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.

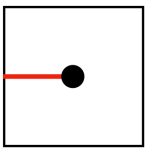
The grid may be of any rectangular size and will be given as a list of non-empty lists of Tile values. A Tile value is value of the data type given by:

```
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 ] ]
```

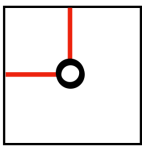
where a Tile simply lists which of its edges offer connection of wires. The Source and Sink tiles must contain at least one connector edge and Wire tiles must contain either zero (an empty Tile) or at least two connector edges. Duplicate entries in the edges list are ignored and order does not matter. 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 :

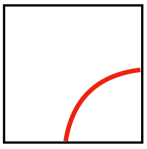
Source [West] could be represented visually as



Sink [North, West] could be represented visually as

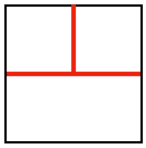


Wire [East, South] could be represented visually as



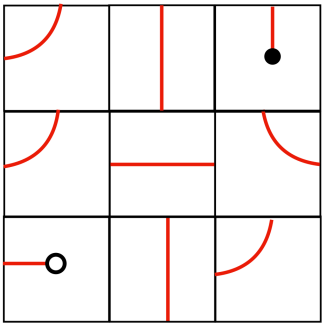
and finally

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

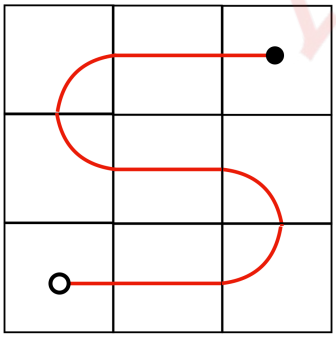


An example 3x3 puzzle is given below followed by a visual representation of the puzzle:

[[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)`