# Lab 3: Interpreter with theory

## Table of Contents



## 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

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

- 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).

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

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

Each step rule follows the operational semantic $e \longrightarrow e'$. 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.

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

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
$$\frac{. \quad e \longrightarrow e' \quad .}{\text{uop } e \longrightarrow \text{uop } e'}$$

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)
}
```

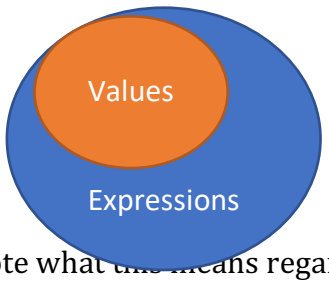
Note that ep is defined in the premise by recusing on the judgment form "⟶". Again, I will not complete the body of this case statement here, but you should try it yourself.

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

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

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*.

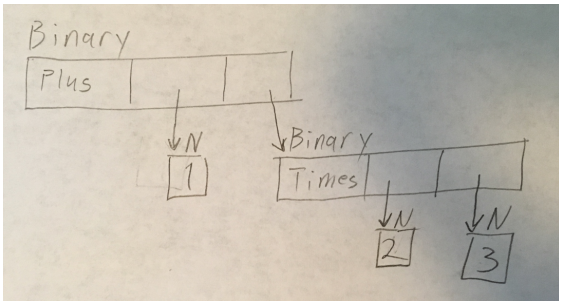
Let's look at what this means about drawing Parse Trees based on Grammar 3 vs Grammar 4.

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.

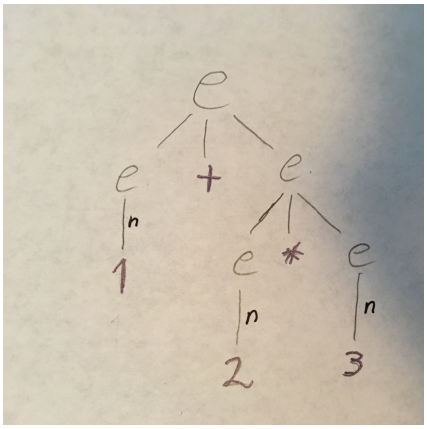
Note that I am NOT looking for the AST of Binary(Times,Binary(Plus,N(1),N(2)),N(3)). I want the tree of Binary(Plus,N(1),Binary(Times,N(2),N(3)))

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!

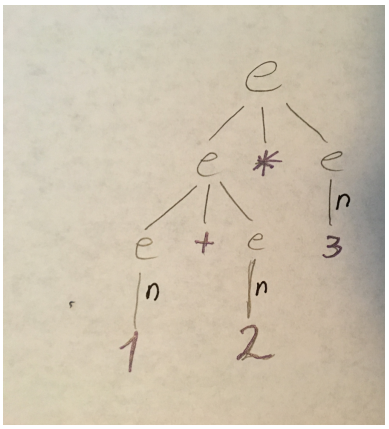
Note how the tree has a subtree that encapsulates 2 valued arguments to '*'




PARSE TREE 2 :

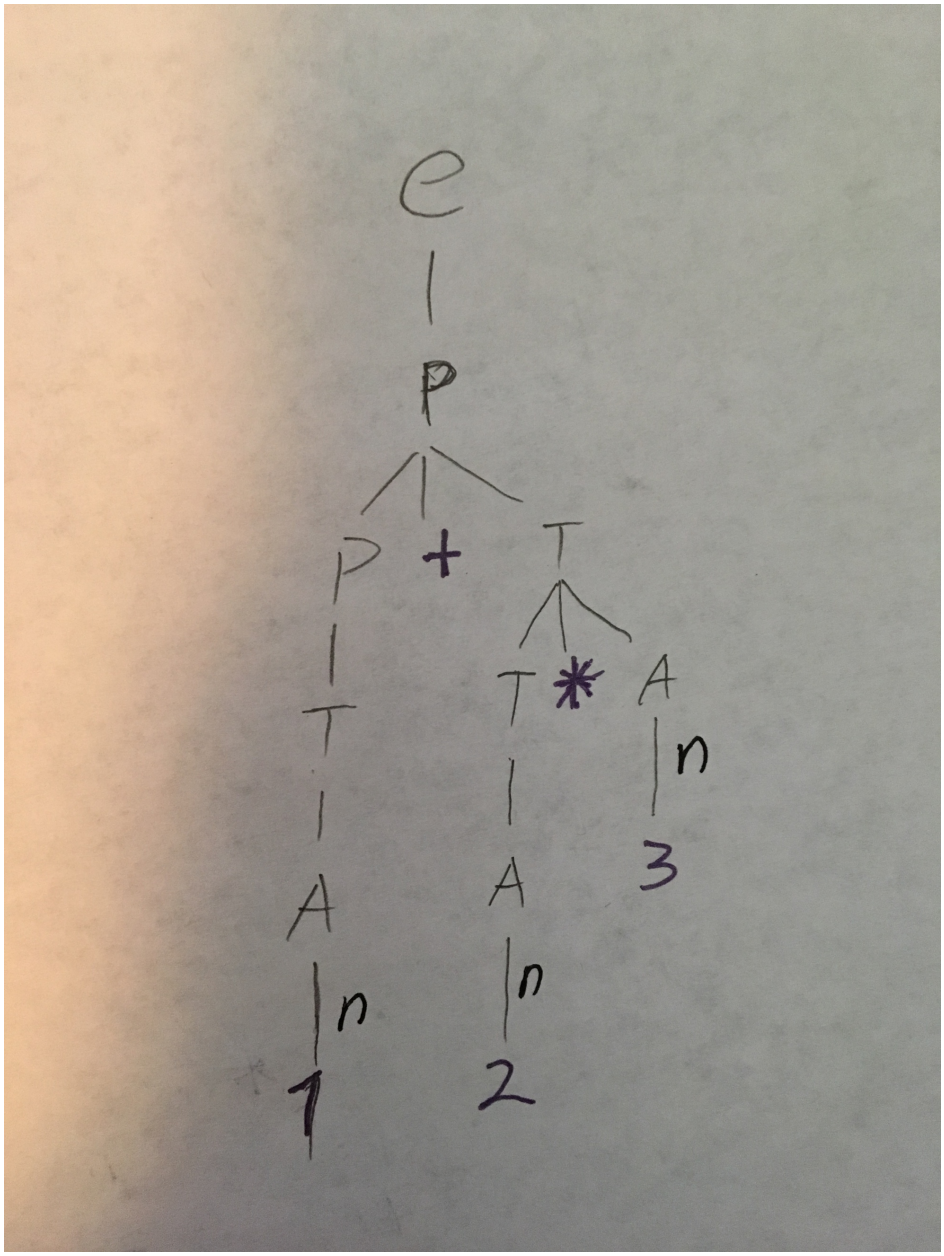
Has the AST: Binary(Times,Binary(Plus,N(1),N(2)),N(3)) - NOT what I want!

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


e
e
*
e
e
+
e
n
3
n
n
1
2

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 ::= $\epsilon$ | + TimesDiv PlusMinusPrime | - TimesDiv PlusMinusPrime
TimesDiv ::= Neg TimesDivPrime
TimesDivPrime ::= $\epsilon$ | * 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