Text

Description automatically generated with medium confidence

|  |
| --- |
| **Lenguajes de Programación** |
|  |
| Guías de Trabajos Prácticos |

# Fraction

## Instructions

You have to implement the arithmetic operations over fractions. You will be using a Tuple (Int, Int) as the fraction type. A type synonym is already defined as:

type Fraction = (Int, Int)

## 1. Implement the operations add, sub, mul and divide.

> add (1,2) (1,4)  
(6, 8)  
> sub (1,2) (1,4)  
(2, 8)  
>mul (1,2) (1,4)  
(1,8)  
>divide (1,2) (1,4)  
(4,2)

## 2. Implement the Highest common factor function hcf (AKA greatest common divisor)

> hcf 4 2  
2  
> hcf 3 2  
1  
> hcf 2 0  
2

## 3. Modify the functions defined above to simplify the results

> add (1,2) (1,4)  
(3, 4)  
> sub (1,2) (1,4)  
(1, 4)  
>mul (1,2) (1,4)  
(1,8)  
>div (1,2) (1,4)  
(2,1)

To Tests this go to **Test.hs** and comment/uncomment the 2 main functions alternatives

-- Comment this line and uncomment the line below to run the tests doing simplification  
main = hspec $ test id  
-- main = hspec $ test simplify

Should become

-- Comment this line and uncomment the line below to run the tests doing simplification  
-- main = hspec $ test id  
main = hspec $ test simplify

# Instructions

The Collatz Conjecture or 3x+1 problem can be summarized as follows:

Take any positive integer n. If n is even, divide n by 2 to get n / 2. If n is odd, multiply n by 3 and add 1 to get 3n + 1. Repeat the process indefinitely. The conjecture states that no matter which number you start with, you will always reach 1 eventually.

Given a number n, return the number of steps required to reach 1.

## Examples

Starting with n = 12, the steps would be as follows:

1. 12
2. 6
3. 3
4. 10
5. 5
6. 16
7. 8
8. 4
9. 2
10. 1

Resulting in 9 steps. So for input n = 12, the return value would be 9.

# List Exercises

## Instructions

You have to implement exercises working over lists

## 1. List as Sets

Suppose that lists describe a Set of elements.

The member and union functions are defined as examples. You need to define the intersection and difference operations

> intersection [1..5] [7, 3, 1]  
[1, 3]  
> difference [1..5] [7, 3, 1]  
[2,4,5]  
> difference [7, 3, 1] [1..5]  
[7]

## 2. Insertion Sort

Implement an operation to insert an element in the right position on an ordered list.

You should keep **duplicated** elements.

> insert 4 [1,2,5,7]  
[1,2,4,5,7]  
> insert 2 [1,2,5,7]  
[1, 2, 2, 5, 7]

Based on the above implement an insertion sort.

> insertionSort [6, 3, 5]  
[3,5,6]

Reimplement it using a foldr

### 3. Numeral Systems

First define a function that gets a list of Binary digits.

And gets the value of it as an Integer

Remember that if the number is:

xn … x1 x0

x0 2 0 + x1 2 1 + … + xn 2 n

> binaryToDecimal [1, 1, 0, 1]  
13

Hint: Use an auxiliary function that receives the list in reverse order.

Create an adicional function that receives the base as an argument:

> toDecimal 8 [7, 3]  
59  
> toDecimal 16 [1, 1]  
17

Create a function that instead of receiving a list receives a string

> toDec 8 "73"  
59  
> toDec 16 "1F"  
31

Hint use map and the function digitToInt from Data.Char

Repeat the last exercise using a List Comprehension and the zip function

> decimal 8 "73"  
59  
> decimal 16 "1F"  
31

## 4. Firsts Elements of a list

Given a List, return a **Nested List** containing:

The first element, the first 2 elements, the first 3 elements, etc.

> firsts [1, 3, 5]  
[[1],[1,3],[1,3,5]]  
> firsts "Hello"  
["H","He","Hel","Hell","Hello"]

## 5. Binary Operations – BONUS exercise

Given two Strings that represents numbers in binary. Implement the ‘binaryAdd’ function.

**DO NOT USE** any predefined arithmetic operation

> binaryAdd "10" "1"  
"3"  
> binaryAdd "1111" "11"  
"10010"

# List Exercises

## Instructions

You have to implement exercises working over lists

You have a type defined as:

data Bit = F | T  
  
type Bits = [Bit]

## 1. Convert a Char to Bits

You need to create function to convert a Char to a list of bits Bits

> charToBits 'A'  
[F,T,F,F,F,F,F,T]  
  
> charToBits 'a'  
[F,T,T,F,F,F,F,T]  
  
> charToBits '\0'  
[F,F,F,F,F,F,F,F]

### Hints:

You have already being provided with a function:

bitAt :: Int -> Char -> Bit

That given a Char and an Int (The position of the bit) returns T if the bit at that offset is set

So you can implement charToBits with a simple List Comprehension

## 

## 2. Convert a String to Bits

Using the function defined in the previous exercise define a function bits to convert an String to a list of bits (Bits)

> bits "AB"   
[F,T,F,F,F,F,F,T,F,T,F,F,F,F,T,F]

It is a plus if it works with infinite lists:

> take 15 $ bits $ cycle "AB"  
[F,T,F,F,F,F,F,T,F,T,F,F,F,F,T]

Try to use the map and foldr functions to implement it

## 4.BONUS Solve the ‘8’ Queens problem

The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another.

You Should return a List of the solutions. Where each solution is a List of rows with the position of a queen in this row.

For example: [2, 1] means a queen in column 2 in row 1 and a queen in column 1 in row 2.

Actually the function accepts any board size:

> queens 4   
[[2,4,1,3],[3,1,4,2]]

# Binary Trie

## Instructions

You have to implement a Binary Trie

data Trie a = ........ -- complete it

## 1. Implement a method to build a simple trie from 2 elements:

> t = (Leaf 'A' :-: Leaf 'B') :-: Leaf 'C'  
> t  
(Leaf 'A' :-: Leaf 'B') :-: Leaf 'C'

## 2. Implement the left and right methods

They should return the left or right trie respectively.

For example:

> left t  
Leaf 'A' :-: Leaf 'B'  
> right $ left t  
Leaf 'B'  
> left $ right $ left t  
\*\*\* Exception: Left of: Leaf 'B'

## 3. Using the Bit type (As way defined in a previous exercise)

data Bit = T | F  
  
type Bits = [Bit]

Implement a function that search the Trie based on a List of Bits (Bits) and return the value found. (F moves left and T )

> find [F, T] t  
'B'  
> find [T] t  
'C'

Implement a function decode, that given Bits and a Trie, generate an String. Consuming the List of Bits until its empty

> decode [F, T, F, F, F, T, F, F] t  
"BABA"  
> decode [T, F, F, F, T, F, F] t  
"CABA"

## 4. Given a Trie return a List of Tuples with the mapping of the value to Bits

> toList t  
[('A', [F, F]), ('B', [F, T]), ('C', [T])]

# Project Shape

## Create a Point and Shape types

### 1. Implement methods for the Point type

λ> origin  
Point { x = 0.0, y = 0.0 }  
λ> point (1.0, 1.0)  
Point { x = 1.0, y = 1.0 }

### 2. Implement methods for the Rectangle type

λ> rectangle (1.0, 2.0)  
Rectangle (Point {x = 0.0, y = 0.0}) (Point {x = 1.0, y = 2.0})  
λ> r = rectangle (1.0, 2.0)  
λ> base r  
1.0  
λ> height r  
2.0

### 3. Implement methods for the Circle type

λ> circle 2.0  
Circle (Point {x = 0.0, y = 0.0}) 2.0

### 4. Create a Shift class type with a shift method and implement it

λ> shift (point (1, 3)) (1, 2)  
Point {x = 2.0, y = 5.0}  
  
λ> shift (circle 2.0) (1, -1)  
Circle (Point {x = 1.0, y = -1.0}) 2.0  
  
λ> shift (rectangle (1.0, 2.0)) (1, -1)  
Rectangle (Point {x = 1.0, y = -1.0}) (Point {x = 2.0, y = 1.0})

### 5. Create a Surface class type and implement the surface method for Rectangle and Circle

λ> surface (rectangle (1.0, 2.0))  
2.0  
  
λ> surface (circle 2.0)  
12.566370614359172

# Project Frequencies

## Create a Char Frequency App

#### You should use the Trie implemented in the previous TP

### 1. Create a Function that given any List [a] returns a Map a Int.

The Int is the count of the number of instances of a in the List

> frequencyMap [1, 2, 3, 1, 3, 1]  
fromList [(1,3),(2,1),(3,2)]

### 2. Adapt the insert and insertionSort from TP1 to work with any Ord type

> insertionSort "PAPAYA"  
"AAAPPY"  
> insert 'T' $ insertionSort "PAPAYA"  
"AAAPPTY"

### 3. Create a function that returns a List of Frequencies ordered by number of occurrences

The Frequency type, is defined by:

type Frequency = (Int, Char)

You should use the frequencyMap and insertionSort functions

You can also use the swap function from Data.Tuple

Examples:

> frequencies "PAPAYA"  
[(1,'Y'),(2,'P'),(3,'A')]  
> frequencies "Zapallo con Papa"  
[(1,'P'),(1,'Z'),(1,'c'),(1,'n'),(2,' '),(2,'l'),(2,'o'),(2,'p'),(4,'a')]

### 4.Create a main function that:

* Takes as an argument a file name (Use getArgs)
* If the file name is not specified prints an error message
* Reads the file (Using readFile).
* Counts the frequencies of the letters in the file
* Prints the table of frequencies in **REVERSE** order (Starting from the higher ones).

> freq example.txt | head  
' ': 292  
'e': 148  
'o': 124  
't': 98  
'h': 96  
'n': 95  
'a': 93  
's': 92  
'r': 68  
'i': 65

# Project Huffman

## Build an App to compress/decompress using the Huffman method

You can use the functions defined in the previous TPs

### 1. Create the Huffman Trie for a given String

Huffman coding uses a specific method for choosing the representation for each symbol. Resulting in a prefix code:

That is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol.

A Binary Trie provides a representation of this code:

* Each Leaf corresponds to a symbol to be coded. (A Char in our case)
* The Path to the leaf corresponds to the code to be assigned to that symbol

The function Should have the following type:

huffmanTrie::String -> Trie Char

How to build the Trie ?

You are going to work with a List of Pairs representing the frequency and a partial Trie:

[(Int, Trie Char)]

1. Start with the frequencyMap and create an ordered list in the form: [(Int, Leaf Char)]
2. Remove the first 2 elements and join them in a new Pair: (n+m, a :-: b)
3. Insert this new pair in the list, keeping the order
4. When the list has only one element you are done.

### 2. Encode/Decode a String to/from a List of Bits

encode :: String -> Trie Char -> Bits

decode::Bits -> Trie Char -> String

### 3. Create an app (function main) to run it from the command line

You should accept the file as the first argument, and print the % of compression

### 4. (Optional) Add a step to the encode/decode process to convert the output to/from an String

On the main app write the result to a File

# Fraction

## Instructions

You have to implement the arithmetic operations over fractions. You will be using a tuple defined as (i32, i32) for the fraction type. A type synonym is already defined as:

type Fraction = (Int, Int);

## 1. Implement the operations add, sub, mul and divide.

> add((1,2), (1,4))  
(6, 8)  
> add((1,2), (1,4))  
(2, 8)  
> mul ((1,2), (1,4))  
(1,8)  
>divide (1,2) (1,4)  
(4,2)

## 2. Implement the Highest common factor function hcf (AKA greatest common divisor)

> hcf(4, 2)  
2  
> hcf(3, 2)  
1  
> hcf(2, 0)  
2

## 3. Modify the functions defined above to simplify the results

To test this remove the #[ignore] marker in the tests.

> add((1,2), (1,4))  
(3, 4)  
> sub((1,2), (1,4))  
(1, 4)  
>mul((1,2), (1,4))  
(1,8)  
>div((1,2), (1,4))  
(2,1)

# Collatz Conjecture

The Collatz Conjecture or 3x+1 problem can be summarized as follows:

* Take any positive integer n.
* If n is even, divide n by 2 to get n/2.
* If n is odd, multiply n by 3 and add 1 to get 3n + 1.
* Repeat the process indefinitely.

The conjecture states that no matter which number you start with, you will always reach 1 eventually.

Given a number n, return the number of steps required to reach 1.

## Examples

Starting with n = 12, the steps would be as follows:

12, 6, 3, 10, 5, 16, 8, 4, 2, 1

Resulting in 9 steps. So for input n = 12, the return value would be 9.

# Fraction as a Type

## Instructions

You have to implement the arithmetic operations over fractions. The idea is now to create tuple struct for the fraction type. The type is defined as:

struct Fraction(i32, i32);

## 1. Implement the operations add, sub, mul and divide.

> Fraction(1, 2).add(Fraction(1,4))  
Fraction(3, 4)  
> Fraction(1,2).sub(Fraction(1,4))  
Fraction(1, 4)  
> Fraction(1, 2).mul(Fraction(1, 4))  
Fraction(1,8)  
>Fraction(1, 2).divide(Fraction(1, 4))  
Fraction(2,1)

## 2. Implement the Add trait

> Fraction(1, 2) + Fraction(1,4)  
Fraction(3, 4)

# Health Monitoring

# Instructions

You’re working on implementing a health-monitoring system. As part of that, you need to keep track of users’ health statistics.

You’ll start with some stubbed functions in an impl block as well as the following struct definition:

pub struct User {  
 name: String,  
 age: u32,  
 weight: f32,  
}

Your goal is to implement the stubbed out methods on the User struct defined in the impl block.

For example, the new method should return an instance of the User struct with the specified name, age, and weight values.

let mut bob = User::new(String::from("Bob"), 32, 155.2);  
// Returns: a User with name "Bob", age 32, and weight 155.2

The weight method should return the weight of the User.

bob.weight();  
// Returns: 155.2

The set\_age method should set the age of the User.

bob.set\_age(33);  
// Updates Bob's age to 33; happy birthday Bob!

Have fun!

# Queen Attack

## Instructions

Given the position of two queens on a chess board, indicate whether or not they are positioned so that they can attack each other.

In the game of chess, a queen can attack pieces which are on the same row, column, or diagonal.

A chessboard can be represented by an 8 by 8 array.

So if you’re told the white queen is at (2, 3) and the black queen at (5, 6), then you’d know you’ve got a set-up like so:

\_ \_ \_ \_ \_ \_ \_ \_  
\_ \_ \_ \_ \_ \_ \_ \_  
\_ \_ \_ W \_ \_ \_ \_  
\_ \_ \_ \_ \_ \_ \_ \_  
\_ \_ \_ \_ \_ \_ \_ \_  
\_ \_ \_ \_ \_ \_ B \_  
\_ \_ \_ \_ \_ \_ \_ \_  
\_ \_ \_ \_ \_ \_ \_ \_

You’d also be able to answer whether the queens can attack each other. In this case, that answer would be yes, they can, because both pieces share a diagonal.

# Clock

## Instructions

Implement a clock that handles times without dates.

You should be able to add and subtract minutes to it.

Two clocks that represent the same time should be equal to each other.

You will also need to implement .to\_string() for the Clock struct. We will be using this to display the Clock’s state. You can either do it via implementing it directly or using the [Display trait](https://doc.rust-lang.org/std/fmt/trait.Display.html).

Did you implement .to\_string() for the Clock struct?

If so, try implementing the [Display trait](https://doc.rust-lang.org/std/fmt/trait.Display.html) for Clock instead.

Traits allow for a common way to implement functionality for various types.

For additional learning, consider how you might implement String::from for the Clock type. You don’t have to actually implement this—it’s redundant with Display, which is generally the better choice when the destination type is String—but it’s useful to have a few type-conversion traits in your toolkit.

# Sum Of Multiples

## Instructions

Given a number, find the sum of all the unique multiples of particular numbers up to but not including that number.

If we list all the natural numbers below 20 that are multiples of 3 or 5, we get 3, 5, 6, 9, 10, 12, 15, and 18.

The sum of these multiples is 78.

> sum\_of\_multiples(20, &[3, 5])  
78  
> sum\_of\_multiples(100, &[3, 5])  
2318

# Word Count

## Instructions

Given a phrase, count the occurrences of each *word* in that phrase.

Words are separated by **whitespace**

When counting words you can assume the following rules:

1. The count is *case insensitive* (i.e. “You”, “you”, and “YOU” are 3 uses of the same word)
2. The count is *unordered*; the tests will ignore how words and counts are ordered
3. The words can be separated by *any* form of whitespace (i.e. “,”“,” “).

Example:

For the phrase "One two three four one one three" the count would be:

one: 3  
two: 1  
three: 2  
four: 1

#### Hint

Check the .split\_whitespace() method

# Circular Buffer

# Instructions

A circular buffer, cyclic buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end.

A circular buffer first starts empty and of some predefined length. For example, this is a 7-element buffer: [ ][ ][ ][ ][ ][ ][ ]

Assume that a 1 is written into the middle of the buffer (exact starting location does not matter in a circular buffer): [ ][ ][ ][1][ ][ ][ ]

Then assume that two more elements are added — 2 & 3 — which get appended after the 1: [ ][ ][ ][1][2][3][ ]

If two elements are then removed from the buffer, the oldest values inside the buffer are removed. The two elements removed, in this case, are 1 & 2, leaving the buffer with just a 3: [ ][ ][ ][ ][ ][3][ ]

If the buffer has 7 elements then it is completely full: [5][6][7][8][9][3][4]

When the buffer is full an error will be raised, alerting the client that further writes are blocked until a slot becomes free.

Finally, if two elements are removed then what would be returned is 5 & 6 yielding the buffer: [ ][ ][7][8][9][3][4]

## Hint

To increment an index n and wrap-around once you reach len you can do:

n = (n + 1) % len

To decrement x position wrapping around you can do:

(len + n - x) % len

# Protein Translation

## Instructions

Translate RNA sequences into proteins.

RNA can be broken into three nucleotide sequences called codons, and then translated to a polypeptide like so:

RNA: "AUGUUUUCU" => translates to

Codons: "AUG", "UUU", "UCU" => which become a polypeptide with the following sequence =>

Protein: "Methionine", "Phenylalanine", "Serine"

There are 64 codons which in turn correspond to 20 amino acids; however, all of the codon sequences and resulting amino acids are not important in this exercise. If it works for one codon, the program should work for all of them. However, feel free to expand the list in the test suite to include them all.

There are also three terminating codons (also known as ‘STOP’ codons); if any of these codons are encountered (by the ribosome), all translation ends and the protein is terminated.

All subsequent codons after are ignored, like this:

RNA: "AUGUUUUCUUAAAUG" =>

Codons: "AUG", "UUU", "UCU", "UAA", "AUG" =>

Protein: "Methionine", "Phenylalanine", "Serine"

Note the stop codon "UAA" terminates the translation and the final methionine is not translated into the protein sequence.

Below are the codons and resulting Amino Acids needed for the exercise.

| Codon | Protein |
| --- | --- |
| AUG | Methionine |
| UUU, UUC | Phenylalanine |
| UUA, UUG | Leucine |
| UCU, UCC, UCA, UCG | Serine |
| UAU, UAC | Tyrosine |
| UGU, UGC | Cysteine |
| UGG | Tryptophan |
| UAA, UAG, UGA | STOP |

#### You should use the new switch statement to do the codon to protein translation

#### You should use streams to process the RNA String

### Hint

Start with an IntStream