String Param Substitute

Background

In general, any function is collection of tasks defined as statements. Instead of creating different codes for same task with different inputs, it is advisable to create single function with input parameter. A single function can be called multiple times which performs a specific task. It is easier to review or debug the code in case any defects to be fixed or to understand the specific task. In this session, parameter substitution will be addressed to find several approaches to solve the problem of replacement with real values. A given String with several substitute parameters should be replaced by actual values.

Several Approaches:

There are several approaches can be experimented to address string substitution with actual values. The approaches can be recursive new string, recursive string re-uses and comprehension and folding approach. These approaches detail out the statements to achieve the solution. Once these approaches are evaluated a better solution can be recommended.

Given Inputs

```
val inp = "select col1 from tab1 where id > ${props-a} and name like ${attrs-o}nothing works"
val maps = Map(
  "props" -> Map("a" -> "one", "b" -> "two"),
  "attrs" -> Map("z" -> "Zebra", "o" -> "Ox")
          )

val stringPattern = "\\$\\{([0-9a-zA-Z\\._-]+)\\}"
val regexPattern = stringPattern.r
def fetch(key: String): String = {
  var ret = ""
  val splts = key.split("-")
  if (!splts.isEmpty && splts.length == 2) {
    ret = maps.getOrElse(splts(0), Map()).getOrElse(splts(1), "")
  }
  ret
}
```

Recursive approach:

A recursive function which breaks the problems into multiple segments and calls itself to solve each segment or calling itself again different parameter values. The called function output will be combined with calling output. This combined output from all recursive function will generated final result.

```
def sbRecursive(input: String): String = {
  regexPattern findFirstIn input match {
    case Some(index) =>
      sbRecursive (input.replaceFirst(stringPattern, fetch(index.replaceAll("[$|{|}]",""))))
    case _ => input
  }
}
```

StringBuilder approach:

In the above recursive function, the output string is created many times for each recursive call. StringBuilder approach will eliminate generating new String every time but appends the output from multiple recursive function calls. In the end the StringBuilder instance will return the final result.

```
def subIndex(input: String, start: Int): Option[Int] = {
  regexPattern findFirstMatchIn input.substring(start) match {
    case Some(mach) => Some(start+mach.start)
    case _ => None
  }
}
def subLength(input: String, start: Int): Int = {
  val last = regexPattern findFirstMatchIn input.substring(start) match {
    case Some(mach) => start+mach.end
    case _ => start
  }
  last - start
}
def subParam(input: String, start: Int): String = {
  regexPattern findFirstIn input.substring(start) match {
    case Some(index) => index.replaceAll("[$|{|}]","")
    case _ => ""
  }
}
def sbRecursive (input: String): String = {
  val sb = new StringBuilder(input.length)
  def recursive(start: Int): String = if (start < input.length) {
    subIndex(input, start) match {
      case Some(index) => {
        sb.append(input subSequence (start, index))
        sb.append(fetch(subParam (input, index)))
        recursive (index + subLength(input, index))
      }
      case None => {
        sb.append(input subSequence (start, input.length))
        sb.toString
      }
    }
  } else {
    sb.toString
  }
  recursive(0)
}
```

Comprehension approach:

Scala has feature of comprehension in the form of for enumerator yield e, where enumerators refer to list of enumerators separated by semicolon. An enumerator is either a generator which introduces a new variable or a filter. A comprehension evaluates the body e for each binding generated by enumerator and returns a sequence of these values. A given string with substitutable params will be divided into sequence of non-replaceable and replaceable chunks where replaceable chunks simply replaced with actual values. In the end these values will be formed as string to return its final result.

```
val regexPatternX = """((?:[^$]+|\$\{(?![0-9a-zA-Z\._-])\})+)|\$\{([0-9a-zA-Z\._-]+)\}""".r
def tokenize2X(input: String): Iterator[List[String]] =
regexPatternX.findAllIn(input).matchData.map(_.subgroups)
def comprehension(input: Iterator[List[String]]) =
  for (token <- input) yield (token: @unchecked) match {
    case List(_, placeholder: String) => fetch(placeholder)
    case List(other: String, _) => other
  }
def sbComprehension(input: String) =
  comprehension(tokenize2X(input)).mkString
```

Conclusion:

The above steps explained to resolve the final substituted string from parameters and actual values in different ways. In recursive way it creates new string from every recursive call and concatenates and StringBuilder way creates one string but appends the output from multiple recursive calls and comprehension divides tasks in sequential chunks to convert into final result. The better approach would be comprehension as it generates the final result without any complex concatenations manner or complex recursive fashion.

References

https://docs.scala-lang.org/