1. *Introduction*

   In some languages (C and Java are two notable examples we've dealt with), it can be difficult to make strings do what you want them to do. "Java string" has been typed into my browser so many times that it's one of my first autocomplete options when I type "j" into google. Our language, Super Simple String Stuff (S4 for short for now, official name pending), will make it much easier to write string manipulation based programs. In S4,

   the user won't have to worry about indexing into char arrays - everything string-related will be string type. The language will also provide useful string methods that may not be present in other languages, such as "middle" or "replace". We will also try to implement "string arithmetic" to allow users to clearly and precisely modify their strings.

2. *Design Principles*

   Strings will be mutable, thought of almost like arrays of characters, that can be manipulated as needed. Programs should be streamlined, so the arrows pass one argument into another seamlessly. Every function falls into one of two categories: Check functions will return true or false if the given string satisfies the given property. Modify functions will mutate a string in some way, and return a string.

3. *Examples*

```
> Dan replace("an", "uane")
        "Duane"
        > Dan anagram() prefix("R") multiply(3)
        "RanDRanDRanD"
        > Dan shuffle() tail() shuffle()
        "n"
```

4. *Language Concepts*

   The user should be familiar with strings and how and why we use them, but does not need to be concerned with the technical way strings are stored and accessed. They should also be familiar with booleans, integers, and conditional statements. As for combining forms, the user should understand piping output from one function into another function, arithmetic operations with strings, and creating functions that modify input or produce new output.

5. *Syntax*

   An expression will have a starting string, and a rule. Rules can be one or more functions that modify the string. Rules either output a string, or end with a check function, which outputs true or false. As of now, our BNF looks like the following

```
<start>         ::= <string>_<expr>
<expr>          ::= "NOP"
                | <seq>
                | func
<seq>
<func>          ::= <name> (<variable>*)
<name>          ::= (in list of function names)
<variable>      ::= <number>
                | <string>
<number>        ::= n in N
```

6. *Semantics*

| Syntax | Abstract Syntax | Type | Description |
|---|---|---|---|
| s e1 | Start(s, e1) | string → string * Variable list → string | The input string for the program is passed to the first expression e1, which is a function, along with additional arguments. |
| e1 e2 | Seq(e1,e2) | string * Variable list → string * Variable list → string | A sequence of two expressions, which are functions. The first function e1 is evaluated, outputs a string, and is passed with the additional variable list to the second function e2. |

(a) Our primitives will be strings (represented on the machine as char arrays), integers, and booleans. Integers will primarily be used as inputs to certain functions (such as multiply) and booleans will mainly be outputs of check type functions (like a user-written palindrome function). The user will be able to interact with these types as well as the string type, which we will deal with as a char array in our language implementation.

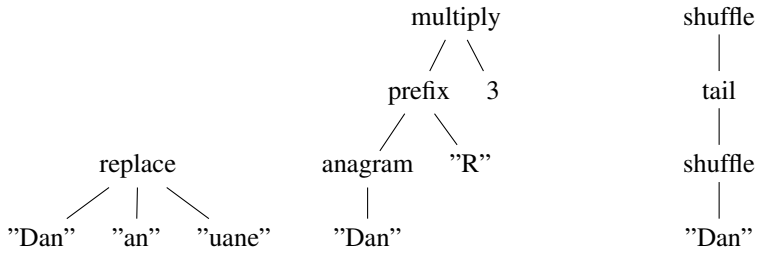
(b) The actions of our program fall into one of two categories, as noted before: with a given string, the user can modify it in some way or check it to see if it has some user-defined property. An example of user modification is something like "prefix", and an example of a check is something like "isUpper". Methods like "first" and "last" fall into the category of modification. The user can write their own modifying and checking functions.

(c) Algebraic Data types:

```
type Variable =
    | String of string
    | Number of int

type Expr =
    | Function of (string * Variable list)
    | Seq of Expr * Expr
    | NOP
```

(d)




(e) i. Programs can read in a string (maybe eventually a file)

ii. Programs either output a string or a boolean

iii. Each function that doesn't return a string can only be used at the end of a function, so any function in the middle of the tree must take in a string and output a string. Functions are evaluated left to right, and return values are passed as arguments to other functions.