Isabelle/HOL Exercises
Advanced

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`:

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

primrec alist :: "('a, 'v) trie ⇒ ('a * ('a,'v) trie) list" where
"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:

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

primrec lookup :: "('a, 'v) trie ⇒ 'a list ⇒ 'v option" where
  "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:

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

```
lemma [simp]: "lookup (Trie None []) as = None"
  by (case_tac as, simp_all)

declare Let_def[simp] option.split[split]

theorem [simp]: "∀t v bs. lookup (update t as v) bs =
                    (if as = bs then Some v else lookup t bs)"
  apply (induct_tac as, auto)
  apply (case_tac[!] bs, auto)
done
```

As a warm-up exercise, define a function

`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`.

```
primrec modify :: "('a, 'v) trie ⇒ 'a list ⇒ 'v option ⇒ ('a, 'v) trie"
where
  "modify t []     vo = Trie vo (alist t)"
| "modify t (a#as) vo =
     (let tt = (case assoc (alist t) a of
                 None ⇒ Trie None []
               | Some at ⇒ at)
      in Trie (value t) ((a, modify tt as vo) # alist t))"

theorem [simp]: "∀t v bs. lookup (modify t as v) bs =
                    (if as = bs then v else lookup t bs)"
  apply (induct_tac as, auto)
  apply (case_tac[!] bs, auto)
done
```

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

*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.

```
primrec overwrite :: "'a ⇒ 'b ⇒ ('a * 'b) list ⇒ ('a * 'b) list" where
  "overwrite a v []     = [(a,v)]"
| "overwrite a v (p#ps) = (let (b, u ) = p in (if a=b then (a,v)#ps else (b,u) #
overwrite a v ps))"

lemma [simp]: "∀ a v b. assoc (overwrite a v ps) b = assoc ((a,v)#ps) b"
  by (induct_tac ps, auto)

primrec modify2 :: "('a, 'v) trie ⇒ 'a list ⇒ 'v option ⇒ ('a, 'v) trie"
where
  "modify2 t []     vo = Trie vo (alist t)"
| "modify2 t (a#as) vo =
     (let tt = (case assoc (alist t) a of
                 None ⇒ Trie None []
               | Some at ⇒ at)
      in Trie (value t) (overwrite a (modify2 tt as vo) (alist t)))"

theorem "∀t v bs. lookup (modify2 t as vo) bs =
                    (if as = bs then vo else lookup t bs)"
  apply (induct_tac as, auto)
  apply (case_tac[!] bs, auto)
done
```

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 option*, let *f a = Some b* if *a* should be mapped to some *b*, and let *f a = None* otherwise.

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

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

```
primrec value3 :: "('a, 'v) trieP ⇒ 'v option" where
"value3 (TrieP ov m) = ov"

primrec mapping3 :: "('a,'v) trieP ⇒ 'a ⇒ ('a, 'v) trieP option" where
```

```
"mapping3 (TrieP ov m) = m"

primrec lookup3 :: "('a,'v) trieP ⇒ 'a list ⇒ 'v option" where
  "lookup3 t [] = value3 t"
| "lookup3 t (a#as) = (case mapping3 t a of
                        None ⇒ None
                      | Some at ⇒ lookup3 at as)"

lemma [simp]: "lookup3 (TrieP None (λc. None)) as = None"
  by (case_tac as, simp_all)

primrec modify3 :: "('a,'v) trieP ⇒ 'a list ⇒ 'v option ⇒ ('a,'v) trieP"
where
  "modify3 t []     vo = TrieP vo (mapping3 t)"
| "modify3 t (a#as) vo =
     (let tt = (case mapping3 t a of
                  None ⇒ TrieP None (λc. None)
                | Some at ⇒ at)
      in TrieP (value3 t)
              (λc. if c = a then Some (modify3 tt as vo) else mapping3 t c))"

theorem "∀t v bs. lookup3 (modify3 t as vo) bs =
                    (if as = bs then vo else lookup3 t bs)"
  apply (induct_tac as, auto)
  apply (case_tac[!] bs, auto)
done
```