midterm

Monday, October 20, 2014 6:31 PM

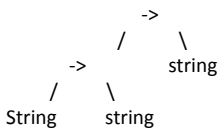
List.filter: (a'->bool) -> 'a list -> 'alist
(bool list -> bool) -> bool list list
List.head: 'b list -> b
Bool list -> bool

Nsub[1;2;3]
  nsub [2;3] = [[2;3];[2];[3]]
==> [1] [[1;2;3];[1;2];[1;3];[2;3];[2];[3]]

Fun x y is a curried definition
((Fun->x) ->y)

float -> ((string * (bool list)) -> int)

Arrow binds rightest to the right



Infix precedence binds more tightly the prefix

1::(2::[3])

xs@(ys@zs)<--- cheaper to do
(xs@ys)@zs

Experienced F# programmers do not ordinarily program with List.isEmpty, List.head, and List.tail, preferring to use pattern matching style. There are four advantages to using pattern matching:

1. It's pretty!
2. If you accidentally apply List.head or List.tail to an empty list, you get an exception. In contrast, the pattern x::xs matches only a nonempty list, safely binding x to the head and binding xs to the tail.
3. The F# compiler checks whether you have included enough patterns to cover all possible inputs. If not, it gives an *Incomplete pattern matches* warning.
4. The F# compiler checks whether a pattern is *redundant*, in the sense that no input will cause it to be used. If so, F# gives a *This rule will never be matched* warning.

Note that the indentation of nested match expressions can be significant:

```
  let foo x = function
  | 1 -> match x with
         | 0 -> true
         | 1 -> false
  | 2 -> true
```

val foo : x:int -> _arg1:int -> bool

```
> let roots (a,b,c) =
    let disc = sqrt (b*b - 4.0*a*c)
    let twoa = 2.0*a
    ((-b+disc)/twoa, (-b-disc)/twoa);;

val roots : float * float * float -> float * float
```

```
> let map f xs =
    let rec map_aux = function
    | []    -> []
    | y::ys -> f y :: map_aux ys
    map_aux xs;;

val map : ('a -> 'b) -> 'a list -> 'b
list
```

One possibility is to use the binding in effect where the function is *defined*; this is called *static scoping* (or *lexical scoping*). Another possibility is to use the binding in effect when the function is *called*; this is called *dynamic scoping*. Here's an example that shows that F# uses static scoping.

```
> let x = 3 in let f y = x+y
               let x = 6
               f x;;
val it : int = 9
```

From the point of view of language design, there is now a consensus that static scoping is better than dynamic scoping. The reason is that, under dynamic scoping, one cannot understand a function with free identifiers by simply studying its definition---at the time of a *call* to the function, there could be new bindings for the free identifiers that would completely change the function's behavior.

```
let myand (e1,e2) = if e1 then e2 else false;;

val myand : bool * bool -> bool
```

The trouble is that, under eager evaluation, we don't get short-circuit evaluation. Under eager evaluation, F# immediately evaluates both e1 and e2, even though myand needs the value of e2 only when e1 is true.

```
> let myand (e1, e2) = if e1() then e2() else false;;

val myand : (unit -> bool) * (unit -> bool) -> bool

> myand ((fun () -> x > 0), (fun () -> 10/x > 2));;
  val it : bool = false
```

- e1 :: e2
  Here the values of e1 and e2 are pointers *p1* and *p2*. We allocate a new cons cell, put *p1* and *p2* into it, and return a pointer to the new cons cell.
- List.isEmpty e
  Here the value of e is a pointer *p*. We just test whether *p* is the null pointer.
- List.head e
  Again, the value of e is a pointer *p*. If *p* is a null pointer, then we raise an ArgumentException. Otherwise, we return the *left* pointer of the cons cell that *p* points at.
- List.tail e
  Again, the value of e is a pointer *p*. If *p* is a null pointer, then we raise an ArgumentException. Otherwise, we return the *right* pointer of the cons cell that *p* points at.

```
let rec append = function
 | ([],    ys) -> ys
 | (x::xs, ys) -> x :: append (xs,ys)
```

If xs has length *n* and ys has length *m*, then notice that we get a total of *n*+1 invocations. And each of these invocations again does O(1) work. So in total the running time is O(*n*). In other words, xs @ ys takes time proportional to the length of xs but independent of the length of ys. This asymmetry is very important to keep in mind when you use @.

```
let rec split = function
| []       -> ([],[])
| [a]      -> ([a],[])
| a::b::cs -> let (M,N) = split cs
              (a::M, b::N)
```

```
| a::b::cs -> (a :: fst (split cs), b :: snd (split cs))
```

then we would recursively split cs *twice*. What are the asymptotic time complexities of these two definitions of split?

First, with either definition of split, it is clear that each invocation does a *constant* amount of work directly. How many recursive invocations are there on an input of length *n*? When $n > 1$, the first definition makes *one* recursive call on a list of length *n*-2, while the second definition makes *two* such calls.

Hence the first definition makes about *n*/2 recursive invocations total, and hence takes *linear* time, O(*n*).

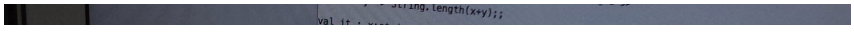
But the second definition makes 2 calls on lists of length *n*-2, 4 calls on lists of length *n*-4, 8 calls on lists of length *n*-6, 16 calls on lists of length *n*-8, and so on, until finally making (if *n* is even) $2^{n/2}$ calls on lists of length 0. Hence the second definition takes *exponential* time, $O(2^{n/2})$! (Try the two definitions on lists of size 55, and you will already see a huge difference in performance.)

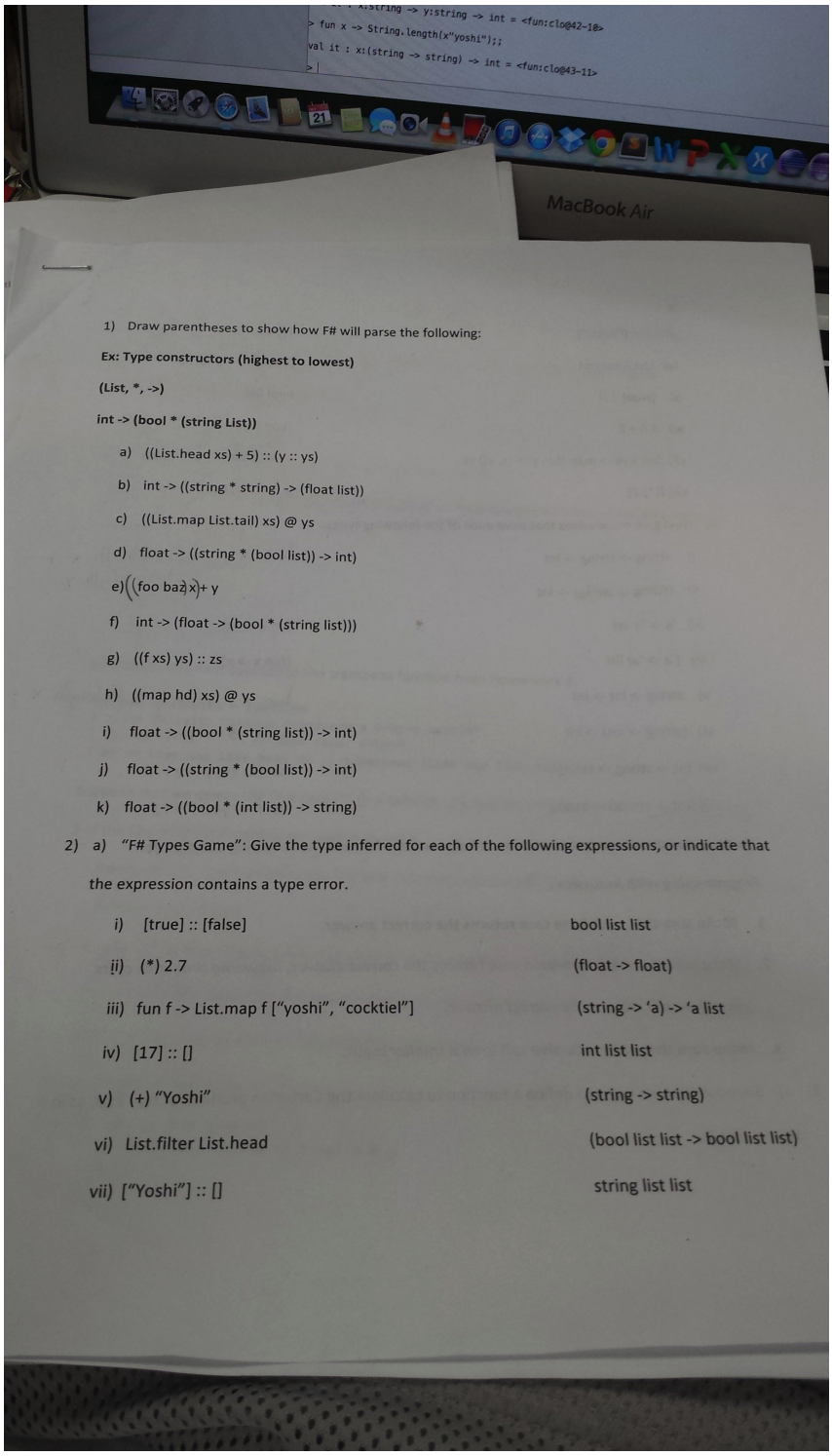
```
let rec rev = function
| []    -> []
| x::xs -> rev xs @ [x]
```

On a list of length *n*, this definition makes *n*+1 recursive invocations. But each invocation here does linear work, since it does an append where the left argument has length O(*n*) and the right argument has length 1. (This is an example where the asymmetry of the running time of @ is critical.) Hence this definition of rev takes *quadratic* time.

To develop your cases systematically, it helps to keep in mind the following *Checklist for Programming with Recursion*:

1. Make sure that each base case returns the correct answer.
2. Make sure that each non-base case returns the correct answer, *assuming that each of its recursive calls returns the correct answer*.
3. Make sure that each recursive call is on a *smaller* input.





1) Draw parentheses to show how F# will parse the following:

Ex: Type constructors (highest to lowest)

(List, *, ->)

int -> (bool * (string List))

a) ((List.head xs) + 5) :: (y :: ys)

b) int -> ((string * string) -> (float list))

c) ((List.map List.tail) xs) @ ys

d) float -> ((string * (bool list)) -> int)

e) ((foo baz) x) + y

f) int -> (float -> (bool * (string list)))

g) ((f xs) ys) :: zs

h) ((map hd) xs) @ ys

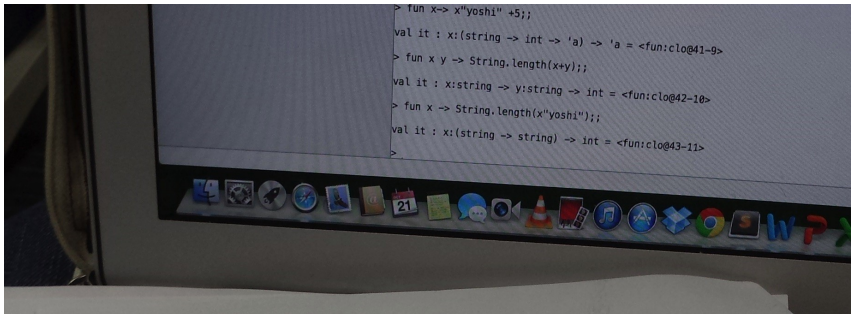
i) float -> ((bool * (string list)) -> int)

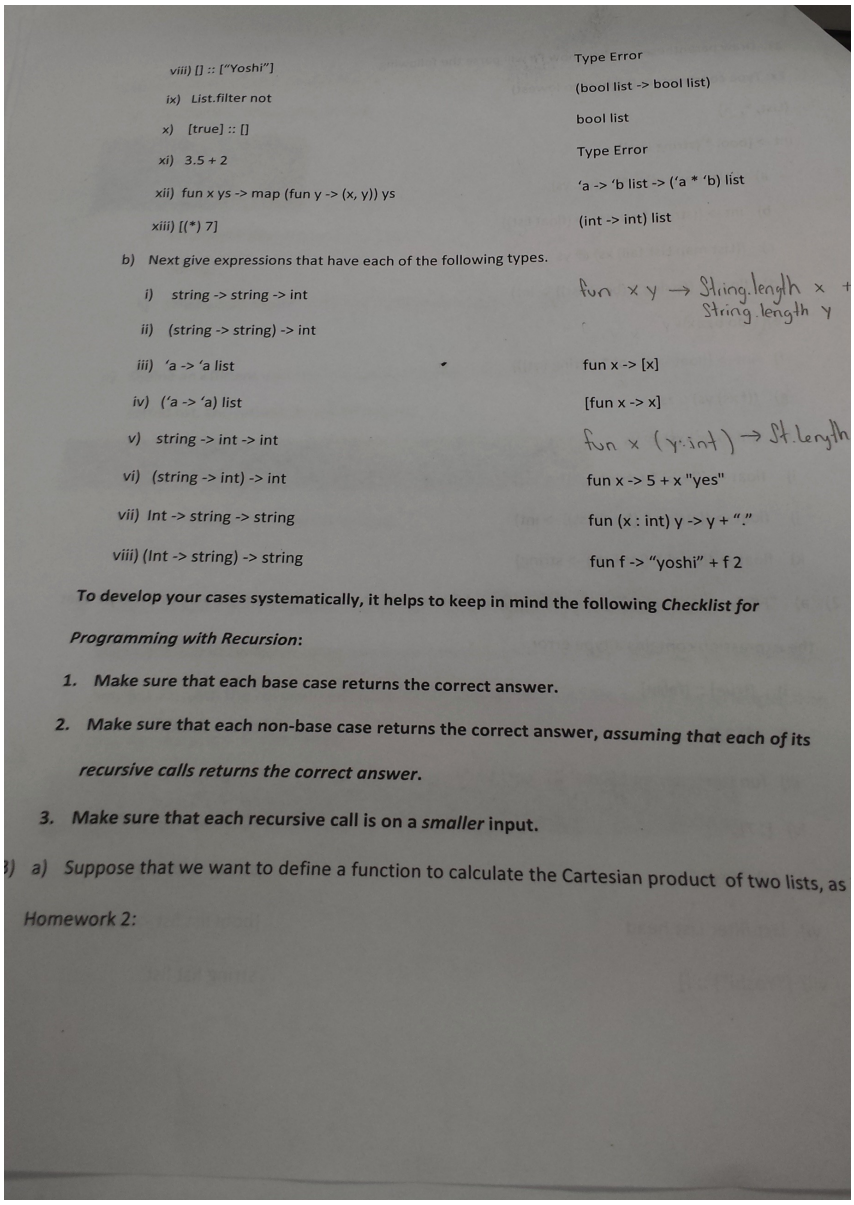
j) float -> ((string * (bool list)) -> int)

k) float -> ((bool * (int list)) -> string)

2) a) "F# Types Game": Give the type inferred for each of the following expressions, or indicate that the expression contains a type error.

i) [true] :: [false]	bool list list
ii) (*) 2.7	(float -> float)
iii) fun f -> List.map f ["yoshi", "cocktiel"]	(string -> 'a) -> 'a list
iv) [17] :: []	int list list
v) (+) "Yoshi"	(string -> string)
vi) List.filter List.head	(bool list list -> bool list list)
vii) ["Yoshi"] :: []	string list list





viii) [] :: ["Yoshi"] — Type Error

ix) List.filter not — (bool list -> bool list)

x) [true] :: [] — bool list

xi) 3.5 + 2 — Type Error

xii) fun x ys -> map (fun y -> (x, y)) ys — 'a -> 'b list -> ('a * 'b) list

xiii) [(*) 7] — (int -> int) list

b) Next give expressions that have each of the following types.

i) string -> string -> int — fun x y → String.length x + String.length y

ii) (string -> string) -> int

iii) 'a -> 'a list — fun x -> [x]

iv) ('a -> 'a) list — [fun x -> x]

v) string -> int -> int — fun x (y:int) → St.length

vi) (string -> int) -> int — fun x -> 5 + x "yes"

vii) Int -> string -> string — fun (x : int) y -> y + "."

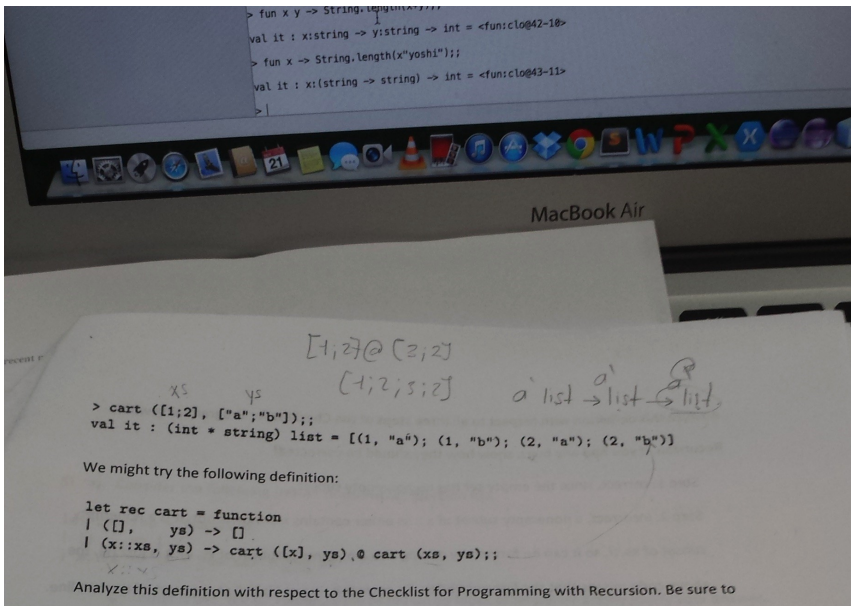
viii) (Int -> string) -> string — fun f -> "yoshi" + f 2

To develop your cases systematically, it helps to keep in mind the following Checklist for Programming with Recursion:

1. ***Make sure that each base case returns the correct answer.***
2. ***Make sure that each non-base case returns the correct answer, assuming that each of its recursive calls returns the correct answer.***
3. ***Make sure that each recursive call is on a smaller input.***

3) a) Suppose that we want to define a function to calculate the Cartesian product of two lists, as

Homework 2:



```
> fun x y -> String.length(x)+y;;
val it : x:string -> y:string -> int = <fun:clo@42-10>
> fun x -> String.length(x"yoshi");;
val it : x:(string -> string) -> int = <fun:clo@43-11>
>
```

MacBook Air

[1,2]@[3,2]

[1;2;3;2]

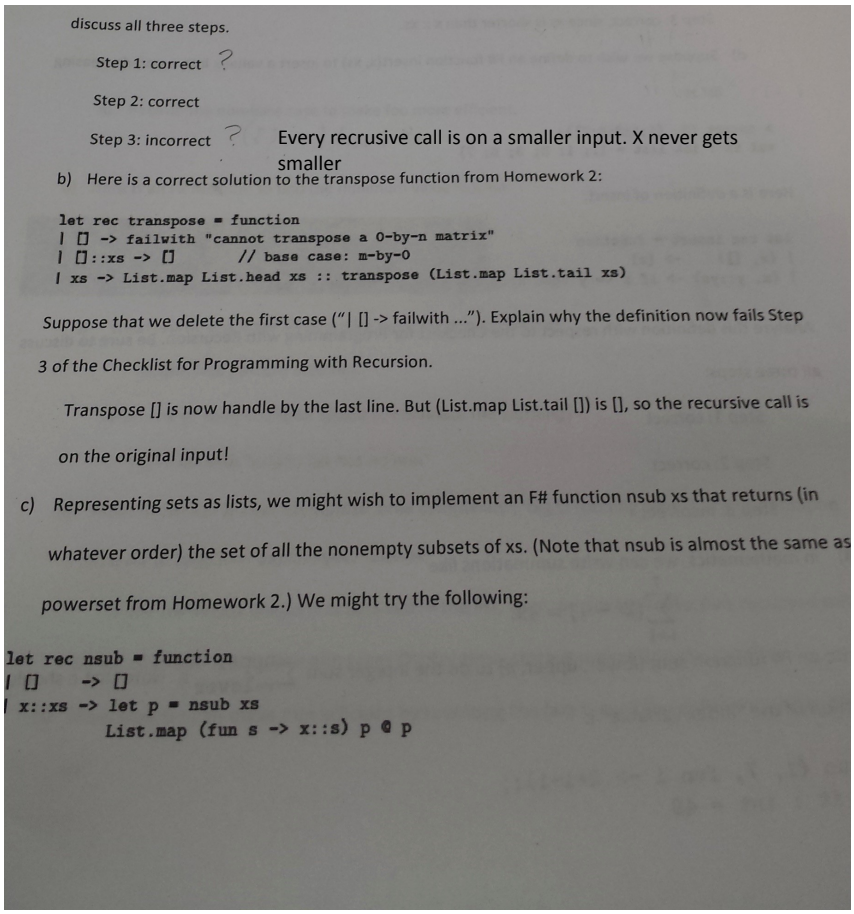
'a list -> 'a list -> 'a list

```
> cart ([1;2], ["a";"b"]);;
val it : (int * string) list = [(1, "a"); (1, "b"); (2, "a"); (2, "b")]
```

We might try the following definition:

```
let rec cart = function
| ([],     ys) -> []
| (x::xs, ys) -> cart ([x], ys) @ cart (xs, ys);;
```

Analyze this definition with respect to the Checklist for Programming with Recursion. Be sure to



discuss all three steps.

Step 1: correct ?

Step 2: correct

Step 3: incorrect ?

Every recrusive call is on a smaller input. X never gets smaller

b) Here is a correct solution to the transpose function from Homework 2:

```
let rec transpose = function
| [] -> failwith "cannot transpose a 0-by-n matrix"
| []::xs -> []        // base case: m-by-0
| xs -> List.map List.head xs :: transpose (List.map List.tail xs)
```

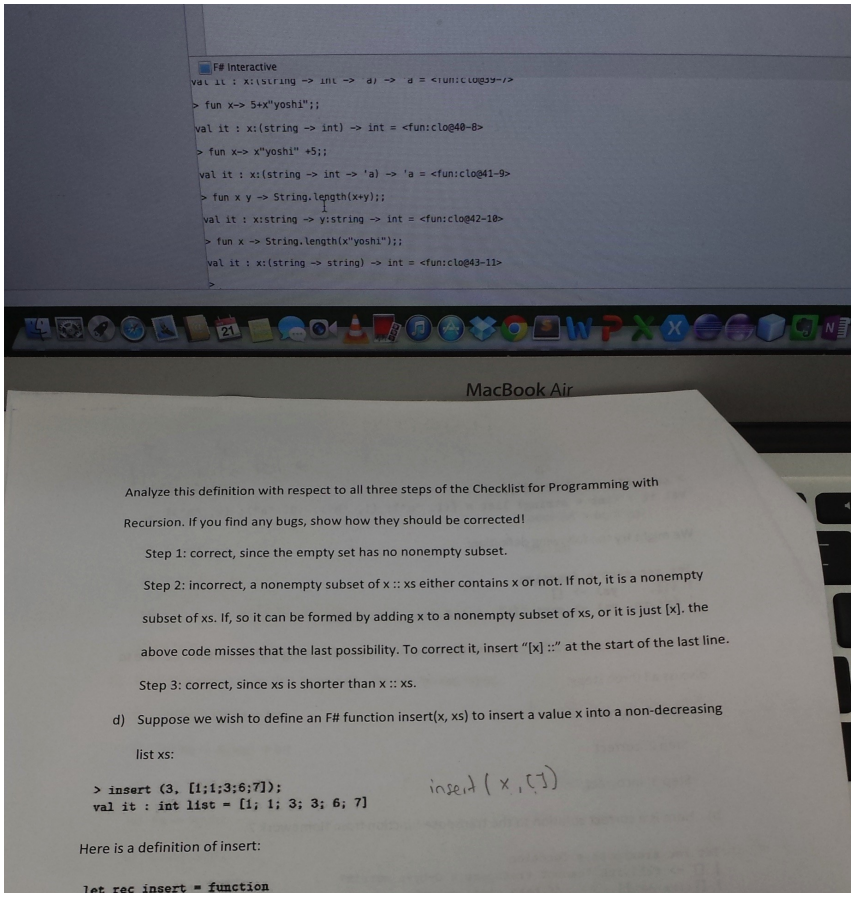
Suppose that we delete the first case ("| [] -> failwith ..."). Explain why the definition now fails Step 3 of the Checklist for Programming with Recursion.

Transpose [] is now handle by the last line. But (List.map List.tail []) is [], so the recursive call is on the original input!

c) Representing sets as lists, we might wish to implement an F# function nsub xs that returns (in whatever order) the set of all the nonempty subsets of xs. (Note that nsub is almost the same as powerset from Homework 2.) We might try the following:

```
let rec nsub = function
| []    -> []
| x::xs -> let p = nsub xs
           List.map (fun s -> x::s) p @ p
```

Calling the samething over and over on the empty list []::xs only checks for empty head not tail so it will run forever.



F# Interactive

```
> fun x-> 5+x"yoshi";;
val it : x:(string -> int) -> int = <fun:clo@40-8>
> fun x-> x"yoshi" +5;;
val it : x:(string -> int -> 'a) -> 'a = <fun:clo@41-9>
> fun x y -> String.length(x+y);;
val it : x:string -> y:string -> int = <fun:clo@42-10>
> fun x -> String.length(x"yoshi");;
val it : x:(string -> string) -> int = <fun:clo@43-11>
>
```

MacBook Air

Analyze this definition with respect to all three steps of the Checklist for Programming with Recursion. If you find any bugs, show how they should be corrected!

Step 1: correct, since the empty set has no nonempty subset.

Step 2: incorrect, a nonempty subset of x :: xs either contains x or not. If not, it is a nonempty subset of xs. If, so it can be formed by adding x to a nonempty subset of xs, or it is just [x]. the above code misses that the last possibility. To correct it, insert "[x] ::" at the start of the last line.

Step 3: correct, since xs is shorter than x :: xs.

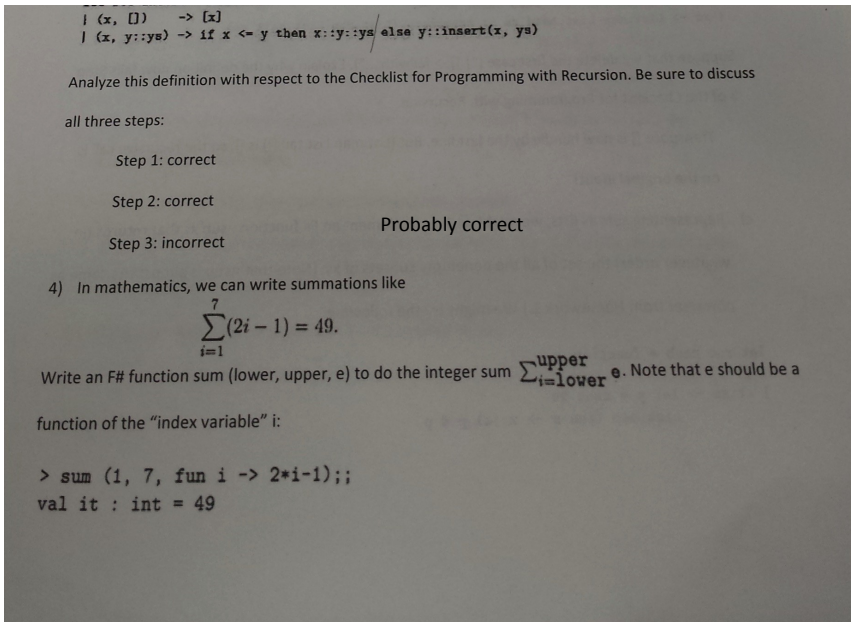
d) Suppose we wish to define an F# function insert(x, xs) to insert a value x into a non-decreasing list xs:

```
> insert (3, [1;1;3;6;7]);
val it : int list = [1; 1; 3; 3; 6; 7]
```

insert (x , [7])

Here is a definition of insert:

```
let rec insert = function
```



```
| (x, [])    -> [x]
| (x, y::ys) -> if x <= y then x::y::ys else y::insert(x, ys)
```

Analyze this definition with respect to the Checklist for Programming with Recursion. Be sure to discuss all three steps:

Step 1: correct

Step 2: correct

Probably correct

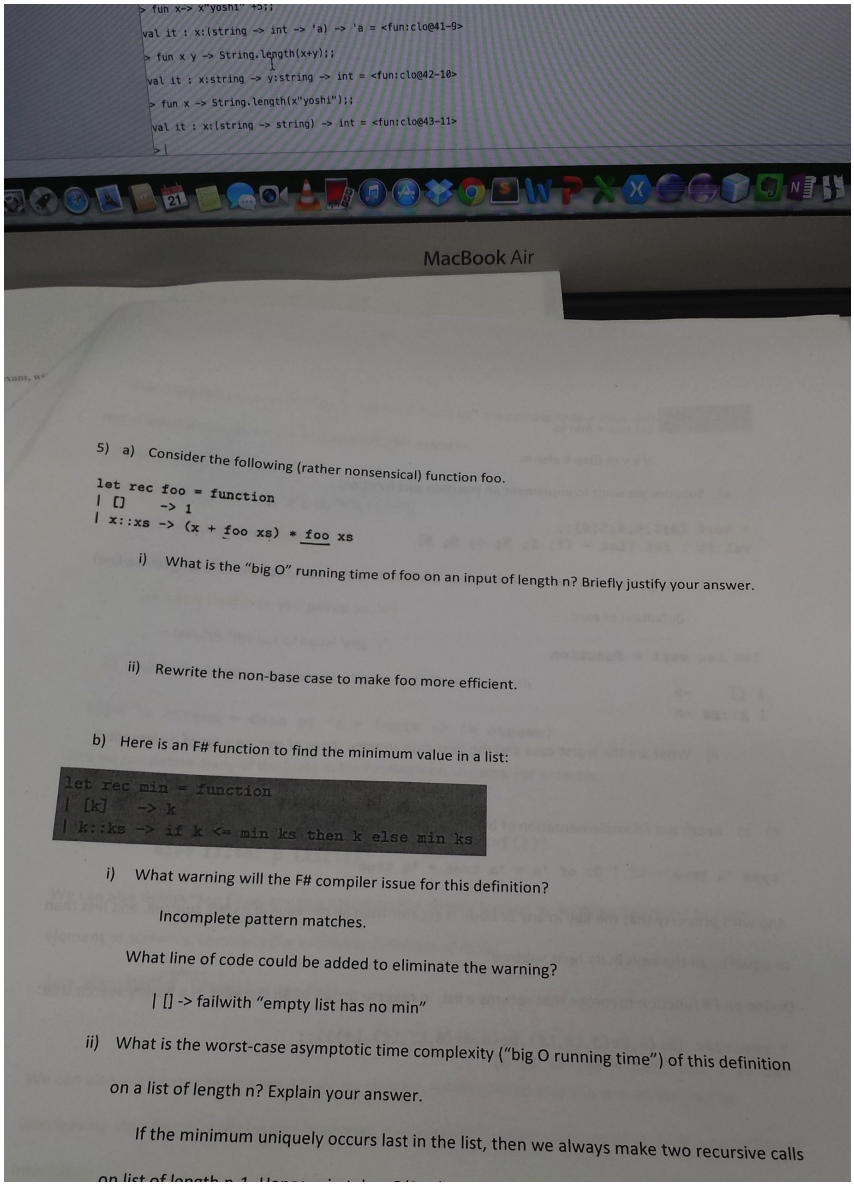
Step 3: incorrect

4) In mathematics, we can write summations like

$$\sum_{i=1}^{7}(2i-1) = 49.$$

Write an F# function sum (lower, upper, e) to do the integer sum $\sum_{i=lower}^{upper} e$. Note that e should be a function of the "index variable" i:

```
> sum (1, 7, fun i -> 2*i-1);;
val it : int = 49
```



```
> fun x-> x"yoshi" ;;
val it : x:(string -> int -> 'a) -> 'a = <fun:clo@41-9>
> fun x y -> String.length(x+y);;
val it : x:string -> y:string -> int = <fun:clo@42-10>
> fun x -> String.length(x"yoshi");;
val it : x:(string -> string) -> int = <fun:clo@43-11>
> |
```

MacBook Air

5) a) Consider the following (rather nonsensical) function foo.

```
let rec foo = function
| []    -> 1
| x::xs -> (x + foo xs) * foo xs
```

i) What is the "big O" running time of foo on an input of length n? Briefly justify your answer.

ii) Rewrite the non-base case to make foo more efficient.

b) Here is an F# function to find the minimum value in a list:

```
let rec min = function
| [k]    -> k
| k::ks -> if k <= min ks then k else min ks
```

i) What warning will the F# compiler issue for this definition?

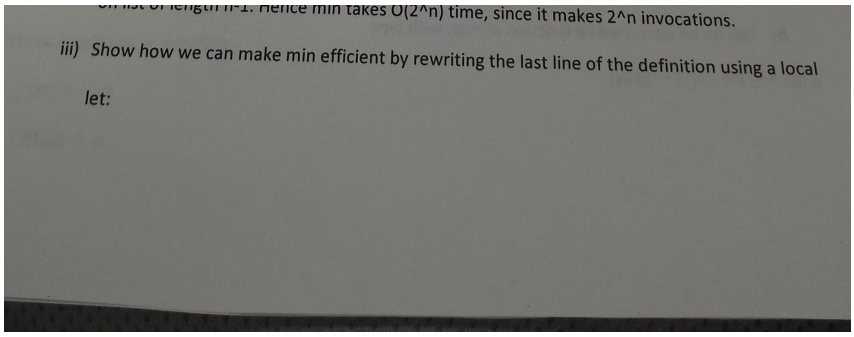
Incomplete pattern matches.

What line of code could be added to eliminate the warning?

| [] -> failwith "empty list has no min"

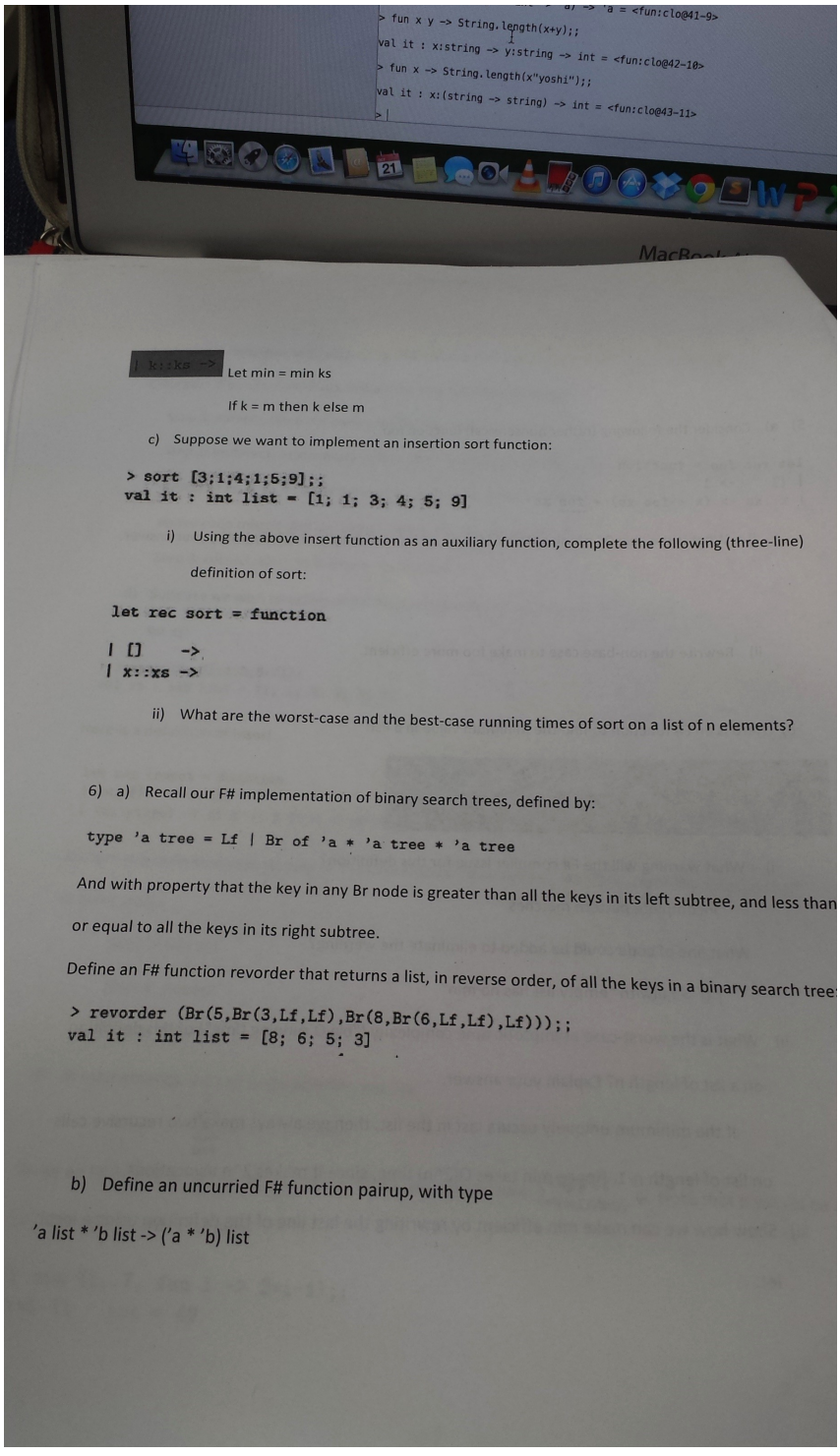
ii) What is the worst-case asymptotic time complexity ("big O running time") of this definition on a list of length n? Explain your answer.

If the minimum uniquely occurs last in the list, then we always make two recursive calls



on list of length n-1. Hence min takes O(2^n) time, since it makes 2^n invocations.

iii) Show how we can make min efficient by rewriting the last line of the definition using a local let:



| k::ks -> Let min = min ks

If k = m then k else m

c) Suppose we want to implement an insertion sort function:

```
> sort [3;1;4;1;5;9];;
val it : int list = [1; 1; 3; 4; 5; 9]
```

i) Using the above insert function as an auxiliary function, complete the following (three-line) definition of sort:

```
let rec sort = function

| []    ->
| x::xs ->
```

ii) What are the worst-case and the best-case running times of sort on a list of n elements?

6) a) Recall our F# implementation of binary search trees, defined by:

```
type 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```

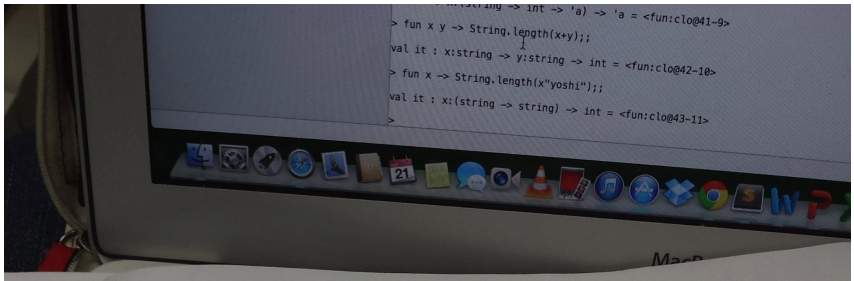
And with property that the key in any Br node is greater than all the keys in its left subtree, and less than or equal to all the keys in its right subtree.

Define an F# function revorder that returns a list, in reverse order, of all the keys in a binary search tree:

```
> revorder (Br(5,Br(3,Lf,Lf),Br(8,Br(6,Lf,Lf),Lf)));;
val it : int list = [8; 6; 5; 3]
```

b) Define an uncurried F# function pairup, with type

'a list * 'b list -> ('a * 'b) list



that takes two equal-length lists as input and "pairs up" the corresponding elements. (If the lists are not of equal length, then raise an exception.) For example,

```
> pairup ([1;2;3], ["a";"b";"c"]);;
val it : (int * string) list = [(1, "a"); (2, "b"); (3, "c")]
```

```
let rec pairup = function
|([], []) -> []
|(x :: xs, y :: ys) -> (x, y) :: pairup (xs, xy)
|_ -> failwith "list not of equal length"
```

c) Recall that in F# we can create a type of infinite stream with

```
type 'a stream = Cons of 'a * (unit -> 'a stream)
```

and we can define many of the build-in list functions on streams. For example,

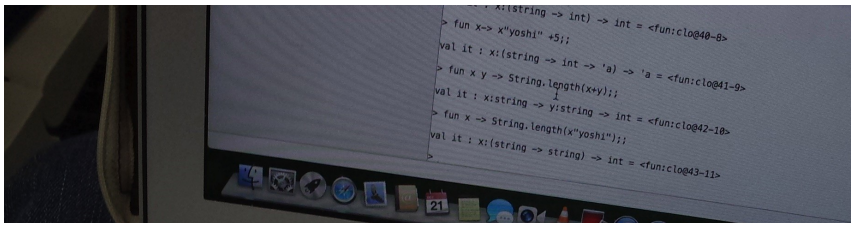
```
let rec filter p (Cons(x, xsf)) =
  if p x then Cons(x, fun () -> filter p (xsf()))
         else filter p (xsf())
```

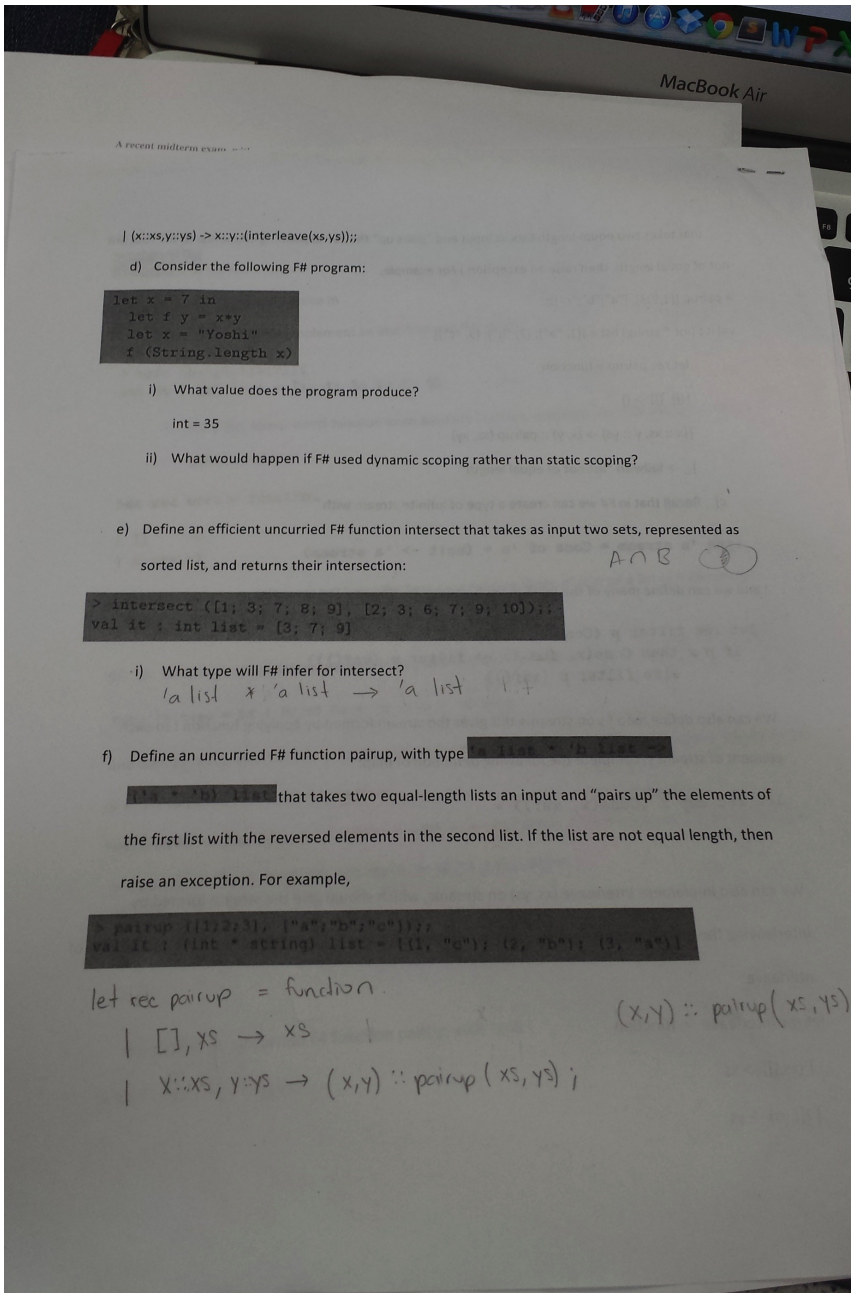
We can also define map f s on streams-this gives the stream formed by applying function f to each element of stream s. complete the following definition of map:

```
let rec map f (Cons(x, xsf)) =
```

We can also implement interleave (xs, ys) on streams, which should give the stream formed by interleaving the elements of streams xs and ys, starting with the first element of xs. Give a definition of interleave:

```
let rec interleave = function
| (xs,[]) -> xs
| ([],ys) -> ys
```





A recent midterm exam ...

| (x::xs,y::ys) -> x::y::(interleave(xs,ys));;

d) Consider the following F# program:

```
let x = 7 in
  let f y = x*y
  let x = "Yoshi"
  f (String.length x)
```

i) What value does the program produce?

int = 35

ii) What would happen if F# used dynamic scoping rather than static scoping?

e) Define an efficient uncurried F# function intersect that takes as input two sets, represented as sorted list, and returns their intersection:

A ∩ B

```
> intersect ([1; 3; 7; 8; 9], [2; 3; 6; 7; 9; 10]);;
val it : int list = [3; 7; 9]
```

i) What type will F# infer for intersect?

'a list * 'a list → 'a list

f) Define an uncurried F# function pairup, with type [redacted] that takes two equal-length lists an input and "pairs up" the elements of the first list with the reversed elements in the second list. If the list are not equal length, then raise an exception. For example,

```
> pairup ([1;2;3], ["a";"b";"c"]);;
val it : (int * string) list = [(1, "c"); (2, "b"); (3, "a")]
```

```
let rec pairup = function
  | [], xs → xs
  | x::xs, y::ys → (x,y) :: pairup (xs, ys);;
```

(x,y) :: pairup(xs,ys)

Which of the following F# expressions is *not* well typed?
Select one:
"4" + "5.6"
2 + 5 * 10
4 + 5.6
10l * 20l

Feedback

Your answer is correct.

The correct answer is: 4 + 5.6

Question 2

Correct

Mark 1.00 out of 1.00

Flag question

Question text

A curried function has a type of which form?

Select one:

t1 -> (t2 -> t3)

t1 -> t2 * t3

t1 * t2 -> t3

(t1 -> t2) -> t3

Feedback

Your answer is correct.

The correct answer is: t1 -> (t2 -> t3)

Question 3

Correct

Mark 1.00 out of 1.00

Flag question

Question text

If an F# function has type 'a -> 'b when 'a : comparison, which of the following is *not* a legal type for it?

Select one:

(float -> float) -> bool

string -> int

int -> int

int list -> bool list

Feedback

Your answer is correct.

The correct answer is: (float -> float) -> bool

Question 4

Correct

Mark 1.00 out of 1.00

Flag question

Question text

Which of the following statements about F# lists is *not* true?

Select one:

They can be heterogeneous.

They can be of any length.

Their built-in functions are polymorphic.

They are immutable.

Feedback

Your answer is correct.

The correct answer is: They can be heterogeneous.

Question 5

Correct

Mark 1.00 out of 1.00

Flag question

Question text

Which of the following F# expressions evaluates to [1; 2; 3]?

Select one:

1@2@3@[]

1::2::3::[]

[1; 2; 3]::[]

((1::2)::3)::[]

Feedback

Your answer is correct.

The correct answer is: 1::2::3::[]

How does F# interpret the expression List.map List.head foo @ baz?

Select one:

(List.map List.head) (foo @ baz)

List.map (List.head (foo @ baz))

((List.map List.head) foo) @ baz

(List.map (List.head foo)) @ baz

Feedback

Your answer is correct.

The correct answer is: ((List.map List.head) foo) @ baz

Question 2
Correct
Mark 1.00 out of 1.00
Flag question

Question text

How does F# interpret the type int * bool -> string list?
Select one:
(int * (bool -> string)) list
((int * bool) -> string) list
int * (bool -> (string list))
(int * bool) -> (string list)

Feedback

Your answer is correct.
The correct answer is: (int * bool) -> (string list)

Question 3
Correct
Mark 1.00 out of 1.00
Flag question

Question text

Let F# function foo be defined as follows:

```
let rec foo = function
| ([], [])    -> []
| (xs, y::ys) -> foo (xs@[y], ys)
```

If foo is supposed to append its two list parameters, which of the following is true?
Select one:
foo fails Step 1 of the *Checklist for Programming with Recursion*.
foo fails Step 2 of the *Checklist for Programming with Recursion*.
foo fails Step 3 of the *Checklist for Programming with Recursion*.
foo will get an *Incomplete pattern matches* warning.

Feedback

Your answer is correct.
The correct answer is: foo will get an *Incomplete pattern matches* warning.

Question 4
Correct
Mark 1.00 out of 1.00
Flag question

Question text

Which of the following is the type that F# infers for (fun f -> f 17)?
Select one:
('a -> 'b) -> 'b
(int -> int) -> int
(int -> 'a) -> 'a
('a -> 'a) -> 'a

Feedback

Your answer is correct.
The correct answer is: (int -> 'a) -> 'a

Question 5
Correct
Mark 1.00 out of 1.00
Flag question

Question text

Which of the following has type int -> int list?
Select one:
(@) [5]
[fun x -> x+1]
fun x -> 5::x
fun x -> x::[5]

Feedback

Your answer is correct.
The correct answer is: fun x -> x::[5]

What type does F# infer for the expression (3, [], true)?
Select one:
int * 'a list * bool
Type error.
int * int list * bool
int * 'a * bool

Feedback

Your answer is correct.

The correct answer is: int * 'a list * bool

Question 2

Correct

Mark 1.00 out of 1.00

Flag question

Question text

What type does F# infer for the expression fun x y -> x+y+"." ?

Select one:

string * string -> string

string -> string -> string

Type error.

int -> int -> string

Feedback

Your answer is correct.

The correct answer is: string -> string -> string

Question 3

Correct

Mark 1.00 out of 1.00

Flag question

Question text

What type does F# infer for the expression fun xs -> List.map (+) xs ?

Select one:

Type error.

int list -> int list

int list -> int -> int list

int list -> (int -> int) list

Feedback

Your answer is correct.

The correct answer is: int list -> (int -> int) list

Question 4

Correct

Mark 1.00 out of 1.00

Flag question

Question text

Which of the following does F# infer to have type string -> string -> string ?

Select one:

fun (x, y) -> x + y + "."

fun x y -> String.length x * String.length y

fun x -> fun y -> x + y + "."

(+)

Feedback

Your answer is correct.

The correct answer is: fun x -> fun y -> x + y + "."

Question 5

Correct

Mark 1.00 out of 1.00

Flag question

Question text

Which of the following does F# infer to have type (string -> string) -> string ?

Select one:

fun x y -> x + " " + y

fun f -> f "yoshi"

fun f -> f (f "yoshi")

fun f -> String.length (f "yoshi")

Feedback

Your answer is correct.

The correct answer is: fun f -> f (f "yoshi")