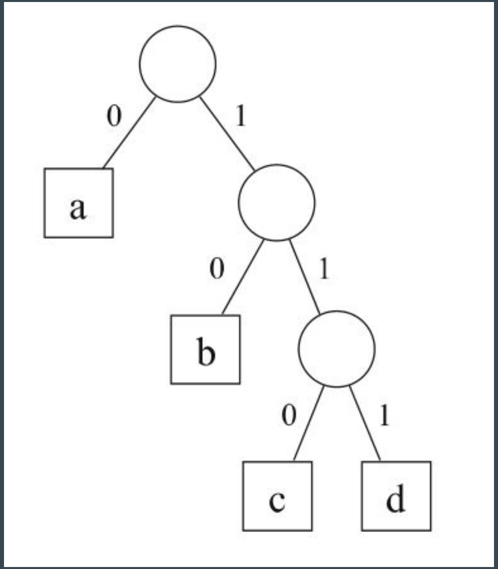
Implementation and verification of encoder and decoder for prefix-free codes

Formal verification (CS-550)
final project

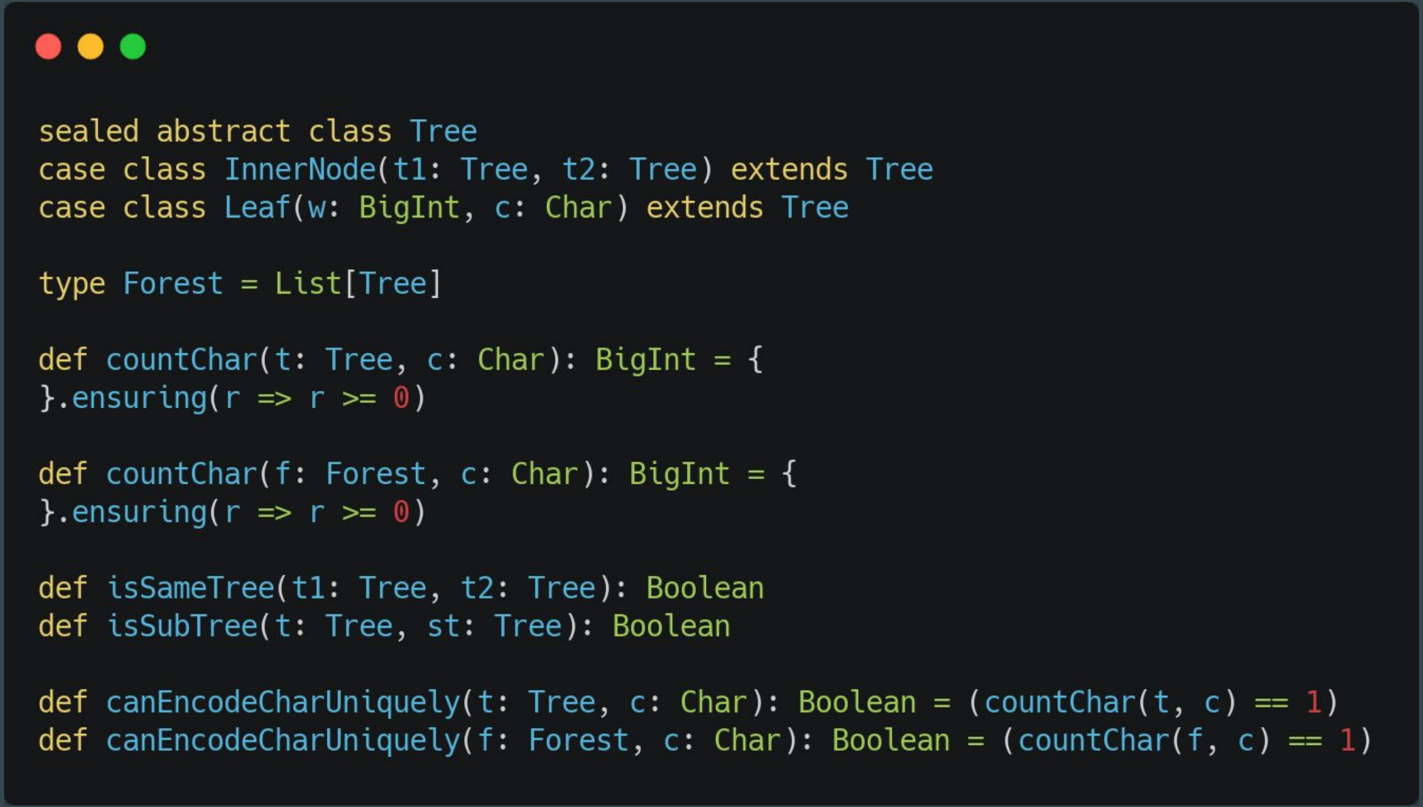
Samuel Chassot & Daniel Filipe Nunes Silva

Prefix-free codes and goal

- Code, $\boldsymbol{C} = \{ a \rightarrow 0, b \rightarrow 10, c \rightarrow 110, d \rightarrow 111 \}$
- Full binary tree
- Huffman's algorithm produces the optimal tree
- Implement and verify encoder and decoder for prefix-free codes

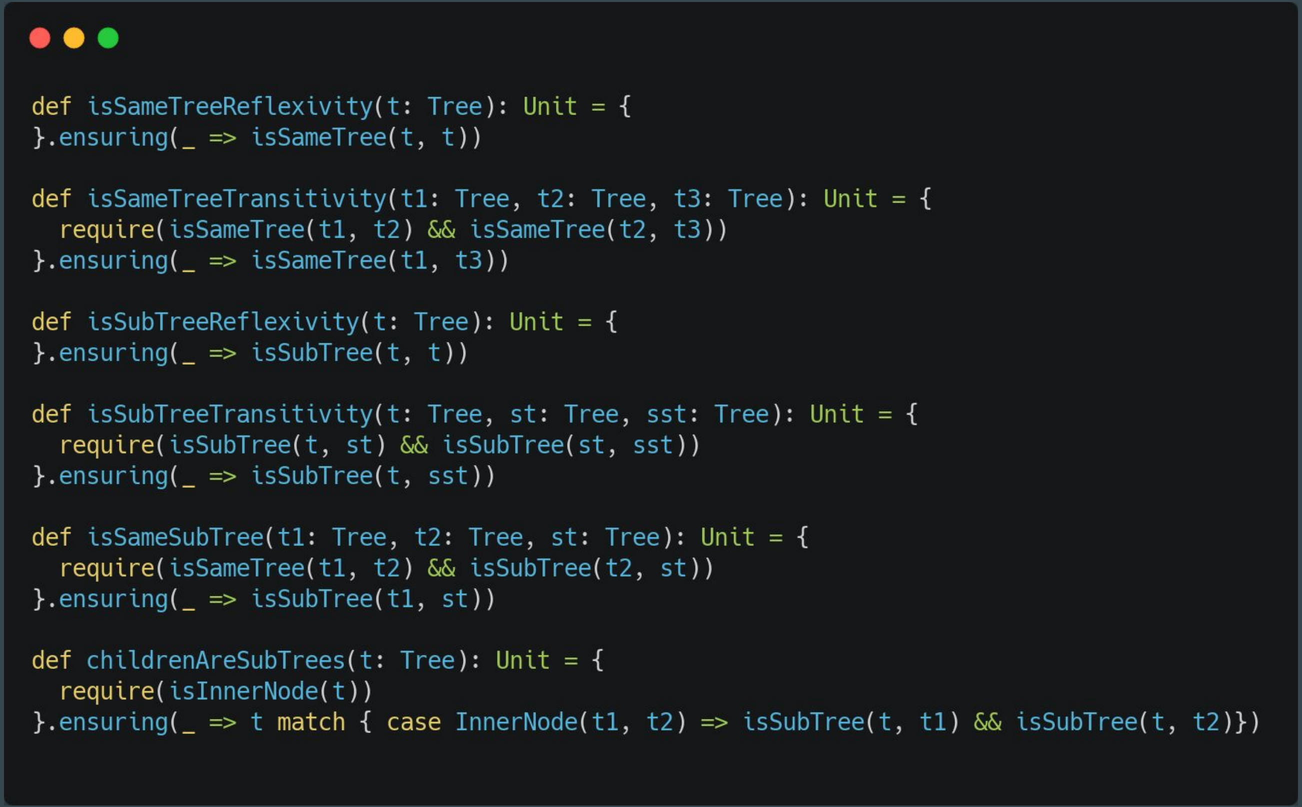


Classes and types



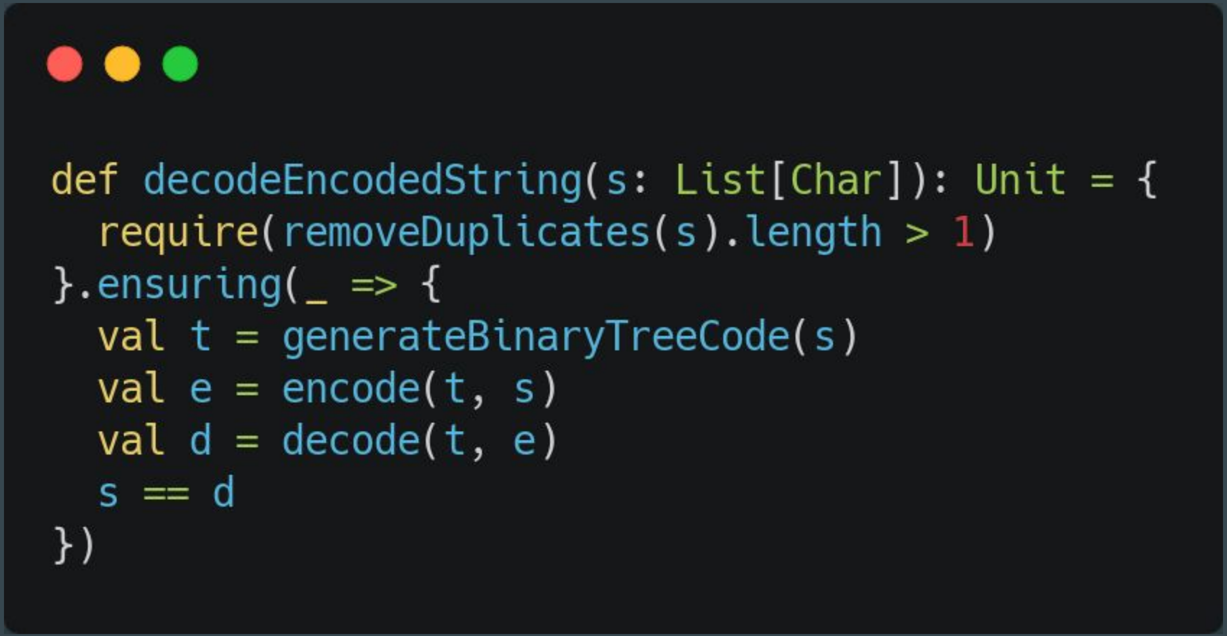
```scala
sealed abstract class Tree
case class InnerNode(t1: Tree, t2: Tree) extends Tree
case class Leaf(w: BigInt, c: Char) extends Tree

type Forest = List[Tree]

def countChar(t: Tree, c: Char): BigInt = {
}.ensuring(r => r >= 0)

def countChar(f: Forest, c: Char): BigInt = {
}.ensuring(r => r >= 0)

def isSameTree(t1: Tree, t2: Tree): Boolean
def isSubTree(t: Tree, st: Tree): Boolean

def canEncodeCharUniquely(t: Tree, c: Char): Boolean = (countChar(t, c) == 1)
def canEncodeCharUniquely(f: Forest, c: Char): Boolean = (countChar(f, c) == 1)
```

Lemmas on trees



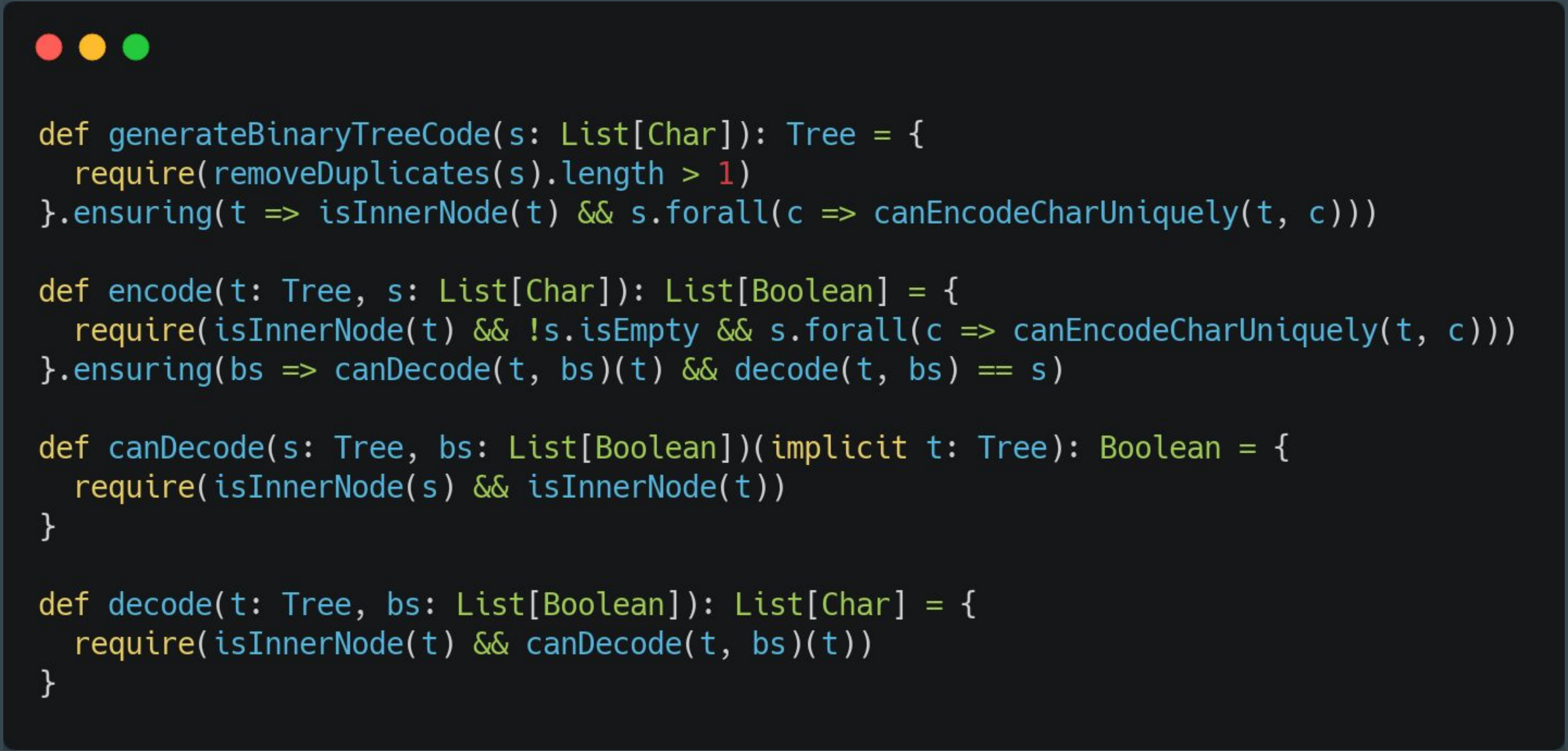
```scala
def isSameTreeReflexivity(t: Tree): Unit = {
}.ensuring(_ => isSameTree(t, t))

def isSameTreeTransitivity(t1: Tree, t2: Tree, t3: Tree): Unit = {
  require(isSameTree(t1, t2) && isSameTree(t2, t3))
}.ensuring(_ => isSameTree(t1, t3))

def isSubTreeReflexivity(t: Tree): Unit = {
}.ensuring(_ => isSubTree(t, t))

def isSubTreeTransitivity(t: Tree, st: Tree, sst: Tree): Unit = {
  require(isSubTree(t, st) && isSubTree(st, sst))
}.ensuring(_ => isSubTree(t, sst))

def isSameSubTree(t1: Tree, t2: Tree, st: Tree): Unit = {
  require(isSameTree(t1, t2) && isSubTree(t2, st))
}.ensuring(_ => isSubTree(t1, st))

def childrenAreSubTrees(t: Tree): Unit = {
  require(isInnerNode(t))
}.ensuring(_ => t match { case InnerNode(t1, t2) => isSubTree(t, t1) && isSubTree(t, t2)})
```

Main theorem



```
def decodeEncodedString(s: List[Char]): Unit = {
  require(removeDuplicates(s).length > 1)
}.ensuring(_ => {
  val t = generateBinaryTreeCode(s)
  val e = encode(t, s)
  val d = decode(t, e)
  s == d
})
```

Preconditions and postconditions

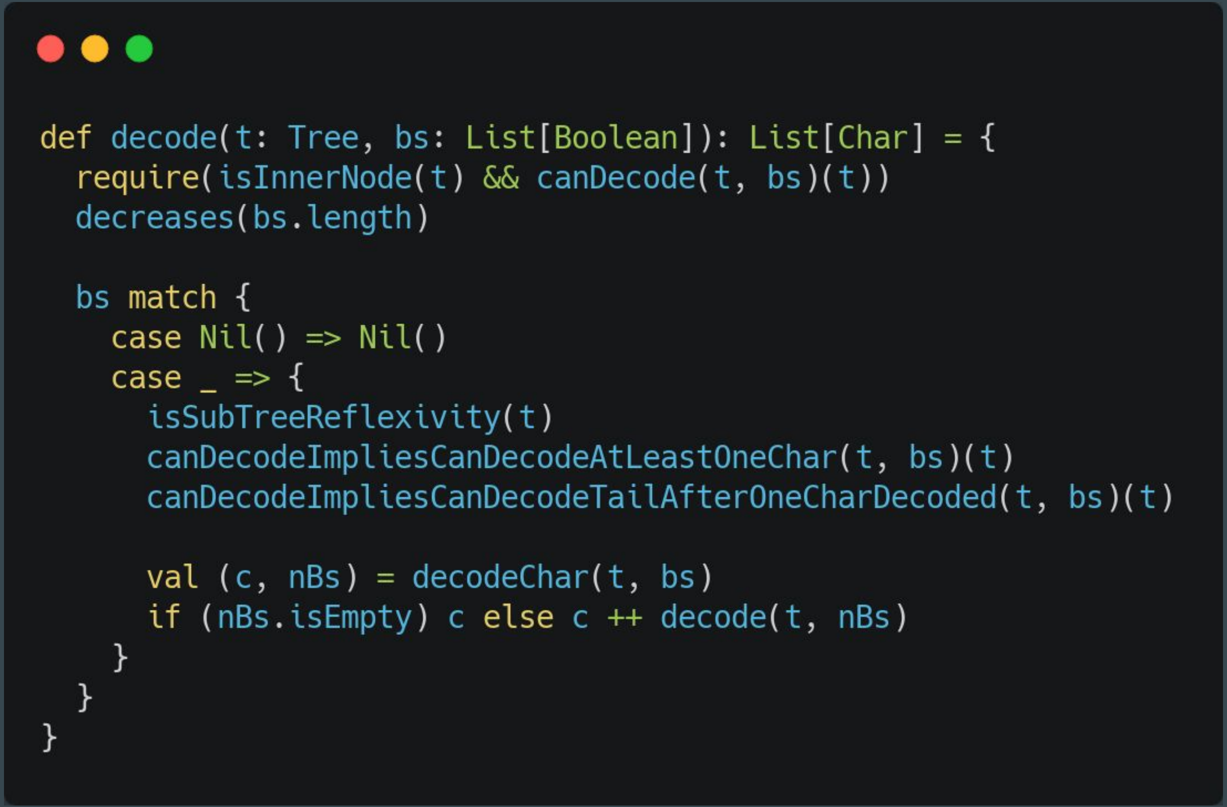


```scala
def generateBinaryTreeCode(s: List[Char]): Tree = {
  require(removeDuplicates(s).length > 1)
}.ensuring(t => isInnerNode(t) && s.forall(c => canEncodeCharUniquely(t, c)))

def encode(t: Tree, s: List[Char]): List[Boolean] = {
  require(isInnerNode(t) && !s.isEmpty && s.forall(c => canEncodeCharUniquely(t, c)))
}.ensuring(bs => canDecode(t, bs)(t) && decode(t, bs) == s)

def canDecode(s: Tree, bs: List[Boolean])(implicit t: Tree): Boolean = {
  require(isInnerNode(s) && isInnerNode(t))
}

def decode(t: Tree, bs: List[Boolean]): List[Char] = {
  require(isInnerNode(t) && canDecode(t, bs)(t))
}
```

Roadmap

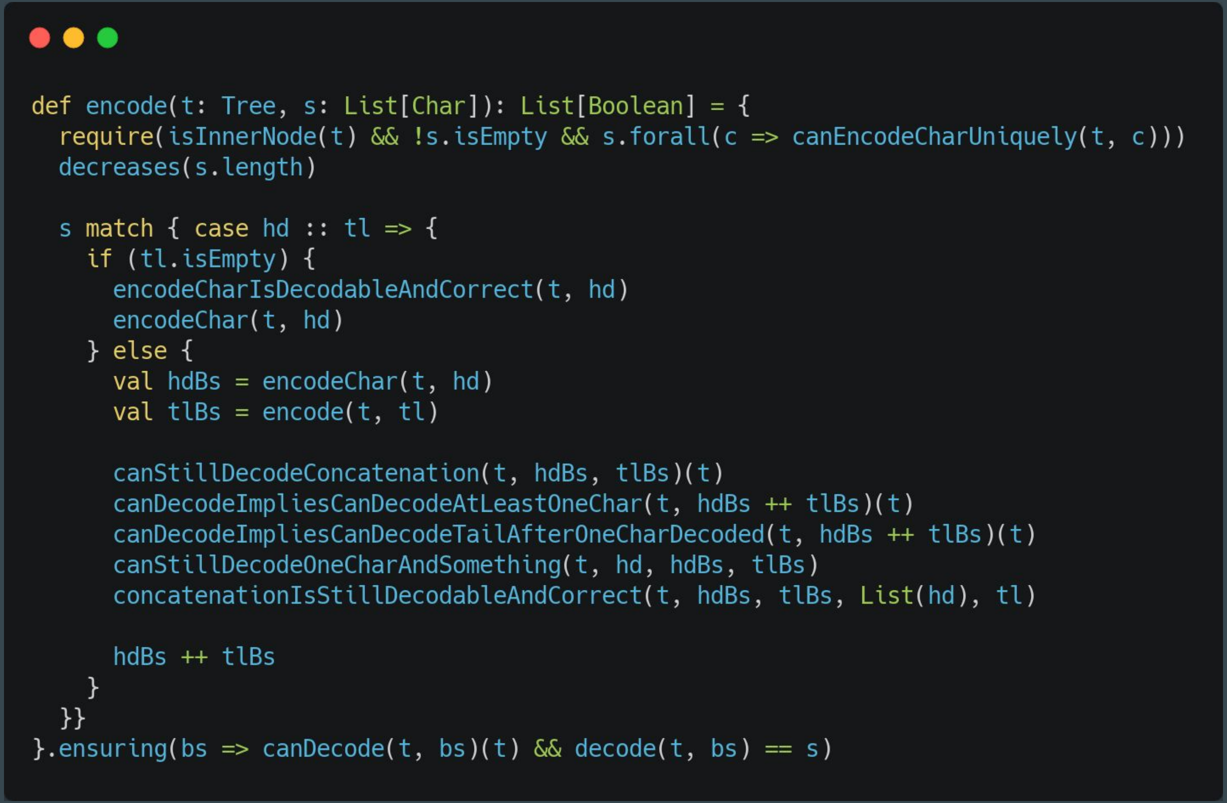
- Naive implementation
  - Types
  - Huffman’s algorithm
  - Encode
  - Decode
- Adapt decode functions and prove preconditions
- Adapt encode functions and prove postconditions

Proof of decode



```
def decode(t: Tree, bs: List[Boolean]): List[Char] = {
  require(isInnerNode(t) && canDecode(t, bs)(t))
  decreases(bs.length)

  bs match {
    case Nil() => Nil()
    case _ => {
      isSubTreeReflexivity(t)
      canDecodeImpliesCanDecodeAtLeastOneChar(t, bs)(t)
      canDecodeImpliesCanDecodeTailAfterOneCharDecoded(t, bs)(t)

      val (c, nBs) = decodeChar(t, bs)
      if (nBs.isEmpty) c else c ++ decode(t, nBs)
    }
  }
}
```

Proof of encode



```scala
def encode(t: Tree, s: List[Char]): List[Boolean] = {
  require(isInnerNode(t) && !s.isEmpty && s.forall(c => canEncodeCharUniquely(t, c)))
  decreases(s.length)

  s match { case hd :: tl => {
    if (tl.isEmpty) {
      encodeCharIsDecodableAndCorrect(t, hd)
      encodeChar(t, hd)
    } else {
      val hdBs = encodeChar(t, hd)
      val tlBs = encode(t, tl)

      canStillDecodeConcatenation(t, hdBs, tlBs)(t)
      canDecodeImpliesCanDecodeAtLeastOneChar(t, hdBs ++ tlBs)(t)
      canDecodeImpliesCanDecodeTailAfterOneCharDecoded(t, hdBs ++ tlBs)(t)
      canStillDecodeOneCharAndSomething(t, hd, hdBs, tlBs)
      concatenationIsStillDecodableAndCorrect(t, hdBs, tlBs, List(hd), tl)

      hdBs ++ tlBs
    }
  }}
}.ensuring(bs => canDecode(t, bs)(t) && decode(t, bs) == s)
```

Conclusion and challenge

- What we have so far
  - Verified implementations of encode and decode functions pair for prefix-free codes
  - Implementation of Huffman's algorithm and an other naive algorithm to generate prefix-free codes
- What we plan to do next
  - Verify naive algorithm
  - Write report
- Continuously question our implementations