(* power: int * int -> int
REQUIRES: k ≥ 0
ENSURES: power(n,k) -> n^k
*)

Binary Power

```
fun power (n: int, 0: int) : int = 1
  | power (n,k) = n * power (n, k-1)
```

(slow)

```
fun power (n: int, 0: int) : int = 1
  | power (n,k) =
      if (even k)
      then square (power (n, k div 2))
      else n * power (n, k-1)
```

(fast)

Proof By standard induction on k.

Base Case) k = 0

W.T.S. power(n,0) = 1 = n^0

IS) IH: For some k ≥ 0, power(n,k) = n^k

power (n, k+1)

=> n * power (n, k+1-1)

=> n * power (n, k) => n^k ✓

proof By strong induction on k.

BC) Same as left

IH) For some value k ≥ 0, for every value n, for every $0 \le k' < k$, power (n, k') -> $n^{k'}$

even (k) -> false --- same as before

-> true square (power (n, k div 2))

=> square ($n^{k'}$) $2k' = k$ $k' < k$

=> $n^{k'} \times n^{k'}$ => $n^{2k'}$ => n^k

Lists

Type : t list for any type t.

Value : [v_1, ..., v_n] with each v_i (value) & n ≥ 0

Expressions : All the values e::es with e:t & es:t list

e.g. 1 :: [2,3] = [1,2,3] : int list = 1::2::3::nil

Evaluation [] is a value (pronounced "nil")

e::es => e'::es if e => e'

v::es => v::es' if v is a value & es => es'

```
(* length: int list -> int *)
fun length ([]: int list) : int = 0
  | length (x::xs) = 1 + length xs
```

THM length is total

Claim : For all length (L) evaluates to some value

Proof By structural induction on L.

BC) L = [] length ([]) = 0

IS) IH: length (xs) -> v (some value)

W.T.S. length (x :: xs) -> v'

length (x :: xs)

=> 1 + length (xs) => 1 + v => v'

Tail Recursion

```
fun tlength ([]: int list, acc: int) : int = acc
  | tlength (x::xs, acc) = tlength (xs, 1+acc)
```

accumulator

tail call

efficiency ↗

fun length (L : int list) : int = tlength (L , 0)

A function is tail recursive if it's recursive and if it performs no computations after calling itself recursively. "carry the result with you"

THM For all values L : int list & acc: int, tlength(L, acc) ≅ (length L) + acc.

Proof By structural induction on L.

BC) tlength ([ ] , acc) = acc

IS) L = x :: xs for some values x * xs

IH: tlength (xs, acc') ≅ (length xs) + acc' for all values acc' : int

W.T.S. tlength (x :: xs, acc) ≅ length (x :: xs) + acc

tlength (x :: xs, acc) ≅ tlength (xs, 1 + acc) [2nd cl tlength]
≅ (length xs) + (1 + acc) [IH : acc' = 1 + acc]
≅ (1 + length (xs)) + acc [comm of +]
≅ length (x :: xs) + acc [2nd cl of length]

```
fun append ( [ ] : int list, Y : int list ) : int list = Y
  | append ( x :: xs, Y ) = x :: append ( xs, Y )
```

=> Time complexity $O(|x|) \cong O(n_1)$

Append is predefined in SML as the right - associative infix operator @

```
(* rev : int list -> int list
   REQUIRES : true
   ENSURES : rev L ==> L in reverse order
*)
fun rev ( [ ] : int list ) : int list = [ ]
  | rev ( x :: xs ) = ( rev xs ) @ [ x ]
```

$O(n^2)$

```
fun trev ( [ ] : int list, acc : int list ) : int list = acc
  | trev ( x :: xs, acc ) = trev ( xs, x :: acc )
```

$O(n_1)$

trev (L, acc) ≅ (rev L) @ acc

Proof BC) L = [ ]

IS) trev ([ ], acc) ≅ acc [1st clause trev] ≅ [ ] @ acc [1st clause of @] ≅ (rev [ ]) @ acc [1st clause of rev]