<div align="center">

Isabelle/HOL Exercises
Advanced

</div>

Tries

Section 3.4.4 of the Isabelle/HOL tutorial is a case study about so-called *tries*, a data structure for fast indexing with strings. Read that section.

The data type of tries over the alphabet type `'a` und the value type `'v` is defined as follows:

datatype `('a, 'v) trie = Trie "'v option" "('a * ('a,'v) trie) list"`

A trie consists of an optional value and an association list that maps letters of the alphabet to subtrees. Type `'a option` is defined in Section 2.5.3 of the Isabelle/HOL tutorial.

There are also two selector functions `value` and `alist`:

consts `"value" :: "('a, 'v) trie ⇒ 'v option"`
primrec `"value (Trie ov al) = ov"`

consts `alist :: "('a, 'v) trie ⇒ ('a * ('a,'v) trie) list"`
primrec `"alist (Trie ov al) = al"`

Furthermore there is a function `lookup` on tries defined with the help of the generic search function `assoc` on association lists:

consts `assoc :: "('key * 'val)list ⇒ 'key ⇒ 'val option"`
primrec
  `"assoc []     x = None"`
  `"assoc (p#ps) x =`
          `(let (a, b) = p in if a = x then Some b else assoc ps x)"`

consts `lookup :: "('a, 'v) trie ⇒ 'a list ⇒ 'v option"`
primrec
  `"lookup t []    = value t"`
  `"lookup t (a#as) = (case assoc (alist t) a of`
                        `None ⇒ None`
                      `| Some at ⇒ lookup at as)"`

Finally, `update` updates the value associated with some string with a new value, overwriting the old one:

consts `update :: "('a, 'v) trie ⇒ 'a list ⇒ 'v ⇒ ('a, 'v) trie"`

primrec
```
"update t []      v = Trie (Some v) (alist t)"
"update t (a#as) v =
   (let tt = (case assoc (alist t) a of
                None ⇒ Trie None []
              | Some at ⇒ at)
    in Trie (value t) ((a, update tt as v) # alist t))"
```

The following theorem tells us that `update` behaves as expected:

theorem `"∀ t v bs. lookup (update t as v) bs =`
` (if as = bs then Some v else lookup t bs)"`

As a warm-up exercise, define a function

consts `modify :: "('a, 'v) trie ⇒ 'a list ⇒ 'v option ⇒ ('a, 'v) trie"`

for inserting as well as deleting elements from a trie. Show that `modify` satisfies a suitably modified version of the correctness theorem for `update`.

The above definition of `update` has the disadvantage that it often creates junk: each association list it passes through is extended at the left end with a new (letter,value) pair without removing any old association of that letter which may already be present. Logically, such cleaning up is unnecessary because `assoc` always searches the list from the left. However, it constitutes a space leak: the old associations cannot be garbage collected because physically they are still reachable. This problem can be solved by means of a function

consts `overwrite :: "'a ⇒ 'b ⇒ ('a * 'b) list ⇒ ('a * 'b) list"`

that does not just add new pairs at the front but replaces old associations by new pairs if possible.

Define `overwrite`, modify `modify` to employ `overwrite`, and show the same relationship between `modify` and `lookup` as before.

Instead of association lists we can also use partial functions that map letters to subtrees. Partiality can be modelled with the help of type `'a option`: if `f` is a function of type `'a ⇀ 'b`, let `f a = Some b` if `a` should be mapped to some `b`, and let `f a = None` otherwise.

datatype `('a, 'v) trieP = Trie  "'v option" "'a ⇒ ('a,'v) trieP option"`

Modify the definitions of `lookup` and `modify` accordingly and show the same correctness theorem as above.