```haskell
{-|
Module      : GPACalculator
Author      : COMP1100/COMP1130 Team, Your name and UID here
Date        :
Description : This module contains functions to calculate GPA from grades and marks.
-}

module GPACalculator where

data Grade = Fail | Pass | Credit | Distinction | HighDistinction
   deriving (Show,Eq)

type GPA = Double
type Mark = Double

data Course = Art Int -- Int denotes the course code
            | Comp Int
            | Busn Int
            | Math Int


-- | Exercise 3
-- Convert marks to grade

markToGrade :: Mark -> Grade
markToGrade mark
  | mark < 0 || mark > 100 = error "markToGrade: Not a valid mark"
  | mark >= 80             = HighDistinction
  | mark >= 70             = Distinction
  | mark >= 60             = Credit
  | mark >= 50             = Pass
  | otherwise              = Fail
```

```haskell
-- | Exercise 4
-- Your comment here
markToGrade2 :: (Course, Mark) -> Grade
markToGrade2 = Art Int
            | Comp Int
            | Busn Int
            | Math Int
-- | Exercise 5
-- Your comment here
markToGradeSafe ::  Mark -> Maybe Grade
markToGradeSafe mark
  | mark < 0 || mark > 100 = Nothing
  | mark >= 80 = Just HighDistinction
  | mark >= 70 = Just Distinction
  | mark >= 60 = Just Credit
  | mark >= 50 = Just Pass
  | otherwise  = Just Fail

-- | Exercise 6
-- Your comment here
maybeGradeToGPA :: Maybe Grade -> GPA
maybeGradeToGPA = undefined

-- | Exercise 7
-- Your comment here
markToGPA :: Mark -> GPA
markToGPA = undefined

-- | Exercise 8
-- Your comment here
markToGradeScaled :: (Course,Mark) -> Grade
markToGradeScaled = undefined
```

# GPA Calculator

## Exercise 3

Finding your grade at the ANU from your raw mark isn't a smooth, continuous function, but it has discrete output values. Depending on your mark, your grade will be calculated as:

```
data Grade  = Fail | Pass | Credit | Distin
  deriving Show

type Mark = Int

markToGrade :: Mark -> Grade
markToGrade mark
  | mark >= 80 && mark <= 100 = HighDistinc
  | mark >= 70 && mark < 80 = Distinction
  | mark >= 60 && mark < 70 = Credit
  | mark >= 50 && mark < 60 = Pass
  | mark >=  0 && mark < 50 = Fail
```

### Task 1

Open `GPACalculator.hs` and then load the module into GHCi by executing the command `ghci -Wall GPACalculator.hs` in the Terminal. You will get the following warning:

```
GPACalculator.hs:25:1: warning: [-Wincompl
    Pattern match(es) are non-exhaustive
    In an equation for 'markToGrade': Patt
```

Try out:

```
ghci> markToGrade 116
...
ghci> markToGrade (-16)
```



Since the mark is an `Int`, numbers below 0 and above 100 are also valid inputs, but not valid as a mark. Your task is to add a guard that returns an error message "**markToGrade: Not a valid mark**" if the input *mark is strictly less than 0* OR *strictly greater than 100*.

Hint: Have a look at the `isEven` function previously for the guards syntax, and for returning an error, see the lecture material on Conditional Expressions in Week 2.

Once the task is completed, reload the module, and try evaluating `markToGrade 116` again. You should see the following:

```
ghci> markToGrade 116
*** Exception: markToGrade: Not a valid mark
CallStack (from HasCallStack):
  error, called at GPACalculator.hs:20:32 in main:GPACalculatc
```

### Task 2

If you reload the module with the flag `-Wall`, you may get the following warning (if not, you can just read through this section, since you've already fixed it):

```
GPACalculator.hs:14:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for 'markToGrade': Patterns not matched: _
```

What does this mean? Can we modify our guarded expression to remove this warning?

Haskell is a language that takes code correctness and reliability very seriously. As such, it will give warnings for functions which it believes do not have a definition for *every* possible valid input (i.e., every possible value of the input(s) type). Even though it is likely that you have matched every possible Integer input to this function, Haskell isn't *sure*, and so it gives the above warning. To satisfy Haskell, we typically will end a set of guards with an `otherwise` — upon reaching an `otherwise` case as the last line (when evaluating guard conditions from top to bottom) Haskell will **always** evaluate the function as the definition on the right hand side of the `otherwise`.

**This is not a test.** Ask your tutor or your classmate if you're confused.

Submission required: `Exercise 3`.

## Exercise 4

Recall from Lab 2, a *tuple* is a collection of elements, possibly of different types. A *2-tuple* is called a pair of elements, a *3-tuple* is a triple of elements, a *4-tuple* is a grouping of 4 elements, in general an *N-tuple* is a grouping of *n* elements for some *n*.

Tuples are useful when we want to group information together. In Haskell we represent tuple types with *parentheses*. For example, consider your overall grade for a particular course, we can represent this as a pair of a `Course` and a positive Integer like so: `(Comp 1100, 85)` or `(Busn 1001, 97)`. The tuple can be thought of as the Haskell representation of the *product type*.

Your task is to write a function `markToGrade'` that takes an input value of type `(Course, Mark)` and returns a `Grade`.

You can define `Course` as

```
data Course = Art Int -- Int denoted the course code
            | Comp Int
            | Busn Int
            | Math Int
```

As before, `Mark` is another name for `Int`, and `Grade` is as defined in Exercise 3.

Example outputs:

```
ghci> markToGrade' (Comp 1100, 45)
Fail
ghci> markToGrade' (Busn 1001, 95)
HighDistinction
```

**Hint: This function can be written in a single line.**

**Hint: This function can be written in a single line.**

What happens when we enter an invalid mark as input?

Submission required: `Exercise 4`

## Exercise 5

There are several other ways to handle exceptional values or situations which you will learn about. One way is to use types that can also hold some exceptional values "on top of" their "normal" values. The simplest of these is the **Maybe type**, which can hold values of a specific type or one special value (which is defined as the value **Nothing** here). This way we can write function which will return (or accept) values that are of a specific type, but also have the option to say they cannot return anything and actually returns the value **Nothing** instead.

For example, supposing you write a function to divide numbers. What should the function return if we divide by zero? It doesn't seem sensible to return any number at all (except perhaps the value infinity), and throwing an error and killing the program also may not be ideal (imagine if this software is running on an aeroplane, and division by zero occurs, so shutting down the plane's control computer in mid-flight — not ideal!) Returning `Nothing` instead is a more graceful way to handle the situation where we would not otherwise be able to return a value.

After you have read the above you will probably notice that the name "Nothing" is perhaps not the most sensible choice of name, as it is not literally "nothing", but in fact a special value. But, planet Haskell has defined it thus, so we are stuck with it here.

Consequently, the **Maybe Type** as already pre-defined in Haskell looks like this:

```
data Maybe a = Nothing | Just a
```

where `a` stands for "any type". We can now write another version of our grading program.

Your task is to use the `Maybe` type to stop the `markToGrade` function throwing an error when the input mark is out of bounds. Write function `markToGradeSafe` with the type signature:

```
markToGradeSafe :: Mark -> Maybe Grade
```

Note that when the mark is within bounds, instead of a value `X` of the type `Grade`, we now return `Just X`, which is of type `Maybe Grade`. When the mark is out of bounds, we now return `Nothing`. Test your implementation of `markToGradeSafe` and see what GHCi displays as return values. Other numerical types use a similar concept. For instance, try to divide 1 by 0 in GHCi, or take the square root of a negative number, and see if the output is what you expect:

```
ghci> 1 / 0
...
ghci> sqrt (-1)
```

See the following example for how you might write a safe division function using a Maybe type (maybe use something like this to write `markToGradeSafe`):

```
safeDiv :: Double -> Double -> Maybe Double
safeDiv x y
    | y == 0 = Nothing
    | otherwise = Just (x / y)
```

Submission required: `Exercise 5`.

## Exercise 6

Here's another problem. At some point, you will want to figure out what GPA corresponds to which Grade. To simplify things, here's a table modified from ANU GPA reference.

| Grade | Grade Point Value |
|---|---|
| HighDistinction | 7 |
| Distinction | 6 |
| Credit | 5 |
| Pass | 4 |
| Fail | 0 |

It would be handy if you could enter your mark, and it automatically calculated the GPA associated with that mark. For example, `markToGPA 80` would return 7 as the result.

Let's break this down to two steps. We have already written the part where you convert your mark to a grade. The next step is to write another function to convert grades to a GPA. Therefore the new function should have following type signature (where `GPA` is another name for `Double`): `Maybe Grade -> GPA`

You have already learnt that the `Maybe` type is defined as follows:

```
data Maybe a = Just a | Nothing
```

How do you actually use this productively? Let's start with an example:

```
maybeToInt :: Maybe Int -> Int
maybeToInt m = case m of
    Just i  -> i
    Nothing -> 0
```

The example above takes in a `Maybe Int` which is either `Just Int` or `Nothing`. If it follows the pattern of `Just <some Int value i>`, return `i`, if it's `Nothing`, return `0`.

You can see we've used pattern matching here to "look inside" the input, and extract out the number contained inside the `Maybe` type (if such a number exists). Have a closer look at the third line of this example — what is the type of `i`?

Your task is to add a new function to `GPACalculator.hs`:

```
maybeGradeToGPA :: Maybe Grade -> GPA
```

Note, if the Maybe Grade is `Nothing`, return `0`.

⚠ Submission required: `Exercise 6`

## Exercise 7

Let's link all of these things together! First, let's recall the functions you've already written and are going to use:

- `markToGradeSafe :: Mark -> Maybe Grade`: This takes a raw `Mark` and outputs a `Maybe Grade`.
- `maybeGradeToGPA :: Maybe Grade -> GPA`: This takes a `Maybe Grade` and outputs a `GPA` associated with that grade.

What's the next step?

Your task is to write a function: `markToGPA :: Mark -> GPA`, (using the functions already written to help you) which takes in a raw mark as an `Int` in the range 0 to 100 and outputs a GPA (0 to 7) associated with that mark.

⚠ Submission required: `Exercise 7`.

## Nesting guards and cases

A useful property of cases and guards is being able to nest them (i.e. use cases within cases). Remember how we defined the data type `Shape` by combining sum and product types.

Let us define a valid circle as a `Circle` with non-negative radius, and a valid rectangle as a `Rectangle` with non-negative length and width.

```
data Validity = Valid | Invalid
```

Using this let us define a function that takes a `Shape` and returns the `Validity` of that shape.

Using this let us define a function that takes a `Shape` and returns the `Validity` of that shape.

```
validShape :: Shape -> Validity
validShape shape = case shape of
    Circle rad
        | rad < 0 -> Invalid
        | otherwise -> Valid
    Rectangle h w
        | h < 0 || w < 0 -> Invalid
        | otherwise -> Valid
```

In this example we have used a guard within a case. The case checks what the shape is and the guard considers the criteria for the shape to be valid.

ℹ Note how the nested guard uses an `->` and not a `=`

## Exercise 8 (Tricky)

Using the above concept, we will be trying to remake the function `markToGrade'`. This time we are given the additional information that 1000 level `Busn` courses will be scaled down by 10% (i.e. multiplied by 0.90) and all other `Busn` courses will be scaled down by 5% (i.e. multiplied by 95%). All the other courses calculate the grade without scaling.

Your task is to write a function: `markToGradeScaled` that outputs a `Grade` according to the above rules for scaling.

Example outputs:

```
ghci> markToGradeScaled (Busn 1101, 77)
Credit
ghci> markToGradeScaled (Busn 3001, 62)
Pass
ghci> markToGradeScaled (Math 1115, 81)
HighDistinction
```

⚠ Submission required: `Exercise 8`.

描述

# Testing your code with Doctest

## Exercise 9

Just because your code compiles, it doesn't mean it will do what it is supposed to. Consider the following:

```
addOne :: Int -> Int
addOne x = x + 50
```

Does this compile? Of course it does! Does this do what it's supposed to? Deducing from the name, it should take a number, and add one to it. However, what it actually does is to take a number and add 50 to it. For a simple function like this, it's relatively easy to figure out that it's wrong before even compiling it. However, for a more complicated function it is often necessary to compile it and run the code with a pre-determined input and expected output, to see if it does what you want it to.

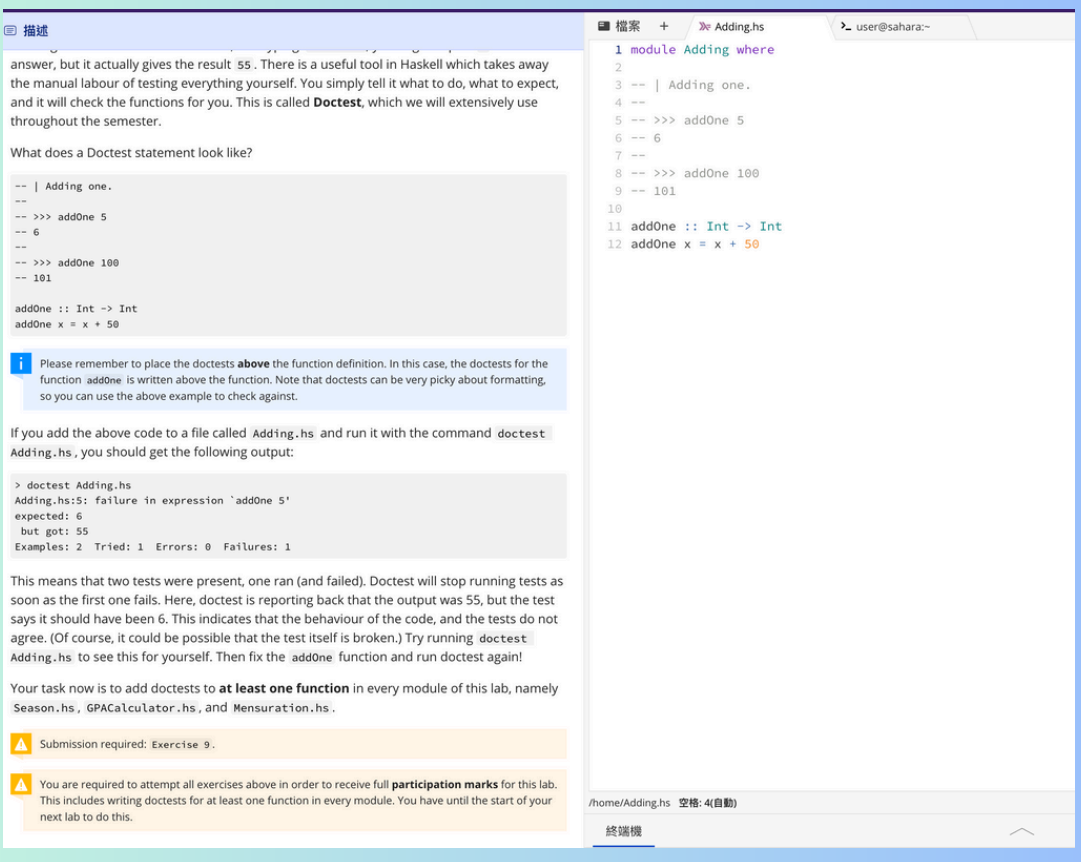
Loading the above function into GHCI, and typing: `addOne 5`, you might expect `6` to be the answer, but it actually gives the result `55`. There is a useful tool in Haskell which takes away the manual labour of testing everything yourself. You simply tell it what to do, what to expect, and it will check the functions for you. This is called **Doctest**, which we will extensively use throughout the semester.

What does a Doctest statement look like?

```
-- | Adding one.
--
-- >>> addOne 5
-- 6
--
-- >>> addOne 100
-- 101

addOne :: Int -> Int
addOne x = x + 50
```

Please remember to place the doctests **above** the function definition. In this case, the doctests for the function `addOne` is written above the function. Note that doctests can be very picky about formatting, so you can use the above example to check against.



描述

# [Optional] Area of a triangle done properly

**[Optional exercise]**

The following additional exercises are optional but encouraged if you have time. These exercises should only be attempted after completing all the exercises above. Feel free to do this at home and discuss with your peers on Ed, or with tutors at a drop-in session.

## Exercise 10: Is the triangle even valid?

If you look back at last week's lab, you will now see that the function `areaTriangle` could not have been the last word. This function should not return an area value if the provided edge lengths cannot form a triangle. So as an intermediate step we will first write a function `isValidTriangle` that will tell you whether the sum of any two sides of the triangle is larger than the remaining side.

Copy your work from the `Area.hs` file from Lab 2 to the current file. You can reuse some of that code in the next task and fix the definition of `areaTriangle` using `isValidTriangle`.

The first challenge is to work out what the **type** of the function should be (Hint: remember the output of the function can be either `True` or `False`). Write a type signature for the function.

Next write the actual function definition, which can well be a single line, or alternatively a sequence of guarded expressions. Test your function with some values in GHCi before you proceed to the final exercise. For instance, $\{1, 2, 4\}$ is a set of edge lengths which cannot form a triangle. What does your function make of it?

### Area of a valid triangle

You now have all the tools at hand to do the job properly. Your new function `areaTriangleSafe` has the type:

```
areaTriangleSafe :: Double -> Double -> Double -> Maybe Double
```

It will return a numerical value as `Just <Double value>` for all valid triangles and `Nothing` for invalid triangles. Put all the bits from this lab together for this function.

Write your function and a few doctests for both valid and invalid triangles to justify your function.

Submission for this task is optional. Make sure to submit all previous exercises for participation marks.

檔案 + Area.hs

```
1
```

/home/Area.hs 空格: 4(自動)

終端機