CS421 Spring 2008 Final Exam

Tuesday, May 6, 2008

Name:	**Answer sheet**
NetID:	

- You have **three hours** 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 a proctor.
- Including this cover sheet, there are 19 pages to this exam. Please verify that you have all pages.
- Please write your name and NetID in the spaces above, and also at the top of every page.

Question	**Possible points**	**Points earned**	**Graded by**
1	6		
2	5		
3	5		
4	6		
5	6		
6	6		
7	7		
8	6		
9	9		

Question	**Possible points**	**Points earned**	**Graded by**
10	6		
11	8		
12	8		
13	6		
14	8		
15/16	8		
Total	**100**		
15/16	8		
17	5		
EC total	**13**		

1. (6 pts.) Define the following OCaml functions. Avoid the use of library functions (including @), except hd and tl.

a. `sum` : int $\rightarrow$ int such that if n > 0, then sum n = 1 + ... + n,and sum n = 0 otherwise.

```
let rec sum x = if x<1 then 0 else x+sum (x-1);
```

b. `zip`: α list $\rightarrow$ β list $\rightarrow$ (α * β) list, such that `zip` *[a1;a2;...] [b1;b2;...] = [(a1,b1);(a2,b2);...]* Assume the two lists have the same length.

```
let rec zip x y = if x=[]
                  then []
                  else (hd x, hd y):: zip (tl x) (tl y);
```

c. `unzip`: (α * β) list $\rightarrow$ (α list * β list), the inverse of `zip`, i.e. `unzip` *[(a1,b1);(a2,b2);...]* = *([a1;a2;...], [b1;b2;...]).*

```
let rec unzip x = match x with
            [] -> ([],[])
          | (a,b)::c -> let (lis1,lis2) = unzip c
                        in (a::lis1, b::lis2);
```

2. (5 pts) Assume the following abstract syntax:

```
type stmt = Assign of string * expr
          | If of expr * stmt * stmt
          | While of expr * stmt
          | Block of stmt list

and expr = Var of string | Const of int
        | Plus of expr * expr | Less of expr * expr | Not of expr
```

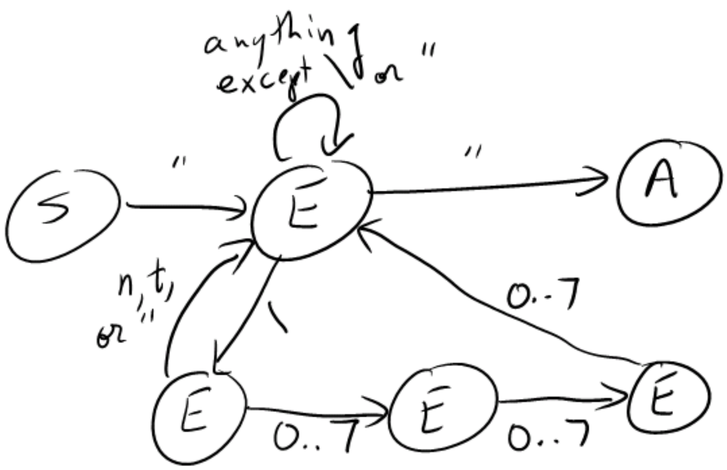
Write a function `trans`: `stmt` → `stmt` that makes the following transformations:

- `if (!e) then s1 else s2` ⇒ `if (e) then s2 else s1`
- `{ s }` ⇒ `s` (i.e. a block with a single statement doesn't need to be a block)

These transformations should be performed recursively throughout the term – inside the body of a while, the statements in a block (as well as the block itself), and the true and false branches of an if (as well as the if itself).

```
let rec transform s = match s with
    Assign(x,e) -> s
  | If(Not e,s1,s2) -> If(e, transform s2, transform s1)
  | If(e,s1,s2) -> If(e, transform s1, transform s2)
  | While(e,s) -> While(e, transform s)
  | Block [s] -> transform s
  | Block sl -> Block (map transform sl);
```

3. (5 pts) A string is a sequence of characters within double quotes. Further, it may contain escape sequences; \n, \t, \", and \nnn, where the n's are octal digits (0-7). Write a finite-state machine for strings (including opening and closing quotes). Note that a backslash *must* be followed by n, t, “, or three octal digits, or it is an error. The states should be marked with one of the letters S (start state), A (accept state), and E (error state).



4. (6 pts) Given this definition of an abstract syntax for expressions and a "fold" function on expressions:

```
type expr = Int of int  |  Add of expr * expr

let rec fold (f,g) e = match e with
    Int i  -> f i
  | Add(e1,e2) -> g (fold (f,g) e1, fold (f,g) e2)
```

fill in the blanks in the following OCaml session. (Recall that `string_of_int` is the OCaml function to convert an int to a string.):

```
val e1 = Add(Int 3, Add(Int 4, Int 5))

let evaluate e = fold (_(fun n -> n)_______________________,

                       _(fun (x,y) -> x+y) _______________) e;;
evaluate e1;;
-: int = 12

let prettyprint e = fold

             (____string_of_int __________________________,

              ___ fun (x,y) -> "("^x^"+"^y^")" _____________) e;;
prettyprint e1;;
-: string = "(3+(4+5))"

let countadds e = fold

             (__(fun n -> 0)_______________________________,

              _ fun (n,p) -> n+p+1_______________________) e;;
countadds e1;;
-: int = 2
```

5. (6 pts) Consider this grammar:

S $\rightarrow$ id int
| id id int
| D int

D $\rightarrow$ ε
| D $

This grammar is not ambiguous, but it is not LL(1).

a. Give two reasons why this isn't an LL(1) grammar.

1. FIRST sets of first two right-hands sides of S overlap.

2. D is left-recursive

b. Give an LL(1) grammar for this language.

S $\rightarrow$ id T
| D int

T $\rightarrow$ int | id int
| D int

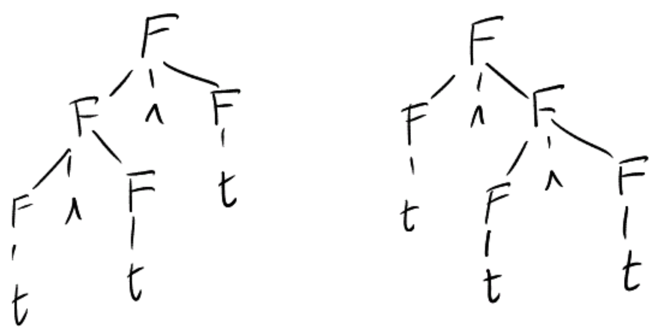
D $\rightarrow$ ε
| $ D

6. (6 pts) Propositional formulas have variables, constants t and f, and operators $\wedge$ (and) and $\vee$ (or):

$$F \to t \mid f \mid var \mid F \wedge F \mid F \vee F \mid (F)$$

(a) Give a sentence in this grammar that has two parse trees, and show those trees.

t $\wedge$ t $\wedge$ t



(b) Give a stratified grammar that gives precedence to $\wedge$ over $\vee$, and gives both left-associativity.

$$\begin{aligned}
&\mathbf{F} \to \mathbf{T} \ \mid \ \mathbf{F} \vee \mathbf{T} \\
&\mathbf{T} \to \mathbf{P} \ \mid \ \mathbf{T} \wedge \mathbf{P} \\
&\mathbf{P} \to \mathbf{t} \ \mid \ \mathbf{f} \ \mid \ \mathbf{var} \ \mid \ (\mathbf{F})
\end{aligned}$$

7. (7 pts) Fill in the blanks:

a) In a language like Java or C++, local variables have space allocated in the ____**stack**_________; primitive values like integers go directly in those locations, but objects, allocated using "new", go in the ___**heap**______.

b) Immediate execution of a program, without translation to a more primitive language, is called __**interpretation**_______.

c) Translation of a program into a more primitive language is ___**compilation**______.

d) A chunk of data giving the return address and arguments of a function call, created at the time of the call, is the _**activation record**, or **stack frame**__.

e) An example compiled language is _**C, C++, Java, Fortran, etc.**_____**.**

f) An example interpreted language is _**Lisp, OCaml, Python, Perl, etc**. _____.

8. (7 pts) This question concerns the translation of programs to a 3-address intermediate representation. Recall that we defined the following translation schemes in class:

[S] = instructions to compute statement S
[e]$_{\mathbf{tlab,flab}}$ = instructions to calculate boolean-valued expression e and jump to tlab if it is true, flab otherwise. (This is called the "short-circuit evaluation" scheme.)

Some languages have multi-level break statements: **break *n*** breaks out of *n* levels of while statements. (For purposes of this question, ignore switch statements.) In such a language, we need a translation scheme of the form:

[S]$_{\mathbf{BL}}$**,** where BL is a list of labels, b_1,..., b_n. This is the translation of S, given that it is within *n* while statements, and labels b_1,..., b_n are the labels of the instructions that follow those containing statements, from innermost to outermost. E.g., "break 1" in S should jump to b_1, a "break 2" should jump to b_2, etc.

For this translation scheme, we can give these translations:

[while (e) S]$_{\mathbf{BL}}$ = let wlab, tlab, flab = new labels in
  wlab: $[e]_{tlab,flab}$
  tlab: $[S]_{flab::BL}$ (where flab::BL is BL with flab added at the front)
  JUMP wlab
  flab:

[break *n*]$_{\mathbf{BL}}$ = JUMP b_n

Such languages also have multi-level continue, where **continue *n*** terminates the current iteration of the ***n***th enclosing while loop and goes on to the next iteration; **continue** is equivalent to **continue 1**. This requires a translation scheme like this:

[S]$_{\mathbf{BL,CL}}$**,** where BL is a list of labels b_1, b_2, ..., b_n and CL is a list of labels c_1, c_2, ..., c_n. This is the translation of S to intermediate form, given that it is within *n* while statements, labels b_1, b_2, ..., b_n are the labels of the instructions that follow those containing statements, from innermost to outermost, and c_1, c_2, ..., c_n are the labels beginning the next iteration of those containing loops. That is, a "break 1" in S should jump to b_1, and a "continue 1" should jump to c_1, etc.

Give the new translations for **while**, **break *n***, and **continue *n***:

[while (e) S]$_{\mathbf{BL,CL}}$ = **let wlab, tlab, flab = new labels in**
  wlab: **[e]**$_{\mathbf{tlab,flab}}$
  tlab: **[S]**$_{\mathbf{flab::BL,\ wlab::CL}}$
  JUMP wlab
  flab:

[break *n*]$_{\mathbf{BL,CL}}$ = **JUMP** $\mathbf{b_n}$

[continue *n*]$_{\mathbf{BL,CL}}$ = **JUMP** $\mathbf{c_n}$

9. (9 pts) For this question, recall the definition of fold_right:

```
let rec fold_right f lis accu =
     match lis with
            [] -> accu
          | h::t -> f h (fold_right f t accu)
```

Write the following OCaml functions:

(a) Write `map` using `fold_right` (not using explicit recursion). (The definition of map is given in problem 15.)

```
let map f lis = fold_right (fun x y -> f x :: y) lis []
```

(b) `repeat`: int $\rightarrow$ ($\alpha \rightarrow \alpha$) $\rightarrow \alpha \rightarrow \alpha$, where `repeat` *n f* produces a function that applies *f n* times.

```
let rec repeat n f x = if n = 0 then x else repeat (n-1) f (f x)
```

(c) `graph_fun`: ($\alpha \rightarrow \beta$) $\rightarrow \alpha$ list $\rightarrow$ ($\alpha * \beta$) list, where `graph_fun` *f [x1; x2; ...; xn] = [(x1, f x1); (x2, f x2); ...]*

```
let rec graph_fun f x =
 if x=[] then [] else (hd x, f (hd x)):: graph_fun f (tl x)
```

10. (6 pts) Give the environment after each of the following OCaml definitions. Assume the execution starts with the environment ∅. We've named each environment, and you can use these names in subsequent environments; we've also filled in the first line.

let x = 4;;

ρ_0: **$\{x \rightarrow 4\}$**

---

let f y = fun z -> x + y + z;;

ρ_1: **$\rho_0[f \rightarrow$ <y, z -> x + y + z, ρ_0>]**

---

let x = 8;;

ρ_2: **$\rho_1[x \rightarrow 8]$**

---

let g = f 6;;

ρ_3: **$\rho_2[g \rightarrow$ <z, x + y + z, $\rho_0[y \rightarrow 6]$>]**

---

let x = g x;;

ρ_4: **$\rho_3[x \rightarrow 18]$**

---

11. (8 pts) The expression

```
e = (fun x -> let f = fun y -> x+y in f 4)5
```

evaluates to 9 in an empty environment. We have given part of the derivation tree for the judgment $\emptyset$, e $\Downarrow$ 9, in the environment-based dynamic semantics. The rules of this system are given at the end of this exam. **Complete the derivation by filling in the dashed blank lines.** (For space reasons, the proof is broken into sections.)

Define: e' = `fun x -> let f = fun y -> x+y in f 4` (so e = e' `5`)
e'' = `let f = fun y -> x+y in f 4`
e''' = `fun y -> x+y`

$$\dfrac{\dfrac{}{\emptyset,\ e' \Downarrow \underline{\mathbf{<x, e'', \emptyset>}}} \qquad \dfrac{}{\emptyset,\ 5 \Downarrow 5} \qquad \dfrac{}{\{x\to 5\},\ e'' \Downarrow 9}\,(A)}{\emptyset,\ e \Downarrow 9}$$

(A)

$$\dfrac{\dfrac{}{\{x\to 5\},\ e''' \Downarrow \underline{\mathbf{<y, x+y, \{x\to 5\}>}}} \qquad \dfrac{}{\rho,\ f\ 4 \Downarrow 9}\,(B)}{\{x\to 5\},\ e'' \Downarrow 9}$$

where ρ = $\underline{\mathbf{\{x\to 5,\ f \to <y, x+y, \{x\to 5\}> \}}}$

(B)

$$\dfrac{\dfrac{}{\rho,\ f \Downarrow \underline{\mathbf{<y, x+y, \{x\to 5\}}}>} \qquad \dfrac{}{\rho,\ 4 \Downarrow 4} \qquad \dfrac{\dfrac{}{\rho',\ x \Downarrow 5} \qquad \dfrac{}{\rho',\ y \Downarrow 4}}{\rho',\ x+y \Downarrow 9}}{\rho,\ f\ 4 \Downarrow 9}$$

where ρ' = $\underline{\mathbf{\{ x\to 5,\ y\to 4 \}}}$

12. (8 pts) As you know, the expression

```
let x = e1 and y = e2 in e
```

evaluates e1 and e2 in the same environment (that is, e2 is not in the scope of x). Suppose we had a different let expression:

```
let x = e1 then y = e2 in e
```

in which e2 was in the scope of x. That is, "`let x=3 then y=x+4 in y`" would yield 7.

a. Give a dynamic semantics rule for this expression:

$$\frac{\rho, e_1 \Downarrow v_1 \qquad \rho[x \to v_1], e2 \Downarrow v_2 \qquad \rho[x \to v_1, y \to v_2], e \Downarrow v}{\rho, \texttt{let x = e1 then y = e2 in e} \Downarrow v}$$

b. Give a type rule for this expression (in the non-polymorphic type system):

$$\frac{\Gamma \vdash e1: \tau_1 \qquad \Gamma[x: \tau_1] \vdash e2: \tau_2 \qquad \Gamma[x: \tau_1, y: \tau_2] \vdash e: \tau}{\Gamma \vdash \texttt{let x = e1 then y = e2 in e}: \tau}$$

The dynamic semantics and type-checking rules are given at the end of this exam.

13. (6 pts) Consider these definitions in OCaml:

```
let newCounter () =
    let cnt = ref 0
    in (fun n -> cnt := n,
        fun () -> (cnt := !cnt + 1; !cnt)
let reset n (a,b) = a n
let next (a,b) = b ()
```

Recall that the type of () is unit, and the value of an assignment $e_1 := e_2$ is unit.

a. Give the types of newCounter, reset, and next:

`type counter =` __ **(int → unit) * (unit → int)**______

`newCounter`: unit → counter

`reset`: **_int → counter → unit, or $\alpha \rightarrow (\alpha \rightarrow \beta) * \gamma \rightarrow \beta$** ________

`next`: **_ counter → int, or $\alpha * (\text{unit} \rightarrow \beta) \rightarrow \beta$**__________

b. Fill in the blanks in this OCaml session, giving the value returned by the prior expression:

```
let c1 = newCounter();;
next c1;;

__1_______________
next c1;;

__2_______________
let c2 = c1;;
reset 10 c2;;
next c1;;

__11______________
```

14. (8 pts) For this problem, we ask you to construct a type derivation (using the non-polymorphic type system). Let the type environment Γ be

$$\Gamma = \{\text{cons: int} \rightarrow \text{int list} \rightarrow \text{int list, nil: int list}\}.$$

Give the proof tree for the type judgment below, using the lines provided. On each line, give the name of the inference rule being used. Recall that axioms have a line with nothing above it. The axioms and rules of inference for the system are given at the end of the exam.

__________________________*Variable* ___ _____ *Variable* __

Γ[x:int] |- **cons : int → int list → int list** **Γ[x:int]** |- **x : int**

*App*____ _________ *Variable* _

Γ[x:int] |- cons x : int list → int list **Γ[x:int]** |- nil : int list

____*Const* ____*App*

Γ |- 1:int **Γ[x:int]** |- cons x nil : int list

____*Let*

Γ |- let x = 1 in cons x nil : int list

Do either 15 or 16 (your choice). You may do the other for extra credit.

15. (8 pts) Given these definitions of compose and map:

```
let compose f g = fun x -> f (g x)
let map f x = if x=[] then [] else f (hd x) :: map f (tl x)
```

prove that

`map f (map g x) = map (compose f g) x,` for all `x.`

We provide part of the proof, and you are to complete it.

Proof By induction on the length of x.

Base case: x = []: `map f (map g [])` = `map f []` = `[]` = `map (compose f g) []`

Inductive case: Assume `map f (map g x) = map (compose f g) x`, and prove `map f (map g (a::x)) = map (compose f g) (a::x).`

```
map f (map g (a::x))

=  if map g (a::x) = []                              (def of map f)
   then []
   else f (hd (map g (a::x))) :: map f (tl (map g (a::x)))

=  if (g a :: map g x) = []                          (def of map g)
   then []
   else f (hd (map g (a::x))) :: map f (tl (map g (a::x)))
```

(complete this proof; make sure to justify each step)

```
= f (hd (map g (a::x))) :: map f (tl (map g (a::x)))
                                                     (g a::… is not null)

= f  (hd (g a :: map g x)) :: map f (tl (map g (a::x)))  (def of map)

= f (g a) :: map f (tl (map g (a::x)))               (def of hd)

= f (g a) :: map f (tl (g a :: map g x))             (def of map)

= f (g a) :: map f (map g x)                         (def of tl)

= f (g a) :: map (compose f g) x                     (ind. hyp.)

= (compose f g) a :: map (compose f g) x             (def. of compose)

= map (compose f g) a::x                             (def. of map)
```

16. (8 pts) The following code is similar to the "partition" portion of quicksort:

```
i = 0; j = n-1;
while (i < j) {
   if (a[i] <= x)
      i = i+1;
   else if (a[j] > x)
      j = j-1;
   else {
      temp = a[i]
      a[i] = a[j]
      a[j] = temp
      i = i+1
      j = j-1
   }
}
```

The correctness formula for this statement is:

true { `i=0; j=n-1; while ...` } $\exists k.\ (0 \le k \le n\text{-}1$
$\wedge\ (\forall m.\ 0 \le m < k \Rightarrow a[m] \le x)$
$\wedge\ (\forall m.\ k < m < n \Rightarrow a[m] > x))$

(a) Give the loop invariant for the loop.

$\boldsymbol{\exists i, j.\ (0 \le i \le j \le n\ \wedge\ (\forall m.\ 0 \le m < i \Rightarrow a[m] \le x)}$
$\boldsymbol{\wedge\ (\forall m.\ j \le m < n \Rightarrow a[m] > x))}$

(b) Give a well-founded ordering on the variables that proves the termination of the loop.

Numerical ordering on j-i. (Declines on every iteration; cannot go below -1.)

17. (**Extra credit,** 5 pts) Suppose the following C++ class and function were defined:

```
abstract class FunObj {
   virtual int apply (int) = 0;
}

void map (FunObj f, int[] a, int n) {
   for (int i=0; i<=n; i++)
      a[i] = f.apply(a[i]);
}
```

a. Define classes **`decrobj`** and **`sqrobj`** as subclasses of **`FunObj`** so that **`map (new decrobj(), a, n)`** decrements each element of **a**, and **`map (new sqrobj(), a, n)`** squares each element of **a.**

```
class decrobj : FunObj {
     virtual int apply (int n) { return n-1; }
}

class sqrobj : FunObj {
     virtual int apply (int n) { return n*n; }
}
```

b. Define a subclass **`compose`** of **`FunObj`**

```
class Compose : public FunObj {
  FunObj *f, *g;

  Compose (FunObj *f, FunObj *g) {
       this->f = f;
       this->g = g;
  }

  int apply (int n) {

       return f->apply (g->apply n);


  }

}
```

that composes function objects, so that, for example, **`map(new Compose(new sqrobj(), new decrobj()), a, n)`** changes every element a[i] to $(a[i]-1)^2$.

Dynamic semantics

$$\frac{}{\rho\,,\ \mathtt{n} \Downarrow n}\ \text{(constant rule)}$$

$$\frac{}{\rho\,,\ \mathtt{x} \Downarrow \rho(\mathtt{x})}\ \text{(variable rule)}$$

,

$$\frac{}{\rho\,,\ \mathtt{fun\ x\ ->\ e} \Downarrow \langle x, e, \rho\rangle}\ \text{(function rule)}$$

$$\frac{\rho\,,\mathtt{e}_1 \Downarrow \langle x, e, \rho'\rangle \qquad \rho\,,\mathtt{e}_2 \Downarrow v' \qquad \rho'[x\to v']\,,\, e \Downarrow v}{\rho\,,\mathtt{e}_1\ \mathtt{e}_2 \Downarrow v}\ \text{(app rule)}$$

$$\frac{\rho\,,\mathtt{e}_1 \Downarrow n_1 \qquad \rho\,,\mathtt{e}_2 \Downarrow n_2}{\rho\,,\mathtt{e}_1 + \mathtt{e}_2 \Downarrow n_1+n_2}\ \text{(plus rule)}$$

$$\frac{\rho\,,\mathtt{e} \Downarrow v' \qquad \rho[x\to v']\,,\, e' \Downarrow v}{\rho\,,\mathtt{let\ x\ =\ e\ in\ e'} \Downarrow v}\ \text{(let rule)}$$

Non-polymorphic type system

$$\frac{}{\Gamma \ \vdash\ \mathtt{n\ :\ int}}\ \text{(constant rule)}$$

$$\frac{}{\Gamma \ \vdash\ \mathtt{x}\ :\ \Gamma(\mathtt{x})}\ \text{(variable rule)}$$

$$\frac{\Gamma[x:\tau]\ \vdash\ \mathtt{e}\ :\tau'}{\Gamma\ \vdash\ \mathtt{fun\ x\ ->\ e}:\tau\to\tau'}\ \text{(function rule)}$$

$$\frac{\Gamma\ \vdash\ \mathtt{e}\ :\tau\to\tau' \qquad \Gamma\ \vdash\ \mathtt{e'}\ :\tau}{\Gamma\ \vdash\ \mathtt{e\ e'}:\tau'}\ \text{(app rule)}$$

$$\frac{\Gamma\ \vdash\ \mathtt{e}\ :\tau' \qquad \Gamma[x:\tau']\ \vdash\ \mathtt{e'}\ :\tau}{\Gamma\ \vdash\ \mathtt{let\ x\ =\ e\ in\ e'}:\tau}\ \text{(let rule)}$$