

```
(* CSC324 Assignment 2a *)
```

```
(*****  
  Some help functions  
  *****)
```

```
(* reverse a list *)
```

```
fun shunt([],L) = L  
  | shunt (x::L1, L2) = shunt(L1, x::L2)
```

```
fun rev L = shunt(L,[]);
```

```
(* member function *)
```

```
fun member(x,[]) = false  
  | member(x,y::list) = if x=y then true else member(x,list);
```

```
(*****  
  Question 1  
  *****)
```

```
(* Q1a *)
```

```
fun sumTimes1(L: (int*int) list)  
  = if (null L)  
    then 0  
    else #1(hd L) * #2(hd L) + sumTimes1(tl L);
```

```
(* Q1b *)
```

```
fun sumTimes2 [] = 0  
  | sumTimes2((I,J)::L) = I*(J:int) + (sumTimes2 L);
```

```

(*****
  Question 2
  *****)

(* Q2(a) *)

type employee = { id:int,
                  name: string,
                  age: int,
                  salary: int};

(* Q2(b) *)

fun avgAgeHelp [] = 0
  | avgAgeHelp(record::tl : employee list)
    = #age record + avgAgeHelp(tl);

fun avgAge(L: employee list)
  = Real.fromInt(avgAgeHelp(L)) / Real.fromInt(length(L));

(* Q2(c) *)

fun payroll [] = 0
  | payroll ((record::tl): employee list)
    = #salary(record) + payroll(tl);

(* Q2(d) *)

fun personnel([],: employee list) = []
  | personnel(e::elist) = (#id e, #name e)::personnel(elist)

(* Q2(e) *)

fun increase([],:employee list, [],:int list) = []
  | increase({id=i,name=n,age=a,salary=s}::elist, r::rlist)
    = {id=i,name=n,age=a,salary=s+r}::increase(elist,rlist);

```

```

(*****
  Question 3
  *****)

(* Q3a *)

datatype grade = A | B | C | D | F;

(* Q3b *)

fun convert(N:int) =
  if (N>=85)
  then A
  else if (N>=73)
  then B
  else if (N>=67)
  then C
  else if (N>=57)
  then D
  else F;

(* Q3c *)

fun   gp A = 4
    | gp B = 3
    | gp C = 2
    | gp D = 1
    | gp F = 0

(* Q3d *)

fun  gpah [] = 0
    | gpah (a::x : grade list) = gp(a) + gpah(x);

fun gpa L = Real.fromInt(gpah(L)) / Real.fromInt(length(L));

```

```

(*****
  Question 4
  *****)

(* Q4a *)

datatype tree = Node of (int*tree list);

(* Q4b *)

fun countH [] = 0
  | countH (a::x) = count(a) + countH(x)

and count (Node(n, [])) = 1
  | count (Node (n, (x::y))) = count(x) + countH(y);

(* Q4c *)

fun numH [] = []
  | numH (a::x) = numbers(a) @ numH(x)

and numbers (Node(n, [])) = n::nil
  | numbers (Node (n, (x::y))) = [n] @ numbers(x) @ numH(y);

(* Q4d *)

fun doubleH [] = []
  | doubleH (a::x) = double(a) :: doubleH(x)

and double (Node(n, [])) = Node(2*n, [])
  | double (Node (n, (x::y))) = Node (2*n, double(x) :: doubleH(y));

(* define some values *)

val bt5 = Node(5,[]);
val bt2 = Node(2,[]);
val bt4 = Node(4,[]);
val bt3 = Node(3,[bt5]);
val bt1 = Node(1,[bt2, bt3, bt4]);

```

```

(*****
  Question 5
  *****)

type City = string;
type Road = City * City;
type Roads = Road list;
type Cities = City list;
type RoadMap = Cities * Roads;

type AdjList = City * Cities; (* An adjacency list *)
type RoadAdj = AdjList list; (* Road map represented as adjacency
lists *)

(* get a list of cities which connect to a city. *)

fun connection(city:City, []:Roads):Cities = []
  | connection(city, (city1, city2)::roads) =
    if city=city1 then city2::connection(city, roads)
    else if city=city2 then city1::connection(city, roads)
    else connection(city, roads);

(* transform one city to an adjacency list. *)

fun transformOne(city:City, roads:Roads):AdjList
  = (city, connection(city, roads));

(* transform a list of cities to a list of adjacency lists. *)

fun transform([]:Cities, _:Roads):RoadAdj = []
  | transform(city::cities, roads) =
    transformOne(city, roads)::transform(cities, roads);

(* find neighbours of a city. *)

fun neighbours (city:City, []:RoadAdj):Cities = []
  | neighbours (city, (roadAdj1::roadAdj)) =
    if city = #1 roadAdj1 then #2 roadAdj1
    else neighbours(city, roadAdj);

```

```

(* Extend path by adding city to it *)

fun extend(city:City, path:Cities, roadAdj:RoadAdj, n):Cities =
  if member(city, path) then []
  else choose(neighbours(city, roadAdj), city::path, roadAdj, n-1)

(* Extend path by choosing a city from the cities list *)

and choose(_:Cities, path:Cities, _:RoadAdj, 0):Cities = path
  | choose([], _, _, _) = []
  | choose(cities, path, roadAdj, n) =
    let val tmp = extend(hd(cities), path, roadAdj, n)
    in if tmp=[] then choose(tl(cities), path, roadAdj, n)
    else tmp
    end;

(* Find a Hamiltonian path in RoadMap string at city. *)

fun path((cities, roads):RoadMap, city:City):Cities =
  rev(choose([city], [], transform(cities, roads), length(cities)));

```