

## COMP2209 Assignment Instructions

<b>Module:</b>	<i>Programming III</i>			<b>Examiners</b> :	Julian Rathke, Nick Gibbins
<b>Assignment:</b>	<i>Haskell Programming Challenges</i>			<b>Effort:</b>	30 to 60 <i>hours</i>
<b>Deadline:</b>	16:00 on 13/1/2022	<b>Feedback:</b>	4/2/2022	<b>Weighting:</b>	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 Haskell 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, thus, it is recommended that you attempt 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>

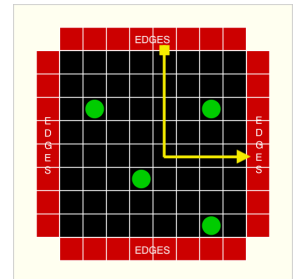
<http://book.realworldhaskell.org/read/testing-and-quality-assurance.html>

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.

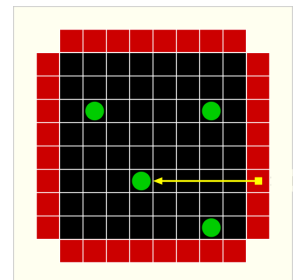
## The Challenges

### Part I – The Black Box Game

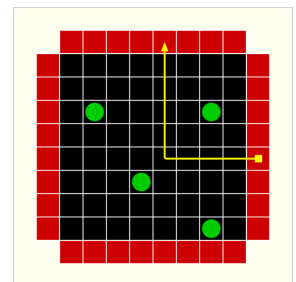
The board game “Black Box” consists of an  $N \times N$  non-empty square grid in which are hidden a given number of “atoms”. The player tries to determine the location of the atoms by firing a “ray” in to the grid from one of its edges. The ray will potentially deflect, reflect or be absorbed by the atoms and may exit the grid at one of the edges. The aim of the game is to deduce the location of the atoms using the least number of rays. There are a number of rules that determine the path of each ray detailed below:



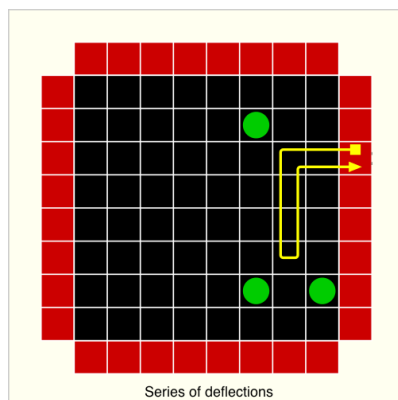
Rays will travel in straight lines directly through the grid unless disturbed by the presence of an atom. A ray that strikes an atom directly is “Absorbed” and does not exit the grid. In the example shown to the right, the ray is absorbed and does not leave the grid. Rays that are absorbed take priority over any deflections or reflections.



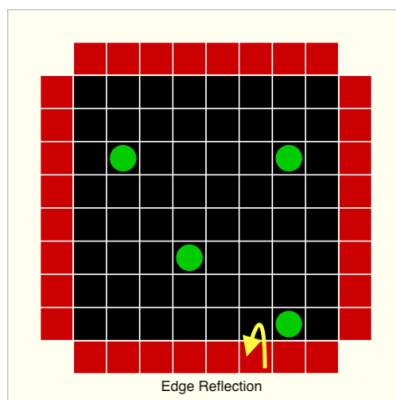
A ray that strikes an atom at its corner is deflected through 90 degrees away from the atom and its path continues. In this particular example shown to the right, the ray exits the grid at a different edge point than which it entered.



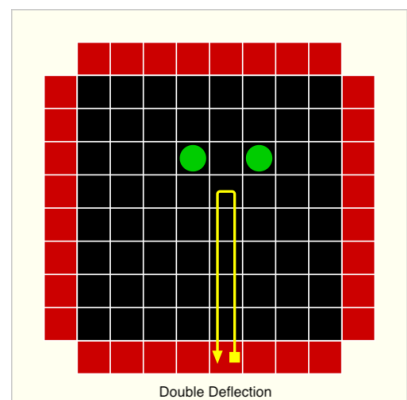
A ray that exits the grid at the same point at which it enters is considered to have been “Reflected”. This may occur via a series of deflections but may also occur in the special cases of an Edge Reflection and a Double Deflection. An edge reflection occurs when a ray immediately meets the corner of an atom at the edge of the grid. A double deflection occurs with two simultaneous deflections. Examples of each are given below:



Series of deflections



Edge Reflection



Double Deflection

For further information on the Black Box game, please see the Wikipedia page at ([https://en.wikipedia.org/wiki/Black\\_Box\\_\(game\)](https://en.wikipedia.org/wiki/Black_Box_(game)))

This part requires you to write functions for both calculating the paths of rays in a grid given a list of atoms, and deducing the location of a list of atoms given a list of paths of rays in the grid.

You will not be required to provide a graphical front-end for the game as we will limit ourselves to manipulating a back-end representation. We are going to make use of the following type synonyms:

```
type Pos = (Int,Int)
```

representing a (col,row)-coordinate within the 2D grid (**rows and column numbers within the grid begin at 1**) and

```
type EdgePos = ( Side, Int )
data Side = North | East | South | West deriving (Show,Eq,Ord)
```

representing the entry / exit points to the grid. For example, (North, 5) represents the entry point in the 5th column on the North face of the grid. Again, row column numberings begin at 1.

The type

```
type Atoms = [ Pos ]
```

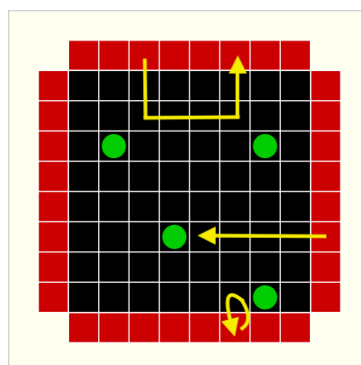
is used to represent the location of the hidden atoms in the grid and

```
type Interactions = [ (EdgePos , Marking) ]
data Marking = Absorb | Reflect | Path EdgePos deriving (Show, Eq)
```

represents the list of the outcomes of firing rays in to the grid from the stated entry position. Where the marking is a "Path", the exit position is recorded with the path also. For example, the list

```
[ ( ( East, 6 ) , Absorb ) , ( ( South, 6 ) , Reflect ) , ( (North, 3) , Path (North,6) ) ]
```

represents the interactions depicted in the game board below.



## Challenge 1: Calculate all interactions.

The first challenge requires you to define a function

```
calcBBInteractions :: Int -> Atoms -> Interactions
```

that, given an integer representing the NxN size of the grid (not including edges) and a list of atoms placed within the grid, returns the **set** of interactions from all possible edge entry points. That is, the order of the interactions returned is not important and there should be no duplicate entries starting at any edge point.

For the example board given above the function should return:

```
calcBBInteractions 8 [(2,3), (7,3), (4,6), (7,8)] =  
[ ((North,1), Path (West,2)), ((North,2), Absorb),  
  ((North,3), Path (North,6)), ((North,4), Absorb),  
  ((North,5), Path (East,5)), ((North,6), Path (North,3)),  
  ((North,7), Absorb), ((North,8), Path (East,2)),  
  ((East,1), Path (West,1)), ((East,2), Path (North,8)),  
  ((East,3), Absorb), ((East,4), Path (East,7)),  
  ((East,5), Path (North,5)), ((East,6), Absorb),  
  ((East,7), Path (East,4)), ((East,8), Absorb),  
  ((South,1), Path (West,4)), ((South,2), Absorb),  
  ((South,3), Path (West,7)), ((South,4), Absorb),  
  ((South,5), Path (West,5)), ((South,6), Reflect),  
  ((South,7), Absorb), ((South,8), Reflect),  
  ((West,1), Path (East,1)), ((West,2), Path (North,1)),  
  ((West,3), Absorb), ((West,4), Path (South,1)),  
  ((West,5), Path (South,5)), ((West,6), Absorb),  
  ((West,7), Path (South,3)), ((West,8), Absorb)]
```

## Challenge 2: Solve a Black Box

This challenge requires you to define a function

```
solveBB :: Int -> Interactions -> Atoms
```

that, given an integer N, and a list of the outcomes of firing rays in to the black box, returns a list of the positions of exactly N atoms that gives rise to the given list of interactions. Where no such list exists you should return the empty list. Where multiple possible placements exist then you may return any such placement. The order in which the positions of the atoms are listed is also unimportant. For example, when called with the integer value 4 and the output from the example given in Challenge 1 above, your function should return [(2,3), (7,3), (4,6), (7,8)].

## 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](https://en.wikipedia.org/wiki/Lambda_calculus), or Selinger's notes <http://www.mscs.dal.ca/~selinger/papers/papers/lambdanotes.pdf>, or both.

### Challenge 3: Pretty Printing a Lambda Expression with Scott Numerals

It is well-known that one may encode natural numbers within the lambda calculus. The Scott encoding  $[[n]]$  of naturals is defined recursively as follows:

$$[[0]] \text{ is } \lambda x \rightarrow (\lambda y \rightarrow x)$$
$$[[n]] \text{ is } \lambda x \rightarrow (\lambda y \rightarrow y \ [[n-1]]) \quad \text{for } n > 0$$

When working with lambda term that contain encodings it is often convenient to write the number  $n$  in place of the lambda term  $[[n]]$ .

We will be using the following data type for the abstract syntax of the lambda calculus:

```
data LamExpr = 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  $x_0$ ,  $x_1$ , and  $x_{456}$ . These are then represented in the abstract syntax as *Var 0*, *Var 1*, and *Var 456* respectively. Likewise, for example, the lambda expression  $\lambda x_1 \rightarrow x_2 x_1$  is represented as *LamAbs 1 (LamApp (Var 2) (Var 1))*.

Write a “pretty printer” to display a simple lambda expression. That is, define a function

```
prettyPrint :: LamExpr -> String
```

that un-parses a lambda expression to a string. Use a `"\"` symbol (suitably escaped) rather than  $\lambda$  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 the Scott encoding of some natural number. In such cases, print the corresponding number rather than the lambda expression encoding. 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:

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

## Challenge 4: Parsing Let Expressions

The “let” notation can also be used as an alternative to the lambda calculus as a small language of functions. 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 more concrete, numeric literals are also allowed in these expressions.

Consider the following BNF Grammar of the concrete syntax of let expressions.

```

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"

```

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)

```

As with the lambda calculus, application associates to the left and binds tighter than any other operator in each of the notations used here.

Your challenge is to define a function:

```

parseLet :: String -> Maybe LetExpr

```

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.

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:

"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)))



### Part III – Lambda Calculus and Combinators

Another alternative to lambda calculus for expressing functions is Combinatory Logic ( [https://en.wikipedia.org/wiki/Combinatory\\_logic](https://en.wikipedia.org/wiki/Combinatory_logic) ). This is a system of fixed functions used in combination by application that avoids the need for lambda binding. The language simply consists of three primitive combinators called S , K and I , named variables and application. As for lambda calculus, application associates to the left in combinatory logic. There are rewrite rules associated with each of the combinators as follows:

$$I\ x \rightarrow x$$

$$K\ x\ y \rightarrow x$$

$$S\ x\ y\ z \rightarrow x\ z\ (y\ z)$$

An interesting feature of combinatory logic is that it is sufficiently powerful to express all of lambda calculus. We can translate a lambda calculus term E to a corresponding combinatory logic term <E> as follows :

$$\langle x \rangle \text{ is } x$$

$$\langle e_1\ e_2 \rangle \text{ is } \langle e_1 \rangle \langle e_2 \rangle$$

$$\langle \lambda x \rightarrow e \rangle \text{ is } K \langle e \rangle \quad (\text{if } x \text{ does not occur free in } e)$$

$$\langle \lambda x \rightarrow x \rangle \text{ is } I$$

$$\langle \lambda x \rightarrow \lambda y \rightarrow e \rangle \text{ is } \langle \lambda x \rightarrow \langle \lambda y \rightarrow e \rangle \rangle \quad (\text{if } x \text{ occurs free in } e)$$

$$\langle \lambda x \rightarrow e_1\ e_2 \rangle \text{ is } S \langle \lambda x \rightarrow e_1 \rangle \langle \lambda x \rightarrow e_2 \rangle \quad (\text{if } x \text{ occurs free in } e_1 \text{ or } e_2)$$

For example, if we translate the lambda term  $\lambda x \rightarrow \lambda y \rightarrow y$  we get

$$\langle \lambda x \rightarrow \lambda y \rightarrow y \rangle$$

$$= \langle \lambda x \rightarrow \langle \lambda y \rightarrow y \rangle \rangle$$

$$= \langle \lambda x \rightarrow I \rangle$$

$$= K\ I$$

See how this lambda term is essentially a function that, when given any argument, reduces to the identity function. The reduction behaviour of K I can be seen to be similar.

For a longer example, if we translate the lambda term  $\lambda x \rightarrow \lambda y \rightarrow y\ x$  we get

$$\langle \lambda x \rightarrow \lambda y \rightarrow y\ x \rangle$$

$$= \langle \lambda x \rightarrow \langle \lambda y \rightarrow y\ x \rangle \rangle$$

$$= \langle \lambda x \rightarrow S \langle \lambda y \rightarrow y \rangle \langle \lambda y \rightarrow x \rangle \rangle$$

$$= \langle \lambda x \rightarrow S\ I \langle \lambda y \rightarrow x \rangle \rangle$$

$$= \langle \lambda x \rightarrow S\ I\ (K\ x) \rangle$$

$$\begin{aligned}
&= \langle \lambda x \rightarrow S I (K x) \rangle \\
&= S \langle \lambda x \rightarrow S I \rangle \langle \lambda x \rightarrow K x \rangle \\
&= S (K (S I)) \langle \lambda x \rightarrow K x \rangle \\
&= S (K (S I)) (S \langle \lambda x \rightarrow K \rangle \langle \lambda x \rightarrow x \rangle) \\
&= S (K (S I)) (S (K K) \langle \lambda x \rightarrow x \rangle) \\
&= S (K (S I)) (S (K K) I)
\end{aligned}$$

### Challenge 5: Converting Lambda Calculus to Combinatory Logic

Write a function

```
clTransform :: LamExpr -> CLEExpr
```

that translates a lambda expression in to a combinatory logic expression according to the above translation rules where the datatype CLEExpr is defined as

```
data CLEExpr = S | K | I | CLVar Int | CLEApp CLEExpr CLEExpr deriving (Show,Read,Eq)
```

Usage of the clTransform function on the examples show above is as follows:

$\lambda x_1 \rightarrow \lambda x_2 \rightarrow x_2$	<code>CLEApp K I</code>
$\lambda x_1 \rightarrow \lambda x_2 \rightarrow x_2 x_1$	<code>CLEApp (CLEApp S (CLEApp K (CLEApp S I))) (CLEApp (CLEApp S (CLEApp K K)) I)</code>

### Challenge 6: Counting and Comparing Lambda Calculus and Combinatory Logic Reductions

For this challenge you will define functions to perform reduction of a lambda expressions 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 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. We are also going to compare the same lengths for the same expression when it is converted to combinatory logic. You should not assume that the lambda expressions are closed.

Define functions:

```
innerRedn1 :: LamExpr -> Maybe LamExpr
outerRedn1 :: LamExpr -> Maybe LamExpr
```

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

Similarly, define

```
innerCLRedn1 :: CLEExpr -> Maybe CLEExpr
outerCLRedn1 :: CLEExpr -> Maybe CLEExpr
```

that take combinator expressions and perform a single reduction on that expression, if possible, by returning the reduced expression. Again, the functions should implement the (leftmost) innermost and outermost reduction strategies respectively.

Define a function

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

that takes a lambda expression 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 (in this order) :

1. Innermost reduction on e
2. Outermost reduction on e
3. Innermost reduction on the CL translation of e: < e >
4. Outermost reduction on the CL translation of e: < e >

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

(\x1 -> x1 x2) using bound 10	(Just 0,Just 0,Just 0,Just 0)
(\x1 -> x1) (\x2 -> x2) using bound 10	(Just 1,Just 1,Just 1,Just 1)
(\x1 -> x1 x1)(\x1 -> x1 x1) using bound 100	(Nothing,Nothing,Nothing,Nothing)
(\x1->\x2->x1) x3 ( (\x1->x1) x4 ) using bound 4	(Just 3,Just 2,Nothing,Just 4)
(\x1 -> \x2 -> x2) ((\x1 -> x1 x1) (\x1 -> x1 x1)) (\x1 -> x1) using bound 100	(Nothing,Just 2,Nothing,Just 2)

## 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*. 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" CW2TestSuite.hs`, where *CW2TestSuite.hs* is **my** test harness that 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. *We **will not** compile and execute your Tests.hs file when marking.*

## Appendix: Marking Scheme

Grade	Functional Correctness	Readability and Coding Style	Development & Testing
Excellent 5 / 5	Your code passes all of our test cases; 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 almost all of our test cases.	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 some of our tests cases.	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 a few of our 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 <i>let</i> and <i>where</i> 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.