This is an outline of the structure of the paper with a little bit of flesh in it.

1 The Flow of thought process

Definition 1. Regular Expressions and values

`TODO`

Value is.a parse tree for the regular expression matching the string.

Definition 2. nullable

`TODO`

The idea behind nullable: whether it contains the empty string.

Definition 3. derivatives TODO

This definition can be used for matching algorithm.

Definition 4. matcher TODO

Definition 5. POSIX values TODO

Definition 6. POSIX lexer algorithm TODO

Definition 7. POSIX lexer algorithm with simplification TODO

This simplification algorithm is rather complicated as it entangles derivative, simplification and value reconstruction. We need to split the regular expression structure of the information for lexing so that simplifcation only changes the regex but does not destroy the information for reconstructing the resulting value.

Introduce regex with auxiliary information:

Definition 8. annotated regular expression TODO

Definition 9. encoding TODO

Encoding translates values into bit codes with some information loss.

Definition 10. decoding TODO

Decoding translates bitcodes back into values with the help of regex to recover the structure.

During different phases of lexing, we sometimes already know what the value would look like if we match the branch of regex with the string(e.g. a STAR with 1 more iteration, a left/right value), so we can partially encode the value at diffferent phases of the algorithm for later decoding.
Examples of such partial encoding:

Definition 11. internalise TODO

When doing internalise on ALT:
Whichever branch is chosen, we know exactly the shape of the value, and therefore can get the bit code for such a value: Left corresponds to Z and Right to S

Definition 12. bmkeps TODO

bmkeps on the STAR case:
We know there could be no more iteration for a star if we want to get a POSIX value for an empty string, so the value must be Stars [], corresponding to an S in the bit code.

Definition 13. bder TODO

SEQ case with the first regex nullable:
bmkeps will extract the value for how a1 mathces the empty string and encode it into a bit sequence.

Definition 14. blexer

`TODO`

adding simplifcation.

size of simplified regex: smaller than Antimirov's pder.

The rest of the document is the residual from a previous doc and may be deleted.

Definition 15. bsimp

```
def bsimp(r: ARexp): ARexp = r match {
  case ASEQ(bs1, r1, r2) => (bsimp(r1), bsimp(r2)) match {
      case (AZERO, _) => AZERO
      case (_, AZERO) => AZERO
      case (AONE(bs2), r2s) => fuse(bs1 ++ bs2, r2s)
      case (r1s, r2s) => ASEQ(bs1, r1s, r2s)
  }
  case AALTS(bs1, rs) => {
    val rs_simp = rs.map(bsimp)
    val flat_res = flats(rs_simp)
    val dist_res = distinctBy(flat_res, erase)
    dist_res match {
      case Nil => AZERO
      case s :: Nil => fuse(bs1, s)
      case rs => AALTS(bs1, rs)
    }
  }
  //case ASTAR(bs, r) => ASTAR(bs, bsimp(r))
  case r => r
}
```

Definition 16. sub-parts of bsimp

- flats
  flattens the list.
- dB
  means distinctBy
- Co
  The last matching clause of the function bsimp, with a slight modification to suit later reasoning.

  ```
  def Co(bs1, rs): ARexp = {
        rs match {
          case Nil => AZERO
          case s :: Nil => fuse(bs1, s)
          case rs => AALTS(bs1, rs)
        }
  ```

Definition 17. fuse

```
def fuse(bs: Bits, r: ARexp) : ARexp = r match {
  case AZERO => AZERO
  case AONE(cs) => AONE(bs ++ cs)
  case ACHAR(cs, f) => ACHAR(bs ++ cs, f)
  case AALTS(cs, rs) => AALTS(bs ++ cs, rs)
  case ASEQ(cs, r1, r2) => ASEQ(bs ++ cs, r1, r2)
  case ASTAR(cs, r) => ASTAR(bs ++ cs, r)
}
```

Definition 18. mkepsBC

```
def mkepsBC(r: ARexp) : Bits = r match {
  case AONE(bs) => bs
```

```
    case AALTS(bs, rs) => {
      val n = rs.indexWhere(bnullable)
      bs ++ mkepsBC(rs(n))
    }
    case ASEQ(bs, r1, r2) => bs ++ mkepsBC(r1) ++ mkepsBC(r2)
    case ASTAR(bs, r) => bs ++ List(Z)
  }
```

Definition 19. mkepsBC equicalence
Given 2 nullable annotated regular expressions r1, r2, if mkepsBC(r1) == mkepsBC(r2) then r1 and r2 are mkepsBC equivalent, denoted as r1 $\sim_{m\epsilon}$ r2

Definition 20. shorthand notation for ders
For the sake of reducing verbosity, we sometimes use the shorthand notation $d_c(r)$ for the function application bder(c, r) and $s(r)$(s here stands for simplification) for the function application bsimp(r) .
We omit the subscript when it is clear from context what that character is and write $d(r)$ instead of $d_c(r)$.
And we omit the parentheses when no confusion can be caused. For example ders_simp(c, r) can be written as $s(d_c(r))$ or even *sdr* as we know the derivative operation is w.r.t the character c. Here the s and d are more like operators that take an annotated regular expression as an input and return an annotated regular expression as an output

Definition 21. distinctBy operation expressed in a different way–how it transforms the list
Given two lists rs1 and rs2, we define the operation $--$:
$rs1 -- rs2 := [r \in rs1 | r \notin rs2]$ Note that the order each term appears in $rs_1 -- rs_2$ is preserved as in the original list.

2 Main Result

Lemma 1. *simplification function does not simplify an already simplified regex*
bsimp(r) == bsimp(bsimp(r)) holds for any annotated regular expression r.

Lemma 2. *simp and mkeps*
When r is nullable, we have that mkeps(bsimp(r)) == mkeps(r)

Lemma 3. *mkeps equivalence w.r.t some syntactically different regular expressions(1 ALTS)*
When one of the 2 regular expressions $s(r_1)$ and $s(r_2)$ is of the form ALTS(bs1, rs1), we have that $ds(ALTS(bs, r1, r2)) \sim_{m\epsilon} d(ALTS(bs, sr_1, sr_2))$

Proof. By opening up one of the alts and show no additional changes are made.
Details: $ds(ALTS(bs, r1, r2)) = dCo(bs, dB(flats(sr1, sr2)))$ □

Lemma 4. *mkepsBC invariant manipulation of bits and notation*
ALTS(bs, ALTS(bs1, rs1), ALTS(bs2, rs2)) $\sim_{m\epsilon}$ ALTS(bs, rs1.map(fuse(bs1, _)) ++ rs2.map(fuse(bs2, _))).
We also use $bs2 >> rs2$ as a shorthand notation for rs2.map(fuse(bs2,_)).

Lemma 5. *mkepsBC equivalence w.r.t syntactically different regular expressions(2 ALTS)*
$sr_1 = ALTS(bs1, rs1)$ and $sr_2 = ALTS(bs2, rs2)$ we have $d(sr_1 + sr_2) \sim_{m\epsilon} d(ALTS(bs, bs1 >> rs1 + + bs2 >> rs2))$

Proof. We are just fusing bits inside here, there is no other structural change. □

Lemma 6. *What does dB do to two already simplified ALTS*
$dCo(ALTS(bs, dB(bs1 >> rs1 + + bs2 >> rs2))) = dCo(ALTS(bs, bs1 >> rs1 + + ((bs2 >> rs2) -- rs1)))$

Proof. We prove that $dB(bs1 >> rs1 + + bs2 >> rs2) = bs1 >> rs1 + + ((bs2 >> rs2) -- rs1)$.

□

Lemma 7. *after opening two previously simplified alts up into terms, length must exceed 2*
If sr1, sr2 are of the form ALTS(bs1, rs1), ALTS(bs2, rs2) respectively, then we have that $Co(bs, (bs1 >> rs1) + + (bs2 >> rs2) -- rs1) = ALTS(bs, bs1 >> rs1 + + (bs2 >> rs2) -- rs1)$

Proof. $Co(bs, rs) \sim_{m\epsilon} ALTS(bs, rs)$ if rs is a list of length greater than or equal to 2. As suggested by the title of this lemma, ALTS(bs1, rs1) is a result of simplification, which means that rs1 must be composed of at least 2 distinct regular terms. This alone says that $bs1 >> rs1 + + (bs2 >> rs2) -- rs1$ is a list of length greater than or equal to 2, as the second operand of the concatenation operator $(bs2 >> rs2) -- rs1$ can only contribute a non-negative value to the overall length of the list $bs1 >> rs1 + + (bs2 >> rs2) -- rs1$. □

Lemma 8. *mkepsBC equivalence w.r.t syntactically different regular expressions(2 ALTS+ some deletion after derivatives)* $dALTS(bs, bs1 >> rs1 + +bs2 >> rs2) \sim_{m\epsilon} dALTS(bs, bs1 >> rs1 + +((bs2 >> rs2) - -rs1))$

Proof. Let's call $bs1 >> rs1\, rs1'$ and $bs2 >> rs2\, rs2'$. Then we need to prove $dALTS(bs, rs1' + +rs2') \sim_{m\epsilon} dALTS(bs, rs1' + +(rs2' - -rs1'))$.
We might as well omit the prime in each rs for simplicty of notation and prove $dALTS(bs, rs1 + +rs2) \sim_{m\epsilon} dALTS(bs, rs1 + +(rs2 - -rs1))$.
We know that the result of derivative is nullable, so there must exist an r in rs1++rs2 s.t. r is nullable.
If $r \in rs1$, then equivalence holds. If $r \in rs2 \wedge r \notin rs1$, equivalence holds as well. This completes the proof. □

Lemma 9. *nullability relation between a regex and its simplified version*
r nullable $\iff$ *sr nullable*

Lemma 10. *concatenation + simp invariance of mkepsBC*
$mkepsBCr1 \cdot sr2 = mkepsBCr1 \cdot r2$ *if both r1 and r2 are nullable.*

Theorem 1. *Correctness Result*

- *When s is a string in the language L(ar),*
  ders_simp(ar, s) $\sim_{m\epsilon}$ *ders(ar, s),*

- *when s is not a string of the language L(ar) ders_simp(ar, s) is not nullable*

Proof. Split into 2 parts.

- When we have an annotated regular expression ar and a string s that matches ar, by the correctness of the algorithm ders, we have that ders(ar, s) is nullable, and that mkepsBC will extract the desired bits for decoding the correct value v for the matching, and v is a POSIX value. Now we prove that mkepsBC(ders_simp(ar, s)) yields the same bitsequence. We first open up the ders_simp function into nested alternating sequences of ders and simp. Assume that s = $c_1 ... c_n (n \geq 1$) where each of the c_i are characters. Then $ders\_simp(ar, s) = s(d_{c_n}(...s(d_{c_1}(r))...)) = sdsd.....sdr$. If we can prove that $sdr \sim_{m\epsilon} dsr$ holds for any regular expression and any character, then we are done. This is because then we can push ders operation inside and move simp operation outside and have that $sdsd...sdr \sim_{m\epsilon} ssddsdsd...sdr \sim_{m\epsilon} ... \sim_{m\epsilon} s....sd....dr$. Using **??** we have that $s...sd....dr = sd...dr$. By **??**, we have $RHS \sim_{m\epsilon} d...dr$.
  Notice that we don't actually need **??** here. That is because by **??**, we can have that $s...sd....dr \sim_{m\epsilon} sd...dr$. The equality above can be replaced by mkepsBC equivalence without affecting the validity of the whole proof since all we want is mkepsBC equivalence, not equality.

  Now we proceed to prove that $sdr \sim_{m\epsilon} dsr$. This can be reduced to proving $dr \sim_{m\epsilon} dsr$ as we know that $dr \sim_{m\epsilon} sdr$ by **??**.

  we use an induction proof. Base cases are omitted. Here are the 3 inductive cases.

  - $r_1 + r_2$
    The most difficult case is when $sr1$ and $sr2$ are both ALTS, so that they will be opened up in the flats function and some terms in sr2 might be deleted. Or otherwise we can use the argument that $d(r_1 + r_2) = dr_1 + dr_2 \sim_{m\epsilon} dsr_1 + dsr_2 \sim_{m\epsilon} ds(r_1 + r_2)$, the last equivalence being established by **??**. When $s(r_1), s(r_2)$ are both ALTS, we have to be more careful for the last equivalence step, namelly, $dsr_1 + dsr_2 \sim_{m\epsilon} ds(r_1 + r_2)$.
    We have that $LHS = dsr_1 + dsr_2 = d(sr_1 + sr_2)$. Since $sr_1 = ALTS(bs1, rs1)$ and $sr_2 = ALTS(bs2, rs2)$ we have $d(sr_1 + sr_2) \sim_{m\epsilon} d(ALTS(bs, bs1 >> rs1 + +bs2 >> rs2))$ by **??**. On the other hand, $RHS = ds(ALTS(bs, r1, r2)) = dCo(bs, dB(flats(s(r1), s(r2)))) = dCo(bs, dB(bs1 >> rs1 + +bs2 >> rs2))$ by definition of bsimp and flats.
    $dCo(bs, dB(bs1 >> rs1 + +bs2 >> rs2)) = dCo(bs, (bs1 >> rs1 + +((bs2 >> rs2) - -rs1)))$ by **??**.
    $dCo(bs, (bs1 >> rs1 + +((bs2 >> rs2) - -rs1))) = d(ALTS(bs, bs1 >> rs1 + +(bs2 >> rs2) - -rs1))$ by **??**.
    Using **??**, we have $d(ALTS(bs, bs1 >> rs1 + +(bs2 >> rs2) - -rs1)) \sim_{m\epsilon} d(ALTS(bs, bs1 >> rs1 + +bs2 >> rs2)) \sim_{m\epsilon} RHS$.
    This completes the proof.
  - $r*$
    $s(r*) = r*$. Our goal is trivially achieved.
  - $r1 \cdot r2$
    When r1 is nullable, $dsr1r2 = dsr1 \cdot sr2 + dsr2 \sim_{m\epsilon} dr1 \cdot sr2 + dr2 = dr1 \cdot r2 + dr2$. The last step uses **??**. When r1 is not nullable, $dsr1r2 = dsr1 \cdot sr2 \sim_{m\epsilon} dr1 \cdot sr2 \sim_{m\epsilon} dr1 \cdot r2$

- Proof of second part of the theorem: use a similar structure of argument as in the first part.
- This proof has a major flaw: it assumes all dr is nullable along the path of deriving r by s. But it could be the case that $s \in L(r)$ but $\exists s' \in Pref(s)$ $s.t.$ $ders(s', r)$ is not nullable (or equivalently, $s' \notin L(r)$). One remedy for this is to replace the mkepsBC equivalence relation into some other transitive relation that entails mkepsBC equivalence.

□

Theorem 2. *This is a very strong claim that has yet to be more carefully examined and proved. However, experiments suggest a very good hope for this.*
Define pushbits as the following:

```
def pushbits(r: ARexp): ARexp = r match {
    case AALTS(bs, rs) => AALTS(Nil, rs.map(r=>fuse(bs, pushbits(r))))
    case ASEQ(bs, r1, r2) => ASEQ(bs, pushbits(r1), pushbits(r2))
    case r => r
  }
```

Then we have $\boldsymbol{pushbits(ders\_simp(ar, s)) == simp(ders(ar, s))}$ *or* $\boldsymbol{ders\_simp(ar, s) == simp(ders(ar, s))}$.
Unfortunately this does not hold. A counterexample is

```
baa
original regex
STA
 └-ALT
    └-STA List(Z)
    |  └-a
    └-ALT List(S)
       └-b List(Z)
       └-a List(S)
regex after ders simp
ALT List(S, S, Z, C(b))
 └-SEQ
 |  └-STA List(S, Z, S, C(a), S, C(a))
 |  |  └-a
 |  └-STA
 |     └-ALT
 |        └-STA List(Z)
 |        |  └-a
 |        └-ALT List(S)
 |           └-b List(Z)
 |           └-a List(S)
 └-SEQ List(S, Z, S, C(a), Z)
    └-ALT List(S)
    |  └-STA List(Z, S, C(a))
    |  |  └-a
    |  └-ONE List(S, S, C(a))
    └-STA
       └-ALT
          └-STA List(Z)
          |  └-a
          └-ALT List(S)
             └-b List(Z)
             └-a List(S)
regex after ders
ALT
 └-SEQ
 |  └-ALT List(S)
 |  |  └-SEQ List(Z)
```

```
 |  |  |  └-ZERO
 |  |  |  └-STA
 |  |  |     └-a
 |  |  └-ALT List(S)
 |  |     └-ZERO
 |  |     └-ZERO
 |  └-STA
 |     └-ALT
 |        └-STA List(Z)
 |        |  └-a
 |        └-ALT List(S)
 |           └-b List(Z)
 |           └-a List(S)
 └-ALT List(S, S, Z, C(b))
    └-SEQ
    |  └-ALT List(S)
    |  |  └-ALT List(Z)
    |  |  |  └-SEQ
    |  |  |  |  └-ZERO
    |  |  |  |  └-STA
    |  |  |  |     └-a
    |  |  |  └-SEQ List(S, C(a))
    |  |  |     └-ONE List(S, C(a))
    |  |  |     └-STA
    |  |  |        └-a
    |  |  └-ALT List(S)
    |  |     └-ZERO
    |  |     └-ZERO
    |  └-STA
    |     └-ALT
    |        └-STA List(Z)
    |        |  └-a
    |        └-ALT List(S)
    |           └-b List(Z)
    |           └-a List(S)
    └-SEQ List(S, Z, S, C(a), Z)
       └-ALT List(S)
       |  └-SEQ List(Z)
       |  |  └-ONE List(S, C(a))
       |  |  └-STA
       |  |     └-a
       |  └-ALT List(S)
       |     └-ZERO
       |     └-ONE List(S, C(a))
       └-STA
          └-ALT
             └-STA List(Z)
             |  └-a
             └-ALT List(S)
                └-b List(Z)
                └-a List(S)
regex after ders and then a single simp
ALT
 └-SEQ List(S, S, Z, C(b))
 |  └-STA List(S, Z, S, C(a), S, C(a))
 |  |  └-a
 |  └-STA
 |     └-ALT
 |        └-STA List(Z)
 |        |  └-a
```

```
|        └-ALT List(S)
|           └-b List(Z)
|           └-a List(S)
└-SEQ List(S, S, Z, C(b), S, Z, S, C(a), Z)
   └-ALT List(S)
   |  └-STA List(Z, S, C(a))
   |  |  └-a
   |  └-ONE List(S, S, C(a))
   └-STA
      └-ALT
         └-STA List(Z)
         |  └-a
         └-ALT List(S)
            └-b List(Z)
            └-a List(S)
```