8 Debugging and Functional Programming (16 points)

Consider these OCaml programs that *do not type-check* and their corresponding error messages (including the implicated code, shown underlined). Each has comments detailing what the program *should* do as well as sample invocations that *should* type-check.

```
(* "sumlist xs" returns the sum of the
    integer elements of "xs" *)
let rec sumList xs = match xs with
    | []       -> []
    | y ::  ys -> y + sumList ys

assert( sumList [1;2] = 3 );;

This expression has type
    'a list
but an expression was expected of type
    int
```

```
(* "wwhile (f, x)" returns x' where there exist
    values v_0, ..., v_n such that:

    - x is equal to v_0
    - x' is equal to v_n
    - for each i between 0 and n-2, we have
        (f v_i) equals (v_{i+1}, true)
    - (f v_{n-1}) equals (v_n, false) *)

let f x =
    let xx = x * x in
    (xx, (xx < 100))

let rec wwhile (f,b) =
    match f with
    | (z, false) -> z
    | (z, true)  -> wwhile (f, z)

assert( wwhile (f, 2) = 256 ) ;;

This expression has type
    int -> int * bool
but an expression was expected of type
    'a * bool
```

(a) *[2 pts]* Why is the `sumList` program not well-typed?

(b) *[2 pts]* Fix the `sumList` program.

(c) *[2 pts]* Why is the `wwhile` program not well-typed?

(d) *[2 pts]* Fix the `wwhile` program.

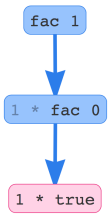
Consider an *execution trace* that shows a high-level overview of a program execution focusing on function calls. For example, the trace on the right tells us that:

i. We start off with `fac 1`.

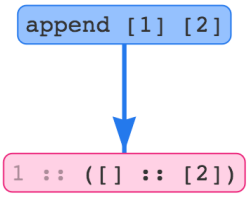
ii. After performing some computation, we have the expression `1 * fac 0`. The `1 *` is grayed out, indicating that `fac 0` is the next expression to be evaluated.

iii. When we return from `fac 0`, we are left with `1 * true`, indicating a program error: we cannot multiply an `int` with a `bool`.

```
let rec fac n =
    if n <= 0 then
        true
    else
        n * fac (n - 1)

assert( fac 1 = 1 ) ;;
```



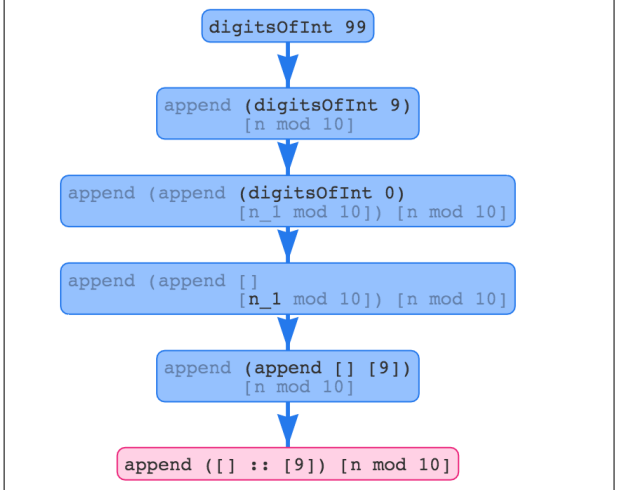

```
(* "append xs ys" returns a list containing the
   elements of "xs" followed by the elements of "ys" *)
let rec append xs ys =
    match xs with
    | []   -> ys
    | h::t -> h ::  t ::  ys

assert( append [1] [2] = [1;2] ) ;;

Error encountered because
    int
is incompatible with
    int list
```



(e) *[2 pts]* Why is the `append` program not well-typed?

(f) *[2 pts]* Fix the `append` program.

```
(* "digitsOfInt n" returns "[]" if "n" is
    not positive, and otherwise returns the
    list of digits of "n" in the order in
    which they appear in "n".  *)
let rec append x xs =
    match xs with
    | [] -> [x]
    | _  -> x ::  xs

let rec digitsOfInt n =
    if n <= 0 then
       []
    else
       append (digitsOfInt (n/10))
              [n mod 10]

assert( digitsOfInt 99 = [9;9] ) ;;

Error encountered because
    'a list
is incompatible with
    int
```



(g) *[2 pts]* Why is the `digitsOfInt` program not well-typed?

(h) *[2 pts]* Fix the `digitsOfInt` program.

9 Debugging and Functional Programming (16 points)

Consider these OCaml programs that *do not type-check* and their corresponding error messages (including the implicated code, shown underlined). Each has comments detailing what the program *should* do as well as sample invocations that *should* type-check.

```
(* "append xs ys" returns a list containing the
   elements of "xs" followed by the elements of "ys" *)
let rec append xs ys =
    match xs with
    | []   -> ys
    | h::t -> h ::  t ::  ys

assert( append [1] [2] = [1;2] ) ;;

This expression has type
    'a list
but an expression was expected of type
    'a
The type variable 'a occurs inside 'a list
```

```
(* "digitsOfInt n" returns "[]" if "n" is
    not positive, and otherwise returns the
    list of digits of "n" in the order in
    which they appear in "n".  *)
let rec append x xs =
    match xs with
    | [] -> [x]
    | _  -> x ::  xs

let rec digitsOfInt n =
    if n <= 0 then
       []
    else
       append (digitsOfInt (n/10))
              [n mod 10]

assert( digitsOfInt 99 = [9;9] ) ;;

This expression has type
    int
but an expression was expected of type
    'a list
```

(a) *[2 pts]* Why is the `append` program not well-typed?

(b) *[2 pts]* Fix the `append` program.

(c) *[2 pts]* Why is the `digitsOfInt` program not well-typed?

(d) *[2 pts]* Fix the `digitsOfInt` program.

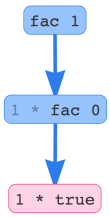
Consider an *execution trace* that shows a high-level overview of a program execution focusing on function calls. For example, the trace on the right tells us that:

i. We start off with `fac 1`.

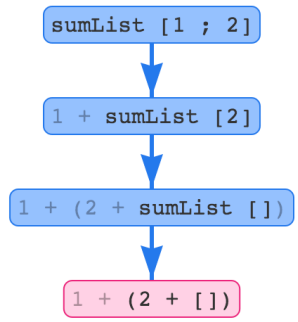
ii. After performing some computation, we have the expression `1 * fac 0`. The `1 *` is grayed out, indicating that `fac 0` is the next expression to be evaluated.

iii. When we return from `fac 0`, we are left with `1 * true`, indicating a program error: we cannot multiply an `int` with a `bool`.

```
let rec fac n =
    if n <= 0 then
        true
    else
        n * fac (n - 1)

assert( fac 1 = 1 ) ;;
```



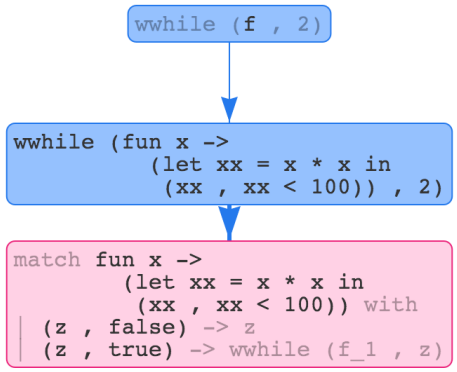

```
(* "sumlist xs" returns the sum of the
    integer elements of "xs" *)
let rec sumList xs = match xs with
    | []       -> []
    | y ::  ys -> y + sumList ys

assert( sumList [1;2] = 3 );;

Error encountered because
    'a list
is incompatible with
    int
```



(e) *[2 pts]* Why is the `sumList` program not well-typed?

(f) *[2 pts]* Fix the `sumList` program.

```
(* "wwhile (f, x)" returns x' where there exist
    values v_0, ..., v_n such that:

    - x is equal to v_0
    - x' is equal to v_n
    - for each i between 0 and n-2, we have
        (f v_i) equals (v_{i+1}, true)
    - (f v_{n-1}) equals (v_n, false) *)

let f x =
    let xx = x * x in
    (xx, (xx < 100))

let rec wwhile (f,b) =
    match f with
    | (z, false) -> z
    | (z, true)  -> wwhile (f, z)

assert( wwhile (f, 2) = 256 ) ;;

Error encountered because
    'a -> 'b
is incompatible with
    'c * 'd
```



(g) *[2 pts]* Why is the `wwhile` program not well-typed?

(h) *[2 pts]* Fix the `wwhile` program.