Lab 1: An intro to Scala and Interpretation

Table of Contents

L1: Preface

L1D1 syllabus based

Print out, read and annotate the lab 1 handout

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

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

L1D2

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

L1D2T1: Functional Programming in Scala

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

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

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

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

L1D2T2: Scala Syntax

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

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

First the function declaration: --

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

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

Start the function body: --

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

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

Filling in blanks: ---

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

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

Reformat: --

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

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

Stylizing: ---

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

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

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

~~>

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

~~>

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

~~>

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

~~>

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

~~>

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

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

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

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

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

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

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

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

L1D2T3: Bindings and Scope

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

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

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

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

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

L1D2T4: Scala Pattern Matching

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

L1D3

- Linked lists
- Grammars (first exposure)
- Types

L1D3T1: Linked Lists

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

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

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

Here is a possible linked list made with the grammar

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

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

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

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

sealed abstract class LL   // S

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

case object End extends LL   // End

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

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

Put it all together:

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

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

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

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

A recipe/heuristic for writing this….

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

Let's walk through the recipe for print.

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

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

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

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

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

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

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

L1D3T2 Grammars (first exposure)

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

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

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

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

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

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

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

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

L1D3T3 Types

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

(-(1) – (2))

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

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

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

Bellow are a few steps I took to write the above solution.

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

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

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

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

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

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

Here is a way to express the types:

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

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

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

Heres the treelike expression of the type of the function body of bar

```
if (x > 2) { (1, x) } else { x }: ???
        x > 0: Boolean
                x: Int
                0: Int
        (1, x): (Int, Int)
        x: Int
```

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

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

L1D3 Solutions

L1D3T1Q1

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

L1D3T3Q1 is baz well typed?

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

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

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

L1D4

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

L1D4T1: Abstract syntax trees

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

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

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

Essentially:

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

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

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

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

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

Essentially:

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

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

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

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

```
sealed abstract class Expr

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

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

sealed abstract class Uop

case object Neg extends Uop /* - */

sealed abstract class Bop

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

Or if you prefer:

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

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

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

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

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

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

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

L1D4T2: More on grammars – you can skim this part

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

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

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

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

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

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

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

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

L1D4T3 JS vs Scala – stop skimming….

Now let us consider why this is important.

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

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

Expr

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

Bop

- Plus
- Times
- Div
- Minus

Uop

- Neg

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

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

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

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

L1D4T4 AST for binary logic

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

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

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

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

we can view this in it's alternate form
Funish

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

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

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

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

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

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

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

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

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

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