1. Introduction

The article first introduces a basic Boolean matcher that uses a shift function and two auxiliary functions, nullable and final, to "move" markers through a regular-expression tree for each character in the input string. Each node in the tree contains a Boolean flag indicating whether it is marked.

2. The shift Function

The shift function takes three arguments:

1. A mode (Boolean) that indicates whether a new mark should be introduced into the node.
2. The regular expression to be marked.
3. The current character from the input string.

Initially, no nodes are marked. At the first character, we call shift with mode = True so that a new mark can be introduced. For subsequent characters, mode = False is used, which means we only shift existing marks without introducing new ones (unless certain conditions within the regular expression require a new mark to be added). When processing each character, the tree is traversed and the following rules apply:

- **Character node:** If the node's character matches the current input character, and the mode is True, then we mark this node.(i.e., set its value to true in a Boolean setting).
- **Alternative node (r1 + r2):** We shift marks of the left subexpression (r1) and the right subexpression (r2) and combine their results. The node is considered marked if either side is marked.
- **Sequence node (r1 · r2):**
    1. We shift a mark into r1 using the current mode.
    2. For r2, we check:
        - If r1 is nullable (i.e., can match the empty string) and the mode is true, then we shift a new mark into r2.
        - If r1 is final (i.e., it contains a marked final character), we shift a new mark into r2.
        - Otherwise, we only shift existing marks through r2 (mode = False).

        The sequence is fully marked only if both parts are marked.
- *Star node (r)**: We shift a new mark if the inner expression is final or if the mode indicates a new mark should be introduced (mode = True). This accounts for the possibility of matching the star subexpression multiple times.

3. Extending to Weighted Regular Expressions

So far, this is a Boolean matcher, meaning each node's mark is true or false depending on the match. Later, the article extends this idea to *weighted regular expressions* by replacing Boolean values with elements from a *semiring*. A semiring is an algebraic structure with zero, one, an addition operator (⊕), and a multiplication operator (⊗). The article demonstrates how to substitute booleans with a semiring to produce extra information about matches.

3.1 Boolean Semiring

The Boolean semiring is defined as:

```
Instance Semiring Bool where
zero=false
one=true
(⊕) = (∨)
(⊗) = (∧)
```

This acts like the original Boolean matcher (where false corresponds to zero, and true corresponds to one). However, it also enables more complex structures to provide additional information about the match.

3.2 Int Semiring

the article defines:

```
Instance Semiring Int where
zero=0
one=1
(⊕)  = (+)
(⊗)  = (*)
```

Using Int, we can count the number of possible matches. I haven't fully tested this part yet, but I plan to try merging it with the marker-based matcher as they only use it with basic matcher that uses accept function.

3.3 Leftmost Semiring

The article then present semirings like **Leftmost**, which tracks the leftmost match position in a string. This semiring (shown below, Figure 1) stores the position of characters in the input. It returns zero if matching fails, and one if we're ignoring a character (like when the expression is nullable). When combining two matches with(⊕) or an alternative , it picks the leftmost position (smallest index), and when combining them in a sequence (⊗), it takes the start of the first sub-match unless the first is ignored.

Figure 1

```
given semiringLeftmost: Semiring[LeftmostT] with {
def zero = NoLeft
def one = Leftmost(NoStart)

def plus(a: LeftmostT, b: LeftmostT): LeftmostT = (a,b) match{
case (NoLeft, x) => x
case (x, NoLeft) => x
case (Leftmost(i), Leftmost(j)) => Leftmost(leftmost(i, j))}

def times(a: LeftmostT, b: LeftmostT): LeftmostT = (a,b) match {
case (NoLeft, _) => NoLeft
case (_, NoLeft) => NoLeft
case (Leftmost(x), Leftmost(y)) => Leftmost(start(x, y)) }

def start(i: StartT,j:StartT): StartT = (i,j) match {
case (NoStart, s) => s
case (s, _) => s}

def leftmost(i: StartT,j:StartT): StartT = (i,j) match {
case (NoStart, NoStart) => NoStart
case (NoStart, Start(i)) => Start(i)
case (Start(i), NoStart) => Start(i)
case (Start(i), Start(j)) => Start(min(i, j)) }
}
given semiringILeftmost: SemiringI[LeftmostT] with {
export semiringLeftmost.*// Inherit all `Semiring` operations
def index(i: Int): LeftmostT = Leftmost(Start(i))}
```

3.4 Left-Longest Semiring

Similarly, **LeftLong** (shown below, Figure 2) finds the leftmost longest match by comparing both start and end positions. It ensures the longest possible match when multiple subexpressions start at the same position. It uses a range (start, end) for each match. With the alternative(⊕), it picks whichever match is leftmost, and if they share the same start, it picks the one with the greater end position. With sequence (⊗), it combines the start index from the first expression with the end index from the second to ensure the longer match.

Figure 2

```
given semiringLeftlong: Semiring[LeftlongT] with {
def zero = NoLeftLong
def one = LeftLong(NoRange)

def plus(a: LeftlongT, b: LeftlongT): LeftlongT = (a,b) match{
case (NoLeftLong, x) => x
case (x, NoLeftLong) => x
case (LeftLong(x), LeftLong(y)) => LeftLong(leftlong(x, y))}

def times(a: LeftlongT, b: LeftlongT): LeftlongT = (a,b) match {
case (NoLeftLong, _) => NoLeftLong
case (_, NoLeftLong) => NoLeftLong
case (LeftLong(x), LeftLong(y)) => LeftLong(range(x, y))}

def leftlong(x: RangeT,y:RangeT): RangeT = (x,y) match {
case (NoRange, NoRange) => NoRange
case (NoRange, Range(i,j)) => Range(i,j)
case (Range(i,j),NoRange) => Range(i,j)
case (Range(i,j), Range(k,l)) => {
if(i<k || (i == k && j >= l)) Range(i,j) else Range(k,l) }
}

def range(x: RangeT,y:RangeT): RangeT = {(x,y) match {
case (NoRange, NoRange) => NoRange
case (NoRange, range) => range
case (range, NoRange) => range
case (Range(i,_), Range(_,j)) =>  Range(i,j)} }
}
given semiringILeftlong: SemiringI[LeftlongT] with {
export semiringLeftlong.*// Inherit all `Semiring` operations
def index(i:Int): LeftlongT = LeftLong(Range(i,i))
}
```

All these semirings use the same shift-based algorithm described for the Boolean case. The difference is how they interpret (⊕) and (⊗). Booleans just do true/false logic, but other semirings gather more detailed information for subword matches. By changing the semiring, we can use a Boolean matcher or find the leftmost/left-longest matches without modifying the underlying shift algorithm.

March 10th

- I tried to construct ntimes. My initial idea was to think of it similarly to star, but with a counter, and this seems to be working so far when testing it with one ntimes constructor.
- The issue i faced occurred when testing it on the 'evil regex SEQ(NTIMES(OPT(CHAR('a')), n), NTIMES(CHAR('a'), n))'. The main problem is that the first and second 'ntimes' instances process each input separately, marking themselves simultaneously without sharing information. Since both 'ntimes' are in a sequence, the marks of a sequence gets shifted in both ntimes at each input character. In the example I used, the input character marks both instances of 'ntimes' simultaneously causing them to incorrectly counting the match. The input string ‘aaa’ for example will cause the current implementation to count 3 marks for both ntimes and gets rejected but it shouldn’t.
- I explored using integers as markers to address this issue, but haven't had success yet.
- I attempted implementing mkeps in a similar way to how it was done in the case of derivatives, aiming to gain new insights. However, When applied, it only finds the marked nodes (needs tweeking in terms of handeling nullables as well) and can calculate values that leads to the marked nodes but for now it loses the information of previously marked nodes so it is not complete yet.
- I attempted to implement the mkeps function in a similar way to how it was done in the case of derivatives, aiming to gain new insights. However, when applied, it only finds the marked nodes of the final marked regex and can calculate values that lead only to the marked nodes, but it loses the information of previously marked nodes, so it is not complete yet. Additionally, it needs tweaking in terms of handling nullables as well.
- With Dr. Christian’s suggestion of using stacks as markers for 'ntimes', I tried applying it to the markers of the base case of 'char'. I used a list as a stack and found that the list could be useful in helping to see how the algorithm generated the final regex or how it matched it. Each level of the stack represents the marks at the point of each input character from the original string.
- Maybe this can be used in mkeps_marked to generate the correct values and also help with the inj function.
- For now, the below is the mkeps function and also an imitation of the derivatives' inj, but modified slightly for the marked approach.

mkeps_marked for Marked Regular Expressions

The function should compute the value that explains how a marked regex matched the string based on the position of the marks, for now it is only uses the final marks without checking the ‘erased’ marks during the processing of the string.

	Input r	Output
mkeps_marked (r) {	If r = ONE	EMPTY
	If r= CHAR(c, marked), marked >=1 If r= CHAR(c, marked), marked <1	CHARV(c) UNMARKED(c)
	If r= r1+r2 If fin(r1) fin(r2) and fin(r1) != 1 // needs tweaking Otherwise	LEFT(mkeps_marked(r1) RIGHT(mkeps_marked(r2) UNMARKED("ALT")
	If r= r1 . r2	SEQV(mkeps_marked(r1) , mkeps_marked(r2))
	If r= star(r1)	STARV([mkeps_marked(r1)])
	If r= star(ONE)	STARV([])
	//needs tweaking If r= ZERO	ZEROV

- It builds a value (Left, Right, Seq, Stars) according to the placement of marks.

	Input	Output
inj_marked(r , char, value) {	inj_marked(•c,c,Empty)	:= Char(c)
	inj_marked(r1+r2,c,Left(v))	Left(inj_marked(r1,c,v)
	inj_marked(r1+r2,c,Right(v))	Right(inj_marked(r2,c,v)
	inj_marked(r1. r2,c,Seq(v1 ,v2)) if fin(r1) && nullable(r2) Otherwise	Seq(inj_marked(r1 ,c,v1),v2) Seq(v1 , inj_marked(r2,c,v2)
	inj_marked(r*,c,Stars(v1,v2,….,vn))	Stars(inj_marked(r,c,v1) :: v2,…,vn)
	inj_marked(r*,c,Stars([]))	Stars([inj_marked(r,c,Empty)])

March 11th

- Similar to bitcodes, an NTIMES could signal: 1 for continue, 0 stop, 2 for the possibility of continuing with other NTIMES/star nodes indicating that the NTIMES is at an accepting state but possible characters could be consumed. If the ntimes can accept more characters in the case of nullable then don't add to the number of markers but add to a counter and change the state to 0 instead of 2.
- Maybe a solution is to check if there's an alternative way to match the previous input characters/markers rather than relying on this NTIMES. This could be problematic as it might involve backtracking.

March 14th - 15th March

- I experimented with using a stack for the marks, my original intention was to experiment for the NTIMES constructor, but I decided to implement a function extractStage and exctractStages. Basically they take a marked regular expression and extract the different stages of the marks at each input character.
- Attempted to create unshift function; at first I implemented a function unshift to reverse the effects of the shift function, basically reversing the conditions in the char case so it will check if a character match then returns false instead of true otherwise keep the mark. Second, I used the implementation of marks as a stack (a list treated as a stack) that I have tried earlier for the ntimes, and this basic version of unshift just removes the head of the marks stack and returns the regular expression. This version might needs a bit of tweaking interns of preventing the removal of marks indefinitely.
- Two issues found in *NTIMES* experiment code (imitating the derivatives approach for NTIMES, SEQ):
    1. First, when testing the simple regex *NTIMES(ALT(CHAR(a),ONE), 2)*: it results to *false* due to *n* not decrementing if tested with string of one 'a'. in this case, the solution to use a flag or state to indicate that *NTIMES* didn't consume n characters but at accepting state could work. The *fin* function for *NTIMES* may need updating. For example, in the case where *counter < n && nullable(r)* for the int mark, or just *nullable(r)* for the boolean mark. should it return fin(r)?
    2. In the case of the following regex: reg=*SEQ(NTIMES(OPT(CHAR('a')), 2), NTIMES(CHAR('a'), 2))* : Starting with the first character 'a', the shift function will test the BSEQ constructor and produce a BALT. This is due to the code that imitates the derivatives by evaluating *(mark && nullable r1)* which evaluates to true in the case of the first 'a'. "aabjnbjdnj

       The regular expression will be the following after consuming the second 'a'

       ```
       BALT(
        BSEQ(BNT(BALT(BCHAR(true, 'a'), BONE), 0), BNT(BCHAR(false, 'a'), 2)),
        BNT(BCHAR(true, 'a'), 0)
       )
       ```

       Then, after consuming 'b' (third character), the regular expression becomes:

       ```
       BALT(
        BSEQ(BNT(BALT(BCHAR(true, 'a'), BONE), 0), BNT(BCHAR(false, 'a'), 2)),
        BNT(BCHAR(true, 'a'), 0)
       )
       ```

       I believe this happens because the BSEQ will remain evaluated as *false* after consuming the three input characters. However, the overall BALT will still be true due to the second branch BNT(BCHAR(true, 'a'), 0), which will remain unchanged since it will just return BNT with every additional character.

       Update:

        1. For the first issue, an edit was made to fin to handle the case where counter < n when r is nullable. This change appears to be working so far. The edit is shown below:

           ```
           if(n == counter || (counter<n && nullable(r) == 1))
           fin(r) else 0
           ```

That code is suited for the int mark, for the boolean mark, it might look like:

case BNT(r, n) => if (n == 0 || (nullable(r)) fin(r) else false

2. For the second issue, the current implementation of boolean NTIMES (BNT) does not shift any markers after NTIMES has completed. This allows NTIMES to be accepted for strings that start with valid characters but then ignore subsequent characters.

March 16th

Bitcodes implementation

• I spent time working on creating an implementation that imitates the bitcodes approach of extracting values with mkeps and rebuilding the matched regular expression using decode. The function mkeps_marked was updated to follow the specification of the bitcodes approach but with some modification to make it work with the shift algorithm, additionally the function intern was updated to create the regular expression with bit sequences. The functions fuse and decode were also implemented in a similar manner .

- Implemented mkeps_marked2, which tracing the path to the final marked node and returns the bitcode.
- Fuse was implemented to 'fuse' bitcode lists as was with the bitcodes way.
- Intern was implemented to include bitcode sequences with regular expression.
- Decode now reconstructs values based on bit sequences, but there are still issues when handling the last bits—it needs to correctly resolve the cases where the bit sequence is exhausted (one last bit remaining).
- Still the implementation is not finished, especially in decode, where handling of final bits need further work. However, it's a good starting point to explore how the bitcode approach can be useful in the marked approach.
- The code for the new functions is below, from the file marked-BitcodeTest.sc:

```
def mkeps_marked2(r: Rexpb): List[Int] = r match {
    case ONEb(bs) => bs
    case CHARb(_, marked, bs) => if (marked) bs else List()
    case ALTb(r1, r2, bs) =>
        if (fin(r1)) bs ++ mkeps_marked2(r1)
        else if (fin(r2)) bs ++ mkeps_marked2(r2)
        else bs
    case SEQb(r1, r2, bs) =>
        if (fin(r1) && nullable(r2)) bs ++ mkeps_marked2(r1) ++ mkeps_marked2(r2)
        else if (fin(r2)) bs ++ mkeps_marked2(r2)
        else bs
    case STARb(r, bs) =>
        if (fin(r)) bs ++ mkeps_marked2(r)
        else bs
    case NTIMESb(r, n, nmark, counter, bs) =>
        if (counter == n && fin(r)) bs ++ nmark ++ mkeps_marked2(r)
        else bs
    case INITb(r, bs) => mkeps_marked2(r)
    case ZEROb => List()
}
 def fuse(cs: List[Int], r: Rexpb): Rexpb = r match {
    case ZEROb => ZEROb
    case ONEb(bs) => ONEb(cs ++ bs)
    case CHARb(c, marked, bs) => CHARb(c, marked, cs ++ bs)
    case ALTb(r1, r2, bs) => ALTb(fuse(cs, r1), fuse(cs, r2), cs ++ bs)
    case SEQb(r1, r2, bs) => SEQb(fuse(cs, r1), fuse(cs, r2), cs ++ bs)
    case STARb(r, bs) => STARb(fuse(cs, r), cs ++ bs)
    case NTIMESb(r, n, nmark, counter, bs) => NTIMESb(fuse(cs, r), n, nmark, counter, cs ++ bs)
    case INITb(r, bs) => INITb(fuse(cs, r), cs ++ bs)
}
```

```
def decode(r: Rexpb, bs: List[Int]): (VALUE, List[Int]) = r match {
  case ONEb(_) => (EMPTY, bs) // not sure this should be included
  case CHARb(c, _, _) => (CHARV(c), bs) // (2) decode (c) bs = (Char(c), bs)
  case ALTb(r1, r2, _) => bs match {
    case 0 :: bs1 => // (3) decode (r1 + r2) 0 :: bs =  (Left(v), bs')  where decode r1 bs => v,bs'
        val (v, bsp) = decode(r1, bs1)
        (LEFT(v), bsp)

    case 1 :: bs1 => // (4) decode (r1 + r2) 1 :: bs = (Right(v), bs') where decode r2 bs => v,bs'
        val (v, bsp) = decode(r2, bs1)
        println(s"decode r1: $v  and bsp= $bsp")
        (RIGHT(v), bsp)
    case bs1::Nil =>
        if(bs1 == 0){
            val (v, bsp)= decode(r1, bs1::Nil)
            (LEFT(v), bsp)
        }
        else {
            val (v, bsp)= decode(r2, bs1::Nil)
            (RIGHT(v), bsp)
        }
    case _ => (EMPTY, bs)
         // in case of something else, may need to remove it but just incase
  }

  case SEQb(r1, r2, _) =>  // (5) decode (r1 · r2) bs = (Seq(v1, v2), bs3) where decode r1 bs => v1,bs2 and decode r2 bs2
=>v2,bs3
    val (v1, bs2) = decode(r1, bs)
    val (v2, bs3) = decode(r2, bs2)
    (SEQV(v1, v2), bs3)
  case STARb(r, _) => bs match {
    case 1 :: bs1 => // (6) decode (r*) 1 :: bs = (Stars [], bs)
        (STARV( List() ), bs1)

    case 0 :: bs1 =>   // (7) decode (r*) 0 :: bs = (Stars (v :: vs), bs')
      val (v, bs2) = decode(r, bs1)
      val (STARV(vs), bsv) = decode(STARb(r,bs2), bs2) // Recursive case for matching multiple times
      (STARV(v :: vs), bsv)

    case _ => (STARV(List()), bs) //  maybe not needed
  }

  //case _ => (EMPTY, bs) // maybe not needed
}

def intern2(r: Rexp) : Rexpb = INITb(intern(r),List())

def intern(r: Rexp) : Rexpb = r match{
  case ZERO => ZEROb
  case ONE => ONEb(List())
  case CHAR(c,marked) => CHARb(c,marked,List())
  case ALT(r1, r2) =>
    ALTb(fuse(List(0), intern(r1)), fuse(List(1), intern(r2)), List())
  case SEQ(r1, r2) => SEQb(intern(r1),intern(r2),List())
  case STAR(r) => STARb(intern(r),List())
  case NTIMES(r, n, nmark,counter) => NTIMESb(intern(r),n,nmark,counter, List())
}
```

March 19th

- The code provided by Dr. Christian can help in testing by enumerating regular expressions and strings, including both matching and non-matching cases. This allows comparing the results of proven algorithms (e.g., Derivatives or simple marked) with any new code for an improved marked algorithm.

Continuo - Bitcodes:

- We can explore methods to record all constructors, enabling us to reconstruct the regular expression that matched the string only from the bit code, without needing the original regular expression to decode the bits
- Bitcodes could potentially be included in shift, allowing us to later extract them using mkeps and decode them to generate values. For example, each marked CHAR could store a sequence of bits indicating how it was marked. Then, in mkeps, we check if r is final and return the bit sequence that traces the path of how the nodes in r were marked.

Current implementation of Bitcodes:

Decode should receive a normal regular expression (`Rexp`) rather than one containing bits (Rexpb) . **Fixed**.

There are two main issues with implementing bitcodes for the marked approach:

1. The SEQ case doesn't indicate how it was matched, it simply returns a SEQV (v1,v2). For example if the regular expression is ALT(a,SEQ(a,b)) and the input string is "ab", then the return value is RIGHT(SEQV(a,b)) .However, this doesn't specify whether the match is due to r1 alone or both r1 and r2.
2. When there are repetitions, previously marked nodes are overwritten. With each shift, nodes might become unmarked, wiping out the marks that should generate values and leaving only the last marks. For instance, in STAR(r), if r is repeated twice, the bitcode will only record how r was matched in the last repetition, losing the history of the previous match.

Update:

1. So far, the ALT constructor works, possibly due to the fusing that occurs in the intern function. A similar approach might be needed for SEQ to indicate exactly how it was matched. We could differentiate which part of the sequence was matched in mkeps by using fin checks.

   If fin(r1) & nullable (r1) & ! fin(r2) then return mkeps r1 otherwise return mkeps r1+ mkeps r2

   In decode, we may need a flag, similar to ALT, to produce a value that states whether the sequence was matched due to r1 alone or both r1 and r2.
2. Maybe the bitcode for repetitions should keep track of how each repetition matched the string to prevent marks being wiped. A possible solution is to fuse the bitcode at each repetition if there is a match? This way, instead of overwriting previous marks, each repetition contributes to the final bitcode, keeping a full history of how the matches occurred across multiple repetitions.

———————————————————————————————

March 20th

Point constructor:

- Point constructors uses int for marks, whereas regular constructors (CHARs) are not marked. You can wrap constructors with Point.
- With this constructor, we may be able to implement suspended marks - marks that are temporarily halted under certain conditions and resumed later.
- For example, when r in NTIMES reaches a final state, we can mark it by wrapping r with Point. Alternatively, when the count of NTIMES reaches n, we can wrap the entire NTIMES constructor with Point.

Current implementation of the Point constructor:

NTIMES with the point constructor:

- I first tried to wrap a Point around NTIMES when it met the final conditions—then at fin, it would be accepted if it was inside a Point constructor. But the same issues as before appeared.
- I also tried wrapping r in a Point each time r was final, but to no avail—it had the same problems as before.
- I had an idea of using Point with a position, so that when NTIMES is final, the Point wrapping it could store the position. Maybe a comparison of positions might help?

Point Constructor and int:

- I experimented with using List[Int] with the Point constructor, and adjusted the shift function to receive the character and the position from the input string.
- I did this to record the history of marked chars. shift now wraps a Point around a marked char with the position added to the List[Int]. If the marked char is in shift again (already wrapped in a Point), then it adds 0 to the list to indicate that this char (Point-wrapped) is unmarked.
- This allowed me to first create a simple function that I called extractPoints, which extracts all the points of a final marked regular expression. The output is something like: *CHAR(b) with index = 1*, where the index is the position of the character from the input string that marked this part of the regular expression.
- Then I created a function called ***popPoints***, which, when applied to the final regular expression, pops out the top mark of the list in a Point node and returns two regexes:
    - the regex with this node updated by the top mark (since it could still be marked or unmarked)
    - the regex with the remaining marks (retaining the rest so that it can be applied again).
- Because the history of the marks is recorded, I was able to extract the regular expression at each stage of the input string. The functions for now need to be called for the length of the input string (this is done in my code by a helper function).
- The code for the new functions is below, from the file play_point2.sc:

```
// in shift

case CHAR(d) => if(m && d == c) POINT(CHAR(d), List(pos+1)) else CHAR(d)
case POINT(CHAR(d),tag) => if(m && d == c) POINT(CHAR(d), pos+1 ::tag ) else POINT(CHAR(d), 0::tag)

def extractPoints(r: Rexp): List[(Rexp, Int)] = r match {
  case POINT(r, pos) => pos.filter(_ != 0).map(p => (r, p-1)).reverse
  case ALT(r1, r2)   => extractPoints(r1) ++ extractPoints(r2)
  case SEQ(r1, r2)   => extractPoints(r1) ++ extractPoints(r2)
  case STAR(r)       => extractPoints(r)
  case NTIMES(r, _, _) => extractPoints(r)
  case _             => List()
}

// helper function , popPoints is in the next page
def traverseStages2(r: Rexp, inputLength: Int): List[Rexp] = {
  val (stages, _) = (0 until (inputLength + 1)).foldLeft((List(r), r)) { case ((acc, state), _) =>
    val (currentStage, nextState) = popPoints(state)
    (acc :+ currentStage, nextState)
  }
  stages
}
```

```
def popPoints(r: Rexp): (Rexp, Rexp) = r match {
  case ZERO => (ZERO, ZERO)
  case ONE  => (ONE, ONE)
  case CHAR(c) => (CHAR(c), CHAR(c)) // Return the same character for both current and next state because it doesnt have
marks or points
  case POINT(inner, tags) =>
    val (currentInner, nextInner) = popPoints(inner)
    if (tags.isEmpty) {
     (currentInner, nextInner)
    } else {
     val updatedNext = POINT(inner, tags.tail)
      tags.head match {
        case 0 =>
         (currentInner, updatedNext)
        case x =>
          (POINT(currentInner, List(x-1)), updatedNext)
    }}
  case ALT(r1, r2) =>
    val (curr1, next1) = popPoints(r1)
    val (curr2, next2) = popPoints(r2)
    (ALT(curr1, curr2), ALT(next1, next2))

  case SEQ(r1, r2) =>
    val (curr1, next1) = popPoints(r1)
    val (curr2, next2) = popPoints(r2)
    (SEQ(curr1, curr2), SEQ(next1, next2))

  case STAR(inner) =>
    val (curr, next) = popPoints(inner)
    (STAR(curr), STAR(next))

  case NTIMES(inner, n, counter) =>
    val (curr, next) = popPoints(inner)
    (NTIMES(curr, n, counter), NTIMES(next, n, counter))
}
```

March 21st

Point Constructor and Bitcodes:

- Similar to what was done in implementing the bitcodes approach (p. 7, 8, 9), I tried the same structure but with the Point constructor. ***mkeps*** was kept fairly similar to the previous implementation. The changes were made in the case of ***CHAR***, which now returns an empty list since it represents an unmatched character.
- I also added a case for ***POINT***, which represents a matched character—it now returns the bit code of the matched or marked (Points) in the regular expression.
- A change was also made in the case of ***STAR***, particularly to keep the history of how that ***STAR*** was matched. With each iteration, it appends how each individual inner r of the ***STAR*** was matched (its ***mkeps***), then appends a 1 to indicate the end of an iteration.
- The code for the new ***mkeps*** function is below as well as the code for ***decode*** and ***fuse*** functions which were kept almost the same, from the file play_point3.sc:

```
def mkeps(r: Rexp): List[Int] = r match {
    case ONE => List()
    case CHAR(_, bs) => List()
    case POINT(CHAR(_, b),tag, bs) => bs
    case POINT(r,tag , bs) => bs
    case ALT(r1, r2, bs) =>
        if (fin(r1))  bs ++ mkeps(r1)
        else if (fin(r2)) bs ++ mkeps(r2)
        else List() //bs
    case SEQ(r1, r2, bs) =>
    if (fin(r1) && nullable(r2)) bs ++ mkeps(r1) ++ mkeps(r2)
    else if (fin(r2)) bs ++  mkeps(r2)
    else List() //bs
    case STAR(r, bs) =>
        //r match case point add zero else 1 ? also tag?
        if (fin(r)) bs ++ mkeps(r) ++ List(1)
        else List()
    case NTIMES(r, n, counter, bs) =>
        if (counter == n && fin(r)) bs ++ mkeps(r)
        else List() //bs
    case ZERO => List()
}
```

```
 def fuse(cs: List[Int], r: Rexpb): Rexpb = r match {
    case ZEROb => ZEROb
    case ONEb(bs) => ONEb(cs ++ bs)
    case CHARb(c, marked, bs) => CHARb(c, marked, cs ++ bs)
    case ALTb(r1, r2, bs) => ALTb(r1, r2, cs ++ bs)
    case SEQb(r1, r2, bs) => SEQb(r1, r2, cs ++ bs)
    case STARb(r, bs) => STARb(r, cs ++ bs)
    case NTIMESb(r, n, nmark, counter, bs) => NTIMESb(r, n, nmark, counter, cs ++ bs)
    case INITb(r, bs) => INITb(r, cs ++ bs)
}
def decode(r: Rexp, bs: List[Int]): (VALUE, List[Int]) = r match {
  case ONE => (ONEV, bs) // not sure this should be included
  case CHAR(c, _) => (CHARV(c), bs) // (2) decode (c) bs = (Char(c), bs)
  case ALT(r1, r2) => bs match {
    case 0 :: bs1 => // (3) decode (r1 + r2) 0 :: bs =  (Left(v), bs')  where decode r1 bs => v,bs'
        val (v, bsp) = decode(r1, bs1)
        (LEFT(v), bsp)
    case 1 :: bs1 => // (4) decode (r1 + r2) 1 :: bs = (Right(v), bs') where decode r2 bs => v,bs'
        val (v, bsp) = decode(r2, bs1)
        (RIGHT(v), bsp)
    case x =>
      (ZEROV, bs)
  }
  case SEQ(r1, r2) =>  // (5) decode (r1 · r2) bs = (Seq(v1, v2), bs3) where decode r1 bs => v1,bs2 and decode r2 bs2
=>v2,bs3
    val (v1, bs2) = decode(r1, bs)
    val (v2, bs3) = decode(r2, bs2)
    (SEQV(v1, v2), bs3)
  case STAR(r) => bs match {
    case 1 :: bs1 =>
      (STARV(List()), bs1) // Correctly terminate recursion for STAR
    case 0 :: bs1 =>
      val (v, bs2) = decode(r, bs1)
      val (STARV(vs), bsv) = decode(STAR(r), bs2)
      (STARV(v :: vs), bsv)
    case _ =>
      (STARV(List()), bs) // Edge case: No matches in STAR
  }
}
def intern2(r: Rexp) : Rexpb = INITb(intern(r),List())
def intern(r: Rexp) : Rexpb = r match{
  case ZERO => ZEROb
  case ONE => ONEb(List())
  case CHAR(c,marked) => CHARb(c,marked,List())
  case ALT(r1, r2) =>
    ALTb(fuse(List(0), intern(r1)), fuse(List(1), intern(r2)), List())
  case SEQ(r1, r2) => SEQb(intern(r1),intern(r2),List())
  case STAR(r) => STARb(intern(r),List())
  case NTIMES(r, n, nmark,counter) => NTIMESb(intern(r),n,nmark,counter, List())
}
```

March 22nd

Issues and performance of the current implementation of Point & Bitcodes:

- Nested ***STAR*** constructors cause issues when extracting bits in ***mkeps***. ***SEQ*** of them as well.
- The performance of the current implementation of bitcodes is poor:
- **Testing was done on the performance of both the Bitcodes and Point implementations:**
  1. Using the simple ***Point*** constructor instead of ***Boolean*** marks on ***CHARs*** maintained similar performance to the simple marked approach algorithm.
  2. The Bitcodes implementation (whether using the ***Point*** constructor in **play_Point3_noTags.sc** or ***Boolean*** marks in **marked-BitcodeTest.sc**) performed the worst when tested on the ***EVIL1*** regex ***(a*)*b***. When ***i = 2000***, it took 55 seconds, and When ***i = 1000***, it took 1.23 seconds (***Boolean***) and 1.29 seconds (***Point***).
  3. Using tags (as a list of the history of marks on ***CHAR***, file: **play_point2.sc**) performed better than the Bitcodes implementation but still worse than the simple ***Point***/marked algorithm. It ran out of memory when ***i = 5,000,000*** (took 5.60915 seconds for that) and 2.39494 seconds for ***i = 4,500,000***.

March 23rd

Implementation of BIT constructor:

- For the implementation, a new constructor called BIT was created, which is responsible for collecting bit codes. So far, the current implementation successfully wraps each marked character with a BIT. This BIT collects the correct bits by passing the bit codes through the shift function. When it encounters a character and it matches the string, it wraps it as BIT(POINT, bs).
- Bit codes are accumulated through the shifting process. The way it works is as follows: the shift function takes the whole regular expression along with an initially empty bit list. As the function traverses the expression, it appends bit codes depending on the type of regular expression components. For example, if it finds an alternative (ALT), it shifts both branches as before but appends a 0 to the bit list for the left constructor and a 1 for the right.
- At the end, if a matching character is found, it is wrapped in a BIT(POINT) to record the bits passed to it. A change was also made for the case where a character has already been matched (i.e., already wrapped in a POINT); in this case, it is also wrapped in a BIT constructor to preserve the bit codes for later use.
- Unfortunately, the current implementation only collects the bits. Further changes are needed in the decode and mkepsfunctions to extract and transform the collected bits into the correct values.

March 25 to 27th

Reversing Shifts with popPoints:

- Created a file play_tags.sc which is an adjusted implementation of the point with tags method where tags record the history of the marked charachters. The ***POINT*** constructor is removed now and the ***CHAR*** constructor now holds both a boolean marked flag and a list of integers as tags. Each time a character is matched or unmatched, the shift function updates the boolean value accordingly and appends a corresponding entry to the tags list. The top of the list corresponds to the boolean value. The ***popPoints2*** function pops the top value from the tag list and updates the boolean flag based on the next value on the list. It essentially "unshifts" the effect of the last shift call, reversing one step of the match history.
- The implementation seems to be working effectively, the code is below:

```
enum Rexp {
  case ZERO
  case ONE
  case CHAR(c: Char, marked: Boolean = false, tags: List[Int] = List())
  case ALT(r1: Rexp, r2: Rexp)
  case SEQ(r1: Rexp, r2: Rexp)
  case STAR(r: Rexp)
  case NTIMES(r: Rexp, n: Int , counter: Int = 0)
}
import Rexp._

def shift(m: Boolean, re: Rexp, cp: (Char, Int)) : Rexp = {
  val (c, pos) = cp
  re match {
  case ZERO => ZERO
  case ONE => ONE
  case CHAR(d,marked,tags) =>
    if(m && d == c)
    CHAR(d,true, pos+1 :: tags)
    else CHAR(d,false, 0 :: tags)
  case ALT(r1, r2) => ALT(shift(m, r1, cp), shift(m, r2, cp))
  case SEQ(r1, r2) => SEQ(shift(m, r1, cp), shift((m && nullable(r1)) || fin(r1), r2, cp))
  case STAR(r) => STAR(shift(m || fin(r), r, cp))
  case NTIMES(r, n,counter) => if (counter == n) re else{
        if (m || fin(r)) NTIMES(shift(m || fin(r), r, cp), n, counter+1)
        else NTIMES(shift(false, r, cp), n, counter)
        }  } }
```

```
def popPoints2(r: Rexp): Rexp = r match {
  case ZERO => ZERO
  case ONE  => ONE
    /* case CHAR(c, _, tags) =>
        tags match {
            case Nil => CHAR(c, false, Nil)
            case h :: tail =>CHAR(c, h > 0, tail)
            } */
  case CHAR(c, _, tags) => tags match {
    case Nil => CHAR(c, false, Nil)
    case _ :: next :: rest => CHAR(c, next > 0, next :: rest)
    case _ :: Nil => CHAR(c, false, Nil)
    }
  case ALT(r1, r2) => ALT(popPoints2(r1),popPoints2(r2))
  case SEQ(r1, r2) => SEQ(popPoints2(r1), popPoints2(r2))
  case STAR(inner) => STAR(popPoints2(inner))
  case NTIMES(inner, n, counter) =>NTIMES(popPoints2(inner), n, counter)
}

// helper function to check if any CHAR in the regex still has tags left
def containsTags(r: Rexp): Boolean = r match {
  case CHAR(_, _, tags) => tags.nonEmpty
  case ALT(r1, r2)      => containsTags(r1) || containsTags(r2)
  case SEQ(r1, r2)      => containsTags(r1) || containsTags(r2)
  case STAR(r1)         => containsTags(r1)
  case NTIMES(r1, _, _) => containsTags(r1)
  case _                => false
}

// repeatedly calling popPoints2 until there is no tags left
def collectStages(finReg: Rexp): List[Rexp] = {
  def loop(current: Rexp, stages: List[Rexp]): List[Rexp] = {
    if (!containsTags(current)) stages
    else {
      val next = popPoints2(current)
      loop(next, stages :+ next)
    }
  }
 loop(finReg, List())
}
```

Encoding Bit Codes Within Marks:

- During the last meeting with Dr. Christian, we discussed a potential approach to implementing bit codes using marks that store bit code information. I am currently working on implementing this method. The idea is for the marks to record the bit codes so that, as they are passed along, they retain this information. As a result, marks now need to store more than just a simple true/false value—they must also include the bit sequence that led to the marking.
- I'm considering defining a constructor for marks that includes both a boolean and a list of integers as attributes. When the shift function passes a mark, it will update the mark by appending its own bit sequence. Then, when the boolean value is eventually set to true, the mark will have retained the complete bit code sequence.

March 28th - April 1st

- My first attempt to implement bitcodes involved creating a Mark class that would store a boolean value indicating whether something was marked, along with a list of bitcodes and an additional color attribute for later experimentation. I also changed the ***CHAR*** constructor to store a ***Mark*** object instead of just a boolean value. However, I faced an issue with this approach, especially in the ***mat*** matching function, since that function shifts with a ***true*** boolean initially for the first character of the input string and then with ***false*** afterwards. The main issue was that every time mat called the ***shift*** function, it passed a newly initialized list of bits, like this: ***mat(shift(Mark(false, List(), Color.WHITE), r, c), cs)***
- So I changed the shift function to pass a boolean flag along with a separate list of integers representing the bitcode. Then, in the case of a matched character (a ***CHAR*** with a ***true*** boolean flag), the ***shift*** function

appends the shifted bits to the existing bits inside that ***CHAR***. Otherwise, if there's no match, it keeps the current bits list unchanged.

- Then I refactored the function ***mkeps_marked*** from the previous bitcodes implementation (**marked-BitcodeTest.sc**) into a new version, simply named ***mkeps***. The new version basically extracts the bitcode list from marked characters.
- Then I refactored the function ***decode*** from the previously bitcodes implementation (**marked-BitcodeTest.sc**). The new version has a similar structure—matching different cases of the regular expression constructors and consuming bits along the way—but now it explicitly checks if the bits list is empty in the case of ***CHAR***, and if so, returns a value of ***EMPTY***.
- This implementation seems to be working correctly for simple cases.
  - For instance, if we test it with the simple example ***a + ((b + c) + d)*** and an input string ***"c"***, the output would be: ***mkeps = [1,0,1]*** and ***decode = RIGHT(LEFT(RIGHT(EMPTY)))***
- Still issues with some examples with multiple paths and next step would be to work on more of them, such as those involving ***STAR*** and ***SEQ***.

April 4th

New Shift Code

- The adjustment to shift seems to be working. The change was made in the SEQ case to handle the following scenarios:
  - Nullable left side: Uses ***mkeps*** to carry over the right-hand bits.
  - Final left side: Uses ***mkfin*** to continue into the right-hand side.
  - Third case,neither: doesn't ***shift*** nor pass bits into r2.
- I will conduct testing using the enumerator provided by Dr. Christian. It will enumerate regular expressions along with matching and non-matching input strings to test the new changes.
- I will also explore and test new strategies for the ***STAR*** case using the updated version of shift.
- An adjustment to shift was also made for the ***STAR*** case.
  - (m = ***true***; a new mark to be shifted inside ***r***) and final ***r***: the bit code for ***STAR*** will be { bs ::: mkfin(r) :+ Z }. So to say, append the bitcode for how *r* reaches a final position to the bitcode of the new mark to be introduced to the ***STAR***.
  - ***r*** reaches a final position: just the bit code {mkfin(r) :+ Z} to record how ***r*** reaches this final position.
  - (m=***true***): just passing the bitcode for the mark. continues shifting using the current bit sequence bs.
  - Neither m nor ***fin(r)***: shift the existing marks with m=***false*** and reseting the bit code (Nil).

April 5th

- I have set up my KCL computer to run the longer test using the enumerator provided by Dr. Christian.
- I started by testing basic constructors first, then I will move on to more complex cases to catch mismatches between the marked bitcode algorithm and the proven derivatives approach for comparison. I began with the very basic constructors such as ***CHAR***, ***ALT***, and ***SEQ***.

- The marked approach appears to produce bitcodes identical to those from the derivatives approach when enumerating around 10,000 regular expressions. However, after increasing the enumeration to 150 million, a considerable number of mismatches began to appear (around 40,000).
- Mismatches also occur when the empty string constructor (ONE) is included in the test cases. From my tests, approximately 1/4 of the generated regular expressions showed mismatches—all involving the ONE constructor. The values generated are technically correct, but they do not match the POSIX-preferred results.
- In some observed examples, correct POSIX values were produced when ***mkeps*** was called at the final marked regex instead of ***mkfin***.