for your explanation of how you implemented and tested these, and up to 5 for your equational reasoning:

5 / 5 completely correct, professional grade work
4 / 5 essentially correct with one or two minor flaws, very good or excellent work
3 / 5 mostly correct with a few minor flaws or one major one, good work
2 / 5 a promising start but requiring significant additional work to complete correctly, marginal
1 / 5 work showing limited understanding, far from correct and complete
0 / 5 not attempted or no understanding demonstrated

This gives a maximum of 35 marks for the 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 definitions and signatures given here. In addition some of your code will be read to confirm that it is comprehensible and maintainable.

Initial Draft, 16 *November* 2017