Problem 1

* My answer is 100% original
I have refferred stack overflow for
"remove-duplicates" *

Ans

```
myFlatten([], _).

myFlatten([Head|Tail], Y) :-
    is_list(Head),
    myFlatten(Head, Z),
    myFlatten(Tail, A),
    append(Z, A, B),
    remove_duplicates(B, C),
    Y is C.
```

P.T.O.

```
myFlatten([Head | Tail], Y) :-
        \+ is_list(Head),
        myFlatten(Tail, Z),
        append([Hed | []], Z, A),
        remove_duplicates(A, B),
        Y is B.

remove_duplicates([], []).
remove_duplicates([Head | Tail], Result) :-
        member(Head, Tail),
        !,
        remove_duplicates(Tail, Result).
remove_duplicates([Head | Tail], [Head | Result]) :-
        remove_duplicates(Tail, Result).
```

—X—

Problem 2   * It is my orignal answer *

(1) Height is defined by induction on the structure of the formula of Preposition logic as

$$\left.\begin{aligned} height(\top) &= 1 \\ height(\bot) &= 1 \end{aligned}\right\}$$ atomic formulas height. (trivial).

$$hieght(p) = 1$$

$$height(\neg\phi) = 1 + height(\phi)$$

$$height(\phi \circ \psi) = \max(height(\phi), height(\psi)) + 1.$$

where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

where BNF of proposition logic is given by

$$\phi, \psi ::= \bot \mid \top \mid p \in A \mid (\neg\phi) \mid \phi \circ \psi$$

where A is the set of variables.

(5) $$\underbrace{\forall x \exists y [P(x,y)]}_{(i)} \rightarrow \underbrace{\exists x \forall y [Q(x,y)]}_{(ii)}$$

(1) Renaming variables

x, y to a, b in (i)

and x, y to c, d in (ii)

We have,

$$\forall a \exists b [P(a,b)] \longrightarrow \exists c \forall d [Q(c,d)]$$

(2) Prenex form

$$\forall a \exists b \exists c \forall d [P(a,b) \rightarrow Q(c,d)]$$

using ($P(x) \equiv \exists y \; P(x) \equiv \forall z \; P(x)$)

(3) Skolemisation

· First removing $\exists b$ using a function $f(a)$ since $\forall$ quantifier of variable a exists before b.

$$\forall a \exists b \exists c \forall d [P(a,b) \rightarrow Q(c,d)]$$

$$\equiv \forall a \exists c \forall d [P(a,f(a)) \rightarrow Q(c,d)]$$

$$\equiv \forall a \forall d [P(a,f(a)) \rightarrow Q(g(a),d)]$$

Similarly replaced $\exists c$ using function $g(a)$ in the above line.

<u>Now</u> considering

$$\forall a \forall d \left[P(a, f(a)) \rightarrow Q(g(a), d) \right]$$

$$\forall a \forall d \left[\neg P(a, f(a)) \vee Q(g(a), d) \right]$$

— X —

3 We know that, a formula is satisfiable if it has a mode m such that

$$m \models \phi$$

Now ~~valid~~ a formula is valid if and only if $\underline{\forall m}$

$$m \models \phi \text{ holds.}$$

$\therefore$

$$\forall m \quad m \models \phi$$

Now not valid would become

$$\neg \forall m \quad m \models \phi$$

$$\equiv \exists m \quad m \not\models \phi$$

Now we have to prove that model is SAT iff $\neg\phi$ is NOT valid,

therefore

$$\neg (\forall m \quad m \models \neg \phi)$$

$$\equiv \neg \forall m \, (m \rightarrow \neg\phi)$$

$$\equiv \exists m \, (\neg (m \rightarrow \neg\phi))$$

$$\equiv \exists m \, (m \rightarrow \phi)$$

$$\equiv \exists m \quad m \models \phi$$

Hence Q.E.D.

2021 MCS 2152 → This is my original answer.

Shray Gahallot.

(3) $C \equiv_\alpha \lambda x y z\ [(x\ (y\ z))]$

To prove $C =_\beta S\ (K\ S)\ K$

where
$$S \equiv_\alpha \lambda x y z\ [((x\ z))\ (y\ z))]$$
$$K \equiv_\alpha \lambda x y\ [x]$$

Lets first take term opening K.

$$(K\ S) \equiv (\lambda x y\ [x]\ S)$$
$$=_\beta \{S/x\}\ (\lambda x y\ [x])$$
$$\rightarrow_\beta \lambda y\ [S] \quad -(1)$$

Now lets take

$$S\ (K\ S)\ K$$
$$\equiv S\ (\lambda y\ [S])\ K$$

Now we know

$$S \equiv_\alpha \lambda xyz[((x z)(y z))]$$

$$\{a/x\}\{b/y\}\{c/z\}S \Rightarrow_\beta \lambda abc[((a c)(b c))]$$

Now,

$$S \quad (\lambda y[S]) \quad K$$

$$\equiv \{\lambda y[S]/a\}\{K/b\}\ S$$

$$=_\beta \lambda c[((\lambda y[S]\ c)\ (K\ c))]$$

$$=_\beta \lambda c[(S\ (K\ c))]$$

$$=_\beta \lambda c[(S\ (\lambda xy[x]\ c))]$$

$$=_\beta \lambda c[(S\ \lambda y\ c)] = \lambda c[\lambda xyz\ (xz)(yz)\ (\lambda y c)]$$

$$= \lambda cxy[c\ (x\ y)]$$

Renaming

$$= \lambda xyz[x\ (y z)]$$

$$\cancel{=_\beta \lambda c[(\lambda xyz[(x z)(y z)]\ c)]}$$

$$\cancel{=_\beta \lambda c[\{c/x\}(S)]}$$

$$\cancel{=_\beta \lambda c.[\lambda yz[((c z)(y z))]]}$$

— x —

Problem 100% orignal answer.

4

1

(a) To ensure that the arguments to functions are well typed. Since there can be operators in function, which expect a particullar type. So its better to catch type errors at compile time than run time violations.

(b) To ensure λ applications occur only btw terms of appropriate types so that the result is meaningful.

[Given in lecture notes].

(2) Type Enviornment basically helps us to capture any assumptions that we might make when assigning a type. which can be later used to verify the constraints.

(3) $\sigma = \tau \rightarrow 'a$

For an app. expr (X Y).

σ is the type of X.

τ is the type of Y

and a' is the type of result produced by the application expr result.

(5) **and** $\equiv \lambda x\, y\, [\text{ite } x \ y \ \text{False}]$

or $\equiv \lambda x\, y\, [\text{ite } x \ \text{True} \ y]$

not $\equiv \lambda x\, [\text{ite } x \ \text{False} \ \text{True}]$

2021MCS2152
These are my 100% original answers.
For part 2 I have referred my own solution, which i had submitted in assignment 1.

Q5. (1)

```
mymap func list =  foldr (\h res = (func h):res) [] list

{- Where h is the head of list...and res is the result stored….. -}

myconcat list = foldr (\h res = h++res) “” list

myconcatMap list func = myconcat (mymap func list)
```

Q5. (2)

```
merge l1 l2 = mergeHelper l1 l2 []
mergeHelper [] [] lnew  = lnew
mergeHelper l1 [] lnew  = lnew ++ l1
mergeHelper [] l2 lnew  = lnew ++ l2
mergeHelper (x:xs) (y:ys) lnew  =
	if x < y
	then mergeHelper xs (y:ys) (lnew++[x])
	else mergeHelper (x:xs) ys (lnew++[y])
```

{-
Proof of Correctness :
Assumptions-> Input lists l1,l2 are sorted in ascending order.

Loop invariant => At start of the each recursive call 'n' of mergeHelper,
we have 'lnew' containing the smallest 'n-1' elements of
l1 and l2 in sorted order. Also the head of l1 and l2 depict
the smallest elements in list l1 and l2.

Initialisation -> At the first call, the list 'lnew' is empty and head of l1 and l2
depict the smallest element in l1 and l2.

Maintainence -> Now, only the smaller element of l1[0] and l2[0]
is appened to the list 'lnew'. Then the function is recursively calls again.

Termination -> Case 1, l2 becomes empty : In this case, l1 is appened to lnew, since all
elements
in l1 are greater than the elements in lnew. And l1 is a sorted list.

Case 2, l1 becomes empty : In this case, l2 is appened to lnew, since all elements in l1 are greater than the elements in lnew. And l2 is a sorted list.

Hence, our new list, 'lnew' contains the merged elements in sorted fashion.

Time Complexity :
T(n,m) = O(n + m)
since, we will iterate over both the list of size n and m.
Hence, O(n+m).
-}