1 Representing Complex Data

1.1 Building Data Types

There are three ways to build complex types/values:

- **Product Types** (each-of): a value of `T` contains a value of `T1` and a value of `T2`.
- **Sum Type** (one-of): a value of `T` contains a value of `T1` *or* a value of `T2`.

1.1.1 Product Type

Suppose we wanted to represent a date as a tuple of `Int`s.

```
deadlineDate :: (Int, Int, Int)
deadlineDate = (4, 29, 2022)

deadlineTime :: (Int, Int, Int)
deadlineTime = (11, 59, 59)

-- Deadline date extended by one day
extendedDate :: (Int, Int, Int) -> (Int, Int, Int)
```

There are a few issues here.

- This is verbose and unreadable.

  A **type synonym** for `T` is a name that can be used interchangeably with `T`. For example,

  ```
  type Date = (Int, Int, Int)
  type Time = (Int, Int, Int)

  deadlineDate :: Date
  deadlineDate = (4, 29, 2022)

  deadlineTime :: Time
  deadlineTime = (11, 59, 59)

  -- | Deadline date extended by one day
  extendedDate :: Date -> Date
  ```

- You can put the time into the `extendedDate` function.

  We want `extendedDate deadlineTime` to fail at compile-time. A solution is to construct two different **datatypes**.

  ```
  data Date = Date Int Int Int
  data Time = Time Int Int Int
  -- constructor ^ ^^^^^^^^^^^ parameter types
  deadlineDate :: Date
  deadlineDate = Date 4 29 2022

  deadlineTime :: Time
  deadlineTime = Time 11 59 59
  ```

(Quiz.) Consider the following datatype.

```
data Date = Date Int Int Int
```

What would GHCi say to the following?

```
>:t Date 4 29 2022
```

(a) Syntax error.

(b) Type error.

(c) `(Int, Int, Int)`

(d) `Date`

(e) `Date Int Int Int`

The answer is **D**.

1.1.2 Record Syntax

Consider the following datatype:

```
data Date = Date Int Int Int
```

It might be hard for us to tell what each `Int` means. So, we can use Haskell's **record syntax** to name the constructor parameters:

```
data Date = Date {
    month   :: Int,
    day     :: Int,
    year    :: Int
}
```

Then, we can do:

```
deadlineDate = Date { month = 4, date = 29, year = 2022 }
```

To extract a field value, we can treat the name as a function:

```
deadlineMonth = month deadlineDate
```

1.1.3 Sum Type

Suppose I want to represent a *text document* with simple markup. Each paragraph is either

- Plain text (`String`).
- Heading: level and text (`Int` and `String`).
- List: whether it is ordered and items (`Bool` and `[String]`).

Now, let's suppose we store all paragraphs in a list:

```
doc = [
    (1, "Notes from 130"),                                    -- Level 1 heading
    "There are two types of languages:",                      -- Plain text
    (True, ["those people complain about", "those no one uses"])  -- Ordered list
]
```

This won't work because lists need to have one type only. The solution is to use **sum types** – construct a new type that is a sum of (one-of) the three operations:

```
data Paragraph = PText String
    | PHeading Int String
    | PList Bool [String]
```

(Quiz.) Consider the following datatype:

```
data Paragraph = PText String | PHeading Int String | PList Bool [String]
```

What would GHCi say to

```
>:t PText "Hey there!"
```

(a) Syntax error

(b) Type error

(c) `PText`

(d) `String`

(e) `Paragraph`

Tha answer is **E**. Here, the type of `PText` is

```
PText :: String -> Paragraph
```

So, to construct a sum type, we use the notation

```
data T = C1 T11 .. T1k
    | C2 T21 .. T2l
    | ..
    | Cn Tn1 .. Tnm
```

Here, `T` is the new datatype and `C1 .. Cn` are the constructors of `T`. A value of type `T` is either

- `C1 v1 .. vk` with `v1 :: T1i`
- or `C2 v1 .. vl` with `vi :: T2i`
- ...
- or `Cn v1 .. vm` with `vi :: Tni`

How would we actually interpret a sum type? Using pattern matching!

```
html :: Paragraph -> String
html (PText str)        = ...
html (PHeading lvl str) = ...
html (PList ord items)  = ...
```

1.1.4 Dangers of Pattern Matching

- Consider the following:

```
html :: Paragraph -> String
html (PText str)        = ...
html (PList ord items)  = ...
```

1.1.5 Pattern Matching Syntax

We can also use pattern-matching in a program using the `case` expression. For example,

```
html :: Paragraph -> String
html p =
    case p of
        PText str           -> ...
        PHeading lvl str    -> ...
        PList ord items     -> ...
```

(Quiz.) Consider the following datatype:

```
data Paragraph = PText String | PHeading Int String | PList Bool [String]
```

What is the type of the following?

```
case PText "Hey there!" of
    PText _      -> 1
    PHeading _ _ -> 2
    PList _ _    -> 3
```

(a) Syntax error

(b) Type error

(c) `Paragraph`

(d) `Int`

(e) `Paragraph -> Int`

The answer is **D**. Here, we passed in an value `PText "Hey there!"`, which matches with the first branch and returns `1`.

(Quiz.) Consider the following datatype:

```
data Paragraph = PText String | PHeading Int String | PList Bool [String]
```

What is the type of the following?

```
case PText "Hey there!" of
    PText str      -> str
    PHeading lvl _ -> lvl
    PList ord _    -> ord
```

(a) Syntax error

(b) Type error

(c) `String`

(d) `Paragraph`

(e) `Paragraph -> String`

The answer is **B**. The `case` expression takes in a `Paragraph` and appear to return either a `String` (the first branch) or an `Int` (the second branch) or a `Bool` (the third branch). However, it is required that the return type is the same across all branches.