lib/search/symbol-comparison-exercise.ath

```
For another example, suppose we have a binary tree containing symbols:
\begin{Athena}
domain D
declare alpha, beta, gamma, delta, epsilon, phi, pi: D

define tree2 :=
  (node (node null
              alpha
              (node (node null
                          beta
                          null)
                    delta
                    null))
        epsilon
        (node (node null
                    gamma
                    null)
             phi
              (node null
                    pi
                    null)))
\end{Athena}
%
In Athena there is no built-in definition comparison operator on
symbols, so we must create one.  We can convert a symbol to a string
containing the characters of the symbol's name (recall that in Athena
a string is a list of characters); e.g.,
\begin{AthenaIO}
> (symbol->string alpha)

List: ['a 'l 'p 'h 'a]
\end{AthenaIO}
%
We can compare characters with the \code{compare} procedure, so we can
define a lexicographical ordering of strings, but we run into the
problem that neither characters nor strings are considered to be
terms, and we need terms in order to be able to define functions
axiomatically.  One way we can proceed is to define a new
\code{String} sort, which differs from the built-in \code{string} in
using \code{List}s rather than Athena lists \K{They would be \code{List}s of
what sort? Characters are not a sort.},
define \code{String.<} as lexicographical ordering of \code{String}s, define an instance of
\code{binary-search} that works with such \code{Strings}, and then use
input expansion to obtain a function that accepts a symbol and a
symbol-tree.  And we can use output transformation to translate the
resulting \code{String}-tree into a symbol-tree.
\K{I think we should omit or rework this example. This assumes that
a symbols are to be ordered with respect to their names, but symbols
are usually ordered in accordance with their intended meanings, not
their printed representations, e.g. \smtt{zero} might precede \smtt{one}
even though the string ``zero'' comes after the string ``one''. I think
you can either replace symbols with meta-identifiers (\smtt{'a}, \smtt{'foo}, etc.),
or else you can just assume or define a comparison function on domain D and use
that. If we go the latter route it'd be better if D were a datatype rather than
an open domain. But I think it'd be best to leave out this exercise altogether.}
\begin{Exercise}
  Work out the details of the approach just described, and test your
  resulting \code{bs-symbol} function with the following evaluations:
\begin{Athena}
(eval (bs-symbol gamma tree2))
(eval (bs-symbol beta tree2))
(eval (bs-symbol beta tree2))
(eval (bs-symbol epsilon tree2))
\end{Athena}
\begin{sol} ~
\begin{Athena}
define-sort charcode := Int
```

```
define-sort String := (List charcode)
module String {
declare <: [String String] -> Boolean
define [x y A B] := [?x:charcode ?y:charcode ?A:String ?B:String]
module < {
assert definition :=
   (fun [((x :: A) < (y :: B)) <==> (x Top.< y | x = y & A < B)
         (nil < (y :: B))      <==> true
         ((x :: A) < nil)      <==> false
         (nil < nil)           <==> false])
} # close module <
} # close module String

declare bs-symbol: [String (BinTree String)] -> (BinTree String)

define swo1 := (renaming [SWO.binary-search bs-symbol  SWO.< String.<])

assert (swo1 SWO.binary-search.axioms)

define string->String :=
  lambda (s)
    letrec {loop := lambda (L acc)
                      match L {
                        (list-of x rest) => (loop rest ((char-ord x) :: acc))
                      | [] => acc
                      | _ => L
                      }}
     (loop (rev s) nil)

define symbol->String := (o string->String symbol->string)

define map-tree :=
  lambda (f T)
    match T {
      (node L x R) => (node (map-tree f L) (f x) (map-tree f R))
    | null => null
    }

expand-input bs-symbol [symbol->String lambda (S) (map-tree symbol->String S)]

define String->string :=
  lambda (S)
    letrec {loop := lambda (S acc)
                      match S {
                        (x :: M) => (loop M (add (char x) acc))
                      | nil => (rev acc)
                      }}
     (loop S [])

transform-output eval [lambda (S) (map-tree (o string->symbol String->string) S)]
\end{Athena}
\end{sol}
\end{Exercise}
```