The Gist of PL

CSCI 3155: Principles of Programming Languages

Author: Mr. Spencer D. Wilson

Disclaimer: This text is a series of notes designed with sections to be read prior to each day of class during the 10 week summer version of CSCI 3155. There are plenty of typos and bugs in the document but the core ideas are solid. If you would like to get involved in improving the notes please contact the author at: spencer.wilson@colorado.edu

Inspirers, Informants:

- Dr. Danial P. Friedman
- Dr. Evan Chang

Texts worth noting:

- ???The little schemer
- ???Csci3155_notes
- ???composition
- ??? GO
- ???little prover
- ???our actual text book
- ???

Lab 1: An intro to Scala and Interpretation

L1: Preface

L1D1 syllabus based

Print out, read and annotate the lab 1 handout

If you have not done so already. Please read the course syllabus.

Also, please skim chapter 1 of csci3155-notes provided in moodle

L1D2

- Functional Programming in Scala
- Scala Syntax
- Binding and Scope
- Scala Pattern Matching

L1D2T1: Functional Programming in Scala

In this course we will be programming in the functional subset of Scala. This means, none of our variables are mutable, they are all immutable, they cannot change (we declare them useing val rather than var).

"[if x = 2 you can't later say that x = 3… you said x = 2. What are you? A liar?!]" – Danial Freidman, pg???, The Little Schemer

This also means that we don't use traditional looping structures (e.g. we won't use for, while, until).

You might wonder, how can we accomplish anything in a language that doesn't allow loops and mutable variables?! Stick around, you might be surprised what you discover.

L1D2T2: Scala Syntax

Today's reading is meant to provide a brief example lead introduction to some simple Scala syntax. This shows one of many ways to solve the task at hand and frankly is a bit of overkill.

Consider the following mathematic equation: $a = (dt - v0*t) / (0.5 * t \wedge 2)$
Let's transform this into code using Scala syntax. Let's write this out in multiple lines.

First the function declaration: ----------

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double
```

- Note that the definition of a is dependent on 3 variables. This function operates on three doubles to return a double
- note that the comment above the function definition is not required

Start the function body: ----------

Let's define this using some intermediate variable. Since '(' ')' enforce ordering in most, if not all programming languages, I think the lowest precedence is on the '/' operator.

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        val numerator = ??? ;
        val denominator = ??? ;
        val result = numerator / denominator;
        return result;
}
```

Filling in blanks: ----------

Here I use a LOT of extra intermediates. This is not at all necessary. But this is an important (optional) intermediary step in code development that I believe is worth demonstrating.

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        val min2 = v0 * t;
        val numerator = dt – min2;
        val mult2 = t * t;   // seems easier then figuring out the power operator
        val denominator = 0.5 * mult2;
        val result = numerator / denominator;
        return result;
}
```

Reformat: ----------

The above code is valid Scala syntax but stylistically this has many extra characters that do not add any meaning to the code. Note that in Scala the ';' at the end of each line is not necessary and the 'return' key term is also option. On a call to a function, Scala will always return the value of the function body subject to the arguments provided (we'll come to better understand that statement before the semester is over). So, we could re-write the above code as:

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        val min2 = v0 * t
        val numerator = dt – min2
        val mult2 = t * t   // seems easier then figuring out the power operator
        val denominator = 0.5 * mult2
        val result = numerator / denominator
        result
}
```

Stylizing: ----------

This seems like overkill… can't I just write this as a 1 liner? Yes. But this multi-line code is easier to write and check for correctness… so you might consider – when learning a new programming language (PL) or just in general – to write code in multiple lines and then transform it to clearner code. I'll use the following judgment form '~~>' to define the operation semantic of 're-writing' my code. (We'll formally discuss the concept of Judgment Forms at a later date).

Moving forward I'll use `~~>` to denote that some changes are taking place between the code above and the code bellow. These changes are not always necessary but merely demonstrate changes that one could make to the code if they so desired.

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        val min2 = v0 * t
        val numerator = dt – min2
        val mult2 = t * t   // seems easier then figuring out the power operator
        val denominator = 0.5 * mult2
        val result = numerator / denominator
        result
}
```

~~>

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        val numerator = dt – (v0 * t)
        val mult2 = t * t  // seems easier then figuring out the power operator
        val denominator = 0.5 * mult2
        val result = numerator / denominator
        result
}
```

~~>

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        val numerator = dt – (v0 * t)
        val denominator = 0.5 * (t * t)
        val result = numerator / denominator
        result
}
```

~~>

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        val denominator = 0.5 * (t * t)
        val result = (dt – (v0 * t)) / denominator
        result
}

~~>

// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        val result = (dt – (v0 * t)) / (0.5 * (t * t))
        result
}

~~>

// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = (dt – (v0 * t)) / (0.5 * (t * t))
```

And voila… a 1 liner solution to the `a` function. I could make this a bit easier to read by removing unnecessary parenthesis…

‘*’ has higher precedence than ‘-‘ so the parentheses are unnecessary in the numerator.
‘*’ has equal precedence to ‘*’ so the parentheses are unnecessary in the denominator.
‘/’ has a higher precedence than ‘-‘ so the numerator must have parentheses around it
Math is left associative. Removing the parentheses from the denominator would change our semantic

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = (dt – v0 * t) / (0.5 * t * t)
```

Note that we could also remove the parentheses from the denominator but we would have to do some fancy math and the solution will not look as much like our mathematical equation

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = (dt – v0 * t) / 0.5 / t / t
```

If you prefer you could* technically* write the code in the form bellow… But I wouldn’t because that doesn’t look like clean Scala code.

```
// a = (dt – v0*t) / (0.5 * t ^ 2)
def a(dt:Double, v0:Double, t:Double):Double = {
        return (dt – (v0 * t)) / (0.5 * (t * t));
}
```

L1D2T3: Bindings and Scope

A common mistake for young programmers is confusing a functions parameter with the arguments of a function call.

Note that in Scala our function `a` (defined in reading L1D2T2) has it's own scope. Within the scope of the function our variable dt, v0, and t each have type Double. For this reason, I can write a script like the following and it will operate fine.

```
1. val t = "I am not a Double. But that is okay"
2. val v0 = "Still"::"Not"::"A"::"DOUBLE"::"Actually a List"::Nil
3. val my_special_double = 9.3
4. def a(dt:Double, v0:Double, t:Double):Double = {
5.         (dt – v0 * t) / (0.5 * t * t)
6. }
7. a(my_special_double, my_special_double, my_special_double)
```

Here we have lots of variable each with binding and use sights. Let's state which are which

- The instance of t on line 1 is a Binding site. It has no use sites
- The instance of v0 on line 2 is a Binding site. It has no use sites
- The instance of my_special_double on line 3 is a Binding site. It has 3 use sites, all of which are on line 7.
- The instance of `a` on line 4 is a Binding site. It has one use site on line 7 (yeah… we are binding the function to the name `a`)
- The instances of dt, v0, and t on line 4 are sort of binding sites but they aren't generally considered binding sites, they're just parameters. At a call to the function `a` these variables must be given a value for the execution of the function. In this course we won't worry about distinguishing parameters as call or binding sight. But, as seen in this example, the arguments found at a call-site might be use-sites, and we do care about those.

L1D2T4: Scala Pattern Matching

Please review the L1D2_notes_patternMatchingIntro.scala file (found on Moodle) for a brief introduction to Scala pattern matching semantics and use.

L1D3

- Linked lists
- Grammars (first exposure)
- Types

L1D3T1: Linked Lists

In learning a new programming language, I often find it useful to understand how a library for linked list could be made in the language.

Consider the following grammar to visualize a linked list structure over numbers ‘n’ (We’ll discuss grammars in depth later on but trust that this is sufficient to define a linked list).

S ::= End | n -> S
n is a metavariable that represents numbers

Here is a possible linked list made with the grammar

13 -> 16 -> 2 -> -134 -> End

Here are a few things that cannot be made from the above grammar. Figure out why… (don’t worry if you don’t get this yet, well explore the concept deeper at a later date.)

example	reason
‘hello’	
1 -> hi -> End	
8	
17 -> Nil	

I can turn grammar that visually depicts the concept of a linked list and transform it into a Scala with some ease. In this course well use sealed abstract classes available in Scala. First let’s pick the name for our linked list object. I think “LL” is a good name.

```
sealed abstract class LL    // S
```

Now LL is representative of ‘S’ in the above grammar. S has 2 productions… the “End” and the “n -> S”… so I will need to extensions of my LL class. Lets start with “End”. I think “End” is good enough as is… Note that this uses a case object…

```
case object End extends LL    // End
```

What about this “n -> S” thing? I interpret this to represent a ***node*** over ‘n’ and ‘S’. I know I’ve already declared the type of ‘S’ as “LL”. I don’t like the S so I’ll call it ‘lp’ for list_prime. What about the type of ‘n’? I’d like it to represent a number. I’ll use a Double because that makes for rather interesting abilities in the language. For this we will use a case class.

```
case class Node(n:Double, lp:LL):LL    // n -> S
```

Put it all together:

```
sealed abstract class LL    // S
case object End extends LL    // End
case class Node(n:Double, lp:LL):LL    // n -> S
```

Note that most of these expressions (LL, End, Node, n, lp) are defined by me the programmer and do not need to be written this way for any reason other than… I declared them this way

Now to write a few functions of our LL class (we won't cover methods in this course but I encourage you to google those if you're interested). Let's start with the easy and arguably important one, print. Below is one solution for printing something of type LL.

```
// Note : This code has not been tested for accuracy.
def print (l:LL):Unit = l match{
        case End => println("End")
        case Node( n, lp ) => {
                println( n.toString + "->" )
                print(lp)
        }
}
```

A recipe/heuristic for writing this….

1. Write the function definition to define a transformation from some data type to some new data type
2. Pattern match on the most interesting data type
    a. Select data
    b. Get type of data
    c. Determine cases of interest for that type and the task at hand
    d. Write out all the cases (or at least the ones that seem important)
3. Fill out the base cases first
4. Think inductively and fill out the other cases

Let's walk through the recipe for print.

1. Define the function:
    a. Name of the function: 'print' seems like a good name
    b. Input: I need a linked list to print, lets name it `l`. The type of `l` is LL
    c. Output: this is a print function so it doesn't really need to return anything. But it will print and the output type of Scalas println function is 'Unit'. So, I'll return Unit.
    d. Code:
    ```
    def print( l:LL ): Unit = ???
    ```
2. Pattern match on the most interesting data type
    a. Select data for matching: The function only has one input so this was easy. We should match on `l`.
    b. Type of `l`: LL
    c. Patters of LL: [End, Node(_:Double, _:LL)]
    d. Code:
    ```
    def print( l:LL ):Unit = l match {
          case End => ???
    ```

```
        case Node(n, lp) => ???
    }
```

3. Fill in the base case
    a. Base case: The base case is the 'End' case here. How do I know? Our other case Node(n, lp) has the potential of recursion, I am matching on something of type LL and lp also has type LL.
    b. What to do: We are printing a linked list, there a lot of options for HOW to do that, but personally I want to print the visual depiction of the linked list. So I'll print 'End'
    c. Code:
    ```
    def print( l:LL ):Unit = l match {
        case End => println(End)
        case Node(n, lp) => ???
    }
    ```
4. Inductive cases
    a. We only have one other case, Node(n, lp). Regardless I of the value of n, I want to print *n -> STUFF* where STUFF is the result of printing or sub-list `lp`. So I'll print `n ->` then I'll print STUFF
    b. Code
    ```
    def print( l:LL ):Unit = l match {
        case End => println(End)
        case Node(n, lp) => {
            println(n.toString + '->')
            print(lp)
        }
    }
    ```

Now I'm not totally satisfied with the above action. This prints a lot of new lines and I don't like it. So, I might rewrite it to print the linked list in a single line. Try that on your own if you'd like a moderate challenge. I've posted my solution at L3D3T1Q1

Want a hint for L3D3T1Q1? Consider writing a helper function to transform your linked list into a string.

Try writing some other functions for linked lists… Insert, pop, push, delete…. (please do attempt one of these now). Something worth noting as you embark on this endeavor… Recall, that we are using the functional subset of Scala. We do not have mutable variables. We won't change the value of the list we are operating on. Instead we might construct a modified copy of our list.

Here you might find it useful to look at the "L1D3_notes_linkedList.scala" (on moodle in the GistOfPL file) file to get a sense of how this code might look. Note that the data structure definition has changed slightly between the documents.

L1D3T2 Grammars (first exposure)

Please skim this part… we will revisit it later. I want you to start getting a sense of how these grammars work.

Consider the following grammar for a linked list structure over numbers 'n'.
S ::= End | *n* -> S
n is a metavariable that represents numbers

Let's dissect this. This CFG is written in BNF. There are a few things to note about the BNF language. This language has exactly 3 operators : '::=', '|', and 'ϵ'. Note that this particular grammar has only one grammar rule and does not use the 'ϵ' (epsilon) operator.

Each grammar rule in BNF form must have exactly 1 '::=' operator. This is called the defines operator. This is a binary operator of the form *non-terminal-aka-variable ::= expression(s)*

The *non-terminal-aka-variable* is where we declare a name for a variable or non-terminal that we would like to use. This can be anything that is not reserved by BNF or reserved in the language we are trying to define so it can be anything that does not exist in the set { ::=, |, ϵ, n, ->, End }.

expression(s) can use any series of terminals and non-terminals and the 'ϵ' to express the juxtaposition of lexemes in the object language. Each individual expression is separated by the '|' BNF operator.

So…
S ::= End | *n* -> S
n is a metavariable that represents numbers
can be read as: The language of linked list is any set of numbers, '->', and 'End' symbols that can be derived from a single non-terminal S such that S is End or S is n -> S (note that the new S does not need to be the same as the old S.

(That probably doesn't make too much sense but I wanted you to get a flavor for how grammars are formed)

L1D3T3 Types

Types can be quite useful in understanding the behavior of a program. We'll explore this deeper in future labs but today I want to show you some syntax to express the type of an expression. Consider, that in many programming langauges the '-' operation, be it unary or binary should be applied to numbers and it will return a number in the same order. Consider the Scala expression '(-(1) – (2))' what is the type of this expression? You likely already know without thinking but I'll walk through it.

(-(1) – (2))

In English: In Scala, 1 and 2 are integers (Int) and apply subtraction or negation to them will create a new Int. More specifically, applying negation to 1 creates an int -1. And applying subtraction to -1 and 2 will create an int -3.

In the form:
`
```
e: \tau because
    e1: \tau1 because
        …
    e2: \tau2 because
        …
```
`

(-(1) – (2)): Int
    -(1): Int
        1:Int
    (2): Int

Bellow are a few steps I took to write the above solution.
(-(1) – (2)): ???
    -(1): ???
        1:Int
    (2): Int

(-(1) – (2)): ???
    -(1): Int
        1:Int
    (2): Int

(-(1) – (2)): Int
    -(1): Int
        1:Int
    (2): Int

Now consider the following definition: a function is considered to be **well typed** if all paths in the function body return a value of the same type, whatever that type might be.

For example, the following function 'foo' is well typed, regardless of the value of x at a call to foo, the function will always return a tuple of integers (Int, Int):

```
def foo(x:Int) = {
    if (x > 2) { (1, x) } else { (x, 1) }
}
```

Here is a way to express the types:

```
if (x > 0) { (1, x) } else { (x, 1) }: Int
    x > 0: Boolean
        x: Int
        0: Int
    (1, x): (Int, Int)
    (x, 1): (Int, Int)
```

Meanwhile the bar function is not well typed because sometimes it returns a tuple of integers (Int, Int) and other times it just returns an Int.

```
function bar(x: Int) = {
    if (x > 2) { (1, x) } else { x }
}
```

Heres the treelike expression of the type of the function body of bar
if (x > 2) { (1, x) } else { x }: ???
    x > 0: Boolean
        x: Int
        0: Int
    (1, x): (Int, Int)
    x: Int

L1D3T3Q1: express the type of the baz function using the tree like structure presented to you. Determine if the function is well typed. (see solutions below)

```
function baz(x: Boolean, y:Boolean) = {
    if ( x && y ) { (1, (2, 3)) } else { ((1, 2), 3) }
}
```

L1D3 Solutions

L1D3T1Q1

```
def stringify( l:LL ): String = l match {
    case End => 'End'
    case Node(n, lp) => n.toString + ' -> ' + stringify(lp)
}
def print( l:LL ): Unit = println(stringify(l))
```

L1D3T3Q1 is baz well typed?

```
function baz(x: Boolean, y:Boolean) = {
    if ( x && y ) { (1, (2, 3)) } else { ((1, 2), 3) }
}
```

No, baz is not well typed, sometimes it returns (Int, (Int, Int)) and othertimes it returns ((Int, Int), Int). While it always represents three integers the type is actually different dependent on the values of x and y

if (x && y) { (1, (2, 3)) } else { ((1, 2), 3) }: ???
    x && y: Boolean
        x: Boolean
        y: Boolean
    (1, (2, 3)): (Int, (Int, Int))
    ((1, 2), 3): ((Int, Int), Int)

L1D4

- Abstract Syntax Trees
- More on Grammars
- Javascript vs Scala
- An AST for binary logic

L1D4T1: Abstract syntax trees

The next four labs of the course will deal with a neat data structure called the Abstract Syntax Tree (AST). The tree will become significantly more complex with each successive lab. AST are recursive data structures just like Linked Lists (LL) and Binary Search Trees (BST) but they are a bit more interesting in what kind of data they organize.

Consider the following definition of a linked list of integers named LL in Scala syntax

```
sealed abstract class LL
case object Empty extends LL
case class Node( d:Int, lp:LL ) extends LL
```

Essentially:

- LL
    - Empty
    - Node (d : Int , lp : LL)

The data structure has 2 possible core values either it is "Empty" or it is a Node with a value – represented by d – and another list (the recursive part) – represented by lp.

We have a few potentially interesting cases – such as the event where we have a Node but the lp is Empty i.e. I am in a list with only one value or I am almost at the end of my list. Node(_ , Empty)

Consider the following definition of a binary search tree named BST in Scala

```
sealed abstract class BST
case object Empty extends BST
case class Node( l:BST, d:Int, r:BST ) extends BST
```

Essentially:

- BST
    - Empty
    - Node (l : BST , d : Int , r : BST)

The data structure has 2 possible core values either it is "Empty" or it is a Node with a value – represented by d – and two other BST or sub-trees if you will (the recursive part) – represented by l and r.

This has many interesting cases depending on what I want to accomplish. In finding the minimum value of a tree I am interested in trees of the form Node(Empty, _ , _) in that I have a node with an empty left sub-tree. In finding the maximum value of a tree I am interested in trees of the form Node(_ , _ , Empty) in that I have a node with an empty right sub-tree.

Now let us consider the data structure of our AST named Expr used in Lab 1

```
sealed abstract class Expr

/* Literals and Values*/
case class N(n: Double) extends Expr

/* Unary and Binary Operators */
case class Unary(uop: Uop, e1: Expr) extends Expr
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr

sealed abstract class Uop

case object Neg extends Uop /* - */

sealed abstract class Bop

case object Plus extends Bop /* + */
case object Minus extends Bop /* - */
case object Times extends Bop /* * */
case object Div extends Bop /* / */
```

Or if you prefer:

- Expr
  - N(n : Int)
  - Unary(uop : Uop , e1 : Expr)
  - Binary(bop : Bop , e1 : Expr , e2 : Expr)
- Uop
  - Neg
- Bop
  - Plus
  - Minus
  - Times
  - Div

Note that this data structure relies on other data structures begin Bop and Uop. This data structure has three core values abstractions of numbers N(_:Int), abstractions of unary operations Unary(_:Uop,_:Expr), and abstractions of binary operations Binary(_:Bop,_:Expr,_:Expr). And then I have interesting values of Unary and Binary to consider based on the Uop and Bop values. This leads to 2 somewhat natural ways I could structure code when matching on Expr types.

Style 1 : don't match directly on case objects of Uop and Bop

```
def eval(e: Expr): Double = e match {
  case N(n) => ???
  case Unary(uop,e1) => uop match {
    case Neg => ???
    case _ => ???
  }
  case Binary(bop,e1,e2) => bop match {
    case Plus => ???
    case Times => ???
    case Minus => ???
    case Div => ???
    case _ => ???
  }
  case _ => ???
}
```

Style 2 : match directly on case objects of Uop and Bop

```
def eval(e: Expr): Double = e match {
  case N(n) => ???
  case Unary(Neg,e1) => ???
  case Binary(Plus,e1,e2) => ???
  case Binary(Times,e1,e2) => ???
  case Binary(Minus,e1,e2) => ???
  case Binary(Div,e1,e2) => ???
  case _ => ???
}
```

NOTE: The bottom line of each pattern match statement is "case _ => ???". This is a catch all case. This is not required by match statements. It is just something that I do when I code, I find it useful. You can take it or leave it. Here our cases are all inclusive so the final case is not at all necessary.

L1D4T2: More on grammars – you can skim this part

Now let's do a seemingly silly exercise in understanding the expressions handled by a 5-function calculator. It operates on the following grammar:

e ::= e bop e | uop e | n | (e)
*bop ::= + | - | * | /*
uop ::= -
s.t. n is a metavariable for numbers

(it's okay if that doesn't make perfect sense but trust me… it works)

Now when I see the expression "1+2+3" I often read this as "one plus two plus three" and that's great because I am thinking about arithmetic… But in this course, we will be looking at this expression in abstraction. So, I would encourage you to read it as "the-number-one cross the-number-two cross the-number-three". Note: the goal there was to read an expression as a **juxtaposition** of its **lexemes** i.e. the goal there was to read a sentence as a grouping of words. Here, each lexeme/word happened to by only 1 character long, but this won't always happen.

More over I would interpret the operators as:
+ Cross
- Dash
* asteryx
/ forward-slash

and the operands as (less interesting):
1.0 the-number-one-point-zero
5.7 the-number-five-point-seven
NaN the-number-not-a-number-aka-the-most-pain-in-the-#-kind-of-number

Do you have to do this? No… but I think as we advance in this course you'll find this a useful habbit to have.

Expression/sentence: 1.0 + 2.0
Characters identified: one, full-stop, zero, space, cross, space, two, full-stop, zero
Lexemes/words identified in order: the-number-one-point-zero, cross, the-number-two-point-zero

L1D4T3 JS vs Scala – stop skimming….

Now let us consider why this is important.

Now let us consider why this is important. In JavaScript I can write code to perform these basic 5 functions. "1.0+2.0+3.0", "-0.0/-0.0", "26.0-12.0*17.0" and I could probably take that same code and put it in scala and yield the same result. This works for these examples… there are likely a few edge cases where it will not be equivalent. (I won't tell you what they are yet… I want you to try to find them on your own)

Here I can mostly run my JavaScript directly in Scala but we'll use this as an exercise to prepare for the next labs – where JavaScript expressions such as "5+console.log("hello")+"hi"" will not run in Scala. To do this I will use an AST to abstract my JavaScript expressions to a usable form in Scala, below is the representation I prefer for our AST that we named Expr.

Expr
- N(_:Double)
- Binary(_:Bop,_:Expr,_:Expr)
- Unary(_:Uop,_:Expr)

Bop
- Plus
- Times
- Div
- Minus

Uop
- Neg

So that '1' read as "the-number-one-point-zero" becomes N(1.0)
And '1 + 2' read as "the-number-one cross the-number-two" becomes Binary(Plus, N(1.0), N(2.0))

Consider what 1+2*3 will parse to, write your guess down on paper. Then try it out in the worksheet provided. Your Scala worksheet for Lab 1 provides a brief explanation about how to use the parser.

Cool fact… the Lexor and Parser are already written for us so we don't have to worry about how "1+2*3" is transformed to the thing it becomes… that's a topic we'll cover in Lab6. For now we just figure out how to transform the output to N(7.0)

The goal now is to write code that operates on the "Expr" data structure, applying arithmetically logical operations to each piece.

L1D4T4 AST for binary logic

Let us consider a different language (because if I show you this on the 5-function calculator grammar then you might not learn anything)

e ::= e ^ e | ~ e | b
s.t. b is a metavariable for a Boolean

let's define e using the term Funish
let's define Funish as follows

```
sealed abstract class Funish
case class B(b:Boolean) extends Funish
case class Xor(e1:Funish, e2: Funish) extends Funish
case class Not(e1:Funish) extend Funish
```

we can view this in it's alternate form
Funish

- B(_:Boolean)
- Xor(e1:Funish, e2: Funish)
- Not(e1:Funish)

Suppose that a lexor and parser already exist to transform all sentences in the object language to an AST object.

We should be able to write an interpreter for the language without concerning ourselves with the order of operations.

Let us assume that '~' is the unary operation `arithmetic not` and will operate on Booleans the same way in Scala as it does in JavaScript. Let us also assume that '^' is the `logical exclusive-or` binary operator and this too will operate the same way in both Scala and JavaScript.

I'll return a Boolean for this since all expressions can evaluate to a Boolean in the language. Below is an example of this code. With commented out additional options for how to write the bodies of our case statements.

```
def interperet(e:Funish):Boolean = e match {
        case B(b) => b
        case Not(e1) => ~interperet(e1)

        // {
        //      val b1 = interperet(e1)
        //      ~b1
        // }

        // {
        //      val b1 = interperet(e1)
        //      val not_b1 = ~b1
        //      not_b1
        // }

        case Xor(e1,e2) => interperet(e1) ^ interperet(e2)
        // {
        //      val b1 = interperet(e1)
        //      val b2 = interperet(e2)
        //      val b1_xor_b2 = b1 ^ b2
        //      b1_xor_b2
        // }
}
```

Review the above code. Do we understand why this works? Demonstrate your understanding by completing the small interpreter exercise provided in Lab1.

As mentioned in class and Piazza, the "Gist of PL" has a lot of small bugs in it that I want to fix up. But some students have already written notes into the text (poor planning ahead on my part). So I am posting new chapters as we move forward and not altering the base text. Here is Chapter 2.

Table of Contents

Lab 2: Interpreter without theory (although we'll start learning the theory)

L2: Preface

In lab 1 you were introduced to Scala syntax and a few useful tools for understanding and interpreting programming languages including Abstract Syntax Trees (ASTs) and Context Free Grammars (CFGs). In this Lab we will look at some quirks of the syntax of the JavaScript Language and build a larger interpreter than lab 1 over a subset of JavaScript that we'll call *javascripty*$_1$. The history of JavaScript is quite fascinating and worth looking into if you want some interesting facts. It is a very popular programming language with a massive community of developers that you will likely need to use at some point in your career.

L2D 1

- Print out, read and annotate the lab 2 handout
- Types
- Grammars
- If you have time, skim chapter 2 of csci3155_notes.pdf

L2D1T1: Types

In lab 1 we discussed Linked Lists(LL), Binary Trees(BT) - or rather Binary Search Trees (BST) -, and Abstract Syntax Trees (AST).

The AST in lab 1 was named “Expr” and it had the following definition in Scala:

```
sealed abstract class Expr extends

/* Literals and Values*/
case class N(n: Double) extends Expr

/* Unary and Binary Operators */
case class Unary(uop: Uop, e1: Expr) extends Expr
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr

sealed abstract class Uop

case object Neg extends Uop /* - */

sealed abstract class Bop

case object Plus extends Bop /* + */
case object Minus extends Bop /* - */
case object Times extends Bop /* * */
case object Div extends Bop /* / */
```

The AST from Lab1 on *javascripty*$_0$ expresses inputs of a 5-function calculator, specifically, in JavaScript syntax.

Here is the grammar for that language:
e ::= e bop e | uop e | n
bop ::= + | - | * | \
uop ::= -
s.t. n is a meta-variable for numbers

In Lab 2 through Lab 5 will build on this Expr type often looking at how some subset of JavaScript or JavaScript-*esque* language and how node interprets code. Lab 2 has a rather exponential growth to it. Lab 2’s Expr definition is as follows:

```
sealed abstract class Expr extends

/* Variables */
case class Var(x: String) extends Expr

/* Declarations */
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr

/* Literals and Values*/
case class N(n: Double) extends Expr
case class B(b: Boolean) extends Expr
case class S(str: String) extends Expr
case object Undefined extends Expr

/* Unary and Binary Operators */
case class Unary(uop: Uop, e1: Expr) extends Expr
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr

sealed abstract class Uop

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

sealed abstract class Bop
```

```
case object Plus extends Bop /* + */
case object Minus extends Bop /* - */
case object Times extends Bop /* * */
case object Div extends Bop /* / */
case object Eq extends Bop /* === */
case object Ne extends Bop /* !=== */
case object Lt extends Bop /* < */
case object Le extends Bop /* <= */
case object Gt extends Bop /* > */
case object Ge extends Bop /* >= */

case object And extends Bop /* && */
case object Or extends Bop /* || */

case object Seq extends Bop /* , */

/* Intraprocedural Control */
case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr

/* I/O */
case class Print(e1: Expr) extends Expr
```

This AST expresses the grammar on page 5 of the lab 2 handout, shown below:
e ::= x | n | b | str | undefined | uop e1 | e1 bop e2 | e1 ? e2 : e3 | const x = e1; e2 | console.log(e1)
uop ::= - | !
bop ::= , | + | - | * | / | === | !== | < | <= | > | >= | && | ||
s.t. x are variables, n are numbers , b are Booleans, and str are strings

We will go deeper into this later, but for now I want to focus on our values. Below is the grammar rule for values in the *javascripty*$_1$ language.
v ::= n | b | undefined | str

In lab 1 we only had one type in the interpreter, namely Number. In lab 2 we now have four types (four kinds of values), namely Undefined, Boolean, Number, String. Because JavaScipt is so free in moving between types we might find it useful to write a series of functions that will translate between these types for us.

Aside – goes deeper than needed for our class:

> if we think of types as sets of values then each type has an ordinance, a size, to it. Undefined is a type of size one, it only has one value, namely undefined. Boolean is a type of size 2, it's values are true and false. Numbers are typically just Doubles which is a large but finite set, on 64 bit machine it has size 2^64. Strings are infinite.

Let us consider how to translate value Exprs to Booleans. Here is a function declaration in Scala:

def toBoolean(e : Expr) : Boolean

The toBoolean function takes an Expr as input and returns a Boolean as output. Suppose, I call toBoolean with the argument: **true**… toBoolean(true)… well that won't quite work, that input is a Boolean value and not an Expr value…. But I could call toBoolean with an Expr that abstracts a Boolean value. The abstraction of **true** to an Expr is **B(true)**… toBoolean(B(true))… that's better. What should it return? Well it took in an abstract Boolean as input and it must return a Boolean literal so it should probably return the Boolean literal that the abstraction represents: **true**.

Here is a testcase we could write for the toBoolean function in Scala:

```
assertResult( true ){
        toBoolean(B(true))
}
```

Now what if I input **0.0**? That wouldn't work because it is a Double. But I could abstract the Double to an Expr that represents the Double. Let's look at N(0.0) as input to the toBoolean function.

toBoolean(N(0.0)) should evaluate to the literal Boolean false. Why?

N(0.0) is an abstraction of the Number 0. The toBoolean function is always going to return either true or false. Does the number 0 seem more like false or true? 0 is a ***falsey*** value and thus we return a literal false value from toBoolean.

Here is a testcase we could write for the toBoolean function in Scala:

```
assertResult( false ){
        toBoolean(N(0.0))
}
```

How would I confirm this conjecture? I like to use the following trick… in a node interpreter put the following:

<the_valued_expression> ? console.log("It is a truethy value") : console.log("It is a falsey value")

So for 0.0 I would write the following in node:
0.0 ? console.log("It is a truethy value") : console.log("It is a falsey value")

Sure enough this prints "It is a falsey value" (And then it returns undefined.)

We'll look into why that trick works at a later date.

Aside, we extend this to use non-valued expressions as follows, but it's not useful in this lab:

(<any_expression>) ? console.log("It is a truethy value") : console.log("It is a falsey value")

L2D1T1Q1: Now, consider another number like 7.5…. Is this truethy or falsey? Go find out AND write a test case for this into your source code. You'll find a solution to this problem at L2D1T1Q1.

Now how do we test this for all inputs? We wouldn't want to. But we can think about a few general cases and a few edge cases of each type abstraction that can come as an input.

Consider the following number inputs to toBoolean:

- -0.0
- NaN
- -Infinity
- Infinity
- …

And what about our other types?

- Strings
    - “”
    - “0”
    - “false”
    - “hello world”
    - …

- undefined
    - undefined

I’d encourage that you take this opportunity to practice Test Driven Development(TDD) and the Scientific process. I am not an expert with either of these but this is a great place to practice. Below is an example of how this works

Scientific process on data “false”:String
Hypothesis: “false”:String is a falsey value
Experiment: in Node execute the following program

```
(“false”) ? console.log(“it is truethy”) : console.log(“it is falsey”)
```

Observation:

The program printed “it is truethy”. Then it returned undefined

Conclusion: Hypothesis was incorrect. The JavaScript string “false” is not a falsey value. It is actually a truethy value.
I guess, that here a scientist would alter the hypothesis and repeat the experiment, but we as computer scientists wouldn’t bother with such things typically, since we know the value is truethy already.

<Here I might pause to think about why the string “false” is *truethy* even though the Boolean value `false` is *falsey* and this string looks a lot like the Boolean value `false`>

TDD:

1. Write a test for toBoolean on input “false”.
    a. Abstract the string “false” to an Expr: S(“false”)
    b. Identify return value: true
    c. Write the test in your test environment

```
assertResult(true){
    toBoolean(S(“false”))
}
```

2. Implement a partial solution to the toBoolean function that passes all tests you have written thus-far.
3. Write more tests that you know to be good tests. Write code to solve the new tests without breaking your old tests.

L2D1T2: Grammars

I promised I would talk about this more and here it is. A formal introduction to BNF grammars!

Here is a video that was easy enough to follow. It uses slightly different terms than I do but it's a good introduction to BNF grammars if you're interested

- https://www.youtube.com/watch?v=8cEhCx8pwio

CONTEXT: Here in lab 2 we start dealing with a significantly more complex AST than in Lab1. In this lab the goal is to attempt to formalize the logic that JavaScript uses. In Lab 3 we will look at formal ways of expressing this logic (it is super cool). But a building block for this formalization is understanding grammars. So, we'll start working with grammars now, to lessen the difficulty in lab 3.

`$\epsilon$`: the greek symbol 'epsilon' is often used in mathematics and sciences to denote a margin of error or something insignificant. In this course we will see this symbol used in Context Free Grammars (CFG) in Backus-Naur Form (BNF). In BNF-CFG (BFNG) the epsilon stands for a lack of characterization.

Some Terminology : worth glancing at now and reviewing later

- **Language** : a set of sentences/expressions
- **Sentences** : juxtaposition of lexemes/words
- **Lexemes/words** : juxtaposition of characters

- **Object language :** The language you want to describe
- **Meta language :** The language that you use to describe an object language
    - In this course we describe an object language "JavaScript" using a meta language "Scala"
    - When dealing with grammars we describe an object language using the meta language "BNFG"

- **Grammar**: is a set of grammar rules that describe a language
- **Grammar rules** : combine terminals, non-terminals, meta-variables and meta-symbols to express a grammar
- **Terminals** : lexemes/words in the object language OR characters in the alphabet of the object language (in this course we will mostly be working with the flavor of lexemes/words rather than characters The other thing has to do with writing Lexors… we won't cover it but if you are interested you will enjoy our Theory of Computation course and would probably also enjoy our Compilers course)
- **Non-terminals** : variables that we declare to make life easier (allows for recursion in our grammar rule)
- **Meta-variable** : special variables like "n" or "str" that we use to denote numbers or strings. Often used to describe infinite sets. (more on this later)
- **Meta-symbol** : operator of the grammars language (this course focuses on BNF grammars so the meta-symbols are ::=, |, ϵ)
    - **::=** : defines
    - **|** : OR
    - ϵ: the lack of characterization (sort of equivalent to an empty string)

EXAMPLE:

Here is a context free grammar (CFG) with a single grammar rule in BNF form. The grammar operates over the alphabet: {0,1} i.e. it talks about a language of '0's and '1's

My Grammar
S ::= 1S | 0S | ϵ

Let's dissect this. This CFG is written in BNF. There are a few things to note about the BNF language. This language has exactly 3 operators: '::=', '|', and 'ϵ'. Note that this particular grammar rule actually uses all three but this is not always necessary of a BNFG.

- Each grammar rule in BNF must have exactly one definition operator '::='. This is a binary operator of the form *non-terminal-aka-variable ::= production(s)*
  - The ***non-terminal****-aka-variable* is where we declare a name for a variable/non-terminal that we would like to use. This can be anything that is not reserved by BNF (always { ::=, |, ϵ }) or reserved in the object language, the language we are trying to define, aka our **terminals** (here that is { 0, 1 }) so in context it can be anything that does not exist in the union of the sets { ::=, |, ϵ} and { 0, 1 } i.e. { ::=, |, ϵ } U { 0, 1 } i.e. { ::=, |, ϵ, 0, 1 }
  - *expression(s)* can use any series of terminals and non-terminals and the 'ϵ' to express the juxtaposition of **lexemes**/words in the object language. Each individual expression is separated by the '|' BNF operator.

So…
My Grammar
S ::= 1S | 0S | ϵ
- Being as technical as I can be, the above grammar is read as "*the non-terminal S can be the terminal 1 followed by the non-terminal S OR the non-terminal S can be the terminal 0 followed by the non-terminal S OR the non-terminal S can be the meta-symbol epsilon"*
- Being a bit less intense " *S can be 1S OR it can be 0S OR it can be ϵ*"
- I might also state that at a high level this grammar describes "the language of all binary sentences"
- Here, I chose S as my non-terminal because it doesn't exist in the set { ::=, |, ϵ, 0, 1 } and it is a common practice of grammars that the first rule has S as the non-terminal because this is often referred to as your '**S**tart-symbol'

Binary Strings
S ::= 1S | 0S | ϵ

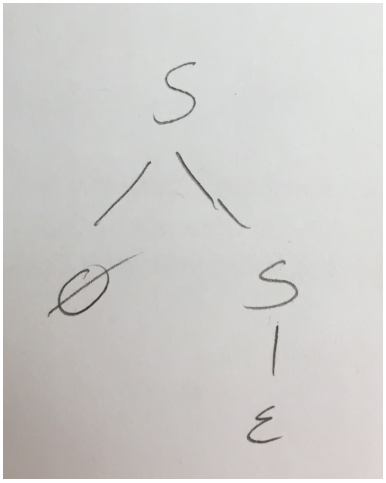
Now I claimed that the grammar describes all binary sentences HOW DO I PROVE IT? Well in this course we won't prove it (that's a topic for Theory of Computation -TOC). We will however demonstrate that a variety of sentences exist in the language defined by our grammar.

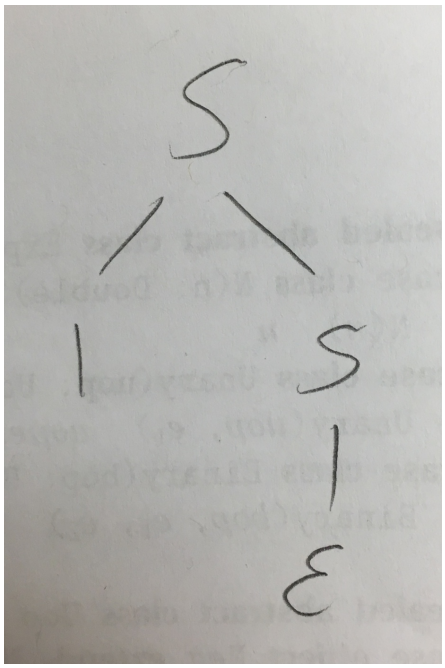
There are two great tools demonstrating that a sentence exists in the language defined by a grammar, Linear Derivations and Parse Trees. This course will test you on Parse Trees only (because they have significantly more value for expressing structure of a grammar) but I will show you some linear derivations here as they are often a helpful tool for understanding grammars.

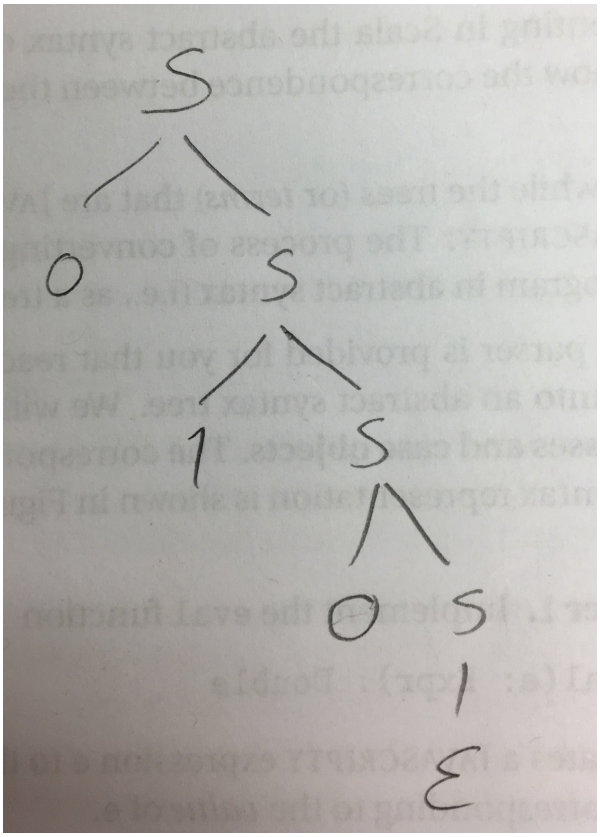
NOTE: We will always need to start with a single instance of our “START Symbol” i.e. the first non-terminal defined in our grammar which, here, is “S”

Lets think of some sentences that exist in the language defined by our grammar.

- 0
- 1
- 010
- 1010
- write a few more so you can try them on your own

- Demonstrate that the sentence “0” exists in the language defined by the grammar rule S ::= 1S | 0S | ϵ
  - linear derivation
    - S => 0S => 0ϵ i.e. 0
  - parse tree
    - 

- Demonstrate that the sentence “1” exists in the language defined by the grammar rule S ::= 1S | 0S | ϵ
  - linear derivation
    - S => 1S => 1ϵ i.e. 1
  - parse tree
    - 

- Demonstrate that the sentence “010” exists in the language defined by the grammar rule S ::= 1S | 0S | ϵ
  - linear derivation
    - S => 0S => 01S => 010S => 010ϵ i.e. 010
  - parse tree
    - 

- L2D1T2Q1 (solution available at the end of todays reading): Demonstrate that the sentence “1010” exists in the language defined by the grammar rule `S ::= 1S | 0S | $\epsilon$` using both a linear derivation and a parse tree.

Now that is neat and all but I think we could stand to see more examples. Bellow I have provided a few more grammars for languages with the alphabet { 0, 1 } :

0^n1^m n >= 0, m >= 0
S ::= 0S | S1 |

0^n1^m n>0, m > 2
S ::= 0AB11
A ::= 0A | ϵ
B ::= 1B | ϵ

0^n1^n n>=0
S ::= 0S1 | ϵ

Here is one that uses a different alphabet: { +, -, /, *, *NUMBERS-im-lazy-and-wont-write-that-out-as-a-set… }:*
5 function calculator
S ::= S A S | B S | n
A ::= + | - | / | *
B ::= -
s.t. n is a meta variable for numbers

I encourage you to think of strings that exist in the language defined by each grammar and demonstrate that the string exists using a parse tree. If you do so, feel free to post them publicly on Piazza for feedback from your peers (as long as they aren't solutions to lab problems).

L2D1: Solutions to in reading questions

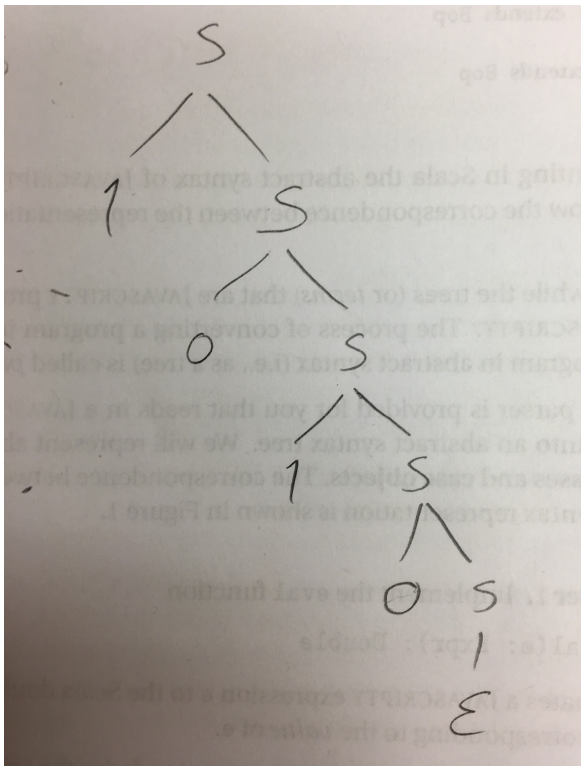
L2D1T1Q1: is 7.5 truethy or falsey

7.5 is a truethy value. For this reason, the following is a valid unit test in correct Scala syntax for our toBoolean function

```
assertResult( true ){
        toBoolean(N(7.5))
}
```

Don't forget that toBoolean takes an Expr as input, so I need to write **N(7.5)**, not just **7.5**

L2D1T2Q1: demonstrate 1010 is in the language definded by S ::= 0S | 1S | ϵ

- linear derivation
  - S => 1S => 10S => 101S => 1010S => 1010ϵ i.e. 1010
- parse tree
  - 

L2D2

We have four topics today:

- Using toBoolean
- Testing
- Grammatic ambiguity
- Grammars to inference rules

L2D2T1: Using toBoolean in the Interpreter

We talked about truethy and falsey… and the toBoolean function, specifically how to test the toBoolean function.

But how do we use it? When is it useful?

Well in most PL (programming languages) the types we can input to operators are constrained so the following expressions aren't possible in most PL:

- 1 + "hi"
- ! "goodbye"
- 22 <= "hello"
- undefined && true

Here are some interesting ones… that we won't cover in the course

- {} + []
- [] + {}
- Lookup javascript WAT for more info

In most PL these operators have input types that it expects

- '/' forward-slash and '-' Dash generally expects numbers
- '+' Cross often expects numbers but some languages also expect strings
    - Cross might refer to addition
    - Cross might refer to concatenation

Look at the following operators and think about the types you typically expect of the operands (in C? in Python? In Matlab? In Scala? In JavaScript? In other languages that you know and use, what is common?):

- !
- *
- <=
- ||
- ,
- &&
- ===
- ==

Which of the above do you think have a preference toward operating on Booleans?

Prior to this course I've seen the '!' Bang operator is applied exclusively to Booleans. JavaScript allows me to apply it to Booleans

- !true
- !false

it also allows me to apply it to values of other types

- !""
- !"false"
- !"hello world"
- !0.0
- !NaN
- !5.7

It can also be applied to complex expressions such as (or rather the value of the expression):
! (console.log("Programming Languages") || console.log("are awesome") && "fact")

But why can it do this? Think about it and take a few guesses. I would recommend discussing it with a peer or a rubber duck before continuing this reading…

I think that for any expression of the form !e_1 where e_1 can be literally any expression in *javascript*$_1$ will interpret e_1 to a value, lets call that v_1. It will then cast the value to a Boolean, lets call that b_1. It then finally applies '!' to b_1 to construct some new Boolean value that I might call b_1' (b-one-prime). Thus, it seems like I can apply logical not (!) to anything but I really only ever apply it to Booleans. This is just one interpretation of what it does. Below is an inference rule that beautifully explains the concept

EvalNot

$$\frac{E \vdash e_1 \Downarrow v_1 \qquad b' = !\text{toBoolean}(v_1)}{E \vdash !e_1 \Downarrow b'}$$

Note that all inference rules are subject to a grammar and some operational semantics. Here are the ones I am thinking about when writing that inference rule

Operational semantic: $E \vdash e \Downarrow v$

Grammars

expressions	e ::= x | n | b | str | undefined | uop e_1 | e_1 bop e_2 | e_1 ? e_2 : e_3 | const x = e_1; e_2 | console.log(e_1)
values	v ::= n | b | undefined | str
unary operators	uop ::= - | !
binary operators	bop ::= , | + | - | * | / | === | !=== | < | <= | > | >= | && | ||
Variables	x
Numbers (doubles)	n
Booleans	b ::= true | false
Strings	str

We will learn more about these later. But look over the pattern. Write your code to evaluate Not, see if you can guess how the inference rule "EvalNot" works.

L2D2T2: Testing

In lab 2 we look at a decent sub-set of JavaScript that we name *javascripty$_1$.* We would like our interpreter (the eval function) to do the same things that *javascripty$_1$* will do. We have many types. Some of these types have interesting values that lead to important edge cases that we should handle.

Heuristic for success.

1. Pick an operator in the language
2. Use the scientific process to test that operator against a variety of inputs in a JavaScript environment (use Node)
3. Write test cases for each of these in your Scala environment
4. Implement `eval` to pass what those tests represented
5. Iterate while finding a work flow that makes this easiest for you

Consider the following when thinking of test cases about each operator:

- Numbers
  - 0
  - -0
  - NaN
  - Infinity
  - –Infinity
  - 6.7 not an edge case but you should have a non-edge case in your tests
- Strings
  - ""
  - "hello" not an edge case
- Booleans
  - true
  - false
- Undefined
  - This only has the value **undefined** but I encourage you to test this using an expression of the form *console.log(e1)* where e1 is any expression in the language. NOTE: console.log(<anything>) evaluates to undefined in Java Script

Writing tests….

- In a file path something like *<root of project>/src/test/scala/jsy/student* you will find a file 'Lab<n>Spec.scala'
- In this file, near the bottom you will find a line like

```
class Lab2Suite extends Suites(
  new Lab2SpecRunner,
  new Lab2JsyTests
)
```

- Often you will run the Lab2 Suite but you might find it useful to only run the individual Flat Specs that are contained in the Suites: Lab2SpecRunner, Lab2JsyTests
- You might also want to create new Flat Specs

- Here is an example of how I might do this. Just before the class Lab2Suite I place

```
class AndTests(lab2: Lab2Like) extends FlatSpec {

  import lab2._

  "And" should "return non-intuitive results and short circuit" in {
    val e1 = N(-0)
    val e2 = eval(Binary(And,N(-0),Print(S("Hello"))))
    assert(e1 === e2)
  }

}

class Lab2AndTestsRunner extends  AndTests(Lab2)
```

- I then extend the test Suites

```
class Lab2Suite extends Suites(
  new Lab2SpecRunner,
  new Lab2JsyTests,
  new Lab2AndTestsRunner
)
```

- Now, If I want, I can continue to run the Lab2Suite as I often do. Or I can actually just run the Lab2AndTestsRunner if I would prefer to run less tests (which might be wise at times)
- If I wanted to do a sanity check and hard code the pass for this test I would go to eval in lab2.scala and add a line to pass the test:

```
case Binary(And,N(-0),Print(S("Hello"))) => N(-0)
```

- To get a bit closer to coding the logic of the test I might write

```
case Binary(And,v1@N(-0.0),_) => v1
```

- In the end I might find it useful to write a line that looks more like:

```
case Binary(And,e1,_) if !toBoolean(eval(e1)) => ???
```

  But I leave it to you to figure out why that line is useful. Your end solution might not use this line and that is totally cool, but if it does, that's rad too.

L2D2T3: Grammatical Ambiguity

See L2D1T2 for a review of terms

Previously we saw grammars over the alphabet : { 0 , 1 }. Let's look into a concept about CFG called ambiguity.

- L2D2T3Q1 (solutions in their usual place, bottom of the days reading) Warm up :
  - Write a grammar over the alphabet { 0 , 1 } that represents the language of any binary string of length 3 or more
    - i.e. in regular expressions (0|1) (0|1) (0|1) (0|1)*
    - e.g. the language is a the set {000,001,010,… 1111, …, 000010101010001, … }
  - Write a grammar over the alphabet { a , b } that represents the language of any combinations of `a` and `b` or (lack of combination) s.t. the string of length 4 or more
    - i.e. in regular expressions (a|b) (a|b) (a|b) (a|b) (a|b)*
  - Write a grammar over the alphabet { x , y , z } so that each string in the described language uses x exactly 1 time
    - i.e. in regular expressions (y|z)*x(y|z)*

- Some facts about languages (history and context):
    - There are 3 kinds of languages: Regular Languages, Context Free Languages, and Context Sensitive Languages.
    - Regular Languages
        - Require no memory to interpret
        - We describe these languages using Regular Expressions
            - (0|1) (0|1) (0|1) (0|1)*
            - (a|b) (a|b) (a|b) (a|b) (a|b)*
            - (y|z)*x(y|z)*
        - We will talk about these more during lab 6
    - Context Free Languages (CFL)
        - Require 1 stack to interpret
        - I dare say most programming languages are Context Free Languages but don't quote me on that…
        - We describe these languages using Context Free Grammars (CFG)
            - There are many flavors of CFG such as BNF which is what we use in our course
            - S ::= 0S | 1S | ϵ in Regular Expression: (0|1)*
            - S ::= 0S0 | 1S1 | 0 | 1 cannot be represented in Regular Expressions
    - Context Sensitive Languages
        - Require 2 stacks to interpret
        - Note: adding more stacks beyond that do not allow for more computational power
        - These languages are described using Context Sensitive Grammars (CSG)
        - We won't cover these in this course. Take Theory of Computation (TOC) to learn more

- Terminology
    - You might want to review terminology from the first grammars reading L2D1T2
    - **Ambiguity** : in a language refers to an ability to construct the same sentence in a language in 2 different ways

- Context : Why do I care about ambiguity
    - This becomes rather important when we go into Lab 6. But we choose to talk about this now so that you get experience with the concept now and more practice understanding grammars. Ultimately ambiguity can be a good thing because it allows you to quickly express concepts but it can also hurt us if we wanted to code the logic of the grammar directly (causes recursive paths without exit conditions).

Continue on next page

A short recipe for finding ambiguity in a grammar

1. Look at each grammar rule of your BNF
    a. Look at each production of the grammar rule
        i. If any of the productions are recursive about their non-terminal
            1. Determine if the grammar rule is Left-recursive, right-recursive, or both, or neither
            2. if both
                a. the grammar rule is ambiguous and thus the grammar as a whole is ambiguous. END. AMBIGUOUS
            3. else
                a. continue
        ii. else
            1. continue
2. End. Not necessarily unambiguous. But we have evidence that supports it to be unambiguous.

NOTE: this only finds a particular kind of ambiguity. There are some languages that are naturally ambiguous and this recipe likely won't find the ambiguity. But that's a topic for Theory of Computation and not our course.

Consider the following grammar

- Grammar 0
  S ::= 0 | 1 | SS
- This grammar is a CFG in BNF with 1 grammar rule and thus has 1 non-terminal – "S"
- The non-terminal S has 3 productions
    - 0 – a non-recursive production
    - 1 – a non-recursive production
    - SS – a recursive production
- Because S has at least 1 recursive production I must ask, are the productions left or right recursive? Or both? Or Neither?
    - The cheap way to do this is:
        - If a production begins with a recursive instance of the non-terminal it describes then the production is left recursive
        - If a production ends with a recursive instance of the non-terminal it describes then the production is right recursive
        - It is possible that the production is both left and right recursive
        - It is possible that the recursion is infixed in the production and it is neither left or right recursive (I think we call this linearly recursive)
    - Our recursive production "SS" both starts and ends in our non-terminal S and thus this single production is both left and right recursive
- Because the combination of our productions is both left and right recursive the grammar is ambiguous

Consider the following grammar that describes the same language as Grammar 0, but in a different manner

- Grammar 1
  S ::= 0S | 1S | S0 | S1 | ϵ
- This grammar is a CFG in BNF with 1 grammar rule and thus has 1 non-terminal – "S"
- The non-terminal S has 5 producitons
    - 0S – a recursive production
    - 1S – a recursive production
    - S0 – a recursive production

    - S1 – a recursive production
    - ϵ - a non-recursive production
- Because S has at least 1 recursive production (it has 4 of them) I must ask, are the productions left or right recursive? Or both? Or Neither?
    - 0S – ends in S – right recursive
    - 1S – ends in S – right recursive
    - S0 – starts in S – left recursive
    - S1 – starts in S – left recursive
- Because some of the productions of "S" are left recursive and others are right recursive, the grammar is ambiguous

L2D2T3Q2 Try it yourself:

- Consider the following grammar that describes the same language as Grammar 0 and Grammar 1, but in a different manner
    - Grammar 2
      S ::= TS | ST | ϵ
      T ::= 0 | 1

Consider the following grammar that describes the same language as Grammar 0, 1, and 2, but in a different manner

- Grammar 3
  S ::= TS | ϵ
  T ::= 0 | 1
- Grammar 3 has 2 grammar rules
    - Rule 1 describes S and has 2 productions
        - TS – recursive – ends in S – right recursive
        - ϵ – not recursive
    - Rule 1 about S is right recursive
    - Rule 2 describes T and has 2 productions
        - 0 – not recursive
        - 1 – not recursive
    - Rule 2 is not recursive
    - The grammar does not seem to be ambiguous. But I haven't proven it and I don't need to.

Aside: Now many of you might be wondering, how do I prove that something is unambiguous? GREAT QUESTION, take Theory of Computation to find out.

Now how do I demonstrate that a language is ambiguously defined by a grammar that I know to be ambiguously defined? That we do cover in this course! I must find a sentence that exists in the language. If I can draw two parse trees for that sentence then the grammar ambiguously defines the sentence. Note the sentence will need to use a combination of productions that makes the grammar ambiguous to begin with. This might require rather lengthy sentences at times.

You might take a moment to practice drawing parse trees.

Lets look at our grammars again.

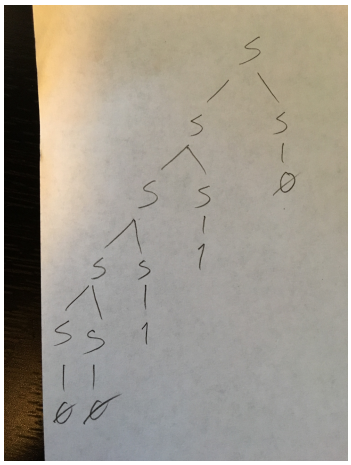
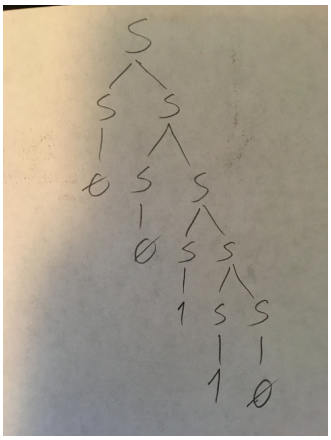
- Grammar 0
  S ::= 0 | 1 | SS
- Grammar 1

S ::= 0S | 1S | S0 | S1 | ϵ

- Grammar 2
  S ::= TS | ST | ϵ
  T ::= 0 | 1
- Grammar 3
  S ::= TS | ϵ
  T ::= 0 | 1

Consider the sentence '00110'. This sentence exists in all 4 of our grammars. Draw as many parse trees as you can for each of the above grammars. If you draw more than 1 parse tree for the sentence then you have sufficiently shown that the grammar is ambiguous. If you can't then you haven't demonstrated anything (and that's okay *unless of course I'm asking you to demonstrate ambiguity, in which case you might need a different sentence*)

Here are left-associative and right-associative parse trees to demonstrate that the sentence '00110' exists in the language defined by Grammar 0, I am not drawing all possible trees because this has A LOT of them

- Left Associative
  - 
- Right associative
  - 

- L2D2T3Q3 Wrap up exercise: (limited solutions available)
  - Consider the following grammar
    - G0
      S ::= aS0 | Sb | 1S | ϵ
  - Note the alphabet here is { a, b, 0, 1 }
  - Use the recipe provided to determine if the grammar is ambiguously defined?
  - Consider the following sentences

        - a0
        - 111b
        - ba10
        - a10bb
    - for each example sentence provided
        - does the sentence exist in the language defined by G0?
        - If so, demonstrate this by drawing a parse tree for the sentence?
        - If the sentence is ambiguously defined by the grammar then provide at least 1 additional parse tree to demonstrate the abiguity.
- Summary Notes
    - If a production begins with a recursive instance of the non-terminal it describes then the production is left recursive
    - If a production ends with a recursive instance of the non-terminal it describes then the production is right recursive
    - A grammar rule is ambiguous if the combinations of its productions are both left and right recursive
    - A grammar is ambiguous if any of its grammar rules are ambiguous
    - We demonstrate ambiguity of a grammar by selecting a sentence in the language defined by the grammar that is ambiguously defined and showing 2 parse trees for the sentence
    - In this course we do not cover how to prove that a grammar is ambiguous or unambiguous

L2D2T4: Grammars to Inference Rules

Please skim this part.

A few terms well see used:
Judgment, judgment form, rule of inference / inference rule, derivation, brush up on grammars, |- OR \Vdash OR turnstyle, ⇓ OR \Downarrow, ⟶ OR \longrightarrow, ⟶* OR \longrightarrow *

Context. Grammars are great at defining what a language looks like i.e. how multiple characters from an alphabet can be placed next to each other to produce meaning but the grammar does not inherently state very much about the meaning of the sentences in the language it defines. One great tool for expressing this meaning is to use inference rules. Here we will talk about inference rules dealing with the Funish language discussed during Lab 1 and move on to creating other inference rules which allow us to add operations on top of a language (because that is cool)

To start writing inference rules I need a grammar, an operational semantic (which also requires a judgment form), and a reason to do this in the first place

Reason…
I want to know how a sentence in the language evaluates to a Boolean

Grammar…
Funnish
S ::= S ^ S | ~S | b
s.t. b is a meta-variable for Boolean values

operational semantic…

First I need to write the judgment form that will govern my semantic. Since my goal is evaluation I will use the evaluation judgment form

⇓

Specifically Ill write in the form

Input ⇓ *output*

⇓ is a common judgment form that is binary in nature. To the left of it I place a sentence variable to the right I place a target variable

Since I want to turn sentences in the Funnish language into Booleans I will write the operational semantic as :

$S \Downarrow b$

Note* this is only one of many ways in which we could write this operational semantic

So now that we have all the prerequisites we can start writing inference rules. As a rule of thumb, I need at least 1 inference rule per production of my object sentence (S)

RULE1

$S_1 \Downarrow b_1$ $S_2 \Downarrow b_2$ $b = b_1 \text{ ^ } b_2$

$S_1 \text{ ^ } S_2 \Downarrow b$

RULE2

$S_1 \Downarrow b_1$ $b = \sim b_1$

$\sim S_1 \Downarrow b$

RULE3

$b \Downarrow b$

Or alternatively for typesetting

C : S_1 ^ S_2 \Downarrow b

J0 : S_1 \Downarrow b_1

J1 : S_2 \Downarrow b_2

J2 : b = b_1 ^ b_2

C : ~S_1 \Downarrow b

J0 : S_1 \Downarrow b_1

J1 : b = ~ b_1

C : b \Downarrow b

A few tricks worth noting. It is useful to subscript your non-terminals. Note that the premise will often use bold lettering or special colors to denote execution of operator as opposed to the abstraction of it.

Please download the L2D2_notes_inferenceRulesToCode.scala file for details on how this translates to code

L2D2: Solutions

L2D2T3Q1:

- Warm up solutions:
  - NOTE: there are many ways to write grammars the solutions given here are not the only possible way to write them
  - Write a grammar over the alphabet { 0 , 1 } that represents the language of any binary string of length 3 or more
    - S ::= TTT | TS
    - T ::= 0 | 1
  - Write a grammar over the alphabet { a , b } that represents the language of any binary string of length 3 or more
    - S ::= TTTT | TS
    - T ::= a | b
  - Write a grammar over the alphabet { x , y , z } so that each string in the described language uses x exactly 1 time i.e. (y|z)*x(y|z)*
    - S ::= x | TS | ST
    - T ::= y | z

L2D2T3Q2:

- Try it yourself solution – ambiguity of a grammar
  - Consider the following grammar that describes the same language as Grammar 0 and Grammar 1, but in a different manner
  - Grammar 2
    S ::= TS | ST | ϵ
    T ::= 0 | 1
  - Grammar 2 has 2 grammar rules
    - The fist rule describes S and has 3 productions
      - TS – recursive – ends in S - right recursive
      - ST – recursive – starts in S – left recursive
    - The grammar rule for S is ambiguous and thus Grammar 2 is ambiguous

L2D2T3Q3

- Wrap up exercise limited solutions
  - G0
    S ::= aS0 | Sb | 1S | ϵ
  - G0 is ambiguous.
  - a0
    - exists in the language defined by G0
    - it is not ambiguously defined
  - 111b
    - exists in the language defined by G0
    - it is ambiguously defined
  - ba10
    - DNE in the language defined by G0
  - a10bb
    - exists in the language defined by G0
    - it is ambiguously defined
  - for each example sentence provided

- does the sentence exist in the language defined by G0?
- If so, prove this by drawing a parse tree for the sentence?

If the sentence is ambiguously defined by the grammar then provide at least 1 other parse tree: No trees provided. Please feel free to post your solutions on Piazza to confirm correctness.

L2D3

- Environments of Evaluation
- Rules of Inference

L2D3T1: Environments

An environment of evaluation or **scope** is the region of code in which we perform some operations.

Evaluate the script "x+y" … Depends on the programming language, but the program will probably thow some kind of error in that it is observing unbound/free variables. You can't really do the work if you don't know the value of x and y.

But if I said evaluate the script "x+y" in the environment [x -> 5 ; y -> 8] (where x is mapped to 5 and y is mapped to 8) then you can tell me the value of the expression is 13

In lab 2 we look at one of many ways to allow for variables to be used in our scripts, namely the use of an environment.

We will let [] be the empty environment. To the environment we will add x_i -> v_i in the environment to map variable x_i to value v_i.

Consider the following:

[]
```
const x = 5;
const y = 8;
x + y
```

You are well aware by now that this above expression evaluates to 13 but let's look at a few steps involved in deriving this value. I start by consuming the first line and change the environment that I am working in. changing the expression to the following:

[x -> 5]
```
const y = 8;
x + y
```

I can then repeat the process, adding the value of y to the environment we get the following:

[x -> 5 ; y -> 8]
```
x + y
```

And then interpret the final expression in the environment (perhaps skipping a few steps) we find the following:

[x -> 5 ; y -> 8]
```
13
```

Our Lab uses a self-defined type named 'Env' which is defined as follows:

type Env = Map[String , Expr]

A Map in Scala is essentially a dictionary. It is a set of key value pairs such that each key is unique. In our example environment: [x -> 5 ; y -> 8] the environment has 2 keys. The keys are 'x' and 'y'. Each of them is mapped to a value, 5 and 8 respectively.

To lookup the value associated with a key in a map we say: <map_name> (<key_name>)

To extend the map with a new key value pair we say: <map_name> + (<key_name> -> <value>)

To construct our environment [x -> 5 ; y -> 8] I wrote the following code:

```
Map() + ("x" -> N(5)) + ("y" -> N(8))
```

Note that Env maps Strings to Expr. This is why the x and y have quotations around them (to make them strings) and this is why the 5 and 8 are not literal 5 and 8 but rather are Expr abstractions of 5 and 8, namely N(5) and N(8) (which makes them Expr instances)

The definition of Env and its helpers are as follows

```
type Env = Map[String, Expr]
val empty: Env = Map()
def lookup(env: Env, x: String): Expr = env(x)
def extend(env: Env, x: String, v: Expr): Env = {
  require(isValue(v))
  env + (x -> v)
}
```

If you prefer you can use the lookup and extends functions as well as the empty key-term:
empty, here is another way to code the environment [x -> 5 ; y -> 8]

```
extend(extend(empty, 'x', N(5.0)), 'y', N(8.0))
```

Recall the expression:

[]
```
const x = 5;
const y = 8;
x + y
```

Here is another way of viewing the evaluation of our expression from earlier,
eval(Map() , ConstDecl(x,N(5.0),ConstDecl(y,N(8.0),Binary(Plus,Var(x),Var(y))))) === N(13.0)

I am using the judgment ~~> to denote that the eval function sort of takes a step. In truth our eval function in lab 2 implements a big step logic and so it does not really have "steps" but I think this is useful to walk through anyway. I can add a Kleene star `*` to the judgement to state that it happens 0-or-many-times. (more on this later) THIS IS NOT HOW OUR EVAL FUNCTION WORKS**, its just a useful visualization of how it could work**…
Here is the big picture:

```
eval( Map(), ConstDecl(x,N(5.0),ConstDecl(y,N(8.0),Binary(Plus,Var(x),Var(y)))) )
~~>*
eval( Map(x -> N(5.0)), ConstDecl(y,N(8.0),Binary(Plus,Var(x),Var(y))) )
~~>*
eval( Map(x -> N(5.0),y -> N(8.0)) , Binary(Plus,Var(x),Var(y)) )
~~>*
N(13.0)
```

If taking really small steps we get the following:

```
eval( Map(), ConstDecl(x,N(5.0),ConstDecl(y,N(8.0),Binary(Plus,Var(x),Var(y)))) )
~~>
eval( Map(x -> N(5.0)), ConstDecl(y,N(8.0),Binary(Plus,Var(x),Var(y))) )
~~>
eval( Map(x -> N(5.0),y -> N(8.0)) , Binary(Plus,Var(x),Var(y)) )
~~>
eval( Map(x -> N(5.0),y -> N(8.0)) , Binary(Plus, N(5.0),Var(y)) )
~~>
eval( Map(x -> N(5.0),y -> N(8.0)) , Binary(Plus, N(5.0), N(8.0)) )
~~>
eval( Map(x -> N(5.0),y -> N(8.0)) , N(13.0) )
~~>
N(13.0)
```

L2D3T2: Rules of inference

- L1D1T1Q2 Terminology
  - See csci3155-notes section 2.3 Jugements Context
  - See csci3155-notes section 1.2.2 Jugements Context
  - ⇓ evaluates in 0 or many steps
  - ⟶ steps in exactly 1 step
  - ⟶ * steps in 0 or many steps
- Warm up – thinking exercise
  - Consider the language defined by the grammar below:
    - WarmUpGrammar
      Stuff ::= Stuff Bit | Bit
      Bit ::= 0 | 1
    - Explain what this language represents.
  - How would you invert each bit? In a sentence in this language?

- Content
  - When writing inference rules, we need to have a starting set(s) and a goal set(s) and then an idea of how we might transform the sets. So, let's start with this example…
  - Consider the language of binary expressions (a set… all languages are sets…) defined by the following grammar
    - SomeBinary
      S ::= S 0 | S 1 | ϵ
  - I want to write a few inference rule that allow us to transform a sentence from this language into a different sentence in this same language namely it's inverse. Let's have our judgment form be TheThing. TheThing will take 2 sets the input and output set. Here our sets are both defined as "S" and so our **operational semantic** will be : S TheThing S'
  - Note that S' is read S prime. It is the output of applying TheThing to S.
  - Here are the rules for this transformation
    - Rule 1 :

      $$\frac{\text{S TheThing S'}}{\text{S 0 TheThing S' 1}}$$
    - Rule 2 :

      $$\frac{\text{S TheThing S'}}{\text{S 1 TheThing S' 0}}$$
    - Rule 3 :

      $$\frac{}{\epsilon \text{ TheThing } \epsilon}$$
  - Note that our input sentence is found in the conclusion (the judgment form below the line) and to the left of our judgment (in this case "TheThing")
  - Let's dissect those Rules:
    - Rule 1
      - If our input sentence looks like S0 then I should return S'1 where S' is the result of recusing our judgment, "TheThing", on the S noted in our input sentence.
    - Rule 2
      - If our input sentence looks like S1 then I should return S'0 where S' is the result of recusing our judgment on the original S
    - Rule 3
      - If we have the empty sentence 'ϵ' then we should return the empty sentence 'ϵ'
  - Now let's perform a sanity check to see if this works. First, think of a few sentences that exist in the language defined by our grammar
    - SomeBinary
      S ::= S 0 | S 1 | ϵ
    - Sentences in the language
      - '' i.e. the instantiation of ϵ, an empty sentence
      - 0
      - 1
      - 01
      - 00101
    - Extra practice: consider drawing parse trees for each these sentences subject to the provided grammar

- Suppose I was to apply our judgment to these sentences. What should happen?
  - '' TheThing ''
  - 0 TheThing 1
  - 1 TheThing 0
  - 01 TheThing 10
  - 00101 TheThing 11010
- So if I invert the bits of the sentence '01' I should get '10'. Our operational semantic S TheThing S' should take in a sentence S and return S' as the inversion of S. So 01 TheThing 10 should be true. Let's do a **derivation** to prove that this is in fact true. This one will be rather guided.

Step 1 : write the bar and the judgment in your conclusion

. _____ .
TheThing

Step 2 : write the input to your judgment subject to the operational semantic

. _____ .
01 TheThing

Step 3 : identify which Rule you should apply. Here my input is 01 being a sentence ending in the value 1. So, I should use Rule 2.
Recall: Rule 2 :
. S TheThing S' .
S 1 TheThing S' 0
NOTE: you do not need to write the name of the rule on the line but I personally often find it helpful when dealing with more complex sets of inference rules and I would encourage you to get in the habit of labeling your lines.

Rule 2 . _____ .
01 TheThing

Step 4 : Start the first premise of your rule. Here, I have only one premise. It states that I must apply the thing to S. What is S? To answer this I might ask what is my input form 'S1' with respect to my input '01'? And here I find that S is '0'. NOTE: when doing derivations our premise need to have that horizontal bar.

. _____ .
Rule 2 . 0 TheThing .
01 TheThing

Step 5 : The same as Step 3. Identify which rule to apply to this premise. Here my input is '0'. My sentence is of the form S0 and I apply Rule 1.
Recall Rule 1 :

. S TheThing S' .
S 0 TheThing S' 1

Rule 1. .
Rule 2 . 0 TheThing .
01 TheThing

Step 6 : is the same as Step 4 but here I am applying this to the 1st premise of my root. What is S? Compare 'S0' to '0'. This is a bit harder to see then the previous step… But here, S is ϵ. '0' has nothing before it and the lack of characterization is represented by the ϵ of BNF CFG.

. .
Rule 1. ϵ TheThing .
Rule 2 . 0 TheThing .
01 TheThing

Step 7 : Same as Step 3 and Step 5. Identify which rule to apply. Here I apply Rule 3.
Recall Rule 3 :

. .
ϵ TheThing ϵ

Rule 3 . .
Rule 1. ϵ TheThing .
Rule 2 . 0 TheThing .
01 TheThing

Step 8 : Here I check to see if there are any premise for Rule 3. There are none, Rule 3 is an axiom. Since I have satisfied all of the premise (or in this case lack thereof) I can complete this line. Given ϵ, TheThing should output ϵ.

Rule 3 . .
Rule 1. ϵ TheThing ϵ .
Rule 2 . 0 TheThing .
01 TheThing

Step 9: I have completed the first premise of Rule 1. I look at Rule 1 and note that the rule only has a single premise. Because I have satisfied all the premise I can complete the line. Given S0, The thing should output S'1 where S' is the output of applying TheThing to S. What is S' for us here? It is found to the right of the "TheThing" in the conclusion of our premise. (highlighted above). S' is ϵ so S'1 is ϵ1.

Rule 3 . .
Rule 1. ϵ TheThing ϵ .
Rule 2 . 0 TheThing ϵ 1 .
01 TheThing

Here, the aesthetics are bothering me in my mind we should either always write the 'ϵ' or we should not write it at all.

With ϵ

$$\dfrac{\dfrac{\dfrac{}{\epsilon\ 0\ \text{TheThing}\ \epsilon\ 1}\ \text{Rule 3}}{\epsilon\ \text{TheThing}\ \epsilon}\ \text{Rule 1}}{\epsilon\ 01\ \text{TheThing}}\ \text{Rule 2}$$

Without ϵ – where reasonable.

$$\dfrac{\dfrac{\dfrac{}{0\ \text{TheThing}\ 1}\ \text{Rule 3}}{\epsilon\ \text{TheThing}\ \epsilon}\ \text{Rule 1}}{01\ \text{TheThing}}\ \text{Rule 2}$$

Step 10 : I have completed the first premise of Rule 2. I look at Rule 2 and note that the rule only has a only one premise. Because I have satisfied all the premise I can complete the line. Given S1, The thing should output S'0 where S' is the output of applying TheThing to S. What is S' for us here? It is found to the right of the "TheThing" in the conclusion of our premise. (highlighted above in both flavors of the derivation). S' is ϵ 1 or just 1 if you prefer so S'0 is ϵ10 or 10 if you prefer.

With ϵ

$$\dfrac{\dfrac{\dfrac{}{\epsilon\ 0\ \text{TheThing}\ \epsilon\ 1}\ \text{Rule 3}}{\epsilon\ \text{TheThing}\ \epsilon}\ \text{Rule 1}}{\epsilon\ 01\ \text{TheThing}\ \epsilon\ 10}\ \text{Rule 2}$$

Without ϵ – where reasonable.

$$\dfrac{\dfrac{\dfrac{}{0\ \text{TheThing}\ 1}\ \text{Rule 3}}{\epsilon\ \text{TheThing}\ \epsilon}\ \text{Rule 1}}{01\ \text{TheThing}\ 10}\ \text{Rule 2}$$

Q.E.D. (quod erat demonstratum). TADA! Viola!

I set out to show that the 'TheThing' will indeed correctly invert the bits of our example sentence '01' to the sentence '10'.

NOTE: This did not definitively prove that our inference rules are correct for all possible inputs to "TheThing" this merely provides evidence to support the matter. We will not cover such a proof in this course.

L2D3T2Q1 Try it yourself: Take a moment and try a derivation for the following sentence '0110', or if you prefer 'ϵ0110' using the inference rules defined over 'TheThing' (solution at bottom)

Consider the following operational semantic that will return 1 if all the values are 0 and 0 if any of the bits are 1. We'll use the judgement '! =', to relate the set S to the set b (grammar bellow) following the operational semantic ! S = b…

Okay, I lied a little bit to you earlier in this text, the input doesn't ALWAYS have to be to the left of the judgment, sometimes it is infixed in the judgment. As seen here we have, '! *input* = *output*'. Later on we will see that sometimes the judgment also takes multiple inputs…

Grammar:
S ::= Sb | b
b ::= 0 | 1

Rule 1 $\dfrac{\quad ! S = b \quad}{! S0 = b}$

Rule 2 $\dfrac{\quad}{! S1 = 0}$

Rule 3 $\dfrac{\quad}{! 0 = 1}$

L2D3T2Q2 Try it yourself: Does this system of inference rules work on an S '001010'? What should the be the value of b for !001010 = b? How many levels will there be in our derivation tree if we were to draw the derivation? Draw the derivation.

L2D3T2Q3 Try it yourself: Where this gets really interesting is when you write your own inference rules. Try writing your own set of inference rules. Here is a particularly challenging one that I think you're ready for.

- Judgement: ⇓
- Alternate judgment: input ⇓ output
- Operational Semantic: S ⇓ n
- S ::= Sb | b
  b ::= 0 | 1
- n is a number
- NOTE: not all of your premise need to be subject to the same judgment as their conclusion
- After you write the rules you should perform a derivation to check your work

L2D3T2: Solutions to in text Questions

L2D3T2Q1 Solution1: derivation of TheThing on input 'ϵ0110'

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon \text{ TheThing } \epsilon}\text{ Rule 1}}{\epsilon 0 \text{ TheThing } \epsilon 1}\text{ Rule 2}}{\epsilon 01 \text{ TheThing } \epsilon 10}\text{ Rule 2}}{\epsilon 011 \text{ TheThing } \epsilon 100}\text{ Rule 1}}{\epsilon 0110 \text{ TheThing } \epsilon 1001}\text{ Rule 3}$$

L2D3T2Q1 Solution2: derivation of TheThing on input '0110'

Rule 3. ———————————— .
Rule 1. ϵ TheThing ϵ .
Rule 2. 0 TheThing 1 .
Rule 2. 01 TheThing 10 .
Rule 1 . 011 TheThing 100 .
0110 TheThing 1001

L2D3T2Q2 Solution: ! 001010 = ???

! 001010 = 0 because there is at least a single 1 in the sentence
The derivation tree will have a depth of 2.

Rule 2. ———————— .
Rule 1. ! 00101 = 0 .
! 001010 = 0

L2D3T2Q3 Solution: S ⇓ n

NOTE: there are many ways we could write this. Below is the preferred flavor for this course.

Rule 1 : . S ⇓ n n' = n*2 .
S0 ⇓ n'

Rule 2 : . S ⇓ n n' = n*2 + 1 .
S1 ⇓ n'

Rule 3 : . ———————————— .
1 ⇓ 1

Rule 4 : . ———————————— .
0 ⇓ 0

L2D4

- Only one topic today, but it is a very useful one, translating inference rules to code
- Please also consider re-skimming Chapter 2 of csci3155-notes.pdf (stored at the top of Moodle)

L2D4T1: Inference Rules to Code

Review L2D2T4 and L2D2_notes_inferenceRulesToCode.scala file

So, inference rules are super awesome. They are mathematic models that express how to work with a judgement and transform information between various sets. Consider the following Grammar, Judgment, Operational Semantic, and Inference rule

1. Grammar: bellow is the grammar for lab 3 which has a lot of the features of our lab 2 grammar
    a. Expressions e ::= x | n | b | str | undefined | uop e_1 | e_1 bop e_2 | e_1 ? e_2 : e_3 | const x = e_1; e_2 | console.log(e_1) | p(x) => e_1 | e_1 (e_2) | typeerror
    b. Values v ::= n | b | undefined | str | p(x) => e_1 | typeerror
    c. unary operations uop ::= - | !
    d. Binary operations bop ::= , | + | - | * | / | === | !== | < | <= | > | >= | && | ||
    note the final || is the or operator, not a use of our BNF metavariable |
    e. Booleans b ::= true | false
    f. Strings str
    g. Function names p ::= x | ϵ
    h. Value environments E ::= ϵ | E [x -> v]
2. Judgment
    a. |- ⇓
    b. Typset as \vDash \Downarrow
    c. Spoken as turnstyle downarrow
    d. This judgment takes 2 inputs and spits out a single output
    e. Alternate form : *$input_1$* |- *$input_2$* ⇓ *output*
3. Operational Semantic
    a. E |- e ⇓ v
    b. Typset as E \vDash e \Downarrow v
    c. Spoken as E turnstyle e downarrow v
    d. Meaning in environment `E` expression `e` evaluates to value `v`
    e. Here we use our judgment to relate inputs `E` and `e` to the output `v`
    f. Note that E, e, and v are all declared in our above grammar
        i. `E` is the language of value environment that represents a mapping of 0-or-many x -> v where x are “variables” and v are “values”
        ii. `e` is the language of all *javascripty* expressions
        iii. `v` is the language of all *javascripty* values and is proper a subset of `e`
    g. Note these symbols are abstractions.
    h. The type of e and v in our interpreter (the Eval function) are both Expr
4. Inference rule example
    a. EvalNeg:

$$\frac{E \vdash e_1 \Downarrow v_1 \qquad n' = -\text{toNumber}(v_1)}{E \vdash -e_1 \Downarrow n'}$$

    b. This inference rule tells me what to do on the inputs “E” and “- e_1”.
    c. In English, we could say: In environment E our abstract expression – e_1 evaluates to the abstract number n’ where n’ is –1 * the number represented by the abstract value v_1. And v_1 is the output of evaluating e_1 in environment E

So how do I turn this into code? There are many ways to do this and it often changes based on what the inference rule looks like and sometimes is dependent on your inference rule with relation to other rules in the set of rules. Below is one recipe for success that I hope you find useful.

First note that the operational semantic defines the eval function so any work that I might do will be performed in that function

```
def eval(E : Env , e : Expr) : Expr = ???
```

Now in Scala we often want to pattern match on one of our inputs. Should we match on `E` or `e` or perhaps both? Well I would compare the conclusion of my inference rule to the operational semantic.

- Operational semantic E |- e ⇓ v
- Conclusion of EvalNeg E |- – e_1 ⇓ n'

Just looking at the inputs, the expression e in my inference rule is different (its not just 'e' it's '- e_1') while the environment E is just E.

So, I will pattern match on my input expression e

```
def eval(E : Env , e : Expr) : Expr = e match {
        case _ => ???
}
```

Now let's consider what pattern of e is interesting to me. That would be the specific e that I have '- e_1'. What is the Expr that represents '- e_1'? This has an operator in it and the operator is a unary as it only has a single operand. I need Unary (_:Uop , _:Expr). I can fill in the second blank as e1 since that represents the operand (you could actually use whatever you want, it's a variable… but I encourage you to use meaningful variable names). I need Unary (_: Uop , e1:Expr). My Uop type is either 'Neg' or 'Not'. Neg represents the '-' operator as detailed in the lab handout so I can write the following code:

```
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => ???   // Note that here I do not HAVE to state the type of Neg and e1
                                        // (although Scala does allow me to do that if I wanted to)
        case _ => ???
}
```

From here we will take a bottom up approach. What do I return? I find that in the conclusion to the right of ⇓. It says to return `n'` but I can't use that as a variable name so I'll call it np

```
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => {
                np
                // if only I were programming in Haskell, then I could call it n'….
        }
        case _ => ???
}
```

np DNE (does not exist) in scope… I have to declare it

```
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => {
                val np = ???
                np
        }
        case _ => ???
}
```

What is the value of np? It is stated in a premise of my inference rule (the second premise)

```
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => {
                val np = -toNumber(v1)
                np
        }
        case _ => ???
}
```

v1 DNE in scope I must declare it

```
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => {
                val v1 = ???
                val np = -toNumber(v1)
                np
        }
        case _ => ???
}
```

What is v1? That is declared in the first premise. E |- $e_1 \Downarrow v_1$. Wouldn't it be great if I had a function that took in an environment E and an expression e and returned a valued expression v? I DO! It's the function that I am defining… I need to recurs.

```
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => {
                val v1 = eval( E , e1 )
                val np = -toNumber(v1)
                np
        }
        case _ => ???
}
```

This is great and all but I think I can do a better job of rewriting this. Lets use ~~> to mean 1 step of rewriting our source code

```
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => {
                val v1 = eval( E , e1 )
                val np = -toNumber(v1)
                np
        }
        case _ => ???
}
~~>
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => {
                val v1 = eval( E , e1 )
                -toNumber(v1)
        }
        case _ => ???
}
~~>
def eval(E : Env , e : Expr) : Expr = e match {
        case Unary( Neg , e1 ) => -toNumber( eval( E , e1 ) )
        case _ => ???
}
```

You SHOULD start practicing writing the code in multiple lines. It should help you better learn the programming language and is useful when debugging. You should also be able to rewrite the code to a one liner quickly, although I'll never require you to write one liners as they are often bad practice and impractical. By the end of Lab 3 I expect you to be able to look at an inference rule like this one and quickly write the clean code solving the problem that it represents. Often on Moodle I might ask you about the one liner solutions as they are easier to typeset (and as previously mentioned, I'm lazy). Often in readings I will write one liners as well where I think it's reasonable that you should be able to read the one liner…

NOTE: most of the eval (E |- e ⇓ v) rules in lab 3 also apply to lab 2. If you are struggling with lab 2 you might try using these rules to complete your code.

L2: Additional Resources

- Csci3155 course notes
- Course text books
- Theory of Computation readings on context free grammars

L2: Closing Thoughts

Lab 3: Interpreter with theory

L3: Preface

L3D1

- Print out, read and annotate the lab 3 handout
- Review chapter 2 of the csci3155 course notes
- Skim chapter 3 of the csci3155 course notes
- static vs dynamic scoping

L3D1T1: Static vs dynamic scoping

A video :

- https://www.youtube.com/watch?v=6m_RLTfS72c

I was not able to find an amazing video on the topic but this one is pretty good. In lab 3 we address the issue of static vs dynamic scoping and one way that we can resolve the issues of dynamic scoping and implement static scoping.

Today's reading here is light as I do expect you to read over the csci 3155 course notes from lab2 and lab3. They provide an exceptionally detailed report of the course topics.

L3D2

- Deparsing
- Small step

L3D2T1 Deparsing

Potentially useful in understanding how the parser works, we might look at how to de-parse our expressions. Here is some setup for writing such code. Take a look. And try it yourself.

- Consider the language defined by the following grammar:

  e ::= e_1 ^ e_2 | b
  b ::= true | false

- Here is an AST that represents the language

  ```
  sealed abstract class AnAST
  case class B( b : Boolean ) extends AnAST
  case class Xor( e1 : Expr , e2 : Expr ) extends AnAST
  ```

- Let s denote the language of strings

- Operational semantic

  - $e \Downarrow_{deparse} s$
  - denotes that an expression e should deparse to a string s in 0-or-many steps
  - We will let this represent the code to De-parse an AST of the form Expr (described above) into a String

- Inference rules

  DeparseBool

  $$\frac{. \quad s = \textbf{toString}(b) \quad .}{b \Downarrow_{deparse} s}$$

  DeparseXor

  $$\frac{. \ e_1 \Downarrow_{deparse} s_1 \quad e_2 \Downarrow_{deparse} s_2 \quad s' = s1 + \text{“}\wedge\text{”} + s2 \ .}{e_1 \wedge e_2 \Downarrow_{deparse} s'}$$

- Helper function

```
def toString( v : Expr ) : Boolean = v match {
        case B(true) => “true”
        case B(false) => “false”
        case _ => ???
}
```

- Examples
  - deparse(B(true)) = “true”
  - deparse(Xor(B(true),B(false)) = “ true^false “

- L3D2T1Q1: complete the code using the stub provided below

```
def deparse(e:AnAST):String = ???
```

L3D2T2 Small Step

Note : |- denotes the ‘turnstyle’ modifier of judgment forms. (I am still trying to find a paste-able thing for this, I think the actual LaTex is something like \vDash)

Context :

- There are many judgment forms that have a well-accepted meaning.
  - “ = + “ common arithmetic
  - “⇓”evaluates in 0-or-many-steps
  - “⟶” steps in 1 step
- We can also add modifiers such as
  - * the Kleene star : 0-or-many
    - e.g. “⟶*” steps in 0-or-many-steps
  - |- : use of an environment
    - E.g. “E |- e ⇓ v” evaluates `e` in the environment `E`

- In lab 3 we will look at implementing our interpreter in 2 ways. First, we will interpret *javascripty* using a big-step operational semantic E |- e ⇓ v meaning that in the environment `E`, expression `e` evaluates to value `v` in 0-or-many steps. These rules unfortunately (as is) cause dynamic scoping

which is often not desired behavior of programming languages. And so, we need a way to implement static scoping. **We chose** to implement static scoping by creating a small step interpreter that uses substitution of values for the variables they are bound to (let me know if you want to discuss other ways to accomplish static scoping in the interpreter, we can chat outside of class). This uses the operational semantic e ⟶ e' meaning expression e steps to expression e' in exactly 1 step. We note that this operational semantic will not operate on a value i.e.

`v` ⟶ "Gremlins" (you do actually get a message about Gremlins in your code if you attempting to step on a value).

- So how then do we get a value? We have an iteration function which calls the step function as many times as needed. Here is an inference rule that describes the semantic of iterate. We will look at this in greater detail at a later date. The iterate function takes as input an expression `$e_1$` and a function that transforms expressions `$e_2$` to options for expressions `$?e_3$` and will return `$e_{1i}$` where `i` is the i^{th} output to a call to our input function representing the first instance of a None as a return value (I know… weird right? As I said we'll discuss this further at a later date).

$$\frac{\text{Some}(e_1') = (e_2 \Rightarrow ?e_3)(e_1) \qquad (e_1', e_2 \Rightarrow ?e_3) \Downarrow_{\text{iterate}} e_1^i}{(e_1, e_2 \Rightarrow ?e_3) \Downarrow_{\text{iterate}} e_1^i}$$

$$\frac{\text{None} = (e_2 \Rightarrow ?e_3)(e_1)}{(e_1, e_2 \Rightarrow ?e_3) \Downarrow_{\text{iterate}} e_1}$$

Each step rule follows the operational semantic e ⟶ e'. This relates to the step function declared in our code which begins by pattern matching on its input `e`. This `e` refers to the expression in our inference rules, in the conclusion, to the left of the judgment form. Let's dissect one of the provided step rules.

NOTE: I am about to show you one of many ways that you can write the code. You might decide to go a different rout and that is totally cool but the logic demonstrated here should still apply in your solution.

$$\text{DoNeg} \quad \frac{n' = -\text{toNumber}(v)}{-v \longrightarrow n'}$$

In code I want to express the pattern in the conclusion, left of the judgment form: '-v'. At first, I would think the following is sufficient:

```
def step(e:Expr):Expr = e match { case Unary(Neg,v) => ??? }
```

but this is not true. Recall that our operational semantics are dependent on grammars. In the grammar provided in the handout `v` symbols has a specific meaning (valued expressions) BUT scala's pattern matching is not intelligent enough to know that the use of a variable `v` denotes a value. So, I must change the code a bit to denote this value. I like to use a guard on my case as follows:

```
def step(e:Expr):Expr = e match { case Unary(Neg,v) if isValue(v) => ??? }
```

And then I can complete the body of the case statement which I will not be doing here but I encourage you to try it yourself. Perhaps look over the notes in Chapter 2 of this text for more details

Let us dissect one of our search rules.

SearchUnary

$$\frac{e \longrightarrow e'}{uop\ e \longrightarrow uop\ e'}$$

I can represent this pattern quite easily as I have a unary expression but I am not too concerned about the particular Uop being used or the subexpression being used

```
def step(e:Expr):Expr = e match { case Unary(uop, e) => ??? }
```

In the body I must return an expression `uop e'` for some item `e'` that is defined in my premise. In Scala, unlike Haskell, we cannot label any variable using an apostrophize so I'll call this variable ep, since in English I would call it e-prime but I'm too lazy to write e_prime (snake_case) or ePrime (camelCase) as my variable name.

```
case Unary(uop,e) => {
        val ep = ???
        Unary(uop,ep)
}
```

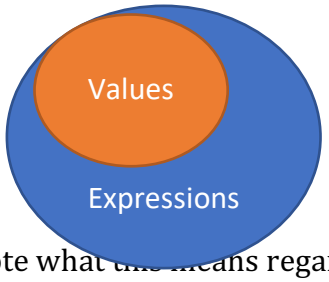
Note that ep is defined in the premise by recusing on the judgment form "⟶". Again, I will not complete the body of this case statement here, but you should try it yourself.

Now let us consider how these chunks of code relate. The following code has a bug in it (supposing that I want all of the code to be reachable):

```
// e : Expr
e match {
        case Unary(uop,e) => {
                val ep = ???
                Unary(uop,ep)
        }
        case Unary(Neg,v) if isValue(v) => ???
}
```

What is the bug? My second case is not reachable code! Values v are a subset of all possible expressions represented by variable e. Also, the value Neg is a subset of all values that could be represented by variable uop. Scala case statements are read sequentially and thus I need to be careful about the ordering of my case statements when they are describing similar patterns. In general, we will write the step function with the "Do_" rules before the "Search_" rules. Similar issues arise in ordering a few of our Search Rules.

Here's a graphic:



It is important to note what this means regarding our derivations. Consider the javascripty expression:

`!-console.log("hello")`

which would be parsed by the parser into the following Expr object:

Unary(Not,Unary(Neg,Print(S("hello"))))

Let [ ] denote an empty environment.

In the large step rules we derive:

----------EvalVal
"hello" hello is printed
---------- EvalPrint
[ ] ⊢ console.log("hello") ⇓ undefined NaN = -toNumber(Undefined)
---------- EvalNeg
[ ] ⊢ -console.log("hello") ⇓ NaN true = !toBool(NaN)
---------- EvalNot
[ ] ⊢ !-console.log("hello") ⇓ true

but in the small step interpreter, in a single step we derive:

hello is printed
---------- DoPrint
console.log("hello") ⟶ undefined
---------- SearchUnary
-console.log("hello") ⟶ -undefined
---------- SearchUnary
!-console.log("hello") ⟶ !-undefined

In general, it will take many derivations to derive the value that expression evaluates to. We are generally more interested in the fifty-foot view in that we do not particularly care about deriving each step but rather to quickly discerning the flow of evaluation. You might find it useful to note what rules are applied during each step. Below is the flow that I mentioned. Note that for each step we could* derive, prove that the step is correct:

!-console.log("hello") ⟶ /* SearchUnary, SearchUnary, DoPrint */
!-undefined ⟶ /* SearchUnary, DoNeg */
!NaN ⟶ /* DoNot */
true

L3D2T2Q1: derive each step in the second and third step in the above statement

Note that the derivation of each step will be a tree (derivations are trees) with 0-or-many "search rules" applied and exactly 1 "do rule" i.e. at each step of evaluation we only really **do** 1 thing but it might take a bit of searching to figure out what that one thing is.

L3D2 Solutions

L3D2T1Q1 Solution:

```
def deparse(e:anAST): String = e match {
        case B(_) => toString(e)
        case Xor(e1, e2) => deparse(e1) + "^" + deparse(e2)
}
```

L3D2T2Q1 Solution:

Derive each step

1. !-console.log("hello") ⟶ /* SearchUnary, SearchUnary, DoPrint */
2. !-undefined ⟶ /* SearchUnary, DoNeg */
3. !NaN ⟶ /* DoNot */
4. true

Step 1: defined in the notes

Step 2: NaN = -toNumber(undefined)
---------- DoNeg
-undefined ⟶ NaN
---------- SearchUnary
!-undefined ⟶ !NaN

Step 3: true = !toBoolean(NaN)
---------- DoNot
!NaN ⟶ true

L3D3

- Options
- Functions

In today's reading I note several code files for you to take a look at. You don't need to read and understand all of the code, but you should consider reading them at your earliest convenience to reinforce the topics here.

L3D3T1: Options

The option data type is a rather useful tool in Scala. It is often a work around a type pair that we might use to solve some problems, the tuple of Boolean and some type A i.e. (_:Boolean,_:A) where I am only interested in the second value of the tuple in the event that the first value is "true".

See L3D3_options.scala for the example of this in action on a LinkedList

Suppose the A type happens to be an Int. Suppose I want to write a function where if the input represents an interesting Int (true, _:Int) or Some(_:Int) then I want to return a Double

Using Tuples

```
def checkingTuple(x:(Boolean,Int)):Double = x match {
        case (false, _) => ??? // do something, lets call it e1
        case (true, i) => ??? // do something else that LIKELY uses variable i, let's call it e2
}
```

Using Option

```
def checkingOption(x:Option[Int]):Double = x match {
        case None => ??? // do e1 from above
        case Some(i) => ??? // do e2 from above
}
```

More generally I can use type objects. Here I declare types A and B. type A and B do not need to equal each other but all instances of A must be the same type in scope and B the same. We'll go deeper into types like in lab 4.

```
def checkingTuple[A,B](x:(Boolean,A)):B = x match {
        case (false,_) => ??? // e1
        case (true,i) => ??? // e2
}

def checkingOption[A,B](x:Option[A]):B = x match {
        case None => ??? // e1
        case Some(i) => ??? // e2
}
```

L3D3T2: Functions

A fantastic and sometimes mind-blowing concept is this idea that "functions are values". Fun fact, functions are in fact values.

Consider that in Scala the following expression prints nothing

```
def foo(x: String): Unit = println("you entered: \"" + x + "\"")
```

meanwhile the following script does print something

```
def foo(x: String): Unit = println("you entered: \"" + x + "\"")
foo("hello")
```

This is because in the first statement I do absorb the value of my function and store it to the name foo. But I do not evaluate the body of the function until the function is executed.

(you might skip this one, its rather complex, not my best example) Consider a different way to express ConstDecl from lab2.

```
sealed abstract class Expr
case class ConstDecl(ebind:Expr, ebody:Double => Expr ) extends Expr
```

In the definition of the AST bold we see a somewhat strange type in the ConstDecl class "ebody : **Double => Expr".** What is this type? ebody has a function type. It's a function that when provided a Double will yield an Expr. NOTE, ebody is not itself an Expr but rather for some variable x of type Double, ebody(x) is an Expr.

See L3D3_altConstDecl.scala for a sandbox file and solved eval function over this definition

Iterate

The iterate function in Lab 3 has yet another instance of passing a function as a value:

def iterate(e0: Expr)(next: **(Expr, Int) => Option[Expr]):** Expr

The type of an input to the iterate function is itself a function. next: **(Expr, Int) => Option[Expr]** states that if you call the function next with a tuple of an Expr and an Int then you will receive an option of an Expr.

The iterate function also demonstrates the concept of "***currying***" (Compliments of Haskell Curry). *We will talk about that later in Lab 3*. The short of it is that if I have a function f with 2 inputs x and y I can declare my function pass both of the inputs at once as in :
def f(x,y) // f of x and y

or I can curry the inputs in:
def f(x)(y) // f of x of y

Curried inputs are useful when performing partial function applications. We'll cover this deeply in lab 4. For now, please familiarize yourself with the syntax of curried inputs without worrying about what it allows. Just know that this is possible

Practicing function types:

Consider the function in Scala

def f(x:Int):Double

this is a function that takes as input and Int and returns a Double. Its type definition is:

f : Int => Double

f is not itself a double but rather a function that, when called with an input of type Int will evaluate to a Double

f(_:Int) : Double

Try it yourself: (solutions at bottom)

What is the type of the function? Solutions provided at L3D3T2Q1

1. def toNumber(v:Expr):Double
2. def eval(env:Env , e:Expr):Expr
3. def substitute(e:Expr,v:Expr,s:String):Expr
4. def iterate(e0: Expr)(next: (Expr, Int) => Option[Expr]): Expr

See L3D3_valuedFunctions.js

> You are probably wondering when is this useful? Most examples of when this is useful are quite complex but let us look at some examples in javascript. The provided file is written in valid *javascripty* so you could write an integration test on it for our interpreter if you were so inclined.

Programming Languages (PLs) vary in how easy it is to pass functions as values. PLs that make it easy to do are often more functional then those that do not. This is one of many concepts that make a language functional. Note that Scala makes currying functions a lot easier that *javascripty* does. This is one reason that we might consider Scala to be more functional. Note here I am talking about *javascripty* and not true JavaScript (which does have easy ways to curry inputs while using lambda functions).

As we have seen, JavaScript has this capability and we would like our subset *javascripty* to also have this ability. So, we introduced functions to our grammar. Most often seen in our readings as "p(x) => e1" we could also write "function p (x) = e1". If you **look in your lab 3 worksheets** you will see many examples of what the parser can and cannot handle. This also highlights the importance of the "return" keyword in JavaScript(*y*). Ultimately, we can write our functions in JavaScript(*y*) using the keyword function or by using the arrow notation lambda functions.

> COOL FACT : In true JavaScript there is a difference between these kinds of functions in that the arrow notations are sort of "lightweight" in that they do not have a "THIS". Totally out of scope, our interpreter is not implementing the object-oriented nature of JavaScript so it doesn't affect us and our interpreter but I thought a few of you reading this might be interested to know this bit of trivia. Also the lambda functions always return the value of their function bodies.

So how does this tie into Lab 3? We will go deeper into this later on but you should know:

Our AST namely Expr abstracts functions as follows

> Function(_:Option[String],_:String,_:Expr)

This represents

> Function(optionForAName, singleParameter, functionBody)

If I want to pattern match on a nameless function I write

> Function(None, _ , _)

If I want to know the name I write

> Function(Some(varForTheFunctionName) , _ , _)

Sure enough functions aren't overly useful if they are never called.

Our AST namely Expr abstracts function calls as follows

> Call(_:Expr,_:Expr)

This represents

> Call(somethingThatHopefullyEvaluatesToAFunctionType , argument)

Here it is important to note that a function declaration on its own does not do much for us, whereas a function call might do a lot. (that is actually a bit of a lie, it depends on what you hope to accomplish)

L3D3 Solutions

L3D3T2Q1:

1. toNumber: Expr => Double

2. eval: (Env, Expr) => Expr
3. substitute: (Expr, Expr, String) => Expr
4. iterate: (Expr)((Expr, Int) => Option[Expr]) => Expr

L3D4

- Iterate
- Substitute
- Call

L3D4T1: Iterate

The iterate function in lab 3 is pretty nifty. Let's look at how to complete the function.

Partially complete function declaration and body that we are provided in the lab:

```
def iterate(e0: Expr)(next: (Expr, Int) => Option[Expr]): Expr = {
  def loop(e: Expr, n: Int): Expr = ???
  loop(e0, 0)
}
```

Test case provided:

```
val one = parse("1")
"iterate" should "stop if the callback body returns None" in {
  assertResult(one) {
    iterate(one) { (_, _) => None }
  }
}
```

Here is another way we could* write the same test case:

```
val one = N(1.0)
def f [A,B](x:A, y:A):Option[B] = {
  None
}
"iterate" should "stop if the callback body returns None" in {
  assert(one === iterate(one)(f))
}
```

note the differences in how those are written. While I would like for you to become comfortable writing and reading the first form, the second form is a good stepping stone toward that goal.

If I call iterate with variables one and f I should get the value of variable one
i.e. iterate(one)(f) === one
i.e. iterate(N(1.0))((x,y)=>None) === N(1.0)

Note that f is a function and has the type (A, A) => Option[B] where A and B are arbitrary types. And the functions body "None" operates independent of the parameters x and y

If I prefer I could have written the function using arrow notation and store the function to a constant variable 'f' :

```
val one = parse("1")
val f = (_,_) => None
```

```
"iterate" should "stop if the callback body returns None" in {
  assert(one === iterate(one)(f))
}
```

The lab 3 handout tells us :
“””

Implement:

def iterate(e0: Expr)(next: (Expr, Int) => Option[Expr]): Expr

that iterates calling the callback next until next returns None. The callback next takes an Expr to transform and the iteration number, which is initially (e0,0). This function is used by the interface method iterateStep to repeatedly call step to reduce a JAVASCRIPTY expression to a value.

“””

What does that mean?

Callback: fancy name for a function that is a parameter to another function - in my experience, it is often a curried input, but it doesn’t have to be.

Iterate takes 2 parameters that are named “e0” and “next”. They are curried in that we call it as `iterate(e0)(next)` and not as `iterate(e0 , next)`. The currying can help depending on what I want to accomplish. It isn’t needed here but we should leave it in this way since the testing framework expects it as so. Besides, we need to become comfortable working with these kinds of function declarations.

The handout says that I should call “next” until it returns None.

So I should start the loop function by calling next with a valid input. Next takes as input (_:Expr , _:Int). In the scope of “loop” do I have an Expr and Int that I can use? I sure do. I could use e0 as my input or I can use e. I think e is more likely the best input to next as I am calling it in loop. In course tradition it is likely that I want to pattern match. Here we’ll pattern match on the output of this call to next…

```
def iterate(e0: Expr)(next: (Expr, Int) => Option[Expr]): Expr = {
  def loop(e: Expr, n: Int): Expr = next(e,n) match {
    case _ => ???
  }
  loop(e0, 0)
}
```

What are the patterns that I might hit in matching on this output? Well a call to next should evaluate to something of type Option[Expr] as seen our type def… next: (Expr, Int) => Option[Expr]

I am matching on an Option[Expr] object. It can either represent a failed Option[Expr] i.e. None or it will represent a successful Option[Expr] i.e. Some(_:Expr). When matching I might find it helpful to use a variable rather than saying Some(_) I’ll say Some(myVariable). Since the “next” function uses an Expr to create an Expr and I know this is somehow related to the step function, I’ll call my variable ep as a stand in for e’ or e-prime. (You can name it whatever you want)

```
def iterate(e0: Expr)(next: (Expr, Int) => Option[Expr]): Expr = {
  def loop(e: Expr, n: Int): Expr = next(e,n) match {
```

```
    case None => ???
    case Some(ep) => ???
  }
  loop(e0, 0)
}
```

From my test case :
Assert (iterate(N(1.0))((_,_)=>None) === N(1.0))

I call “next” on input N(1.0) and 0 i.e. ((_,_)=>None)(N(1.0),0) and that evaluates to None. So my test case tells me what to return in the none case. Here I have 2 variables that are equivalent to N(1.0) in scope, namely both variables e0 and e happen to represent N(1.0). But since e is a parameter to the function I am currently writing - “loop” - I think I should return the variable e.

```
def iterate(e0: Expr)(next: (Expr, Int) => Option[Expr]): Expr = {
  def loop(e: Expr, n: Int): Expr = next(e,n) match {
    case None => e
    case Some(ep) => ???
  }
  loop(e0, 0)
}
```

What about the Some(ep) case? That is a bit more complicated but in the course tradition it probably has something to do with recuring the loop function.

But we are given one other test for the iterate function

```
val one = parse("1")
it should "increment the loop counter on each iteration and use e if the
callback body returns Some(e)" in {
  assertResult(parse("--1")) {
    iterate(one) { (e: Expr, n: Int) =>
      if (n == 2) None else Some(Unary(Neg, e))
    }
  }
}
```

Or if you prefer

```
val one = N(1.0)
def g(e:Expr,n:Int) = {
  if (n == 2) {
    None
  } else {
    Some(Unary(Neg,e))
  }
}
it should "increment the loop counter on each iteration and use e if the
callback body returns Some(e)" in {
  assert(Unary(Neg,Unary(Neg,N(1.0))) === iterate(one)(g))
}
```

This says that if you take 2 steps on -(-(1)) you should get out 1.

Ultimately our goal here is to help the "iterateStep" function work based on how we write iterate.

iterateStep(e0) relies on how we write iterate should print something like this (I am not sure it is worth formalizing at this time):

```
#   Stepping…
##  0: e0
##  1: e1
……. all the steps needed until I reach a value….
##  <n>: v
v
```

To see a bit more of how iterateStep will operate you should look at an answer file to one of the integration test. Note that iterateStep is written for us… Open file
<project_path>/src/test/resources/lab3/<whatever>.ans

Pay attention to the part after it says "# Stepping…" these lines are of the form :
n_i: e_i

where n_i and e_i are are the inputs to the i^{th} call of the loop function. You should have all that you need to write the loop function now.

L3D4T2: Substitute

Let us consider the operational semantic e' = e[v/x] which expresses how to perform substitution. Let's note something weird about this operational semantic. The judgment form is =[/]. The Operation semantic takes 3 inputs to create 1 output, and here, the output is on the far left: Output = $input_1$ [$input_2$ / $input_3$]

Below is one correction I would make to the labs small step rules, it's stylistic but arguably important wrt (with respect to) substitution. While the [/] operation is closed on its values and returns an `e` it is not itself an `e` and accordingly it should be declared in a premise

Old DoDecl

```
                    ---------------------------------------
                    const x = v1 ; e2 -> e2[v1/x]
```

New DoDecl

```
                              e2' = e2[v1/x]
                    ---------------------------------------
                       const x = v1 ; e2 -> e2'
```

Note that since the rule is not using recursion on -> in it's premise this is still a leaf node in the derivation and should be Do rule.

Lets look at a few examples

1+1 = x+x[1/x]
1+y = x+y[1/x]
(y)=>{3+y}(3+1) = (y)=>{x+y}(x+1)[3/x]
(x)=>{x+1}(3) = (x)=>{x+1}(x)[3/x]

In general for e' = e[v/x] I should find each "free" instance of the variable x in the expression e and replace it with the value v

1+1 = x+x[1/x]

look at x + x . Here both variables are free instances of x. So I replace them both with the value 1. So e' is 1 + 1

This comes from taking a step on an expression like : const x = 1 ; x + x

1+y = x+y[1/x]

Look at x + y . here both x and y are free variables but I am looking for x, not y. So I replace only the x with the value 1. So e' is 1 + y

This comes from taking a step on an expression like : const x = 1 ; x + y

(y)=>{3+y}(3+3) = (y)=>{x+y}(x+x)[3/x]

This is where substitute gets a bit more interesting. Look at (y)=>{x+y}(x+x) . I have a function that takes y as a parameter and I call it with x + 1. Since the parameter is not the same as the variable I am looking for I need to perform substitution on the function body as any instance potential free instances of 'x' in the body of the. I would look at x + y (which we already covered). I should also perform substitution on the arguments as they might contain free instance of x. So e' is (y)=>{3+y}(3+3)

This comes from taking a step on an expression like : const x = 3 ; (y)=>{x+y}(x+x)

(x)=>{x+1}(3) = (x)=>{x+1}(x)[3/x]

This is where substitute gets difficult. Look at (x)=>{ x+1 }(x) . I have a function that takes x as a parameter and I call it with x. The parameter should never be replaced with the value. Since the parameter is the same as the variable I am looking for I do not perform substitution on the function body as any instance of 'x' in the body would be a bound instance of x. I should however perform substitution on the arguments as they might contain free instance of x. So e' is (x)=>{x+1}(3)

This comes from taking a step on an expression like : const x = 3 ; (x)=>{x+1}(x)

I encourage you to write inference rules over this. You can change the operational semantic if you prefer to whatever makes the most sense to you.

I have provided two of them below for the operational semantic: e' = e [v / x]

SubstitutePrint $\dfrac{e1' = e1\ [\ v\ /\ x]}{\text{console.log}(e1') = \text{console.log}(e1)\ [\ v\ /\ x\ ]}$

SubstituteBop $\dfrac{e1' = e1\ [\ v\ /\ x\ ] \qquad e2' = e2\ [\ v\ /\ x\ ]}{e1'\ \text{bop}\ e2' = e1\ \text{bop}\ e2\ [\ v\ /\ x\ ]}$

Most of the rules are quite simple. The ones about functions, constant declarations and variable uses are a bit more complicated. COOL FACT : the call site is actually quite simple. I mention this because many people overcomplicate that case.

L3D4T3: Call

Javascripty in lab 3 introduces 3 new productions of the non-terminal "e" to our grammar. (This grammar is found at the top of page 4 of the lab 3 handout. The productions are:

p(x) => e1
e1(e2)
typeerror

We will focus on 2 of these today:

p(x) => e1
e1(e2)

Functions are of the form p(x) => e1 in our grammar. This is the arrow representations of functions. This is the abstract form of the AST named Expr as Function(p : Option[String] , x : String , e1 : Expr). This expresses something interesting about the flavors of functions which I am allowed to create in the language. The p represents the option to name our functions. The x represents a parameter. The e1 represents the body of the function. All of our functions MUST have **exactly one** parameter "x" and **exactly one** function body "e1". We can name our functions if we would like but we are **not** required to.

As you are already aware, a big reason that someone might want to name a function is if we want to use recursion. This statement comes with an implication that you might not have realized… Any function that is named is potentially recursive and must be handled accordingly (whereas unnamed functions cannot be recursive).

Regardless, functions are values and are handled accordingly in our inference rules.

Now functions get really interesting when I actually call them. e1(e2) is the grammar production of call. This is instantiated in the AST named Expr as Call(e1 : Expr , e2 : Expr). Here we hope that e1 is a function, if not we will throw a type error. We will talk about type error later on.

The big step interpreter's rules for Call represent nothing particularly new to us.

The small step interpreter's rules for Call boils down to the following control flow in the case of successful function calls:
e1(e2) -->* (p(x)=>e1)(e2) -->* (p(x)=>e1)(v2) -> e1[v2/x]

Here the e1[v2/x] represents its own operational semantic e[v/x] which expresses a call to the substitute function. *(YES, I know… this contradicts some things that I claimed in a previous learning opportunity, e[v/x] is not an instance of e and should not be on the right side of a longrightarrow in the conclusion. I am glad that you are paying attention. This is quite arguably a bug in the lab. I don't have the skills to fix it in the handout. I apologize for that. There is a similar issue in EvalVar. But it is a stylistic error so I think we can live with it)*

We'll look deeper into substitute later.

L3D5

- substitution

L3D4T1: Substitution

In addition to this reading I would like to review any older material that you are still struggling with as it should help prepare you for the upcoming midterm exam. I would also like to you to skim chapter 3 of the course

notes (this should be your second exposure to them and they should make more sense than the first time you skimmed them).

NOTE: substitution is not on the exam but it is an important concept in the course. To learn this material and prepare for the exam at the same time I would encourage you to use the info in this reading to write inference rules that explain how e[v/x] transforms to some e'. You may choose any operational semantic you would like. You should then attempt to transform the inference rules to code.

Substitution is a pretty neat idea when you think about it. The moment that I can represent a variable "x" with a value "v", I look for all instances of the variable and I replace them with this value. To do this I must look at the scope in which my variable might be used, an expression "e". I must find all free instances of the variable "x" in that scope. Any reference to a bound variable "x" would denote that our variable has been shadowed and that variable would NOT be our "x", but rather some other "x".

If I had to give unique variable names this would be a much simpler task but most PL allow me to reuse / shadow variables because it would be difficult to write code otherwise. So, first let us look at a *javascripty* expression that uses unique variable names:

const y0 = "hello" ; const y1 = y0 + 2 ; const y2 = y1 + y0 ; y2

this expression is of the form : const x = v1 ; e2
x = y0
v1 = "hello"
e2 = const y1 = y0 + 2 ; const y2 = y1 + y0 ; y2

So since my variable y0 has a value "hello" being bound to it, I am ready to perform substitution in the scope where the variable might be use, in this case that is

const y1 = y0 + 2 ; const y2 = y1 + y0 ; y2

Let's look at this expression and see if it should work without any substitution

const y1 = y0 + 2 ; const y2 = y1 + y0 ; y2

it shouldn't work. There are free variables in the expression, each are named y0. So, I should replace them with my value "hello"

const y1 = "hello" + 2 ; const y2 = y1 + "hello" ; y2

Try it yourself (Solutions at L3D5T1Q1) demonstrate the following. You don't need to derive each step, but write the result of each step and note the rules applied:

const y0 = "hello" ; const y1 = y0 + 2 ; const y2 = y1 + y0 ; y2 $\longrightarrow^{5}$ "hello2hello"

Now let's look at a strange expression that is valid *javascripty* but is not actual valid JavaScript. Try to imagine that you were working with a language that allow expressions of this form

const z = 3 ; const z = z + 4 ; z

this is of the form const x = v1 ; e2

x = z
v1 = 3
e2 = const z = z + 4 ; z

Lets look at e2 and see if there are any free variables in the expression i.e. variables that do not make sense.

const z = z + 4 ; z

const z = **z** + 4 ; z // that second z is weird… How can I add z to 4 and bind it to z if I don't know what z is? This is a free instance of z where the other instances of z are not free. So when I do substitution I should get

const z = 3 + 4 ; z

Try it yourself (solutions at L3D5T1Q2) You don't need to derive each step, but write the result of each step and note the rules applied

const z = 3 ; const z = z + 4 ; z $\longrightarrow^{3}$ 7

This gets really interesting when we think about functions! Consider this *javascripty* expression

```
const x = 20;
function ( x ) {
    const y = 5 + x
    return x + y
}( x + 2 )
```

OR in it's alternate form (which the solutions use)

const x = 20 ; ((x) => { const y = 5 + x ; return x + y })(x + 2)

I won't walk through this one but you should try it for yourself the solution is at L3D5T1Q3. You don't need to derive each step, but write the result of each step and note the rules applied

const x = 20 ; ((x) => { const y = 5 + x ; return x + y })(x + 2) $\longrightarrow^{6}$ 49

L3D5 Solutions

L3D5T1Q1 Solutions to unique named expression

const y0 = "hello" ; const y1 = y0 + 2 ; const y2 = y1 + y0 ; y2 ⟶ // DoConst
const y1 = "hello" + 2 ; const y2 = y1 + "hello" ; y2 ⟶ // SearchConst, DoPlusString1
const y1 = "hello2" ; const y2 = y1 + "hello" ; y2 ⟶ // DoConst
const y2 = "hello2" + "hello" ; y2 ⟶ // SearchConst, DoPlusString1
const y2 = "hello2hello" ; y2 ⟶ // DoConst
"hello2hello"

L3D5T1Q2 Solution to easy shadowing example

const z = 3 ; const z = z + 4 ; z ⟶ // DoConst
const z = 3 + 4 ; z ⟶ // SearchConst, DoPlusNumber
const z = 7 ; z ⟶ // DoConst
7

L3D5T1Q3 Solution to advanced one

const x = 20 ; ((x) => { const y = 5 + x ; return x + y })(x + 2) ⟶ // DoConst
((x) => { const y = 5 + x ; return x + y })(20 + 2) ⟶ // SearchCall2, DoPlusNumber
((x) => { const y = 5 + x ; return x + y })(22) ⟶ // DoCall
const y = 5 + 22 ; return 22 + y ⟶ // SearchConst, DoPlusNumber
const y = 27 ; return 22 + y ⟶ // DoConst
return 22 + 27 ⟶ // DoPlusNumber
49

Note that in the first step, since the parameter of our function is "x", I do not substitute inside the body of my function. See L3D4 reading on substitution for more practice

L3D6

- Precedence
- Substitution

L3D6T1 Precedence

In lab 2 we looked at grammars – specifically context free grammars (CFG) in Backus-Naur Form (BNF) – and the concept of ambiguity in grammars. This was enough information to help us write our own Abstract Syntax Trees (AST) over some grammars. But this is not enough information to understand parsing and the output of the parser that we are using in our labs. While I (Spencer) will not test you directly on the output of the parser in Labs 1 - 5 I hope you have realized by now that understanding the parser output is rather useful when completing these labs. And if you haven't I encourage you to think about it and give this skill a chance.

Consider the *javascripty* expression :

console.log("hello" + 2) ? 4 + "20" : console.log(console.log(true + 5))

What is the form of this expression? What AST node represents this expression? I don't expect that you know this already, just keep reading. This expression is of the grammatical form "*e1 ? e2 : e3*" as console.log("hello" + 2) ? 4 + "20" : console.log(console.log(true + 5)) and accordingly our AST node is If(_:Epxr, _:Expr, _:Epxr). Why is this? How do I know? Well I (Spencer) happen to know this because I have had a LOT of practice. But you all have a lot of practice with this as well, you might just not recognize it. First let's address this JavaScript operator "?:", this is often referred to as the ternary operation because it operates on three inputs as seen in expressions of the form "*e1 ? e2 : e3*". Some languages have this operator and others don't in many ways it is similar to an "if/else" control flow statement, there are significant differences (depending on programming language) but for the purposes of this reading you can thing about expressions of the form "*e1 ? e2 : e3*" as "if (e1) e2 else e3". What other operations do we have in the original statement? Now for something that many languages have in common, do if/else statements have high precedence or low precedence? An example, does "if (true) 2 + 3 else 6" talk about the '+' operation or does it talk about an 'if/else' statement? It talks the If/Else statement. If/Else statements and Ternary operands alike have a rather low precedence in many, likely

most programming languages. So, looking at the above expression it should seem a bit more obvious that ?: is the outer operation and this will create and If(_:Epxr, _:Expr, _:Epxr) node in our AST.

NOTE that our goal is to mimic JavaScript, so, it would be wrong to think of the whole expression as anything other than a ternary as this will yield results other than what JavaScript will do. In this example, it would also create results that are not intuitive to programming. Consider this is not e1 + e2 as in console.log("hello" + 2) ? 4 + "20" : console.log(console.log(true + 5)) . Because e1 (the stuff before the '+' sign) is not a valid expression in the language. It is not any other separation of the sentence as "(" ")" are used in programming to denote enforced presidency – in every PL that I can think of anyway.

Now it would be great if I can apply this logic to the output of the current labs parser but let's back up a bit and talk about precedence on a smaller grammar than that of our lab.

Consider this variant on the lab 1 grammar :

Grammar 1

e ::= *n* | (e1) | - e1 | e1 + e2 | e1 - e2 | e1 * e2 | e1 / e2

Is this grammar ambiguous? Absolutely! If you are not convinced please review the lab 2 notes on ambiguity in grammars.

Let's look at a subset of this grammar :

Grammar 2

e ::= *n* | e1 + e2 | e1 * e2

Let us rewire this grammar to be left recursive. To do this I might try turning my right-hand operands of the binary operations into my most terminating production of what I am defining. I am defining "e" and the most terminating production of e is "*n*". Doing so would create Grammar 3.

Grammar 3

e ::= *n* | e1 + n2 | e1 * n2

Before continuing, please, take a moment to absorb this change in grammars. Note that Grammar 2 and Grammar 3 represent the exact same language but they do this in profoundly different ways.

Grammar 3 is a step in the direction of my parsing grammar - this grammar represent left associative parsing - but it doesn't change the precedence of my operators. My operators have clearly different precedence in mathematics and my parser should handle that for me.

Now to enforce precedence I must put the things with lowest precedence highest in the parse tree. Here my operators are '*' and '+'. '+' has the lowest precedence so I want this to be higher in my parse trees then '*'. I do this by creating extra rules in my grammar. Here is one of many patterns that I can use:

Grammar 4

e ::= P
P ::= T | P + T
T ::= A | T * A
A ::= *n*

My grammar now has four **grammar rules** rather than one. I'm a traditionalist and I like to write them in the flavor above, but you can condense this grammar to two rules if you were so inclined. Here I have the definition of my start symbol and non-terminal "e" as the first grammar rule. I define it as being the language of P i.e. the language of plus symbols. I define P as either the language of T or the language of P + T where P is the languages of plus expressions and T is the language of times expressions. I then define T in a similar manner. It is either A or T * A where T is the languages of times expressions and A is the language of "*atomic expressions"*, here the language of numbers. I then define A as the language of atomic expressions which in this language is known are simply the language of numbers *n*.

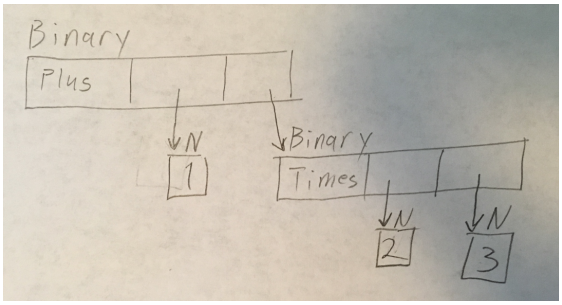
Let's look at what this means about drawing Parse Trees based on Grammar 3 vs Grammar 4.

Consider the sentence 1 + 2 * 3 that exists in both languages. My goal is to construct my Abstract Syntax Tree(AST) in the form of our expr class as

Binary(Plus,N(1),Binary(Times,N(2),N(3))).

This is represented graphically as :

AST of form Expr : Binary(Plus,N(1),Binary(Times,N(2),N(3)))

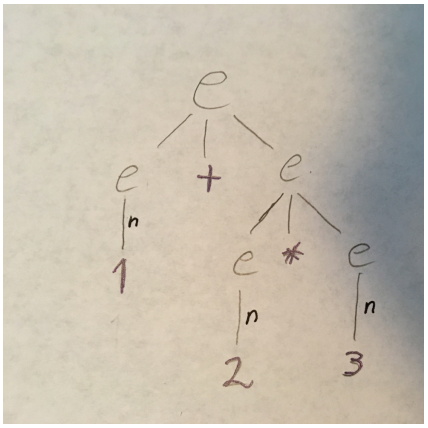


Please note that this is not a parse tree. This is a graphic representation of an abstract syntax tree.

Note that I am NOT looking for the AST of Binary(Times,Binary(Plus,N(1),N(2)),N(3)). I want the tree of Binary(Plus,N(1),Binary(Times,N(2),N(3)))

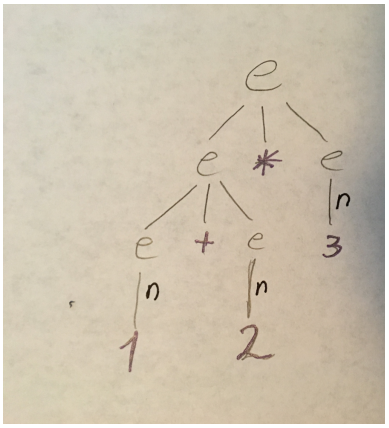
Grammar 3 ambiguously defines the sentence 1 + 2 * 3 and so I actually have 2 parse trees that I could create:

PARSE TREE 1 :
Has the AST Binary(Plus,N(1),Binary(Times,N(2),N(3))) - what I want!
Note how the tree has a subtree that encapsulates 2 valued arguments to '*'



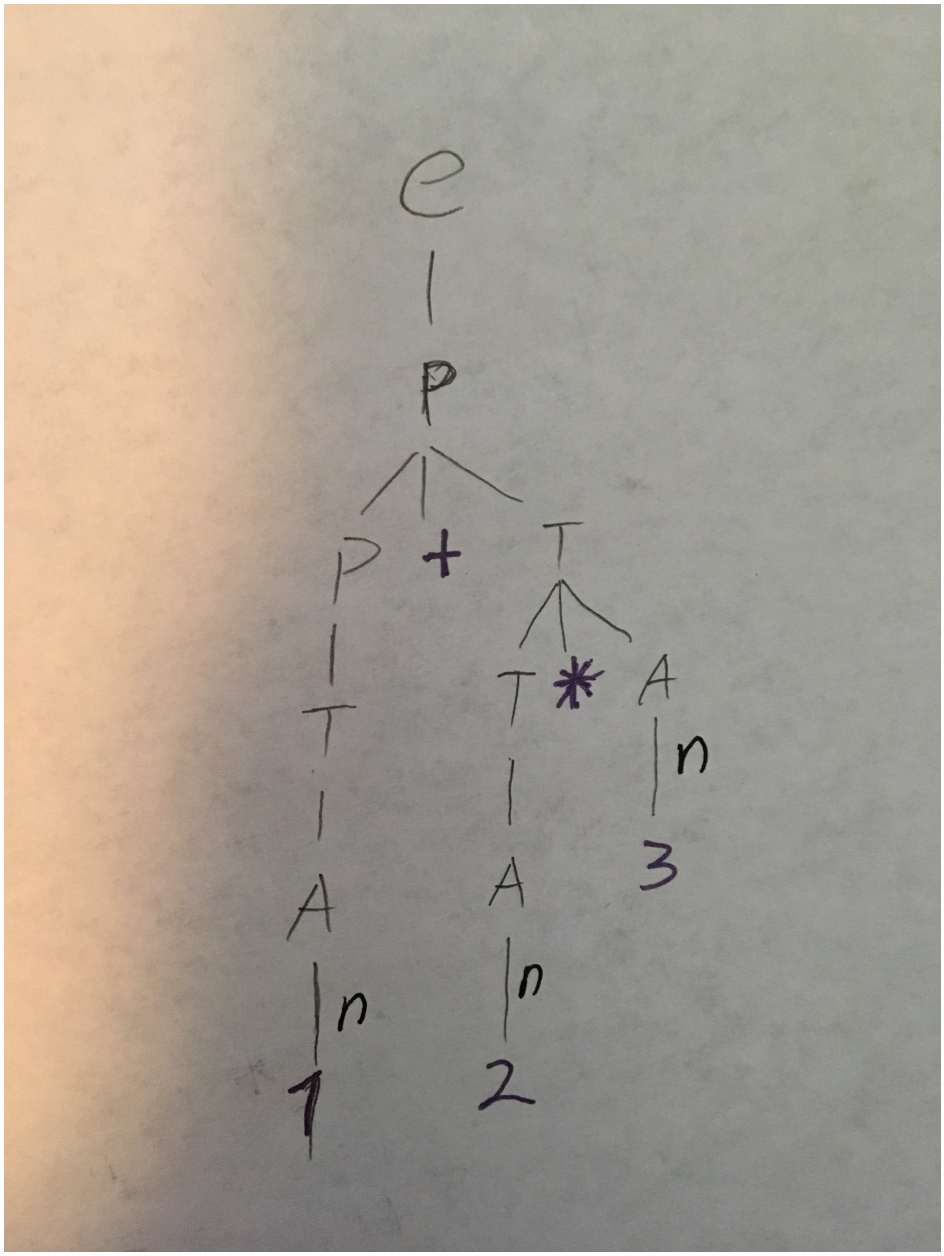
PARSE TREE 2 :
Has the AST: Binary(Times,Binary(Plus,N(1),N(2)),N(3)) - NOT what I want!

Note how the tree has a subtree that encapsulates 2 valued arguments to ‘+’



Now let us look at Grammar 4. The sentence 1 + 2 * 3 can only be parsed in one way by grammar 4 (if you are not convinced then you should try to draw another tree for it) :

PARSE TREE : Binary(Plus,N(1),Binary(Times,N(2),N(3)))
Note how the tree has a subtree that encapsulates 2 valued arguments to '*'



These parse trees carry structure with them. Grammar 3 created parse trees that represent the logic we would like as well as some logic that we do not desire. But Grammar 4 ensures that we always get the correct AST (at least at the level of detail we are looking at).

Try it yourself: L3D6T1Q1

Now that we have seen a bit about how this works, please rewrite Grammar 1 to provide the necessary precedence stated by rules of math. Note that some of the operations have the same precedence and will accordingly be defined by the same non-terminal in your language. Define all grammar rules using only left recursion about binary operations. Solutions are at L3D6T1Q1.

Grammar 1

e ::= *n* | (e1) | - e1 | e1 + e2 | e1 - e2 | e1 * e2 | e1 / e2

Looking ahead *: Please skim this…*

In lab 6 we will write our own parser (a recursive decent parser over a subset of regular expressions)! AWESOME! In writing a parser we find that the left recursion in these grammars we are writing is problematic. To the best of my knowledge, left recursion is non-deterministic i.e. either it will work fine XOR it will run for ever and it might be working fine but it might be failing. This is not desired behavior for a parser (not desired of MOST programs). So we fix this issue by writing our grammars to be right recursive but implement left associative parsing. Here is the grammar that represents one definition of a recursive decent parser for the language used in Lab 1 of our course. Note that this utilizes the concept asked in the Lab 2 write-up.

Grammar of recursive decent parser for the lab 1 language

e ::= *PlusMinus*
PlusMinus ::= TimesDiv PlusMinusPrime
PlusMinusPrime ::= ϵ | + TimesDiv PlusMinusPrime | - TimesDiv PlusMinusPrime
TimesDiv ::= Neg TimesDivPrime
TimesDivPrime ::= ϵ | * Neg TimesDivPrime | / Neg TimesDivPrime
Neg ::= Atom | - Neg
Atom ::= n | (e1)

L3D6T2 Substitution

Value substitution is a Lab 3 concept that helps us perform small step interpretation of *javascripty* with static scoping. The idea is quite simple in at its core : Once a variable is bound to a value, replace all instances of the variable with its known value. This makes it so that I should never need to ask “what is the value of this variable?” This states that if I am doing my work correctly and I find a use site of a variable during interpretation this is a bug in the *javascripty* expression I am interpreting and I must throw an error.

In the small step interpreter (step function) substitute is called from 2 places, Constant Declarations and Function Calls (NOT functions themselves). Since we are only doing substitution when the variable is bound to a known value, our search and do rules are very clear that I only call substitute when x is bound to v.

DoConst

---

const x = v1 ; e2 → e2[v1/x]

DoCall similar pattern

So, in step, for these cases, I call substitute and return the value found.

BUT substitute needs to look through the entirety of our sub-expression and conditionally look through some of it's sub-subexpressions to find the variable x. Moreover, I need to find and replace only **free instances** of x in my subexpression. A bound instance of x in the sub-expression denotes **shadowing** of the variable and thus creates a scope that will NEVER contain a use site our x but rather a different x that happens to also be named x.

Here is a hint in writing substitute. Consider this operational semantic for the substitute function:

SUB(e[v / x]) = e_{sub}.

Here I use the judgment form: *SUB([/]) =*

or rather: *SUB(input1 [input2 / input3]) = output*

This expresses that e_{sub} is a variant of e where all free instances of the variable x in expression e are replaced with the value v. Note that this should not affect any bound instances of variable x. This also, should not affect any variables other than x. Recall that x, v, and e are sets that are each defined in the grammar for the lab.

e.g. inference rule to provided code:

SubPrint $\quad$ SUB(e1[v / x]) = $e1_{sub}$

SUB(console.log(e1) [v / x]) = console.log($e1_{sub}$)

Recall that I need to be careful in substitution to only replace free instances of x in e. So I think… Where might I shadow my variable x? So, I might be tempted to ask, where are variables bound? In lab 3 language variables are bound at constant declarations and function calls. But that doesn't actually tell me where variables can be instantiated. Variables are given scope at constant declarations as well as function definitions. NOTE function definitions do not actually bind a variable to a value but they do create scopes in which I might have shadowed the variable in question.

L3D6T2Q1

Here are some templates that I encourage you to fill out for `???` which will be non-recursive premise of the rules provided. I also encourage you to think of similar rules about function definitions. Solutions at L3D6T2Q1

SubConstShaddow $\quad$??? $\quad$ SUB(e_1 [v / x_2]) = e_{1sub}

SUB(const x_1 = e_1 ; e_2 [v / x_2]) = const x_1 = e_{1sub} ; e_2

Used when stepping on const a = 2 ; const a = 3 + a ; a + a

SubConstNoShaddow $\quad$??? $\quad$ SUB(e_1 [v / x_2]) = e_{1sub} $\quad$ SUB(e_2 [v / x_2]) = e_{2sub}

SUB(const x_1 = e_1 ; e_2 [v / x_2]) = const x_1 = e_{1sub} ; e_{2sub}

Used when stepping on const a = 2 ; const b = 3 + a ; a + b

Here is a freebee that I always confused me when I worked with this in the past. Please take a moment to think about why this works before translating it to code:

SubCall $\quad \frac{SUB(\ e1[\ v\ /\ x\ ]\ ) = e1_{sub} \qquad SUB(\ e2[\ v\ /\ x\ ]\ ) = e2_{sub}}{SUB(\ e1(e2)\ [\ v\ /\ x\ ]\ ) = e1_{sub}\ (\ e2_{sub}\ )}$

L3D6 Solutions

L3D6T1Q1 Solution

Note that non-terminals can be named as you see fit. With the exception of the start symbol "e". Because of this there are multiple possible solutions. This can also be paired down to fewer grammar rules but I recommend leaving the solution in this flavor.

e ::= *PlusMinus*
PlusMinus ::= TimesDiv | TimesDiv + *PlusMinus* | TimesDiv - *PlusMinus*
TimesDiv ::= Neg| Neg * TimesDiv | Neg / TimesDiv
Neg ::= Atom | - Neg
Atom ::= n | (e1)

L3D6T2Q1 Solution

SubConstShaddow $\quad \frac{x_1 == x_2 \qquad SUB(\ e_1\ [\ v\ /\ x_2\ ]\ ) = e_{1sub}}{SUB(\text{const } x_1 = e_1\ ;\ e_2\ [\ v\ /\ x_2\ ]\ ) = \text{const } x_1 = e_{1sub}\ ;\ e_2}$

SubConstNoShaddow $\quad \frac{x_1 \mathrel{!}= x_2 \qquad SUB(\ e_1\ [\ v\ /\ x_2\ ]\ ) = e_{1sub} \qquad SUB(\ e_2\ [\ v\ /\ x_2\ ]\ ) = e_{2sub}}{SUB(\ \text{const } x_1 = e_1\ ;\ e_2\ [\ v\ /\ x_2\ ]\ ) = \text{const } x_1 = e_{1sub}\ ;\ e_{2sub}}$

L3D7

- Precedence Continued
- Intro to Higher Order Functions

L3D4T1 Precedence Continued

Things we have covered on grammars so far

- Grammar terminology – specifically that of context free grammars in Backus-Naur Form
- Grammar construction - given a description of a language build a grammar that represents the language
- Analysis of the language represented by the grammar – be able to define the language represented by a grammar
    - Comparison of a context free languages – provided two grammars be able to discern if they represent the same language
- Ambiguity of a grammar – is a grammar ambiguous

- Precedence in a grammar – be able to express contextual information of you sentences. Suggest behaviors desired of your parse trees.

Let's practice writing grammars.

Walk through

- English Description :
  - Express to me the language of Scala expressions over the binary operations '-' and '<<'. Note that '-' has a higher precedence than '<<'. Note that in Scala, these operations are only applicable to machine numbers. Let us use *n* as a meta-variable for all possible machine numbers. Your grammar should represent the desired precedence. It should be unambiguous and it should be left recursive.
- Grammar in steps:
  - I don't like doing this in one step. I like to take intermediate steps. But you can do whatever makes the most sense to you.
  - Step 1 : represent the grammar with ambiguity and non-enforced precedence
    - e ::= n | e – e | e << e
    - Step1.1 : Double check that this is correct (ask an expert if you need to)
  - Step 2 : Rewrite this grammar to have the desired recursion when possible
    - e ::= n | e – n | e << n
  - Step 3 : Now let us make this grammar left recursive and demonstrate enforced precedence.
    - Step 3.1 : think of extra non-terminals, I'll use L for left shift and M for minus and A for atomic expressions
    - Step 3.2 : Lay out the problem in the order that will enforce precedence
      - e ::= L
        L ::= <things about L vs the thing below me>
        M ::= <things about M vs the thing below me>
        A ::= <my non-operators>
    - Step 3.3 : complete the template
      - e ::= L
        L ::= M | L << M
        M ::= A | M – A
        A ::= n
      - Recall that there are many ways that I can write this. Above is my preferred way to express these grammars

L3D7T1Q1 Try it yourself

Consider the grammar of *javascripty* that will behave the same way as this subset of JavaScript would:

e ::= n | e ? e : e | e + e
where n is the language of machine numbers

Identify the operators in question.

Use a node terminal to discover the relative precedence of these operators on numbers.

Write a grammar that expresses this precedence.

L3D4T2 Intro to Higher Order Functions

Higher order functions (HOF) are super useful tools available in many languages that allow us to apply a function f to a function g. And more commonly function g is used to apply function f to each element of a collection.

We saw an example of a higher order function during lab 3. “iterate” iterate is a function g that takes as input a function f named “next”. Because of this I was able to call iterate with a specific next function to produce the concept of ‘iterateStep’. I was also able to call iterate with some other next function to create different concepts.

A note for those of you that LOVE low level programming: higher order functions are not low level, they are high level. They come with redundancies and that is fine. These things work quite well on parallel systems without taking 200 lines of code. They are meant to make the developers life easier and create large code bases that are maintainable. I encourage you to give them a chance.

Terminology

Collections

- A collection is essentially any data structure
- Examples in general
  - Tuple, Array, Dictionary, Hash Table, Trees, Linked List, Tri
- Examples in scala
  - Map is a native Scala collection that we have used. It is basically a dictionary with it’s own native operators
  - Option is a native collection Scala that only really holds one value
  - List is also a native Scala collection that is basically a linked list with it’s own native operators (we’ll look at this soon)
  - Our Expr is a collection
  - Any self defined class is a collection
- Here is a reference : https://www.scala-lang.org/api/current/scala/collection/index.html

Higher order functions: a function with at least one parameter that is a function

In this lab well look at HOFs that apply a call back to each element of a collection and can work using type objects. Natively in Scala we have fold, foldLeft, foldRight, map, forAll, forEach, and much more that can operate over elements of native Scala collection such as List and Map

We should be pretty comfortable with Linked Lists by now so let’s create higher order functions over Linked Lists. Consider this definition of a Linked List of Int named LL, a collection that we define as:

```
sealed abstract class LL
case object End extends LL
case class Node(h:Int,t:LL) extends LL
```

Now consider this function foldLeftOnLL:

```
def foldLeftOnLL(l:LL)(z:Int)(f:(Int,Int) => Int):Int = {
        case End => z
        case Node(h,t) => foldLeftOnLL(t)(f(z,h))(f)
}
```

This function scans `l` from left to right, accumulating a valued that starts with the value z using the function f that expects as input the current accumulator and the head element of the LL.

Now to test this. Consider the LL named myL that represents the Linked List 5 -> 0 -> 6 -> Nil

```
val myL = Node(5,Node(0,Node(6,End)))
```

What does this line do?:

```
foldLeftOnLL(myL)(0)((acc,h) => acc+h)
```

if you prefer this can be written as:

```
def tmp(acc,h) = acc + h
foldLeftOnLL(myL)(0)(tmp)
```

And what about this line?

```
foldLeftOnLL(myL)(1)((acc,h) => acc*h)
```

The first one will summate all the values of myL. The second one will multiply all elements of myL together.

So I could write methods for the LL class as follows:

```
def sum(l:LL):Int = foldLeftOnLL(l)(0)((acc,h) => acc+h)
def mult(l:LL):Int = foldLeftOnLL(l)(1)((acc,h) => acc*h)
```

So that now I could more easily summate myL or anything of type LL for that matter.

```
sum(myL)
```

Here I have constructed a tool that allows me to more quickly write methods for my LL class. These HOFs can be super useful when building libraries. At it's core it is meant to make your life easier… but it takes time to learn this NEW way of thinking.

Three things to look forward to that I want you to skim now

If I want to write a library for Linked Lists then my previous definition of LL is lacking. I do not always want a linked list over integers or even numbers for that matter. I want a linked list of things lets call that some type `A`. And so I define my LL class using a type object A as follows

```
sealed abstract class LL[+A]
case object End extends LL
case class Node[A](h:A,t:LL[A]) extends LL[A]
```

I also what my higher order functions to operate on abstract types. So I write foldLeftOnLL as follows:

```
def foldLeftOnLL[A,B](l:LL[A])(z:B)(f:(B,A) => B):B = l match {
    case End => z
    case Node(h,t) => foldLeftOnLL(t)(f(z,h))(f)
}
```

Now I can write even more methods then I could before.

```
def sum(l:LL[Int]):Int = foldLeftOnLL(l)(0)((acc,h) => acc+h)

def mult(l:LL[Int]):Int = foldLeftOnLL(l)(1)((acc,h) => acc*h)

def printLL(l:LL[Int]):Unit = println( foldLeftOnLL(l)("Nil")((acc,h)=>h+" -> "+acc))

// this next one should probably be written with a map HOF, but oh well

def incAllBy_n(l:LL[Int],n:Int):LL[Int] = {

        foldLeftOnLL(l)(End:LL[Int])((acc,h) => Node(h+n,acc))

}
```

to test them all at once I can observe the output of :
printLL(incAllBy_n(incAllBy_n(myL,sum(myL)),mult(myL)))

With any luck you think this is amazing and awesome! Don't be afraid of this, it is all quite doable, but it will take time and practice to understand these concepts fully.

Solutions

L3D7T1Q1 Solutions

- e ::= n | e ? e : e | e + e
- operators
  - _?_:_
  - _+_
- the ternary operation has a lower precedence. Reasoning, it's an if else statement and those tend to have a rather low precedence. Evidence to support my claim :
  - 1 + 2 ? 3 : 4 →* 3
  - (1 + 2) ? 3 : 4 →* 3
  - 1 + (2 ? 3 : 4) →* 4
- The grammar (You could name the non-terminals however you would like)
  - e ::= Plus
    Plus ::= If | Plus + If
    If ::= Atom | Atom ? If : If
    Atom ::= n
    For n representing the language of machine numbers

L3: Additional Resources

L3: Closing Thoughts

Lab 4: Higher ordered functions and interpreting types

L4: Preface

L4D1

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

- Review CH3 of csci3155-notes
- Skim CH4 of csci3155-notes, then read them.
- Print out, read and annotate the lab 4 handout
- Skim TwelveDaysOf3155 (listed near the top of moodle, just below csci3155-notes)
- Read below talk on Linked Lists

L4D1T1: Linked Lists

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

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

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

It's base case is Nil

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

Historic Fact : The Cons operator is a concept from the programming languges LISP

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

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

println works on Lists:

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

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

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

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

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

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

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

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

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

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

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

Another friendly way is:

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

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

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

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

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

fold takes 3 inputs in the following form by types:

- L:List[A].fold(z:B)(cb:(B,A) => B)
  - L - the list to fold on
  - z – the base case for the folding operation
  - cb – the callback to be applied to each element of l to accumulate an output
- in our example
  - L – the variable myList
  - z – the value 0
  - cb - the functional value (acc,h) => h + acc
- for those of you that prefer the concrete syntax, the fold method of lists looks a lot like this:
  - def fold[A,B](l:List[A])(z:B)(f:(B,A) => B):B = l match {
    case Nil => z
    case h::t => fold(t)(f(z,h))(f)
    }
  - Note: this is not the concrete syntax of the fold method for List[A]. This is a home brewed fold function for List[A] that has a similar feel to the fold method over lists.

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

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

L4D2

- List HOF
- Map
- L4D2_notes.scala

L4D2T1 List HOF

More readings on folding and Lists in Scala

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

Additional Reference on HOFs

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

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

- fold and foldLeft are basically the same thing. They scan over a list from left to right and apply a call back to each element of the list and a base value to accumulate some value
  - collection:List[A].fold (baseValueOfAccumulator: B) { callbackApplied : (B,A) => B }
  - collection:List[A].foldLeft (baseValueOfAccumulator: B) { callbackApplied : (B,A) => B }

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

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

        - Note that fold left did successfully double the values of each element of the list but it returned the list in the reverse order of what I wanted
    - Here is how we correct this inversion issue on foldLeft

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

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

L4D2T1Q1

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

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

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

- Now as I mentioned there are other HOFs for lists that are worth knowing about
- Map
    - collection:List[A].map { callbackApplied : (A) => B } // has the type List[B]
    - this will fold over the list, apply the callback to each element of the list **AND** construct a new list with those elements in it
    - This would be the ideal HOF to double all elements of the list (that we wrote above using folds)
        - val l = 1 :: 2 :: 3 :: Nil
          l.map{(h) => h* 2 } // List(2,4,6)
    - This has it's drawbacks though. The return type will always be a List of the same length as the list it is operating on.
- Foreach
    - (l : List[A]) . foreach(f : A => Unit) : Unit
    - This doesn't accumulate anything. It just applies f to each element of l.
    - Doesn't seem overly useful in a pure functional language but it could be quite useful in a PL that makes use of side effects
    - ideal for printing the elements in order

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

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

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

    - Does the list contain negative numbers

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

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

L4D2T2: Map

While 'map' is a HOF for Lists Scala also has a data structure called a Map. For clarity, from here on in the course I will do my best to lowercase the HOF `map` and capitalize the data structure `Map`, since Scala takes this syntax as well. A Map (the data structure) can be viewed as a dictionary or a hash table or an object that

relates keys to values – a set of (key,value) pairs. Each key must be unique. The values associated with the keys do not have to be unique. By extending the environment with a key that is already named this actually overwrites the value previously associated with that key. The caveat here is that, in Scala, Map[A,B] states that all of my keys must be of type A and all of my values must be of type B. (this is a discussion for another time)…

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

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

Map has a several methods including

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

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

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

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

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

L4D2T2Q1

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

e.g. keyCharToStr(Map.empty + (‘h’, 1)) == Map.empty + (“h”, 1)

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

Solutions

L4D2T1Q1 Solution

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

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

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

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

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

L4D2T2Q1 Solution

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

L4D3

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

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

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

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

- Grammar level (modified from pg 6 of Lab 4 handout) functions exist in`e` in the form:
    - p(x1:m1t1,…,xn:mntn):tann => e
- AST level
    - case class Function(p: Option[String], params: List[(String,MTyp)], tann: Option[Typ], e1: Expr) extends Expr
    - fields of the function node
        - p : Option[String]
            - represents the option to name name my functions
        - params: List[(String,MTyp)]
            - I can now easily give my functions multiple parameters
            - Each parameter is a tuple of string and MTyp
            - The string represents the name of the parameter
            - The MTyp object represents the mode and type of the parameter
                - case class MTyp(m: Mode, t: Typ)
            - so a single parameter might look like (xi , MTyp(mi , ti))
            - Note that this is a List and so it might be useful to apply some HOFs to it while implementing the interpreter.
        - tann: Option[Typ]

        - this represents the option to declare an output type for the function
        - Note that if a function is named it is supposed to have an output type (if it doesn't we should through an error)
        - The declared output type represents the expected output type which doesn't always end up being the actual output type. Our type checker will have to figure that out for us and return some kind of error if the declared type and the derived type are not the same.
    - e1: Expr
        - this is the body of the function

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

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

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

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

- I must make sure that this is the expected return type. If it is not the same value then I should through a type error on the body of my function (TypeErr_TypeFunctionAnn)
- Alternate form of TypeFunctionAnn

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

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

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

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

L4D3T2 More on HOFs

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

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

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

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

Consider the data structure of Binary Trees.

Grammar :

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

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

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

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

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

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

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

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

Rule1

---

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

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

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

Now to turn it into code….

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

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

Function definition:

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

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

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

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

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

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

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

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

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

- End is represented as Empty

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

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

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

Rule1

$$z = \text{foldRight}(\text{End})(z)(cb)$$

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

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

Now for the second rule which looks like this…

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

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

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

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

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

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

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

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

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

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

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

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

foldRight(l)(cb(d,foldRight(r)(z)(cb)))(cb)

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

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

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

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

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

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

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

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

L4D4

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

L4D4T1 *Javascripty* Functions continued

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

- TypeRecFunction
    - I have a named function and it must have a specified expected output type
    - Acceptable pattern : Function(Some(x),params,Some(t),ep)
    - I need to extend my environment to map the function name x to the functions expected type t'

        - extend(_:TEnv,_:String,_:Typ) : TEnv should be useful for this
    - The expected return type t' is specified in our last premise as
      (x1 : m1 t1 , … , xn : mn tn) => t
    - In AST Typ that is: TFunction(_:List[(String,MTyp)], _:Typ)

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

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

L4D4T2 *JavaScripty* Call

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

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

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

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

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

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

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

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

BACK TO FUNCTIONS

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

L4D4T3 Supplemental reading on Fold

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

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

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

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

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

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

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

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

L4D5

- *Javascripty* Objects
- Rename
- Supplemental MapAB

L4D5T1 *Javascripty* Objects

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

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

```
        Map(
            x       ->      Binary(Plus,N(1.0),N(1.0))    ,
```

y -> Binary(Div,N(10.0),N(2.0))
)
)

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

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

- returns Some((a,b)) on the **first** (a,b) that f finds true
- returns None if there are no (a,b) that f finds true

L4D5T2 Rename

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

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

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

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

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

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

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

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

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

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

Here is how it should evaluate (imagine it managed to pass the type checker)…

name foo = x + 2 ; (true) ? 5 ; foo → // DoDecl
(true) ? 5 ; x + 2 → // DoIfTrue
5

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

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

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

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

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

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

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

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

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

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

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

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

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

L4D5T3 Supplemental MapAB

Here are some additional notes on the Map data type

- Map[A,B]
    - This is a data type native in scala that relates Keys to Values.
        - The Keys are unique
        - The Values are not necessarily unique
    - Often viewed with type Map[K,V]… I prefer to think of it as Map[A,B]
        - K/A represents the type of all unique Keys in the map
        - V/B represents the type all Values in the map
    - This data structure can be viewed as a variant of a dictionary and hash table
    - We have seen this data type previously
        - Lab 2 & Lab 3 our data type Env is actually a Map[String, Expr]

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

L4D6

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

L4D6T1 Map access

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

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

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

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

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

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

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

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

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

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

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

Take aways:

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

L4D6T2 *javascripty* GetField

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

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

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

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

L4D7

- L4D7_getFieldCont.scala

L4: Additional Resources

L4: Closing Thoughts

Lab 5: Monads for Mutability

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.

$< M , e_1 > \rightarrow < M' , e_1' >$

$< M , uop\ e_1 > \rightarrow < M' , uop\ e_1' >$

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

Lab 6: Parsing

L6: Preface

L6D1

- BNF to EBNF
- Continuations

L6D1T1 BNF to EBNF

Consider the following grammars

G1

e ::= n | - e | e + e | e – e | e / e | e * e | (e)

G2

e ::= PM
PM ::= PM + TD | PM – TD | TD
TD ::= TD * N | TD / N | N
N ::= - N | A
A ::= n | (e)

I presume by now you understand why G1 can be translated to G2 and what the value of this grammar translation is. If not, please review L5D8 notes on grammatic precedence. I also assume that you understand the terms 'terminal', 'non-terminal', 'meta-symbol' and 'meta-variable' as it relates to CFG in BNF form. If you do not, please review notes on this topic from Labs 1 – 3 (also summarazied in L5D8 notes).

So far we have learned backus naur formed context free grammars.

Recall that BNF grammars have 3 meta-symbols:

::=	produces
|	or
ϵ	empty sentence

Now I will introduce EBNF CFG, extended backus naur formed context free grammars

Similar to BNF, we now have an additional operator in the language i.e. we have an additional meta-symbol.

::=	produces
|	or
ϵ	empty sentence
{ }	0 - or – many

Consider the language of 0-or-many 'a' symbols defined in BNF

G3

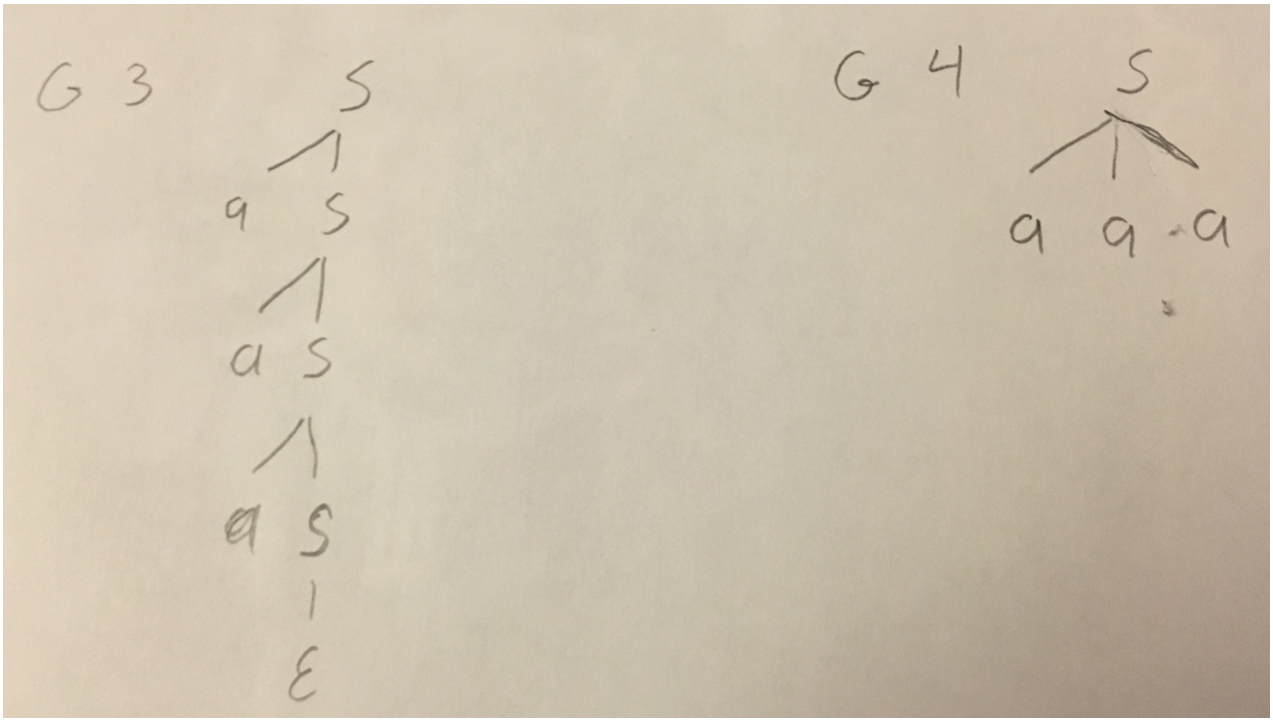
S ::= a S | ϵ

Now consider the same language defined in EBNF

G4

S ::= { a }

Parse trees for ‘aaaa’ in G3 and G4



Reconsider Grammar G2

G2

e ::= PM
PM ::= PM + TD | PM – TD | TD
TD ::= TD * N | TD / N | N
N ::= - N | A
A ::= n | (e)

This grammar is left recursive about the binary operations. Let’s see a use of the { } meta-variable of EBNF. These are great for expresses behavior in a language but I cannot translate this to code very easily (if A ::= A + B | B then the definition of A is A is A is A is A is A is…onward for infinity and finally… + B | B). So let us remove the recursion by flattening grammar in EBNF. I will explain the process for redefining PM. PM ::= PM + TD | PM – TD | TD. Here PM has 3 possible productions, two of the productions are recursive : PM + TD and PM – TD. It also has 1 non-recursive production: TD. The recursive productions hold the same form PM bop TD. So I will re-write the work as PM ::= TD { things… } where the things are defining bop TD. See the solutions below and attempt to reverse the process to better understand what is being accomplished here

e ::= PM
PM ::= TD { + TD | – TD }

```
TD ::= N { * N | / N }
N ::= { - } A
A ::= n | ( e )
```

Tomorrow we will look at using this as a stepping stone toward writing right recursive grammars (with right associative parse trees) that can be used as a reference grammar to write left associative recursive decent parsers.

L6D1T2 Continuations

In this course we have discussed the functional concept that "functions are values". With this we can make function calls using call by value that actually pass functions as arguments to another function. The input function is known as a 'callback'. The function being called is known as a higher order function.

e.g.

```
def f ( x : Int ) ( g : Int => Int ) : Int = {
        x + g ( x * x )
}
```

HERE : f is a higher order function because it accepts functions as input and g is a callback since it is a function passed as input to a function (which by definition will be a higher order function)

Now I will introduce the concept of 'continuations'. A continuation is a specific kind of callback i.e. it is a specific kind of function passed as input to an HOF. Continuations work well in solving problems that require that I traverse multiple paths as well as problems for which I can short circuit my operations based on a variety of edge cases.

In MulList.scala (provided on the main web page) we see an example of the later, a problem that can be short circuited. MulList.scala expresses 12 different ways that I could take a list of Int and return the multiplication of all the elements in the list. Here is a basic version that uses foldLeft. **To simplify the problem, we assume that the multiplication of all the elements in the empty list is equal to 1.**

```
def mulList0( l : List[Int] ):Int = {
        l . foldLeft ( 1 ) { ( acc , h ) => acc * h }          // OR if you're fancy…          ( 1 /: l ){ _*_ }
}
```

Consider the multiplication of all the elements in the List(5,0,6). What is the value? We as college students likely knew the answer to be 0 without actually bothering to multiply 5*0*6. So, maybe we can take that human intuition and translate it to code. The big trick here is to reword the problem. I do not simply want to multiply all the elements of the list together…

Consider l : List[Int]. If `l` contains any 0s, return 0, else multiply all the elements of `l` together and return the value found.

Without continuations the solution to this is pretty simple. Scan the list 1 time to see if there are any 0s. if there are, return 0 else, accumulate the multiplicative of all the values

```
def mulList1( l : List[Int] ):Int = {
```

```
    if ( l.exists { h => h == 0 } ) {                    // fancy….. l exists { _==0 }
        0
    } else {
        l . foldLeft ( 1 ) { ( acc , h ) => acc * h }     // fancy…   ( 1 /: l ){ _*_ }
    }
}
```

But suppose, that there is another constraint on the problem. What if I am only allowed to scan the list 1 time. If I find a 0 then I must return 0 and stop scanning the list. If there are no 0s then I should multiply all the values together. If the list is empty return 1. HOW THE HECK CAN I DO THAT?! That seems really challenging. The short answer is, we can introduce a continuation to our work.

The particular kind of continuation I will use here is called a 'success continuation' (SC) . The sc is a function that says given some input do some work. Here the sc expresses that if I am going to succeed in the goal of multiplying all the elements of the list together, this is how I would accomplish this work provided a single element of the list.

I can't write my work over fold because the problem specifies that I must stop scanning the list the moment I find a 0 (if there is a 0 in the list). Below is my thought process for solving the problem.

```
def mulList2( l : List[Int] ):Int = {
    ???
}
```

I can't use fold, but I need to create a looping structure. I need to have reference of some accumulator while I loop, so I cannot use mulList2 as my looping structure. Ill create another function. Ill give that function a sc to denote my accumulation

```
def mulList2( l : List[Int] ):Int = {
    def myLoop( l : List[Int] ) ( sc : Int => Int ) : Int = {
        ???
    }
    myLoop( l ) ( ??? )
}
```

for the initial call to myLoop I need to set the sc to something intelligent. I notice that sc is defined with the type Int => Int. Currently I have no integers available to me. The best function that I can write is known as the identity function. I might write it as r => r to say, given some residual, return the residual.

As for the body of myLoop, I need to loop over a List[Int], so I'll pattern match to my obvious cases

```
def mulList2( l : List[Int] ):Int = {
    def myLoop( l : List[Int] ) ( sc : Int => Int ) : Int = l match {
        case Nil => ???
        case h :: t => ???
    }
```

```
        myLoop( l ) ( r => r )
}
```

Now, let us assume that what we have so far is correct, and complete the body of the Nil case statement. Prior to now, I stated that mulList of the empty list is 1, I'll keep that value here. I could just return 1, but that doesn't feel right. Is there anything else that I can write that will result in 1? Hmm…. What are my data? I have mulList2, myLoop, l, and sc. mulList2 is just a driving function for the loop so I probably don't need to do anything with that. myLoop is the function I am in and I know it should be recursive, but I am in the base case for the data structure that this operates on so a call to myLoop probably won't help. l is the list that happens to be Nil… sc is a function. Suppose this is the initial call to myLoop, def sc (r : Int) : Int = r …. Can I get this case statement to evaluate to 1 with a statement other than '1' itself? Think about it…. Think harder… YES, absolutely sc(1) will evaluate to 1 in this case.

```
def mulList2( l : List[Int] ):Int = {
        def myLoop( l : List[Int] ) ( sc : Int => Int ) : Int = l match {
                case Nil => sc(1)
                case h :: t => ???
        }
        myLoop( l ) ( r => r )
}
```

Now before we move forward, is there any specific case that I am interested in? Consider the original problem. *I am only allowed to scan the list 1 time. If I find a 0 then I must return 0 and stop scanning the list. If there are no 0s then I should multiply all the values together.* Lets write this code to check if I have a 0. Below is an intelligent way to ask that in scala

```
def mulList2( l : List[Int] ):Int = {
        def myLoop( l : List[Int] ) ( sc : Int => Int ) : Int = l match {
                case Nil => sc(1)
                case 0 :: t => ???
                case h :: t => ???
        }
        myLoop( l ) ( r => r )
}
```

If I find a value that is 0, what should I return? 0. Now again that seems too simple. Should I instead use sc(0)? The short answer is no, I shouldn't. Recall sc, is being defined to express, if there are no 0s in the list then this is the work I'll do. But here I have found a 0. So I don't actually want to call sc. Take that at face value and try more practice problems.

```
def mulList2( l1 : List[Int] ):Int = {
        def myLoop( l2 : List[Int] ) ( sc : Int => Int ) : Int = l2 match {
                case Nil => sc(1)
                case 0 :: t => 0
                case h :: t => ???
        }
        myLoop( l1 ) ( r => r )
}
```

Note that the case body of “case 0::t” is just “0” and doesn’t actually use declared variable `t` so I might also write the case as “case 0 :: _ => 0”

Now that I have my base cases it’s time for the difficult part. The Inductive case. Let’s first lay out my data. Note the slight modifications to l above. Define, in your own words, myLoop, l2, sc, h, t as they relate to the ??? above and then compare to the following

- myLoop : a function. provided a list l2 and a success continuation sc, returns Scanning the list l2 one time, if we find a 0 then we return 0 and stop scanning l2. If there are no 0s then we return the result of multiplying all the elements of the list together.
- l2 : A list
- sc : a success continuation that says, if I never find a 0, this is the work I would do in order to multiply all the elements of the list together.
- h : an Int, the head value of l2, it does not equal 0
- t : a list of Ints, the tail of l2 after removing the head element h. This might contain 0s.

Take a moment to think about how we can bring all of this together to solve the problem myLoop.

HINT : we will need to use recursion and there must be some kind of consumption going on

HINT : I will need to write a new sc for the next iteration of myLoop to use. I should write it as a lambda in line but I can write it in a different way

HINT : I don’t want to do multiplication unless there are no 0s in the list. I don’t want something like h * myLoop(t)(sc)… that would not satisfy the conditions of our problem.

The solution is at L6D1T2Q1.

Everything we just looked at regarding mulList came with a simplification. **To simplify the problem we assumed that the multiplication of all the elements in the empty list is equal to 1.** But that doesn’t really seem correct does it. Multiply no elements together, would you find 1, or 0 or something more like None. Attempt to rewrite the problem to return Option[Int] where if the list is empty we return None, else we return Some(n) such that n denotes the multiplicative of the list.

For more practice, attempt to solve the problem on a List[Double] as well.

Then attempt this work on a Binary Tree rather then a List.

Now consider applying this work to a Binary Tree. There are many paths to explore.

Now try solving the following problem, no solution provided.
write a function named hasSubList that takes as input a l:list[Int] and n:Int. You can only scan the list once. If the list contains a contiguous sublist such that the sum of the list equals n, then return true, else, return false. Let the sum of an empty list be 0. A sublist can have any length 0 to n for t list l of length n.

HINT : you might want a success continuation that says, if the current path is a part of the solution then this is the partial sum of the path

HINT : you might also want a fail continuation that says, if the current path is wrong, then this is the result of attempting the next possible path

Solutions

L6D1T2Q1

I am only allowed to scan the list 1 time. If I find a 0 then I must return 0 and stop scanning the list. If there are no 0s then I should multiply all the values together. Lets write this code to check if I have a 0.

```
def mulList2( l1 : List[Int] ):Int = {
    def myLoop( l2 : List[Int] ) ( sc : Int => Int ) : Int = l2 match {
        case Nil => sc(1)
        case 0 :: t => 0
        case h :: t => myLoop( t ) ( acc => sc(  h * acc  ) )
    }
    myLoop( l1 ) ( r => r )
}
```

NOTEs for case h :: t => ???

- h * myLoop(t) (sc) is not correct since that potentially causes us to multiply 0 by other values.
- myLoop(t) (acc => h * sc(acc)) would also work
- acc is not some kind of reserved term. We are declaring an input to a function, we can name this parameter whatever we would like.

L6D2

- EBNF back to BNF
- Continuations Continued
- Javascripty Test

L6D2T1 EBNF back to BNF

At this point, I presume that you understand what grammars are, and why they are useful. I think that you have a decent handle on writing EBNF grammar but here are some practice problems to warm you up. The solutions are at L6D2T1Q1. Try these in EBNF as well as BNF if you need the practice

1. Describe, in English, the language defined by this grammar
    a. S ::= a a a { a }
2. Construct a grammar with start symbol B that defines 0-or-many ‘b’ symbols
3. Construct a grammar with start symbol A that defines 0-or-many ‘a’ symbols
4. Construct a grammar with start symbol S that defines 0-or-many ‘a’ symbols followed by 0-or-many-b symbols 0-or-many times. If you will the language of binary over a and b rather than 0 and 1.
5. Construct the grammar to define the language palindromes over ‘a’s and ‘b’s.

TANGENT: Another useful exercise if you are interested (but not at all needed for this class) is to think of other operators we can add to the language we use to describe grammars. BNF is very powerful, EBNF is equally as powerful but it has an extra meta-symbol (or rather a set of 2 symbols that need to go together in a particular

order) that make our job as language designers a bit easier. Are there other meta-symbols or operations that might be useful to have? Are there other ways of writing these grammars? Fun fact, BNF actually has an unnecessary meta-symbol, the |, the pipe, the or, the vertical bar, whatever you call it is not super necessary but has an advantage.

BNF S ::= a | b | c S c

Other form of CFG that I forget the name of

S → a
S → b
S → c S c

TANGENT CONTINUED: Again, if this interests you I highly recommend theory of computation or picking up a book on the matter. Many of them are rather dense with mathematic notation but they tend to be filled with fun jokes and interesting logic puzzles. It WILL teach you new ways to solve problems and new ways to think. It usually teaches you the true nature of recursion. We've seen a bit of recursion's nature this class when discovering if a grammar is ambiguously defined. These text books tend to have great visual representations of the topic.

ONE LAST POINT TO THE TANGENT : Theory of Computation is typically taught by research professors from the PLV (programming language and verification) lab. They are all really brilliant people with neat research going on. In my experience, they are also all incredibly talented and engaging instructors/ guides/ lecturers/ professors/ teachers…

BACK TO THE TOPIC…

So, EBNF is great and all. I can use it to quickly construct non-recursive grammars that denote qualities I want in how the language should be interpreted. We use them in this course to create grammars that demonstrate precedence without using left recursion about binary operations. BUT these grammars are not representative of what I would code in a parser.

Super ambiguous grammar – shows all that can be made in the language – great for constructing AST/or any data structure really – but states nothing about precedence or associativity

e ::= n | - e |e + e | e – e | e * e | e / e | (e)

BNF grammar with left recursion when possible – can show precedence in the grammar – left recursion is problematic to code (I'm not sure that it is impossible to code, but coding it often creates infinite loops, it can make our process slow and in some cases the work becomes non-deterministic)

e ::= PlusMinus
PlusMinus ::= PlusMinus + TimesDiv | PlusMinus – TimesDiv | TimesDiv
TimesDiv ::= TimesDiv * Neg | TimesDiv / Neg | Neg
Neg ::= - Neg | Atom
Atom ::= n | (e)

EBNF grammars without left recursion – resolve the issue of left recursion – flatten our work a bit – can be useful for coding if you know how to use them (but not semantically correct) - we use this as a great intermediate to the final step in building the desired grammar (helps us avoid errors)

e ::= PlusMinus
PlusMinus ::= TimesDiv { + TimesDiv | – TimesDiv }
TimesDiv ::= Neg { * Neg | / Neg }
Neg ::= - Neg | Atom
Atom ::= n | (e)

NOTE: PlusMinus and TimesDiv are re-writen in the same way.

Condition
- the grammar rule has recursive productions and non-recursive productions
- the recursive productions have the same form Recur bop Don'tRecur

Action
- take the non-recursive production of the grammar rule and use that to replace 'Recur'
- wrap all of the 'bop Don'tRecur' in '{' '}' and separate them by '|'

NOTE : Neg is not rewritten. We can rewrite it, but we don't have to since the goal here is to remove left recursion. Still, you might find it useful to re-write to EBNF for practice. Try that on your own and see solutions at bottom of this document. It follows the exact same logic as our binary cases here

Now here is where it gets interest. How do I take that grammar and re-write it back to BNF without left recursion, still unambiguous, still with the same precedence, and in a manner such that I can easily use the grammar as a guide to write a recursive decent parser with left associative parsing despite the grammar itself being right recursive (has right associative parse trees)

Consider this snippet of brilliant provided back in the write-up of lab 1 that describes the language of binary operations applied to operands in 2 ways.

e ::= e operator operand | operand

e ::= operand es
es ::= operator operand es | ϵ

Now let us write in the intermediate step that was not provided

e ::= e operator operand | operand

e ::= operand { operator operand }

e ::= operand es
es ::= operator operand es | ϵ

We have discussed how to transform from the first grammar to the second grammar, and with a little creative thinking we can assure ourselves that it works. What about the logic for transforming the second grammar to the third grammar?

Think about it…

Think a little more….

Okay, so here is how I see it. I had in EBNF Thing { Stuff } meaning, Thing happens once followed by stuff 0-or-many times. So what if I wrote it in BNF as Thing Bleh, where Bleh is an additional non-terminal in the language that I need to write to express { Stuff }, or rather 0-or-many instances of stuff. I have seen how to do that when Stuff is small. The logic is the same when Stuff is larger.

RECALL EBNF S ::= { a } BNF-right-recursive S ::= ϵ | a S

So EBNF S ::= b { c } BNF S ::= b A A ::= ϵ | c A

Or, If you prefer

EBNF S ::= Thing { Stuff }
BNF S ::= Thing Bleh
Bleh ::= ϵ | Stuff Bleh

Take a moment to absorb all of that

L6D2T1Q2
Rewrite this grammar to BNF in the manor descirebed above "S ::= { a | b }"

L6D2T1Q3
Attempt to re-write our grammar for a 5 function calculator into a BNF grammar of the form shown above.

L6D2T2 Continuations Cont.

Continuations, what are the good for? Write both implementations. Run both of them. See which one is faster from the human perspective. Google how to benchmark these and find out which one is actually faster. Try it in different langauges.

This section will discuss a particularly challenging problem that I don't expect you to know how to solve yet. I think that walking through the solution might help you find and solve problems that interest you using the technique of programming with continuations.

Complete a function named hasSubList that takes as input a list[Int] `l` and an Int `n`. **You can only scan the list once.** If the list contains a contiguous sub-list such that the sum of the list equals n, then return true, else, return false. Let the sum of an empty list be 0. A sublist can have any length 0 to n for the list `l` of length n.

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean =  ???
```

One of my conditions, **You can only scan the list once**, is what makes this question particularly challenging. In practice, I might consider relaxing this constraint, solving a similar problem and then re-adding this constraint. You might find that to be helpful. Please take a moment to solve this problem without that constraint. Implement your solution in Scala. I have provided 1 solution at L6D2T2Q1.

There are many way's in which I can answer this question, some of which are more intelligent then the solution we will go through here. But I want to walk through the solution to this that uses continuations to solve the problem

Suppose I want to do this by scanning my list only once. I need both a success continuation and a fail continuation:

> As I scan the list, I might find a sub-list that has the sum n, in which case, I do not need to scan the rest of the list – a success continuation will help me with this. If the current work is a part of the solution, here is how you would find the solution.
>
> There are many paths through the list that I must explore. For a list of length 'm' s.t. m > 2, there are (m+1)*2 + 1 paths to explore. A fail continuation will help me in this…. Provided that the current path fails this is the work that I should do next.

So I set up the problem with an inner loop to recur on `l` with some sc and fc that might also change as I recurs.

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = ???
    myLoop( ??? )( ??? )( ??? )
}
```

Before I work on myLoop, I want to attempt to find the inputs to the initial call to myLoop.

- Initial value of l:List[Int] for myLoop
  - I could just write this to be Nil, since that is the base value for lists
  - I also have a data value l:List[Int] passed to hasSubList which is a list of variable legth
  - My gut tells me `l` should use the List[Int] passed to hasSubList.
  - I might be wrong, so later, if I can't get it to work I might change this
- sc: Int => Int
  - I need to construct a function from Int to Int. For simplicity I should write this as an anonomous lambda function (define it inline using the '=>' notation)
  - My function doesn't change types… What is the easiest function that says given something of a type return something of the same type?
  - The Identity function!
  - I'll write it as ' r => r ' for, given a residual 'r' return '=>' the residual 'r'
  - Again,I might be wrong, so later, if I can't get it to work I might change this
    - I could probably also implement this to do something with our data 'n'…
- fc : () => Boolean
  - I need another lambda that says given nothing return a Boolean
  - I need to return a Boolean. I have no data of type Boolean at the moment and the fc doesn't take any input so here I have 2 options, I can assume innocents or assume guilt.
    - Innocent : () => true
    - Guilt : () => false
  - Think about the base case of the problem. If the list is empty what do I know what to return? Well no… if l = Nil and n = 0 then I return true, else I return false.
  - Having said that, I usually return false... so lets assume guilt
  - () => false
  - Again, I might be wrong, so later, if I can't get it to work I might change this

Bringing that into the stub above I find:

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
```

```
        def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = ???
        myLoop( l )( r => r )( () => false )
}
```

now what about the body of myLoop. What data can I work with? I have a list. I know that it is often useful to check if my list is empty or not…

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
        def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
                case Nil => ???
                case h :: t => ???
        }
        myLoop( l )( r => r )( () => false )
}
```

Lets start with the Nil case since that is probably easier. If the list is empty what do I do? Well, as stated earlier "*if l = Nil and n = 0 then I return true, else I return false*" directly coding this I find :

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
        def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
                case Nil =>  n == 0     ||   false
                case h :: t => ???
        }
        myLoop( l )( r => r )( () => false )
}
```

And yes, I could just write n == 0… but that above stub is more helpful. The stub is wrong… Think about why it might not work.

Think about this, I bothered to set up continuations and I am not using that data. Can I re-write the body of case Nil => ??? the use sc and fc and still state the same logic of "*if l = Nil and n = 0 then I return true, else I return false*"?

I can, it looks like this :

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
        def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
                case Nil =>  n == sc( 0 )    ||    fc()
                case h :: t => ???
        }
        myLoop( l )( r => r )( () => false )
}
```

BONUS: in addition to solving the base case of the empty list this also solves my inductive cases for reaching the end of a non-empty list! WOO!!!

Now what about the case h::t => ???

I have found an Int `h` that I might want to use. Potentially h denotes the end of the sublist s.t. the sum of the sublist is equal to n. Given the data, how would I ask the question of whether I am finished?

Look back near the top of the document consider the reason why I have constructed a success continuation and a fail continuation. Given the data, how would I ask the question of whether I am finished?

Consider this slight change for why I want a success continuation: As I scan the list, I might find a sub-list ending in some value h such that the sum of the sub-list is equal to n, in which case, I do not need to scan the rest of the list. Given the data, how would I ask the question of whether I am finished?

Absolutely, something like n == sc(h). sc of h should add h to some partially accumulated value

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil =>  n == sc( 0 )    ||    fc()
        case h :: t => n == sc( h )
    }
    myLoop( l )( r => r )( () => false )
}
```

Am I done? Does this work for all possible inputs to hasSubList? Of course not, why would it. All I've done is return things for empty lists and return things based on the first element of non-empty lists. I still need to exploit some natural recursion.

For lists, matched to h::t, natural recursion involves recusing the my function on t.

I might follow the same pattern as my Nil case and hope that works, i.e. I might need to say ||

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil =>  n == sc( 0 )    ||    fc()
        case h :: t => n == sc( h )  || myLoop( t )( ??? )( ??? )
    }
    myLoop( l )( r => r )( () => false )
}
```

so what should the continuations be?

- sc
    - as mentioned earlier this holds a partial sum, it's a function that says if I know the rest of the sum, here is what I would do with it
- fc
    - as mentioned earlier this states, if the current path doen't work then what should I do next?

- sc
    - it's a function from Int => Int
    - Let's name our input to represent a partial sum. I like the term 'acc'
    - So rather than ??? I can say acc => ??? and I have a little less work to do
    - As I look over the tail, what should happen?
        - acc
        - sc(acc)
        - h+acc
        - sc(h+acc)
- fc
    - it's a function from () => Boolean
    - () => false, () => true
    - something more complex?

- Sc : To use all the data I think acc => sc(h+acc) is my best bet
- Fc : To express that there is another path to try I think that () => myLoop(t) (???) (???) is the route I need to go

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
        def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
                case Nil =>  n == sc( 0 )    ||    fc()
                case h :: t => n == sc( h )  || myLoop( t )( acc => sc(acc+h) ){
                        () => myLoop(t)( ??? )( ??? )
                }
        }
        myLoop( l )( r => r )( () => false )
}
```

Again I need to define sc and fc for a call to myLoop. In solving the problem it is same to assume, that here, I am defining that the tail might be the start of the valid sublist. Do your best to solve this. See the solution at L6D2T2Q2. Before looking over the solution, test your work. A few tests are provided on the next page.

And finally, how can I check if this works?

I recommend painstakingly walking through one simple true case, and simple false case, then one complex case of your choosing.

Defined a few tests…

```
def tester(l:List[Int],n:Int,b:Boolean):Unit = {
        assert( hasSubList( l , n ) === b )
}

tester(List(1,2),1,true)
tester(List(1,2,3),0,true)
tester(List(1,2,3),1,true)
tester(List(1,2,3),2,true)
tester(List(1,2,3),3,true)
tester(List(1,2,3),4,false)
```

```
tester(List(1,2,3),5,true)
tester(List(1,2,3),6,true)
tester(List(1,2,3),7,false)
tester(List(1,2,3,4,5),9,true)
tester(List(1,2,3,4,5),8,false)
```

Do any of these tests seem wrong to you? (if so then you might not understand the problem fully)

Does our implementation work? (it should pass all of these tests)

Can you think of an edge case that will break our implementation?

SOLUTION

WITHOUT the constraint **You can only scan the list once.**

L6D2T3 Javascripty Test

If you are not already familiar with what regular expressions are, I highly recommend watching a YouTube video on the topic prior to this reading as in this reading I presume you all have a basic understanding of regular expressions.

NOTE: regular expressions is a white space sensitive language. Here I am ignoring this quality of regular expressions to make it a bit easier to view…

In lab 6 we are asked to implement a function named 'test' provided the following function definition:

```
def test(re: RegExpr, chars: List[Char])(sc: List[Char] => Boolean): Boolean
```

The goal is to return a literal true or false expressing whether chars exists in the language defined by re.

This all operates on a call to 'retest'. So we'll need to implement that. Make a guess at how to complete retest and refine the guess as you work on test. Retest is partially defined for us as :

```
def retest(re: RegExpr, s: String): Boolean = test(re, s.toList) { chars => ??? }
```

hint… I am converting my Sting to a List of Characters, the continuation takes as input a list of characters and should do work on that. I am writing a recursive function over a list of characters. Natural recursion has properties of consumption. This will inform the base case of sc in test, i.e. the sc applied to test when implementing rename.

Let us begin this discussion with *javascripty* literals (you can test these using node…)

/^ ! $/.test("")	false, the empty string "" does not exists (DNE) in the language of /^ ! $/

/^ ! $/.test(“a”)	false, the string “a” (DNE) in the language of /^ ! $/ because no strings exist in this language
/^ # $/.test(“”)	true, the empty string “” exists in the language of /^ # $/
/^ # $/.test(“a”)	false, the string “a” DNE in the language of /^ # $/ because only the empty string exists in that langauge
/^ a $/.test(“a”)	true, the string a exists in the language of /^ a $/
/^ a $/.test(“b”)	false, the string b does not exists (DNE) in the language of /^ a $/
/^ . $/.test(“a”)	true, the string a exists in the language of /^ . $/
/^ . $/.test(“b”)	true, the string b also exists in the language /^ . $/ because this is the language of all strings of length 1
/^ . $/.test(“ab”)	false, the string “ab” DNE in the language of /^ . $/ because it is a string of length 2

There are many others but I’ll let you figure out how those work on your own.

Let us write assertions that should hold true for each of these. I recommend writing the assertions to work off of our retest function. I recommend taking your best guess at how expressions parse and double checking that work, following the information provided to you in the lab worksheet.

```
//      /^ ! $/.test(“”)  should evaluate to false as stated above
val     re:RegExp       =       RNoString     // ! defined on page 5 of the handout
val     s:String        =       “”            // “” literally the string in test
val     b:Boolean       =       false
assert( retest(re,s) === b )
```

alternatively if I am not willing to guess and double check the ! is the RegExpr RNoString
I can import the reference parser and define re as :

```
val re:RegExp       =       RegExprParser.parse("!")
```

To go a step farther, when I am ready, I can write integration tests by using the parser we are implementing…

```
val RE( re:RegExp ):Expr      =       RegExprParser.parse("!")
```

Try writing the others on your own. I have provided a test script at the end of this topic.

Be careful when coding… we need to use the success continuation wisely example that informs how we write the REmptyString case :

/^ # $/.test(“a”)	false, the string “a” DNE in the language of /^ # $/ because only the

	empty string exists in that language
/^ #. $/.test(“a”)	true, the string “a” exists in the language of /^ #. $/ because this is the language of empty strings followed by the language of strings of length 1 i.e. this is the language of all strings of length 1.

A few edge cases to be concerned with, worth writing tests over are :

/^ !* $/ /^ #* $/ /^ ! | re $/ /^ re** $/

Let re be any regular expression

Semantically, these tests are NOT defined super well as they might be interpreted to be tests for iterateStep about the Call(GetField(e1,”test”),List(e2)) node but… hey, I never claimed to be perfect

Execute using MyTestSpecRunner.

```
class MyTestSpec(lab6: Lab6Like) extends FlatSpec {
  import Lab6Harness._
  import lab6._

  "  /^!$/.test(str_i)" should "return false" in{
    val re = RNoString
    val str_1 = ""
    val str_2 = "a"
    assert(!retest(re,str_1))
    assert(!retest(re,str_2))
  }
  " /^#$/.test( str_i ) " should "return true" in {
    val re = REmptyString
    val str_1 = ""
    assert(retest(re,str_1))
  }
  it should "return false" in {
    val re = REmptyString
    val str_1 = "a"
    assert(!retest(re,str_1))
  }
  "/^a$/.test( str_i )" should "return true" in {
    val re = RSingle('a')
    val str_1 = "a"
    assert(retest(re,str_1))
  }
  it should "return false" in {
    val re = RSingle('a')
    val str_1 = "b"
    val str_2 = ""
    assert(!retest(re,str_1))
    assert(!retest(re,str_2))
  }
  " /^.$/.test( str_i ) " should "return true" in {
    val re = RAnyChar
    val str_1 = "a"
    val str_2 = "b"
    assert(retest(re,str_1))
    assert(retest(re,str_2))
  }
  it should "return false" in {
    val re = RAnyChar
    val str_1 = ""
    val str_2 = "ab"
    assert(!retest(re,str_1))
    assert(!retest(re,str_2))
```

```
  }
}

class MyTestSpecRunner extends MyTestSpec(jsy.student.Lab6)
```

Solutions

L6D2T1Q1

1. Describe, in English, the language defined by this grammar
    a. S ::= a a a { a }
    b. This is the language of 3 or many a's. I have aaa followed by {a} and I know that {a} in EBNF applys the meta-symbols '{' '}' to something that needs to be repeated 0-or-many times, so in this case, that means 0 or many a's.
2. Construct a grammar with start symbol B that defines 0-or-many 'b' symbols
    a. B ::= { b }
3. Construct a grammar with start symbol A that defines 0-or-many 'a' symbols
    a. A ::= { a }
4. Construct a grammar with start symbol S that defines 0-or-many 'a' symbols followed by 0-or-many-b symbols 0-or-many times. If you will the language of binary over a and b rather than 0 and 1.
    a. Using 3 and 4
        i. S ::= { A B }
    b. From scratch
        i. S ::= { { a } { b } }
    c. In regular BNF (which is technically also EBNF)
        i. S ::= ϵ | aS | bS
        ii. NOTE: how I need the ϵ to create the concept of 0 characters, whereas the {} take care of that for me
5. Construct the grammar to define the language palindromes over 'a's and 'b's.
    a. This was sort of a trick question. EBNF's extra operator or meta-symbol if you will '{ }' allows me to create the concept of 0-or-many but that is not need or useful in constructing the language of palindromes.
    b. Having said that, technically all BNF grammars are written in EBNF as well since BNF is a sub-set of EBNF. On an exam or quiz I will be clear about what is expected.
    c. S ::= ϵ | a | b | a S a | b S b
    d. NOTE: this still uses the epsilon symbol, why?

L6D2T1Q2

EBNF – BNF(right-recursive) **EBNF** S ::= { a | b } **BNF** S ::= ϵ | a S | b S

L6D2T1Q3

Grammar re-writing process of constructing the reference grammar for the lab 1 parser, also representative logic of a 5 function calculator that has left associative parsing and precedence without () required

e	::=	n | - e |e + e | e – e | e * e | e / e | (e)
e	::=	PlusMinus
PlusMinus	::=	PlusMinus + TimesDiv | PlusMinus – TimesDiv | TimesDiv
TimesDiv	::=	TimesDiv * Neg | TimesDiv / Neg | Neg
Neg	::=	- Neg | Atom

```
Atom        ::=   n | ( e )

e           ::=   PlusMinus
PlusMinus   ::=   TimesDiv { + TimesDiv | – TimesDiv  }
TimesDiv    ::=   Neg { * Neg | / Neg }
Neg         ::=   - Neg | Atom
Atom        ::=   n | ( e )

e           ::=   PlusMinus
PlusMinus   ::=   TimesDiv PlusMinusS
PlusMinusS  ::=   ϵ | + TimesDiv PlusMinusS | - TimesDiv PlusMinusS
TimesDiv    ::=   Neg TimesDivS
TimesDivS   ::=   ϵ | * Neg TimesDivS | / Neg TimesDivS
Neg         ::=   - Neg | Atom
Atom        ::=   n | ( e )
```

L6D2T2Q1 iterate on the list as many times as you'd like

NOT ACTUALLY CORRECT… but seems reasonable

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l:List[Int] , acc:Int ) : Boolean = l match {
        case Nil => acc==n
        case h::t => h+acc==n || myLoop(t,acc+h) || myLoop(t,acc)
    }
    myLoop(l,0)
}
```

CORRECT

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myInnerLoop ( l:List[Int] , acc:Int ) : Boolean = l match {
        case Nil => acc == n
        case h :: t => h+acc == n | myInnerLoop(t,acc+h)
    }
    def myOuterLoop ( l:List[Int] ):Boolean = l match {
        case Nil => myInnerLoop(l,0)
        case _::t => myInnerLoop(l,0) || myOuterLoop(t)
    }
    myOuterLoop(l)
}
```

L6D2T2Q2 iterate on the list only once

With continuations…

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil => sc(0)==n || fc()
        case h :: t => {
            sc(h)==n || myLoop(t)( acc => sc(acc+h) )( () => myLoop(t)(acc => acc )(fc) )
        }
```

```
        }
        myLoop(l)( r => r )(()=>false)
    }
```

L6D3

- Continutation practice: Depth first search
- Grammars practice
- Test continued
- Beginning the parser

L6D3T1 Continuation practice: DepthFirstSearch (DFS)

DFS with continuations over a tree

Review documentation of dfs provided in the lab handout.

In performing dfs on a tree for some value, I could use continuations. Performing dfs on a tree for some function applied to an int accumulated over time is a more advanced problem that can follow the same logic of continuations and can actually be used to solve the original problem.

Our task is to complete the following function:

```
def dfs[A](t: Tree)(f: Int => Boolean)(sc: List[Int] => A)(fc: () => A): A = ???
```

t is the Binary tree I must scan from left-to-right **pre-order**

f is a function I must apply to all my data in the tree

sc – there are succeed early cases in this function. The success continuations state: if I knew the partial list that represents a part of the actual solution, this is the solution I will return

fc- in traversing a tree there are many paths to explore. The fc says, if the path you were on was not the correct path, this is what I want you to do next

You might start by writing the simple version of dfs on a tree that doesn't use continutations first, and learning from that.

Then try write the simple version of dfs on a tree with continuations.

Then attempt this actual problem (without and then with continuations).

Or, you can just take the challenge head on.

I highly recommend looking at the types of the data in the problem and using that to build your solution. There are only a few natural solutions from the types.

Recall that in recursion we want to write our base case, our case of interest and then our natural recursion case. Here the natural recursion case is the interesting part. See yesterday's notes on programming with continuations for examples of how to write natural recursion with success and fail continuations for examples of the structure.

For a tree that structure of the recursion case looks a bit like :

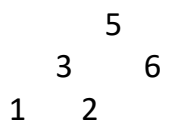
```
nameOfRecursiveFunction
        ( naturalRecursionOfDataStructurePart1 )
        ( newSuccessContinuation)
        ( newFailContinuation )
```

OR Rather

```
nameOfRecursiveFunction
        ( naturalRecursionOfDataStructurePart1 )
        ( newSuccessConintuation )
        ( () =>  nameOfRecursiveFunction
                        ( naturalRecursionOfDataStructurePart2 )
                        ( differentNewSuccessConintuation)
                        ( fc ) // provided from current call
        )
```

There is a test provided for you in the lab. I recommend reviewing it. There is one perhaps strange behavior here, that the path we accumulate is actually in the reverse order. Consider the tree:

Node(Node(Node(Empty,1,Empty),3,Node(Empty,2,Empty)),5,Node(Empty,6,Empty))



If 2 is in the tree show me the path from root to the first found instance of 2.

I would hope I can write the test with ease.

dfs(t)(d => d == 2)(path => Some(path))(() => None)

but this isn't entirely true for the example tree this returns Some(List(2,3,5)). My test might need to read

```
dfs(t)( d => d == 2 )( path => Some(path) )( () => None ) match {
        case Some(rpath) => path.foldLeft(Nil){(acc,h)=>h::ac}
        case None => None
}
```

L6D3T2 Grammars practice

Review the following re-writing of our lab 1 gramar

e ::= n | (e) | e+e | e-e |e*e | e/e| -e
n is a number
uses arithmetic precedence

e ::= A
A ::= B | A+B | A-B
B ::= C | B*C | B/C
C ::= D | -C
D ::= n | (e)

e ::= A
A ::= B { +B | -B }
B ::= C { *C | /C }
C ::= { - } D or C ::= D | -C is also allowed
D ::= n | (e)

e ::= A
A ::= B As
As ::= ϵ | + B As | - B As
B ::= C Bs
Bs ::= ϵ | * C Bs | / C Bs
C ::= Cs D or C ::= D | -C is also allowed
Cs ::= ϵ | - Cs
D ::= n | (e)

Repeat the process over the following grammar that describes regular expressions
re ::= ! | # | . | c | (re) | rere | re'|'re | re* | re+ | re? | re&re | ~re
c is some specific character

note that one of the | has quotes, on it, I am stating that this is a | in the object language regular expressions and not the meta-symbol of BNF

Use the following non-terminals while writing the first and second step :
re, union, intersect, concat, not, start, atom

When writing the 3rd step, add an ‘s’ to the end of the some non-terminals when creating new non-terminals

The precedence are defined on page 7 of the handout (along with quite a bit of valuable information)

We provide the expected definition of union in the EBNF grammar and the expected definition of both union and unions in the final grammar on page 8 of the handout.

L6D3T3 Test cont.

Note that in regex, white space does matter. Here I am acting as though it doesn’t for the sake of cleanliness.

Warm up, why is /^ #* $/ an edge case of the javascript test method of RegExpr?

Now, consider the nature of the Kleene-star and expression that use this of the form ‘re*’

‘re_1*’ says the pattern re happens 0-or-many times
‘re_1+’ says the pattern re happens 1-or-many times
‘re_1re_2’ says the pattern re_1 is immediately followed by re_2

Given the information above, can you define re_1+ in terms of the other operations?

If I told you it is possible both in math and in application, does that change your answer?

re_1re_1* or if you prefer re_1re_2 where re_2 is re_1*

Now if I say
‘re_1+’ is equivalent to re_1re_1*
‘#’ means that empty string, or that a pattern happened 0 times
‘$re_1|re_2$’ says that I should try re_1 and if that doesn’t work then I should try re_2

Given this information can you define re_1* recursively? Can you define re_1* using re_1*?

If I tell you it is possible to do this from the mathematic perspective, does this change your answer?

| re_1re_1* or if you prefer re_a | re_b where re_a is # and re_b is re_cre_d and re_c is re_1 and re_d is re_1*

Now is it true that # | re_1re_1* is mathematically equivalent to re_1* ?

Yeah, they are the indistinguishable from eachother.

In the application of this, in actually writing the code for test can I use this definition?

case (ReStar(re1),chars) => test(RUnion(REmptyString,RConcat(re1,RStar(re1))) , chars)

hmmm… maybe. Presuming the other things are implemented… What happens when re1 happens to be "#" by which I mean, what would happen in the following assertion (which is not provided in the lab)

assertResult(false){ retest(RStar(REmptyString)) , "hello") }

We have made something that works perfectly in theory (in the math) but in practice, might run forever without actually solving the problem. Note, there are other edge cases with the similar issues.

L6D3T4 Beginning the Parser

Prior to writing the parser it is recommended that you write the grammar for the parser detailed in writeup exercise 3.b.iv. of the lab handout

In writing the parser I highly recommend using test driven development and only linking 1 part of the parser at a time.

We provide the definition of 're' and 'union' as well as a stub for 'atom'. Without getting into the details of why this works (because you are all smart enough to figure that out for yourself) I recommend the following.

Implement the body of intersect as 'atom(next)'

Start writing tests over atomic expression

```
assertResult(     RE( RSingle('a') )    ){  JsyParser.parse("/^a$/") }
assertResult(     RE( RAnyChar )    ){  JsyParser.parse("/^.$/") }
assertResult(     RE( ??? )    ){  JsyParser.parse("/^#$/") }                    // find ???
assertResult(     RE( RNoString )    ){  JsyParser.parse("/^    ???     $/") }    // find ???
assertResult(     RE( RAnyChar )    ){  JsyParser.parse("/^(.)$/") }
assertResult(     RE( RAnyChar )    ){  JsyParser.parse("/^((.))$/") }
and more… Though I recommend saving tests such
```

Implement the logic to pass those test inside the atom function

Once that works, you should actually be passing union over atom cases as well

```
assertResult(     RE( RUnion(RSingle('a'),RSingle('b') )    ){  JsyParser.parse("/^a|b$/") }
assertResult(     RE( ??? )    ){  JsyParser.parse("/^.|#$/") }          // find ???
assertResult(     RE( ??? )    ){  JsyParser.parse("/^.|#|!$/") }        // find ???
```

You are now ready to actually implement something like intersect. Well, almost…

Write some tests for intersect first

```
assertResult(     RE( RIntersect(RSingle('a'),RSingle('b') )    ){  JsyParser.parse("/^a&b$/") }
```

```
assertResult(     RE( ??? )    ){  JsyParser.parse("/^.&#$/") }          // find ???
assertResult(     RE( ??? )    ){  JsyParser.parse("/^.&#&!$/") }        // find ???
```

Write the body of concat as 'atom(next)' and re-implement intersect to not say atom(next) but rather to implement the logic mandated by the grammar you have written for the lab writeup. Note that the grammar might look similar to something that is already implemented so you might make use of that.

You should now be able to pass tests over both union and intersect combined

```
assertResult(     RE( RIntersect(RUnion(RAnyChar,REmptyString),RNoString) )    ){
        JsyParser.parse("/^.|#&!$/")
}
```

From here, it is up pretty much up to you where you go next, the remaining cases are more challenging. And require an understanding of the work detailed here. Review what you have completed so far and attempt to understand how it all works.

L6D4

- Test Inference rules
- Union

L6D4T1 Test Inference Rules

To clarify what is being asked in question 3.c.i. of the lab 6 handout I would like you to write a set of inference rules that express how expressions of the form /^ re $/ and e1.test(e2) could be integrated to the lab 5 interpreter.

This will require you to write rules for both the step method using the lab 5 operational semantic over monads AND the type checker.

There are many ways to do this. Personally, I work from the bottom up, but sometimes this isn't the best way to learn.

Note that /^ re $/ is a value, and RegExp is a type

Note that the logic e1.test(e2) in javascripty is implemented in the 'test' function you wrote in this lab.

You will likely need 5 rules. 2 type rules. 2 search rules. And a single do rule. But you might find other creative ways to solve the problem and that's cool too.

To remind you how this process works, suppose I want to extend the language of *javascripty* from lab 5 to include the '++' operator apply to variables. This is a difficult task, so for now I will just extend the language to apply '++' to numbers (since that is also something useful to have in a language).

extend the grammar, only where needed

$$e ::= \ldots \mid ++\, e_1 \mid e_1\, ++$$

type checking, recall that my goal for now is simply to allow ++ to apply to numbers

$$\frac{\Gamma \vdash e_1 : \text{number}}{\Gamma \vdash {+}{+}\, e_1 : \text{number}} \qquad \frac{\Gamma \vdash e_1 : \text{number}}{\Gamma \vdash e_1 \,{+}{+} : \text{number}}$$

Or if you prefer to get fancy about it (though perhaps less readable and not the preferred solution)

$$\frac{\Gamma \vdash e_1 : \text{number} \qquad e \in \{\, e_1 \,{+}{+}\,,\, {+}{+}\, e_1 \,\}}{\Gamma \vdash e : \text{number}}$$

Now for step. I have single sub expressions that should become numbers I probably need a search rule and do rule for each.

Note that the searchUnary case could actually cover our search needs but I wrote extra cases anyway.

Note that the solution below is a tad odd, it has to do with the nature of e++ vs ++e.
e++ should increment e and return the old value of e, ++e will increment e and return the new incremented value. This becomes interesting when working with mutable variables. But I am not covering that here today.

$$\frac{\langle M , e_1 \rangle \longrightarrow \langle M' , e_1' \rangle}{\langle M , e_1 \,{+}{+} \rangle \longrightarrow \langle M' , e_1' \,{+}{+} \rangle} \qquad \frac{n' = n + 1}{\langle M , n \,{+}{+} \rangle \longrightarrow \langle M , n \rangle}$$

$$\frac{\langle M , e_1 \rangle \longrightarrow \langle M' , e_1' \rangle}{\langle M , {+}{+}\, e_1 \rangle \longrightarrow \langle M' , {+}{+}\, e_1' \rangle} \qquad \frac{n' = n + 1}{\langle M , {+}{+}\, n \rangle \longrightarrow \langle M , n' \rangle}$$

… Yes, functionally the step on n++ shouldn't *need* to create n' = n + 1, but technically the specification for e++ dictates that I should perform the task.

For an interesting thinking exercise, you might think about how to extend the language of lab 5 to include this concept, in addition to the concept of '++' applied to variables keeping the current notation for variables and noting that they have modes associated with them that effect what the significance of '++' should be.

But again, this is simply to demonstrate what is expected in question 3.c.i. of Lab 6 relating to declaring regular expressions and performing the re_1.test(e_2) operation on regular expressions agains strings. See earlier notes for example of this syntax.

If you are particularly careful you might be able to write the rules in less than 5 rules. HINT, it is not about condensing them, it is about one of them being ridiculous at the moment subject to the current

language. Note that for purposes of extensibility it is probably worth keeping the rule that I am thinking of. (… I know, vague right? …)

L6D4T2 Union

By now I expect that you have correctly coded most of the atomic cases of the parser and have attempted operators using the definition of union as a guide. So, I thought I would explain how I understand the definition of union with respect to the grammar provided to us. Please review this topic and see if this explanation agrees with you current understanding of how union works. If you think that you have a better way to explain it, and are comfortable sharing it, please do.

This discussion will take place without discussing the details of our type ParseResult[RegExpr] in great detail… because that is complicated and not actually needed to understand the problem. It is also easier to understand after you have already solved the problem

Example sentence and flow with goal output (ignoring whitespace)

```
# | . | a          RUnion(RUnion(REmptyString,RAnyChar),RSingle('a'))
    # | .          RUnion(REmptyString,RAnyChar)
        #          REmptyString
        .          RAnyChar
    a              RSingle('a')
```

Function definition provided with slight modifications to avoid some confusion from shadowed variables.

```
def union(next: Input): ParseResult[RegExpr] = intersect(next) match {
    case Success(r, nextp) => {
        def unions(acc: RegExpr, next: Input): ParseResult[RegExpr] = {
            if (next.atEnd) Success(acc, next)
            else (next.first, next.rest) match {
                case ('|', nextp) => intersect(nextp) match {
                    case Success(r, nextpp) => unions(RUnion(acc, r), nextpp)
                    case _ => Failure("expected intersect", nextp)
                }
                case _ => Success(acc, next)
            }
        }
        unions(r, nextp)
    }
    case _ => Failure("expected intersect", next)
}
```

grammar provided

union ::= intersect unions
unions ::= ϵ | '|' intersect unions

The grammar states that non-terminal Union is created via intersect unions. The fact is, union doesn't actually define that a '|' takes place. That is defined in union**s**. Consider the following partial implementation from the provided implementation.

```
def union(next: Input): ParseResult[RegExpr] =
    intersect(next) match {
        case Success(r, nextp) => ???
        case _ => Failure("expected intersect", next)
    }
```

the first thing that union asks, is was there an intersect. This seems reasonable since,
union ::= intersect unions

Note that the code matches on the result of intersect(next) to ask, was the work successful? If not, then the work failed, the pattern to match on might be something such as Failure(str,nextp). But we don't actually match on that pattern, because for this particular function we need to state that the error/the Failure was that I "expected intersect" when inside of 'next' and I was not able to find one.

Now what do we do if intersect succeeds? In calling intersect(next) I have found Success(r,nextp), where are r is the regular expression discovered intersect and nextp is the rest of the input.

Presuming intersect is correct down to atom, for the example sentence """# | . | a"""
    r == REmpty
    nextp == """| . | a"""

Now recall union ::= intersect unions

Since I found intersect I need to call unions. It turns out that it helps if unions is defined inside of union. And here is the partial implementation.

```
def union(next: Input): ParseResult[RegExpr] = intersect(next) match {
    case Success(r, nextp) => {
        def unions(acc: RegExpr, next: Input): ParseResult[RegExpr] = ???
        unions(r, nextp)
    }
    case _ => Failure("expected intersect", next)
}
```

I define unions to be able to accumulate a regular expressions given a starting RegExpr and a beinging Input.

for the example sentence """# | . | a""", the first call to unions has input

acc = REmpty
next = “””| . | a”””

Now how is unions going to be defined? See the reference grammar `union ::= $\epsilon$ | ‘|’ intersect unions` Either it is empty or its not, if it’s not I hope that the first element is ‘|’, if that isn’t the case then I might have succeeded, in that I didn’t need to find ‘|’ (the reason why this is true is rather complex. After you have a better understanding of the parser I encourage you to think about why it will work – the short version is : this is a side step in the recursion and so unlike finding an initial intersect the ‘|’ is not required for all sentences and so I return a Success rather then a failure… But I succeed without consuming anything).

Defining the problem in prose

```
def unions(acc: RegExpr, next: Input): ParseResult[RegExpr] = {

    // if it’s empty then I’m done with unions

    // otherwise I need to ask more questions

        // I might have found ‘|’ to use to construct an RUnion with recursion

        // I might not have found a ‘|’ which means that I am done with unions
        // This doesn’t mean that I have failed at anything

}
```

Using some syntax to replace the prose

```
def unions(acc: RegExpr, next: Input): ParseResult[RegExpr] = {

    if (next.atEnd) ???

    else (next.first, next.rest) match {

        case ('|', nextp) => ???

        case _ => ???

}       }
```

Now how should I fill in the implementation? Consider the table of desired behavior for unions. See if you can use that to code unions without looking at the implementation provided to you. Then check your work. Repeat the process until you are correct.

i : the time I am calling unions
acc : the partial RegExp accumulated thus far
next : the partial Input that I have not looked at yet (a great name for this is suffix)

example sentence to union “””# | . | a”””

should call unions with the input at time 0

i	acc	next	next.isEmpty	next.first == ‘|’
0	REmptyString	“””|.|a”””	False	True
1	RUnion(REmptyString,RAnyChar)	“””|a”””””	False	True
2	RUnion(RUnion(REmptyString,RAnyChar),RSingle(‘a’))	“””””””	True	N/A

It returns RUnion(RUnion(REmptyString,RAnyChar),RSingle(‘a’))

Here is an extended example
example sentence to union “””# | . | a | ! | d”””

should call unions with the input at time 0

i	acc	next	next.isEmpty	next.first == ‘|’
0	REmptyString	“””|.|a|!|d”””	False	True
1	RUnion(REmptyString,RAnyChar)	“””|a|!|d”””	False	True
2	RUnion(RUnion(REmptyString,RAnyChar),RSingle(‘a’))	“””|!|d”””	False	True
3	RUnion(acc2, RNoString)	“””|d”””	False	True
4	RUnion(acc3, RSingle(‘d’))	“””””””	True	N/A

Returns RUnion(acc3, RSingle(‘d’))

i.e. RUnion(RUnion(acc2, RNoString), RSingle(‘d’))

i.e. RUnion(RUnion(RUnion(RUnion(REmptyString,RAnyChar),RSingle(‘a’)), RNoString),RSingle(‘d’))

An example where nextp is not an intersect and we fail inside unions

example sentence that should fail “””# | & .”””

should call unions with the input at time 0

i	acc	next	next.first == ‘|’	nextp	Intersect(nextp)
0	REmptyString	“””|&. ”””	True	“””&.”””	Failure(“expected concat”, “””&.”””)

Returns Failure(“Expected Intersect”,”””&.””””)

And finally an example where the next.first != “|” comes into play and we succeed. It involves the difficult problem from atom being the ‘(‘ ‘)’ s. anything with parens is enough but I’ll show a test that is somewhat more interesting

example sentence to union “””(a|b)””” should become RUnion(RSingle(‘a’),RSingle(‘b’)

re(“””(a|b)”””) → union(“””(a|b)”””) → intersect(“””(a|b)”””) →* atom(“””(a|b)”””) →*
re(“””a|b)”””) → union(“””a|b)”””) → intersect(“””a|b)”””) →* **unions(RSingle(‘a’), “””|b)”””)**

should call unions with the input at time 0

i	acc	next	next.isEmpty	next.first == ‘|’
0	RSingle(‘a’)	“””	b) ”””	False	True
1	RUnion(RSingle(‘a’),RSingle(‘b’)	“””) ”””	False	False

Unions returns Success(RUnion(RSingle(‘a’),RSingle(‘b’), “””)”””) and if other things are coded correctly then we will eventually get the expected RegExpr RUnion(RSingle(‘a’),RSingle(‘b’)

L6: Additional Resources
L6: Closing Thoughts

Closing thoughts