Warm up: Algebraic Datatypes

Haskell is famous for its type system, its type checker and its strongly statically-typed compilation process. However, up until now you've only encountered predefined types. Using *algebraic data types* it is possible to define new types yourself.

Defining Algebraic Datatypes

A newly created type has to be *defined* by specifying all possible *(data) constructors*. Each constructor is a function that can be used to create a value of this type. The different constructors are separated by the `|` symbol. The `deriving`-clause is optional. Syntactically, this is done in the following manner:

```
data TypeName = Constructor1 ArgType1 ArgType2 ...
              | Constructor2 ArgType1 ArgType2 ...
              | ...
              | ConstructorN ArgType1 ArgType2 ...
```

For example, a boolean can be either true or false:

```
data Bool = True | False
```

Note: to avoid confusion, we advise you to always pick a different name for the constructor than for the data type. For example:
BAD: `data Age = Age Int`
GOOD: `data Age = MkAge Int`

Define algebraic datatypes (ADTs) to represent the following concepts:

- `Name`: a name is just a `String`.
- `Pair`: a pair consists of two integers (`Int`).
- `Gender`: a gender is either male, female, or other.
- `Person`: a person consists of a name (`Name`), an age (`Int`), and a gender (`Gender`).
- `TestResult`: a result of a test is either a *pass*, along with a grade (`Int`) or a *fail*, along with a list of comments from the teacher. You can use a `String` to represent a comment.

Don't forget to add "`deriving (Show)`" at the end of the datatype definition! The error "No instance for (Show ...) arising from ..." means that you have forgotten to add it.

Using Algebraic Datatypes

- Write a function `stringToGender :: String -> Gender` that returns the correct gender for the given string. If the string is "Male" or "Female" (correctly capitalised), the right constructor of `Gender` should be picked. All other strings are considered to be "Other".
- Write a function `genderToString :: Gender -> String` that converts a gender to a string: "Male", "Female", or "Other".

Examples

```
Main> genderToString (stringToGender "Male")
"Male"
Main> genderToString (stringToGender "Hamster")
"Other"
```

- Write a function `passing :: Int -> TestResult` that creates a passing `TestResult` with the given grade.
- Write a function `failing :: [String] -> TestResult` that creates a failed `TestResult` with the given comments.
- Write a function `grade :: TestResult -> Int` that returns the grade of a `TestResult`. A fail results in 0.
- Write a function `comments :: TestResult -> [String]` that returns the comments of a `TestResult`. A passing result has no comments.

Examples

```
Main> grade (passing 10)
10
Main> grade (failing ["Incorrect datatype syntax"])
0
Main> comments (passing 10)
[]
Main> comments (failing ["Incorrect datatype syntax"])
["Incorrect datatype syntax"]
```