CSE 130, Fall 2005: Final Examination

Name: ____

ID: ____

Instructions, etc.

1. Write your answers in the space provided.
2. Wherever it says **explain**, write no more than **three lines** as explanation. The rest will be ignored.
3. The points for each problem are a rough indicator (when converted to minutes), of how long you should take for the problem.
4. Good luck!

1 (15)	
2 (15)	
3 (10)	
4 (20)	
5 (20)	
6 (30)	
7 (15)	
8 (10)	
Total (135)	

1. **[15 Points]** For each of the following Ocaml programs, if the code is well-typed, write down the value of `ans`, otherwise, if the code has a type problem, write "type error".

   (a)
   ```
   let ans =
     let x = 10 in
     let f y =
       let a = x + 1 in
       let b = y + a in
       a + b in
     f 100
   ```

   (b)
   ```
   let ans =
     let f n = 10 in
     let f n = if n > 0 then n + (f (n-1)) else 0 in
     f 5
   ```

   (c)
   ```
   let ans =
     let f g x =  g (g x) in
     let h0 = fun x -> x * x in
     let h1 = f h0 in
     let h2 = f h1 in
     h2 2
   ```

2. **[15 Points]** For each of the following Ocaml programs, write down the type of `ans`.

(a)
```
let ans =
  let f f = f + 1 in
  f
```

(b)
```
let ans f g x =
  if x > 0 then f x else g x
```

(c)
```
let ans l =
  match l with
    [] -> []
  | (hx,hy)::t -> (hx hy)::(ans t)
```

3. Consider the Ocaml module described below:

```
module Set : SETSIG =
  struct
     exception Duplicates

     type 'a set = 'a list

     let new x = [x]

     let rec mem s x =
       match s with
         [] -> false
       | h::t -> if x <> h then mem t x
                 else if mem t x then raise Duplicates
                 else true

     let add s x =
       if mem s x then s else (x::s)

     let union s1 s2 =
       match s1 with
         [] -> s2
       | h::t -> union t (add s2 h)

     let choose s =
       match s with
         [] -> None
       | h::t -> Some (h,t)
  end
```

and the *two* possible signatures:

(A)

```
module type SETSIG =
  sig
    type 'a set = 'a list
    val new    : 'a -> 'a set
    val mem    : 'a set -> 'a -> bool

    val choose : 'a set -> ('a * 'a set) option
    val union  : 'a set -> 'a set -> 'a set
  end
```

(B)

```
module type SETSIG =
  sig
    type 'a set
    val new    : 'a -> 'a set
    val mem    : 'a set -> 'a -> bool
    val add    : 'a set -> 'a -> 'a set
    val choose : 'a set -> ('a * 'a set) opti
    val union  : 'a set -> 'a set -> 'a set
  end
```

(a) [**5 Points**] For which *one* of the signatures (A) or (B), can a *client* can cause the exception `Duplicates` to get raised ? Write down a client expression that would cause this exception to get raised. For the other signature **explain** why the exception will never get raised.

Signature:

Client Expression:

Explanation:

(b) [**5 Points**] Recall the `filter` function described in class:

```
let rec filter f l =
  match l with
    [] -> []
  | h::t -> if f h then h::(filter f t) else filter f t
```

Consider the *client* function:

```
let intersection s1 s2 =
  filter (mem s2) s1
```

For *one* of the signatures (A) or (B), the the client function `intersection` compiles, i.e. is well typed. Which one ? What is the inferred type of `intersection` using this signature ?
Signature:

Inferred Type: `intersection : _________________ -> ________________ -> ________________`

(c) [**10 Points**] Write an equivalent version of `intersection` that would compile with *both* signatures.

4. Consider the following Ocaml datatype used to represent trees.

   `type 'a tree = Leaf of 'a | Node of 'a * 'a tree * 'a tree`

   (a) [**5 Points**] Write the value of type `int tree` that corresponds to the following pictorial representation of a tree.

```
          .
         / \
        /   \
       /     \
      .       .
     / \     / \
    /   \   /   \
    1   2   3   4
```

   (b) [**5 Points**] Consider the following function:

```
let rec tf f b t =
  match t with
    Leaf x -> f (b,x)
  | Node (t1,t2) -> tf f (tf f b t1) t2
```

   What is the *type*of the function `tf` ? Answer this by filling in the blanks:

   `_______________ -> ________________ -> _______________ -> _______________`

   (c) [**5 Points**] Fill in the blanks below to obtain an implementation of:
   `to_list : 'a tree -> 'a list`
   that returns the list of values occurring as leaves of the tree.

```
let to_list t =

  let f ____ = ________________________ in


  let b  =  ___________________________  in


  tf f b t
```

   (d) [**5 Points**] Fill in the blanks below to obtain an implementation of:
   `size : 'a tree -> int`
   that returns the list of values occurring as leaves of the tree.

```
let size t =

  let f ____ = ________________________ in


  let b  =  ___________________________  in


  tf f b t
```

(e) **[5 Points]** Write a *tail-recursive* version of `tf`. **Hint:** This is difficult. You may need a helper function.

5. For each of the following Python programs, write down the value of `ans`, or write **error** together with an explanation, if an error occurs. Write your answers on the blank space on the right.

   (a) [**5 Points**]

```
x = [1,2,3]
y = ["a","b","c"]

def f(x):
  x = y

f(x)
ans = x[0]
```

   (b) [**5 Points**]

```
def f(x):
  def g(y):
    return a(x+y)
  return g

a = f(10)
ans = a(0)
```

   (c) [**8 Points**]

```
a = [0]

def f(x):
  a = [10]
  def g(y):
    a[0] = a[0] + x + y
    return a[0]
  return g

foo = f(10)
foo(1000)
ans = (a[0],foo(1))
```

(d) **[7 Points]**

```
class A():
  def __init__(self):
    self.x = []

  def a(self):
    self.x += ["a"]
    self.d()

class B(A):
  def b(self):
    self.x += ["b"]

class C(A):
  def a(self):
    self.x += ["ca"]
  def c(self):
    self.x += ["c"]

class D(B,C):
  def d(self):
    self.x += ["d"]
    self.b()
    self.c()

o = D()
o.a()
ans = o.x
```

(e) **[5 Points]**

```
def foo(n):
  i = 1
  while (i <= n):
    i += i
    yield i

ans = 0
x = foo(10)
for i in x:
  ans += i
```

(f) **[5 Points]**

```
something with decorators ?
```

(a) [**5 Points**] Use `yield` to write a function `element_and_rest` which takes a list as input and returns an *iterator* over tuples which consist of an element of the list, and the list with that element removed. The elements of the list should be in the same order as in the original list. The function `element_and_rest` *should not* return a list. When you are done, the following:

```
>>> for t in element_and_rest([1,2,3,4,5]):
      print t
```

should result in:

```
(1, [2, 3, 4, 5])
(2, [1, 3, 4, 5])
(3, [1, 2, 4, 5])
(4, [1, 2, 3, 5])
(5, [1, 2, 3, 4])
```

The body of the function should be at most 3 lines long. Write it by filling in the blanks below:

```
def element_and_rest(l):

  __________________

  __________________

  __________________
```

(b) [**10 Points**] Write a function permutations which takes a list as input and returns an *iterator* over permutations of the given list. The function *should not* compute all permutations before returning. When you are done, the following:

```
for p in permutations([1,2,3]):
   print p
```

should result in:

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

The body of the function should be at most 5 lines long. Write it by filling in the blanks below:

```
def permutations(l):

  __________________________

  __________________________

  __________________________
```

----

6. Recall that we say `P <: Q` if `P` is a *structural subtype* of `Q`. Consider the following Java code.

```
interface A {
  Object a;
}

interface B {
  int a;
  int b;
}

interface C {
  A f(B x);
}

interface D {
 /* OUT */ _____ f ( /* IN */ ____ x);
}
```

(a) [**2 Points**] True or False: `A <: B` ?

(b) [**2 Points**] True or False: `B <: A` ?

(c) [**6 Points**] Write *four* possible ways of filling in the blanks in the definition of `D` (i.e. of completing the type of `f`) such that `D <: C`.

i. `/* IN */ _____________ , /* OUT */ _____________`
ii. `/* IN */ _____________ , /* OUT */ _____________`
iii. `/* IN */ _____________ , /* OUT */ _____________`
iv. `/* IN */ _____________ , /* OUT */ _____________`

7. [**5 Points**] In **less than three lines**, explain how decorators are different from aspects.

8. [**5 Points**] Consider the following C-like code.

```
int y = 1;

void f(int x){
  int y;
  y = x + 1;
  x = x + 10;
  g(x);
  printf("x = %d \n",x);
}

void g(int x){
  y = x + 1;
}

void main(){
  f(y);
  printf("y = %d \n",y)
}
```

What is the output of executing this code under

(a) *static scoping* ?

(b) *dynamic scoping* ?

9. Consider the following Prolog code:

```
actor(xmen,jackman).
actor(xmen,berry).
actor(scoop,jackman).
actor(scoop,johanssen).
actor(lost_in_translation,murray).
actor(lost_in_translation,johanssen).
actor(ghostbusters,murray).
actor(ghostbusters,akroyd).
actor(batmanreturns,bale).
actor(batmanreturns,caine).
actor(dirtyrottenscoundrels,martin).
actor(dirtyrottenscoundrels,caine).
actor(shopgirl,danes).
actor(shopgirl,martin).
```

(a) [**2 Points**] Write a predicate `costar(X,Y)` that is true when `X,Y` have acted in the same movie.

(b) [**3 Points**] Write a predicate `busy(X)` that is true when `X` has acted in more than one movie.

(c) [**5 Points**] Write a predicate `bacon(X,Y)` that is true when there is a sequence of actors $Z_1, Z_2, \ldots, Z_n$ such that for each i, the pair Z_i, Z_{i+1} have acted in the same movie, and `X` is Z_1 and `Y` is Z_n.

10. For this problem, you will write Prolog code to implement the magic algorithm whereby ML is able to *infer* the types of all expressions. First, we shall encode (nano) ML expressions as Prolog terms via the following grammar.

$$
\begin{array}{lll}
\textit{expr} & ::= & \\
 & & \texttt{const(i)} \\
 & & \mid \texttt{var(x)} \\
 & & \mid \texttt{plus}(\textit{expr}, \textit{expr}) \\
 & & \mid \texttt{leq}(\textit{expr}, \textit{expr}) \\
 & & \mid \texttt{ite}(\textit{expr}, \textit{expr}) \\
 & & \mid \texttt{letin}(\texttt{x}, \textit{expr}, \textit{expr}) \\
 & & \mid \texttt{fun}(\texttt{var(x)}, \textit{expr}) \\
 & & \mid \texttt{app}(\textit{expr}, \textit{expr})
\end{array}
$$

Similarly, we shall encode ML types as Prolog terms using the following grammar:

$$\textit{type} ::= \texttt{int} \mid \texttt{bool} \mid \texttt{arrow}(\textit{type}, \textit{type})$$

The table below shows several examples of Ocaml expressions, the Prolog term encoding that expression, and the Prolog term encoding the type of the expression.

ML Expression	Prolog Expression Term	Pr
`2`	`const(2)`	ir
`x`	`var(x)`	
`2 + 3`	`plus(const(2),const(3))`	ir
`2 <= 3`	`leq(const(2),const(3))`	bo
`fun x -> x <= 4`	`fun(var(x),leq(var(x),const(4)))`	ar
`fun x -> fun y -> if x then y else 0`	`fun(var(x),fun(var(y),ite(var(x),var(y),const(0))))`	ar
`let x = 10 in x`	`letin(var(x),const(10),var(x))`	ir
`fun x -> let y = x in y + y`	`fun(var(x),letin(var(y),var(x),plus(var(y),var(y))))`	ar

(a) **[5 Points]** Write a Prolog predicate `envtype(Env,X,T)`, such that `envtype([[x1,t1],[x2,t2],...,[xn,vn]]` is true if `X` equals the *first* term `xi` corresponding to variable `xi` and `T` equals the corresponding `ti` corresponding to the type of the variable `xi` in the type environment `ti`. When you are done, you should get the following behavior:

```
?- envtype([[x,int],[y,bool]],x,T).
   T = int
   Yes

?- envtype([[x,int],[x,bool]],x,T).
   T = int
   Yes

?- envtype([[x,int],[x,bool]],x,bool).
   No
```

(b) [**20 Points**] Write a Prolog predicate `typeof(Env,E,T)` that is true when the term `T` is the correct ML type of the ML expression corresponding the term `E` in the type environment corresponding to the list `Env`. Write your solution by filling in the grid below:

`typeof(Env,const(I),T) :-`	
`typeof(Env,var(X),T) :-`	
`typeof(Env,plus(E1,E2),T) :-`	
`typeof(Env,leq(E1,E2),T) :-`	
`typeof(Env,ite(E1,E2,E3),T) :-`	
`typeof(Env,letin(var(X),E1,E2),T) :-`	
`typeof(Env,fun(var(X),E),T) :-`	
`typeof(Env,app(E1,E2),T) :-`	

When you are done, you should get the following output:

```
?- typeof([[x,int],[y,bool]],Var(x),T).
   T = int
   Yes

?- typeof([],plus(const(2),const(3)),T).
   T = int
   Yes

?- typeof([],leq(const(2),const(3)),T).
   T = bool
   Yes

?- typeof([],fun(var(x),leq(var(x),const(4))),T).
   T = arrow(int,bool)
   Yes

?- typeof([],fun(var(x),fun(var(y),ite(var(x),var(y),const(0)))),T).
   T = arrow(bool,arrow(int,int))
   Yes

?- typeof([],letin(var(x),const(10),var(x)),T).
   T = int
   Yes

?- typeof([],fun(var(x),letin(var(y),var(x),plus(var(y),var(y)))),T).
   T = int
   Yes

?- typeof([],app(fun(var(x),plus(var(x),const(1))),const(19)),T).
   T = int
   Yes
```

(c) [[**5**] **Points**] Does your predicate infer polymorphic types ? In other words, using your implementation of `typeof` will the result of the following query be `Yes` or `No` ? Explain.

```
?- typeof([],letin(var(id),fun(var(x),var(x)),
          letin(var(y),app(var(id),leq(const(2),const(3))),
    app(var(id),const(1)))),T).
```

(d) [[**Extra Credit**] **Points**] Extend your solution so that the the above query succeeds. type inference is polymorphic. That is, it should successfully find an appropriate solution for `T` for the query above.