Write the *type* of each of the following OCaml expressions in the first blank provided, or *ill-typed* if the expression does not type check. Then, after the ⇒ symbol, write the most simplified *value* of the expression, or leave it blank if it's ill-typed.

(a) `let a : _____ = [3::[]]` ⇒ _____
(b) `let b : _____ = [(1::2)::[]]` ⇒ _____
(c) `let c : _____ = [1::[2;3;4]]` ⇒ _____
(d) `let d : _____ = [[1::2]::[3::4]]` ⇒ _____
(e) `let e : _____ = [[1;2;3;4]::[]]` ⇒ _____
(f) `let f : _____ = [(1::[])::[2]::[3;4]::[]]` ⇒ _____
(g) `let g : _____ = [(5,12,7);(2,5)]` ⇒ _____
(h) `let h : _____ = [[5;12;7];[2;5]]` ⇒ _____

Simplify the complex expression below, using step by step substitution. What is the value computed for 'answer' in the following program? (Note: the @ operator glues two lists of the same type together.)

```
let answer : int list =
  let list = [4;5] in
  let f (num : int) : int list = num :: list in
  let list = [6;7] in
  (f 3) @ list
```

Which of the following pieces of code are well-formed OCaml expressions? Refer to the production rules on the back of this sheet. For expressions that do not follow the syntax rules, write *ill-formed*. For those that do, add parentheses and/or underline subexpressions to clarify the boundaries of each expression. The first one had been done for you as an example.

(a) `if (x > 1) then if (y > 2) then 0 else 1 else if (z > 3) then 2 else 3`

if (x > 1) then (if (y > 2) then 0 else 1) else (if (z > 3) then 2 else 3)

(b) `let x = 5 in if (x = r) then 7 else x`

(c) `if (a = 0) then 9 else if (a = 1) then 8 else if (a = 2) then 7`

(d) `if (a = 0) then 9 else if (a = 1) then 8 else if (a = 2) then 7 else 6`

(e) `if (a = 0) then 9 else if (a = 1) then 8 else 7 else 6`

(f) `if (x < y) then let z = y-x in z*z else let z = x-y in z*z`

(g) `let b = if (c = 6) then d else e in if (f = b) then g else h`

Some OCaml expression rules for building programs (not exhaustive):

```
<top-expr> := <let-fun-expr> | <let-expr>

<expr> := <let-in-expr> | <let-fun-expr> | <if-expr> | <match-expr> | <value>

<let-fun-expr> := let <arg-decl>* : <type> = <expr>

      <arg-decl> := ( <identifier> : <type> )

<let-expr> := let <binding>

      <binding> := <identifier> = <expr>

<let-in-expr> := let <binding> in <expr>

<if-expr> := if <bool-expr> then <expr> else <expr>

<match-expr> := begin match <identifier> with <match-line>* end

      <match-line> := | <pattern> -> <expr>
```