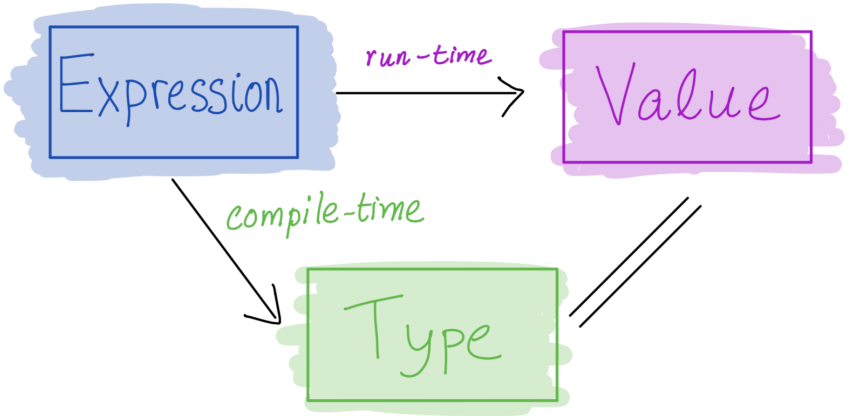
# *Datatypes and Recursion*

## *Plan for this week*

**Last week:**

- built-in *data types*
  - base types, tuples, lists (and strings)
- writing functions using *pattern matching* and *recursion*

**This week:**




- user-defined *data types*
    - and how to manipulate them using *pattern matching* and *recursion*
- more details about *recursion*

# *Representing complex data*

**We've seen:**

- *base* types: `Bool`, `Int`, `Integer`, `Float`
- some ways to *build up* types: given types `T1, T2`
  - functions: `T1 -> T2`
  - tuples: `(T1, T2)`
  - lists: `[T1]`

$T = Int \mid Bool \mid Char$
$\mid (T, T)$
$\mid (T_1, \ldots, T_k)$
$\mid [T] \quad \mid T \to T$

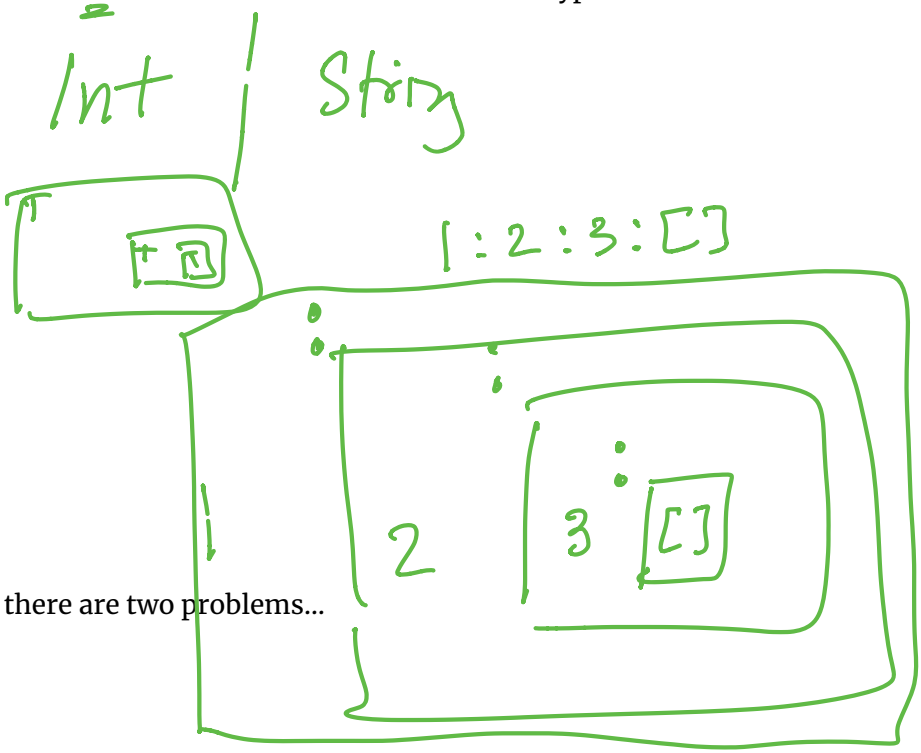
**Algebraic Data Types:** a single, powerful technique for building up types to represent complex data

- Lets you define *your own* data types
- Tuples and lists are *special* cases

# *Building data types*

Three key ways to build complex types/values:

1. **Product types (each-of)**: a value of `T` contains a value of `T1` *and* a value of `T2`
2. **Sum types (one-of)**: a value of `T` contains a value of `T1` *or* a value of `T2`
3. **Recursive types**: a value of `T` contains a *sub-value* of the same type `T`




# *Product types*

Tuples can do the job but there are two problems...

```
deadlineDate :: (Int, Int, Int)
deadlineDate = (2, 4, 2019)

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

-- | Deadline date extended by one day
extension :: (Int, Int, Int) -> (Int, Int, Int)
extension = ...
```

Can you spot them?

## *1. Verbose and unreadable*

A **type synonym** for `T` : a name that can be used interchangeably with `T`

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

deadlineDate :: Date
deadlineDate = (2, 4, 2019)

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

-- | Deadline date extended by one day
extension :: Date -> Date
extension = ...
```

## 2. Unsafe

We want this to fail at compile time!!!

```
extension deadlineTime
```

*Solution:* construct two different **datatypes**

```
data Date = Date Int Int Int
data Time = Time Int Int Int
-- constructor^   ^parameter types

deadlineDate :: Date
deadlineDate = Date 2 4 2019

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

# *Record syntax*

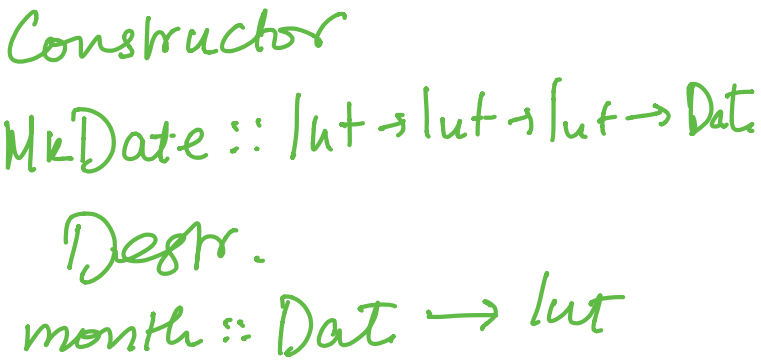
Haskell's **record syntax** allows you to *name* the constructor parameters:

- Instead of

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

- you can write:

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

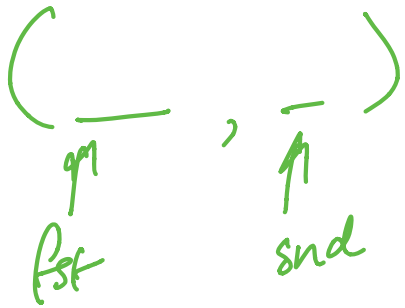



- then you can do:

```
deadlineDate = Date 2 4 2019

dealineMonth = month deadlineDate -- yikes, use field name as a function
```

# *Building data types*




Three key ways to build complex types/values:

1. **Product types** (**each-of**): a value of `T` contains a value of `T1` *and* a value of `T2` **[done]**
2. **Sum types** (**one-of**): a value of `T` contains a value of `T1` *or* a value of `T2`
3. **Recursive types**: a value of `T` contains a *sub-value* of the same type `T`

# *Example: NanoMarkdown*

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: ordered? and items ( `Bool` and `[String]` )

I want to store all paragraphs in a *list*

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

But this *does not type check*!!!

# *Sum Types*

Solution: construct a new type for paragraphs that is a *sum* (*one-of*) the three options!

Each paragraph is either:

- plain text ( `String` )
- heading: level and text ( `Int` and `String` )
- list: ordered? and items ( `Bool` and `[String]` )

```
data Paragraph             -- ^ 3 constructors, w/ different parameters
  = PText String           -- ^ text    : plain string
  | PHeading Int String    -- ^ heading: level and text (`Int` and `String`)
  | PList Bool [String]    -- ^ list    : ordered? and items (`Bool` and `[String]`)
```

# *QUIZ*

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

(PText "hey")

What is the type of `PText "Hey there!"`? i.e. How would GHCi reply to:

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

**A.** Syntax error

**B.** Type error

**C.** `PText`

**D.** `String` ✓

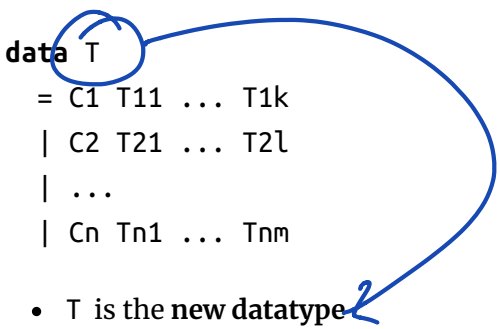
**E.** `Paragraph` ✓

PList True ["blah"]

PText "hello"

PHeading 2 "CSE130"

# *Constructing datatypes*



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

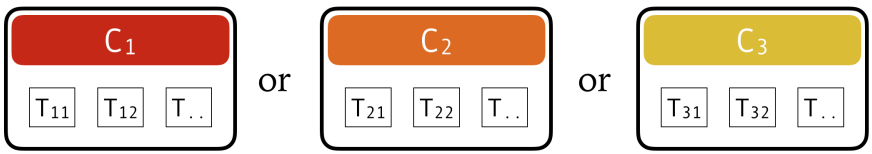
- `T` is the **new datatype**
- `C1 .. Cn` are the **constructors** of `T`

A **value** of type `T` is

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

You can think of a `T` value as a **box**:

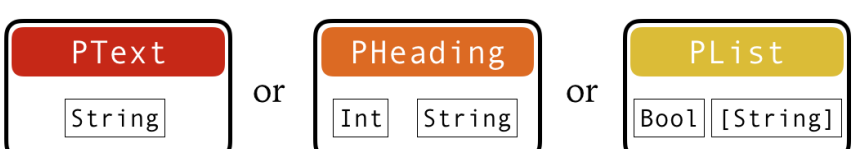
- *either* a box labeled `C1` with values of types `T11 .. T1k` inside
- *or* a box labeled `C2` with values of types `T21 .. T2l` inside
- *or* ...
- *or* a box labeled `Cn` with values of types `Tn1 .. Tnm` inside




One-of Types

Apply a constructor = pack some values into a box (and label it)

- `PText "Hey there!"`
  - put `"Hey there!"` in a box labeled `PText`
- `PHeading 1 "Introduction"`
  - put `1` and `"Introduction"` in a box labeled `PHeading`
- Boxes have different labels but same type ( `Paragraph` )



The `Paragraph` Type

with example values:



The `Paragraph` Type

# *QUIZ*

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

What would GHCi say to

```
>:t [PHeading 1 "Introduction", Text "Hey there!"]
```

**A.** Syntax error

**B.** Type error

**C.** `Paragraph`

**D.** `[Paragraph]`

**E.** `[String]`

# *Example: NanoMD*

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

Now I can create a document like so:

```
doc :: [Paragraph]
doc = [ PHeading 1 "Notes from 130"
      , PText "There are two types of languages:"
      , PList True ["those people complain about", "those no one uses"])
      ]
```

Now I want **convert documents in to HTML**.

I need to write a function:

```
html :: Paragraph -> String
html p = ??? -- depends on the kind of paragraph!
```

How to tell what's in the box?

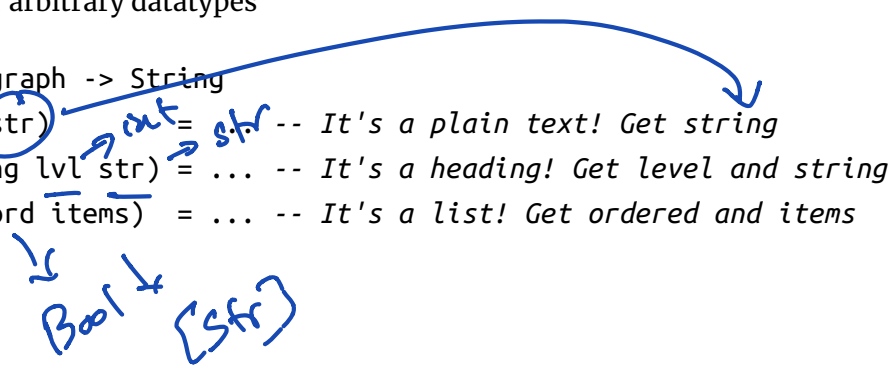
- Look at the label!

# *Pattern matching*

**Pattern matching** = looking at the label and extracting values from the box

- we've seen it before
- but now for arbitrary datatypes

```
html :: Paragraph -> String
html (PText str)          = ... -- It's a plain text! Get string
html (PHeading lvl str)   = ... -- It's a heading! Get level and string
html (PList ord items)    = ... -- It's a list! Get ordered and items
```



```haskell
html :: Paragraph -> String
html (PText str)            -- It's a plain text! Get string
  = unlines [open "p", str, close "p"]

html (PHeading lvl str)     -- It's a heading! Get level and string
  = let htag = "h" ++ show lvl
    in unwords [open htag, str, close htag]

html (PList ord items)      -- It's a list! Get ordered and items
  = let ltag   = if ord then "ol" else "ul"
        litems = [unwords [open "li", i, close "li"] | i <- items]
    in unlines ([open ltag] ++ litems ++ [close ltag])
```

# *Dangers of pattern matching (1)*

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

What would GHCi say to:

```
html (PHeading 1 "Introduction")
```

# *Dangers of pattern matching (2)*

```
html :: Paragraph -> String
html (PText str)        = unlines [open "p", str, close "p"]
html (PHeading lvl str) = ...
html (PHeading 0 str)   = html (PHeading 1 str)
html (PList ord items)  = ...
```

What would GHCi say to:

```
html (PHeading 0 "Introduction")
```

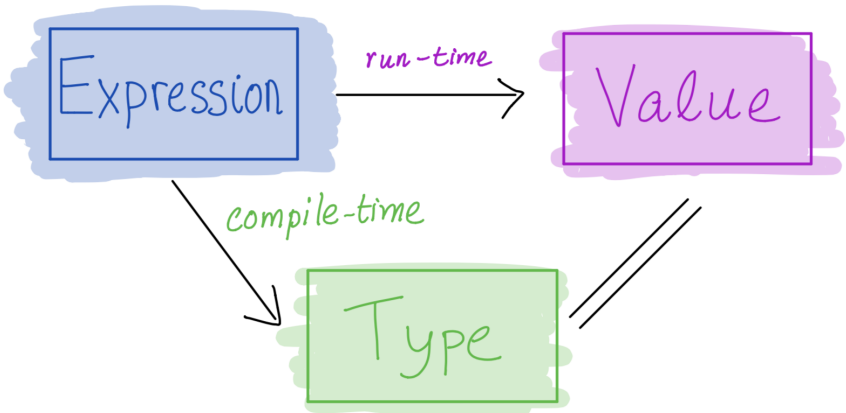
## *Dangers of pattern matching*

Beware of **missing** and **overlapped** patterns

- GHC warns you about *overlapped* patterns
- GHC warns you about *missing* patterns when called with `-W` (use `:set -W` in GHCi)

## *Pattern-Match Expression*

*Everything is an expression?*




We've seen: pattern matching in *equations*

Actually, pattern-match is *also an expression*

```
html :: Paragraph -> String
html p = case p of
           PText    str     -> unlines [open "p", str, close "p"]
           PHeading lvl str -> ...
           PList    ord items -> ...
```

The code we saw earlier was *syntactic sugar*

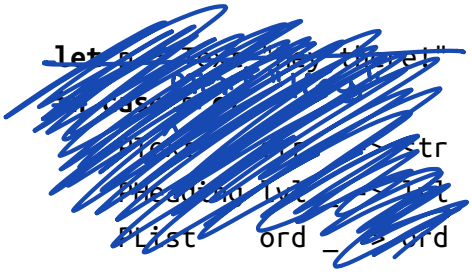
```
html (C1 x1 ...) = e1
html (C2 x2 ...) = e2
html (C3 x3 ...) = e3
```

is just for *humans*, internally represented as a **`case-of`** expression

```
html p = case p of
           (C1 x1 ...) -> e1
           (C2 x2 ...) -> e2
           (C3 x3 ...) -> e3
```

# *QUIZ*

What is the **type of**




case PText "Hey" of
PText str → str
PHead lvl _ → lvl
PList ord _ → ord

Q: What is the TYPE?

**A.** ~~Syntax error~~

**B.** Type error

**C.** `String` ✓

**D.** `Paragraph`

**E.** `Paragraph -> String` ✓

# *Pattern matching expression: typing*

The **`case`** expression

```
case e of
  pattern1 -> e1
  pattern2 -> e2
  ...
  patternN -> eN
```

has type `T` if

- each `e1` ... `eN` has type `T`
- `e` has some type `D`
- each `pattern1` ... `patternN` is a *valid pattern* for `D`
    - i.e. a variable or a constructor of `D` applied to other patterns

The expression `e` is called the **match scrutinee**

# *QUIZ*

What is the type of

```
let p = Text "Hey there!"
in case p of
    PText _      -> 1
    PHeading _ _ -> 2
    PList _ _    -> 3
```

**A.** Syntax error

**B.** Type error

**C.** `Paragraph`

**D.** `Int`

**E.** `Paragraph -> Int`

*Building data types*