COMP2209 Assignment Instructions

UNIVERSITY OF Southampton

Module:	*Programming III*			**Examiners:**	Julian Rathke, Nick Gibbins
Assignment:	*Haskell Programming Challenges*			**Effort:**	30 to 60 *hours*
Deadline:	16:00 on 12/1/2023	**Feedback:**	3/2/2023	**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 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 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:

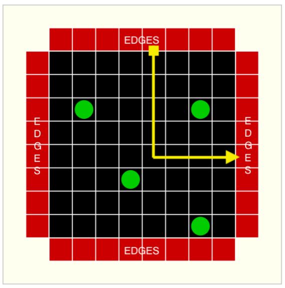
https://wiki.haskell.org/Debugging
https://www.quora.com/How-do-Haskell-programmers-debug
http://book.realworldhaskell.org/read/testing-and-quality-assurance.html

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.

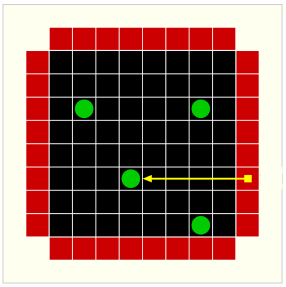
The Challenges

Part I – The Black Box Game

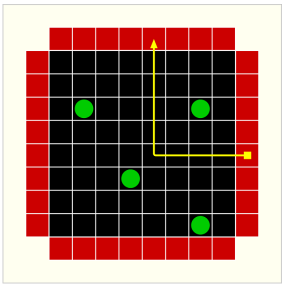
The board game "Black Box" consists of an N x N non-empty square grid in which are hidden a given number of "atoms". The player tries to determine the location of the atoms by firing a "ray" in to the grid from one of its edges. The ray will potentially deflect, reflect or be absorbed by the atoms and 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:



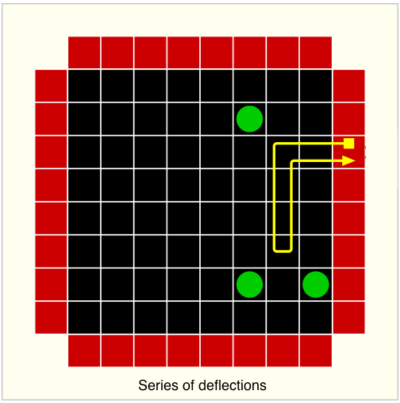
Rays will travel in straight lines directly through the grid unless disturbed by the presence of an atom. A ray that strikes an atom directly is "Absorbed" and does not exit the grid. In the example shown to the right, the ray is absorbed and does not leave the grid. Rays that are absorbed take priority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

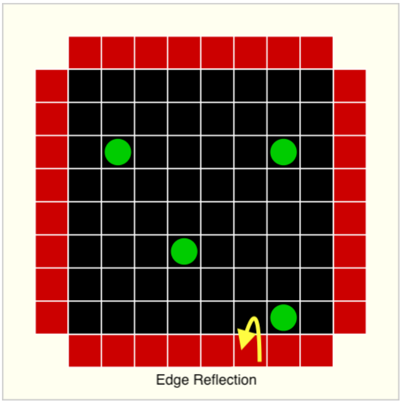


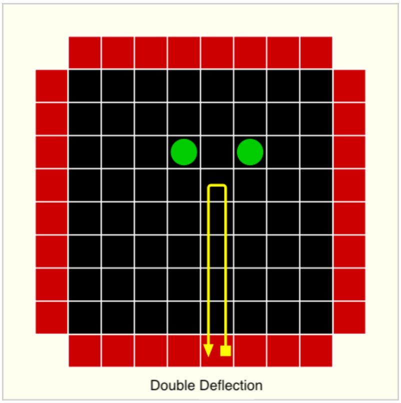
A ray that strikes at atom at its corner is deflected through 90 degrees away from the atom and its path continues. In this particular example shown to the right, the ray exits the grid at a different edge point than which it entered.



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))

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 ]
```

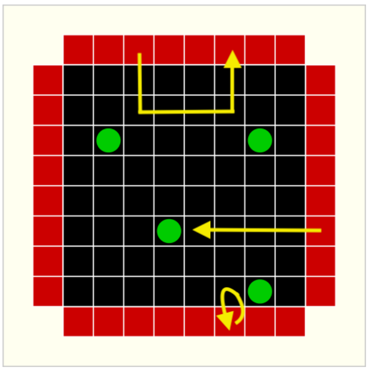
is used to represent the location of the hidden atoms in the grid
and

```
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 interactions.

The first challenge requires you to define a function

calcBBInteractions :: Int -> Atoms -> [EdgePos] -> Interactions

that, given an integer representing the NxN size of the grid (not including edges), a list of atoms placed within the grid, and a list of entry points at the edge of the grid, returns the **set** of interactions from the given 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) , (North,5) , (East, 7) , (South 7) , (West,3) , (West,1)] =

```
[
 ((North,1),Path (West,2)),
 ((North,5),Path (East,5)),
 ((East,7),Path (East,4)),
 ((South,7),Absorb),
 ((West,1),Path (East,1)),
 ((West,3),Absorb))
]
```

Challenge 2: Solve a Black Box

This challenge requires you to define a function

solveBB :: Int -> Interactions -> Atoms

that, given the size of the grid and a **partial** list of the outcomes of firing rays in to the black box, returns a **minimal** list of positions of atoms that could rise to the given list of interactions. You may assume that for the given list of interactions there always exists a possible placement of some atoms that is consistent with those inputs. Where multiple possible placements exist then you may return any such placement provided that there is no shorter list that also generates the same interactions. The order in which the positions of the atoms are listed is also unimportant. For example

solveBB 8 [((North,1),Path (West,2)), ((North,5),Path (East,5)), ((East,7),Path (East,4)), ((South,7),Absorb), ((West,1),Path (East,1)), ((West,3),Absorb)]

should return a placement of exactly **four** atoms that gives rise to these interactions. e.g. [(2,3) , (7,3) , (4,6) , (7,8)] but any four such atoms would be acceptable.

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 purposes of this coursework we will 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.

We will also make use of the datatype

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

Challenge 3: Pretty Printing a Lambda Expression in Alpha-Normal Form

Two Lambda expressions are said to be alpha-equivalent if there is a consistent safe renaming of their bound variables to make them identical expressions. For example

λx2 -> λx3 -> x2 x3 and λx3 -> λx2 -> x3 x2

are alpha-equivalent as they can both be consistently renamed to

λx4 -> λx5 -> x4 x5

We say that a lambda expression is in Alpha-Normal Form if each of its bound variables is named using the first available identifier where the ordering starts at "x0" and increases with integer value. Therefore, the expression λx2 -> λx3 -> x2 x3 is not in Alpha-Normal Form, but the alpha-equivalent expression λx0 -> λx1 -> x0 x1 is. Any lambda expression can be converted to an equivalent Alpha-Normal Form but 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. Two alpha equivalent expressions always have the same Alpha-Normal Form. Here are some examples of this alpha normal form:

Lambda Expression	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

This challenge requires you to write a function that takes a data type representation of a lambda calculus expression and to "pretty print" it by returning a string representation of an alpha-equivalent expression given in Alpha-Normal form. That is, define a function

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

that converts a lambda expression to an equivalent expression in Alpha-Normal Form and then un-parses it to a string. Use a "\" symbol (suitably escaped) rather than λ *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. 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))	"(\\x0 -> x0) \\x0 -> x0"
LamAbs 1 (LamApp (LamVar 1) (LamAbs 1 (LamVar 1)))	"\\x0 -> x0 \\x0 -> x0"
LabApp (LamVar 2) (LamAbs 1 (LamAbs 2 (LamVar 1)))	"x2 \\x0 -> \\x1 -> x0"
LamAbs 1 (LamAbs 1 (LamApp (LamVar 1) (LamAbs 1 (LamAbs 2 (LamVar 1)))))	"\\x0 -> \\x0 -> x0 \\x0 -> \\x1 -> x0 "

Challenge 4: Parsing Arithmetic Expressions

We can define a small language of arithmetic expressions that features addition, multiplication and "operator sections" such as we find in Haskell. For example, consider the following BNF Grammar of the concrete syntax (ignoring whitespace) of arithmetic expressions.

```
Expr ::= Expr1 "*" Expr |  Expr1
Expr1 ::= Expr2 "+" Expr1 | Expr2
Expr2 ::= Num | Section Expr2 | "(" Expr ")"
Section ::= "(" "+" Expr ")"
Num ::= Digits
Digits ::= Digit | Digits Digit
Digit ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”
```

An example expression in this concrete syntax is " (+ (5 * 2)) 6 + 7 ".

To represent expressions of this language using abstract syntax we will be using the following Haskell data type :

```
data ArithExpr = Add ArithExpr ArithExpr | Mul ArithExpr ArithExpr | Section ArithExpr  |
                 SecApp ArithExpr ArithExpr |  ArithNum Int     deriving (Show, Eq,Read)
```

The example expression above would be represented as the Haskell value

```
Add  ( SecApp ( Section ( Mul ( ArithNum 5 ) ( ArithNum 2 )))  ( ArithNum 6 ))  ( ArithNum 7 )
```

Your challenge is to define a function:

```
parseArith ::  String -> Maybe ArithExpr
```

that returns Nothing if the given string does not parse correctly according to the rules of the concrete grammar 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:

parseArith "1 + 2"	Just (Add (ArithNum 1) (ArithNum 2))
parseArith "(+1) 2"	Just (SecApp (Section (ArithNum 1) (ArithNum 2)))
parseArith "2 (+1)"	Nothing
parseArith "(+1) (+2) 3"	Just (SecApp (Section (ArithNum 1)) (SecApp (Section (ArithNum 2)) (ArithNum 3)))

Part III – Lambda Calculus and Arithmetic

It is well known that the Lambda Calculus can be used to encode arithmetic. In the next challenges we will explore this idea. To start, it would be helpful to read about the Church encoding of the natural numbers in lambda calculus (https://en.wikipedia.org/wiki/Church_encoding). We see from this that each natural number can be represented as a lambda calculus expression as follows :

0 is λf -> λx -> x
1 is λf -> λx -> f x
2 is λf -> λx -> f (f x)
3 is λf -> λx -> f (f (f x))
...
n is λf -> λx -> f (f (... n times ... f x) ...))

We can use this as a basis for encoding our language of arithmetic expressions:

To add two Church numerals we may use the lambda expression

λm -> λn -> (λf -> λx -> m f (n f x))

call this "plus". So the lambda encoding of "2 + 3" is the lambda expression "plus" applied to the encoding of 2 and the encoding of 3.

Similarly, we can multiply two Church numerals using the lambda expression

λm -> λn -> (λf -> λx -> m (n f) x)

that we refer to as "mult". So the lambda encoding of "2 * 3" is the lambda expression "mult" applied to the encoding of 2 and the encoding of 3.

To encode the application of section operator "(+ E1) E2" for some expression E, we should apply "plus" to the encoding of E1 and E2.

Challenge 5: Converting Arithmetic Expressions to Lambda Calculus

Write a function

```
churchEnc :: ArithExpr -> 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 churchEnc function on the examples show above is as follows:

churchEnc (ArithNum 2)	LamAbs 0 (LamAbs 1 (LamApp (LamVar 0) (LamApp (LamVar 0)(LamVar 1)
churchEnc (SecApp (Section (ArithNum 1)) (ArithNum 1))	LamApp (LamApp (EPlus) (E1)) (E1) where E1 = LamAbs 0 (LamAbs 1 (LamApp (LamVar 0) (LamVar 1))) and EPlus = LamAbs 0 (LamAbs 1 (LamAbs 2 (LamAbs 3 (LamApp (LamApp (LamVar 0) (LamVar 2)) (LamApp (LamApp (LamVar 1) (LamVar 2)) (LamVar 3))))))

Challenge 6: Counting and Comparing Lambda Calculus and Arithmetical Reductions

For this challenge you will define functions to perform reduction of both arithmetic and lambda expressions. We will implement an innermost leftmost reduction strategy for both. A good starting point is to remind yourself of the definitions of innermost and leftmost evaluation in Lecture 34 - Evaluation Strategies.

We are going to compare the differences between the lengths of reduction sequences to terminated values for innermost reduction for a given arithmetic expression and the lambda expression obtained by converting the arithmetic expression to a lambda expression as defined in Challenge 5.

In order to understand evaluation in the language of arithmetic expressions, we need to identify the redexes of that language. We will express these using the Haskell datatype :

```
Add (ArithNum n) (ArithNum m)  ------>     ArithNum ( n + m )
Mul (ArithNum n) (ArithNum m)  ------>     ArithNum ( n * m )
SecApp (Section (ArithNum n)) (ArithNum m) -----> ArithNum ( n + m )
```

all reductions of arithmetic expressions are an instance of one of these within a possibly larger expression.

Define a function:

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

that takes a lambda expression and performs a single reduction on that expression, if possible, by returning the reduced expression. The function should implement the (leftmost) innermost reduction strategy.

Similarly, define

```
innerArithRedn1 :: ArithExpr -> Maybe ArithExpr
```

that takes an arithmetic expression and performs a single reduction on that expression, if possible, by returning the reduced expression. Again, the function should implement the (leftmost) innermost reduction strategy.

Define a function

```
compareArithLam :: ArithExpr -> ( Int, Int )
```

that takes an arithmetic expression and returns a pair containing the length of two reduction sequences In each case the number returned should be the number of steps needed for the expression to terminate. Given an input **arithmetic expression** E, the pair should contain lengths for (in this order) :

1. Innermost reduction on E
2. Innermost reduction on the lambda calculus translation of E: < E >

Example usages of the compareArithLam function (inputs are pretty printed) are:

"4"	(0,0)
"4 + 5"	(1,6)
"(4 + 5) * 3"	(2,28)
"(((+1) 4) + 5) * 3"	(3,36)

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 *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 *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.