Computer Systems Modeling and Verification (USEEN1)

Recursion on lists

1 Simple functions

Implement the following functions[1] using case analysis and recursion, if required.

a) `length[A](l: list[A]) -> int`

   returns the number of elements in the list `l`.

b) `is_empty[A](l: list[A]) -> bool`

   returns `True` if the list `l` is empty.

c) `append[A](l1: list[A], l2: list[A]) -> list[A]`

   returns the list that is the concatenation of `l1` and `l2`.

d) `hd[A](l: list[A]) -> A`

   returns the first element of `l`. It raises `ValueError` if `l` is empty.

   `tl[A](l: list[A]) -> list[A]`

   returns all but the first element of `l`. It raises `ValueError` if `l` is empty.

e) `head[A](l: list[A]) -> A | None`

   returns `None` if the list is empty, and `hd(l)` otherwise.

   `tail[A](l: list[A]) -> list[A] | None`

   returns `None` if the list is empty, and `tl(l)` otherwise.

f) `get_item[A](l: list[A]) -> tuple[A, list[A]] | None`

   returns `None` if the list is empty, and `(hd(l), tl(l))` otherwise.

g) `last[A](l: list[A]) -> A`

   returns the last element of *l*. It raises `ValueError` if `l` is `[]`.

h) `nth[A](l: list[A], i: int) -> A`

   returns the `i`$^{(th)}$ element of the list `l`, counting from 0. It raises `IndexError` if `i` $<$ 0 or `i` $\geqslant$ `len(l)`. We have `nth(l, 0) = hd(l)`, ignoring exceptions.

i) `take[A](l: list[A], i: int) -> list[A]`

   returns the first `i` elements of the list `l`. It raises `IndexError` if `i` $<$ 0 or `i` $>$ `len(l)`. We have `take(l, len(l)) = l`.

j) `drop[A](l: list[A], i: int) -> list[A]`

   returns what is left after dropping the first `i` elements of the list *l*. It raises `IndexError` if `i` $<$ 0 or `i` $>$ `len(l)`. It holds that `take(l, i) + drop(l, i) = l` when $0 \leqslant$ `i` $\leqslant$ `len(l)`. We also have `drop(l, length(l)) = []`.

1. This assignment is adapted from the documentation of the `List` module in the Standard ML Basis Library.

k) `reversed[A](l: list[A]) -> list[A]`

   returns a list consisting of `l`'s elements in reverse order.

l) `concat[A](l: list[list[A]]) -> list[A]`

   returns the list that is the concatenation of all the lists in `l` in order.

   `concat([l1, l2, ... ln]) = l1 + l2 + ... + ln`

2 Higher-order functions

a) `map[A, B](f: Callable[[A], B], l: list[A]) -> list[B]`

   applies `f` to each element of `l` from left to right, returning the list of results.

b) `find[A](f: Callable[[A], bool], l: list[A]) -> A | None`

   applies `f` to each element `x` of the list `l`, from left to right, until `f(x)` evaluates to `True`. It returns `x` if such an `x` exists; otherwise it returns `None`.

c) `filter[A](f: Callable[[A], bool], l: list[A]) -> list[A]`

   applies `f` to each element `x` of `l`, from left to right, and returns the list of those `x` for which `f(x)` evaluated to `True`, in the same order as they occurred in the argument list.

d) `exists[A](f: Callable[[A], bool], l: list[A]) -> bool`

   applies `f` to each element `x` of the list `l`, from left to right, until `f(x)` evaluates to `True`; it returns `True` if such an `x` exists and `False` otherwise.

e) `forall[A](f: Callable[[A], bool], l: list[A]) -> bool`

   applies `f` to each element `x` of the list `l`, from left to right, until `f(x)` evaluates to `False`; it returns `False` if such an `x` exists and `True` otherwise.

f) `tabulate[A](n: int, f: Callable[[int], A]) -> list[A]`

   returns a list of length `n` equal to `[f(0), f(1), ..., f(n-1)]`, created from left to right. It raises `ValueError` if `n` < 0.