CS 421 Spring 2010 Final

Monday, May 10, 2010

Name	
NetID	

- You have **180 minutes** to complete this exam
- This is a **closed book** exam.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, seek clarification from one of the TAs. You must use a whisper, or write your question out.
- Including this cover sheet, there are 18 pages to the exam. Please verify that you have all 18 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	20	
2	20	
3	20	
4	15	
5	15	
6	20	
7	15	
8	10 XC	
9	20	
10	15	
11	20	
12	10	
13	10 + 10 XC	
Total	**200 + 20 XC**	

1. (20 pts) Write the following OCaml functions, *and* give their types:

   (a) (5 pts) `lteq n lis` returns the number of numbers in `lis` that are less than or equal to `n`.

   ```
   lteq 10 [3; 4; 12; 15]
   2
   ```

   Type: 'a -> 'a list -¿ int
   Code:

   ```
   let rec lteq n lis = match lis with
       [] -> 0
     | x::xs -> let m = lteq n xs in if x<=n then m+1 else m;;
   ```

   (b) (5 pts) Using `fold_right` instead of explicit recursion, write an OCaml function `remove_empty` that takes a list of strings and removes any empty strings from it, leaving the remaining strings in their original order. Recall that the type of `fold_right` is `('a -$>$ 'b -$>$ 'b) -$>$ 'a list -$>$ 'b -$>$ 'b`.

   ```
   remove_empty ["Return"; "your"; ""; "bottles"]
   ["Return"; "your"; "bottles"]
   ```

   Type: string list -¿ string list
   Code:

   ```
   let remove_empty lis =
      fold_right (fun x y -> if x="" then y else x::y) lis [];
   ```

(c) (5 pts) `mapfuns flst x` applies each function in the list of functions `flst` to the value `x`.

```
mapfuns [fun x -> x*x; fun x -> x-1; fun x -> x/2] 8
[64; 7; 4]
```

Type: ('a -> 'b) list -> 'a -> 'b list
Code:

```
let rec mapfuns flis x = match flis with
    [] -> []
  | f::fs -> f x :: mapfuns fs x
```

(d) (5 pts) `filter` takes a boolean-valued function f and a list L and returns a pair of lists, one with the elements of L for which f was true, and one with the elements for which it was false.

```
filter (fun x -> x > 0) [4; 5; -3; 2; -7; 0]
([4; 5; 2]; [-3; -7; 0])
```

Type: ('a -> bool) -> 'a list -> 'a list * 'a list
Code:

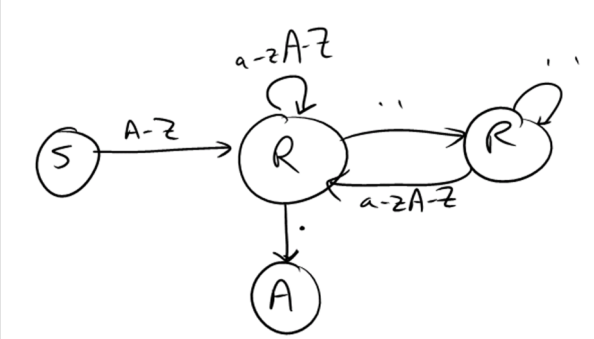
```
let rec filter f lis = match lis with
    [] -> ([], [])
  | x::xs -> match filter f xs with
             (fs, notfs) -> if f x then (x :: fs, notfs)
                                   else (fs, x :: notfs)
```

2. **Lexing** (20 pts) A *sentence* is a sequence of letters and spaces that begins with a capital letter and ends with a period (words my be separated by any number of spaces.) In addition, the last character before the period must not be a space.

   (a) (10 pts) Write a regular expression that describes the set of sentences.

   ```
   ['A'-'Z']('.' | ['a'-'z' 'A'-'Z' ' ']* ['a'-'z' 'A'-'Z'] '.')
   ```

   (b) (10 pts) Write a DFA that recognizes sentences. Label the start state with S, accepting states with A, and other states with R.



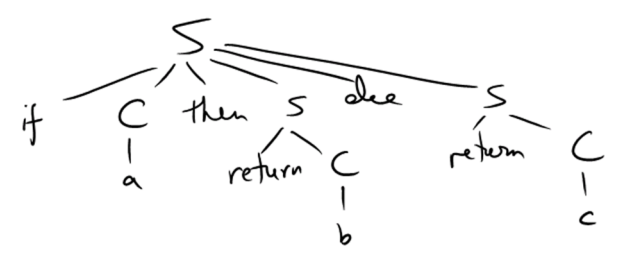
3. **Parsing** (20 pts) Consider the following type of tokens:

```
type token = ID of string | OR | SEMICOLON | IF | THEN | ELSE | RETURN
```

Now consider the following grammar, with non-terminals S and C:

```
S -> IF C THEN S ELSE S | RETURN C
C -> ID | ID OR C | S SEMICOLON C
```

(a) (5 pts) Give a parse tree for the string "if a then return b else return c".



(b) (5 pts) Give one reason that this grammar is not LL(1).

FIRST(ID) and FIRST(ID OR C) intersect. (Needs left-factoring.)

(c) (10 pts) Transform the grammar to an LL(1) grammar for the same language. (Hint: this requires introducing one new non-terminal.)

```
S -> IF C THEN S ELSE S | RETURN C
C -> ID D | S SEMICOLON C
D -> epsilon | OR C
```

4. **Parsing** (15 pts) Consider the following type of tokens:

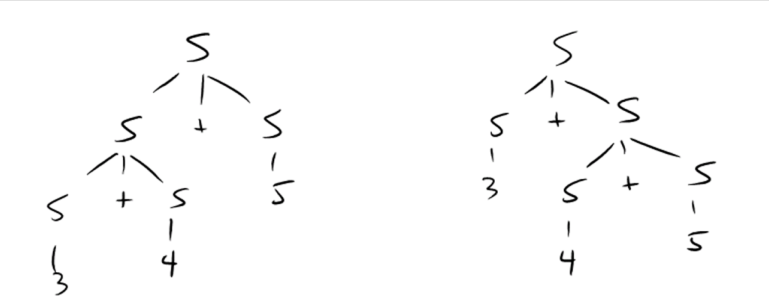
```
type token = INT of int | PLUS | TIMES
```

Now consider the following ambiguous grammar over tokens:

```
S -> S PLUS S | S TIMES S | INT
```

(a) (5 pts) Give a sentence that has two parse trees, and show the two parse trees.

3+4+5



(b) (10 pts) Give a grammar that recognizes the same language but is unambiguous, and makes PLUS and TIMES left-associative and gives TIMES higher precedence.

```
S -> S PLUS T | T
T -> T TIMES P | P
P -> INT
```

5. **Abstract syntax** (15 pts) Referring to the previous problem, suppose we define an abstract syntax for the language as follows:

```
type exp = Integer of int | Plus of exp * exp | Times of exp * exp
```

   (a) (5 pts) Write the function `compute : exp -> int` that computes the value of a given expression.

```
let rec compute e = match e with
    Integer i -> i
  | Plus(e1,e2) -> compute e1 + compute e2
  | Times(e1,e2) -> compute e1 * compute e2
```

   (b) (10 pts) Write the function `expand : exp -> exp` that takes an expression and replaces every instance of "e1 * (e2 + e3)" with "e1 * e2 + e1 * e3", and replaces every instance of "(e1 + e2) * e3" with "e1 * e3 + e2 * e3". Note that applying the transformation can produce new expressions that are subject to futher transformation; e.g. (1+2)*(3+4) can be transformed to (1+2)*3 + (1+2)*4, after which more transformations can be applied. Your function should result in an expression in which no further transformations are applicable.

```
let rec expand exp = match exp with
    Integer i -> Integer i
  | Plus(e1,e2) -> Plus(expand(e1), expand(e2))
  | Times(e1,e2) ->
      let e1' = expand(e1)
      and e2' = expand(e2)
      in match e1' with
           Plus(e3,e4) -> Plus(expand(Times(e3, e2')),
                               expand(Times(e4, e2')))
         | _ -> (match e2' with Plus(e3,e4) -> Plus(expand(Times(e1', e3)),
                                                    expand(Times(e1', e4)))
                  | _ -> Times(e1', e2'));;
```

6. **Parsing** (20 pts) Consider the following grammar:

```
S ::= if C then S else S | return C
C ::= id | C or C
```

Suppose that the "or" operator is declared to be left-associative with the %left directive. Show the first thirteen steps of the shift-reduce parse of the string "if x or y or z then return x else return y". We have filled in the first and last lines for you:

Action	**Stack**	**Input**
Shift		if x or y or z then ...
Shift	if	x or y or z then ...
Red C → id	if x	or y or z then ...
Shift	if C	or y or z then ...
Shift	if C or	y or z then ...
Red C → id	if C or y	or z then ...
Red C → C or C	if C or C	or z then ...
Shift	if C	or z then ...
Shift	if C or	z then ...
Red C → id	if C or z	then ...
Red C → C or C	if C or C	then ...
Shift	if C	then ...
Shift	if C then	return x else return y

7. **Code generation** (15 pts) In class, we gave the following translation schemes for translating source programs into an intermediate representation (IR). All but the first take an AST (expression or statement) to a sequence of IR instructions.

   [e] : translate expression e to IR; returns pair (IR instruction list, location of value)

   [S] : translate statement S to IR

   $[e]_x$: translate expression e to code that stores value of e in variable x

   $[S]_L$: translate statement S in context of a loop or switch statement, where L is the target of a break statement

   $[e]_{Lt,Lf}$: translate expression e to code that branches to Lt if e is true, or Lf otherwise (the short-circuit evaluation scheme)

   The instructions in our intermediate representation were: x = n; x = y; x = y + z (for any operation +); JUMP L; CJUMP x, L1, L2; and x = LOADIND y.

   Give the translation for the following constructs:

   (a) (5 pts) [`if` e `then` S_1 `else` S_2] Use the short-circuit translation scheme, $[e]_{Lt,Lf}$ for the condition.

   let L1, L2, L3 = new labels in
   $[e]_{L1,L2}$
   L1: $[S_1]$
   JUMP L3
   L2: $[S_2]$
   L3:

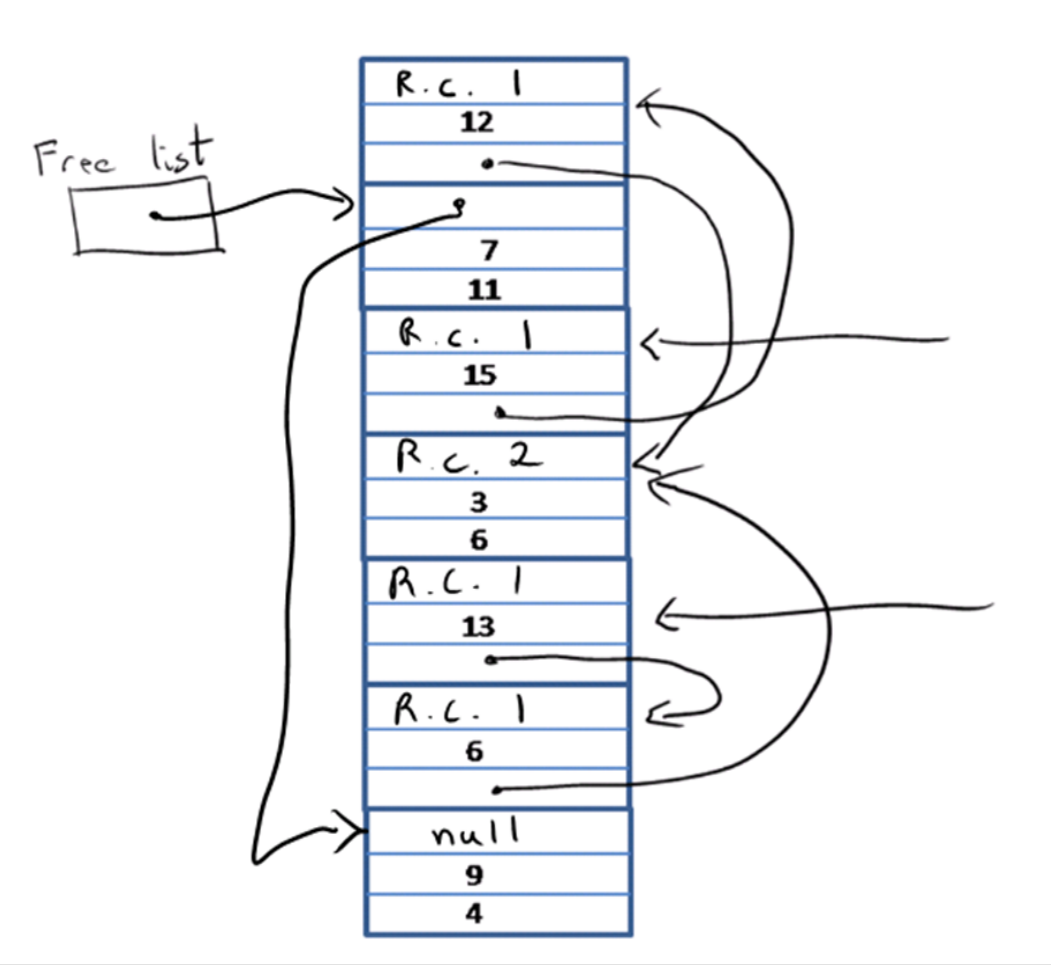
   (b) (5 pts) $[e_1$ && $e_2]_x$, for boolean-valued expressions e_1 and e_2. Recall that schemes of the form $[e]_x$ put the value of e in variable x. In this case, you should avoid evaluating e_2 if e_1 turns out to be false.

   let L1, L2, L3, L4 = new labels in
   $[e_1]_{L1,L2}$
   L1: $[e_2]_{L3,L2}$
   L2: x = false
   JUMP L4
   L3: x = true
   L4:

(c) (5 pts) $[e_1 \mid\mid e_2 \mid\mid e_3]_{Lt,Lf}$.

let L1, L2 = new labels in
$[e_1]_{Lt,L1}$
L1: $[e_2]_{Lt,L2}$
L2: $[e_2]_{Lt,Lf}$

8. **Reference-counting** (10 pts XC) In the following picture of the heap, each large rectangle represents an object containing three words (for simplicity, we're assuming all object have the same size), and the lines within it are the word boundaries. In this heap structure, the first word of each object either gives the reference count, or, if the object is on the free list, it points to the next object on the free list (if any); the box labeled "Free list" points to the head of the free list. Lastly, the Pointers coming from the right side are from the stack. Complete this picture in the following way: Construct the free list to contain all the non-reachable objects (their order does not matter). Fill in the reference counts for each reachable object. (To simplify grading, draw all your pointers on the left side.)



9. **Representing data with higher-order functions** (20 pts) A sparse matrix is a matrix in which most elements are zero. Such a matrix can be represented as a function that takes a row and column number, and returns the value at that location.

```
type sparse_matrix = int * int -> float;;
```

(a) (5 pts) Define `emptymat`, the sparse matrix in which all elements are zeroes.

```
let emptymat (r,c) = 0.0
```

(b) (5 pts) Define the subscripting operation `get m (r,c)` that returns `m[r,c]`.

```
let get m (r,c) = m (r,c)
```

(c) (5 pts) Define the function `set m (r, c) v`, which returns a new matrix `m'` just like `m` except that `m'[r,c] = v`.

```
let set m (r,c) v = fun (r',c') -> if r'=r & c'=c then v else m(r',c')
```

(d) (5 pts) Define the function `get_interval m r c1 c2`, which returns the list of the elements of `m` in row `r` from columns `c1` to `c2` inclusive.

```
let rec get_interval m r c1 c2
   = if c1>c2 then [] else m (r,c1) :: get_interval m r (c1+1) c2
```

10. **Functional programming in object-oriented languages** (15 pts) Complete the definitions of `emptymat`, `get`, and `set` in the following Java implementation of sparse matrices:

```
interface IntSparseMatrix{
   float apply (int r, int c);
}

class MapOps {
   static IntSparseMatrix emptymat =
        new IntSparseMatrix () {
             public float apply (int r, int c) { return 0f; }
        };

   static float get (IntSparseMatrix m, int r, int c){
        return m.apply(r,c);
   }

   static IntSparseMatrix set (final IntSparseMatrix m,
                               final int r, final int c, final float v) {
        return new IntSparseMatrix () {
           public float apply (int r1, int c1) {
              if (r1==r && c1==c)
                 return v;
              else
                 return m.apply(r1, c1);
           }
        };
   }
}
```

11. **Operational semantics** (20 pts)

(a) (5 pts) The expression `let x, y = e1, e2 in e3` is an abbreviation for `let x = e1 in let y = e2 in e3`. Give an operational semantics rule in OS_{clo} for this new type of expression. You must give it directly in terms of the subcomputations for e1, e2, and e3, and not simply by translation.

$$\frac{\eta, e_1 \Downarrow v_1 \qquad \eta[x \to v_1], e_2 \Downarrow v_2 \qquad \eta[x \to v_1][y \to v_2], e_3 \Downarrow v}{\eta, \text{let } x, y = e_1, e_2 \text{ in } e_3 \Downarrow v}$$

(b) (5 pts) Give a proof in OS_{clo} of the following judgment:

$$\emptyset, (\texttt{fun x} \to \texttt{fun y} \to \texttt{x} + \texttt{y})\ 3\ 4 \Downarrow 7$$

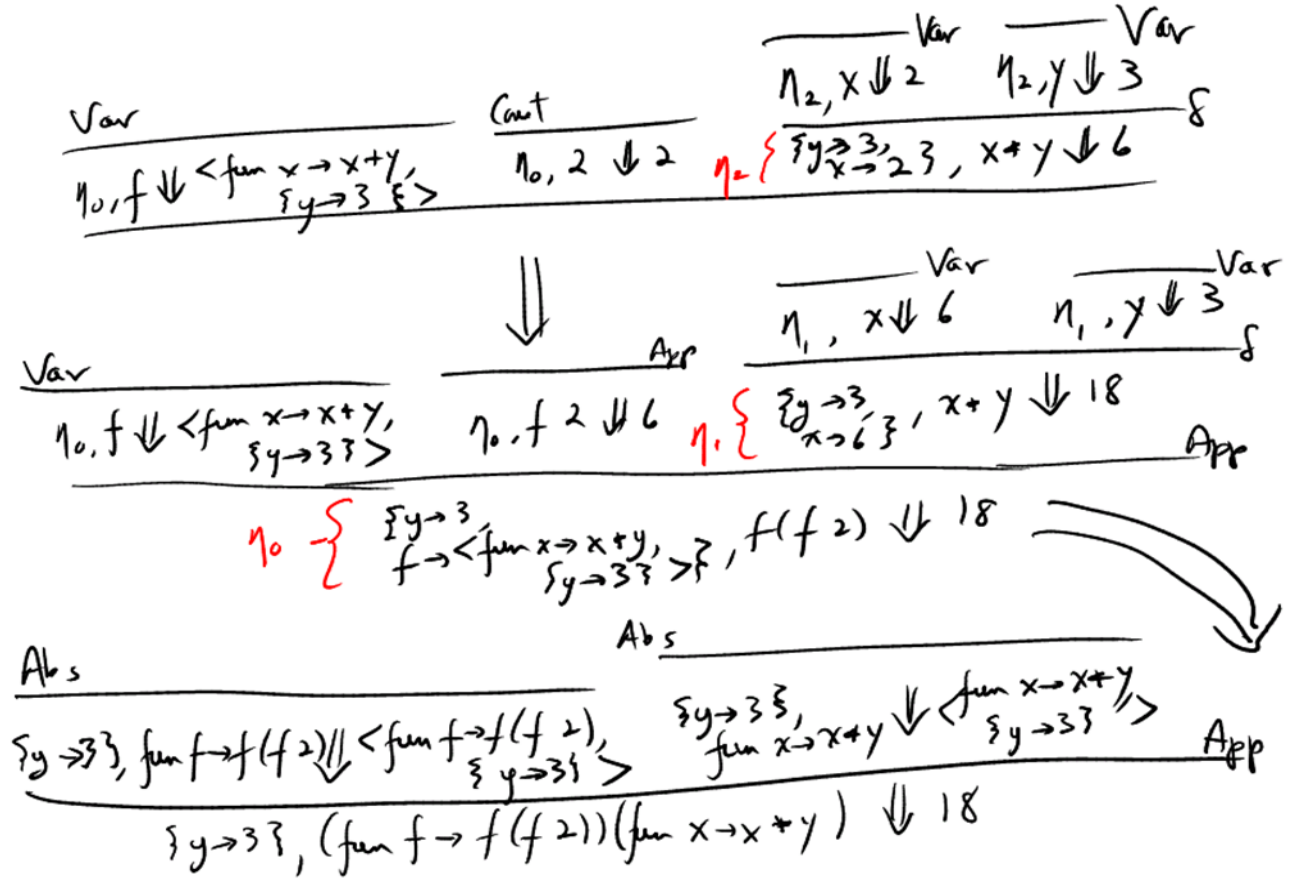
You may wish to write your answer on the back of this sheet; if so, please indicate that you have done this.

$$\dfrac{\dfrac{\dfrac{}{\emptyset, \text{fun } x \to \ldots \Downarrow \langle \text{fun } x \to \ldots, \emptyset \rangle}\,\text{Abs} \quad \dfrac{}{\emptyset, 3 \Downarrow 3}\,\text{Const} \quad \dfrac{}{\{x \to 3\}, \text{fun } y \to \ldots \Downarrow \langle \text{fun } y \to \ldots, \{x \to 3\} \rangle}\,\text{Abs}}{\emptyset, (\text{fun } x \to \text{fun } y \to x+y)\ 3 \Downarrow \langle \text{fun } y \to x+y, \{x \to 3\} \rangle}\,\text{App} \quad \dfrac{\dfrac{}{\eta_0, x \Downarrow 3}\,\text{Var} \quad \dfrac{}{\eta_0, y \Downarrow 4}\,\text{Var}}{\eta_0, x+y \Downarrow 7}\,\delta \quad \dfrac{}{\emptyset, 4 \Downarrow 4}\,\text{Const}}{\emptyset, (\text{fun } x \to \text{fun } y \to x+y)\ 3\ 4 \Downarrow 7}\,\text{App}$$

(c) (10 pts) Give a proof tree for the following judgment in OS_{clo}:

$$\{\texttt{y} \mapsto 3\}, (\texttt{fun f} \to \texttt{f(f 2)})(\texttt{fun x} \to \texttt{x} * \texttt{y}) \Downarrow 18$$

You may wish to rotate the page.



12. **Type-checking** (10 pts) The following questions use type system T_{OCaml}, which is given at the end of this exam.

    (a) (5 pts) Consider the following type judgment:

    $\emptyset \vdash (\texttt{fun f} \rightarrow \texttt{f 5})(\texttt{fun x} \rightarrow \texttt{x} + 1) : \tau$

    What type should be filled in for τ?

    int

    What is the first rule that would be applied in the derivation of this judgment? Apply the rule and show the premises that remain. You may fill in further steps if it helps to figure out the correct types.

    Application rule:

$$\frac{\emptyset \vdash \text{fun f -> f 5: } (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \qquad \emptyset \vdash \text{fun x -> x+1: int} \rightarrow \text{int}}{\emptyset \vdash \text{(fun f -> f 5)(fun x -> x+1): int}}$$

    (b) (5 pts) Consider the following type judgment:

    $\emptyset \vdash \texttt{let f} = \texttt{fun x y} \rightarrow (\texttt{x}, \texttt{y})\ \texttt{in}\ (\texttt{f 2 "a"},\ \texttt{f true}\ 5) : \tau$
    What type should be filled in for τ?

    τ = **(int * string) * (bool * int)**

    What is the first rule that would be applied in the derivation of this judgment? Apply the rule and show the premises that remain. You may fill in further steps if it helps to figure out the correct types.

    Let rule:

$$\frac{\emptyset \vdash \text{fun x y -> (x,y)} : \alpha \rightarrow \beta \rightarrow \alpha * \beta \qquad \emptyset[f : \forall \alpha, \beta . \alpha \rightarrow \beta \rightarrow \alpha * \beta] \vdash \text{(f 2 "a", f true 5): } \tau}{\emptyset \vdash \texttt{let f} = \texttt{fun x y} \rightarrow (\texttt{x}, \texttt{y})\ \texttt{in}\ (\texttt{f 2 "a"},\ \texttt{f true}\ 5) : \text{(f 2 "a", f true 5): } \tau}$$

    where τ is as given above.

13. **Hoare logic** (10 pts + 10 pts XC)

(a) (10 pts) Let S be the following code, where i is an integer and b a boolean:

```
while (i > 0)
{
  i = i - 1;
  b = !b;
}
```

Let P be the pre-condition "b = true & i=i0 & i $\geq$ 0." Write condition Q that *fully defines* `b` in terms of `i` at the end of the loop, such that: P { S } Q can be proven. Then write condition I that can serve as an invariant of the loop to prove P { S } Q.

Q:

b = even(i0)

I:

b = even(i0 - i)

(b) (10 pts XC) In the following loop — call it S — x, y, and z are lists; S just splits x in two, putting alternate elements on y and z:

```
while (x != []) {
   y = hd(x) :: y;
   x = tl(x);
   if (x != []) {
      z = hd(x) :: z;
      x = tl(x);
   }
}
```

Write a termination function $\phi(x, y, z)$. Recall that this is an integer-valued function of the variables of the loop that has these properties: it is always non-negative; and its value at the end of each iteration is strictly less than its value at the start of the iteration.

$\phi(x, y, z)$ = **length(x)**

Rules of T_{OCaml}:

$$\textbf{(Const)}\ \frac{}{\Gamma \vdash c : int}$$

$$\textbf{(Var)}\ \frac{\Gamma(x) = \sigma \qquad \tau \leq \sigma}{\Gamma \vdash x : \tau}$$

$$\textbf{(Application)}\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\textbf{(Abstraction)}\ \frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \texttt{fun}\ \ x \to e : \tau \to \tau'}$$

$$\textbf{(let)}\ \frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma[x : GEN_\Gamma(\tau')] \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau}$$

Rules of OS_{CLO}:

$$\textbf{(Const)}\ \frac{}{\eta, k \Downarrow k}$$

$$\textbf{(Var)}\ \frac{}{\eta, x \Downarrow \eta(x)}$$

$$\textbf{(Abstr)}\ \frac{}{\eta, \texttt{fun}\ \ x \to e \Downarrow \langle \texttt{fun}\ \ x \to e, \eta \rangle}$$

$$(\delta)\ \frac{\eta, e_1 \Downarrow v_1 \qquad \eta, e_2 \Downarrow v_2 \qquad v = v1 \oplus v2}{\eta, e_1 \oplus e_2 \Downarrow v}$$

$$\textbf{(App)}\ \frac{\eta, e_1 \Downarrow \langle \texttt{fun}\ \ x \to e, \eta' \rangle \qquad \eta, e_2 \Downarrow v \qquad \eta'[x \to v], e \Downarrow v'}{\eta, e_1 e_2 \Downarrow v'}$$