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, SMPL, will allow simpler string manipulation programs.

   In SMPL, 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 built-in string methods that may not be present in other languages, such as "middle" or "contains".

2. **Design Principles**

   We want users to be able to apply many functions in a sequence to an input string, much like a SQL database query. Every function takes in a string input and potentially additional arguments, and outputs a string that can be then manipulated by the next function. All outputs are strings, so functions that might return booleans like "contains()" or numbers like "length" return strings containing the values.

3. **Examples**

   Programs can be called as file names like examples 1 and 3 or run in the shell as command line arguments like example 2. You can run them from the lang folder like this:

```
dotnet run "input" "../examples/example-1.smpl"
"Prepended input Appended"

dotnet run "ex2" "repeat(length()) length()"
"9"

dotnet run "input with CAPS" "../examples/example-3.smpl"
"False"
```

   Here are the contents of the example programs:

```
example-1.smpl
prepend("Prepended ") append(" Appended")

example-2.smpl
repeat(length()) length()

example-3.smpl
append(toLower() reverse()) isPalindrome()
```

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 and integers. As for combining forms, the user should understand piping output from one function into another function, various string operations, and nested functions. The user should know that additional arguments, like those to "substring()", can be integers or strings of numbers, so they need not worry about converting inputs to strings themselves.

5. **Formal Syntax**

Programs in SMPL follow the BNF grammar below. Note that the program does not include the input, which is passed in as a separate argument on the command line.

```
<expression>    ::= <number>
                | <string>
                | <builtin>
                | <sequence>
<sequence>      ::= <builtin>_<expression>
<builtin>       ::= <name> (<expression>*)
<name>          ::= "length"
                | "first"
                |"last"
                |"middle"
                |"getEnd"
                |"isUpper"
                |"isLower"
                |"toUpper"
                |"toLower"
                |"isPalindrome"
                |"reverse"
                |"repeat"
                |"prepend"
                |"append"
                |"substring"
                |"contains"
<string>        ::= <character>*
<character>     ::= \alpha \in {a...z, A...Z, ' '}
<number>        ::= n in N
```

6. **Semantics**

(a) *Primitive Types*

Our primitives are Strings and Numbers. Booleans exist as the output of certain functions (e.g. "isPalindrome()"), but are cast to strings to maintain closure of the language. Numbers are non-negative integers are used as inputs to certain functions (e.g. "repeat()"), as well as outputs of certain functions (e.g. "length()"). Numbers are also cast to strings when functioning as output, to maintain closure. Strings serve as the output of all functions, but can also be inputs to functions (e.g. "append()").

(b) *Program Actions*

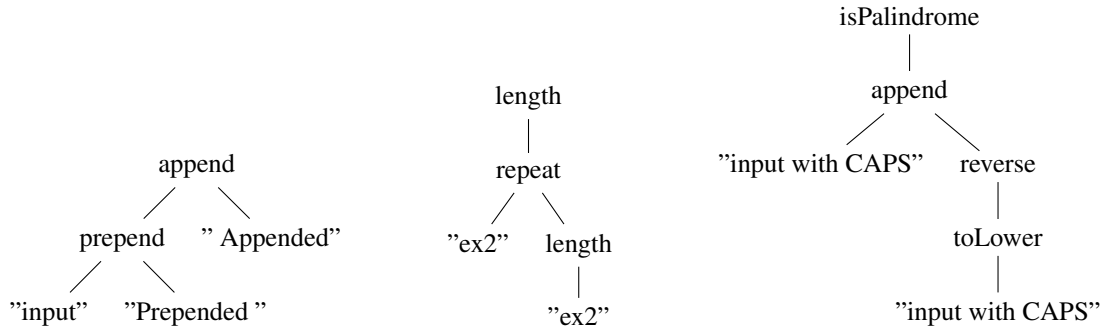
Functions are in a sequence pattern, with one following another. Each function takes at least a string as input, and potentially more variables. For example, "length()" needs no additional input while "substring()" needs two integer inputs. Functions are evaluated in sequence from left to right, with the input string passed to the first function, the result of that function passed to the next, and so on.

Arguments passed to functions literally (e.g. "repeat(3)") are evaluated internally, so the user should just worry about the intuitive type of an input. Strings passed as literals (e.g. "append("Appended")") must be surrounded by single (') or double (") quotes. When written in the command line, programs must be surrounded by double quotes, so single quotes can be used for passing literals. In program files, either can be used. An integer passed as "2" including quotes will be interpreted as an integer if passed to an integer-seeking function and a string otherwise. An integer passed as 2 without quotes will be interpreted as an integer.

(c) *Algebraic Data Types*

```
type Expr =
| Number of int
| String of string
| Builtin of (string * Expr list)
| Seq of Expr * Expr
| NOP
```

(d) *Example Program ASTs*



(e) *Evaluation*

i. ***Input.*** SMPL programs do not read input themselves. The user provides am input string via the command line. The first command line argument is the input string and the second is the program or the file containing the program.

ii. ***Output.*** SMPL programs always output a string. Programs have no side effects.

iii. ***Evaluation order.*** Functions which take arguments can take either literal inputs (e.g. "append('L')") or can take nested functions (e.g. "append(reverse())"). Nested functions can be a single function, or an entire expression (e.g. "append(toLower() reverse())") as long as the nested expression evaluates to the correct type. The internal expression is evaluated with the given input, and then the original function is evaluated on the original input with the output of the nested function.
In the example of "append(toLower() reverse())" is called with "Lucas", "toLower() reverse()" is evaluated with "Lucas" first resulting in "sacul", then append is called on the original input, resulting in "Lucassacul". This is not a case sensitive palindrome, so "isPalindrome()" returns "False".

7. **Remaining work**

We plan to:

Implement more built-in functions, such as "findAndReplace", "countSubstring", and "isWord".

Eliminate the need to have parentheses after functions that don't take arguments.

Parse strings with symbols as arguments to functions

Create a help menu that can be accessed with "dotnet run help"

Write unit tests

Our stretch goal is to implement a checker, which would check to make sure that arguments to functions are the types they should be and that there are the right number of them. Our "zen yoga" goal is to implement user-defined functions.