CompSci 220

Programming Methodology

16: Understanding FP Error Handling Part 2

Objectives

- Option Composition and Lifting
- For-Comprehension Expansion
- The Either Data Type
- Option and Either in the Standard Library

Examples

- Let us first take a look at some more examples of using Option.

Option Composition and Lifting

- How does Option affect existing code?
- You may have been concerned that **Option** will "infect" an entire code base – one can imagine how any callers of methods that take or return **Option** will need to be modified to handle either **Some** or **None**.
- Fortunately, this does not happen.
- We can *lift* ordinary functions to become functions that operate on **Option**.

Problem Example

- Consider the math.abs function on Double:

  math.abs(v: Double): Double

- What if we wanted to pass an Option to math.abs or pass the result of the result of math.abs to a function to another function that expects an Option[Double]?

- We "could" pattern match on Option and extract the value before calling math.abs. Or, we could wrap the result in Some.
  But, there is a better way…

Lifting to Option

Here is an example of what we want to do:

```
val absO: Option[Double] => Option[Double] = lift(math.abs)
```

Here is the type signature of *lift*:

```
def lift[A,B](f: A => B): Option[A] => Option[B] = ???
```

Take out a piece of paper and write out the implementation of *lift*.

You should use a functional method from the Option type to do this.

Lifting to Option

Here is an example of what we want to do:

```
val absO: Option[Double] => Option[Double] = lift(math.abs)
```

Here is the type signature of *lift*:

```
def lift[A,B](f: A => B): Option[A] => Option[B] = _ map f
```

See how this works?

Lifting to Option

Here is an example of what we want to do:

```
val absO: Option[Double] => Option[Double] = lift(math.abs)
```

Here is the type signature of *lift*:

```
def lift[A,B](f: A => B): Option[A] => Option[B] = _ map f
```

See how this works?

We return a new function by using the _ anonymous function syntax

Lifting to Option

Here is an example of what we want to do:

```
val absO: Option[Double] => Option[Double] = lift(math.abs)
```

Here is the type signature of *lift*:

```
def lift[A,B](f: A => B): Option[A] => Option[B] = _ map f
```

See how this works?

This maps the first option using function f that is provided to lift.

Lift Design Pattern?

What design pattern does
the *lift* function implement?

Lift Design Pattern?

What design pattern does
the *lift* function implement?

Options and Exceptions

- Imagine you are implementing the logic for a car insurance company's website, which contains a page where users can submit a form to request an instance online quote.

Options and Exceptions

- Imagine you are implementing the logic for a car insurance company's website, which contains a page where users can submit a form to request an instance online quote.

- We would like to parse the information from this form and ultimately call our rate function.

Rate Function


```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

- We want to be able to call this function, but if the user is submitting their age and number of speeding tickets in a web form, these fields will arrive as simple strings that we have to (try to) parse into integers.

Rate Function

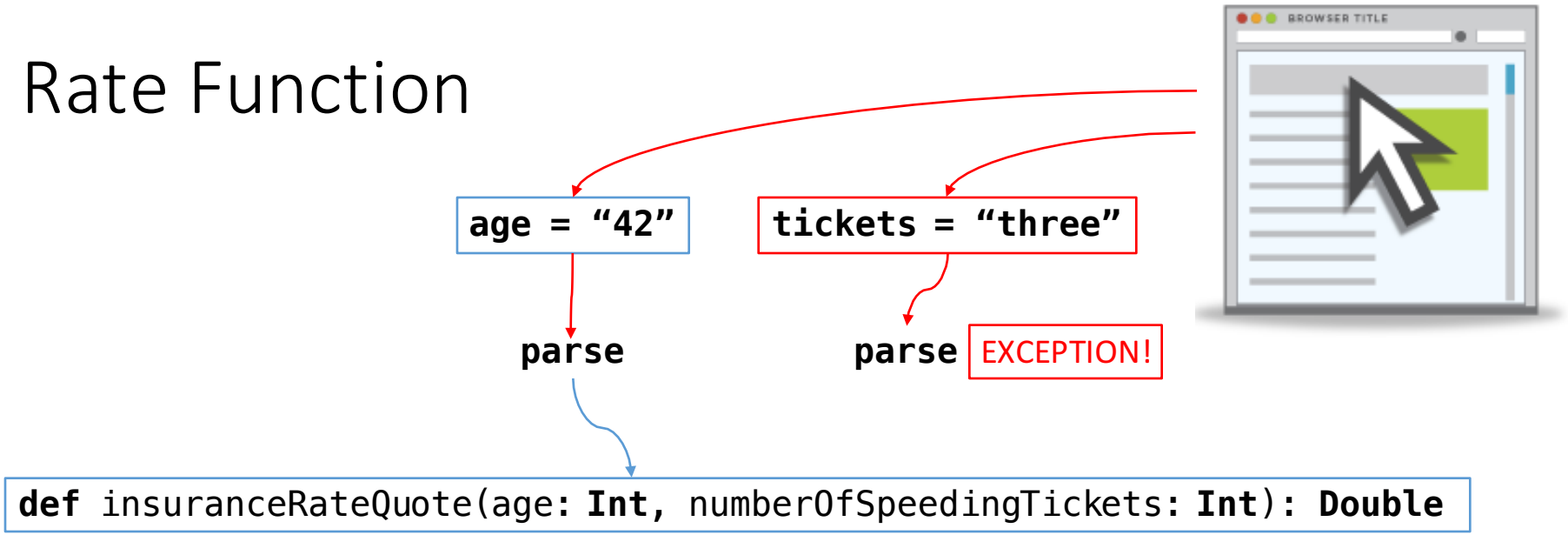
`age = "42"`

`parse`

```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

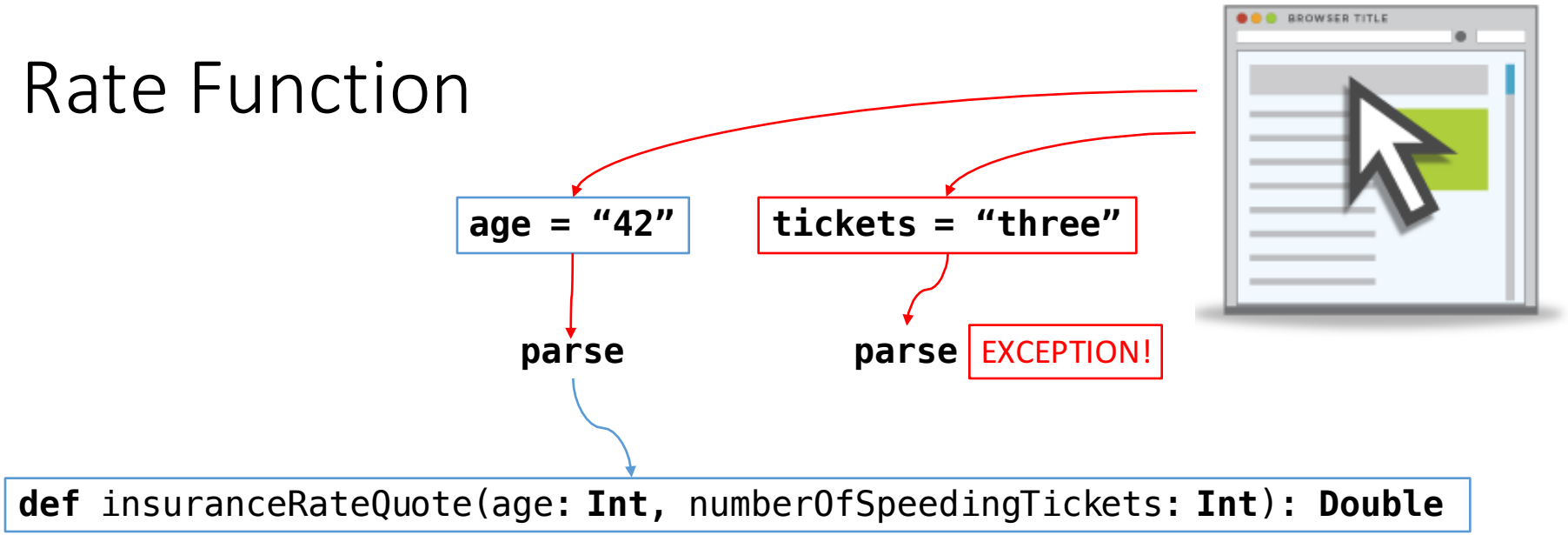
- We want to be able to call this function, but if the user is submitting their age and number of speeding tickets in a web form, these fields will arrive as simple strings that we have to (try to) parse into integers.

Rate Function



- We want to be able to call this function, but if the user is submitting their age and number of speeding tickets in a web form, these fields will arrive as simple strings that we have to (try to) parse into integers.

Rate Function



- So, what if **parse** throws an exception?
- “42”.toInt is ok, but “three”.toInt throws a NumberFormatException!
- This does not fit well into our *functional* model.
- We need a way to convert an Exception into an Option.

Exceptions to Options

- Ideally, we want to be able to do something like this:

```
Try {
  “42”.toInt
}
// Try { … } should evaluate to Option
```

- Have we seen a way to do this before?
- What language technique can we use to implement this?

Implementing Try

- Here is the signature:

```
def Try[A](block: => A): Option[A]
```

- Spend a few minutes to see if you can implement this.
- The given *block* could possibly throw an exception.

Implementing Try

- Here is the signature:

```
def Try[A](block: => A): Option[A] =
  try Some(a)
  catch { case e: Exception => None }
```

- Spend a few minutes to see if you can implement this.
- The given *block* could possibly throw an exception.

Implement parseInsuranceRateQuote

- Ok, so now we can convert exceptions into Options.
- Let us implement a function to parse strings into ints from our insurance website and call insuranceRateQuote. Try it!

```
def parseInsuranceRateQuote(age: String,
                            numberOfSpeedingTickets: String): Option[Double] = ???
```

```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

Implement parseInsuranceRateQuote

- That wasn't so hard.
- But, we have a problem.

```
def parseInsuranceRateQuote(age: String,
                            numberOfSpeedingTickets: String): Option[Double] = ???
  val optAge = Try { age.toInt }
  val optTickets = Try { numberOfSpeedingTickets.toInt }
  insuranceRateQuote(optAge, optTickets)
}
```

```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

Implement parseInsuranceRateQuote

- That wasn't so hard.
- But, we have a problem.
- It doesn't compile!

```
def parseInsuranceRateQuote(age: String,
                            numberOfSpeedingTickets: String): Option[Double] = ???
  val optAge = Try { age.toInt }
  val optTickets = Try { numberOfSpeedingTickets.toInt }
  insuranceRateQuote(optAge, optTickets)
}
```

```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

Implement parseInsuranceRateQuote

- It doesn't compile because there is a type error.
- insuranceRateQuote expects Ints, but we are giving it Options.
- How do we solve this one?

```
def parseInsuranceRateQuote(age: String,
                            numberOfSpeedingTickets: String): Option[Double] = ???
  val optAge = Try { age.toInt }
  val optTickets = Try { numberOfSpeedingTickets.toInt }
  insuranceRateQuote(optAge, optTickets)
}
```

```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

Implement parseInsuranceRateQuote

- What we need is a function extracts the two Ints (age & tickets) from the Option values, plugs them in to the insuranceRateQuote function, and then returns an Option

```
def parseInsuranceRateQuote(age: String,
                            numberOfSpeedingTickets: String): Option[Double] = ???
  val optAge = Try { age.toInt }
  val optTickets = Try { numberOfSpeedingTickets.toInt }
  insuranceRateQuote(optAge, optTickets)
}
```

```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

Implement map2

- Write a generic function map2 that combines two Option values using a binary function. If either Option value is None, then the return value is too. Here is its signature:

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A,B) => C): Option[C] = ???
```

Implement map2

- Write a generic function map2 that combines two Option values using a binary function. If either Option value is None, then the return value is too. Here is its signature:

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A,B) => C): Option[C] =
  (a,b) match {
    case (Some(x), Some(y)) => Some(f(x,y))
    case _ => None
  }
```

Implement map2

- Write a generic function map2 that combines two Option values using a binary function. If either Option value is None, then the return value is too. Here is its signature:

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A,B) => C): Option[C] =
  for {
    x <- a
    y <- b
  } yield f(x,y)
```

Fixing The Problem: map2

- What we need is a function extracts the two Ints (age & tickets) from the Option values, plugs them in to the insuranceRateQuote function, and then returns an Option

```
def parseInsuranceRateQuote(age: String,
                            numberOfSpeedingTickets: String): Option[Double] = ???
  val optAge = Try { age.toInt }
  val optTickets = Try { numberOfSpeedingTickets.toInt }
  insuranceRateQuote(optAge, optTickets)
}
```

```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

Fixing The Problem: map2

- What we need is a function extracts the two Ints (age & tickets) from the Option values, plugs them in to the insuranceRateQuote function, and then returns an Option

```
def parseInsuranceRateQuote(age: String,
                            numberOfSpeedingTickets: String): Option[Double] = ???
  val optAge = Try { age.toInt }
  val optTickets = Try { numberOfSpeedingTickets.toInt }
  map2(optAge, optTickets)(insuranceRateQuote)
}
```

```
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

Either Data Type

- Option is great for many cases.
- However, what if you need to indicate the error that occurred?
  - Option, just gives us None
- For that, we have the Either data type:

```
sealed trait Either[+E, +A]
case class Left[+E](value: E) extends Either[E,Nothing]
case class Right[+A](value: A) extends Either[Nothing,A]
```

Revisiting the Mean Function with Either

- We had this definition for mean using Option:

```
def mean(xs: Seq[Double]): Option[Double] =
  if (xs.isEmpty) None
  else Some(xs.sum / xs.length)
```

We can easily convert this to use the Either type.

Revisiting the Mean Function with Either

- We had this definition for mean using Option:

```
def mean(xs: Seq[Double]): Either[String,Double] =
  if (xs.isEmpty) Left("mean of empty list!")
  else Right(xs.sum / xs.length)
```

Exceptions to Eithers

- Sometimes, we want to include more information about the error:

```
def safeDiv(x: Int, y: Int): Either[Exception,Int] =
  try Right(x / y)
  catch { case e: Exception => Left(e) }
```

Exceptions to Eithers – With Try

- Now, we can generalize this:

```
def Try[A](a: => A): Either[Exception,A] =
  try Right(a)
  catch { case e: Exception => Left(e) }
```