THE LIFT FRAMEWORK AND HOW IT RELATES TO 3155

Preston Dotsey and Pankaj Behera

AGENDA

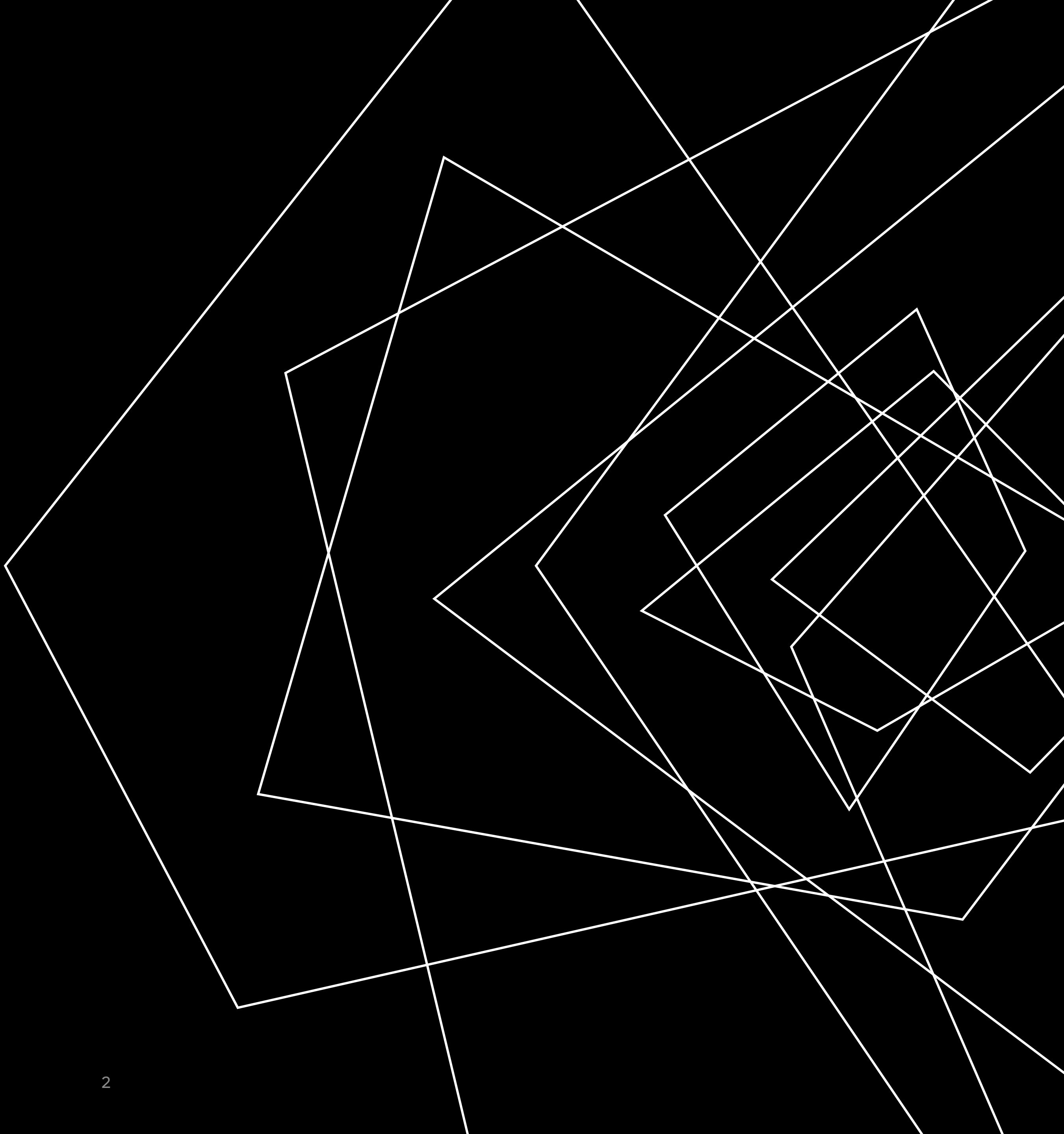
Background of Lift

Abstract Data Types

Higher-order Functions

Functions as Values

Summary



WHAT IS LIFT?

- Lift is a free and open-source web framework that is designed for the Scala programming language.
- It was originally created by David Pollak who was dissatisfied with certain aspects of the Ruby on Rails framework.
- Lift was launched as an open-source project on 26 February 2007 under the Apache License 2.0
- An example of the framework in commercial use is Foursquare, a cloud-based location technology which allows customers and business to connect with each other with the aid of geo data.

WHAT IS LIFT?

- Lift as a framework was designed to specifically write web applications.
- It is like Ruby On Rails in that it favors convention over configuration but differs in that is designed with the “View First” architectural rather than the “model-view-controller” pattern used by Ruby.
- The use of “View First” is intended to make the development more designer friendly, which was inspired by the Wicket framework.

WHAT IS LIFT DESIGNED TO DO?

- Lift is designed to be:
  - Designer friendly
  - Highly speed efficient
  - Secure against online attacks
  - Easily scalable
  - Easily abstractable

SO HOW DOES LIFT ACCOMPLISH THESE GOALS, AND HOW DO THEY RELATE TO CONCEPTS FROM CSCI 3155?

SO HOW DOES LIFT ACCOMPLISH THESE GOALS, AND HOW DO THEY RELATE TO CONCEPTS FROM CSCI 3155?

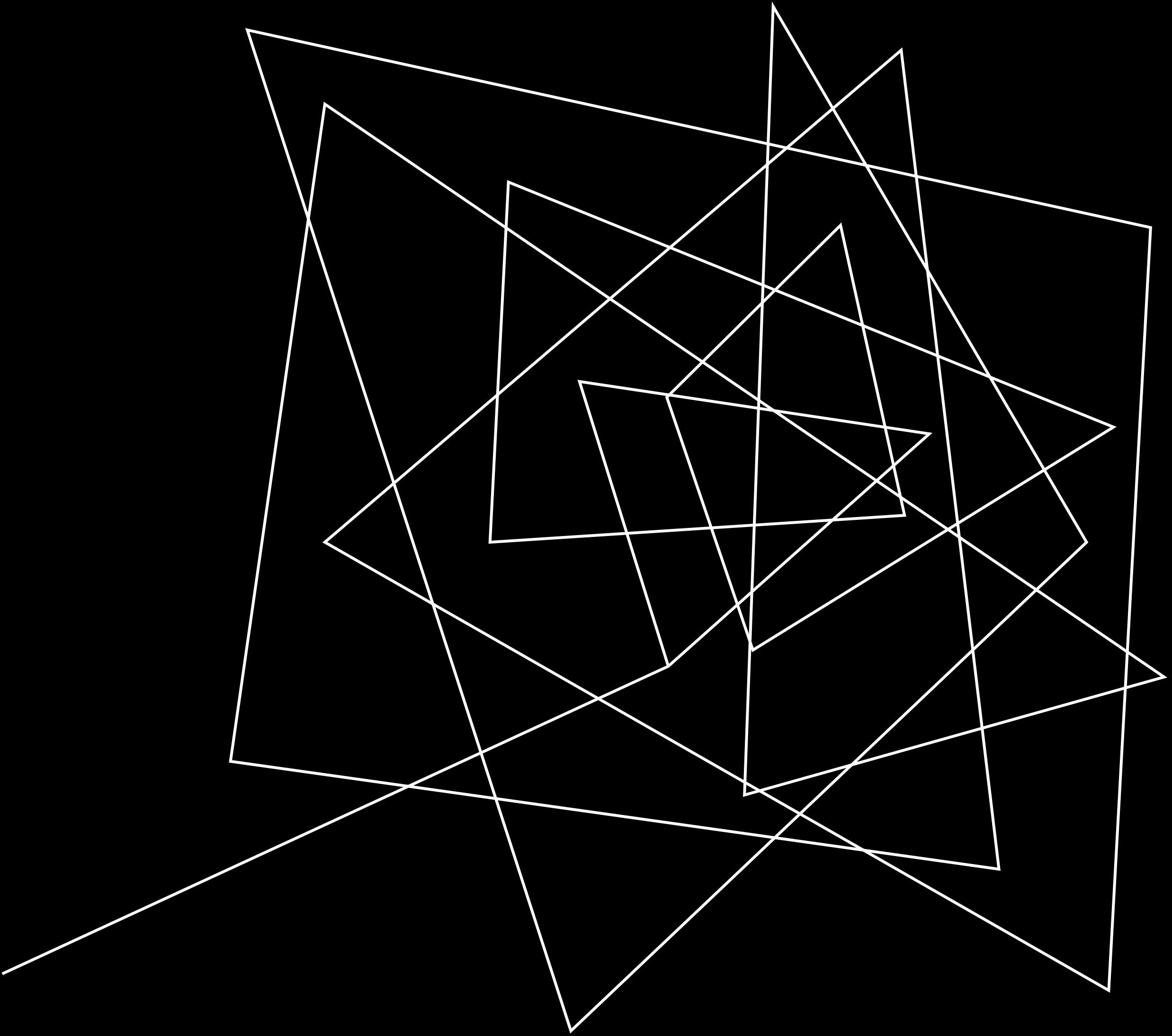
- The Lift framework owes its ability to accomplish its goals through its incorporation of several key concepts which we have learned about over the course of this semester in 3155. Namely:
  - Abstraction of Data Types

SO HOW DOES LIFT ACCOMPLISH THESE GOALS, AND HOW DO THEY RELATE TO CONCEPTS FROM CSCI 3155?

- The Lift framework owes it ability to accomplish its goals through its incorporation of several key concepts which we have learned about over the course of this semester in 3155. Namely:
  - Abstraction of Data Types
  - Use of Higher Order Functions

SO HOW DOES LIFT ACCOMPLISH THESE GOALS, AND HOW DO THEY RELATE TO CONCEPTS FROM CSCI 3155?

- The Lift framework owes it ability to accomplish its goals through its incorporation of several key concepts which we have learned about over the course of this semester in 3155. Namely:
  - Abstraction of Data Types
  - Use of Higher Order Functions
  - Treating Functions as Values



ABSTRACT DATA TYPES

WHAT IS AN ABSTRACT DATA TYPE?

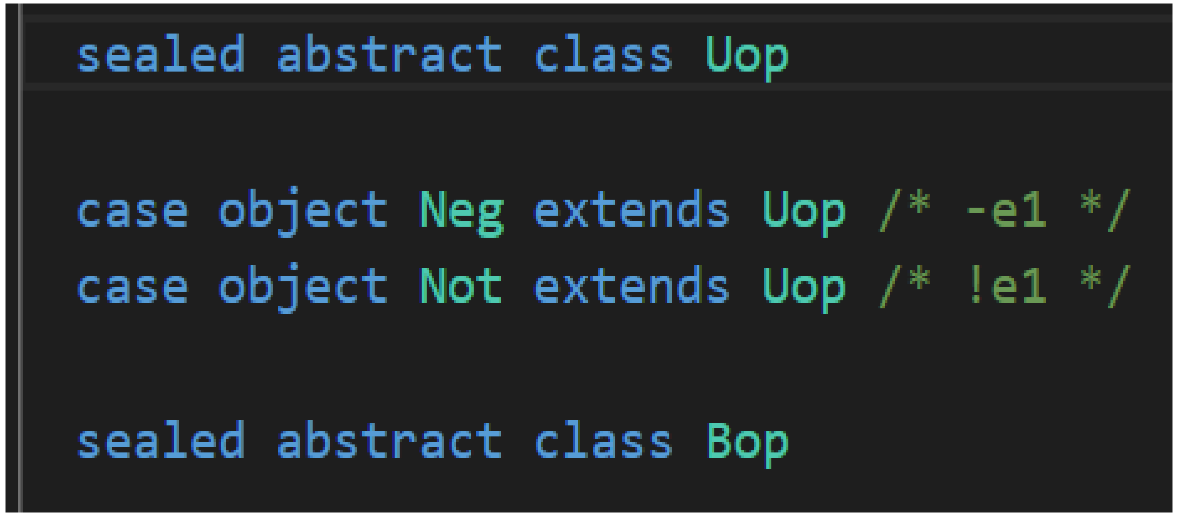
- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called “abstract” because it gives an implementation-independent view.

HOW DID WE USE ABSTRACT DATA TYPES IN CLASS?

- Abstract Data Types are used heavily throughout the labs we have completed in this class.
- Abstract Data Types were especially prevalent in the ast.scala, where abstract classes are declared within the ast object, but the method implementation is not disclosed.
- Instead of implementing the methods of the abstract classes in the classes themselves, we instead implemented the class methods within the lab scala files (e.g., lab4.scala) and the objects then communicate with each other to carry out their functions.
- By incorporating this abstraction into our lab files, we are able to increase the ease at which we make changes to methods, hide private data, increase clarity within the code and make our code more secure by hiding key details.

EXAMPLE

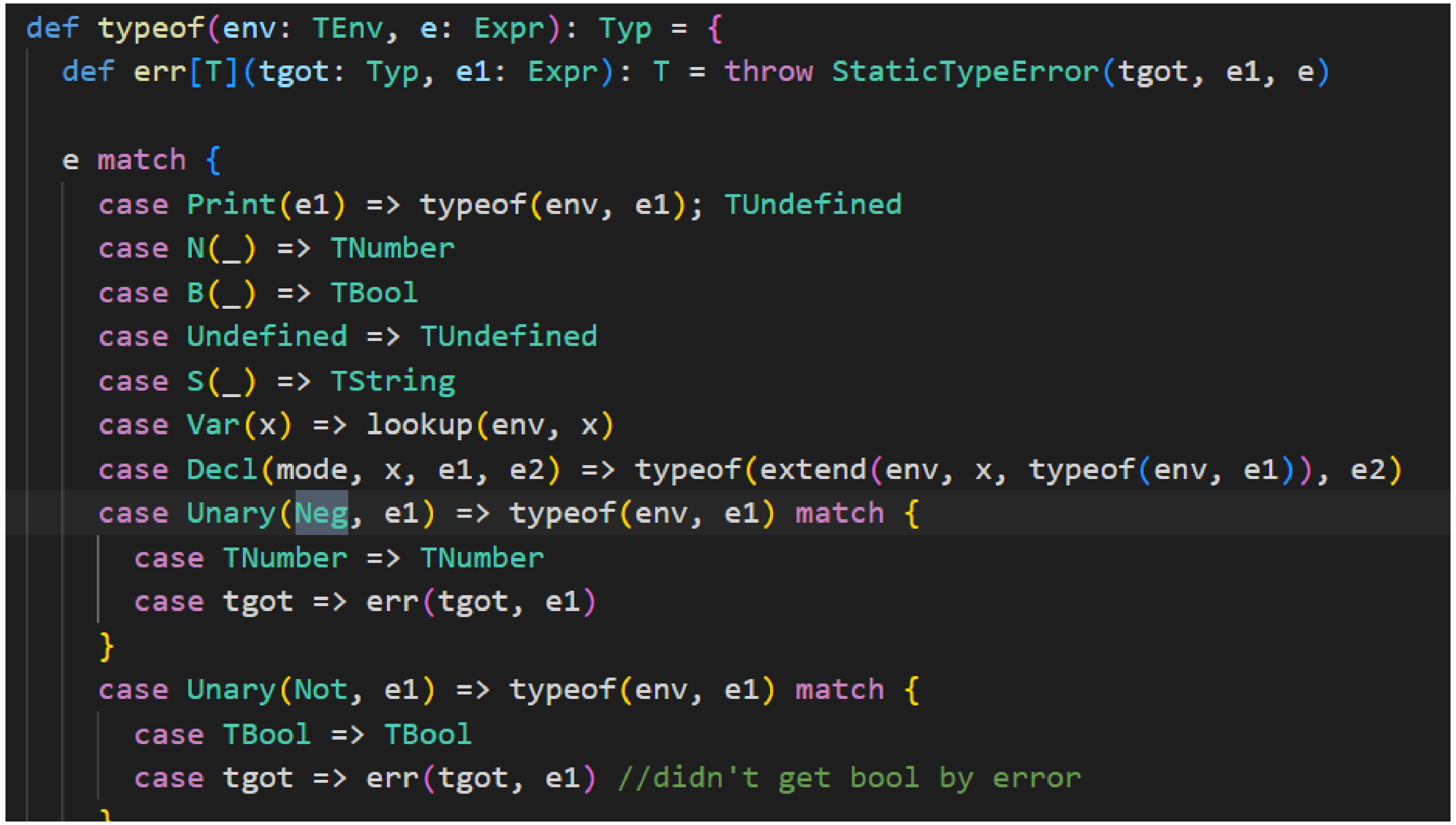
- As an example of an ADT used in our course, below is an excerpt of an abstract data type used in Lab 4, found in the file ast.scala:



```scala
sealed abstract class Uop

case object Neg extends Uop /* -e1 */
case object Not extends Uop /* !e1 */

sealed abstract class Bop
```

EXAMPLE

- Below is the method and class implementation found in lab4.scala:



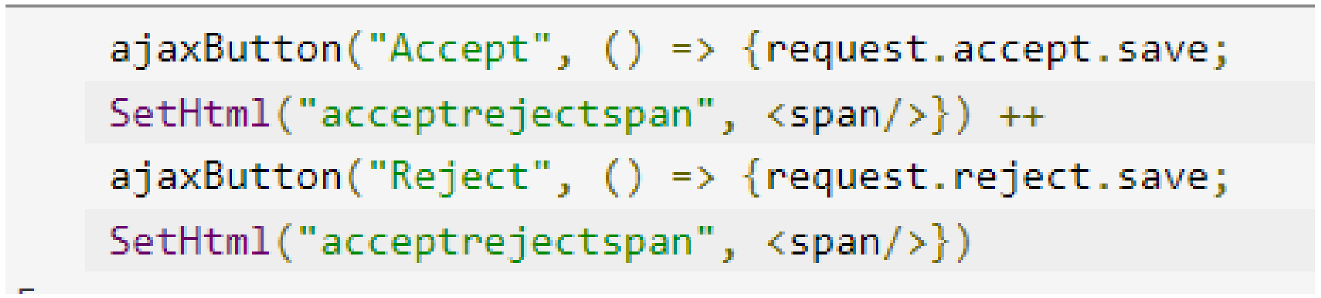
```scala
def typeof(env: TEnv, e: Expr): Typ = {
  def err[T](tgot: Typ, e1: Expr): T = throw StaticTypeError(tgot, e1, e)

  e match {
    case Print(e1) => typeof(env, e1); TUndefined
    case N(_) => TNumber
    case B(_) => TBool
    case Undefined => TUndefined
    case S(_) => TString
    case Var(x) => lookup(env, x)
    case Decl(mode, x, e1, e2) => typeof(extend(env, x, typeof(env, e1)), e2)
    case Unary(Neg, e1) => typeof(env, e1) match {
      case TNumber => TNumber
      case tgot => err(tgot, e1)
    }
    case Unary(Not, e1) => typeof(env, e1) match {
      case TBool => TBool
      case tgot => err(tgot, e1) //didn't get bool by error
```

HOW DOES LIFT USE ADTS?

- Lift also heavily relies on Abstract Data Types.
- The most obvious use of ADTs within the Lift framework is in how Lift abstracts away their http requests.
- Lift at its core seeks to abstract away the HTTP request/response cycle rather than placing object wrappers around the HTTP Request. By incorporating this abstraction into their lab files, they can increase the ease at which they make changes to methods, hide private data, increase clarity within the code and make their code more secure by hiding key details.

EXAMPLE

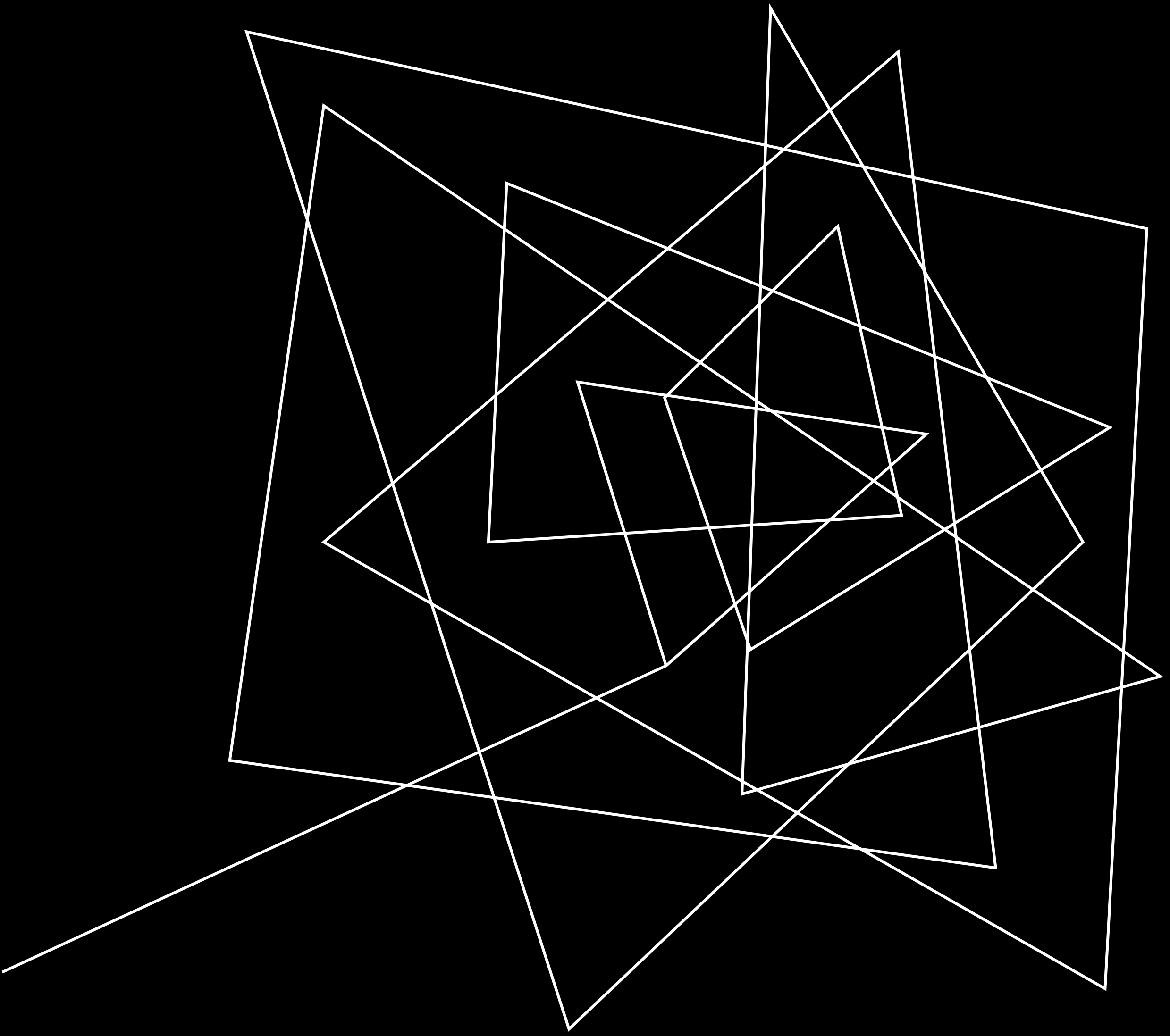
- Below is an example of how they use ADT’s to hide their implementation details in their processing of HTTP requests. Specifically, this is an example of how they implement the accepting of a friend request in Foursquare:



```
ajaxButton("Accept", () => {request.accept.save;
SetHtml("acceptrejectspan", <span/>}) ++
ajaxButton("Reject", () => {request.reject.save;
SetHtml("acceptrejectspan", <span/>})
```

HOW DOES LIFT USE ABSTRACT DATA TYPES?

- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called “abstract” because it gives an implementation-independent view.



HIGHER ORDER FUNCTIONS

WHAT IS A HIGHER ORDER FUNCTION?

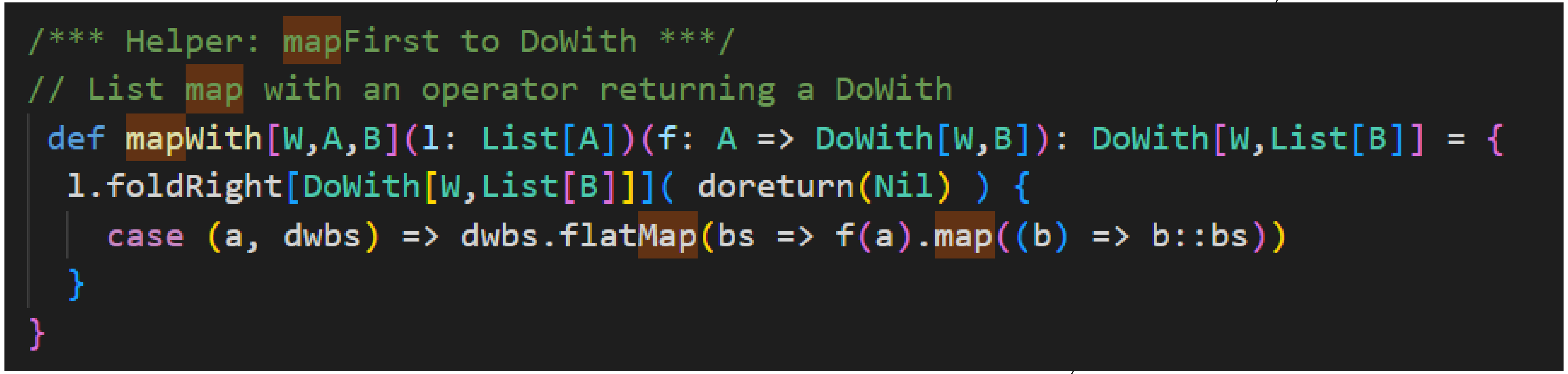
- In computer science, a higher-order function (HOF) is a function that does at least one of the following:
  - takes one or more functions as arguments (i.e., a procedural parameter, which is a parameter of a procedure that is itself a procedure),
  - returns a function as its result.

HOW DID WE USE HIGHER ORDER FUNCTIONS IN 3155?

- Higher order functions are one of the most important concepts we covered during this semester.
- With the use of higher order functions, we are able incorporate another layer of abstraction into our code and introduce a layer of separation between our classes.
- Higher order functions also allows us to use the return values from the functions called within the HOFs parameters in a streamlined and easy to parse manner.
- By incorporating HOFs, we are also able to facilitate function composition.
- This is possible due to the fact that Scala treats functions as first-class values.

EXAMPLE

- As an example of an HOF used in our class is the use of the map functions we used in multiple labs. This example is pulled from the lab5.scala file:



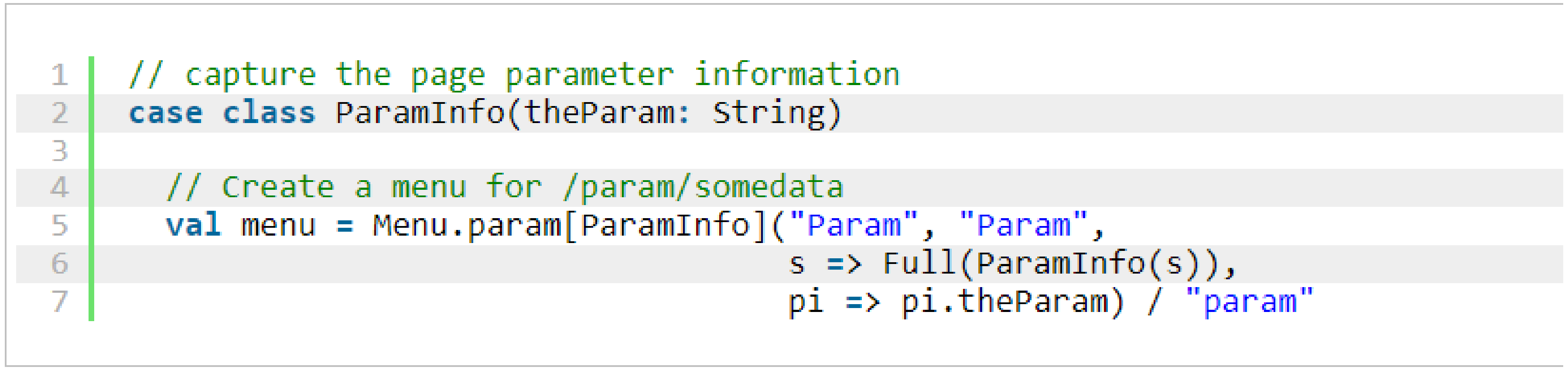
```scala
/*** Helper: mapFirst to DoWith ***/
// List map with an operator returning a DoWith
 def mapWith[W,A,B](l: List[A])(f: A => DoWith[W,B]): DoWith[W,List[B]] = {
  l.foldRight[DoWith[W,List[B]]]( doreturn(Nil) ) {
    case (a, dwbs) => dwbs.flatMap(bs => f(a).map((b) => b::bs))
  }
}
```

HOW DOES LIFT USE HIGHER ORDER FUNCTIONS?

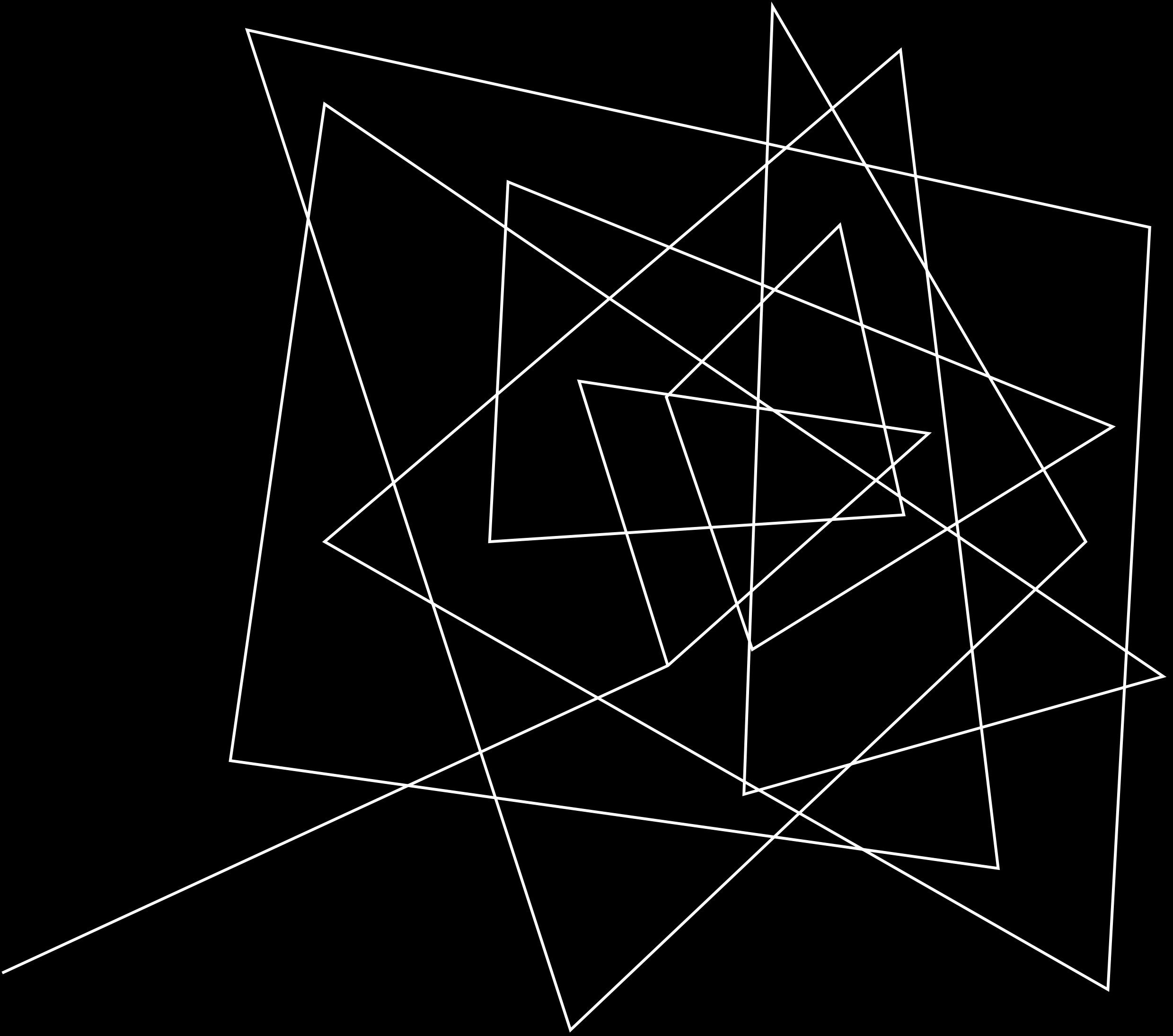
- Lift heavily relies on HOFs as well, as the entire framework is written in Scala, and as such uses the same libraries that we have used in 3155.
- In the building of web applications with Lift, many HOFs are used to facilitate the functionality of these same applications .
- Common HOFs used in the Lift framework include map, flatmap filter.
- Common uses of these HOFs are in the organizing and listing of user data and retrieving internal data from the applications backend.

EXAMPLE

- As an example of an HOF used in the Lift framework. The below example is the code which is implemented to pull parameters from an incoming URL and parse it into type-safe variables. As seen below, the use of HOFs is demonstrated in the use of the function Full() with is taking the function ParamInfo(s) as a parameter inside the function (reference in Lift handbook: https://simply.liftweb.net/index-3.2.html#toc-Subsection-3.2.7) :



```scala
// capture the page parameter information
case class ParamInfo(theParam: String)

  // Create a menu for /param/somedata
  val menu = Menu.param[ParamInfo]("Param", "Param",
                                   s => Full(ParamInfo(s)),
                                   pi => pi.theParam) / "param"
```



FUNCTIONS AS VALUES

WHAT DO WE MEAN FUNCTIONS ARE VALUES?

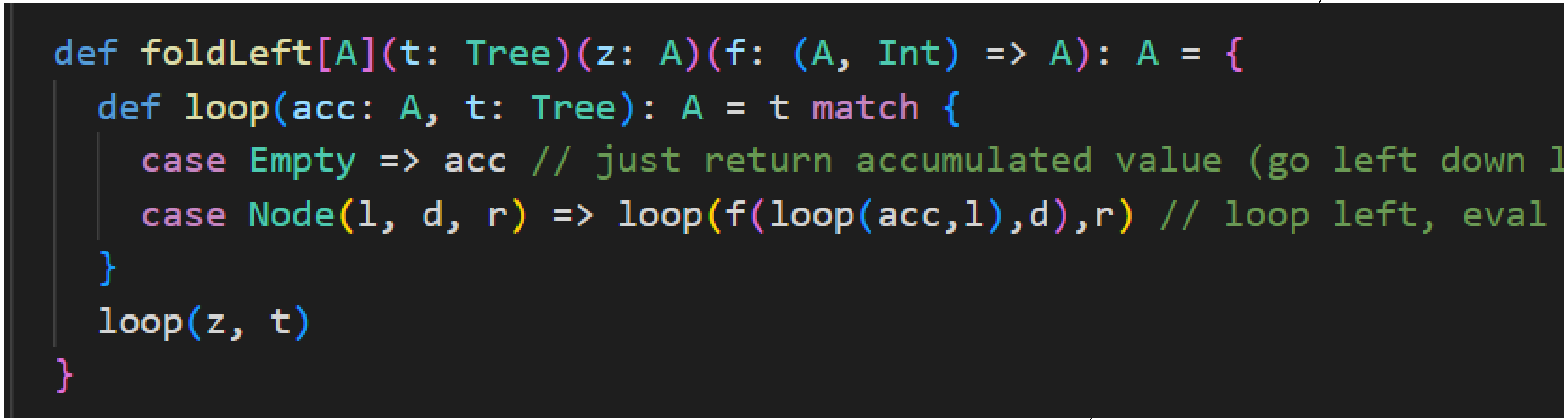
- When we say functions are values, this concept is one of the defining features of a functional programming language. All functional programming language has this concept as a core tenant of how these languages execute.
- The idea behind functions are values is that functions are not separate from the values that they return, but rather are the values themselves.
- By treating the functions, themselves as values, we can now use functions as parameters within higher order functions.
- This allows us to create cleaner and more efficient code, as well as increase abstraction and rely more heavily on the communication between objects instead of making monolithic classes which are much harder to debug than a collection of smaller objects that cooperate.

HOW DOES USING FUNCTIONS AS VALUES FIT INTO 3155?

- As Prof. Chang has said time and time again, functions are values!
- Incorporating this concept into how we think about functional programming has been vital to creating solutions to our assignments this semester, especially when writing code for the lab assignments.
- In our labs, we have been using many, many higher order functions. The reason why we can use the parameter functions within higher order functions is that in Scala, functions are treated themselves as first-class values. This means that when we pass a function to another function, we are not actually passing the functions details to the HOF, but rather passing the *value* that the function itself represents. Because why? BECAUSE FUNCTIONS ARE VALUES!

EXAMPLE

- A perfect example of how functions are values is in how we have treated them as parameters in HOFs in our lab assignments. In the code snippet below, I have pulled this code from lab4.scala. The below function foldLeft incorporates another function loop(). In the case Node(l,d,r), it calls loop() which itself takes loop() as a parameter to recursively loop left and right through the supplied list while evaluating the values. This is possible due to the fact that loop() is treated as a value :



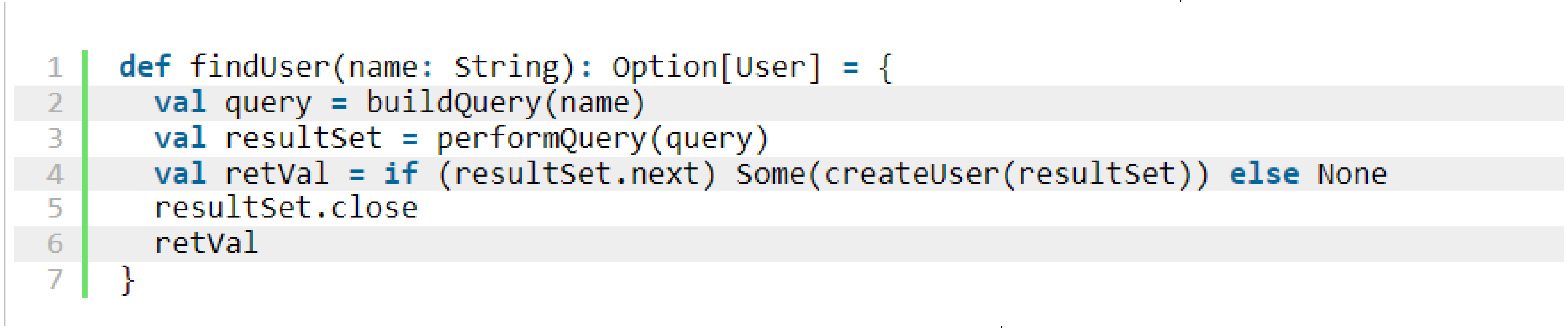
```
def foldLeft[A](t: Tree)(z: A)(f: (A, Int) => A): A = {
  def loop(acc: A, t: Tree): A = t match {
    case Empty => acc // just return accumulated value (go left down l
    case Node(l, d, r) => loop(f(loop(acc,l),d),r) // loop left, eval
  }
  loop(z, t)
}
```

HOW DOES LIFT USE FUNCTIONS AS VALUES?

- Just as we have used functions as values in our lab assignments in 3155, Lift also follows this concept in its framework through it's use of Scala higher order functions.
- Practically all web application frameworks make heavy use of HOFs, which can only execute if the functions in the HOF parameters are treated as values.
- Lift takes advantage of this concept to use functions as values within HOF parameters to facilitate simpler code implementation, ensure code security when deployed to clients and allow developers to use the Scala libraries which make their framework run smoothly.

EXAMPLE

- A perfect example of how functions are treated as values in the Lift Framework is in the use of HOFs in their code, which relies on this concept. The example below is from a function called findUser, which is a Lift method to query their server database to find user ids. In this function, there is a chain of HOFs which use the previous function in their parameters to find their intended data result. If functions were not values in the Scala environment, this would not be possible :



```scala
def findUser(name: String): Option[User] = {
  val query = buildQuery(name)
  val resultSet = performQuery(query)
  val retVal = if (resultSet.next) Some(createUser(resultSet)) else None
  resultSet.close
  retVal
}
```

SUMMARY

What did we learn?

- The use of Abstract Data Types are not just a thought exercise that we used in class, but rather an integral part of building web applications that are secure and easy to understand and develop. The lift framework is no exception and uses ADT’s to their benefit.
- Higher order functions are ubiquitous when working in functional programming languages. While we have worked with HOFs to a large degree in this class, we will work with them to an even greater degree when going out into industry. When looking at Lifts code implementations, there were HOFs in nearly every class, making them a core component of the functionality of the framework. Through the exploration of how HOFs are used within the Lift framework, it has reinforced just how useful and vital are HOFs are to the design and creation of efficient and maintainable code.
- FUNCTIONS ARE VALUES! The reason why Professor Chang has repeated it so many times is that this idea is at the very core of how functional programming languages are able to work. If functions were not values, HOFs would not even exists and all the concepts which lean on HOFs to operate would become severely crippled. Functions are values, and with this concept in mind, we as engineers will be able to take all of these related concepts that build off of the core idea of functions being values into our careers and be able to become better engineers.

THANK YOU FOR WATCHING!

Preston Dotsey

prdo5576@colorado.edu

Pankaj Behera

pabe3647@colorado.edu