**Name:** ______________________

**ID :** ______________________

# CSE 130, Spring 2013: Final Examination

June 11th, 2013

---

- Do **not** start the exam until you are told to.
- This is a closed-book exam, with no computational devices allowed (such as calculators/cellphones/laptops). You many **only** refer to **two sides** of your own notes.
- Do **not** look at anyone else's exam. Do **not** talk to anyone but an exam proctor during the exam.
- Write your answers in the space provided.
- Wherever it gives a line limit for your answer, write no more than the specified number of lines. *The rest will be ignored.*
- Work out your solution in blank space or scratch paper, and only put your answer in the answer blank given.
- In all exercises, you are allowed to use the `len` function in Python and the `@` operator in OCaml.
- Good luck!

| | | |
|---|---|---|
| 1. | 25 Points | |
| 2. | 20 Points | |
| 3. | 10 Points | |
| 4. | 25 Points | |
| 5. | 20 Points | |
| 6. | 15 Points | |
| 7. | 15 Points | |
| TOTAL | 130 Points | |

**1.** [ **25 points** ] Let's warm up with some folding and stretching.

**a.** [ **10 points** ] Use `fold_left` to implement `count : ('a -> bool) -> 'a list -> int`, which takes a boolean tester `f` and a list and returns the number of elements in the list for which `f` returns true. For your reference, the type of `fold_left` is given below:

```
fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Fill in the implementation of `count` below using `fold_left`:

```
let count f l =

  -----------------------------------------------------------------------------

  -----------------------------------------------------------------------------

  -----------------------------------------------------------------------------

  -----------------------------------------------------------------------------
```

**b.** [ **5 points** ] For each expression below, write down what it evaluates to:

```
count ((=) 7) [1;2;3;4;5;6;7;8;9]                         ______

count ((!=) 8) [1;2;3;4;5;6;7;8;9]                        ______

count ((<) 3) [1;2;3;4;5;6;7;8;9]                         ______

count ((>=) 2) [1;2;3;4;5;6;7;8;9]                        ______

count (fun x -> x mod 2 = 0) [1;2;3;4;5;6;7;8;9]          ______
```

**c.** [ **10 points** ] Use `fold_left` to implement `stretch : 'a list -> 'a list`, which takes a list and duplicates each element in the list. The elements in the returned list should be in the same order as in the original list. Do not use `List.rev`. For example:

```
# stretch [1;2;3;4];;
- : int list = [1; 1; 2; 2; 3; 3; 4; 4]

# stretch ["a";"b";"c"];;
- : string list = ["a"; "a"; "b"; "b"; "c"; "c"]

# dup [0;0;1];;
- : int list = [0; 0; 0; 0; 1; 1]
```

Fill in the implementation of `stretch` below using `fold_left`:

```
let stretch l =


   --------------------------------------------------------------------------------


   --------------------------------------------------------------------------------


   --------------------------------------------------------------------------------
```

**2.** [ **20 points** ] Consider the following datatype for representing a tree. A tree is either the empty tree, or a node containing a data value and a list of children.

```
type 'a tree =
  | Empty
  | Node of 'a * 'a tree list;;
```

You will write a function `tree_zip : 'a tree -> 'b tree -> ('a * 'b) tree`, which takes two trees having the same structure and combines them into one tree. Each element of the returned tree is a pair containing the corresponding elements of the two input trees. For example:

```
# let t1 = Node(1,[Node(2, []); Node(3, [])]);;
# let t2 = Node(4,[Node(5, []); Node(6, [])]);;
# tree_zip t1 t2;;
- : (int * int) tree = Node ((1, 4), [Node ((2, 5), []); Node ((3, 6), [])])
```

If the two input trees do not have a similar structure (in that one tree has an element for which there is no corresponding element in the other tree), then `tree_zip` should raise the exception `Mismatch`.

To implement `tree_zip`, your must make use of the following two functions:

- `tree_zip` needs to use `map : ('a -> 'b) -> 'a list -> 'b list`
- `tree_zip` needs to use the following `zip` function on lists:

```
  let rec zip l1 l2 =
     match (l1,l2) with
     | ([],[]) -> []
     | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
     | _ -> raise Mismatch;;
```

Fill in the implementation of `tree_zip` below using `map` and `zip`:

```
let rec tree_zip t1 t2 =

   match (t1,t2) with


      ------------------------------------------------------------------------------

      ------------------------------------------------------------------------------

      ------------------------------------------------------------------------------

      ------------------------------------------------------------------------------

      ------------------------------------------------------------------------------

      ------------------------------------------------------------------------------

      ------------------------------------------------------------------------------
```

**3.** [ **10 points** ] In this question, you will implement a `zip(L1,L2,L3)` predicate in Prolog, which holds if the $i^{th}$ element of `L3` is a list containing the $i^{th}$ element of `L1` and the $i^{th}$ element of `L2`. For example:

```
1 ?- zip([1,2,3],[4,5,6],R).
R = [[1, 4], [2, 5], [3, 6]].

2 ?- zip(X,Y,[[1, 3], [2, 4]]).
X = [1, 2],
Y = [3, 4].

3 ?- zip(X,[3,4],R).
X = [_G917, _G929],
R = [[_G917, 3], [_G929, 4]].

4 ?- zip(X,Y,[]).
X = Y, Y = [].
```

Fill in the implementation of `zip` predicate in Prolog below:

```
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------

--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
```

**4.** [ **25 points** ] In this question, you will implement quick-sort in Prolog.

**a.** [ **15 points** ] We will start by implementing the partitioning step, which separates a list into two lists based on whether the elements are bigger or smaller than a given pivot. In particular, you will implement a predicate `part(L,P,R1,R2)` which holds if `R1` contains all the elements of `L` which are smaller or equal to `P`, and `R2` contains all the elements of `L` which are greater than `P`. For example:

```
1 ?- part([1,2,3,4,5,6,7,8,9], 5, R1, R2).
R1 = [1, 2, 3, 4, 5],
R2 = [6, 7, 8, 9] .

2 ?- part([1,3,5], 10, R1, R2).
R1 = [1, 3, 5],
R2 = [] .

3 ?- part(X, 4, [1,2,3], [5,6]).
X = [1, 2, 3, 5, 6] .
```

Fill in the implementation of the `part` predicate below:

```
part( ________, ________, ________, ________).

part( ________, ________, ________, ________) :- ____________________________________________

part( ________, ________, ________, ________) :- ____________________________________________
```

**b.** [ **10 points** ] You will now implement a predicate `qsort(L,R)` which holds if `R` is the sorted version of `L` using quick-sort. Recall that quick-sort works as follows:

1. pick a pivot, and partition the remainder of the list into two lists based on whether elements are smaller or bigger than the pivot
2. recursively sort the two lists
3. combine the two lists and the pivot to create a sorted result

Use the first element of the list as a pivot. You can of course use `part` which you defined previously. Furthermore, you may use the `append(L1,L2,L3)` predicate, which holds if the list `L3` contains all the elements of `L1` (in the same order as `L1`) followed by all the elements of `L2` (in the same order as `L2`). Fill in the implementation of `qsort` predicate in Prolog below:

**5.** [ **20 points** ] In this question you will implement a Python function `rev` which takes a list and returns a new list with all elements in reverse order. We will implement it in two ways. You are not allowed to use the built-in function `reversed()`.

**a.** [ **10 points** ] First, fill in the implementation of `rev` below:

```
def rev(l):


   return [ ____________________________________________________________________________ ]
```

**b.** [ **10 points** ] Now you will implement `rev` using `reduce`, which is the Python equivalent of OCaml's `fold_left`. For example, here is a use of `reduce`:

```
>>> def sum(l):
...    def fold_fn(acc,elm): return acc + elm
...    return reduce(fold_fn, l, 0)

>>> sum([1,2,3])
6
```

Fill in the implementation of `rev` below using `reduce`:

```
 ____________________________________________________________________________________

 ____________________________________________________________________________________

 ____________________________________________________________________________________

 ____________________________________________________________________________________
```

**6.** [ **15 points** ] In this question you will write a decorator `print_some(l)` which takes a list of integers `l` indicating how to selectively print the arguments and return value of the original function. For this problem, don't worry about keyword arguments, and assume that `l` contains no duplicates.

The function after decoration should call the original function, but in addition, for each element $i$ in `l`, the function after decoration should also do the following:

1. If $i = -1$, print the return value of the original function after it's called.
2. Otherwise, print the $i^{th}$ argument (if it exists) before the original function is called.

For example:

```
>>> @print_some([-1,1,0])
... def plus(x,y):
...     print "-- plus called --"
...     return x+y

>>> plus(1,2)
Arg 1: 2
Arg 0: 1
-- plus called --
Return: 3
3

>>> @print_some([-2,100])
... def plus(x,y):
...     print "-- plus called --"
...     return x + y

>>> plus(1,2)
-- plus called --
3

>>> @print_some([-1,0])
... def fac(n):
...    print "-- fac called --"
...    if n is 0: return 1
...    else:      return n * fac(n-1)

>>> fac(2)
Arg 0: 2
-- fac called --
Arg 0: 1
-- fac called --
Arg 0: 0
-- fac called --
Return: 1
Return: 1
Return: 2
2
```

Write the `print_some` decorator below (use `str(x)` to return the string representation of `x`):

**7.** [ **15 points** ] This is a challenging problem, but it's also an example of something used in practice, namely *unification*. In particular, you will implement Prolog unification in Python. Recall that unification tries to make terms like `foo(X,bar(6))` and `foo(5,Y)` syntactically equal by finding appropriate values for variables like `X` and `Y` (in this case, `X` should be `5`, and `Y` should be `bar(6)`).

To begin, we will represent Prolog terms like `foo(X,bar(6))`, and `foo(5,Y)` using the following `Tree` class:

```
class Tree:
   def __init__(self, name, children):
      self.name = name
      self.children = children

   # Returns True if the Tree represents a prolog variable (e.g. X), and False otherwise
   def is_var(self):    ...

   # Returns the string representation of the Tree as a Prolog term.
   def __repr__(self):  ...

# Constructs a Tree representing a Prolog variable with name n
def var(n): return Tree(n, [])

# Constructs a Tree representing a non-variable term with name n and children c
def node(n, c): return Tree(n, c)
```

In particular, each Prolog term such as `foo(5,bar(6))` or `foo(X,bar(Y))` is represented with a tree that encodes the syntactic structure of the term. Each element of the tree has a name and a list of children. If a tree `t` represents an integer constant, then `t.name` is the integer and `t.children` is `[]`. If `t` represents a variable, then `t.name` is the name of the variable (e.g. "X") and `t.children` is again `[]`. Otherwise `t` is a term like `foo(X,5)`, in which case `t.name` is the name of the term as a string (e.g. `"foo"`) and `t.children` is a list of `Tree`s representing the children of `t` (e.g. `X` and `5` are the children of `foo(X,5)`). For example:

```
>>> node("foo", [node(5, [])])
foo(5)
>>> node("baz", [node(10, []),var("X")])
baz(10, X)
>>> var("X").is_var()
True
>>> node("foo", [node(5, [])]).is_var()
False
>>> node(5, []).is_var()
False
```

**Important:** Throughout this problem, only use the `node` and `var` functions to construct instances of `Tree`.

In this problem, we will also be using the concept of a *substitution*, which is simply a map from variable names (represented as strings) to terms (represented using `Tree`). We will implement substitutions using Python dictionaries. For example, the following builds a simple substitution `s`:

```
>>> s = {}
>>> s["X"] = node("foo", [node(5, []), node(6, [])])
>>> s["Y"] = node("baz", [node(10, [])])
>>> s
{'Y': baz(10), 'X': foo(5, 6)}
>>> s["X"]
foo(5, 6)
>>> s["Y"]
baz(10)
```

**a.** [ **8 points** ] You will first write a function `apply_to_tree(s,t)`, which takes a substitution `s` and a tree `t`, and returns a new tree identical to `t`, but in which each occurrence of a variable is replaced with the tree that `s` maps that variable to. Each time a variable is replaced with a tree, the substitution `s` must also be recursively applied to the replaced tree. If a variable in `t` does not have an entry in `s`, then that variable simply remains untouched in the resulting tree. For example:

```
>>> s1 = {}
>>> s1["X"] = node("foo", [node(5, [])])
>>> s2 = s1.copy()
>>> s2["Y"] = node("baz", [node(10, []), var("X")])
>>> t1 = node("bat", [var("X")])
>>> t2 = node("bat", [var("X"), var("Y")])

>>> s1
{'X': foo(5)}
>>> s2
{'Y': baz(10, X), 'X': foo(5)}
>>> t1
bat(X)
>>> t2
bat(X, Y)

>>> apply_to_tree(s1, t1)
bat(foo(5))
>>> apply_to_tree(s2, t2)
bat(foo(5), baz(10, foo(5)))
>>> apply_to_tree(s1, t2)
bat(foo(5), Y)
```

Fill in the implementation of `apply_to_tree` below (**Hint 1:** you will need to use list comprehension somewhere below. **Hint 2:** recall that `k in d` tests whether `k` belongs to the keys of dictionary `d`):

```
def apply_to_tree(s,t):


   if not t.is_var():


      return ____________________________________________________________________________


   elif _____________________ :


      return ____________________________________________________________________________


   else:


      return ____________________________________________________________________________
```

**b.** [ **7 points** ] Now we are ready to write unification. In particular, your goal is to write a function `unify(t1,t2)`, which takes two Prolog terms `t1` and `t2` (represented as `Trees`), and returns a substitution `s` such that `apply_to_tree(s,t1)` is equal to `apply_to_tree(s,t2)`, or `False` if such a substitution does not exist.
For example:

```
>>> t1 = node("foo", [var("X"),node(5,[])])
>>> t2 = node("foo", [node(10,[]),var("Y")])
>>> t3 = node("foo", [node(10,[]),var("X")])
>>> t4 = node("bar", [var("X"),var("Y"),var("X")])
>>> t5 = node("bar", [node("foo", [var("Y")]),node("3",[]),node("foo", [node("3",[])])])
>>> t6 = node("bar", [node("foo", [var("Y")]),node("3",[]),node("foo", [node("4",[])])])

>>> t1
foo(X, 5)
>>> t2
foo(10, Y)
>>> t3
foo(10, X)
>>> t4
bar(X, Y, X)
>>> t5
bar(foo(Y), 3, foo(3))
>>> t6
bar(foo(Y), 3, foo(4))

>>> unify(t1,t2)
{'Y': 5, 'X': 10}
>>> unify(t1,t3)
False
>>> unify(t4,t5)
{'Y': 3, 'X': foo(Y)}
>>> unify(t4,t6)
False
```

The main strategy for implementing unification is that `unify` will recursively traverse the two trees in parallel, building along the way the substitution that would make corresponding nodes from the two trees equal. If a variable is encountered, the substitution is updated to map that variable to the corresponding node in the other tree. If there is situation where two nodes cannot be made equal (because the nodes don't have the same name, or don't have the same number of children), then `False` is returned.

To keep track of the substitution being built while traversing the two trees, we will add a third optional parameter to `unify`, namely a substitution `s` which by default will be `{}`. This substitution will represent the substitution we have accumulated so far in the recursive traversal. All recursive calls to `unify` will pass in a value for this additional parameter. The only time we call `unify` without this additional parameter is when `unify` is called at the top-level.

Fill in the implementation of `unify` below. Note that the structure of this code is that the `result` substitution is being updated throughout the code, and returned at the end. **You should not have to write any return statements of your own.**

```
def unify(a,b,s={}):

   a = apply_to_tree(s, a)

   b = apply_to_tree(s, b)

   result = s.copy()

   if a.is_var() and b.is_var():  _______________________________________________________


   elif a.is_var() and not b.is_var():


      if a.name in result: ______________________________________________________________


      else:  ____________________________________________________________________________


   elif not a.is_var() and b.is_var():

      return unify(b,a,s)

   elif not a.is_var() and not b.is_var():


      if ___________________________________________________________: return False


      if ___________________________________________________________: return False


      for __________________________________________________:


         result = unify( ______________________, ______________________, result)


   return result
```