A close up of text

Description automatically generatedA close up of a math problem

Description automatically generatedA white background with black and pink text

Description automatically generated

**Tree functions:** in a tree -> type ‘a tree = Empty | Node of ‘a \* ‘a tree \* a’ tree

Insert:

Lookup: We use option because we might

Have None returned, so “Some(d)”

**Variables, Bindings, and Functions:** in a local binding, the new value to the variable applies only in that scope.

* A new binding adds the same variable on another stack level, that stack level is deleted after use.
* Declaring functions -> let name (parameters) = …
  + When declaring a function, all variable values are put on the stack
  + Must redefine function to get new bindings (or else it looks at old bindings)

**Recursive Functions:** In a tail recursive function, the compiler doesn’t consume additional stack space for each recursive call. Instead of creating new stack frames with each call, reuse current frame

A white background with black and pink text

Description automatically generated**Lists:** Append function exists: “@”

**Induction:**

* Inductive definitions give us

a way of thinking recursively

Example:

Induction proof of SUM:

For any list “l” thefunction sum “l”

correctly computes the sum of it’s

elements.

INDUCTIVE STEP:

IH: assume that for list “l”, sum “l” is correct

Now with function prove that with one additional element, “x::l” is correct.

Sum(x::l) = x + sum(l), we know sum(l) is correct, therefore we get what we expect.

BASE CASE:

Empty list []

Sum of [] = 0

**Higher order functions:**

* Takes as input a function and outputs a function
* Ex: (‘a -> ‘b) -> ‘a list -> ‘b list :::: Takes a function ‘a -> ‘b and an input of type ‘a list

Returning a function:

**Impossible tail recursion:**

let rec merge lst1 lst2 =

match lst1, lst2 with

| [], lst | lst, [] -> lst

| x1 :: xs1, x2 :: xs2 ->

if x1 < x2 then x1 :: merge xs1 lst2

else x2 :: merge lst1 xs2

* Example: let add x = fun y -> x+y
  + Let add\_five = add 5;;
  + Let result = add\_five 10;;
  + Result = 15

**Turning a function into tail recursive:**

* Identify recursive call
* Introduce accumulator
* A computer screen shot of white text

  Description automatically generatedFind initial value for accumulator

A screen shot of a computer program

Description automatically generated