We develop a program in Haskell to enumerate the strings of a regular expression in increasing order; the order is defined below.

Let $z$ be a finite alphabet whose symbols are totally ordered. For finite strings $s$ and $t$ over $z$, $s$ is smaller than $t$, written $s < t$, means either (1) the length of $s$ is smaller than that of $t$, or (2) the lengths of $s$ and $t$ are equal and $s$ is lexicographically smaller than $t$, based on the ordering over $z$. Henceforth, all strings are finite.

Let $r$ be a regular expression over $z$. We enumerate the strings of $r$ in increasing order (with respect to $<$). A regular expression $r$ is defined as follows: (1) $r = \phi$; then $r$ denotes the empty set, (2) $r = \text{NIL}$; then $r$ contains only the null string, (3) $r = \text{''a''}$, for some symbol $a$ of the alphabet; then, $r$ contains ‘‘a’’ as the only string, or for regular expressions $u$ and $v$, (4) $r = (u \mid v)$; then $r$ contains the union of the strings in $u$ and $v$ (5) $r = (u \cdot v)$; then $r$ contains all the strings obtained by concatenating a string of $u$ with one of $v$, or (6) $r = (u^*)$; then $r$ contains all the strings obtained by concatenating strings of $u$ any finite number of times (including 0 times, which yields the null string). For the first three cases, enumeration is straightforward. We discuss the remaining cases next.

1 Enumeration Problem

Let $enum$ be a function such that for a regular expression $r$, $enum(r)$ is the ordered list of strings denoted by $r$. For $r$ of the form $u \mid v$, we apply $enum$ to both $u$ and $v$ and then merge the two lists to obtain $enum(r)$, i.e., using an infix operator $+++$ for function $merge$ that is yet to be defined,

$$enum(u \mid v) = (enum u) +++ (enum v)$$

**Note:** Henceforth, we use the Haskell convention that function application is left-associative, and it has the highest binding power. Therefore, parenthesis around the arguments can be eliminated in a number of cases.

For a regular expression of the form $u \cdot v$, we use a function $prod$, written as an infix operator $***$, to concatenate pairs of strings from the argument lists.
\[
\text{enum}(u \cdot v) = (\text{enum } u) \ast\ast (\text{enum } v)
\]

Finally, for a regular expression of the form \(u^*\), we apply a function \(\text{closure}\).

\[
\text{enum}(u^*) = \text{closure}(\text{enum } u)
\]

The remaining task is to define \(\text{merge}\), \(\text{prod}\), and \(\text{closure}\), which we do next. Functions \(\text{merge}\) and \(\text{prod}\) have two arguments each; each argument is an ordered list of strings and the result is an ordered list of strings. Function \(\text{closure}\) has a single argument which is an ordered list of strings and its result is an ordered list of strings.

2 \text{ merge, prod, closure}

For comparisons of strings, we define a metric on a string as follows. Two strings can then be compared by lexicographically comparing the corresponding metrics.

\[
\text{metric } x = (\text{length } x, x)
\]

2.1 Definition of \(\text{merge}\)

Function \(\text{merge}\) has two ordered lists of strings, \(xs\) and \(ys\), as arguments; its result is an ordered list obtained by merging \(xs\) and \(ys\). There are no duplicates in the resulting list provided each of \(xs\) and \(ys\) is duplicate-free. (The Haskell notation \(xs@\(x:xt)\) denotes a list \(xs\) whose first element is \(x\) and the remaining part is the list \(xt\).)

\[
(\ast\ast) : \text{ [String] } \to \text{ [String] } \to \text{ [String]}
\]

\[
[] \ast\ast y = y
\]

\[
x \ast\ast [] = x
\]

\[
x@\(x:xt) \ast\ast y@\(y:yt) =
\]

\[
\text{case compare (metric } x\) (metric } y\) \text{ of}
\]

\[
\text{LT } \to x : (xt \ast\ast y)
\]

\[
\text{EQ } \to x : (xt \ast\ast yt)
\]

\[
\text{GT } \to y : (xs \ast\ast yt)
\]
2.2 Definition of \textit{prod}

Function \textit{prod} has two ordered lists of strings, \textit{xs} and \textit{ys}, as arguments; its result is an ordered list obtained from the concatenations of pairs of strings, from \textit{xs} with those from \textit{ys}. There are no duplicates in the resulting list provided each of the argument lists is duplicate-free.

\begin{equation*}
(\ast\ast\ast) :: [\text{String}] \rightarrow [\text{String}] \rightarrow [\text{String}]
\end{equation*}

\begin{align*}
\ast\ast\ast \; [] & = [] \\
\ast\ast\ast \; [\_] & = [] \\
\ast\ast\ast \; (x:xt) \ast\ast\ast (y:yt) & = (x \ast\ast y) : ((\text{map } (x \ast\ast ) \; \text{yt}) \ast\ast\ast (xt \ast\ast ys))
\end{align*}

The last case needs some explanation. The two argument lists are \((x:xt)\) and \((y:yt)\); the arguments are also denoted by \textit{xs} and \textit{ys}, respectively. Since \textit{xs} and \textit{ys} are assumed to be ordered, \textit{x} is the smallest string in \textit{xs} and \textit{y} in \textit{ys}. Hence, \textit{x \ast\ast y} is the smallest string in \textit{xs \ast\ast ys}, from the monotonicity of concatenation in both of its arguments. The remaining strings in the result list are of two kinds: (1) those that use \textit{x} as part of a concatenation, i.e., the strings resulting from concatenating \textit{x} with the strings in \textit{yt}, and (2) those that do not, namely \textit{xt \ast\ast ys}. The strings in (1) are ordered because \textit{yt} is ordered. And, those in (2) are ordered inductively. Therefore, merging them results in a single ordered list.

2.3 Definition of \textit{closure}

Function \textit{closure} applied to an ordered list of strings, \textit{xs}, creates finite concatenations of all the strings in \textit{xs}. Consider three cases (in the following "" is the null string): (1) \([\_]^* = ["""]\), (2) "" : \textit{xs})* = \textit{xs}*, and (3) if \textit{xs} has no null string, then \textit{xs} = "" : (\textit{xs} \ast\ast \textit{xs})*. Note that the order in which the three cases are written matters; in the last case, \textit{xs} is a list which does not contain the null string.

\begin{align*}
closure :: [\text{String}] \rightarrow [\text{String}]
closure [] & = ["""] \\
closure ("" : xt) & = \text{closure xt} \\
closure \textit{xs} & = ("" : (\textit{xs} \ast\ast \text{closure \textit{xs}}))
\end{align*}
3 The Complete Haskell program

The following program, written in Haskell, uses exactly the same code except for minor syntactic differences. First, we define regular expressions, `Rexp`, over a type `z`. We use infix operators `|` and `.` for alternation and concatenation.

```haskell
-- Define concatenation and alternation to be associative.
-- concatenation has higher binding power than alternation.

infixr 5 :|
infixr 6 :.

data Rexp z =
  Phi   -- empty language
| Nil   -- language containing null string
| Single z -- symbol from the alphabet
| (Rexp z) :| (Rexp z) -- alternation
| (Rexp z) :. (Rexp z) -- concatenation
| Star (Rexp z) -- Kleene closure

The result of function `enum` is an ordered list of strings of a regular expression.

```haskell
enum :: (Ord z, Show z) => Rexp z -> [String]
enum z = case z of
  Phi     -> []       -- empty language
  Nil     -> ["" ]    -- language containing null string only
  Single x -> [show x] -- convert x to string
  x :| y   -> (enum x) +++ (enum y)
  x :. y   -> (enum x) *** (enum y)
  Star x   -> closure (enum x)
```

The following functions — `merge, prod, and closure`— are as given before.

```haskell
(+++) :: [String] -> [String] -> [String]
[] +++ ys = ys
xs +++ [] = xs
xs@(x:xt) +++ ys@(y:yt) =
```
case compare (metric x) (metric y) of
  LT -> x : (xt +++ ys)
  EQ -> x : (xt +++ yt)
  GT -> y : (xs +++ yt)

(***) :: [String] -> [String] -> [String]
[] *** _ = []
_ *** [] = []
xs@(x:xt) *** ys@(y:yt)
    = (x ++ y): ((map (x ++ ) yt) +++ (xt *** ys))

closure :: [String] -> [String]
closure[] = ["""]
closure ("": xt) = closure xt
closure xs = ("": (xs *** (closure xs)))

Acknowledgment  The Tuesday Afternoon Club, under the guidance of Edsger W. Dijkstra, read and commented on an earlier draft of this manuscript. Pete Manolios developed a program in ACL-2 based on the earlier draft. The current version is helped by several features of Haskell, the type definition mechanism and lazy evaluation, in particular. I am indebted to Doug McIlroy who provided me with the merge and prod functions, and explored a number of other solutions with me; the version of prod shown here is slightly revised. Ham Richards has helped me with the intricacies of Haskell.