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.

`ϵ`: 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'

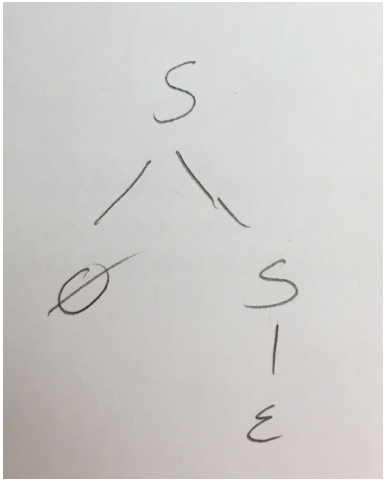
Binary Strings
S ::= 1S | 0S | ϵ

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.

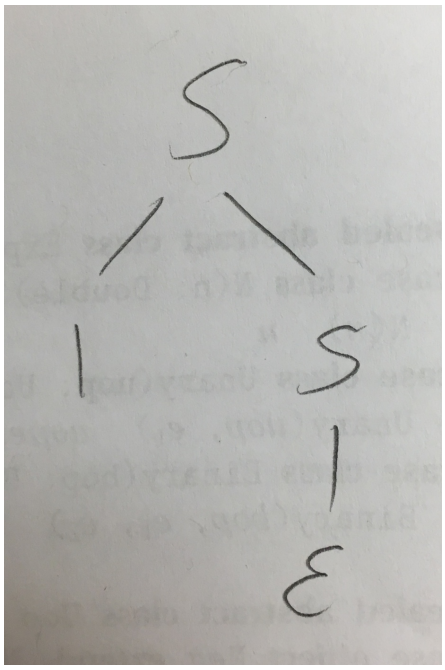
There are two great tools demonstrating that a sentence exists in the language defined by a grammar, Linear Derivations and Parse Trees. This course will test you on Parse Trees only (because they have significantly more value for expressing structure of a grammar) but I will show you some linear derivations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

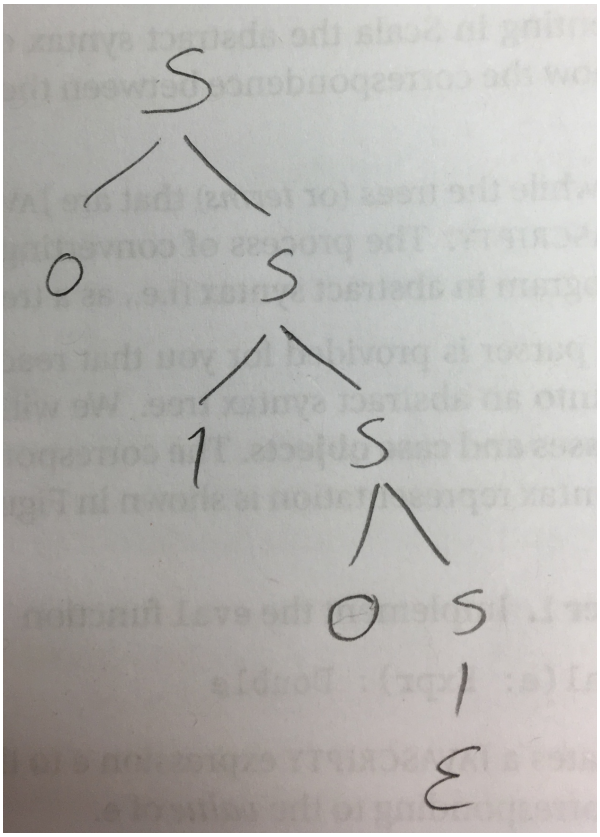
NOTE: We will always need to start with a single instance of our "START Symbol" i.e. the first non-terminal defined in our grammar which, here, is "S"

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

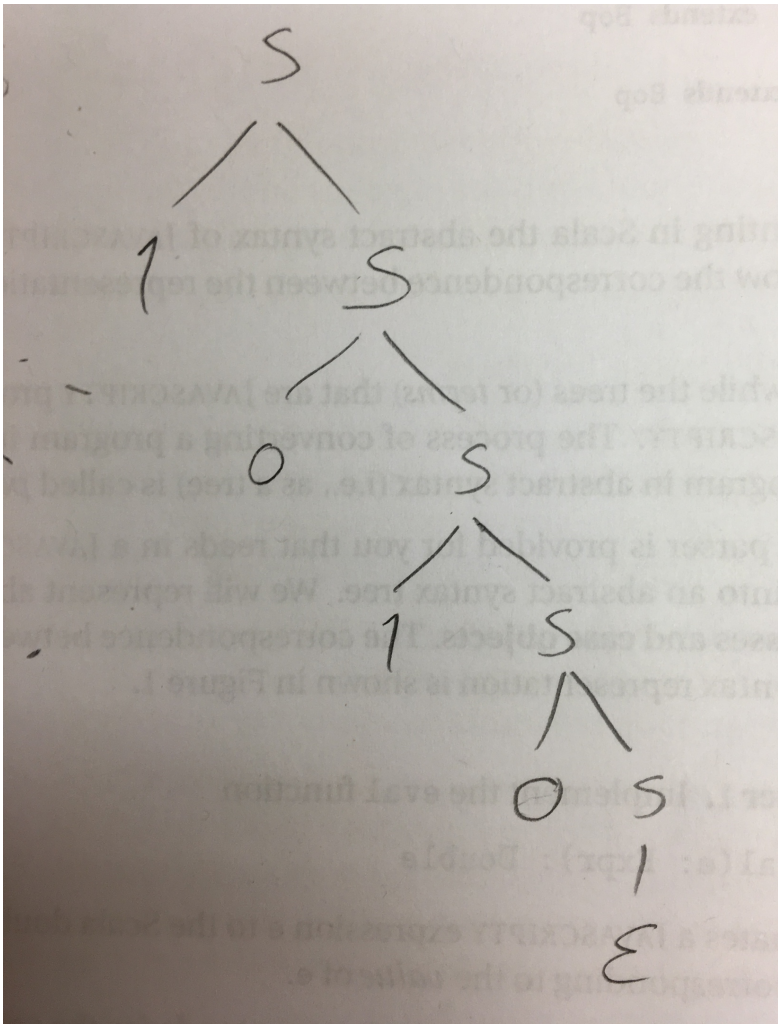
7.5 is a truethy value. For this reason, the following is a valid unit test in correct Scala syntax for our toBoolean function

```
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 *javascripty*$_1$ 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 *javascripty1*. We would like our interpreter (the eval function) to do the same things that *javascripty1* 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).

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.

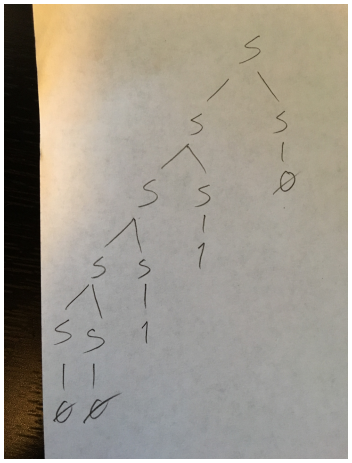
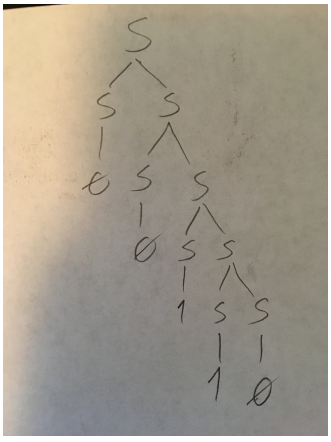
- Grammar 0

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

$S_1 \Downarrow b_1 \qquad S_2 \Downarrow b_2 \qquad b = b_1 \wedge b_2$

$S_1 \wedge S_2 \Downarrow b$

RULE2

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

$\sim S_1 \Downarrow b$

RULE3

$b \Downarrow b$

Or alternatively for typesetting
C : $S_1 \wedge S_2$ \Downarrow b
J0 : S_1 \Downarrow b_1
J1 : S_2 \Downarrow b_2
J2 : $b = b_1 \wedge b_2$

C : $\sim S_1$ \Downarrow b
J0 : S_1 \Downarrow b_1
J1 : $b = \sim b_1$

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
    $S ::= TS \mid ST \mid \epsilon$
    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
    $S ::= aS0 \mid Sb \mid 1S \mid \epsilon$
  - 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 :
      . S TheThing S' .

S 0 TheThing S' 1

- Rule 2 :

  . S TheThing S' .

  S 1 TheThing S' 0

- Rule 3 :

  . .

  ϵ TheThing ϵ

- 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 :

$$\frac{S\ \text{TheThing}\ S'}{S\ 1\ \text{TheThing}\ S'\ 0}$$

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.

$$\frac{}{\text{Rule 2}\quad 01\ \text{TheThing}}$$

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.

$$\frac{\dfrac{}{0\ \text{TheThing}}}{\text{Rule 2}\quad 01\ \text{TheThing}}$$

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 :

$$\frac{S\ \text{TheThing}\ S'}{S\ 0\ \text{TheThing}\ S'\ 1}$$

$$\frac{\dfrac{\text{Rule 1}}{0\ \text{TheThing}}}{\text{Rule 2}\quad 01\ \text{TheThing}}$$

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.

$$\frac{\dfrac{\dfrac{}{\epsilon\ \text{TheThing}}}{0\ \text{TheThing}}}{\text{Rule 2}\quad 01\ \text{TheThing}}$$

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

$$\frac{}{\epsilon\ \text{TheThing}\ \epsilon}$$

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

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

Rule 3 ._____.
Rule 1._____ϵ TheThing ϵ_____.
Rule 2 ._____ϵ 0 TheThing ϵ 1_____.
ϵ 01 TheThing

Without ϵ – where reasonable.

Rule 3 ._____.
Rule 1._____ϵ TheThing ϵ_____.
Rule 2 ._____0 TheThing 1_____.
01 TheThing

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

Rule 3 ._____.
Rule 1._____ϵ TheThing ϵ_____.
Rule 2 ._____ϵ 0 TheThing ϵ 1_____.
ϵ 01 TheThing ϵ 10

Without ϵ – where reasonable.

Rule 3 ._____.
Rule 1._____ϵ TheThing ϵ_____.
Rule 2 ._____0 TheThing 1_____.

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 $\dfrac{\text{! S = b}}{\text{! S0 = b}}$

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

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

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'

Rule 3. ———————————— .
Rule 1. ϵ TheThing ϵ .
Rule 2. ϵ0 TheThing ϵ1 .
Rule 2. ϵ01 TheThing ϵ10 .
Rule 1 . ϵ011 TheThing ϵ100 .
ϵ0110 TheThing ϵ1001

L2D3T2Q1 Solution2: derivation of TheThing on input '0110'

Rule 3. ———————————— .
Rule 1. ϵ TheThing ϵ .
Rule 2. 0 TheThing1 .
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