Lab 6: Parsing

Table of Contents

L6: Preface

L6D1

- BNF to EBNF
- Continuations

L6D1T1 BNF to EBNF

Consider the following grammars

G1

e ::= n | - e | e + e | e – e | e / e | e * e | (e)

G2

e ::= PM
PM ::= PM + TD | PM – TD | TD
TD ::= TD * N | TD / N | N
N ::= - N | A
A ::= n | (e)

I presume by now you understand why G1 can be translated to G2 and what the value of this grammar translation is. If not, please review L5D8 notes on grammatic precedence. I also assume that you understand the terms 'terminal', 'non-terminal', 'meta-symbol' and 'meta-variable' as it relates to CFG in BNF form. If you do not, please review notes on this topic from Labs 1 – 3 (also summarazied in L5D8 notes).

So far we have learned backus naur formed context free grammars.

Recall that BNF grammars have 3 meta-symbols:

::=	produces
|	or
ϵ	empty sentence

Now I will introduce EBNF CFG, extended backus naur formed context free grammars

Similar to BNF, we now have an additional operator in the language i.e. we have an additional meta-symbol.

::=	produces
|	or
ϵ	empty sentence
{ }	0 - or – many

Consider the language of 0-or-many 'a' symbols defined in BNF

G3

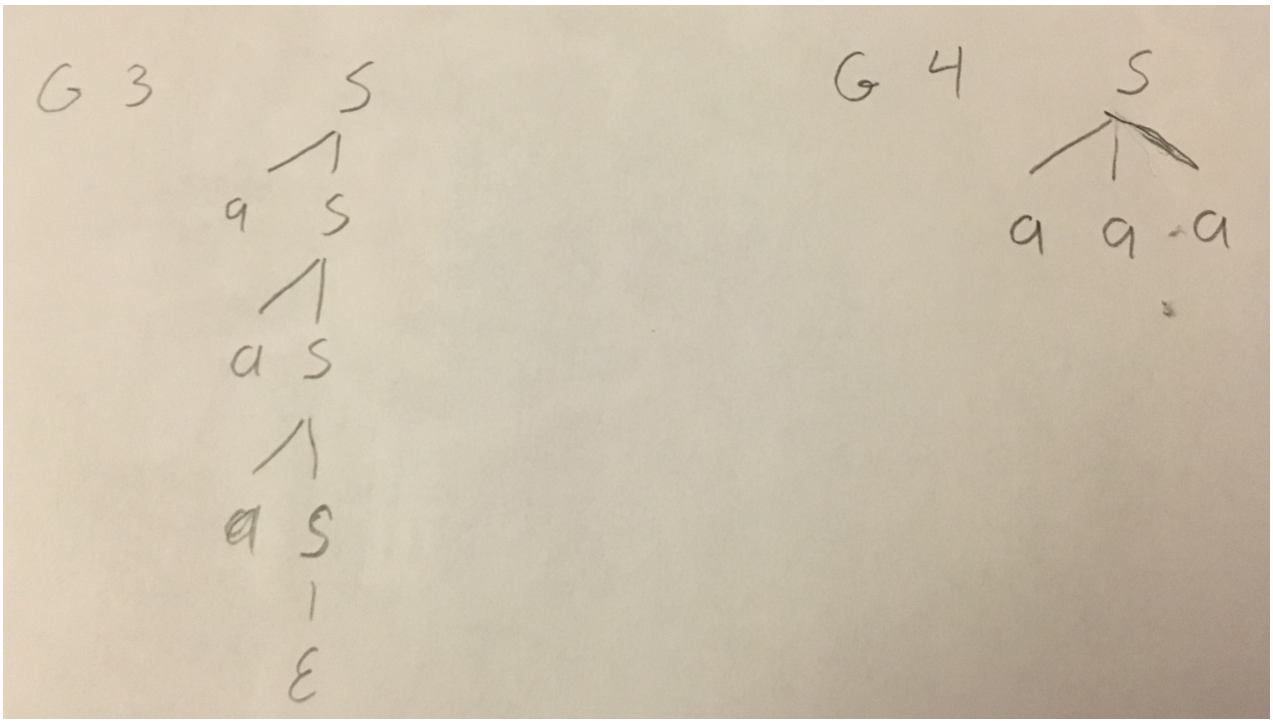
S ::= a S | ϵ

Now consider the same language defined in EBNF

G4

S ::= { a }

Parse trees for 'aaaa' in G3 and G4



Reconsider Grammar G2

G2

e ::= PM
PM ::= PM + TD | PM – TD | TD
TD ::= TD * N | TD / N | N
N ::= - N | A
A ::= n | (e)

This grammar is left recursive about the binary operations. Let's see a use of the { } meta-variable of EBNF. These are great for expresses behavior in a language but I cannot translate this to code very easily (if A ::= A + B | B then the definition of A is A is A is A is A is A is…onward for infinity and finally… + B | B). So let us remove the recursion by flattening grammar in EBNF. I will explain the process for redefining PM. PM ::= PM + TD | PM – TD | TD. Here PM has 3 possible productions, two of the productions are recursive : PM + TD and PM – TD. It also has 1 non-recursive production: TD. The recursive productions hold the same form PM bop TD. So I will re-write the work as PM ::= TD { things… } where the things are defining bop TD. See the solutions below and attempt to reverse the process to better understand what is being accomplished here

e ::= PM
PM ::= TD { + TD | – TD }
TD ::= N { * N | / N }
N ::= { - } A
A ::= n | (e)

Tomorrow we will look at using this as a stepping stone toward writing right recursive grammars (with right associative parse trees) that can be used as a reference grammar to write left associative recursive decent parsers.

L6D1T2 Continuations

In this course we have discussed the functional concept that "functions are values". With this we can make function calls using call by value that actually pass functions as arguments to another function. The input function is known as a 'callback'. The function being called is known as a higher order function.

e.g.

```
def f ( x : Int ) ( g : Int => Int ) : Int = {
        x + g ( x * x )
}
```

HERE : f is a higher order function because it accepts functions as input and g is a callback since it is a function passed as input to a function (which by definition will be a higher order function)

Now I will introduce the concept of 'continuations'. A continuation is a specific kind of callback i.e. it is a specific kind of function passed as input to an HOF. Continuations work well in solving problems that require that I traverse multiple paths as well as problems for which I can short circuit my operations based on a variety of edge cases.

In MulList.scala (provided on the main web page) we see an example of the later, a problem that can be short circuited. MulList.scala expresses 12 different ways that I could take a list of Int and return the multiplication of all the elements in the list. Here is a basic version that uses foldLeft. **To simplify the problem, we assume that the multiplication of all the elements in the empty list is equal to 1.**

```
def mulList0( l : List[Int] ):Int = {
        l . foldLeft ( 1 ) { ( acc , h ) => acc * h }          // OR if you're fancy…          ( 1 /: l ){ _*_ }
}
```

Consider the multiplication of all the elements in the List(5,0,6). What is the value? We as college students likely knew the answer to be 0 without actually bothering to multiply 5*0*6. So, maybe we can take that human intuition and translate it to code. The big trick here is to reword the problem. I do not simply want to multiply all the elements of the list together…

Consider l : List[Int]. If `l` contains any 0s, return 0, else multiply all the elements of `l` together and return the value found.

Without continuations the solution to this is pretty simple. Scan the list 1 time to see if there are any 0s. if there are, return 0 else, accumulate the multiplicative of all the values

```
def mulList1( l : List[Int] ):Int = {
        if ( l.exists { h => h == 0 } ) {                    // fancy….. l exists { _==0 }
                0
        } else {
                l . foldLeft ( 1 ) { ( acc , h ) => acc * h }     // fancy…    ( 1 /: l ){ _*_ }
        }
}
```

But suppose, that there is another constraint on the problem. What if I am only allowed to scan the list 1 time. If I find a 0 then I must return 0 and stop scanning the list. If there are no 0s then I should multiply all the values together. If the list is empty return 1. HOW THE HECK CAN I DO THAT?! That seems really challenging. The short answer is, we can introduce a continuation to our work.

The particular kind of continuation I will use here is called a 'success continuation' (SC) . The sc is a function that says given some input do some work. Here the sc expresses that if I am going to succeed in the goal of multiplying all the elements of the list together, this is how I would accomplish this work provided a single element of the list.

I can't write my work over fold because the problem specifies that I must stop scanning the list the moment I find a 0 (if there is a 0 in the list). Below is my thought process for solving the problem.

```
def mulList2( l : List[Int] ):Int = {
        ???
}
```

I can't use fold, but I need to create a looping structure. I need to have reference of some accumulator while I loop, so I cannot use mulList2 as my looping structure. Ill create another function. Ill give that function a sc to denote my accumulation

```
def mulList2( l : List[Int] ):Int = {
        def myLoop( l : List[Int] ) ( sc : Int => Int ) : Int = {
                ???
        }
        myLoop( l ) ( ??? )
}
```

for the initial call to myLoop I need to set the sc to something intelligent. I notice that sc is defined with the type Int => Int. Currently I have no integers available to me. The best function that I can write is known as the identity function. I might write it as r => r to say, given some residual, return the residual.

As for the body of myLoop, I need to loop over a List[Int], so I'll pattern match to my obvious cases

```
def mulList2( l : List[Int] ):Int = {
        def myLoop( l : List[Int] ) ( sc : Int => Int ) : Int = l match {
                case Nil => ???
                case h :: t => ???
        }
        myLoop( l ) ( r => r )
}
```

Now, let us assume that what we have so far is correct, and complete the body of the Nil case statement. Prior to now, I stated that mulList of the empty list is 1, I'll keep that value here. I could just return 1, but that doesn't feel right. Is there anything else that I can write that will result in 1? Hmm…. What are my data? I have mulList2, myLoop, l, and sc. mulList2 is just a driving function for the loop so

I probably don't need to do anything with that. myLoop is the function I am in and I know it should be recursive, but I am in the base case for the data structure that this operates on so a call to myLoop probably won't help. l is the list that happens to be Nil… sc is a function. Suppose this is the initial call to myLoop, def sc (r : Int) : Int = r …. Can I get this case statement to evaluate to 1 with a statement other than '1' itself? Think about it…. Think harder… YES, absolutely sc(1) will evaluate to 1 in this case.

```
def mulList2( l : List[Int] ):Int = {
        def myLoop( l : List[Int] ) ( sc : Int => Int ) : Int = l match {
                case Nil => sc(1)
                case h :: t => ???
        }
        myLoop( l ) ( r => r )
}
```

Now before we move forward, is there any specific case that I am interested in? Consider the original problem. *I am only allowed to scan the list 1 time. If I find a 0 then I must return 0 and stop scanning the list. If there are no 0s then I should multiply all the values together.* Lets write this code to check if I have a 0. Below is an intelligent way to ask that in scala

```
def mulList2( l : List[Int] ):Int = {
        def myLoop( l : List[Int] ) ( sc : Int => Int ) : Int = l match {
                case Nil => sc(1)
                case 0 :: t => ???
                case h :: t => ???
        }
        myLoop( l ) ( r => r )
}
```

If I find a value that is 0, what should I return? 0. Now again that seems too simple. Should I instead use sc(0)? The short answer is no, I shouldn't. Recall sc, is being defined to express, if there are no 0s in the list then this is the work I'll do. But here I have found a 0. So I don't actually want to call sc. Take that at face value and try more practice problems.

```
def mulList2( l1 : List[Int] ):Int = {
        def myLoop( l2 : List[Int] ) ( sc : Int => Int ) : Int = l2 match {
                case Nil => sc(1)
                case 0 :: t => 0
                case h :: t => ???
        }
        myLoop( l1 ) ( r => r )
}
```

Note that the case body of "case 0::t" is just "0" and doesn't actually use declared variable `t` so I might also write the case as "case 0 :: _ => 0"

Now that I have my base cases it's time for the difficult part. The Inductive case. Let's first lay out my data. Note the slight modifications to l above. Define, in your own words, myLoop, l2, sc, h, t as they relate to the ??? above and then compare to the following

- myLoop : a function. provided a list l2 and a success continuation sc, returns Scanning the list l2 one time, if we find a 0 then we return 0 and stop scanning l2. If there are no 0s then we return the result of multiplying all the elements of the list together.
- l2 : A list
- sc : a success continuation that says, if I never find a 0, this is the work I would do in order to multiply all the elements of the list together.
- h : an Int, the head value of l2, it does not equal 0
- t : a list of Ints, the tail of l2 after removing the head element h. This might contain 0s.

Take a moment to think about how we can bring all of this together to solve the problem myLoop.

HINT : we will need to use recursion and there must be some kind of consumption going on

HINT : I will need to write a new sc for the next iteration of myLoop to use. I should write it as a lambda in line but I can write it in a different way

HINT : I don't want to do multiplication unless there are no 0s in the list. I don't want something like h * myLoop(t)(sc)… that would not satisfy the conditions of our problem.

The solution is at L6D1T2Q1.

Everything we just looked at regarding mulList came with a simplification. **To simplify the problem we assumed that the multiplication of all the elements in the empty list is equal to 1.** But that doesn't really seem correct does it. Multiply no elements together, would you find 1, or 0 or something more like None. Attempt to rewrite the problem to return Option[Int] where if the list is empty we return None, else we return Some(n) such that n denotes the multiplicative of the list.

For more practice, attempt to solve the problem on a List[Double] as well.

Then attempt this work on a Binary Tree rather then a List.

Now consider applying this work to a Binary Tree. There are many paths to explore.

Now try solving the following problem, no solution provided.
write a function named hasSubList that takes as input a l:list[Int] and n:Int. You can only scan the list once. If the list contains a contiguous sublist such that the sum of the list equals n, then return true, else, return false. Let the sum of an empty list be 0. A sublist can have any length 0 to n for t list l of length n.

HINT : you might want a success continuation that says, if the current path is a part of the solution then this is the partial sum of the path

HINT : you might also want a fail continuation that says, if the current path is wrong, then this is the result of attempting the next possible path

Solutions

L6D1T2Q1

I am only allowed to scan the list 1 time. If I find a 0 then I must return 0 and stop scanning the list. If there are no 0s then I should multiply all the values together. Lets write this code to check if I have a 0.

```
def mulList2( l1 : List[Int] ):Int = {
        def myLoop( l2 : List[Int] ) ( sc : Int => Int ) : Int = l2 match {
                case Nil => sc(1)
                case 0 :: t => 0
                case h :: t => myLoop( t ) ( acc => sc(  h * acc  ) )
        }
        myLoop( l1 ) ( r => r )
}
```

NOTEs for case h :: t => ???

- h * myLoop(t) (sc) is not correct since that potentially causes us to multiply 0 by other values.
- myLoop(t) (acc => h * sc(acc)) would also work
- acc is not some kind of reserved term. We are declaring an input to a function, we can name this parameter whatever we would like.

L6D2

- EBNF back to BNF
- Continuations Continued
- Javascripty Test

L6D2T1 EBNF back to BNF

At this point, I presume that you understand what grammars are, and why they are useful. I think that you have a decent handle on writing EBNF grammar but here are some practice problems to warm you up. The solutions are at L6D2T1Q1. Try these in EBNF as well as BNF if you need the practice

1. Describe, in English, the language defined by this grammar
    a. S ::= a a a { a }
2. Construct a grammar with start symbol B that defines 0-or-many ‘b’ symbols
3. Construct a grammar with start symbol A that defines 0-or-many ‘a’ symbols
4. Construct a grammar with start symbol S that defines 0-or-many ‘a’ symbols followed by 0-or-many-b symbols 0-or-many times. If you will the language of binary over a and b rather than 0 and 1.
5. Construct the grammar to define the language palindromes over ‘a’s and ‘b’s.

TANGENT: Another useful exercise if you are interested (but not at all needed for this class) is to think of other operators we can add to the language we use to describe grammars. BNF is very powerful, EBNF is equally as powerful but it has an extra meta-symbol (or rather a set of 2 symbols that need to go together in a particular order) that make our job as language designers a bit easier. Are there other meta-symbols or operations that might be useful to have? Are there other ways of writing these grammars? Fun fact, BNF actually has an unnecessary meta-symbol, the |, the pipe, the or, the vertical bar, whatever you call it is not super necessary but has an advantage.

BNF S ::= a | b | c S c

Other form of CFG that I forget the name of

S → a
S → b
S → c S c

TANGENT CONTINUED: Again, if this interests you I highly recommend theory of computation or picking up a book on the matter. Many of them are rather dense with mathematic notation but they tend to be filled with fun jokes and interesting logic puzzles. It WILL teach you new ways to solve problems and new ways to think. It usually teaches you the true nature of recursion. We've seen a bit of recursion's nature this class when discovering if a grammar is ambiguously defined. These text books tend to have great visual representations of the topic.

ONE LAST POINT TO THE TANGENT : Theory of Computation is typically taught by research professors from the PLV (programming language and verification) lab. They are all really brilliant people with neat research going on. In my experience, they are also all incredibly talented and engaging instructors/ guides/ lecturers/ professors/ teachers…

BACK TO THE TOPIC…

So, EBNF is great and all. I can use it to quickly construct non-recursive grammars that denote qualities I want in how the language should be interpreted. We use them in this course to create grammars that demonstrate precedence without using left recursion about binary operations. BUT these grammars are not representative of what I would code in a parser.

Super ambiguous grammar – shows all that can be made in the language – great for constructing AST/or any data structure really – but states nothing about precedence or associativity

e ::= n | - e |e + e | e – e | e * e | e / e | (e)

BNF grammar with left recursion when possible – can show precedence in the grammar – left recursion is problematic to code (I'm not sure that it is impossible to code, but coding it often creates infinite loops, it can make our process slow and in some cases the work becomes non-deterministic)

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

EBNF grammars without left recursion – resolve the issue of left recursion – flatten our work a bit – can be useful for coding if you know how to use them (but not semantically correct) - we use this as a great intermediate to the final step in building the desired grammar (helps us avoid errors)

e ::= PlusMinus
PlusMinus ::= TimesDiv { + TimesDiv | – TimesDiv }
TimesDiv ::= Neg { * Neg | / Neg }

Neg ::= - Neg | Atom
Atom ::= n | (e)

NOTE: PlusMinus and TimesDiv are re-writen in the same way.

Condition
- the grammar rule has recursive productions and non-recursive productions
- the recursive productions have the same form Recur bop Don'tRecur

Action
- take the non-recursive production of the grammar rule and use that to replace 'Recur'
- wrap all of the 'bop Don'tRecur' in '{' '}' and separate them by '|'

NOTE : Neg is not rewritten. We can rewrite it, but we don't have to since the goal here is to remove left recursion. Still, you might find it useful to re-write to EBNF for practice. Try that on your own and see solutions at bottom of this document. It follows the exact same logic as our binary cases here

Now here is where it gets interest. How do I take that grammar and re-write it back to BNF without left recursion, still unambiguous, still with the same precedence, and in a manner such that I can easily use the grammar as a guide to write a recursive decent parser with left associative parsing despite the grammar itself being right recursive (has right associative parse trees)

Consider this snippet of brilliant provided back in the write-up of lab 1 that describes the language of binary operations applied to operands in 2 ways.

e ::= e operator operand | operand

e ::= operand es
es ::= operator operand es | ϵ

Now let us write in the intermediate step that was not provided

e ::= e operator operand | operand

e ::= operand { operator operand }

e ::= operand es
es ::= operator operand es | ϵ

We have discussed how to transform from the first grammar to the second grammar, and with a little creative thinking we can assure ourselves that it works. What about the logic for transforming the second grammar to the third grammar?

Think about it…

Think a little more….

Okay, so here is how I see it. I had in EBNF Thing { Stuff } meaning, Thing happens once followed by stuff 0-or-many times. So what if I wrote it in BNF as Thing Bleh, where Bleh is an additional non-terminal in the

language that I need to write to express { Stuff }, or rather 0-or-many instances of stuff. I have seen how to do that when Stuff is small. The logic is the same when Stuff is larger.

RECALL EBNF S ::= { a } BNF-right-recursive S ::= ϵ | a S

So EBNF S ::= b { c } BNF S ::= b A A ::= ϵ | c A

Or, If you prefer

EBNF S ::= Thing { Stuff }
BNF S ::= Thing Bleh
Bleh ::= ϵ | Stuff Bleh

Take a moment to absorb all of that

L6D2T1Q2
Rewrite this grammar to BNF in the manor descirebed above "S ::= { a | b }"

L6D2T1Q3
Attempt to re-write our grammar for a 5 function calculator into a BNF grammar of the form shown above.

L6D2T2 Continuations Cont.

Continuations, what are the good for? Write both implementations. Run both of them. See which one is faster from the human perspective. Google how to benchmark these and find out which one is actually faster. Try it in different langauges.

This section will discuss a particularly challenging problem that I don't expect you to know how to solve yet. I think that walking through the solution might help you find and solve problems that interest you using the technique of programming with continuations.

Complete a function named hasSubList that takes as input a list[Int] `l` and an Int `n`. **You can only scan the list once.** If the list contains a contiguous sub-list such that the sum of the list equals n, then return true, else, return false. Let the sum of an empty list be 0. A sublist can have any length 0 to n for the list `l` of length n.

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean =  ???
```

One of my conditions, **You can only scan the list once**, is what makes this question particularly challenging. In practice, I might consider relaxing this constraint, solving a similar problem and then re-adding this constraint. You might find that to be helpful. Please take a moment to solve this problem without that constraint. Implement your solution in Scala. I have provided 1 solution at L6D2T2Q1.

There are many way's in which I can answer this question, some of which are more intelligent then the solution we will go through here. But I want to walk through the solution to this that uses continuations to solve the problem

Suppose I want to do this by scanning my list only once. I need both a success continuation and a fail continuation:

As I scan the list, I might find a sub-list that has the sum n, in which case, I do not need to scan the rest of the list – a success continuation will help me with this. If the current work is a part of the solution, here is how you would find the solution.

There are many paths through the list that I must explore. For a list of length 'm' s.t. m > 2, there are (m+1)*2 + 1 paths to explore. A fail continuation will help me in this…. Provided that the current path fails this is the work that I should do next.

So I set up the problem with an inner loop to recur on `l` with some sc and fc that might also change as I recurs.

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = ???
    myLoop( ??? )( ??? )( ??? )
}
```

Before I work on myLoop, I want to attempt to find the inputs to the initial call to myLoop.

- Initial value of l:List[Int] for myLoop
  - I could just write this to be Nil, since that is the base value for lists
  - I also have a data value l:List[Int] passed to hasSubList which is a list of variable legth
  - My gut tells me `l` should use the List[Int] passed to hasSubList.
  - I might be wrong, so later, if I can't get it to work I might change this
- sc: Int => Int
  - I need to construct a function from Int to Int. For simplicity I should write this as an anonomous lambda function (define it inline using the '=>' notation)
  - My function doesn't change types… What is the easiest function that says given something of a type return something of the same type?
  - The Identity function!
  - I'll write it as ' r => r ' for, given a residual 'r' return '=>' the residual 'r'
  - Again,I might be wrong, so later, if I can't get it to work I might change this
    - I could probably also implement this to do something with our data 'n'…
- fc : () => Boolean
  - I need another lambda that says given nothing return a Boolean
  - I need to return a Boolean. I have no data of type Boolean at the moment and the fc doesn't take any input so here I have 2 options, I can assume innocents or assume guilt.
    - Innocent : () => true
    - Guilt : () => false
  - Think about the base case of the problem. If the list is empty what do I know what to return? Well no… if l = Nil and n = 0 then I return true, else I return false.
  - Having said that, I usually return false... so lets assume guilt
  - () => false
  - Again, I might be wrong, so later, if I can't get it to work I might change this

Bringing that into the stub above I find:

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = ???
    myLoop( l )( r => r )( () => false )
}
```

now what about the body of myLoop. What data can I work with? I have a list. I know that it is often useful to check if my list is empty or not…

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil => ???
        case h :: t => ???
    }
    myLoop( l )( r => r )( () => false )
}
```

Lets start with the Nil case since that is probably easier. If the list is empty what do I do? Well, as stated earlier "*if l = Nil and n = 0 then I return true, else I return false*" directly coding this I find :

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil =>  n == 0     ||   false
        case h :: t => ???
    }
    myLoop( l )( r => r )( () => false )
}
```

And yes, I could just write n == 0… but that above stub is more helpful. The stub is wrong… Think about why it might not work.

Think about this, I bothered to set up continuations and I am not using that data. Can I re-write the body of case Nil => ??? the use sc and fc and still state the same logic of "*if l = Nil and n = 0 then I return true, else I return false*"?

I can, it looks like this :

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil =>  n == sc( 0 )    ||    fc()
        case h :: t => ???
    }
    myLoop( l )( r => r )( () => false )
}
```

BONUS: in addition to solving the base case of the empty list this also solves my inductive cases for reaching the end of a non-empty list! WOO!!!

Now what about the case h::t => ???

I have found an Int `h` that I might want to use. Potentially h denotes the end of the sublist s.t. the sum of the sublist is equal to n. Given the data, how would I ask the question of whether I am finished?

Look back near the top of the document consider the reason why I have constructed a success continuation and a fail continuation. Given the data, how would I ask the question of whether I am finished?

Consider this slight change for why I want a success continuation: As I scan the list, I might find a sub-list ending in some value h such that the sum of the sub-list is equal to n, in which case, I do not need to scan the rest of the list. Given the data, how would I ask the question of whether I am finished?

Absolutely, something like n == sc(h). sc of h should add h to some partially accumulated value

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil =>  n == sc( 0 )    ||    fc()
        case h :: t => n == sc( h )
    }
    myLoop( l )( r => r )( () => false )
}
```

Am I done? Does this work for all possible inputs to hasSubList? Of course not, why would it. All I've done is return things for empty lists and return things based on the first element of non-empty lists. I still need to exploit some natural recursion.

For lists, matched to h::t, natural recursion involves recusing the my function on t.

I might follow the same pattern as my Nil case and hope that works, i.e. I might need to say ||

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil =>  n == sc( 0 )    ||    fc()
        case h :: t => n == sc( h )  || myLoop( t )( ??? )( ??? )
    }
    myLoop( l )( r => r )( () => false )
}
```

so what should the continuations be?

- sc
    - as mentioned earlier this holds a partial sum, it's a function that says if I know the rest of the sum, here is what I would do with it
- fc
    - as mentioned earlier this states, if the current path doen't work then what should I do next?

- sc
    - it's a function from Int => Int
    - Let's name our input to represent a partial sum. I like the term 'acc'
    - So rather than ??? I can say acc => ??? and I have a little less work to do
    - As I look over the tail, what should happen?
        - acc
        - sc(acc)
        - h+acc
        - sc(h+acc)
- fc
    - it's a function from () => Boolean
    - () => false, () => true
    - something more complex?

- Sc : To use all the data I think acc => sc(h+acc) is my best bet
- Fc : To express that there is another path to try I think that () => myLoop(t) (???) (???) is the route I need to go

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
        def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
                case Nil =>  n == sc( 0 )    ||    fc()
                case h :: t => n == sc( h )  || myLoop( t )( acc => sc(acc+h) ){
                        () => myLoop(t)( ??? )( ??? )
                }
        }
        myLoop( l )( r => r )( () => false )
}
```

Again I need to define sc and fc for a call to myLoop. In solving the problem it is same to assume, that here, I am defining that the tail might be the start of the valid sublist. Do your best to solve this. See the solution at L6D2T2Q2. Before looking over the solution, test your work. A few tests are provided on the next page.

And finally, how can I check if this works?

I recommend painstakingly walking through one simple true case, and simple false case, then one complex case of your choosing.

Defined a few tests…

```
def tester(l:List[Int],n:Int,b:Boolean):Unit = {
        assert( hasSubList( l , n ) === b )
}

tester(List(1,2),1,true)
tester(List(1,2,3),0,true)
tester(List(1,2,3),1,true)
tester(List(1,2,3),2,true)
tester(List(1,2,3),3,true)
tester(List(1,2,3),4,false)
```

```
tester(List(1,2,3),5,true)
tester(List(1,2,3),6,true)
tester(List(1,2,3),7,false)
tester(List(1,2,3,4,5),9,true)
tester(List(1,2,3,4,5),8,false)
```

Do any of these tests seem wrong to you? (if so then you might not understand the problem fully)

Does our implementation work? (it should pass all of these tests)

Can you think of an edge case that will break our implementation?

SOLUTION

WITHOUT the constraint **You can only scan the list once.**

L6D2T3 Javascripty Test

If you are not already familiar with what regular expressions are, I highly recommend watching a YouTube video on the topic prior to this reading as in this reading I presume you all have a basic understanding of regular expressions.

NOTE: regular expressions is a white space sensitive language. Here I am ignoring this quality of regular expressions to make it a bit easier to view…

In lab 6 we are asked to implement a function named 'test' provided the following function definition:

```
def test(re: RegExpr, chars: List[Char])(sc: List[Char] => Boolean): Boolean
```

The goal is to return a literal true or false expressing whether chars exists in the language defined by re.

This all operates on a call to 'retest'. So we'll need to implement that. Make a guess at how to complete retest and refine the guess as you work on test. Retest is partially defined for us as :

```
def retest(re: RegExpr, s: String): Boolean = test(re, s.toList) { chars => ??? }
```

hint… I am converting my Sting to a List of Characters, the continuation takes as input a list of characters and should do work on that. I am writing a recursive function over a list of characters. Natural recursion has properties of consumption. This will inform the base case of sc in test, i.e. the sc applied to test when implementing rename.

Let us begin this discussion with *javascripty* literals (you can test these using node…)

/^ ! $/.test("")	false, the empty string "" does not exists (DNE) in the language of /^ ! $/

/^ ! $/.test(“a”)	false, the string “a” (DNE) in the language of /^ ! $/ because no strings exist in this language
/^ # $/.test(“”)	true, the empty string “” exists in the language of /^ # $/
/^ # $/.test(“a”)	false, the string “a” DNE in the language of /^ # $/ because only the empty string exists in that langauge
/^ a $/.test(“a”)	true, the string a exists in the language of /^ a $/
/^ a $/.test(“b”)	false, the string b does not exists (DNE) in the language of /^ a $/
/^ . $/.test(“a”)	true, the string a exists in the language of /^ . $/
/^ . $/.test(“b”)	true, the string b also exists in the language /^ . $/ because this is the language of all strings of length 1
/^ . $/.test(“ab”)	false, the string “ab” DNE in the language of /^ . $/ because it is a string of length 2

There are many others but I’ll let you figure out how those work on your own.

Let us write assertions that should hold true for each of these. I recommend writing the assertions to work off of our retest function. I recommend taking your best guess at how expressions parse and double checking that work, following the information provided to you in the lab worksheet.

```
//      /^ ! $/.test(“”)  should evaluate to false as stated above
val     re:RegExp       =       RNoString     // ! defined on page 5 of the handout
val     s:String        =       “”            // “” literally the string in test
val     b:Boolean       =       false
assert( retest(re,s) === b )
```

alternatively if I am not willing to guess and double check the ! is the RegExpr RNoString
I can import the reference parser and define re as :

```
val re:RegExp       =       RegExprParser.parse("!")
```

To go a step farther, when I am ready, I can write integration tests by using the parser we are implementing…

```
val RE( re:RegExp ):Expr      =       RegExprParser.parse("!")
```

Try writing the others on your own. I have provided a test script at the end of this topic.

Be careful when coding… we need to use the success continuation wisely example that informs how we write the REmptyString case :

/^ # $/.test(“a”)	false, the string “a” DNE in the language of /^ # $/ because only the

	empty string exists in that language
/^ #. $/.test(“a”)	true, the string “a” exists in the language of /^ #. $/ because this is the language of empty strings followed by the language of strings of length 1 i.e. this is the language of all strings of length 1.

A few edge cases to be concerned with, worth writing tests over are :

/^ !* $/ /^ #* $/ /^ ! | re $/ /^ re** $/

Let re be any regular expression

Semantically, these tests are NOT defined super well as they might be interpreted to be tests for iterateStep about the Call(GetField(e1,”test”),List(e2)) node but… hey, I never claimed to be perfect

Execute using MyTestSpecRunner.

```
class MyTestSpec(lab6: Lab6Like) extends FlatSpec {
  import Lab6Harness._
  import lab6._

  "  /^!$/.test(str_i)" should "return false" in{
    val re = RNoString
    val str_1 = ""
    val str_2 = "a"
    assert(!retest(re,str_1))
    assert(!retest(re,str_2))
  }
  " /^#$/.test( str_i ) " should "return true" in {
    val re = REmptyString
    val str_1 = ""
    assert(retest(re,str_1))
  }
  it should "return false" in {
    val re = REmptyString
    val str_1 = "a"
    assert(!retest(re,str_1))
  }
  "/^a$/.test( str_i )" should "return true" in {
    val re = RSingle('a')
    val str_1 = "a"
    assert(retest(re,str_1))
  }
  it should "return false" in {
    val re = RSingle('a')
    val str_1 = "b"
    val str_2 = ""
    assert(!retest(re,str_1))
    assert(!retest(re,str_2))
  }
  " /^.$/.test( str_i ) " should "return true" in {
    val re = RAnyChar
    val str_1 = "a"
    val str_2 = "b"
    assert(retest(re,str_1))
    assert(retest(re,str_2))
  }
  it should "return false" in {
    val re = RAnyChar
    val str_1 = ""
    val str_2 = "ab"
    assert(!retest(re,str_1))
    assert(!retest(re,str_2))
```

```
  }
}

class MyTestSpecRunner extends MyTestSpec(jsy.student.Lab6)
```

Solutions

L6D2T1Q1

1. Describe, in English, the language defined by this grammar
    a. S ::= a a a { a }
    b. This is the language of 3 or many a's. I have aaa followed by {a} and I know that {a} in EBNF applys the meta-symbols '{' '}' to something that needs to be repeated 0-or-many times, so in this case, that means 0 or many a's.
2. Construct a grammar with start symbol B that defines 0-or-many 'b' symbols
    a. B ::= { b }
3. Construct a grammar with start symbol A that defines 0-or-many 'a' symbols
    a. A ::= { a }
4. Construct a grammar with start symbol S that defines 0-or-many 'a' symbols followed by 0-or-many-b symbols 0-or-many times. If you will the language of binary over a and b rather than 0 and 1.
    a. Using 3 and 4
        i. S ::= { A B }
    b. From scratch
        i. S ::= { { a } { b } }
    c. In regular BNF (which is technically also EBNF)
        i. S ::= ϵ | aS | bS
        ii. NOTE: how I need the ϵ to create the concept of 0 characters, whereas the {} take care of that for me
5. Construct the grammar to define the language palindromes over 'a's and 'b's.
    a. This was sort of a trick question. EBNF's extra operator or meta-symbol if you will '{ }' allows me to create the concept of 0-or-many but that is not need or useful in constructing the language of palindromes.
    b. Having said that, technically all BNF grammars are written in EBNF as well since BNF is a sub-set of EBNF. On an exam or quiz I will be clear about what is expected.
    c. S ::= ϵ | a | b | a S a | b S b
    d. NOTE: this still uses the epsilon symbol, why?

L6D2T1Q2

EBNF – BNF(right-recursive) **EBNF** S ::= { a | b } **BNF** S ::= ϵ | a S | b S

L6D2T1Q3

Grammar re-writing process of constructing the reference grammar for the lab 1 parser, also representative logic of a 5 function calculator that has left associative parsing and precedence without () required

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

```
Atom        ::=   n | ( e )

e           ::=   PlusMinus
PlusMinus   ::=   TimesDiv { + TimesDiv | – TimesDiv  }
TimesDiv    ::=   Neg { * Neg | / Neg }
Neg         ::=   - Neg | Atom
Atom        ::=   n | ( e )

e           ::=   PlusMinus
PlusMinus   ::=   TimesDiv PlusMinusS
PlusMinusS  ::=   ϵ | + TimesDiv PlusMinusS | - TimesDiv PlusMinusS
TimesDiv    ::=   Neg TimesDivS
TimesDivS   ::=   ϵ | * Neg TimesDivS | / Neg TimesDivS
Neg         ::=   - Neg | Atom
Atom        ::=   n | ( e )
```

L6D2T2Q1 iterate on the list as many times as you'd like

NOT ACTUALLY CORRECT… but seems reasonable

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l:List[Int] , acc:Int ) : Boolean = l match {
        case Nil => acc==n
        case h::t => h+acc==n || myLoop(t,acc+h) || myLoop(t,acc)
    }
    myLoop(l,0)
}
```

CORRECT

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myInnerLoop ( l:List[Int] , acc:Int ) : Boolean = l match {
        case Nil => acc == n
        case h :: t => h+acc == n | myInnerLoop(t,acc+h)
    }
    def myOuterLoop ( l:List[Int] ):Boolean = l match {
        case Nil => myInnerLoop(l,0)
        case _::t => myInnerLoop(l,0) || myOuterLoop(t)
    }
    myOuterLoop(l)
}
```

L6D2T2Q2 iterate on the list only once

With continuations…

```
def hasSubList ( l :List[Int] , n : Int ) : Boolean = {
    def myLoop ( l :List[Int] )( sc: Int => Int )( fc : () => Boolean ) : Boolean = l match {
        case Nil => sc(0)==n || fc()
        case h :: t => {
            sc(h)==n || myLoop(t)( acc => sc(acc+h) )( () => myLoop(t)(acc => acc )(fc) )
        }
```

```
        }
        myLoop(l)( r => r )(()=>false)
    }
```

L6D3

- Continutation practice: Depth first search
- Grammars practice
- Test continued
- Beginning the parser

L6D3T1 Continuation practice: DepthFirstSearch (DFS)

DFS with continuations over a tree

Review documentation of dfs provided in the lab handout.

In performing dfs on a tree for some value, I could use continuations. Performing dfs on a tree for some function applied to an int accumulated over time is a more advanced problem that can follow the same logic of continuations and can actually be used to solve the original problem.

Our task is to complete the following function:

```
def dfs[A](t: Tree)(f: Int => Boolean)(sc: List[Int] => A)(fc: () => A): A = ???
```

t is the Binary tree I must scan from left-to-right **pre-order**

f is a function I must apply to all my data in the tree

sc – there are succeed early cases in this function. The success continuations state: if I knew the partial list that represents a part of the actual solution, this is the solution I will return

fc- in traversing a tree there are many paths to explore. The fc says, if the path you were on was not the correct path, this is what I want you to do next

You might start by writing the simple version of dfs on a tree that doesn't use continutations first, and learning from that.

Then try write the simple version of dfs on a tree with continuations.

Then attempt this actual problem (without and then with continuations).

Or, you can just take the challenge head on.

I highly recommend looking at the types of the data in the problem and using that to build your solution. There are only a few natural solutions from the types.

Recall that in recursion we want to write our base case, our case of interest and then our natural recursion case. Here the natural recursion case is the interesting part. See yesterday's notes on programming with continuations for examples of how to write natural recursion with success and fail continuations for examples of the structure.

For a tree that structure of the recursion case looks a bit like :

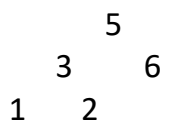
```
nameOfRecursiveFunction
        ( naturalRecursionOfDataStructurePart1 )
        ( newSuccessContinuation)
        ( newFailContinuation )
```

OR Rather

```
nameOfRecursiveFunction
        ( naturalRecursionOfDataStructurePart1 )
        ( newSuccessConintuation )
        ( () =>  nameOfRecursiveFunction
                        ( naturalRecursionOfDataStructurePart2 )
                        ( differentNewSuccessConintuation)
                        ( fc ) // provided from current call
        )
```

There is a test provided for you in the lab. I recommend reviewing it. There is one perhaps strange behavior here, that the path we accumulate is actually in the reverse order. Consider the tree:

Node(Node(Node(Empty,1,Empty),3,Node(Empty,2,Empty)),5,Node(Empty,6,Empty))



If 2 is in the tree show me the path from root to the first found instance of 2.

I would hope I can write the test with ease.

dfs(t)(d => d == 2)(path => Some(path))(() => None)

but this isn't entirely true for the example tree this returns Some(List(2,3,5)). My test might need to read

```
dfs(t)( d => d == 2 )( path => Some(path) )( () => None ) match {
        case Some(rpath) => path.foldLeft(Nil){(acc,h)=>h::ac}
        case None => None
}
```

L6D3T2 Grammars practice

Review the following re-writing of our lab 1 gramar

e ::= n | (e) | e+e | e-e |e*e | e/e| -e
n is a number
uses arithmetic precedence

e ::= A
A ::= B | A+B | A-B
B ::= C | B*C | B/C
C ::= D | -C
D ::= n | (e)

e ::= A
A ::= B { +B | -B }
B ::= C { *C | /C }
C ::= { - } D or C ::= D | -C is also allowed
D ::= n | (e)

e ::= A
A ::= B As
As ::= ϵ | + B As | - B As
B ::= C Bs
Bs ::= ϵ | * C Bs | / C Bs
C ::= Cs D or C ::= D | -C is also allowed
Cs ::= ϵ | - Cs
D ::= n | (e)

Repeat the process over the following grammar that describes regular expressions
re ::= ! | # | . | c | (re) | rere | re'|'re | re* | re+ | re? | re&re | ~re
c is some specific character

note that one of the | has quotes, on it, I am stating that this is a | in the object language regular expressions and not the meta-symbol of BNF

Use the following non-terminals while writing the first and second step :
re, union, intersect, concat, not, start, atom

When writing the 3rd step, add an ‘s’ to the end of the some non-terminals when creating new non-terminals

The precedence are defined on page 7 of the handout (along with quite a bit of valuable information)

We provide the expected definition of union in the EBNF grammar and the expected definition of both union and unions in the final grammar on page 8 of the handout.

L6D3T3 Test cont.

Note that in regex, white space does matter. Here I am acting as though it doesn’t for the sake of cleanliness.

Warm up, why is /^ #* $/ an edge case of the javascript test method of RegExpr?

Now, consider the nature of the Kleene-star and expression that use this of the form ‘re*’

‘re_1*’ says the pattern re happens 0-or-many times
‘re_1+’ says the pattern re happens 1-or-many times
‘re_1re_2’ says the pattern re_1 is immediately followed by re_2

Given the information above, can you define re_1+ in terms of the other operations?

If I told you it is possible both in math and in application, does that change your answer?

re_1re_1* or if you prefer re_1re_2 where re_2 is re_1*

Now if I say
‘re_1+’ is equivalent to re_1re_1*
‘#’ means that empty string, or that a pattern happened 0 times
‘$re_1|re_2$’ says that I should try re_1 and if that doesn’t work then I should try re_2

Given this information can you define re_1* recursively? Can you define re_1* using re_1*?

If I tell you it is possible to do this from the mathematic perspective, does this change your answer?

| re_1re_1* or if you prefer re_a | re_b where re_a is # and re_b is re_cre_d and re_c is re_1 and re_d is re_1*

Now is it true that # | re_1re_1* is mathematically equivalent to re_1* ?

Yeah, they are the indistinguishable from eachother.

In the application of this, in actually writing the code for test can I use this definition?

case (ReStar(re1),chars) => test(RUnion(REmptyString,RConcat(re1,RStar(re1))) , chars)

hmmm… maybe. Presuming the other things are implemented… What happens when re1 happens to be "#" by which I mean, what would happen in the following assertion (which is not provided in the lab)

assertResult(false){ retest(RStar(REmptyString)) , "hello") }

We have made something that works perfectly in theory (in the math) but in practice, might run forever without actually solving the problem. Note, there are other edge cases with the similar issues.

L6D3T4 Beginning the Parser

Prior to writing the parser it is recommended that you write the grammar for the parser detailed in writeup exercise 3.b.iv. of the lab handout

In writing the parser I highly recommend using test driven development and only linking 1 part of the parser at a time.

We provide the definition of 're' and 'union' as well as a stub for 'atom'. Without getting into the details of why this works (because you are all smart enough to figure that out for yourself) I recommend the following.

Implement the body of intersect as 'atom(next)'

Start writing tests over atomic expression

```
assertResult(     RE( RSingle('a') )    ){  JsyParser.parse("/^a$/") }
assertResult(     RE( RAnyChar )    ){  JsyParser.parse("/^.$/") }
assertResult(     RE( ??? )    ){  JsyParser.parse("/^#$/") }                    // find ???
assertResult(     RE( RNoString )    ){  JsyParser.parse("/^    ???     $/") }    // find ???
assertResult(     RE( RAnyChar )    ){  JsyParser.parse("/^(.)$/") }
assertResult(     RE( RAnyChar )    ){  JsyParser.parse("/^((.))$/") }
and more… Though I recommend saving tests such
```

Implement the logic to pass those test inside the atom function

Once that works, you should actually be passing union over atom cases as well

```
assertResult(     RE( RUnion(RSingle('a'),RSingle('b') )    ){  JsyParser.parse("/^a|b$/") }
assertResult(     RE( ??? )    ){  JsyParser.parse("/^.|#$/") }          // find ???
assertResult(     RE( ??? )    ){  JsyParser.parse("/^.|#|!$/") }        // find ???
```

You are now ready to actually implement something like intersect. Well, almost…

Write some tests for intersect first

```
assertResult(     RE( RIntersect(RSingle('a'),RSingle('b') )    ){  JsyParser.parse("/^a&b$/") }
```

```
assertResult(     RE( ??? )    ){  JsyParser.parse("/^.&#$/") }          // find ???
assertResult(     RE( ??? )    ){  JsyParser.parse("/^.&#&!$/") }        // find ???
```

Write the body of concat as 'atom(next)' and re-implement intersect to not say atom(next) but rather to implement the logic mandated by the grammar you have written for the lab writeup. Note that the grammar might look similar to something that is already implemented so you might make use of that.

You should now be able to pass tests over both union and intersect combined

```
assertResult(     RE( RIntersect(RUnion(RAnyChar,REmptyString),RNoString) )    ){
        JsyParser.parse("/^.|#&!$/")
}
```

From here, it is up pretty much up to you where you go next, the remaining cases are more challenging. And require an understanding of the work detailed here. Review what you have completed so far and attempt to understand how it all works.

L6D4

- Test Inference rules
- Union

L6D4T1 Test Inference Rules

To clarify what is being asked in question 3.c.i. of the lab 6 handout I would like you to write a set of inference rules that express how expressions of the form /^ re $/ and e1.test(e2) could be integrated to the lab 5 interpreter.

This will require you to write rules for both the step method using the lab 5 operational semantic over monads AND the type checker.

There are many ways to do this. Personally, I work from the bottom up, but sometimes this isn't the best way to learn.

Note that /^ re $/ is a value, and RegExp is a type

Note that the logic e1.test(e2) in javascripty is implemented in the 'test' function you wrote in this lab.

You will likely need 5 rules. 2 type rules. 2 search rules. And a single do rule. But you might find other creative ways to solve the problem and that's cool too.

To remind you how this process works, suppose I want to extend the language of *javascripty* from lab 5 to include the '++' operator apply to variables. This is a difficult task, so for now I will just extend the language to apply '++' to numbers (since that is also something useful to have in a language).

extend the grammar, only where needed

$$e ::= \ldots \mid ++\, e_1 \mid e_1\, ++$$

type checking, recall that my goal for now is simply to allow ++ to apply to numbers

$$\frac{\Gamma \vdash e_1 : \text{number}}{\Gamma \vdash {+}{+}\, e_1 : \text{number}} \qquad \frac{\Gamma \vdash e_1 : \text{number}}{\Gamma \vdash e_1 \,{+}{+} : \text{number}}$$

Or if you prefer to get fancy about it (though perhaps less readable and not the preferred solution)

$$\frac{\Gamma \vdash e_1 : \text{number} \qquad e \in \{\, e_1 \,{+}{+}\,,\, {+}{+}\, e_1 \,\}}{\Gamma \vdash e : \text{number}}$$

Now for step. I have single sub expressions that should become numbers I probably need a search rule and do rule for each.

Note that the searchUnary case could actually cover our search needs but I wrote extra cases anyway.

Note that the solution below is a tad odd, it has to do with the nature of e++ vs ++e.
e++ should increment e and return the old value of e, ++e will increment e and return the new incremented value. This becomes interesting when working with mutable variables. But I am not covering that here today.

$$\frac{\langle M , e_1 \rangle \longrightarrow \langle M' , e_1' \rangle}{\langle M , e_1 \,{+}{+} \rangle \longrightarrow \langle M' , e_1' \,{+}{+} \rangle} \qquad \frac{n' = n + 1}{\langle M , n \,{+}{+} \rangle \longrightarrow \langle M , n \rangle}$$

$$\frac{\langle M , e_1 \rangle \longrightarrow \langle M' , e_1' \rangle}{\langle M , {+}{+}\, e_1 \rangle \longrightarrow \langle M' , {+}{+}\, e_1' \rangle} \qquad \frac{n' = n + 1}{\langle M , {+}{+}\, n \rangle \longrightarrow \langle M , n' \rangle}$$

… Yes, functionally the step on n++ shouldn't *need* to create n' = n + 1, but technically the specification for e++ dictates that I should perform the task.

For an interesting thinking exercise, you might think about how to extend the language of lab 5 to include this concept, in addition to the concept of '++' applied to variables keeping the current notation for variables and noting that they have modes associated with them that effect what the significance of '++' should be.

But again, this is simply to demonstrate what is expected in question 3.c.i. of Lab 6 relating to declaring regular expressions and performing the re_1.test(e_2) operation on regular expressions agains strings. See earlier notes for example of this syntax.

If you are particularly careful you might be able to write the rules in less than 5 rules. HINT, it is not about condensing them, it is about one of them being ridiculous at the moment subject to the current

language. Note that for purposes of extensibility it is probably worth keeping the rule that I am thinking of. (… I know, vague right? …)

L6D4T2 Union

By now I expect that you have correctly coded most of the atomic cases of the parser and have attempted operators using the definition of union as a guide. So, I thought I would explain how I understand the definition of union with respect to the grammar provided to us. Please review this topic and see if this explanation agrees with you current understanding of how union works. If you think that you have a better way to explain it, and are comfortable sharing it, please do.

This discussion will take place without discussing the details of our type ParseResult[RegExpr] in great detail… because that is complicated and not actually needed to understand the problem. It is also easier to understand after you have already solved the problem

Example sentence and flow with goal output (ignoring whitespace)

```
# | . | a          RUnion(RUnion(REmptyString,RAnyChar),RSingle('a'))
    # | .          RUnion(REmptyString,RAnyChar)
        #          REmptyString
        .          RAnyChar
    a              RSingle('a')
```

Function definition provided with slight modifications to avoid some confusion from shadowed variables.

```
def union(next: Input): ParseResult[RegExpr] = intersect(next) match {
    case Success(r, nextp) => {
        def unions(acc: RegExpr, next: Input): ParseResult[RegExpr] = {
            if (next.atEnd) Success(acc, next)
            else (next.first, next.rest) match {
                case ('|', nextp) => intersect(nextp) match {
                    case Success(r, nextpp) => unions(RUnion(acc, r), nextpp)
                    case _ => Failure("expected intersect", nextp)
                }
                case _ => Success(acc, next)
            }
        }
        unions(r, nextp)
    }
    case _ => Failure("expected intersect", next)
}
```

grammar provided

union ::= intersect unions
unions ::= ϵ | '|' intersect unions

The grammar states that non-terminal Union is created via intersect unions. The fact is, union doesn't actually define that a '|' takes place. That is defined in union**s**. Consider the following partial implementation from the provided implementation.

```
def union(next: Input): ParseResult[RegExpr] =
    intersect(next) match {
        case Success(r, nextp) => ???
        case _ => Failure("expected intersect", next)
    }
```

the first thing that union asks, is was there an intersect. This seems reasonable since,
union ::= intersect unions

Note that the code matches on the result of intersect(next) to ask, was the work successful? If not, then the work failed, the pattern to match on might be something such as Failure(str,nextp). But we don't actually match on that pattern, because for this particular function we need to state that the error/the Failure was that I "expected intersect" when inside of 'next' and I was not able to find one.

Now what do we do if intersect succeeds? In calling intersect(next) I have found Success(r,nextp), where are r is the regular expression discovered intersect and nextp is the rest of the input.

Presuming intersect is correct down to atom, for the example sentence """# | . | a"""
    r == REmpty
    nextp == """| . | a"""

Now recall union ::= intersect unions

Since I found intersect I need to call unions. It turns out that it helps if unions is defined inside of union. And here is the partial implementation.

```
def union(next: Input): ParseResult[RegExpr] = intersect(next) match {
    case Success(r, nextp) => {
        def unions(acc: RegExpr, next: Input): ParseResult[RegExpr] = ???
        unions(r, nextp)
    }
    case _ => Failure("expected intersect", next)
}
```

I define unions to be able to accumulate a regular expressions given a starting RegExpr and a beinging Input.

for the example sentence """# | . | a""", the first call to unions has input

acc = REmpty
next = “””| . | a”””

Now how is unions going to be defined? See the reference grammar `union ::= $\epsilon$ | ‘|’ intersect unions` Either it is empty or its not, if it’s not I hope that the first element is ‘|’, if that isn’t the case then I might have succeeded, in that I didn’t need to find ‘|’ (the reason why this is true is rather complex. After you have a better understanding of the parser I encourage you to think about why it will work – the short version is : this is a side step in the recursion and so unlike finding an initial intersect the ‘|’ is not required for all sentences and so I return a Success rather then a failure… But I succeed without consuming anything).

Defining the problem in prose

```
def unions(acc: RegExpr, next: Input): ParseResult[RegExpr] = {

    // if it’s empty then I’m done with unions

    // otherwise I need to ask more questions

        // I might have found ‘|’ to use to construct an RUnion with recursion

        // I might not have found a ‘|’ which means that I am done with unions
        // This doesn’t mean that I have failed at anything

}
```

Using some syntax to replace the prose

```
def unions(acc: RegExpr, next: Input): ParseResult[RegExpr] = {

    if (next.atEnd) ???

    else (next.first, next.rest) match {

        case ('|', nextp) => ???

        case _ => ???

}       }
```

Now how should I fill in the implementation? Consider the table of desired behavior for unions. See if you can use that to code unions without looking at the implementation provided to you. Then check your work. Repeat the process until you are correct.

i : the time I am calling unions
acc : the partial RegExp accumulated thus far
next : the partial Input that I have not looked at yet (a great name for this is suffix)

example sentence to union “””# | . | a”””

should call unions with the input at time 0

i	acc	next	next.isEmpty	next.first == ‘|’
0	REmptyString	“””|.|a”””	False	True
1	RUnion(REmptyString,RAnyChar)	“””|a”””””	False	True
2	RUnion(RUnion(REmptyString,RAnyChar),RSingle(‘a’))	“””””””	True	N/A

It returns RUnion(RUnion(REmptyString,RAnyChar),RSingle(‘a’))

Here is an extended example
example sentence to union “””# | . | a | ! | d”””

should call unions with the input at time 0

i	acc	next	next.isEmpty	next.first == ‘|’
0	REmptyString	“””|.|a|!|d”””	False	True
1	RUnion(REmptyString,RAnyChar)	“””|a|!|d”””	False	True
2	RUnion(RUnion(REmptyString,RAnyChar),RSingle(‘a’))	“””|!|d”””	False	True
3	RUnion(acc2, RNoString)	“””|d”””	False	True
4	RUnion(acc3, RSingle(‘d’))	“””””””	True	N/A

Returns RUnion(acc3, RSingle(‘d’))

i.e. RUnion(RUnion(acc2, RNoString), RSingle(‘d’))

i.e. RUnion(RUnion(RUnion(RUnion(REmptyString,RAnyChar),RSingle(‘a’)), RNoString),RSingle(‘d’))

An example where nextp is not an intersect and we fail inside unions

example sentence that should fail “””# | & .”””

should call unions with the input at time 0

i	acc	next	next.first == ‘|’	nextp	Intersect(nextp)
0	REmptyString	“””|&. ”””	True	“””&.”””	Failure(“expected concat”, “””&.”””)

Returns Failure(“Expected Intersect”,”””&.””””)

And finally an example where the next.first != “|” comes into play and we succeed. It involves the difficult problem from atom being the ‘(‘ ‘)’ s. anything with parens is enough but I’ll show a test that is somewhat more interesting

example sentence to union “””(a|b)””” should become RUnion(RSingle(‘a’),RSingle(‘b’)

re(“””(a|b)”””) → union(“””(a|b)”””) → intersect(“””(a|b)”””) →* atom(“””(a|b)”””) →*
re(“””a|b)”””) → union(“””a|b)”””) → intersect(“””a|b)”””) →* **unions(RSingle(‘a’), “””|b)”””)**

should call unions with the input at time 0

i	acc	next	next.isEmpty	next.first == ‘|’
0	RSingle(‘a’)	“””	b) ”””	False	True
1	RUnion(RSingle(‘a’),RSingle(‘b’)	“””) ”””	False	False

Unions returns Success(RUnion(RSingle(‘a’),RSingle(‘b’), “””)”””) and if other things are coded correctly then we will eventually get the expected RegExpr RUnion(RSingle(‘a’),RSingle(‘b’)

L6: Additional Resources

L6: Closing Thoughts