Part One

Challenges Chapter 1 (1.4)

Challenge 1 (1.4.1)

Implement, as best as you can, the identity function in your favorite language (or the second favorite, if your favorite language happens to be Haskell).
Solution for Rust:

```rust
fn my_id<T>(a:T)->T{a}
```

Challenge 2 (1.4.2)

Implement the composition function in your favorite language. It takes two functions as arguments and returns a function that is their composition.
Solution for Rust:

```rust
fn compose<'a,F,G,A,B,C>(f:F, g:G) -> Box<(dyn Fn(A)->C + 'a)>
where
F: Fn(A)->B + 'a,
G: Fn(B)->C + 'a,
{
    return Box::new(move |a| g(f(a)) );
}
```

Challenge 3 (1.4.3)

Write a program that tries to test that your composition function respects identity.
This isn't possible. With Rice's theorem ($S = \{f \in \mathcal{R} | f(id, g) = g, f(g, id) = g\}$) it follows, that such questions for Programs are undecidable. But some Tests could be written as follows.

```rust
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_compose_id(){
        let foo = compose(id::<i16>, id::<i16>);
        for i in i16::MIN..i16::MAX {
            assert!(i == foo(i));
        }
```

```
        let foo = compose(|x:i16| x + 5, id);
        for i in i16::MIN..i16::MAX-5 {
            assert!(i + 5 == foo(i));
        }
        let foo = compose(id, |x:i16| x + 5);
        for i in i16::MIN..i16::MAX-5 {
            assert!(i + 5 == foo(i));
        }
    }
}
```

Challenge 4 (1.4.4)

Is the world-wide web a category in any sense? Are links morphisms?

With links as morphisms the world-wide web isn't a category. This is because wo don't have composition for links. If website A has a link to website B and B a link to website C there isn't nesecery a link from A to C. Also not every website has a Link to itself, so also the identity morphism isn't provided.
We can still make it into a category if we don't use links as morphisms but the transitive reflexive closure of those links. like this we get a reachability graph for websites wich is a category.

Challenge 5 (1.4.5)

Is Facebook a category, with people as objects and friendships as morphisms?

No because we don't have composition. if A and B ara friends and B and C are friends A and C don't have to be friends.

Challenge 6 (1.4.6)

When is a directed graph a category?

If it is closed under reflexivity and transitivity.

Challenges Chapter 2 (2.7)

Challenge 1 (2.7.1)

Define a higher-order function (or a function object) memoize in your favorite language. This function takes a pure function f as an argument and returns a function that behaves almost exactly like f, except that it only calls the original function once for every argument, stores the result internally, and subsequently returns this stored result every time it's called with the same argument. You can tell the memoized function from the original by watching its performance. For instance, try to memoize a function that takes a long time to evaluate. You'll have to wait for the result the first time you call it, but on subsequent calls, with the same argument, you should get the result immediately.

```rust
use std::{collections::HashMap, hash::Hash};

struct Memoized<F,A,B>
where
    F: Fn(&A)->B,
    A: Clone + Hash + Eq,
    B: Clone
{
    function : F,
    memory: HashMap<A,B>
}

impl<F,A,B> Memoized<F,A,B>
where
    F: Fn(&A)->B,
    A: Clone + Hash + Eq,
    B: Clone
{
    fn new(f:F) -> Memoized<F,A,B>{
        Memoized {
            function: f,
            memory: HashMap::new(),
        }
    }
    fn apply(&mut self, arg: &A)->B {
        if self.memory.contains_key(&arg) {
            return self.memory.get(&arg).unwrap().clone();
        } else {
            let result = (self.function)(arg);
            self.memory.insert(arg.clone(), result.clone());
            return result;
```

```
32          }
33      }
34  }
```

Challenge 2 (2.7.2)

Try to memoize a function from your standard library that you normally use to produce random numbers. Does it work?

This wouldn't work, because a random number generator isn't a pure function. If memoized it would generate the first number random but then it would always return exactly this memoized random number.

Challenge 3 (2.7.3)

Most random number generators can be initialized with a seed. Implement a function that takes a seed, calls the random number generator with that seed, and returns the result. Memoize that function. Does it work?

This would work, because now we just memoize the seed for which a fixed random number generator is generated. The calls to the random number generator are not memoized, so they produce a new random number every time the function is called. The only artifact ist, that if we try to get a new random number generator with the same seed, we don't realy get a new one. We get the one that already was instantiated and therefore we don't get the random number sequence from the start but from where it is currently located.

Challenge 4 (2.7.4)

Which of these C++ functions are pure? Try to memoize them and observe what happens when you call them multiple times: memoized and not.

(a): pure

(b): impure, because it depends ond what the user types, so it gives a different value depending on user input not on function input

(c): pure in the sense of the return behaviour, but it has an effect an is therefore impure in this regard.

(d): impure because the static int y variable ist an inner state of the function. If we call it f(1) it returns 1 if we then call it again with the same argument f(1) it returns 2

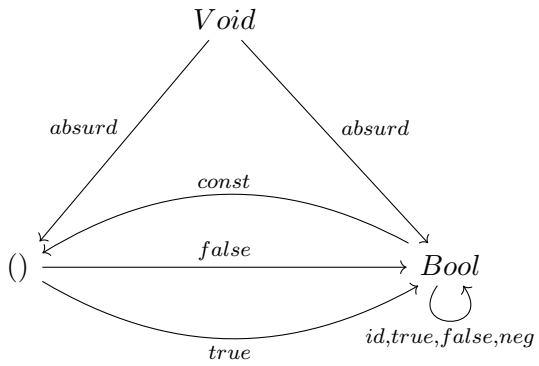

Figure 1: Category for Challenge 2.7.6

Challenge 5 (2.7.5)

How many different functions are there from Bool to Bool? Can you implement them all?

There are 4 different functions because the function can have two different values for a True input and two different values for a False input. All the function are

```
f1 b = if b then True else True
f2 b = if b then True else False
f3 b = if b then False else True
f4 b = if b then False else False
```

Challenge 6 (2.7.6)

Draw a picture of a category whose only objects are the types Void, () (unit), and Bool; with arrows corresponding to all possible functions between these types. Label the arrows with the names of the functions.

Challenges Chapter 3 (3.6)

Challenge 1 (3.6.1)

Generate a free category from:

(a) A graph with one node and no edges

The Category just consists of the one node as an object, and the added identity arrow.

(b) graph with one node and one (directed) edge (hint: this edge can be composed with itself)

The Category also has just the one node as an object with an added identity arrow. This time though we also add the arrows for arbitrary sequences of composing the one initial edge. So if e is our edge we get an arrow for e^n for $n \in \mathcal{N}$

(c) A graph with two nodes and a single arrow between them

The category just consists of the two nodes as objects with added identity arrows and the one arrow between them. There is no pair of arrows we can compose besides those with identity arrows, so we don't generate any new arrows.

(d) A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c ... z.

The category consists of the one node as an object and infinit many arrows labeled with all the possible strings over the alphabet. So for every $w \in \{a, ..., z\}^+$ we have an arrow. The Category can be seen as the category of string concatination. The identity arrow then would be the same as a ϵ arrow

Challenge 2 (3.6.2)

What kind of order is this?

(a) A set of sets with the inclusion relation: A is included in B if every element of A is also an element of B.

It's a partial order, its reflexive, transitive, and antisymetric. Also not all Sets have to be comparable for example neither $\{a\} \subseteq \{b\}$ nor $\{b\} \subseteq \{a\}$ holds

(b) C++ types with the following subtyping relation: T1 is a subtype of T2 if a pointer to T1 can be passed to a function that expects a pointer to T2 without triggering a compilation error.

It's a preorder, its reflexive (Pointer to T1 kann be passed to a function that expects a Pointer to T1), transitive (If *T1 kann be passed to *T2 and *T2 can be passed to *T3 tehn also *T1 can be passed to *T3), but I think it's not antisymetric but I don't know enough about the C++ Typesystem.

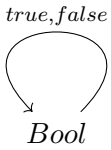

Figure 2: Category for Challenge 3.6.4

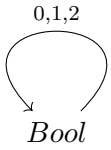

Figure 3: Category for Challenge 3.6.5

Challenge 3 (3.6.3)

Considering that Bool is a set of two values True and False, show that it forms two (set-theoretical) monoids with respect to, respectively, operator && (AND) and || (OR).

&& and || are associative and they always yield a Bool. For && the neutral element is 1 because $1\&\&x = x$ and $x\&\&1 = x$ and for || it is 0 because $0||x = x$ and $x||0 = 0$

Challenge 4 (3.6.4)

Represent the Bool monoid with the AND operator as a category: List the morphisms and their rules of composition.

The id morphism is *true* and the composition is $f \circ g = f\&\&g$

Challenge 5 (3.6.5)

Represent addition modulo 3 as a monoid category.

The id morphism is 0 and the composition is $f \circ g = (f + g) \bmod 3$

Challenges Chapter 4 (4.4)

A function that is not defined for all possible values of its argument is called a partial function. It's not really a function in the mathematical sense, so it doesn't fit the standard categorical mold. It can, however, be represented by a function that returns an embellished type optional:

Challenge 1

Construct the Kleisli category for partial functions (define composition and identity).

The identity Arrow just returns the Puts the Value inside the Option so in C++ Terms it constructs optional(A v). With Algebraic Datatypes we would write Some(v). the composition executes the first function if this returns an empty option the composition returns an empty option. Otherwise the value is passed inside the next Funktion. In rust That wpuld be

```rust
fn option_id<A>(a:A) -> Option<A>{Some(a)}

fn option_compose<'a,F,G,A,B,C>(f:F, g:G) -> Box<(dyn Fn(A)->Option<C> + 'a)>
where
F: Fn(A)->Option<B> + 'a,
G: Fn(B)->Option<C> + 'a,
{
    return Box::new(move |a| {
        match f(a) {
            None => None,
            Some(b) => g(b)
        }
    });
}
```

Challenge 2

Implement the embellished function safe_reciprocal that returns a valid reciprocal of its argument, if it's different from zero.

```rust
pub fn safe_reciprocal(x:f32) -> Option<f32> {
    if x == 0. {None} else {Some(1./x)}
}

pub fn safe_root(x:f32) -> Option<f32>{
```

```
    if x == 0. {None} else {Some(f32::sqrt(x))}
}
```

Challenge 3

Compose the functions safe_root and safe_reciprocal to implement safe_root_reciprocal that calculates $sqrt(1/x)$ whenever possible.

```
pub fn safe_root_reciprocal(x:f32) -> Option<f32>{
    (option_compose(safe_reciprocal, safe_root))(x)
}
```

Challenges Chapter 5 (5.8)

Challenge 1 (5.8.1)

Show that the terminal object is unique up to unique isomorphism.

Suppose the terminal Object c and another c'. because they both are terminal there are unique morphisms $f :: c- > c'$ and $g :: c'- > c$. Those have to compose so $f \circ g :: c' \to c'$ and $g \circ f :: c \to c$ have to exist. But because c and c' are terminal there is just one unique $c \to c$ and $c' \to c'$ and because those objects must have an identity arrow both of those have to be identity arrows. So we get $f \circ g = id_{c'}$ and $g \circ f = id_c$ so they are isomorphic.

Challenge 2 (5.8.2)

What is a product of two objects in a poset? Hint: Use the universal construction.

if it exists it is their infimum. Suppose the poset Category with $a \leq b$ as morphism $a \to b$ if c is the product of a and b it must hold $c \leq a$ and $c \leq b$ so c is smaller then both of them. Additionaly for every other c' with $c' \leq a$ and $c' \leq b$ there is $c' \leq c$. Together it's the definition of the Infimum.

Challenge 3 (5.8.3)

What is a copruduct of two objects in a poset?

It's the supremum. Suppose the poset Category with $a \leq b$ as morphism $a \to b$ if c is the coproduct of a and b it must hold $a \leq c$ and $b \leq c$ so c is bigger then both of them. Additionaly for every other c' with $a \leq c'$ and $b \leq c'$ there is $c \leq c'$. Together it's the definition of the supremum.

Challenge 4 (5.8.4)

Implement the equivalent of Haskell Either as a generic type in your favorite language (other than Haskell).

```
enum Either<L,R> {
    Left(L),
    Right(R),
}
```

Challenge 5 (5.8.5)

Show that Either is a 'better' coproduct than int two injections:

```
i :: Integer -> Bool
i x = x
j :: Bool -> Integer
j b = if b then 0 else 1
```

Hint: Define a function

```
m :: Either Integer Bool -> Integer
```

that factorizes i and j

```
m ::Either Integer Bool -> Integer
m (Left x) = x
m (Right b) = if b then 0 else 1
```

Es gilt nun

```
m (Left x) = x = i x
m (Right b) = if b then 0 else 1 = j b
```

and Left and Right are the two injections for Either. Also we can argue informaly, that this m is unique because given a (Left x) it has to give x back to factorize i because i acts as identity and for a (Right b) there are finitely many Options and this is the only one that factorizes j

Challenge 6 (5.8.6)

Continuing the previous problem: How would you argue that int with the two injections i and j cannot be 'better' than Either?

If we look at i and j we see, that `i 0 = j true` and `i 1 = j false` for every factorization `m :: Integer -> Either Integer Bool` 0 is mapped to (Left 0) or (Right true) to factorize. But suppose it is mapped to (Left 0) then (m (j true) = Left 0) wich is not (Right true). It's analogous for the other cases and so there is no factorization.

Challenge 7 (5.8.7)

Still continuing: What about these injections?

```haskell
i :: Integer -> Integer
i x = if (x < 0) then x else x + 2

j :: Bool -> Integer
j b = if b then 0 else 1
```

Thats an equaly legit coproduct. Because of the factorization with

```haskell
m :: Integer -> Either Integer Bool
m 0 = Right True
m 1 = Right False
m x = if x < 0 then Left x else Left (x - 2)
```

Challenge 8 (5.8.8)

Come up with an inferior candidate for a coproduct of int and bool that cannot be better than Either because it allows multiple acceptable morphisms from it to Either.

```haskell
i :: Integer -> Either Integer (Bool, Bool)
i x = Left x

j :: Bool -> Either Integer (Bool, Bool)
j b = Right (b, b)

m :: Either Integer (Bool, Bool) -> Either Integer Bool
m (Left x) = Left x
m (Right (b1,b2)) = Right b1
```

```haskell

m' :: Either Integer (Bool, Bool) -> Either Integer Bool
m' (Left x) = Left x
m' (Right (b1,b2)) = Right b2
```

both m and m' factorize Left and Right but are not the same because (m (Right (true, false)) = true) but (m' (Right (true, false)) = false)

Challenged Chapter 6 (6.5)

Challenge 1 (6.5.1)

Show the isomorphism between Maybe a and Either () a.

```haskell
maybeToEither :: Maybe a -> Either () a
maybeToEither None = Left ()
maybeToEither (Just a) = Right a

eitherToMaybe :: Either () a -> Maybe a
eitherToMaybe (Left ()) = None
eitherToMaybe (Right a) = Just a
```

That this is an isomorphism follows from a simple case analysis.

Challenge 2 & 3 & 4 (6.5.2&3&4)

2 Here's a sum type defined in Haskell:

```haskell
data Shape = Circle Float | Rec Float Float
```

When we want to define a function like area that acts on a Shape, we do it by pattern matching on the two constructors:

```haskell
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rect d h) = d * h
```

Implement Shape in C++ or Java as an interface and create two classes: Circle and Rect. Implement area as a virtual function.

3 Continuing with the previous example: We can easily add a new function circ that calculates the circumference of a Shape. We can do it without touching the definition of Shape:

```haskell
circ :: Shape -> Float
circ (Circle r) = 2.0 * pi * r
circ (Rect d h) = 2.0 * (d + h)
```

Add circ to your C++ or Java implementation. What parts of the original code did you have to touch?

We had to touch the Definition of the Datatyps (the classes) because in OOP behaviour and Data are coupled

4 Continuing further: Add a new shape, Square, to Shape and make all the necessary updates. What code did you have to touch in Haskell vs. C++ or Java? (Even if you're not a Haskell programmer, the modifications should be pretty obvious.)

We had to write a new class in c++ with all the virtual methods. In Haskell we would need to list a new constructor to Shape and handle this case in all the implemented functions for Shape

```cpp
// #define _USE_MATH_DEFINES
// #include <cmath>
// class Shape {
//     virtual float area() = 0;
//     virtual float circ() = 0;
// }
// class Circle : Shape{
//     float r;
//     virtual float area(){ return this.r * this.r * M_PI; }
//     virtual float circ(){ return 2.0 * M_PI * this.r; }
// }
// class Rect : Shape{
//     float d;
//     float h;
//     virtual float area(){ return this.d * this.h; }
//     virtual float circ(){ return 2.0 * (d + h); }
// }
// class Square : Shape{
//     float d;
//     virtual float area(){ return this.d * this.d; }
//     virtual float circ(){ return 2.0 * (this.d + this.d); }
// }
```

Challenge 5 (6.5.5)

> Show that a + a = 2 * a holds for types (up to isomorphism). Remember that 2 corresponds to Bool, according to our translation table.

```haskell
m :: Either a a -> (Bool, a)
m (Left a) = (True, a)
m (Right a) = (False, a)

m' :: (Bool, a) -> Either a a
m' (True, a) = Left a
m' (False, a) = Right a
```

Challenges Chapter 7 (7.8)

Challenge 1 (7.8.1)

> Can we turn the Maybe type constructor into a functor by defining:

```haskell
fmap x y = Nothing
```

> which ignores both of its arguments? (Hint: Check the functor laws.)

No because it dosn't map id to id because (fmap id X = Nothing) but (id (Just x) = Just x).

Challenge 2 (7.8.2)

> Prove functor laws for the reader functor. Hint: it's realy simple

Given some Type a (-> r a) is also a Type so (-> r) maps objects. Given the id morphism fmap of (-> r) should produce the id morphism of (-> r a)

```haskell
(fmap id) f = id . f = f
```

Given the morphisms (f :: a -> b) (g :: b -> c) and their composition (f . g :: a -> c) it must hold ((fmap f) . (fmap g) = fmap (f . g))

```haskell
f :: a -> b
g :: b -> c
((fmap f) . (fmap g)) h = (fmap f) (fmap g h) = (fmap f) (g . h) = f . (g . h) = (f . g) . h = fmap (f . g
```

Challenge 3 (7.8.3)

Implement the reader functor in your second favorit language (the first being Haskell, of course).

```rust
pub trait Functor<'a>
where Self :'a {
    type Unpluged : 'a ;
    type Plug<B:'a>: Functor<'a> + 'a;

    fn fmap<F:'a, B:'a>(self, f: F) -> Self::Plug<B>
    where
        F: Fn(Self::Unpluged) -> B + 'a;
}

impl<'a,R:'a,A:'a> Functor<'a> for Box<dyn Fn(R) -> A + 'a> {
    type Plug<B:'a> = Box<dyn Fn(R)->B + 'a>;
    type Unpluged = A;
    fn fmap<F:'a, B:'a>(self, f: F) -> Self::Plug<B>
        where
            F: Fn(Self::Unpluged) -> B + 'a {
        Box::new(move |r| f(self(r)))
    }
}
```

Challenge 4 (7.8.4)

Prove the functor laws for the list functor. Assume that the laws are true for the tail part of the list you're applying it to (in other words, use induction)

```haskell
fmap id (x:xs) = (id x) : (fmap xs) = x : (fmap xs) = x : xs

((fmap f) . (fmap g)) (x:xs) = (fmap f) (fmap g (x:xs)) = (fmap f) ( g x : fmap g xs) = (f (g x) : fmap f
```

Challenges Chapter 8 (8.9)

Challenge 1 (8.9.1)

Show that the data type:

```haskell
data Pair a b = Pair a b
```

is a bifunctor. For additional creditimplement all three methods of Bifunctor and use equational reasoning to show that these definitions are compatible with the default implementations whenever they can be applied.

```haskell
data Pair a b = Pair a b

instance Bifunctor Pair where
    bimap f g (Pair a b) = Pair (f a) (g b)
    first f (Pair a b) = Pair (f a) b
    second f (Pair a b) = Pair a (f b)
```

Challenge 2 (8.9.2)

Show the isomorphism between the standard definition of Maybe an this desugaring:

```haskell
type Maybe' a = Either (Const () a) (Identity a)
```

Hint: Define two mappings between the two implementations. For additional credit, show that they are the inverse of each other using equational reasoning.

```haskell
type Maybe' a = Either  (Const () a) (Identity a)

isom :: Maybe a -> Maybe' a
isom Nothing = Left $ Const ()
isom (Just a) = Right $ Identity a

isom' :: Maybe' a -> Maybe a
isom' (Left (Const ())) = Nothing
isom' (Right (Identity a)) = Just a
```

Challenge 3 (8.9.3)

Lets's try another data structure. I call it a PreList because it's a precursor to a List. It replaces recursion with a type parameter b.

```haskell
data PreList a b = Nil | Cons a b
```

You could recover our erlier definition of a List by recursively applying PreList to itself (we'll see how it's done when we talk about fixed points). Show that PreList is an instance of Bifunctor

```haskell
data PreList a b = Nil | Cons a b
instance Bifunctor PreList where
    bimap f g Nil = Nil
    bimap f g (Cons a b) = Cons (f a) (g b)
    first f Nil = Nil
    first f (Cons a b) = Cons (f a) b
    second g Nil = Nil
    second g (Cons a b) = Cons a (g b)
```

Challenge 4 (8.9.4)

Show that the following data types define bifunctors in a and b:

```haskell
data K2 c a b = K2 c
data Fst a b = Fst a
data Snd a b = Snd b
```

For additional credit, check your solutions against Conor McBride's paper Clowns to the Left of me, Jokers to the Right1 .

```haskell
newtype K2 c a b = K2 c
instance Bifunctor (K2 c) where
    bimap _ _ (K2 c) = K2 c

newtype Fst a b = Fst a
instance Bifunctor Fst where
    bimap f _ (Fst a) = Fst (f a)

newtype Snd a b = Snd b
instance Bifunctor Snd where
    bimap _ g (Snd b) = Snd $ g b
```

Challenge 5 (8.9.5)

Define a bifunctor in a language other than Haskell. Implement bimap for a generic pair in that language.

```rust
pub trait BiFunctor<'a>
where Self : 'a {
    type Unpluged1: 'a;
    type Unpluged2: 'a;
    type Plug<B1:'a,B2:'a>: BiFunctor<'a> + 'a;
    fn bimap<F1:'a, F2:'a, B1:'a, B2:'a>(self, f1:F1, f2:F2) -> Self::Plug<B1,B2>
    where
        F1: Fn(Self::Unpluged1) -> B1 + 'a,
        F2: Fn(Self::Unpluged2) -> B2 + 'a;
}

impl<'a,A1:'a,A2:'a> BiFunctor<'a> for (A1,A2) {
    type Plug<B1:'a,B2:'a> = (B1,B2);
    type Unpluged1 = A1;
    type Unpluged2 = A2;
    fn bimap<F1:'a, F2:'a, B1:'a, B2:'a>(self, f1:F1, f2:F2) -> Self::Plug<B1,B2>
        where
            F1: Fn(Self::Unpluged1) -> B1 + 'a,
            F2: Fn(Self::Unpluged2) -> B2 + 'a {
        (f1(self.0), f2(self.1))
    }
}
```