Lab 5: Monads for Mutability

Table of Contents

L5: Preface

L5D1

- Print out read and annotate the lab 5 handout
- Review csci3155_course_notes chapter 4
- Rename

- State Monads and DoWith

L5D1T1 Rename

In the previous lab we looked at the rename function and a few of you likely wondered… why? Do I actually need this in my interpreter. That was a great question… you didn't actually need rename for that interpreter. We simply wanted to introduce you to the idea of free variable capture avoidance prior to working with free variable capture avoidance over state monads

L5D1T2 State Monads and DoWith

It is highly advised that you lightly skim this topic once… take a break… and then actually read this topic…

I won't sugar coat it... Welcome to what is often considered the most challenging lab of the course! The lab itself is not overly challenging relative to other labs but this requires a bit of mental effort on our part to learn a new paradigm in programming over a data structure known as a state monad. To simplify this matter, we will ignore how to create these state monads and simply work on using a premade instance of it that we have named "DoWith".

So far in your career as budding software engineers, or however you self-identify, you have likely seen a number of somewhat advanced data structures, including but not limited to: balanced binary trees, optimized hash tables, B+ Trees, Tri, and abstract syntax trees. I will now introduce you to one that you have likely never seen (unless you regularly program in functional language). I present to you the state monad.

One of many monads, the state monad is a construct that can be used to remove pesky side effects from our computation. We will interpret our expressions independent of state/memory. We will write our expressions to return a function… then, if we choose, we could evaluate that function with any input we'd like, denoting the initial state/memory for evaluation. *I know… that's wild! Take it at face value and move on… we will look at it deeper after we have had more exposure to this way of thinking/computing.*

Consider the following Scala script representing a monad from type `A` to type `A`:

```
def my_monad[A](x:A):A = {
        println(x)
        x + 2
}
```

suppose I want to complete the following function so that it can take in a monad from A to A and update the body of that monad using the callback cb

```
def map[A](monad:(A)=>A)(cb:(A)=>A ):(A)=>A{
        ???
}
```

e.g. map(my_monad)((a) => a + 3) should evaluate to (x : A):A => (println(x); x + 2) + 3 and it should not print anything while constructing this output. This should also not change my_monad in anyway, simply create a new monad that looks similar to my_monad. Note that this returns something like my_monad but it has appended the expression '+ 3' to the original body of my_monad.

Take a moment to think about how this could be accomplished…

Or, just be lazy and move on…

Doing this directly is pretty complicated. Scala has some built in operators that help us accomplish this task known as 'for_yield' expressions. These are cool and all but, we won't cover these in this course. We will look at methods built for us around our specific datatype, DoWith, namely 'map' and 'flatMap'.

If you want to learn about 'for_yield', go right ahead, I won't stop you.

We created a state monad named DoWith it is, in essence a function (A) => (A,B). Why? In lab 5 we finally introduce mutability to our object language *javascripty*. Mutability is often achieved by altering the state of memory over time. But we are programming in the functional subset of Scala and so we can't have mutability in our meta-language Scala. So, using some rather complex maths, we abstract the state of memory using the monad DoWith. Until now we have evaluated expressions by evaluating the expressions based on environments (big-step) or rewriting expressions again and again until you find a value (small-step). Now we will consider expressions to be DoWiths of the form DoWith[Mem, Expr] and I will rewrite the DoWiths until the body Expr of my DoWith is a value. We can thing of a DoWith[Mem, Expr] as a function stating, if I knew the state of memory(Mem) then I could derive the state of memory(Mem) I would return as well as the expression(Expr) that I would return.

DoWith[A, B]: (A) => (A,B)
In this course we often work with DoWith[Mem, Expr]: (Mem) => (Mem, Epxr), which we can think of as a function of the form (initialState) => (newState , expression)

A few methods we should know about

- Map
  - Call form <DoWith[A, B]: (A)=>(A,B)> map <(B) => C>
  - Return type DoWith[A, C]: (A)=>(A,C)
  - Use Used to find the expression of the dowith and potentially do work on that expression
- flatMap
  - Call form <DoWith: (A)=>(A,B)> flatMap < DoWith: (C) => (C,D) >
  - Return type DoWith: (C) => (C,D)
  - Use Used when nesting map structures…
- Doget
  - Value form doget
    - Note this is a value… not a method of DoWiths
  - Return type DoWith: (A) => (A,A)
  - Use Used to find the state of memory
- Doput
- doreturn
- domodify

Consider the following stub provided to you for the step function…

```
def step(e: Expr): DoWith[Mem, Expr] = {
    e match {
        case Print(v1) if isValue(v1) => doget map { m => println(pretty(m, v1)); Undefined }
        case Print(e1) => step(e1) map { e1p => Print(e1p) }
    }
```

}

This relates to the inference rules DoPrint and SearchPrint provided in the lab handout

I'll rewrite them here with some comments in a way that we might be more friendly to read for your first introduction to state monads. We will look at this again later after we have played with DoWiths a bit more

```
def step(e: Expr): DoWith[Mem, Expr] = {   e match {
        // DoPrint
        case Print(v1) if isValue(v1) => {

                // a dowith that I can apply map to in order to know the state of memory
                val dw_mem = doget

                // a function that, if provided a state of memory,
                // can print value v1 subject to the state of memory
                // and then,
                // return the AST for undefined.
                def printSubjectToExpressionFieldThatHappensToRepresentMemory(m) = {
                        println(pretty(m, v1));
                        Undefined
                }

                // map on the dowith that knows the state of memory
                // use that memory as input toprintSubjectToExpressionFieldThatHappensToRepresentMemory
                // construct a new DoWith
                // using the memory from dw_mem
                // but a new expression from printSubjectToExpressionFieldThatHappensToRepresentMemory
                val dw_ep = dw_mem map printSubjectToExpressionFieldThatHappensToRepresentMemory

                // return the do with
                dw_ep
        }

        // SearchPrint
        // do to the nature of pattern matching,
        // I can assume here, that e1 is not a value
        // recall that my first case was `case Print(v1) if isValue(v1) => …`
        case Print(e1) => {
                // A DoWith that knows the state of memory and the expression
                // found by taking a single step on expression e1
                val dw_e1p = step(e1)

                def constructNewExpressionSubjectToExpressionField (e1p) = {
                        Print(e1p)
                }

                val dw_print_e1p =dw_e1p map constructNewExpressionSubjectToExpressionField
```

```
                dw_print_e1p
        }

}       }
```

Now, let's look at some examples of how step works

Consider the expression where this state monads are not very necessary:

1 + 2

Presupposing this passes the type checker… The evaluation looks a bit like this:

dw(m:Mem) => (m , 1 + 2)

→ // DoArith(Plus)

dw(m:Mem) => (m , 3)

Now, consider the expression :

var x = 1 ; x = x + 2 ; x

Presupposing this passes the type checker… The evaluation looks a bit like this:

dw(m:Mem) => (m , var x = 1 ; x = x + 2 ; x)

→ // DoDecl(VarBind) + (Rename would be applied in practice but it wouldn't do anything really)

dw(m:Mem) => (m + (a0 -> 1) , *a0 = *a0 + 2 ; *a0)

→ // SearchAssign2, SearchBinary1, DoDeref

dw(m:Mem) => (m + (a0 -> 1) , *a0 = 1 + 2 ; *a0)

→ // SearchAssign2, DoArith(Plus)

dw(m:Mem) => (m + (a0 -> 1) , *a0 = 3 ; *a0)

→ // DoAssignVar

dw(m:Mem) => (m + (a0 -> 3) , *a0)

→ // DoDeref

dw(m:Mem) => (m + (a0 -> 3) , 3)

we found a value in the second part of the output….

Now… if I wanted I can call that do with using any initial state of memory. Ill often call it with the empty map to denote that memory was empty when we begin our computation… But I might do this differently depending on my goal

dw(m:M) => (m + (a0 -> 3) , 3) (map.empty) evaluates to (Map(a0 -> 3) , 3)

L5D2

- Map with Lists
- Unary Step

L5D2T1 Map with Lists

Use the information here to work on mapWith over lists. Then attempt to complete mapWith over Maps as well as MapFirstWith given what you have learned. Refer back to L5D1T2 for a refresher on the basics of DoWith[A, B]

Here is the provided stub for mapWith over Lists.

```
def mapWith[W,A,B](l: List[A])(f: A => DoWith[W,B]): DoWith[W,List[B]] = {
    l.foldRight[DoWith[W,List[B]]]( _ ) {
        ???
    }
}
```

I want to fold over input `l` and accumulate a DoWith that encapsulates the List that I would find by applying `f` to each element of `l`.

Lets start by removing the DoWith's from this work….

L5D2T1Q1

```
def map[A,B](l: List[A])(f: A => B): List[B] = {
    l.foldRight[List[B]]( _ ) {
        ???
    }
}
```

Complete the above script to accumulate a List[B] by applying `f` to each element of `l`. See solution, with explanations of the logic below at L5D2T1Q1. Attempt this problem and review that solution before continuing.

Now to apply that logic to the actual mapWith function. Let's fill in the _ to represent the base case for our return expression. I want to return a DoWith[W,List[B]]. Which means, I want to return the base case for List[B] encapsulated by a DoWith. The base case for a list is Nil. I can wrap this inside a DoWith using the doreturn method.

```
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 )    ) {
                ???
        }
}
```

Now onto the ??? which represents a function let's start by identifying the parameters for this function. (h,acc) would be fine but I like to give them names that represent their types. This function has the type (A , DoWith[W,List[B]]) => DoWith[W,List[B]]. So, I'll name it as follows

```
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 )    ) {
                ( a , dw_w_lb ) => ???
        }
}
```

I now need to do something about applying f to a to construct dw_w_b. I'll need to crack that open to extract the b. I must also crack open dw_w_lb to extract lb so that I can prepend b to lb. This requires the use of both map and flatMap. See note about flatMap below. See stub for this function at the bottom of this document.

Note :

```
dw_a_b flatMap b => dw_a_c : DoWith[A,C]
        dw_a_b : DoWith[A,B]
        b => dw_a_c : B => DoWith[A,C]
                b : B
                dw_a_c : DoWith[A,C]
```

often seen in structures of the form

```
dw_a_b flatMap b => dw_a_c map cp : DoWith[A,C]
        dw_a_b : DoWith[A,B]
        b => dw_a_c map cp : B => DoWith [A,C]
                b : B
                dw_a_c map c => cp : DoWith [A,C]
                        dw_a_c : DoWith [A,C]
                        c : C
                        cp : C
```

Stub for mapWith on Lists

```
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 )    ) {
                ( a , dw_w_lb ) => {
                        val dw_w_b = f(a)
                        val dw_w_lb_p = dw_w_lb flatMap { lb =>  dw_w_b map { b => ??? } }
                        //val dw_w_lb_p_alt = dw_w_b flatMap { b => dw_w_lb map { lb => ??? } }
                        dw_w_lb_p // or dw_w_lb_p_alt... I don't think it makes a difference
                        // for this particular example
                        // Though if it does matter sometimes.
                        // I believe that dw_w_lb_p_alt is actually the preferred flow of control.
                }
        }
```

```
}
```

L5D2T2 Unary Step

While you complete this reading you might find it beneficial to attempt all unary cases of step (and write a test case for each rule) using this as a reference

The step function is defined as following in lab 5: def step(e: Expr): DoWith[Mem, Expr]

So, looking at the type of step I see that given an Expr I must return a DoWith.

Consider inference rules of the form:

$J_1 \ \ldots \ J_n$

<M,e> → <M’,e’>

Where J_i is a judgment and a premise of the inference rule for all 1 <= i <= n

The e in the conclusion, left of the judgement form → is the expression passed as input to step

DoNeg

So consider the following rule found on page 13 of the lab

DoNeg
n' = -n

< M , -n > → < M , n’ >

The expression *e* as input to step is *–n*.

So suppose I begin step as follows

```
def step(e: Expr): DoWith[Mem, Expr] = e match {
    // DoNeg
    case _ => ???
}
```

I want to fill in the _ to represent *–n.* How do I do that? Well `– ` is a unary operation so the Expr will start with Unary. The – specifically is the Neg operation. n is defined in my grammar on page 7 of the handout as a number and the AST file defines numbers as N(_:Double)

```
def step(e: Expr): DoWith[Mem, Expr] = e match {
    // DoNeg
    case Unary( Neg , N( n ) ) => ???
}
```

Now what about the ??? portion, the body of my case statement? This needs to represent all the logic of my premise as well as the stuff to the right of the judgment form in the conclusion. It also needs to have the type

specified by the definition of step i.e. DoWith[Mem, Expr]. Here is a stub that I created to help you with this. Obviously, there are more direct ways of completing this task

```
def step(e: Expr): DoWith[Mem, Expr] = e match {
        // DoNeg
        case Unary( Neg , N( n ) ) => {
                val np:Double = ???  // You ought to know this one already
                val e_np:Expr = ???  // You ought to know this one already
                val dw:DoWith[Mem,Expr] = ???  // this one is less obvious
                dw
}       }
```

Now the tricky part here is how do I construct the DoWith. I need to construct a DoWith[Mem, Expr] such that state:Mem has not changed. Since the state of our DoWith is not changed doput and domodify are definitely not needed for our solution.

Consider the following examples of doget and doreturn

```
doget : DoWith[Mem,Mem]

doget map f : DoWith[Mem,Expr]
        doget : DoWith[Mem, Mem]
        f : (Mem) => Expr

doreturn(e) : DoWith[Mem,Expr]
        e : Expr
```

doget alone does nothing of real value for us. But, when combined with map we can know something about the state of memory. This would be very useful if I need to do anything with memory (like how I did in DoPrint). But if I don't need to read or write to memory then the ***doget map f*** pattern is sort of overkill. I might consider just using doreturn.

TESTING

We've looked at a stub for DoNeg and we've maybe already tried to implement a solution. How do we test that our implementation works? I leave it to you to figure out where you want to write these tests. I personally would choose a high granularity. I would have a suite that tests step and in that suite I'd place a suite to test unary operations and finally under unary operations I would write these tests for UnaryNeg/DoNeg. This way I have many options as to which test suite to use. Recall there are many ways to write tests. I prefer the following format:

Format 1

```
"thing" should "do stuff" in {
        assertResult(<expected output>) {
                <test expression>
        }
}
```

The test expression here is a step on some javascripty expression of the form –n. one such expression is – 5. -5 is written as an Expr as Unary(Neg,N(5)) . If you don't believe me, go to the worksheet and type parse("-5")

The expected output is a DoWith that encapsulates -5. So if I want to get the -5 out of the expression I must execute my DoWith with some initial state of memory (empty memory will do fine) and then find the second value of the output tuple.

```
"DoNeg" should "return -5 " in {
        assertResult(N(-5.0)) {
                step(Unary(Neg, N(5)))(memempty)._2
                // val dw_m_ep = step(Unary(Neg,N(5)))
                // val m_ep = dw_m_ep(memempty)
                // val ep = m_ep._2
                // ep
}       }
```

Please note that a "DoNeg" test already exists. You might want to write the above test as `**it** should "return -5 " in { … }` rather than `**"DoNeg"** should "return -5" in {…}`

You might go so far as to write a double step test that takes – (- (5)) → - (- 5) → 5 But note that this should not pass until after you have implemented SearchNeg

```
"SearchNeg & DoNeg" should "return the negation of a number value" in {
        val e1 = Unary(Neg,Unary(Neg,N(5)))
        val e2 = Unary(Neg,N(-5))
        val e3 = N(5)
        // <_, e1> → <_, e2>
        assertResult(e2){
                step(e1)(memempty)._2
                // val dw_m_ep = step(e1)
                // val m_ep = dw_m_ep(memempty)
                // val ep = m_ep._2
                // ep
        }

        // <_, e2> → <_, e3>
        assertResult(e3){ step(e2)(memempty)._2 }

        // we can also bring it together in a single assertion, e1 steps to e3 in two steps
        // <_, e1> →² <_, e3>
        assertResult(e3)(step(e1).flatMap{ (e1p) => step(e1p) }(memempty)._2)
}
```

SearchNeg

Lets look at performing SearchNeg.

$$\frac{< M , e_1 > \rightarrow < M' , e_1' >}{< M , uop\ e_1 > \rightarrow < M' , uop\ e_1' >}$$

pattern to match on uop e_1 must be a Unary node. uop is a variable for any Uop. e_1 is a stand in for any expression (implicitly a non-valued expression). This is written as an Expr of ***Unary(uop,e1)***. Here is a possible stub expression

```
def step(e: Expr): DoWith[Mem, Expr] = e match {
        // SearchUnary
        case Unary(uop,e1) => {
            val dw_mp_e1p = ???
            val dw_mp_uope1p = ???  // difficult part
            dw_mp_uope1p
}       }
```

I'll help out with the difficult part by adding another puzzle piece to the stub. I want to create dw_mp_uope1p using dw_mp_e1p. In order to maintain my state 'mp' I'll want to map on dw_mp_e1p as seen in the bellow stub:

```
def step(e: Expr): DoWith[Mem, Expr] = e match {
        // SearchUnary
        case Unary(uop,e1) => {
            val dw_mp_e1p = ???
            val dw_mp_uope1p = dw_mp_e1p map { ( <optionToNameParam> ) => ??? }
            dw_mp_uope1p
        }
 }
```

Again… there are more direct ways of writing this. Complete the binding for dw_mp_e1p to represent the premise of the inference rule < M' , e_1' >. Then map on < M' , e_1' > with a function that does something with e_1'. A great variable to use in place of the <optionToNameParam> would be e1p

Note :

< M , e > map { (e) => *stuff_of_e* } returns < M , stuff of e >

where the e in the second argument to map is actually going to be the e from the first parameter (the e in the state monad)

DoWith[Mem,Expr] map f : DoWith[Mem, **Expr**]

f : Expr => **Expr**

In bold red I have denoted that the output expr is defined by the function f

L5D2T2Q1 complete the stub

```
def step(e: Expr): DoWith[Mem, Expr] = e match {
        // DoNeg
        case Unary( Neg , N( n ) ) => {
            val np:Double = ???  // You ought to know this one already
            val e_np:Expr = ???  // You ought to know this one already
            val dw:DoWith[Mem,Expr] = ???  // this one is less obvious
            dw
}       }
```

Solutions

L5D2T1Q1 Solution to simplified map with on lists

Below is one explanation of how to derive a solution to the problem. There are many other ways to think about and solve this problem.

```
def map[A,B](l: List[A])(f: A => B): List[B] = {
    l.foldRight[List[B]]( _ ) {
        ???
    }
}
```

the _ must represent the base case of my output. I want to construct a List[B] so the base case would be the base case of a list i.e. Nil

```
def map[A,B](l: List[A])(f: A => B): List[B] = {
    l.foldRight[List[B]]( Nil ) {
        ???
    }
}
```

Now the ??? must represent a function from list element a and accumulated list of b to list of b. first lets get some names for the inputs following the expected inputs to f for foldRight

```
def map[A,B](l: List[A])(f: A => B): List[B] = {
    l.foldRight[List[B]]( Nil ) {
        ( h , acc ) => ???
    }
}
OR
def map[A,B](l: List[A])(f: A => B): List[B] = {
    l.foldRight[List[B]]( Nil ) {
        ( a , lb ) => ???
    }
}
```

Now transform the thing of type A to B and prepend it to the thing of type List[B]

```
def map[A,B](l: List[A])(f: A => B): List[B] = {
    l.foldRight[List[B]]( Nil ) {
        ( h , acc ) => {
            val hp = f(h)
            hp :: acc
        }
    }
}
OR
def map[A,B](l: List[A])(f: A => B): List[B] = {
    l.foldRight[List[B]]( Nil ) {
        ( a , lb ) => {
            val b = f(a)
            b :: lb
        }
    }
}
```

L5D2T2Q1 solution: complete the stub DoNeg

```
def step(e: Expr): DoWith[Mem, Expr] = e match {
        // DoNeg
        case Unary( Neg , N( n ) ) => {
                val np:Double = -n
                val e_np:Expr = N(np)
                val dw:DoWith[Mem,Expr] = doreturn(e_np)
                dw
                // doreturn(N(-n))
}       }
```

L5D3

- mapWith on Map[C, D]
- mapFirst
- binary step
- isBindex
- typeOf

Not to worry, while there are a lot of topics today, the readings are fairly short and provide test cases for us to consider.

L5D3T1 mapWith and mapFirst

Building off yesterday's discussion and reading. In addition to wanting to map over Lists to construct a DoWith… we might also want to map over Maps to construct a DoWith. See the below stub provided in our lab:

```
def mapWith[W,A,B,C,D](m: Map[A,B])(f: ((A,B)) => DoWith[W,(C,D)]): DoWith[W,Map[C,D]] = {
    m.foldRight[DoWith[W,Map[C,D]]]( ??? /* BLANK 1 */ ) {
        ??? /* BLANK 2 */
    }
}
```

This takes as input a map m and a function f. The goal is to create a function similar to the map method of Map data structures that will apply f to each element of m to accumulate a new map. Here we must apply f to each element of m but we need to construct a DoWith that encapsulates a map. Note that the function f takes an element of m as input and returns a DoWith as output.

/* BLANK 1 */

With respect to the foldRight method of Map[A,B] this is an argument that denotes the base case of my computation. This represents the begging value of what I will accumulate. I often find it useful to answer the question : "*What do I return if the collection is empty?"*. I need to return something of the type DoWith[W,Map[C,D]] i.e. a dowith that encapsulates some map m'. If my initial m is an empty map then I might want m' to also be an empty map (if we apply f to each element of an empty map we would find an empty map, because there is nothing to apply f to).

Syntax Tips :

Map() and Map.empty will both construct an empty map

doreturn(a:A) : DoWith[W,A]
a call to doreturn(a) with some a:A will create a new DoWith of the type DoWith[W,A] for any type A (even if the type A happens to be the type Map[C,D] for some type C and D)

/* BLANK 2 */

With respect to the foldRight method of Map[A,B] this is an argument that denotes the call back that is applied at each recursive call to foldRight. In general it would have the following type ((A,B),[DoWith[W,Map[C,D]]]) => [DoWith[W,Map[C,D]]]. It is a function that takes some element of the collection Map[A,B] (which would be a tuple (A,B)) and some accumulator whose type was declared for us as [DoWith[W,Map[C,D]]] and returns a new accumulator of the same type [DoWith[W,Map[C,D]]].

Our callback should call f on the current element of the map to find some DoWith[W,(C,D)]. It should then crack open that DoWith to find the inner tuple of type (C,D). It should extend the accumulator to include this tuple

Syntax Tips:

`dw_a:DoWith[W,A] map { (_:A) => _:B }` : DoWith[W,B]
map is used to crack open a DoWith that encapsulates some type A named here dw_a, perform work on that thing of type A to construct a thing of type B. It will then wrap the thing of type B in a new DoWith that uses the old memory state W.
HERE: A === (C,D) && B === Map(C,D)
OR : A === Map(C,D) && B == =Map(C,D)

dw_a :DoWith[W,A] flatMap { (_:A) => _:DoWith[W,B] } : DoWith[W,B]
flatMap is used to crack open a DoWith that encapsulates some type A named here dw_a, perform work on that thing of type A to construct a thing of type DoWith[W,B]. It will then wrap then return the thing of type B
HERE: A === (C,D) && B === DoWith[W,Map(C,D)]
OR : A === Map(C,D) && B === DoWith[W,Map(C,D)]

m:Map[A,B] + (a:A -> b:B) : Map[A,B]
extends the map ‘m’ with the key ‘a’ mapped to the value ‘b’.

m:Map[A,B] + ((a , b):(A,B)) : Map[A,B]
also extends the map ‘m’ with the key ‘a’ mapped to the value ‘b’. But uses a tuple rather than this ‘->’ thing… perhaps less clean to read in general but it might be useful considering the data types available to us in this function.

optional stub located at L5D3T1Q1. Please try this yourself without the stub before consulting the stub.

Testing : here is a test case for mapWith over Map[A,B]

```
"mapWith(Map)" should "map the elements of a map in a DoWith" in {
  val m = Map(("x"->1),("y"->2))
  val r1 = m.map { case (s,i) => (s,i + 1) }

  def dowith1[W]: DoWith[W,Map[String,Int]] = mapWith(m) { case (s:String,i: Int) => doreturn((s,i
+ 1)) }

  // tests where state W doesn't make a difference
  // Using assertResult format… arguably better than a plain assert in that it's optimized
  assertResult((true,r1)) { dowith1(true) } // setting W to Boolean
  // assert( (true,r1) === dowith1(true))
  assertResult((42,r1)) { dowith1(42) } // setting W to Int
  // assert( (42,r1) === dowith1(42) )

  // example where state matters
  val m_length = m.foldRight(0){ (_,acc) => acc + 1}
  assertResult((2 * m_length + 1, r1)) {
    val dw: DoWith[Int,Map[String,Int]] = mapWith(m) { case(s:String,i: Int) =>
      domodify[Int]( mem => mem + 2) map { _ => (s,i + 1) }
    }
    dw(1)
  }
}
```

L5D3T2: MapFirst

In addition to maping on List and Map data types I might find it beneficial to have a mapFirst HOF for Lists that returns a DoWith. Enter:

```
def mapFirstWith[W,A](l: List[A])(f: A => Option[DoWith[W,A]]): DoWith[W,List[A]] = l match {
    case Nil => ??? /* Base Case */
    case h :: t => ??? /* Inductive Case */
}
```

Given a list `l` with elements a_0 through a_{n-1} construct a DoWith[W,List[A]] such that the first observed element of `l` for which f(a_i) == Some(a_i') has been updated to a_i' and the other items of `l` remain unchanged

/* Base Case */

If the list is empty… what should you return? I think we want a DoWith of the empty List since that would be the result of trying to mapFirst f on an empty list…

Syntax Tips :

List() and Nil will both construct an empty List. (in the scope of this case statement, the variable `l` is also an empty list)

doreturn(a:A) : DoWith[W,A]
a call to doreturn(a) with some a:A will create a new DoWith of the type DoWith[W,A] for any type A (even if the type A happens to be the type List[A])

/* Inductive Case */

I have something of type A, namely h… I want to call f on input h and observe the output. There are 2 things that could happen here. The output of f has type Option[DoWith[W,A]] so my output will either be of the form `Some(dw_w_a:DoWith[W,A])` or of the form `None`.

Some(dw_w_a) case

I have a DoWith[W,A] that should* express only the first instance that f was successfully applied to h. I should crack this DoWith open to extract its inner 'a' element. I should prepend that to the rest of my list. I should not recurs on mapFirstWith because I have now successfully "mapped first".

* the goal is that we only observe a Some() case on the first time we can apply f to an element of `l`…

None case

I should** not yet have found the first instance that f was successfully applied to h. I must keep searching for that element. Recurs mapFirstWith on the tail to find some dw_w_tail_prime. Crack open that DoWith to extract that tail_prime. Prepend the current element to the new tail.

** the goal is to stop looking at elements of the list after we successfully apply f once.

Semantic Tips :

val dw_w_b = dw_w_a map { a => b } is a construct that is used to crack open a DoWith, find the data it encapsulates **a** and do work on that data to create new data **b.** See previous tips for syntax tips.

Recall that mapFirstWith returns a DoWith[W,List[A]]

optional stub located at bottom of document. Please try this yourself without the stub before consulting the stub.

Testing : here is some additional testing information

```
"mapFirstDoWith" should "map the first element where f returns Some" in {
  // GOAL: given a list of Int find the first negative element and make it positive

  // definitely more tests then are needed... but you might find it useful to read over this.

  /* Set Up */
  // The list I want to test this work on.
  val l1 = List(1, 2, -3, 4, -5)

  // The list I should get if I perform the goal task on l1
  val l2 = List(1, 2, 3, 4, -5)

  // The list I should get if I perform the goal task on l2 or l3AndUp
  val l3AndUp = List(1, 2, 3, 4, 5)

  // The function f to apply to each element of l and attempt to find a negative element
  def my_f [W]( i : Int ) : Option[DoWith[W,Int]] = {
    if ( i < 0 ) {
      val ip = -i
      val dw_w_ip:DoWith[W,Int] = doreturn(ip)
      val o_dw_w_ip:Option[DoWith[W,Int]] = Some(dw_w_ip)
      o_dw_w_ip
    } else {
      val o_dw = None
      o_dw
```

```
    }
  }

  // a DoWith that would* find the first negative element of l1 and make it non negative... IFF it
knew an initial state
  def dowith1[W]: DoWith[W,List[Int]] = mapFirstWith(l1)(my_f)

  // a DoWith that would* find the first negative element of l1 and make it non negative... IFF it
knew an initial state
  def dowith2[W]: DoWith[W,List[Int]] = mapFirstWith(l2)(my_f)

  // a DoWith that would* find the first negative element of l1 and make it non negative... IFF it
knew an initial state
  def dowith3AndUp[W]: DoWith[W,List[Int]] = mapFirstWith(l3AndUp)(my_f)

  /* The Tests */
  // I'll test to discover what happens if I call the dowith with an initial state.
  // First lets make the state rather useless.
  // here are a few flavors
  // let the state allways be true
  // for dowith1
  assertResult((true,l2)) {
    dowith1(true)
  }
  assertResult(true) { dowith1(true)._1 }
  assertResult(l2) { dowith1(true)._2 }
  assert( (true,l2) === dowith1(true) )
  assert( true === dowith1(true)._1 )
  assert( l2 === dowith1(true)._2 )
  // what if the state is always 42?
  assertResult((42,l2)) { dowith1(42) }

  // what about my other doWiths?
  assertResult((true,l3AndUp)) { dowith2(true) }
  assertResult((42,l3AndUp)) { dowith2(42) }
  assertResult((true,l3AndUp)) { dowith3AndUp(true) }
  assertResult((42,l3AndUp)) { dowith3AndUp(42) }

  // what about if the state is altered???
  assertResult((-8, l2)) {
    val dw: DoWith[Int,List[Int]] = mapFirstWith(l1){ i: Int =>
      if (i<0) {
        val dw:DoWith[Int,Int] = domodify{ state:Int => state+i } map { _ => -i } // discuss this
further at a later date
        val o_dw = Some(dw)
        o_dw
      } else {
        None
      }
    }
    dw(-5)
  }
  assertResult((-10, l3AndUp)) {
    val dw: DoWith[Int,List[Int]] = mapFirstWith(l2){ i: Int =>
      if (i<0) {
        val dw:DoWith[Int,Int] = domodify{ state:Int => state+i } map { _ => -i } // discuss this
further at a later date
        val o_dw = Some(dw)
        o_dw
      } else {
        None
      }
    }
    dw(-5)
  }
  assertResult((-5, l3AndUp)) {
    val dw: DoWith[Int,List[Int]] = mapFirstWith(l3AndUp){ i: Int =>
      if (i<0) {
```

```
        val dw:DoWith[Int,Int] = domodify{ state:Int => state+i } map { _ => -i } // discuss this
further at a later date
        val o_dw = Some(dw)
        o_dw
      } else {
        None
      }
    }
    dw(-5)
  }
}
```

L5D3T3 binary step

Continuing on yesterday's work. My object language *javascripty* has both unary operations and binary operations. The unary operations and binary operations have a lot of similarities on how they are coded in the step function.

DO

Template for unary Do:

- val raw_v1p = <literal unary operation in Scala's syntax> <the value that scala can operate on>
- val v1p = <convert thing to an Expr>
- doreturn(v1p)

example : DoNeg

```
        def step(e: Expr): DoWith[Mem, Expr] = {
                e match {
                        // DoNeg
                        case Unary(Neg, N(n1)) => {
                                // the operation Neg is represented in scala as '-'
                                // the thing that '-' can operation on is n1:Double
                                val raw_v1p = -n1
                                // Doubles are converted to Expr objects useing N(_:Double)
                                val v1p = N(thing)
                                doreturn(v1p)
                        }
                }
        }
```

Template for binary Do (exceptions for And, Or, Seq):

- val raw_v = <first operand><literal binary operation in Scala's syntax> <second operand>
- val v = <convert thing to an Expr>
- doreturn(v)

Template for binary Do And, Or, Seq:

- doreturn(<the thing specified to return in the inference rules>)

SEARCH

Template for print, unary and binary Searchs:

- val dw_eip = step(<expression denoted in premise as <M,ei> → <M',ei'> >)
- val dw_ep = dw_eip map { (eip) => <the correct Expr> }
- return dw_ep

example:

```
	def step(e: Expr): DoWith[Mem, Expr] = {
		e match {
			// SearchPrint
			case Print(e1) => {
				val dw_e1p = step(e1)
				val dw_ep = dw_e1p map {
					(e1p) => Print(e1p)
				}
				dw_ep
	}	}	}
```

Testing: assertions worth making into tests

```
/* presumes that the expression passes the typechecker */

// DoAndTrue
assert( step(Binary(And,B(true),Binary(Or,B(true),B(false))))(memempty)._2 === Binary(Or,B(true),B(false)) )

// DoAndFalse
assert( step(Binary(And,B(false),Binary(Or,B(true),B(false))))(memempty)._2 === B(false) )

// DoOrTrue
assert( step(Binary(Or,B(true),Binary(Or,B(true),B(false))))(memempty)._2 === B(true) )

// DoOrFalse
assert( step(Binary(Or,B(false),Binary(Or,B(true),B(false))))(memempty)._2 === Binary(Or,B(true),B(false)) )

// DoSeq
assert( step(Binary(Seq,S("hi"),Binary(Or,B(true),B(false))))(memempty)._2 === Binary(Or,B(true),B(false)) )

// DoPlusString
assert( step(Binary(Plus,S("he"),S("llo")))(memempty)._2 === S("hello") )

// DoArith(Plus)
assert( step(Binary(Plus,N(1),N(2)))(memempty)._2 === N(3) )

// SearchBinary, DoArith(Plus)
assert( step(Binary(Plus,Binary(Plus,N(1),N(2)),N(3)))(memempty)._2 === Binary(Plus,N(3),N(3)) )

// SearchBinary, DoArith(Plus) AND_THEN DoArith(Plus)
assertResult(N(6)) {
	step(Binary(Plus,Binary(Plus,N(1),N(2)),N(3)))(memempty)._2 flatMap {
		ep => { // should be Binary(Plus,N(3),N(3))
			step(ep) // should be a dowith of N(6)
		}
	}(memempty)._2
}

/* so many more options */
```

L5D3T4 isBindex

- complete isBindex following the semantic provided on page 11 of the lab 5 handout

isBindex is a function that takes in a mode and an expression and returns a Boolean stating whether or not the expression is a bindable expression subject to the mode.

This operational semantic can be seen used in premise throughout the inference rules for the typeof function, namely in TypeDecl and TypeCall.

There are only 2 inference rules for isBindex found on page 11 of the lab handout namely, ValBind and RefBind. They make a distinction of whether something is bound by value or by reference (memory expressions / expressions about pointers and dereferencing).

They operate on m, e, and le all of which are defined in the labs reference grammar on page 7 of the lab handout.

We were introduced to m in Lab 4. It denotes a mode that is attached to some variable. In Lab 5 we extend the language of `m` to not only include const and name but also var and ref. const still represents constant non-mutable bindings where expressions must be evaluated to values prior to being bound to a variable. name still represents lazy evaluation and the ability to bind a non-value expression to a variable. var represents the ability to create a mutable-variable so that I can bind a valued expression to the variable (as I could with const) but I can also, later update the value bound to the variable. ref represents the ability to reference memory i.e. to create pointer types.

e is the same as the previous 4 labs, it denotes all possible expression in the object language *javascripty*. Although the language of `e` is a bit larger than before

le is something totally new to the labs. It is the non-terminal for location expression. It is any expression that should (during execution) become a location value. It should eventually reference to memory. The type checker does not actually have any knowledge of memory so that is why it will look for location expression relative to the mode ref to determine if an expression is bindable rather than looking for location values (we'll discuss that further when we look at implementing getBindex – the rules with the hooked arrows on page 14).

The isLExpr function might be useful in solving isBindex:

```
def isLExpr(e: Expr): Boolean = e match {
        case Var(_) | GetField(_, _) => true
        case _ => false
}
```

Testing… You can write these tests however you'd like but I think it is important to write out a few of these. Here are some assertions that should pass on isBindex

Mode == const | name | val

```
// Pretty much any expression will do…
val e1 : Expr = Binay(Plus,N(1),N(2))
assert( isBinex(MConst,e1) )
assert( isBinex(MName,e1) )
assert( isBinex(MVar,e1) )

// won't pass the type checker…
// that doesn't effect how isBindex is coded
val e2:Expr = Binary(Plus,N(1),S("hi"))
assert( isBinex(MConst,e2) )
assert( isBinex(MName,e2) )
assert( isBinex(MVar,e2) )

// again… this should work for ALL possible expressions. It should be very simple to code and thus there
// is no need to write a lot of test cases. 1 or 2 would be fine
```

Mode == ref

```
// 1 is not a location expression so it should return false
assert( ! isBindex(MRef,N(1)) )

// y is a location expression
assert( isBindex(MRef,Var(y)) )

// o.x is a location expression where o hopefully represents an object that has an x field. (that is
// determined by a different part of the typeof function and not handled by isBindex
assert( isBindex(MRef,GetField( Var(o) ,x )) )
```

L5D3T5 typeOf

Complete typeof except for TypeAssignVar, TypeAssignField

type of in lab 5 is super similar to type of from lab 4 with a few modifications.

- TypeDecl and TypeCall rely on the correct implementation of a function called isBindex denoted by the operational semantic m ⊢ e bindex see notes on isBindex for more details

- We have more types
  - The null type
    - Grammar (page 7) : t ::= … | Null | …
    - AST of Typ (ast file) : case object TNull extends Typ
  - The type variables type (T) as well as the interface type(Interface T t)
    - We will look at this at a later date… it has something to do with type casting an expression and have for any type T…

- Our expressions have changed a bit which effects the type checker (see grammar on page 7 of the lab handout and look over the AST file while coding each Type rule if you get lost)

I leave it to you to code up the type checker.

If I have portions of the type checker working and portions of the step function working I can begin writing integration tests that check to see if my flow of control works as expected. The flow of control is, for an expression **e**, call typeof(e) and iff that returns a valid type then call iterateStep(e) and return the value found.

Here is a test for evaluation of *javascripty* expression " 1 + 2 + 3 "
Relies on successful implementation of TypeArith(Plus), TypeNumber, DoArith(Plus), SearchBinary1
I wrote an evaluate function that might be useful

```
def evaluate(e:Expr,t:Typ,v:Expr):Unit = {
    assertResult(t){typeof(empty,e)}
    assertResult(v){iterateStep(e)}
}

"thingsAboutPlus" should "do things about plus" in {
    val e = Binary(Plus,Binary(Plus,N(1),N(2)),N(3))  // parse('1+2+3')
    evaluate(e,TNumber,N(6))
}
```

Solutions

L5D3T1Q1 additional stub

Not really a solution to anything, but here is a stub that provides additional hints on how to write **mapWith over Map[A,B]**

```
def mapWith[W,A,B,C,D](m: Map[A,B])(f: ((A,B)) => DoWith[W,(C,D)]): DoWith[W,Map[C,D]] = {
    val mapcd_baseCase = ???
    val dw_w_mapcd_baseCase = ???
    m.foldRight[DoWith[W,Map[C,D]]]( dw_w_mapcd_baseCase ) {
        case ((a,b),acc_dw_w_mapcd) => { // could also use (ab_i,acc_dw_w_mapcd) and not say 'case'
                val dw_w_cd = ???
                val acc_dw_w_mapcd_p = ??? // flatMap and map on DoWith structures available
                acc_dw_w_mapcd_p
}   }   }
```

L5D3T2Q1 additional stub

Not really a solution to anything, but here is a stub that provides additional hints on how to write **mapFirstWith over List[A]**

```
def mapFirstWith[W,A](l: List[A])(f: A => Option[DoWith[W,A]]): DoWith[W,List[A]] = l match {
    case Nil => {
        val emptyList = ???
        val dw_w_emptyList:DoWith[W,List[A]] = ???
        dw_w_emptyList
    }
    case h :: t => {
        val o_dw_w_hp = ???
```

```
            o_dw_w_hp match {
                case None => {
                    val dw_w_tp = ???
                    val dw_w_lp = ??? // use map
                    dw_w_lp
                }
                case Some(dw_w_hp) => {
                    val dw_w_lp = ??? // use map
                    dw_w_lp
    }   }   }       }
```

L5D4

- mapFirstWith
- typeof continued
- getBinding
- rename

L5D4T1 mapFirstWith

Here is another viewpoint for mapFirstWith. Here are inference rules over mapFirstWith and all the support for these rules.

- Grammars
  - Lists[A]
    - Lists[A] ::= Nil | A :: Lists[A]
    - Where A can be any language
  - < W , R > a state monad over the language of W and R for any language W and any language R
  - Option[B]
    - Option[B] ::= None | Some(B)
    - Where B can be any language
- Judgment forms + ~judgment forms~
  - = sort of output = input
- Operational semantics
  - < M , List[A] > = mapFirstWith(List[A])(f)
  - Option[< M , A >] = f (A)
- Inference Rules :

Rule1

< M , Nil > = mapFirstWith(Nil)(f)

Rule 2 Some(< M , hp >) = f (h)

< M , hp::t > = mapFirstWith(h::t)(f)

Rule 3 None = f (h) < M , tp > = mapFirstWith(t)(f)

< M , h::tp > = mapFirstWith(h::t)(f)

Syntax notes:

- < M:W , a:A > could be constructed using doreturn(a)
- given < M , a > and b. To construct < M , b > we would say < M , a > map { (a) => b }
    - example with W being the language of Booleans and A the language of Strings :
      dw_m_a = < true , “hello” >
      b = “there”
      < true , “hello there” > = dw_m_a map { a => a + b }

L5D4T2 typeof continued

TypeAssignVar, TypeAssignField + testing

TypeAssignVar

Inference Rule

$$\frac{x \mapsto m\ \tau \in \Gamma \qquad \Gamma \vdash e : \tau \qquad m \in \{\ \texttt{var}\ ,\ \texttt{ref}\ \}}{\Gamma \vdash x = e : \tau}$$

Explanation

Conclusion : In a type environment Γ, to successfully interpret the type of an expression of the form x = e to some type τ the following must hold true :

- Premise 1: x must exist in Γ mapped to both a mode and a type
    - NOTE: our type environment TEnv is defined differently in this lab than it was in the previous lab
    - type TEnv = Map[String, MTyp]
- Premise 2 : the sub-expression e must have the type τ in the environment Γ note that this τ must be the same type τ that x is mapped to in Γ
- Premise 3 : the mode m that x is mapped to in Γ must be either var or ref

Error cases

- If the type t of sub-expression e is not the type of x in our environment then throw an error on sub-expresssion e for type t
- Strange error … If the mode is not var or ref then we should also through an error on the type of sub-expression e for its type t

Note

- The AST of x = e is Assign(Var(x),e)

See stub for this rule at L5D4T1Q1

Testing

```
"typeAssignVar" should "succeed" in {
  assert(typeof(empty + ("y" -> MTyp(MVar,TNumber)),
      Assign(Var("y"),N(2)))===TNumber)
}
it should "fail" in { // not the best way to test this... but a step in
the right direction
```

```
  assertResult("fail"){
    try {
      typeof(empty + ("y" -> MTyp(MVar,TString)),
          Assign(Var("y"),N(2)))
    }
    catch {
      case _:Throwable => "fail"
    }
  }
  assertResult("fail"){
    try {
      typeof( empty + ("y" -> MTyp(MConst,TNumber)),
                Assign(Var("y"),N(2)
              )
    }
    catch { case _:Throwable => "fail"
} } }
```

TypeAssignField

Inference Rule

$$\frac{\Gamma \vdash e1 : \{ \ldots; f : \tau ; \ldots \} \qquad \Gamma \vdash e2 : \tau}{\Gamma \vdash e1.f = e2 : \tau}$$

Explanation

Conclusion : in a type environment Γ expression of the form e1.f = e2 have type τ iff :

- Premise 1 : subexpression e1 has the type { …; f : τ ; … } i.e. TObj(tfields) s.t. tfields contains field f mapped to type τ
- Premise 2 : subexpression e2 has type τ, the same type as f in the type object of e1

Error cases

- If e1 is not an object then throw an error on the type found
- If f is not in the fields of e1 then through an error on the type of e2 and its type
- If the type of e1.f is not the type of e2 then throw an error on e2

Note

- The AST of e1.f is Assign(GetField(e1,f),e2)

Optional stub at L5D4T1Q2

Testing

```
"typeAssignGetField" should "succeed" in { // dependent on Var
implementation
  assert(typeof(empty + ("y" -> MTyp(MConst,TObj(Map("y"->TNumber)))),
      Assign(GetField(Var("y"),"y"),N(2)))===TNumber)
}
it should "fail" in { // not the best way to test this... but a step in
the right direction
  assertResult("fail"){ // if e1 is not an object
    try {
      typeof( empty , Assign(GetField(N(2),"y"),N(2)))
    }
    catch { case _:Throwable => "fail" }
```

```
  }
  assertResult("fail"){ // if the field is not in TObj
    try {
      typeof(empty + ("y" -> MTyp(MConst,TObj(Map()))),
          Assign(GetField(Var("y"),"y"),N(2)))
    }
    catch { case _:Throwable => "fail" }
  }
  assertResult("fail"){ // if e2 is not the correctType
    try {
      typeof(empty + ("y" -> MTyp(MConst,TObj(Map("y"->TNumber)))),
          Assign(GetField(Var("y"),"y"),S("hi")))
    }
    catch { case _:Throwable => "fail" }
  }
}
```

L5D4T3 getBinding

isBindex is to typeof as getBinding is to step… in that they serve a similar purpose.

getBinding is defined on page 14 of the lab 5 handout using the operational semantic that has a turnstyle and a hooked arrow.

Note, that page 14 also defines isRedex using a turnstyle and the term ‘redex’.

Without knowing to much about the purpose of getBinding I am certain that you can code ConstBind, NameBind, and RefBind. Please do this today. I will provide you with a deeper explanation of the use, context and deeper meaning of this function at a later date.

Note that the provided stub for getBinding is dependent on isRedex. You should complete isRedex before going through getBinding. I leave it to you to write test cases for isRedex

I believe that you are also capable of completing VarBind but it is significantly more challenging. Please attempt to complete it. Note that memalloc is a useful helper function for implementing the VarBind case.

Testing

```
"ConstBind" should "do stuff with values" in {
    assert(getBinding(MConst,N(1))(memempty)._2 === N(1))
    // better VVVV
    assertResult( N(1) ){
      getBinding(MConst,N(1))(memempty)._2
    }
    // won't work because memory is private....
//    assertResult( (Map(),N(1)) ){
//      getBinding(MConst,N(1))(memempty)
//    }
  }
  it should "fail to do stuff with non-values" in {
    assertResult("fail"){
```

```
      try{getBinding(MConst,Unary(Neg,N(1)))(memempty)}
      catch { case _:Throwable => "fail" }
    }
  }

  // Write your own for NameBind
  "RefBind" should "do stuff with location-values" in {
    assertResult( Unary(Deref, A(0)) ){
      getBinding(MRef,Unary(Deref, A(0)))(memempty)._2
    }
  }
  it should "fail to do stuff with non-location-values" in {
    assertResult("fail"){
      try{getBinding(MRef,N(1))(memempty)}
      catch { case _:Throwable => "fail" }
    }
  }
  "VarBind" should "do special stuff with values" in {
    assertResult( Unary(Deref,A(1)) ) {
      getBinding(MVar,N(1))(memempty)._2
    }
  }
  it should "fail to do special stuff with non-values" in {
    assertResult("fail"){
      try{getBinding(MVar,Unary(Neg,N(1)))(memempty)._2}
      catch { case _:Throwable => "fail" }
    }
  }
```

L5D4T3 rename

A few quick notes about rename.

The goal of rename is fairly well defined in the lab 5 handout on pages 5 and 6.

Rename is not a recursive function all that rename does is call ren. ren is a recursive function, for this reason the print case provided to us in rename recurs on ren and not rename.

Similar to substitution function in previous labs. Ren will mostly recur all over the place with limited exceptions dealing with when we actually find a variable that might need renaming according to the fresh function or when we find a value.

Here is the print case provided to us in ren

```
case Print(e1) => ren(env,e1) map { e1p => Print(e1p) }
```

Here is the another way to write it… define a helper function named r at the top of the scope of ren prior to pattern matching.

```
/* defined at the top of ren… */
```

```
def r(e:Expr):DoWith[W,Expr] = ren(env,e)

case Print(e1) => r(e1) map { Print( _ ) }
```

Here is yet another way to write it using several temporary variables.

```
case Print(e1) => {
        // using suffix R to denote something is potentially renamed in the expression
        val dw_w_e1R:DoWith[W,Expr] = ren(env,e1)
        val dw_w_printe1R:DoWith[W,Expr] = dw_w_e1R map { e1R => Print( e1R ) }
        dw_w_printe1R
}
```

Note that ren returns a DoWith. Accordingly, map and flatMap will be our friends as we tackle this big step recursive function….

```
ren(a) flatMap {
        aRenamed => ren(b) flatMap {
                bRenamed => ….
                        ren(z) map {
                                zRenamed => /* do things with aRenamed through zRenamed */
}       }       …       }
```

Testing

```
{
  def my_fresh(x:String):DoWith[Int,String] = {
    // doget, doput, map, flatMap are detailed on pag 4 of Lab 5
    // get the state of memory...
    // note that the state type W is an Int, here*
    doget[Int] flatMap {
      // Use the current state of memory to update a new state
      n => doput ( n + 1 ) map {
        // In scope where I know the original state... update the
expression
        _ => x + n.toString()
      }
    }
  }
  "renameVar" should "get variables for the env if possible" in {
    val e1 = Var("y")
    val env1 = empty + ("y" -> "y0")
    val e1R= Var("y0")
    assertResult((1,e1R)) {
      rename(env1, e1){ my_fresh }(1)
    }
  }
  it should "return the var if its not in the environment" in {
    val e1 = Var("y")
    val env1 = empty
    val e1R = Var("y")
```

```
    assertResult((0,e1R)) {
      rename(env1, e1){ my_fresh }(0)
    }
  }
  "renameVar,renameDecl" should "do stuff" in {
    val e = Decl(MConst,"y",Var("y"),Var("y"))
    val env = empty
    val eR = Decl(MConst,"y0",Var("y"),Var("y0"))
    assertResult((1,eR)) {
      rename(env, e){ my_fresh }(0)
    }
  }
}
```

Solutions

L5D4T2Q1

optional Stub for TypeAssignVar
I recommend that you struggle to solve the problem yourself before looking at this stub…

```
def typeof(env: TEnv, e: Expr): Typ = {
        def err[T](tgot: Typ, e1: Expr): T = throw StaticTypeError(tgot, e1, e)
        e match {
                case Assign(Var(x),e2) => { // conclusion pattern
                        val t2 = ??? // find the type of e2
                        val MTyp(m1,t1) = ??? // attempt to get the mt of x in env and throw an error if it's not
                                                // there, I recommend using getOrElse
                        m1 match { // check the mode and do work accordingly... ( can definitely make this in
                                        // fewer cases )
                                case MVar => ???
                                case MRef => ???
                                case MConst => ???
                                case MName => ???
                                case _ => ??? // should be unreachable code
                        }
                }
                case Assign(_, _) => err(TUndefined, e) // provided to us... if the assign is not valid then throw
                                                        // an error
}       }
```

L5D4T2Q2

Optional stub for TypAssignGetField
Again I recommend that you struggle to solve the problem yourself before looking at this stub…

```
def typeof(env: TEnv, e: Expr): Typ = {
        def err[T](tgot: Typ, e1: Expr): T = throw StaticTypeError(tgot, e1, e)
        e match {
                case Assign(GetField(e1,f),e2) => {
                        val t1 = ??? // get the type of e1
```

```
                t1 match {
                    case TObj(tfields) => { // I expect e1 to be an object
                        val t2 = ??? // get the type of e2
                        val t = ??? // get t and throw an error on e2 if not possible... I recommend
                                    // a getOrElse statement
                        // conditionally return t subject to whether e2 has an acceptable type
                        ???
                    }
                    case tgot => ??? // if t1 isn't an object... do stuff
                }
            }
            case Assign(_, _) => err(TUndefined, e) // provided to us... if the assign is not valid then throw
                                                    // an error
}       }
```

L5D5

- Rename continued
- Rules for Rename
- Get binding continued
- Substitution
- Page 15

L5D5T1 Rename cont.

Rename is used in our lab to implement a concept called “free-variable capture avoidance”. Integrating this with our substitution function we can implement free-variable avoidance substitution.

```
def rename[W](env: Map[String,String], e: Expr)(fresh: String => DoWith[W,String]): DoWith[W,Expr] =  {
  def ren(env: Map[String,String], e: Expr): DoWith[W,Expr] =   {
      ???
  }
  ren(env,e)
}
```

rename(env,e)(fresh) = ren(env,e) = …

- env:Map[String,String]
  - often initializes to an empty map, over time this should store variable names mapped to new variable names. It will map the variable names in e to names dictated by fresh (the renaming policy)
- e : Expr
  - this is the expression that I would like to search over for instances of variables and potentially update the variables according to the update policy provided to me.
  - Represents an abstract syntax tree
- fresh: String => DoWith[W,String]
  - this is the renaming policy
  - it is written over a DoWith so it has an abstraction of memory built into it
  - this function determines what rename will actually do
    - will it add ‘$’ signs to the end of things to create unique variable names?

        - Will it increment a counter suffix to create unique variable names?
        - Will it do something useful or just create problems?
    - to understand it you probably want to look over an example
- ren(env,e)
    - the function that is actually recursive
    - it will search over it's parameter e, updating it's parameter env subject to rename's input fresh.

Lets look at the test case provided to us earlier and dissect it to help us drive our development.

```
{
 val e = Binary(Plus,Var("y"),Decl(MConst,"z",N(12),Binary(Plus,Var("z"),Var("y"))))

 "rename" should "do stuff" in {
  def fresh(x: String): String = if (x=="z") fresh(x + "$") else x
  val eRenExpected = Binary(Plus,Var("y"),Decl(MConst,"z$",N(12.0),Binary(Plus,Var("z$"),Var("y"))))
  assertResult(eRenExpected) { rename(e)(){ x => doreturn(fresh(x)) } }
 }
 it should "do otherStuff" in {
  def fresh(x:String):String = if (x=="z") fresh (x + "#") else x
  val eRenExpected = Binary(Plus,Var("y"),Decl(MConst,"z#",N(12.0),Binary(Plus,Var("z#"),Var("y"))))
  assertResult(eRenExpected) { rename(e)(){ x => doreturn(fresh(x)) } }
 }
}
```

The highlighted part expressed the calls made to rename. Notice anything strange about these calls?

This is not a call to our rename. This is a call to a different function rename, a driver for our version of rename. Here is the definition of the rename that is being called:

```
def rename[W](e: Expr)(z: W)(fresh: String => DoWith[W,String]): Expr = {
        val (_, r) = rename(empty, e)(fresh)(z)
        r
}
```

rename[W](e: Expr)(z: W)(fresh: String => DoWith[W,String]):

- e:Expr
    - this will be the expression passed to our implementation of rename
- z: W
    - this is the initial state for our evaluation
- fresh: String => DoWith[W,String]
    - this is the policy passed to our rename function
- note that the env for our rename function will be initialized to empty

```
val e = Binary(Plus,Var("y"),Decl(MConst,"z",N(12),Binary(Plus,Var("z"),Var("y"))))
"rename" should "do stuff" in {
  def fresh(x: String): String = if (x=="z") fresh(x + "$") else x
  val eRenExpected = Binary(Plus,Var("y"),Decl(MConst,"z$",N(12.0),Binary(Plus,Var("z$"),Var("y"))))
  assertResult(eRenExpected) { rename(e)(){ x => doreturn(fresh(x)) } }
}
```

With respect to rename(env,e)(fresh)(initialState)

- env is empty
- e is the abstract representation of the javascripty literal *y + (const z = 12; z + y)*
- fresh is x => *doreturn*(**def** fresh (x: String): String = **if** (x=="z") fresh(x + "$") **else** x; fresh(x))
  which states… that on a call to fresh, if the input x is a "z" literal, then recurse fresh on "z$", else return the input x.
- initial state is literally nothing.

Because of all that, calling rename on *y + (const z = 12; z + y)* with the policy to append a $ to any instance of z… I will find *y + (const z$ = 12; z$ + y)*

The other test does the same thing but the policy requires that I append "#" to the end of any "z" variable instance.

Where this becomes important is during substitution. The goal there is to append "$" to any instance of a free variable of esub that exists in e using rename on e with a fresh policy that appends $ to a specific set of literals.

```
// From these tests I should be able to derive
def rename[W](env: Map[String,String], e: Expr)(fresh: String => DoWith[W,String]): DoWith[W,Expr] = {
        def ren(env: Map[String,String], e: Expr): DoWith[W,Expr] = e match {
                case Binary(Plus,e1,e2) => ??? // ren both sub-expressions
                case Decl(m,x,e1,e2) => ??? // use fresh to conditionally update x. Extend the environment,
                                        // then ren both subexpressions with particular environments.
                case Var(x) => ??? // Lookup the string in the environment
                case _ => ??? // need more tests… Or I can just guess randomly…
        }
        ren(env, e)
}
```

L5D5T2 Rules for Rename

We need to understand free-variable capture avoidance! But I don't want us to spend too much time an energy coding rename so I've written inference rules over rename for all of us. These should work, but there may be bugs in them.

Inference rules for the rename function.
My goal is to find rename(env,e)(fresh)(state). Why? rename(env,e)(fresh) is a DoWith and essentially a function value. But once I curry this with an initial state I will find a tuple of state and expression.

~Judgment Form~: output = input

TestRename Operational Semantic :	(state' , eR) = rename(env , e)(fresh)(state)
Rename Operational Semantic :	< state' , eR > = rename(env , e)(fresh)
Ren Operational Semantic:	< state' , eR > = ren(env , e)
Fresh Operational Semantic:	< state' , sF > = fresh(s)
TestFresh Operational Semantic:	< s2' , s1F > = fresh(s1)(s2)

NOTES:

- < A , B > denotes a DoWith[A,B]
- (A , B) denotes a tuple of A and B
- This uses the reference grammar from Lab 5
- env : Map[String,String]
- e : Expr
- fresh : (String) => DoWith[W,String] for any type W
- state: W for any type W

DISCLAIMER:
- This has not been proven correct…

TestRename < state' , eR > = rename(env , e)(fresh) (state' , eR) = < stateR , eR >(state)

(state' , eR) = rename(env , e)(fresh)(state)

Rename < state' , eR > = ren(env , e)

< state' , eR > = rename(env , e)(fresh)

RenUop < state' , e1R > = ren(env , e1)

< state' , uop e1R > = ren(env , uop e1)

RenBop < state' , e1R > = ren(env , e1) < state'' , e2R > = ren(env , e2)

< state'' , e1R bop e2R > = ren(env , e1 bop e2)

… but that doesn't quite show what I want to show… I will now modify the operational semantics to act as though I know my initial state. Using doget I can know the current state of my DoWith so it's not a huge stretch to act as though the 'e' parameter to rename is a DoWith[W,Expr] rather than just an Expr. The lab handout makes this same jump in logic

~Judgment Form~: output = input

TestRename Operational Semantic :	(W2' , eR)	=	rename(env , < W1 , e>)(fresh)(W2)
Rename Operational Semantic :	< W' , eR >	=	rename(env , < W , e>)(fresh)
Ren Operational Semantic:	< W' , eR >	=	ren(env , < W , e>)
Fresh Operational Semantic:	< W' , sF >	=	fresh(< W , s >)
TestFresh Operational Semantic:	(W' , s1F)	=	fresh(s1)(W)

NOTES:
- < A , B > denotes a DoWith[A,B]
- (A , B) denotes a tuple of A and B
- This uses the reference grammar from Lab 5
- s is a string

DISCLAIMER:
- Again, this has not been proven correct

I encourage you to code these in the order they are provided. Rename and RenPrint and RenInterface have been implemented for you.

Rename
$$\frac{\langle W' , eR \rangle = ren(env , \langle W , e \rangle)}{\langle W' , eR \rangle = rename(env , \langle W , e \rangle)(fresh)}$$

RenSimpleVals
$$\frac{v \in \{ n , str , b , undefined, null, a \}}{\langle W' , v \rangle = ren(env , \langle W , v \rangle)}$$

RenPrint
$$\frac{\langle W' , e1R \rangle = ren(env , \langle W , e1 \rangle)}{\langle W' , uop\ e1R \rangle = ren(env , \langle W , uop\ e1 \rangle)}$$

RenUop
$$\frac{\langle W' , e1R \rangle = ren(env , \langle W , e1 \rangle)}{\langle W' , uop\ e1R \rangle = ren(env , \langle W , uop\ e1 \rangle)}$$

RenBop
$$\frac{\langle W' , e1R \rangle = ren(env , \langle W , e1 \rangle) \qquad \langle W'' , e2R \rangle = ren(env , \langle W' , e2 \rangle)}{\langle W'' , e1R\ bop\ e2R \rangle = ren(env , \langle W , e1\ bop\ e2 \rangle)}$$

RenIf
$$\frac{\langle W' , e1R \rangle = ren(env , \langle W , e1 \rangle) \quad \langle W'' , e2R \rangle = ren(env , \langle W' , e2 \rangle) \quad \langle W''' , e3R \rangle = ren(env , \langle W'' , e3 \rangle)}{\langle W''' , e1R\ ?\ e2R : e3R \rangle = ren(env , \langle W , e1\ ?\ e2 : e3 \rangle)}$$

NOTE: HERE, I stacked the premises In part because I ran out of space also because, this expresses a bit of the nested structure that I will need when coding.

RenAssign
$$\frac{\langle W' , e1R \rangle = ren(env , \langle W , e1 \rangle) \qquad \langle W'' , e2R \rangle = ren(env , \langle W' , e2 \rangle)}{\langle W' , e1R.f \rangle = ren(env , \langle W , e1 = e2 \rangle)}$$

RenInterface
$$\frac{}{\text{“Gremlin err”} = ren(env , \langle W , interface\ T \{ f1:t1, \ldots , fn:tn \}; e1 \rangle)}$$

RenVarF
$$\frac{x \in env \quad xF = env(x)}{\langle W'' , xF \rangle = ren(env , \langle W , x \rangle)}$$

RenVar
$$\frac{x \notin env}{\langle W'' , x \rangle = ren(env , \langle W , x \rangle)}$$

RenDecl
$$\frac{\langle W' , xF \rangle = fresh(\langle W , x \rangle) \quad \langle W'' , e1R \rangle = ren(env , \langle W' , e1 \rangle) \quad \langle W''' , e2R \rangle = ren(env + (x \to xF) , \langle W'' , e2 \rangle)}{\langle W''' , m\ xF = e1R\ ;\ e2R \rangle = ren(env , \langle W , m\ x = e1\ ;\ e2 \rangle)}$$

Note: before moving forward, take a moment to make certain that you understand what this does syntactically. Focus on RenDecl and RenVar. I believe this is enough information to pass and understand the tests provided to you. For expressions m x = e1 ; e2 and xF being the freshified instance of variable x according to the fresh renaming policy provided… Observe in that sub-expression e2 is evaluated in an environment where x is mapped to xF.

I recommend coding RenVarF and RenVar using a getOrElse statement applied to our environment 'env'.

Now for objects… note that object fields are unique and accordingly do not need to have a rename policy applied to them

RenGetField $< W' , e1R > = ren(env , < W , e1 >)$

$< W' , e1R.f > = ren(env , < W , e1.f >)$

This is actually an easy one

RenObj $< W' , enR > = ren(env , < W , en >)$
$\ldots < W^{j} , eiR > = ren(env , < W^{j-1} , ei >)$ for i over (n , 1) and j from (1 , n-1)
$< W^{n} , e1R > = ren(env , < W^{n-1} , e1 >)$

$< W' , \{ f1 : e1R , \ldots , fn : enR \} > = ren(env , < W , \{ f1 : e1 , \ldots , fn : en \} >)$

RECALL that last week we wrote HOF to accumulate DoWiths over a Map

And finally the for challenging cases, Functions and Calls…

RenCall $< W' , e0R > = ren(env , < W , e0 >)$
$< W'' , enR > = ren(env , < W' , en >)$
$\ldots < W^{j} , eiR > = ren(env , < W^{j-1} , ei >)$ for i over (n , 1) and j from (1 , n)
$< W^{n+1} , e1R > = ren(env , < W^{n} , e1 >)$

$< W^{n+1} , e0R(e1R,\ldots enR) > = ren(env , < W , e0(e1,\ldots,en) >)$

for RenCall please note, that last week we wrote HOF to accumulate DoWiths over a List over the foldRight method of lists.

RenFunc $< W' , xnF > = fresh(< W , xn >)$
$\ldots < W^{j} , xiF > = fresh(< W^{j-1} , xi >)$ for i over (n , 1) and j from (1 , n-1)
$< W^{n} , x1F > = fresh(< W^{n-1} , x1 >)$
$< W^{n+1} , e1R > = ren(env + (xn \rightarrow xnR) + \ldots + (x1 \rightarrow x1R) , < W^{n} , e1 >)$

$< W^{n+1} , (x1R:m1t1,\ldots,xnR:mntn):tan \Rightarrow e1R > =$
$ren(env , < W , (x1:m1t1,\ldots,xn:mntn):tan \Rightarrow e1 >)$

RenFuncRec $< W' , x0F > = fresh(< W , x0 >)$
$< W'' , xnF > = fresh(< W' , xn >)$
$\ldots < W^{j} , xiF > = fresh(< W^{j-1} , xi >)$ for i over (n , 1) and j from (1 , n)
$< W^{n+1} , x1F > = fresh(< W^{n} , x1 >)$

$< W^{n+2} , e1R > = ren(env + (x0 -> x0F) + (xn -> xnF) + \ldots + (x1 -> x1F) , < W^{n+1} , e1 >)$

$< W^{n+2} , x0F\ (x1F:m1t1,\ldots,xnF:mntn):tan => e1R > =$

$ren(env , < W , x0\ (x1:m1t1,\ldots,xn:mntn):tan => e1 >)$

NOTE: think about all that needs to be achieved here and code this intelligently using a HOF. It is recommended to use an HOF to accumulate some variant of the date type:

DoWith[W , (List((String,MTyp)) , Map[String,String])] for any state type W

And if you are cool you could add in an extra rule… and update the above rules to be a bit faster

~Judgment Form~: output = input

TestRename Operational Semantic :	(W2' , eR)	=	rename(env , < W1 , e>)(fresh)(W2)
Rename Operational Semantic :	< W' , eR >	=	rename(env , < W , e>)(fresh)
Ren Operational Semantic:	< W' , eR >	=	ren(env , < W , e>)
Fresh Operational Semantic:	< W' , sF >	=	fresh(< W , s >)
TestFresh Operational Semantic:	(W' , s1F)	=	fresh(s1)(W)

R Operational Semantic : < W' , eR > = r (< W , e >)

R $< W' , e' > = ren(env , < W , e >)$

$< W' , e' > = r (< W , e >)$

This would require the function r to be declared inside the scope of ren so that it is aware of the variable 'env'.

L5D5T3 getBinding continued

getBinding uses the operational semantic of read as "mode turnstyle DoWith_of_M_e hookedarrow DoWith_of_Mprime_eprime"

the first line of the getBinding function is a require statement that requires that the expression passed to getBinding is not reducible subject to isRedex. For this reason it is recommended that you correctly implement and test isRedex prior to implementing getBinding. It is also recommended that any call to getBinding is prefaced with a call to isRedex

for some e:Expr and m:Mode

BAD : getBinding(m , e)

GOOD : if (! isRedex(m , e)) { getBinding(m , e) } else { ??? }

Also good to use a guard if (! isRedex(m , e)) on a case statement that uses getBinding in its body.

getBinding should only be called from within step. It is used prior to substitution to discern what the argument esub will be for the substitution call.

L5D5T4 Substitution

Substitution in Lab 5 is nearly the same as lab 4 / perhaps exactly the same?

Substitute should not be a recursive function. Substitute should call subst which is a recursive function. The initial call to subst should call subst on an expression eRen where eRen is determined by calling myRename on e.

myRename has to implement a renaming policy for the rename function namely fresh and call rename with the initial expression and the rename policy. To understand how myRename should be implemented I encourage you to look through the .jsy .ans files provided for you for integration tests. Or look over L5Bonus_Testing.scala in the Rename and Substitute flatSpecs.

You can implement subst before implementing rename but you will need to be careful about your test cases if you are using test driven development (TDD). Prior to implementing rename, any test cases for substitute(e,esub,x) should be such that esub has no free variables. You will also need remember to have substitute make an initial call to subst and later change the call to subst to include the myRename function.

I'd encourage you to write tests over substitute and tests over rename separately and not spend much time looking at whether they work with relation to each other – those integration tests are not overly useful.

Here are a few test cases that looks at whether substitute works mostly* independent of whether rename works.

```
"substDecl" should "perform substitution" in {
  // y [ - ( 5 ) / y ]
  // from step possibly
  //    name y = - ( 5 ) ; y
  val e = Var("y")
  val esub = Unary(Neg,N(5))
  val x = "y"
  val resultExpected = Unary(Neg,N(5))
  assertResult( resultExpected ){
    substitute(e,esub,x)
} }
it should "not perform substitution" in {
  // z [ - ( 5 ) / y ]
  // from step possibly
  //    name y = - ( 5 ) ; <stuff> z
  //    where <stuff> declares what z is
  //    but subst doesn't care about that...
  val e = Var("z")
  val esub = Unary(Neg,N(5))
  val x = "y"
  val resultExpected = Var("z")
  assertResult( resultExpected ){
    substitute(e,esub,x)
} }
```

```
"substBinary" should "attempt substitution on subexpressions" in {
  // y + z [ - ( 5 ) / y ]
  // from step possibly
  //    name y = - ( 5 ) ; <stuff> y + z
  //    where <stuff> declares what z is
  //    but subst doesn't care about that...
  val e1 = Var("y")
  val e2  = Var("z")
  val e = Binary(Plus,e1,e2)
  val esub = Unary(Neg,N(5))
  val x = "y"
  val resultExpected = Binary(Plus,Unary(Neg,N(5)),Var("z"))
  assertResult( resultExpected ){
    substitute(e,esub,x)
} }
```

L5D5T5 Page 15

Page 15 of the lab 5 handout is representative of quite a bit of work that we need to code. I encourage you to make a plan for how to complete this. I will not provide notes on each of these rules. I believe that you all have the skills to complete this code. I will provide test cases for a lot of this as we move forward.

There are many ways to tackle the problem. You could learn through experiments. You could drive via tests. Regardless, you should attempt to write intelligent tests to check your work.

If you want to learn through experiments then you should code some of step, write a test and see if it works. This will, in turn, require you to also code some of isRedex, getBinding, substitute, and rename (and as you write that code I would encourage you to also write tests over your code as well) This can* lead to some AMAZING insights and can be quite fun. BUT… it can also lead us to introduce bugs in our code that are difficult to locate.

Alternatively, if you'd prefer to drive you work via tests. PRIOR to implementing anything in the source code. You will look at the inference rules provided/think about the task at hand (semantics), construct a hypothetical input to your function and discern its output, you then write a test to state what should happen (syntax). This will build a strong system of tests. THEN you can write code to solve the tests and the larger problem. For step to work getBinding and substitute must also work. For getBinding to work isRedex must also work. For substitute to work rename must also work. I recommend doing the easy things first. Think about the goal of isRedex, write tests, implement isRedex make sure you've passed the tests (if you haven't figure out is the issue that the test is wrong [feel free post your tests to piazza if you aren't sure] or the implementation is wrong). Repeat that process for getBinding. Note that your getBinding tests will also be testing isRedex for you (so that's cool). Then repeat the process on rename. Then do it on substitute. AND THEN, finally repeat the process on step. Your tests for step might test all of these things at once.

If you build the systems of test well you should have the option to run each test suite individually or to run all the test suites at once.

There is an art to writing unit tests that only test a single unit of functionality at a time. Until you have a system for doing this is can be incredibly time consuming. Perhaps aim for writing tests that are not unit tests, then if you are failing the test, try to break that test into smaller tests (while keeping to original test as well. E.g. If I am testing 1*2 + 10/5 and my test is fine but my interpreter fails to pass the I will go on to write a tests for 1*2 as

well as 10/5 as well as 2+2 all while keeping the original test. In doing I might isolate the issue in my code. All while increasing the power an value of my test system.

Tips : do the easier ones first (it's all relative)

- SearchAssign1 and 2 before DoAssignVar before DoAssignField
- Decl before Call before Obj before GetField

As we move forward I will provide more tests about these various sub-portions of page 15 of the lab 5 handout.

L5D6

- Substitution continued

L5D6T1 Substitution Cont.

Substitution was introduced as a concept to achieve static scoping in our interpreter back in Lab 3. In lab 4 we introduced the concept of lazy evaluation which required a change to the substitute function to allow non-valued expressions to be passed as input to the substitute function. In lab 5 we introduce mutability to our language using state monads to abstract the state of memory out of our evaluation. Now that we have stateless evaluation and lazy evaluation we run into the issue of free-variable-capturing. We choose to solve this issue by keeping substitution and modifying it to implement what is known as free-variable-capture-avoidance-substitution. This is completed by having substitute make a call to rename prior to doing any real work of its own.

Rename can be used to implement free-variable-capture-avoidance dependent on the policy provided to it. For this reason, I should implement rename prior to implementing substitution. I will also need to provide rename with the correct policy in substitution. The policy is that for any call to substitute of the form substitute(e,esub,x) I should:

1. Find all free variables in esub
2. Implement a policy that states for whatever I apply this policy to… for any variable x declared:
    a. if x is in the set of free variable of esub append a "$" to it and reattempt the policy
    b. else do not modify x
    c. do nothing to the state of the statemonad accumulated
3. Apply that policy on the expression e

Sometimes rename won't do anything significant and other times it does. Regardless that does not affect how we will implement the body of the subst function provided to us

subst will recurs on itself and not the outer substitute function.

subst will look a lot like substitute from lab 4.

I encourage you to write tests for each unit of logic in substitute as well as a single test over all of the units.

Here is a test on the Assign logic in substitution. It relies on unary, binary, and var logic as well so you should write tests for those and pass those prior to attempting to pass this if you are attempting test driven development.

Note that this test doesn't really require rename to work correctly but it should still work after rename is correctly completed.

```
"substAssign, substVar, substUnary" should "substitute on both subexpressions" in {
  // y = !y [ *a1 / y ]
  // from step possibly
  //    var y = true ; <stuff1> y = !y <stuff2>
  //    ignoring the other stuff that might happen
  val e1 = Var("y")               // y
  val e2 = Unary(Not,Var("y"))    // ! y
  val e = Assign(e1,e2)           // y = !y
  val esub = (Unary(Deref,A(1)))  // *a1
  val x = "y"

  // *a1 = !*a1
  val resultExpected = Assign((Unary(Deref,A(1))),Unary(Not,(Unary(Deref,A(1)))))

  assertResult( resultExpected ) {
    substitute(e,esub,x)
  }
}
```

L5D7

- CastOK

L5D7T1 CastOK

I hope that somewhere between lab 2 and lab 4 you became convinced that introducing type declarations to a language provides some value to us as programmers and has some drawbacks as well.

As many of you already know, there are also ramifications of types at a system level. Each type has a certain amount of space associated with it. Luckily, we don't have to worry about that when building our interpreter!

We do however want to implement our interpreter to allow for type casting. For this lab we will only require you to do a portion of the type casting and not all of it. Note Figure 5 states *"Ignore the CASTOKROLL and CASTOKUNROLL rules unless attempting the extra credit implementation."*

We do need to implement the rest of the CASTOK rules inside of the castOk function. We must also complete the TypeCast rule in the typeof function.

CastOkEq states that I can cast a thing of one type to the same type

CastOkNull allows me to upcast a null value to an object of variable length

CastOkObject1 allows me to shrink the size of an object, i.e. all the fields type pairs in the second object f2:t2 must exist in the first object.

CastOkObject2 allows me to increase the size of an object i.e. all the fields type pairs in the first object f1:t1 must exist in the second object.

Note that the implementation of TypeCast is dependent on the correct implementation of CastOk so you should get CastOk working for the above 4 cases prior to attempting TypeCast

Here are test cases for CastOK

```
import lab5._
"CastOkNull" should "perform CastOkEq" in {
  assertResult(true) {
    castOk(TString,TString)
} }
"CastOkNull" should "perform CastOkNull" in {
  assertResult(true) {
    castOk(TNull, TObj(Map.empty))
} }
"CastOkObject" should "perform CastOkObject1" in {
  val tfields0:Map[String,Typ] = Map.empty
  val tfields1:Map[String,Typ] = tfields0 + ("x"->TNumber)
  val tfields2:Map[String,Typ] = tfields1 + ("y"->TString)
  assertResult(true) {
    castOk(TObj(tfields2), TObj(tfields1))
  }
  assertResult(true) {
    castOk(TObj(tfields1), TObj(tfields0))
} }
it should "perform CastOkObject2" in {
  val tfields0:Map[String,Typ] = Map.empty
  val tfields1:Map[String,Typ] = tfields0 + ("x"->TNumber)
  val tfields2:Map[String,Typ] = tfields1 + ("y"->TString)
  assertResult(true) {
    castOk(TObj(tfields0), TObj(tfields1))
  }
  assertResult(true) {
    castOk(TObj(tfields1), TObj(tfields2))
} }
```

Here is a test case for typeof TypeCast

```
"TypeCast" should "perform TypeCast over CastOkEq" in {
  val t = TString
  val e = Unary(Cast(t),S("I can't do that dave."))
  assertResult(t){ typeof(empty,e) }
}
it should "perform TypeCast over CastOkNull" in {
  val t = TObj(Map.empty)
  val e = Unary(Cast(t),Null)
  assertResult(t){ typeof(empty,e) }
}
it should "perform TypeCast over CastOkObject1" in {
  val tfields0:Map[String,Typ] = Map.empty
  val tfields1:Map[String,Typ] = tfields0 + ("x"->TNumber)
  val tfields2:Map[String,Typ] = tfields1 + ("y"->TString)
  val t = TObj(tfields2)
  val e = Unary(Cast(t),parse("{x:5,y:'hello',z:true}"))
  assertResult(t){ typeof(empty,e) }
}
it should "perform TypeCast over CastOkObject2" in {
  val tfields0:Map[String,Typ] = Map.empty
  val tfields1:Map[String,Typ] = tfields0 + ("x"->TNumber)
  val tfields2:Map[String,Typ] = tfields1 + ("y"->TString)
  val t = TObj(tfields2)
  val e = Unary(Cast(t),parse("{x:5}"))
  assertResult(t){ typeof(empty,e) }
}
it should "fail to perform TypeCast" in {
  val tfields0:Map[String,Typ] = Map.empty
  val tfields1:Map[String,Typ] = tfields0 + ("x"->TNumber)
  val tfields2:Map[String,Typ] = tfields1 + ("y"->TString)
  val t = TObj(tfields2)
  val e = Unary(Cast(t),parse("{z:5}"))
  assertResult("fail"){
    try{ typeof(empty,e);"success" }
```

```
    catch { case _:Throwable => "fail" }
} }
```

L5D8

- grammars

L5D8T1 Grammars

This reading is not about lab 5 material. This reading is designed to prepare us to work on lab 6.

In lab 6 we will create a recursive decent parser for a subset of regular expressions. Then, if you'd like you can integrate regular expression testing to the javascripty interpreter that we wrote in lab 5.

Much of Lab 6 goes into topics from our Theory of Computation course. If you find that you really enjoy working with this material and would like to learn more about it please consider taking that theory focused course. These topics are also used in our Introduction to Compiler Construction course, you might also enjoy that theory and code focused course.

Some background facts that I want you to be aware of but that you don't need to KNOW.
Languages

- Regular languages (RL) :
    - rather expressive but not as expressive as CFL
    - can be expressed using a mathematic notation called regular expressions (very similar to the programming language of regular expressions but not the same thing)
    - in computer science we have a language known as regular expressions or RegEx. This language is an actualization of the mathematic notation for regular expressions
    - can represent 0^n for any value n
        - 0*
- Context free languages (CFL):
    - very expressive but not as expressive as CSL
    - can be used to express programming languages (which is a basis of this course)
    - can be expressed using a mathematic notation called context-free grammars (CFG)
        - there are many flavors of CFG
        - in this course we learn BNF as well as EBNF
    - can represent 0^n1^n for any value n
        - S ::= 0S1 | ϵ
- Context sensitive languages (CSL):
    - The most expressive of these three languages
    - This is representative of the language of Turing machines and effectively all that a modern computer is capable of (although I've been given the impression that the state-of-the art is moving away from this model and creating things that are even more powerful)
    - can be expressed using a mathematic notation called context-sensitive grammars (CSG)
    - can represent $0^n1^n2^n$ for any value n
        - S -> 0S2A | ϵ
        - 2A -> A2
        - 0A -> 01
        - 1A -> A1
- There are automata and machines that are also associated with each of these languages that are used to prove a variety of facts about each language category.

Review of CFG in BNF. I expect that we all already KNOW this but that its been a while so let us review…

- Terms for BNFCFGs
  - Object language : the language you want to describe
  - Meta language : the language that you use to describe the object language
  - Grammar : set of grammar rules that defines a language
  - Grammar rule : defines a single non-terminal
  - Meta-symbols : the operators of your CFG, in BNF there are exactly 3 meta-symbols : '::=', '|', and 'ϵ' meaning 'produces/is', 'or', 'lack of characterization/null lexeme' respectively.
  - Terminals : lexemes/words from the object language that are used in the meta language
  - Non-terminals : the variables that I define in the grammar, in BNF these are declared prior to the '::='
  - Meta-variable : a variable that denotes its own language but is not a non-terminal
  - Example:

    <u>G1</u>

    S ::= 0S0 | 1S1 | T
    T ::= 0 | 1 | ϵ

    The object language is a language of palindromes over '0's and '1's of any length. The meta language is a CFG in BNF. The CFG in BNF is G1. This grammar is defined using 2 grammar rules. The first rule defines the non-terminal S as being the terminal 0 followed by the non-terminal S followed by the terminal 0 OR the terminal 1 followed by the non-terminal S followed by the terminal 1 OR the non-terminal T. The second rule defines the non-terminal T as being the terminal 0 OR the terminal 1 OR the meta-symbol ϵ.

Demonstrate that a sentence exists in the language : We demonstrate that a sentence exists in the language defined by a grammar by drawing a parse trees for the sentence derived using the provided grammar. Please review old notes on this topic. If you would like additional notes on the topic please let me know via Piazza and I will take the time to typeset notes on the matter. (there are other ways to accomplish this task such as linear derivations but we will not cover those in this course)

Proving that a grammar correctly defines our object language : we do not cover how to do that in this course.

Practice CFG in BNF. Do a few of these to reassure that you know these. You should also practice drawing parse trees for a few sentences in each language

- Define the language of `a` i.e. { 'a' }
- Define the language of `a` 0-or-1 times i.e. { '', 'a' }
- Define the language of `a` 0 or many times i.e. { '', 'a', 'aa', … }
- Define the language of `ab` 0 or many times
- Define the language of `a` n-many times followed by 'b' n-many times
- Define the language of `a` n-many times followed by 'b' 0-or-many times followed my 'c' m-many times s.t. m = 2 * n

What about ambiguity?

- A grammar is shown to be ambiguous if I demonstrate 2 unique parse trees for the same sentence that exists in the language. (see old notes for more details)
- There are ways to prove that a grammar is not ambiguous. We do not cover that in this course.
- Some grammars are ambiguously defined and sometimes that is okay as it allows us to quickly express the form of our language without muddying it with associativity and precedence.
- Some languages are naturally ambiguous.

- Consider the following three definitions for the language of any amount of 0s and 1s in any order
    - A ::= AA | 0 | 1 | ϵ This is an ambiguous definition of the language
    - B ::= 0B | 1B | ϵ this is a right recursive definition of the same language
    - C ::= C0 | C1 | ϵ this is a left recursive definition of the same language
- For the practice problems above discern if your grammar is ambiguous or not

What about precedence? We discussed this once in a previous lab. Here are some formal notes on the matter. Please do your best to absorb this as we will be adding to this throughout lab 6.

- I can enforce precedence of words in the object language by introducing additional non-terminals and nesting higher precedence to be deeper in the grammar structure and thus deeper in my parse trees
- Higher precedence - lower in the parse tree, lower precedence - higher in the parse tree
- ALSO, atomic expressions of a language have the ultimate precedence, with respect to mathematic languages these are our operands. It should seem rather logical that an operand has higher precedence than any operation. consider the expression 12 + 34. The '+' sign is our operator and it has precedence… but the '12' and the '34' also have precedence. Their precedence is higher, as is the nature of them being atomic expressions
- I think the best way to learn is through examples

S ::= S + S | - S | n
n is a meta-variable for numbers.

The above grammar is ambiguous. I can rewrite it to be left recursive about any binary operations as follows but it will still be ambiguous.
S ::= S + n | - S | n

I can rewrite it to be right recursive as follows and it will not be ambiguous.
S ::= n + S | - S | n

Suppose that I want to rewrite this grammar to enforce the following precedence, n are atomic expressions, the '+' operator has a lower precedence than the '-' operator.

There are many ways that we can write such a grammar, below is my solution followed by an explanation of the structure and why this works.

S ::= A
A ::= B + A | B
B ::= - B | C
C ::= n

I swear to you there is a reason for having 4 rules written in this fashion… I introduced 3 new non-terminals, one for each operation and one for the atomic expressions. I let S simply fall down one step in the structure to the A rule. The A rule defines the nature of the '+' operation because I am told that '+' has the lowest precedence of all the words in the original language. I wanted it to be right recursive so I write _ + A. I also need this to link into the rest of my structure so I let the _ + A production fall down to B, I also provide the option that no '+' signs ever happen, hence the ' | B '. B is the non-terminal defining the language about the '-' operation. It follows the same logic as A but does about a unary operation rather than a binary operation. Then finally I drop down into C. These are my atomic expressions.

I will now rewrite that grammar using better names for my non-terminals

S ::= OptionManyPlus
OptionManyPlus ::= OptionManyNeg + OptionManyPlus | OptionManyNeg
OptionManyNeg ::= - OptionManyNeg | Atoms
Atoms ::= n

Or , the shorter version that I will use for the rest of this reading

S ::= P
P ::= N + P | N
N ::= - N | A
A ::= n

Now many of you are probably wondering why I don't just write this as 2 rules? Wont this next grammar work?

P ::= N + P | N
N ::= -N | n

And you're correct. All of these grammars shown define the same language. However, the larger structure becomes more useful when I want to extend the language and begin writing more complex languages. Consider the C/Atoms/A non-terminal of the longer structure. I want this to define the language of atoms in my language. My original language was defined as S ::= S + S | -S | n, but what if instead I want to define S ::= S + S | -S | (S) | n, s.t. () enforce ordering effectively creating new atomic expressions? I can add in a single thing to the language of A non-terminals. And my A non-terminal has some meaning that I can use to continue to expand the language quickly.

S ::= P
P ::= N + P | N
N ::= - N | A
A ::= n | (S)

As for the S non-terminal used throughout the longer structures the extensibility purpose of this really only comes into play often with naturally ambiguous languages. You don't need this for the materials in this course but I encourage you to use this structure regardless as it is a good habit to have. Here is an example of when the additional start symbol is useful.

Consider the following grammar, a subset of *javascripty* from Lab 2.

S ::= A | B
A ::= A + A | -A | n | (S)
B ::= B && B | !B | b | (S)
n denotes numbers
b denotes booleans

Write a new grammar that expresses the same language with precedence :
'-' > '+'
'!' > '&&'
S ::= A | B

A ::= AA
AA ::= AB + AA | AB
AB ::= - AB | AC
AC ::= n | (S)
B ::= BA
BA ::= BB && BA | BB
BB ::= ! BB | BC
BC ::= b | (S)

I can quickly write this form without worrying about its meaning or whether it is correct. Then, after I'm done I can analyze my work to attempt to understand the meaning of the work and create better names for my non-terminals and such.

L5: Additional Resources

L5: Closing Thoughts