Lab 4: Higher ordered functions and interpreting types

L4: Preface

L4D1

Lab 4 marks the beginning of a shift in difficulty of the course material. Labs 4 and 5 cover topics that students often find particularly challenging. Please use this weekend wisely to start in on the lab 4 material.

- Review CH3 of csci3155-notes
- Skim CH4 of csci3155-notes, then read them.
- Print out, read and annotate the lab 4 handout
- Skim MulList.scala (provided on Moodle in our readings and our code folder as a zip file)
- Read below talk on Linked Lists

L4D1T1: Linked Lists

So as mentioned earlier higher order functions (HOFs) can be super useful to developers. Scala has a few of these built into the library of List[A] to operate on List over any type A.

But what is the List type in Scala? It is essentially a singly linked list. I can describe List[A] were A is an Int quite easily with a grammar as follows:

List[Int] ::= Nil | Int :: List[Int]

It's base case is Nil

It uses '::' (the CONS) operator to concatenate an element of the list with a list of that sort of element.

Historic Fact : The Cons operator is a concept from the LISP programming language, it's close friends with the CLDR operator

In Scala, to construct the list of 1,2,3 and store that to a variable myList I might say:

```
val myList:List[Int] = 1 :: 2 :: 3 :: Nil
```

In Scala, println works on Lists:

```
println(myList) // will print `List(1, 2, 3)`
```

This cons operator can also be used during pattern matching. It takes the head element and stores that to the left of the cons operator and it takes the rest of the list, a.k.a. the tail, and stores that to the right of the operator

```
/*
        This will print:
                1
                List(2,3)
 */
val myList:List[Int] = 1 :: 2 :: 3 :: Nil
myList match {
        case Nil => ??? // doesn't really matter since my list isn't empty
        case h :: t => println(h); println(t)
        case _ => ??? // unreachable code for this example
}
```

So, suppose I want to summate over the list there are many ways that I can do that. Let's start by writing a function to do this for us

```
def sum(l:List[Int]):Int = l match {
        case Nil => 0
        case h :: t => h + sum(t)
}

val myList:List[Int] = 1 :: 2 :: 3 :: Nil
println(sum(myList))  // prints 6
```

Since this is a native collection to Scala I can also use Scala's native fold method for Lists which will scan over my list from left to right and accumulate a value. Here is how I will call this fold method on my list. Note that this method is a HOF in that it takes a function as input.

Here is the type definition of the fold method for lists in Scala (not very pretty, I know)

```
_:List[A].fold(_:B)(_:(B, A) => B): B
```

Perhaps the friendliest way to write a script that summates all values of `myLists` if you are not yet comfortable with the concept of passing functions as arguments at a call sight directly is as follows:

```
def tmp ( acc , h ) = h + acc
myList.fold(0)(tmp)
```

Another friendly way is:

```
myList.fold(0)({
        def ( acc , h ) = h + acc
})
```

The way you should be comfortable writing this by the time that you complete lab 4 is:

```
myList.fold(0)({
        (acc,h) => {
                h + acc
        }
})
```

The way that you should write a simple use of HOF by the time you complete the course is:

```
myList.fold(0){ (acc,h) => h + acc }
```

fold takes 3 inputs in the following form by types:

- L:List[A].fold(z:B)(cb:(B,A) => B)
  - L - the list to fold on
  - z – the base case for the folding operation
  - cb – the callback to be applied to each element of l to accumulate an output
- in our example
  - L – the variable myList
  - z – the value 0
  - cb - the functional value (acc,h) => h + acc
- for those of you that prefer the concrete syntax, the fold method of lists looks a lot like this:
  - def fold[A,B](l:List[A])(z:B)(f:(B,A) => B):B = l match {

```
        case Nil => z
        case h::t => fold(t)(f(z,h))(f)
    }
```

  - Note: this is not the concrete syntax of the fold method for List[A]. This is a home brewed fold function for List[A] that has a similar feel to the fold method over lists.

So that you are aware… there is one other syntax that you can use for fold method of lists. The following line would accomplish the same goal, but I don't recommend using this syntax in this course, because it is a bit awkward

```
(0   /: myList)( (acc,h) => acc + h )
```

And if you are super fancy you could write it the following way (but I wouldn't write it this way…):

```
(0 /: myList)( _ + _ )
```

L4D2

- List HOF
- Map
- L4D2_notes.scala

L4D2T1 List HOF

More readings on folding and Lists in Scala

https://coderwall.com/p/4l73-a/scala-fold-foldleft-and-foldright

Additional Reference on HOFs

<lab4_path>/src/main/scala/jsy/lab4/Meeting15-HigherOrderFunctions.sc

In the last reading we looked at folding over lists and neat as it is there are many things that fold/foldLeft cannot easily accomplish. In addition to fold, I have foldLeft and foldRight, map, forEach and exists (and many MANY more)

- fold and foldLeft are basically the same thing. They scan over a list from left to right and apply a call back to each element of the list and a base value to accumulate some value

- collection:List[A].fold (baseValueOfAccumulator: B) { callbackApplied : (B,A) => B }
- collection:List[A].foldLeft (baseValueOfAccumulator: B) { callbackApplied : (B,A) => B }
- Note that the call back is often written { (acc , h) => <stuff> } so that the input of type B is called the accumulator named `acc` and the current head element of the list we are working on is named `h` for head
- Ultimately fold has some strange behavior that is difficult to explain in text. I recommend using foldLeft

- foldRight is similar to foldLeft and fold but it will apply its call back in the reverse order. It scans the List from **right to left** applying the callback to accumulate a value. There is a slight difference in type def:
  - collection:List[A].foldRight (baseValueOfAccumulator: B) { callbackApplied : (A,B) => B }
  - Note that the call back is different. It takes input (_:A, _:B). We often write this as { (h , acc) => <stuff> }
  - Why does fold right's call back take (A,B) rather than (B,A) like fold and foldLeft? I have no idea, but I find it helpful to remember that the accumulator is associated with the fold i.e. folding from the left to the right , acc is the left parameter; Folding from the right to the left, acc is the right parameter of the callback.
- At their root there isn't a huge difference between these 2 they both fold over a list and accumulate an output
  - Example
    ```
    val l = 1 :: 2 :: 3 :: Nil
    l.fold(0){(acc,h) => h+acc} // 6
    l.foldLeft(0){(acc,h) => h+acc} // 6
    l.foldRight(0){(h,acc) => h+acc} // 6
    ```
  - Note that the base value of my accumulator here is 0. The nature of addition is that I can add 0 to a series of addition and it won't effect the computation
- Why such the fuss about fold left and fold right? A part of the answer is : side effects
  - Example
    ```
    val l = 1 :: 2 :: 3 :: Nil
    l.fold(()){(acc,h) => println(h)} // prints :  1\n2\n3
    l.foldLeft(()){(acc,h) => println(h)} // prints : 1\n2\n3
    l.foldRight(()){(h,acc) => println(h)} // prints : 3\n2\n1
    ```
  - Note that the base value of my accumulator here is (). That's called a Unit. It is the output of println(_:A) and is not a super useful value. I have to make sure that the base value of the accumulator and the output of the callback are the same so I chose the only thing of type Unit that doesn't have side effects – like printing.
- Another reason for having two directions for fold is, sometimes, based on what I want to accumulate it is easier to foldRight over the list. For example, often times if I want to accumulate a list I will use foldRight (or I won't use fold at all)
  - Lets use fold to create a list that looks like our original list but where every element is twice what it once was
  - E.g. List(1,2,3) will become List(2,4,6)
  - Lets start by trying this the way we might want it to work
    ```
    val l = 1 :: 2 :: 3 :: Nil
    // this is one of those places where the strange behavior of "fold" is not worth messing
    // with… I recommend just use foldLeft
    l.foldLeft(Nil:List[Int]){(acc,h) => (h*2)::acc}  // List(6,4,2)
    l.foldRight(Nil:List[Int]){(h,acc) => (h*2)::acc} // List(2,4,6)
    ```

        - Note that I am using Nil as the base value of accumulator. Since I want to accumulate a List I use the base value of a list as the base value of the accumulator
        - Note that fold left did successfully double the values of each element of the list but it returned the list in the reverse order of what I wanted
    - Here is how we correct this inversion issue on foldLeft

```
val l = 1 :: 2 :: 3 :: Nil
l.foldLeft(Nil:List[Int]){(acc,h) => acc:::List((h*2))}  // List(2,4,6)
```

        - Note that the body of my call back in foldLeft uses the ':::' operator. This concatenates 2 lists so long as they are of the same type. Note the body also has to cast the h*2 element to a list. This is pretty inefficient. The way to go here would be foldRight (or better yet, use map)

L4D2T1Q1

- Try it yourself. Write a function named revList1 that uses foldLef to reverse a list. Then write a function named revList2 that uses foldRight to reverse a List
    - E.g.

```
val x = List(1,2,3)
revList1(x) == revList2(x) == List(3,2,1)
```

- Look at your solutions. Which one seems like the better solution?

- Now as I mentioned there are other HOFs for lists that are worth knowing about
- Map
    - collection:List[A].map { callbackApplied : (A) => B } // has the type List[B]
    - this will fold over the list, apply the callback to each element of the list **AND** construct a new list with those elements in it
    - This would be the ideal HOF to double all elements of the list (that we wrote above using folds)
        -

```
val l = 1 :: 2 :: 3 :: Nil
l.map{(h) => h* 2 }  // List(2,4,6)
```

    - This has it's drawbacks though. The return type will always be a List of the same length as the list it is operating on.
- Foreach
    - (l : List[A]) . foreach(f : A => Unit) : Unit
    - This doesn't accumulate anything. It just applies f to each element of l.
    - Doesn't seem overly useful in a pure functional language but it could be quite useful in a PL that makes use of side effects
    - ideal for printing the elements in order

```
l.foreach{ (h) => println(h) }
```

- exists
    - (l : List[A]).exists(f : A => Boolean) : Boolean
    - Tells you if any of the elements of the list hold true against f.
    - It does NOT tell you which elements hold true on f
    - does the list contain odd numbers?

```
l.exists{ (h) => (h%2) == 1 }
```

    - Does the list contain negative numbers

```
l.exists{ (h) => h < 0 }
```

- AND Sooooo many more of these are available to us. I think that is enough for today.

L4D2T2: Map

While 'map' is a HOF for Lists Scala also has a data structure called a Map. For clarity, from here on in the course I will do my best to lowercase the HOF `map` and capitalize the data structure `Map`, since Scala takes this syntax as well. A Map (the data structure) can be viewed as a dictionary or a hash table or an object that relates keys to values – a set of (key,value) pairs. Each key must be unique. The values associated with the keys do not have to be unique. By extending the environment with a key that is already named this actually overwrites the value previously associated with that key. The caveat here is that, in Scala, Map[A,B] states that all of my keys must be of type A and all of my values must be of type B. (this is a discussion for another time)…

We have seen this data structure, Map, in our labs already:

- Env : Map[String , Expr]
- TEnv : Map[String , Typ]

Map has a several methods including

- get
    - (m : Map[A,B]).get(a : A) // Option[B]

There are also methods of over the Map data structure that are HOFs

- map
    - (m : Map[A,B]).map(f : ((A,B)) => (C,D)) // Map[C,D]
    - We can write:
        - m.map{ kv => kv match { case (k:A, v:B) => … } }
    - In use we often write cased functions, its some syntax sugar around pattern matching
        - m.map{ case (k:A, v:B) => … }
- mapValues
    - (m : Map[A,B]).mapValues(f : B => C) // Map[A,C]

For examples of these methods and their use please reference testMapHOF() in L4D2_notes.scala

PLEASE be sure to read over that… it has useful information about some dark voodoo magic available in Scala

L4D2T2Q1

Write a function keyCharToStr that takes as input a Map[Char, Int] and returns a Map[String, Int]. This merely up-casts the keys of our input from characters to strings. Use an HOF method of Map in your solution

e.g. keyCharToStr(Map.empty + ('h', 1)) == Map.empty + ("h", 1)

Hint: In scala all Char objects have a method named `toString`.

Solutions

L4D2T1Q1 Solution

- Try it yourself. Write a function named revList1 that uses foldLef to reverse a list. Then write a function named revList2 that uses foldRight to reverse a List
    - E.g.
      val x = List(1,2,3)
      revList1(x) == revList2(x) == List(3,2,1)
- Look at your solutions. Which one seems like the better solution?

// both of these functions that could take as input a List[A] and return as output another List[A] but using an
// A of Int is fine too for the purpose of this exercise

```
def revList1(l:List[Int]):List[Int] = l.foldLeft(Nil)((acc, h) =>  h :: acc )
revList1(x)

def revList2(l:List[Int]):List[Int] = l.foldRight(Nil)((h, acc) => acc ::: (h :: Nil) )
revList2(x)

/*
        Supposing that a function that completes a desired task in as few operations as possible while still be
        human readable is considered desireable/good behavior. It seems to me that revList1 one is a better
        solution to the problem of reversing lists. The call back only needs to create one list. Whereas the
        callback in revList2 has to create 2 lists and uses ':::' list concatenation operation.
 */
```

L4D2T2Q1 Solution

```
def keyCharToStr(m:Map[Char, Int]):Map[String, Int] = {
        m.map{ case (k:Char, v:Int) => (k.toString(), v) }
}
```

L4D3

- p(x1:m1t1,…,xn:mntn):tann => e i.e. Functions
- More on HOFs
- L4D3_notes.scala

L4D3T1 p(x1:m1t1,…,xn:mntn):tann => e i.e. Functions

Do your best with the information provided here to implement logic about Functions into typeof and substitute. I will give you more information in the future to help you double check your work. I want you to try coding it without knowing all the pieces of the puzzle. Then I'll give you more pieces and we'll try coding it again. Please give it about an half hour tonight coding up the logic of Function(_,_,_,_) nodes in the lab 4 interpreter.

Functions can be super useful tools in programming. Our interpreter has been able to handle these since lab 3 but in lab 4 we change them to be able to handle multiple parameters

- Grammar level (modified from pg 6 of Lab 4 handout) functions exist in`e` in the form:
    - p(x1:m1t1,…,xn:mntn):tann => e
- AST level
    - case class Function(p: Option[String], params: List[(String,MTyp)], tann: Option[Typ], e1: Expr) extends Expr
    - fields of the function node
        - p : Option[String]
            - represents the option to name name my functions
        - params: List[(String,MTyp)]
            - I can now easily give my functions multiple parameters
            - Each parameter is a tuple of string and MTyp
            - The string represents the name of the parameter
            - The MTyp object represents the mode and type of the parameter
                - case class MTyp(m: Mode, t: Typ)

        - so a single parameter might look like (xi , MTyp(mi , ti))
        - Note that this is a List and so it might be useful to apply some HOFs to it while implementing the interpreter.
    - tann: Option[Typ]
        - this represents the option to declare an output type for the function
        - Note that if a function is named it is supposed to have an output type (if it doesn't we should through an error)
        - The declared output type represents the expected output type which doesn't always end up being the actual output type. Our type checker will have to figure that out for us and return some kind of error if the declared type and the derived type are not the same.
    - e1: Expr
        - this is the body of the function

In addition, Lab 4 has introduced static time type checking. Functions are values and so they have types. While numbers are represented as TNumber and strings are represented as TStrings… Function types are a bit more fancy.

- Grammar level (modified from pg 6 of Lab 4 handout) in `t` function types look like:
    - (x1:m1t1,…,xn:mntn) => t
- AST level
    - case class TFunction(params:List[(String,MTyp)] , tp : Typ) extends Typ
        - params:List[(String,MTyp)]
            - exactly like parameters in the Function node of the Expr class
            - This is different from how Scala declares function types because now our function types know the names of the parameters in addition to their type.
            - While not necessary for the interpreter, this is to make our work in the lab a bit easier.
        - tp : Typ
            - the output type of the function

We'll need to code up things about function in typeof, substitute and rename.

- Typeof
    - The code gives you some scaffolding for coding these rules. I recommend commenting that out and trying to code it from scratch. Then… bring back the scaffolding and complete what was provided, using your work from scratch as a reference. Then, remove whatever looks the harder to understand. This should help us better understand our code while also getting a sense of different ways to solve the problem.
    - TypeFunction
        - This looks at the type of an anonymous function without a specified return type
        - An acceptable pattern for the input : Function(None,params,None,e)
        - Note that we are returning t' where t' looks like (x1:m1t1,…,xn:mntn) => t
            - That means I will return some kind of TFunction
        - t is calculated in the first premise
        - I recommend trying to code this up before continuing in today's reading
    - TypeFunctionAnn

- Now I have an anonymous function that has a specified return type
- A pattern for the input : Function(None,params,Some(t),e)
- Here I calculate the actual return type of my function
- I must make sure that this is the expected return type. If it is not the same value then I should through a type error on the body of my function (TypeErr_TypeFunctionAnn)
- Alternate form of TypeFunctionAnn

$$\frac{T[x1 \rightarrow t1]...[xn \rightarrow tn] \vdash e : t_{Found} \qquad t_{Expected} == t_{Found} \qquad t' = (x1 : m1\ t1 , \ldots , xn : mn\ tn) \Rightarrow t}{T \vdash (x1 : m1\ t1 , \ldots , xn : mn\ tn) : t_{Expected} \Rightarrow e : t'}$$

- TypeRecFunction
  - I have a named function and it must have a specified expected output type
  - If the function is named and there is no specified output type … throw an error (TypeErr_RecFunctionNoAnn)
  - We calculate the type of the body of the function by assuming that the function body e is calculated in the environment where the function name x is mapped to the functions expected type t' (where the expected t' is declared by the second premise)
  - We then see if the output type found is the same as the expected type
    - If it's not… through a type error on the function body (TypeErr_RecFunction)
  - If it is the same then return the t' that you already made earlier before you looked at the type of the function body
- (TypeErr_RecFunctionNoAnn)
- (TypeErr_RecFunction)
- (TypeErr_TypeFunctionAnn)
- Type*Function*

*** These all require that I extend my provided type environment 'Gamma' with the parameters x_i mapped to their respective types t_i. I recommend doing this by folding over the params field of the Funciton node and accumulate some new type environment using the provided type environment as your base case. If you are not comfortable using fold then write a recursive function in line and iterate on that to complete the work. Before submitting the lab, please rewriting using an HOF ***

- Substitute
  - Use what you know from lab 3 to code substitute. Note that we now have multiple parameters that might indicate a shadowing of the variable that substitute is looking for.
  - The HOF on lists : exist is recommended here
  - The HOF on lists : foldLeft/foldRight might be useful here if exists will not work for you
- Rename
  - Ignore rename for now… I'll talk more about that in the future
- (Step)
  - We cannot step on functions because functions are _____ !

L4D3T2 More on HOFs

This section is mostly designed to be skimmed as opposed to read.

Higher order functions are rather useful in that, if you know how to use them you can accomplish work faster and more legibly than you might otherwise be able to. If you don't believe me, imagine programming without a for loop or a while loop…

We've looked at HOFs that were native methods of the List data structure in Scala. Namely, *fold, foldLeft, foldRight, map, exists* and so on… But these are not unique to Scala's List data type. These are ideas/concepts/philosophies that can be created and applied to data structures as we see fit.

This reading will use the other tools learned so far in the class to explain the construction of HOFs on BinaryTrees (regardless of whether they are ordered trees or not)

Consider the data structure of Binary Trees.

Grammar :

S ::= (S) <- d -> (S) | End
d is a data point

Wouldn't it be great if I could fold over the tree and accumulate a value based on all the data points observed? I think it would be useful.

So lets think about writing a foldRight method for the tree. I must find the right most data value, then the next to right most and so on until I find the left most node of the tree. All the while applying a callback and accumulating a value.

Judgment form	=
~Judment Form~	output = input
Operational semantic	zp = foldRight (S)(z)(cb)

zp: B
S: BinaryTree[A]
z: B
cb: (A , B) => B

Other forms used (we'll consider these to be trivial):

Judgment form	=
~Judment Form~	output = input
Operational semantic	zp = cb(d,z)

zp : B
cb: (A, B) => B
d: A
z: B

Rule1

---

z = foldRight(End)(z)(cb)

Rule2 zp = foldRight (S_{Right})(z)(cb) zpp = cb(d,zp) zppp = foldRight(S_{Left})(zpp)(cb)

zppp = foldRight (S_{Left} <- d -> S_{Right})(z)(cb)

Now to turn it into code….

Operational semantic : zp = foldRight (S)(z)(cb)
Literal types:

… for some type A and some type B…
zp : B
S : BinaryTree[A]
z : B
cb : (A , B) => B

Function definition:

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb: ( A , B ) => B ): B = ???
```

The function body is dependent on looking at the form of the inputs. You'll note that the inference rules, in the conclusion, about the inputs only look at the pattern of S, and ignore the pattern of z and cb.

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        case _ => ???
}
```

Now I want to match on the pattern 'End' since that is the first pattern… but I don't know how to represent that in Scala. 'End' is a lexeme/word from my reference grammar and I have not yet defined the BinaryTree[A] class locally. So lets define the class… This should do fine :

```
/* S ::= S <- d -> S | End */
sealed abstract class BinaryTree[+A] // S
case object Empty extends BinaryTree // End
case class Node[A](l:BinaryTree[A],d:A, r:BinaryTree[A]) extends BinaryTree[A]
        // S<-d->S or rather Sl <- d -> Sr
```

Note that I named the 'End' from my grammar as 'Empty'. This is to reinforce that there is a slight difference from my Grammar to my Code.

Now, again, I want to fill in the body of my function foldRight. I like to start by giving myself the lay of the land… So I'll start with comments to represent the patterns that I need to meet

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        // End
        case _ => ???
        // Sl <- d -> Sr
        case _ => ???
}
```

Then I'll instantiate the patterns using the definition of my data structure.

- End is represented as Empty
- S_{Left} <- d -> S_{Right} is represented as Node(_,_,_) where I can name the things just about whatever I want but I, personally, will choose to use the names l and r because those are commonly used for binary tree data structures i.e. Node(l,d,r)

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        // End
        case Empty => ???
        // Sl <- d -> Sr
        case Node(l,d,r) => ???
}
```

Now to fill in the body of the cases. We'll just go in order of the rules. As a reminder, here is the first rule:

Rule1

z = foldRight(End)(z)(cb)

seems easy enough to code up… just return z, which I literally named z because I oddly named my variables like that.

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        // End
        case Empty => z
        // SLeft <- d -> SRight
        case Node(l,d,r) => ???
}
```

Now for the second rule which looks like this…

Rule2 zp = foldRight (S_{Right})(z)(cb) zpp = cb(d,zp) zppp = foldRight(S_{Left})(zpp)(cb)

zppp = foldRight (S_{Left} <- d -> S_{Right})(z)(cb)

There are a lot of ways to do this. By now I imagine many of you can code it directly but Ill take it in steps… I'll build backwards from my goal… It tells me to return zppp where zppp must have some value declared in the premises

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        // End
        case Empty => z
        // Sl <- d -> Sr
        case Node(l,d,r) => {
                val zppp = ???
                zppp
        }
}
```

the last premise is… zppp = foldRight(S_{Left})(zpp)(cb)… where zpp must have been declared somewhere else and S_{Left} was declared in the conclusion and has already been declared in code as Sl. Cb is also declared in the conclusion and already declared in code.

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        // End
        case Empty => z
        // Sl <- d -> Sr
        case Node(l,d,r) => {
                val zpp = ???
                val zppp = foldRight(l)(zpp)(cb)
                zppp
        }
}
```

the next to last premise… zpp = cb(d,zp)… for some zp declared in a premise

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        // End
        case Empty => z
        // Sl <- d -> Sr
        case Node(l,d,r) => {
                val zp = ???
                val zpp = cb(d,zp)
                val zppp = foldRight(l)(zpp)(cb)
                zppp
        }
}
```

the first premise… zp = foldLeft(S_{Right})(z)(cb)… where all of those variables already exist in scope.

```
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        // End
        case Empty => z
        // Sl <- d -> Sr
        case Node(l,d,r) => {
                val zp = foldRight(r)(z)(cb)
                val zpp = cb(d,zp)
                val zppp = foldRight(l)(zpp)(cb)
                zppp
        }
}
```

and if you are cool you might rewrite the expression to a one liner… but I think the above code is fine too. I wouldn't actually rewrite the code to a one liner unless you find that you are able to read it and your peers can read it as well.

```
{ val zp = foldRight(r)(z)(cb); val zpp = cb(d,zp); val zppp = foldRight(l)(zpp)(cb); zppp }
```

rewrites to

```
{ val zp = foldRight(r)(z)(cb); val zpp = cb(d,zp); foldRight(l)(zpp)(cb) }
```

rewrites to
{ val zp = foldRight(r)(z)(cb); foldRight(l)(cb(d,zp))(cb)}
rewrites to
foldRight(l)(cb(d,foldRight(r)(z)(cb)))(cb)

NOTE: this process only works well because each of my constants are only used once, so in rewriting the expression to a one liner I have not caused any over-evaluation. Be careful rewriting expression so that you do not accidently over evaluate, because, depending on what our `cb` is that could have unintended consequences.

```
// My final solution
def foldRight[A,B](S:BinaryTree[A])(z:B)(cb : ( A , B ) => B ) : B = S match {
        // End
        case Empty => z
        // Sl <- d -> Sr
        case Node(l,d,r) => foldRight(l)(cb(d,foldRight(r)(z)(cb)))(cb)
}
```

Now to test it. I like to test it with derivations over the mathematic model before testing the code directly but I don't like to typeset derivations so I'm not going to do that here… To test this I need to think of some inputs to the function and test them against my expected output. How about we start with an easy one…Summing over the tree

```
// Some variables
val bt : BinaryTree[Int] = Node(Node(Empty,20,Empty),25,Node(Empty,30,Empty))
val myInitAcc = 0 // very important that this is set to not affect your computation
def myCallBack( d:Int , z:Int ):Int = { d + z }

// the call… Ill store the output…
val sumOfbt = foldRight(bt)(myInitAcc)(myCallBack)

// and if I print that output? I expect find the evaluation of 20+25+30 i.e. 75
println(sumOfbt)  // sure enough this prints 75
```

See the code in L4D3_notes.scala for more examples of how we can use this HOF and write other HOF over the data type we defined i.e. BinaryTree.

^ that is a part of the required reading for today.

L4D4

- *Javascripty* Functions continued
- *JavaScripty* Call
- Supplemental reading on Fold
- L4D4_notes.js

L4D4T1 *Javascripty* Functions continued

Short and sweet. Here is some more info on TypeRecFunction rule to be implemented in our type checker.

- TypeRecFunction
    - I have a named function and it must have a specified expected output type

- Acceptable pattern : Function(Some(x),params,Some(t),ep)
- I need to extend my environment to map the function name x to the functions expected type t'
  - extend(_:TEnv,_:String,_:Typ) : TEnv should be useful for this
- The expected return type t' is specified in our last premise as
  (x1 : m1 t1 , … , xn : mn tn) => t
- In AST Typ that is: TFunction(_:List[(String,MTyp)], _:Typ)

I hope this helps. I believe in the scaffolding provided to us we are expected to use env to construct envp where envp either has the environment extension mentioned here, or has no extension to the environment

We then accumulate envpp by folding over the parameters and continually extend the accumulated environment. We will start with envp as our environment.

L4D4T2 *JavaScripty* Call

What use are functions if I am not able to execute/call these functions? Our interpreter can handle function calls!
Note that I can have many arguments at any function call…
When evaluating this I need to be careful to only work on one argument at a time… wouldn't it be great if I had a function that looked through a list and when it finds something of a particular form, it applies a function to that item and returns a new list that leaves everything the same except for that one item we found? … see warm ups from last week

Call sights don't have their own special type because they are not values. But there are type rules out there… they get pretty interesting…
I need all of my arguments to have the type required by the parameters of the function… so that is pretty interesting… especially in code….

- Grammar level (adapted from pg 6 of Lab 4 handout) in `e` call sights have the pattern
  - e0(e1,…,en)
- AST level
  - case class Call(e1: Expr, args: List[Expr]) extends Expr
    - e1 : Expr
      - the sub-expression that is being called.
      - For this to be a reasonable sub-expression e1 better evaluate to a function.
    - args:List[Expr]
      - the arguments that will be applied to the presumed function e1
      - for these to be reasonable the types of my arguments should align with the declared types of the parameters in e1
- Typeof
  - TypeCall
    - T |- e(e1,…,en)
    - Find the type of subexpression e
      - If it is not a function then throw an error
    - Type of e should look like TFunction(params :List[(String,MTyp)] , t:Typ)
      - params :List[(String,MTyp)]
        - this should be the parameter list of our function observed
      - t:Typ

        - the return type of the function
    - see notes on Functions for a deeper understanding of this.
  - look at the types of each of the arguments and make sure that they are the types expected by the functions known parameter type
    - will likely want to zip the params and args lists
      - if the lists aren't the same length then we should through a type error on the arguments
      - It might be easiest to zip just the types of the parameters with the arguments
    - Check that all the arguments have the type that is specified by the parameter
      - Throw a type error if any of the arguments have a type that is not that of the expected type specified by the function parameter
  - Return the return type of the function (found in TFunction)
- Substitute
  - Recurse substitute on everything in a call sight
  - This will require us to call substitute on each of our arguments and return a list of updated arguments. Note that this transforms List[Expr] of some length n to another List[Expr] of the same length n. Wouldn't it be cool if Scala had a construct to do most of that work for me?
- Rename
  - I'll talk more about this later
- Step
  - Here I need to talk a little bit about the concept of reducibility…

For the purposes of our lab reducibility is the ability to step on an expression. Prior to now, when asking is an expression reducible, we asked, is the expression is value. That has worked well for us. But it turns out there are other ways to ask about reducibility. Depending on features of your programming language these other modes of reduction can provide performance gains. We'll talk about these gains as time permits. For now just be aware that other modes exist.

As stated in the Lab 4 handout… page 11… Redex is:

- Operational semantic: m ⊢ e redex
- isRedex(m:Mode,e:Expr):Boolean
  - m : Mode
    - this is either MConst or MName
    - MConst expressions are reducable if they are not yet values
    - MName expressions are not reducable
  - e:Expr
    - the expression I am interested in subject to its mode
    - I want to know if expression 'e' is reducible subject to it's mode
- used in Decl sights and function Calls in the step function
- MName is interesting… it allows me to substitute non-valued expressions into my other expressions. Whereas, MConst requires the actor to be a value.
- The mode can change observed side effects in the evaluation of the expression.
- e.g. mode Cost
  const a = console.log("hi") ; if (true) ? 1 : a → searchDecl, DoPrint
  #Prints "hi"
  const a = undefined ; if (true) ? 1 : a → doDecl
  if (true) ? 1 : undefined → DoIfTrue

1
NOTE : it printed “hi” in the first step

- e.g. mode Name
  name a = console.log(“hi”) ; if (true) ? 1 : a → DoDecl
  if (true) ? 1 : console.log(“hi”) → doIfTrue
  1
  NOTE : it **never** prints “hi”
- Take a moment and attempt to use this information to refine your SearchDecl, DoDecl code in step before moving forward to applying this idea to function calls
- We will talk about the greater implications of reducability in the near future

BACK TO FUNCTIONS

- Step on Functions
  - SearchCall1
    - step on e1
    - (iterateStep will do this until e1 is a function)
  - SearchCall2
    - Step on the leftmost reducible argument.
    - reducibility requires an expression and a mode… the argument would be the expression of interest… the mode of an argument is located in the parameters of the function we hope to apply our argument to.
    - (iterateStep will do this until the argument is non-reducible. It will then step on the next reducible argument… and so on until all my arguments are non-reducible)
    - Wouldn’t it be great if I had a function that could scan a list and apply some function to the first element that it can apply the function to?... (see warm ups from last week)
  - DoCall
    - Now that all the arguments are non-reducible… Accumulate a substituted function body by substituting each of the arguments for their parameter in the body of the function
  - DoCallRec
    - Now that all the arguments are non-reducible… Accumulate a substituted function body by substituting each of the arguments for their parameter in the body of the function
    - Then, substitute the function definition for the function name in the function body

L4D4T3 Supplemental reading on Fold

Here is some addition explanation of what ‘fold’ does on a list.

- The goal of fold is to **scan** the collection and **accumulate** a value.
- It does this by using an element of the collection and the current value of the accumulator as inputs to the callback. This will in turn update the value of accumulator.

Here is a foldLeft function for lists in Scala… It will **scan** our list from left to right while accumulating a value:

- Collection : List[A]
- Accumulator : B
- Callback function: (B,A) => B

```
def foldLeft[A,B](l:List[A])(z:B)(cb:(B,A)=>B):B = l match {
        case Nil => z
        case h :: t => foldLeft(t)(cb(z,h))(cb)
}
```

Here is a foldRight function for lists in Scala… We often think of this as **scanning** over the list from right to left to accumulate a value … That's a fine way to conceptualize it but we should note that it's not actually what happens. In Scala List is head accessible only. foldRight actually will **scan** our list from left to right while accumulating an expression that can be used to accumulate a value by observing the values of the list from right to left :

- Collection : List[A]
- Accumulator : B
- Callback function: (A,B) => B

```
def foldRight[A,B](l:List[A])(z:B)(cb:(A,B)=>B):B = l match {
    case Nil => z
    case h :: t => cb(h,foldRight(t)(z)(cb))
}
```

L4D5

- *Javascripty* Objects
- Rename
- Supplemental MapAB

L4D5T1 *Javascripty* Objects

Context : Objects are a data type native to JavaScript that are quite useful in practical JavaScript. As such we decided it would be nice to have these in our object language *javascripty* we mostly copied the data form from JavaScript there are some nuances though in that our Lab 4 *javascripty* Objects are non-mutable. (that nuance will change in Lab 5)

- Grammar level (Lab 4 handout page 6)
    - Expression
        - $\{ f_1 : e_1 , \dots , f_n : e_n \}$ exist in e
    - Type (modified a bit)
        - $\{ f_1 : \tau_1 , \dots , f_n : \tau_n \}$ exist in τ
- Ast level (ast file of lab 4)
    - Expr
        - case class Obj(fields: Map[String, Expr]) extends Expr
            - Obj(_) contains a mapping of Strings to abstract expressions of *javascripty* i.e. Expr
            - fields: Map[String, Expr]
                - just like real JavaScript Objects have unique field names, Scala's Map data structure has unique field names.
                - The String would be the Key
                - The Expr is the Value mapped to that key (NOTE: This Expr does not have to represent a valued expression… I am saying Value with respect to the definition of Maps having key-value pairs, not value with respect to the grammar of our lab)
            - e.g.
                - parse(" { x : (1+1) , y : (10/2) } ")
                - Obj(
                Map(

```
                                    x       ->      Binary(Plus,N(1.0),N(1.0))   ,
                                    y       ->      Binary(Div,N(10.0),N(2.0))
                                )
                            )
```

  - Typ
    - Some objects are values.
    - similar to how we declared TFunction, we must have a type for objects. We called it TObj(_) and filled it in with just the right about of information.
    - case class TObj(tfields: Map[String, Typ]) extends Typ
      - TObj(_) contains a mapping of Strings to abstract types of *javascripty* i.e. Typ
      - tfields: Map[String, Typ]
        - The String would be the Key
        - The Typ is the Value mapped to that Key

- TypeOf
  - TypeObject
    - For each of the field_expression pairs in the object ' $f_i : e_i$ '
    - Find the type of the expression
    - Map the field f_i to the type of the expression e_i, namely t_i
    - Observe. In the code provided Obj(fields) where fields is a map of order/length 'n' we should return TObj(tfields) where tfields is also a map of order 'n'. This should give you a good hint about a higher order function will help us complete our code.
    - Also note that a portion of our work is to transform types Map[String,Expr] -> Map[String,Typ] … the keys remain unchanged in type and they remain unchanged in their respective names(the key's value, not to be confused with the value the key maps to)… so while the higher order function 'map' would work, there is actually a far better option than map for this task.
- Substitute
  - Note, in *javascripty,* that field names are not variable… a fields name will **NEVER** be able to shadow a variable
  - Each thing of type Expr in Obj(fields) might however contain the variable of interest… so I should recurs substitution over those sub-expressions
- Rename
  - Saving for later… See notes on rename for today's reading and work on the simple cases before attempting rename objects
- Step
  - Not all objects are values… if there exists a single f:e pair in the object s.t. the e is non-valued, then the parent object itself is not a value. Once the object is comprised only of f:v pairs then the Object is a value. iterateStep must be able to evaluate non-valued-objects to valued-objects in a deterministic manner
  - SearchObject
    - Make sure that the object is not a valued object
      - If it is a valued object then you should throw a Stuck Error, it means that some other part of the step function has a bug
    - Locate the first non-valued expression e_i
    - Step on that expression to create some e_i'
    - Return something that looks a lot like the old object but having updated e_i for the found e_i'
    - 'find' is a useful HOF for this that I haven't mentioned before now in the readings:

- (m : Map[A,B]).find(f : ((A,B)) => Boolean) : Option[(A,B)]
- returns Some((a,b)) on the **first** (a,b) that f finds true
- returns None if there are no (a,b) that f finds true

L4D5T2 Rename

In lab 4 we introduce the concept of call-by-name and examine how this is different from call-by-value. I will explain this with respect to the following definition of substitute. Substitute has 3 parameters e, esub, and x and returns e with all free instances of x replaced with esub. Here we name the second parameter esub rather than vsub to denote that our substitution does not have to be performed with valued actors.

Until you have a working interpreter with typeof, substitute, and step fully implemented use this following stub for substitution:

```
def substitute(e: Expr, esub: Expr, x: String): Expr = {
  def subst(e: Expr): Expr = e match {
    case Print(e1) => Print(subst(e1))
    case _ => ???
  }
  subst(e)
}
```

When you are ready… there are 2 things that we must tackle….

```
def substitute(e: Expr, esub: Expr, x: String): Expr = {
  def subst(e: Expr): Expr = e match {
    case Print(e1) => Print(subst(e1))
    case _ => ???
  }
  val fvs = freeVars(???)
  def fresh(x: String): String = if (???) fresh(x + "$") else x
  subst(???)
}
```

fist… I need to figure out all these darn ??? left at the bottom of the implementation of substitute.

Consider the expression :
name foo = x + 2 ; foo

Here is how it should evaluate (imagine the expression managed to pass the type checker)…
name foo = x + 2 ; foo → // DoDecl
x + 2 → // SearchBinary1, StuckErr
StuckErr

Note that x is a free variable in the expression that is bound by name to the variable foo.

Now consider this expression:
name foo = x + 2 ; (true) ? 5 ; foo

Here is how it should evaluate (imagine it managed to pass the type checker)…
name foo = x + 2 ; (true) ? 5 ; foo → // DoDecl
(true) ? 5 ; x + 2 → // DoIfTrue
5

For this reason, it is important that I look through the binding of foo for its free variables prior to substitution.

For a variety of reasons, I will be rather hand wavy here and tell you that the fresh function should only "freshify" its input 'x' if that variable was free in esub expression i.e. fresh should only recurse on its input if the input exists in the set of the free variable of esub.

After that… I then need to use that knowledge to inform how I will write the rename function.

Rename is designed to rename all free variables in the provided expression to unique names

Consider the expression :
const a = (const a = 1; a); (const a = 2; a)

Here, I used the variable name `a` quite a lot. But many of them are not representative of the same `a`. The expression ought to read a bit more like :

const a0 = (const a1 = 1; a1); (const a2 = 2; a2)

Using the fresh function provided above if 'a' were an x of interest we would get

const a$ = (const a$$ = 1; a$$); (const a$$$ = 2; a$$$)

Rename works quite a lot like substitute in that I mostly just recurs the function on all subexpressions found.

The meat of the work for rename is accomplished in the following cases:

case Var(y) => ???
case Decl(mode, y, e1, e2) => ???
case Function(p, params, retty, e1) => ???

The work is dependent on what exists in my variable environment. We will talk more about this at a later date.

L4D5T3 Supplemental MapAB

Here are some additional notes on the Map data type

- Map[A,B]
    - This is a data type native in scala that relates Keys to Values.
        - The Keys are unique
        - The Values are not necessarily unique
    - Often viewed with type Map[K,V]… I prefer to think of it as Map[A,B]
        - K/A represents the type of all unique Keys in the map
        - V/B represents the type all Values in the map
    - This data structure can be viewed as a variant of a dictionary and hash table
    - We have seen this data type previously

        - Lab 2 & Lab 3 our data type Env is actually a Map[String, Expr]
        - Lab 4 our data type TEnv is actually a Map[String, Typ]
        - These also had some extensions to the original Map[A,B] data type
        - empty = Map.empty
        - extend(m,x,v) = m + (x -> v)
        - lookup(m,x) = m(x)
    - We abstractly represent these in the notes as
        - [ ] the empty map
        - [<some_key> ↦ <some_value>] The map with one extention
        - [“x” ↦ 2 , “y” ↦ 5] example
            - Map[String,Int]
            - Mapped the string “x” to the Int 2
            - Mapped the string “y” to the Int 5
    - The code literals are as follows
        - Map.empty
            - [ ]
        - Map.empty + (<some_key> -> <some_value>)
            - [<some_key> ↦ <some_value>]
            - Map(<some_key> -> <some_value>) is also acceptable
        - Map.empty + (“x” -> 2) + (“y” -> 5)
            - [“x” ↦ 2 , “y” ↦ 5]
            - Map (“x” -> 2 , “y” -> 5) is also acceptable
    - As previously mentioned, each key is unique. If I attempt to extend [“x” ↦ 2 , “y” ↦ 5] with [“y” ↦ 3] Scala will overwrite the original definition of y
        - Map.empty + (“x” -> 2) + (“y” -> 5) + (“y” -> 3)
            - [“x” ↦ 2 , “y” ↦ 3]
        - other languages handle this update differently… Potentially throwing an error.

L4D6

- Map access
- *javascripty* GetField
- L4D6_ObjCont.scala

L4D6T1 Map access

To keep with the notation of the course, `e: t` denotes that expression `e` has type `t`

Let
- m: Map[A,B]
- key: A
- e: B

We observe that
- m(key): B
- m.get(key): Option[B]
- m.getOrElse(key, e): B

Here is the output of some tests that I ran for a Map[String,Int]

```
scala> val m = Map("x" -> 2,"y"->10/2)
m: scala.collection.immutable.Map[String,Int] = Map(x -> 2, y -> 5)

scala> m("y")
res0: Int = 5

scala> m("z")
java.util.NoSuchElementException: key not found: z
  at scala.collection.immutable.Map$Map2.apply(Map.scala:129)
  ... 29 elided

scala> m.get("y")
res2: Option[Int] = Some(5)

scala> m.get("z")
res3: Option[Int] = None

scala> m.getOrElse("y",3)
res4: Int = 5

scala> m.getOrElse("z",3)
res5: Int = 3
```

Take aways:

- m(key):B … accessing the Map directly using m(key) can be problematic, because, if key is not in m then this expression will throw an error. To make this safe in code we would need to set up a try-catch block of code which is often ill-advised (depends on the language and the system and lots of things… but in Scala, like it's parent language - Java,not desired behavior in code)
- m.get(key):Option[B] … safer than m(key) but returns an option type which might not be desired
- m.getOrElse(key,<e:B>): B is pretty useful
    - it will attempt m.get(key) : Option[B]
        - if it returns Some(b:B) then getOrElse returns b
        - if it returns None then getOrElse returns the valued of e

L4D6T2 *javascripty* GetField

Context: Objects are awesome in their own right but what makes them particularly useful is when I have the ability to access the object using operators such as get field. And accordingly we extended *javascripty* to include this operator on objects. For any expression of the form $e_1.f_{seek}$, if e_1 evaluates to value v_1, and v_1 is an object that contains the field f_{seek} mapped to some value v_{seek}, then $e_1.f_{seek}$ evaluates to v_{seek}.

```
e.g. { x : 2 , y : 5 }        // a value
e.g. { x : 2 , y : 5 } . y    // an expression that be evaluated to the value 5
```

- Grammar level
    - Expressions
        - e_1 . f exists in e
    - types
        - by nature of this being an operation it does not exists in the language of types

- AST level
  - Expr
    - case class GetField(e1: Expr, f: String) extends Expr
      - e1: Expr
        - expression that ideally evaluates to a valued object expression
        - the expression that I will search over for f
      - f: String
        - field of interest
  - Typ
    - Since it's not in the languages of τ, I don't need to extend the definition of this AST.
- Type checker
  - TypeGetField
    - Get the type of e1, t1 and see if t1 has an object type (i.e. TObj(_))
      - If t1 is not an object throw an error on this subexpression
    - See if f exists in t1 and if it does return the type associated with f
      - If f is not in the object found then throw and error
- Substitution
  - Fields are different from variables and so we don't need to worry about f shadowing anything
  - e1 might contain a free instance of x…
- Renaming (don't worry about this for now)
- Small step interpreter
  - SearchGetField
    - Step on e1 to discover e1' and return a GetField(_,_) object that contains e1' along with the f that we were looking for
  - DoGetField
    - For expressions of the form $e1.f_{seek}$
    - if e1 is a value v1 where that value is a valued-object o1 (an object such that all subexpression of the object are values)
      - if v1 is not an object, throw stuck error
      - if o1 is not a valued object then I am actually in SearchGetField (note, you might consider implementing these under a single case body)
    - find what f_{seek} is mapped to in o1, lets call that v_{seek}
      - If f_{seek} is not a key of o1 then return a stuck error
    - return v_{seek}

L4D7

- L4D7_getFieldCont.scala

L4: Additional Resources

L4: Closing Thoughts