CS421 Spring 2009 Final

Monday, May11, 2009

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.
- If you believe there is an error, or an ambiguous question, seek clarification from one of the TAs.
- Including this cover sheet, there are 16 pages to the exam. Please verify that you have all pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Possible points**	**Points earned**	**EC points**
1	10		
2	8		
3	8		
4	12 + 4EC		
5	8		
6	8		
7	6		
8	6		
9	4		
10	12		
11	10		
12	8 + 4EC		
Total	100 + 8EC		

1. (10 pts) Define the following OCaml functions:

(a) maxmin: int list -> int * int returns a pair of the largest number and the smallest number in a list. E.g. maxmin[1;5;4;2;6;3;8;7;2;9] = (9,1). You may not use any library function except hd and tl; you may assume that the argument is a non-empty list.

```
let rec maxmin lis = match lis with
        [x] -> (x, x)
      | x::xs -> let (hi,lo) = maxmin xs
             in if x<lo then (hi,x)
                else if x>hi then (x,lo)
                     else (hi,lo)
```

(b) remove: α -> α list -> α list removes all occurrences of an element from a list. remove 5 [1;3;5;6;5;8] = [1;3;6;8].

```
let rec remove x lis = match lis with
      [] -> []
    | y::ys -> if x=y then remove x ys else y::remove x ys
```

(c) The higher-order function removeAll has type (α -> bool) -> α list -> α list; removeAll p l removes all the elements of l that satisfy p.

```
let rec removeAll p lis = match lis with
      [] -> []
    | y::ys -> if p y then removeAll p ys else y::removeAll p ys
```

(d) Now use removeAll to define remove:

remove x lis = removeAll (_____**fun y -> y=x**__________) lis

(e) Suppose you want to remove only the *first* element of a list that satisfies a predicate. Define removeFirst: (α -> bool) -> α list -> α list such that if you apply removeFirst instead of removeAll in problem (d) it would remove just the first occurrence of x.

```
let rec removeFirst p lis = match lis with
      [] -> []
    | y::ys -> if p y then ys else y::removeFirst p ys
```

2. (8 pts) We can define an extremely simple set of expressions with this grammar:

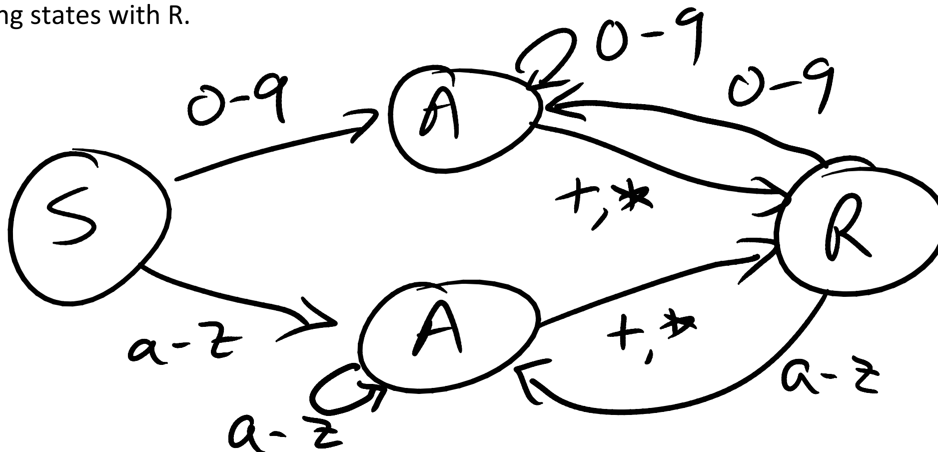
Exp -> int | variable | Exp + Exp | Exp * Exp

where an int is a sequence of one or more decimal digits, and a variable is a sequence of one or more lower-case letters.

(a) This set of expressions can also be defined by a single regular expression. Write this regular expression in ocamllex style:

(['0'-'9']+ | ['a'-'z']+) (('+'|'*') (['0'-'9']+ | ['a'-'z']+))*

(b) Write a DFA for this set of expressions. Label the start state with S, accepting states with A, and rejecting states with R.



(c) If we were to parse these expressions in ocamlyacc, we would want to use ocamllex to divide the input into tokens, the tokens being: int, identifier, +, and *. Fill in this ocamllex specification to do that:

type token = INTEGR_LITERAL of int | PLUS | TIMES | IDENTIFIER of string
let letter = ['a' - 'z']

let digit = **['0'-'9']**

let letters = **letter+**

let digits = **digit+**

rule tokenize = parse

| letters as s **{ IDENTIFIER s }**

| digits as n { INTEGER_LITERAL (int_of_string n) }

| '+' { PLUS }

| '*' { TIMES }

3. (8 pts) We gave a number of translation schemes for translating ASTs to intermediate form in class. For this question, you will need to remember two of them:

$[e]_x$: translate expression e to code that stores value of e in variable x
$[e]_{Lt,Lf}$: translate expression e to code that branches to Lt if e is true, or Lf otherwise

Suppose a language has logical connectives “nand” and “nor.” In this question, we will explore several translation schemes for these operators. First, we remind you of some translations for other operators:

$[e1 \&\& e2]_x$ =

```
     let L1, L2, L3, L4 = newlabel()
         y, z = newlocation()
     in      [e1]y
             CJUMP y,L1,L2
     L1:     [e2]z
             CJUMP z,L3,L2
     L3:     x = true
             JUMP L4
     L2:     x = false
     L4:
```

$[e1 \&\& e2]_{Lt,Lf}$ = $[e1]_{L1,Lf}$
L1: $[e2]_{Lt,Lf}$

$[e1 \mid\mid e2]_{Lt,Lf}$ = $[e1]_{Lt,L1}$
L1: $[e2]_{Lt,Lf}$

$[!e]_{Lt,Lf}$ = $[e]_{Lf,Lt}$

(a) Fill in the lines in these definitions:

$[e1 \text{ nand } e2]_{Lt,Lf}$ = $[!(e1 \&\& e2)]_{Lt,Lf}$ = ____**$[e1]_{L1,Lt}$**________

_**L1: $[e2]_{Lf,Lt}$**____________

$[e1 \text{ nor } e2]_{Lt,Lf}$ = $[!(e1 \mid\mid e2)]_{Lt,Lf}$ = ____**$[e1]_{Lf,L1}$**____________

_**L1: $[e2]_{Lf,Lt}$**___________

(b) Give this translation (analogous to $[e1 \&\& e2]_x$ above):

$[e1 \text{ nand } e2]_x$ =

```
     let L1, L2, L3, L4 = newlabel()
         y, z = newlocation()
     in      [e1]y
             CJUMP y,L1,L2
     L1:     [e2]z
             CJUMP z,L3,L2
     L3:     x = false
             JUMP L4
     L2:     x = true
     L4:
```

4. (12 pts) This is an abstract syntax for "lambda calculus" – a very simplified OCaml:

```
type lambda = Var of string                     (* Var x corresponds to variable x *)
            | Fun of string * lambda            (* Fun(x,e) corresponds to fun x -> e *)
            | App of lambda * lambda            (* App(e1,e2) corresponds to e1 e2 *)
            | Let of string * lambda * lambda;  (* Let(x,e1,e2) corresponds to let x=e1 in e2 *)
```

For example, "fun x -> let y=x in x z" would correspond to AST: Fun("x", Let("y", Var "x", App(Var "x", Var "z"))).

(a) Write a function that transforms each occurrence of a Let expression to the application of an abstraction ("let x=e1 in e2" -> "(fun x->e2)e1"). It must transform every such occurrence, including ones that occur inside other occurrences. For example:

```
removeLets (Fun("x", Let("y", Var "x", App(Var "x", Var "z"))))
=  Fun ("x", App (Fun ("y", App (Var "x", Var "z")), Var "x"))
```

```
let rec removeLets e = match e with
```

```
let rec removeLets e = match e with
    Var s -> Var s
  | Fun(s,e) -> Fun(s, removeLets e)
  | App(e1,e2) -> App(removeLets e1, removeLets e2)
  | Let(s,e1,e2) -> App(Fun(s, removeLets e2), removeLets e1)
```

(b) (For this and the following questions, ignore "Let" expressions – just pretend they are not included in the type lambda. This is just to make the answers shorter.)

The function reduce will act like "fold" for lambdas. It is defined by:

```
let rec reduce f g h e = match e with
      Var s -> f s
    | Fun(s,e) -> g s (reduce f g h e)
    | App(e,e') -> h (reduce f g h e) (reduce f g h e')
```

Give the (somewhat lengthy) type of reduce (including quantified type variables, if appropriate):

reduce: $\forall\alpha.$ **(string** $\to \alpha) \to$ **(string** $\to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to$ **lambda** $\to \alpha$

(c) Use reduce to define the function freevars: lambda -> string list, which returns all variables occurring free (i.e. not within the scope of a Fun) in its argument; it is okay if it includes a given variable multiple times. (*Hint*: You will probably want to use the function remove, defined in question 1.)

```
let freevars = reduce (fun s -> [s]) (fun s sl -> remove s sl) (@)
```

(d) (4 EC pts) A simpler higher-order function operating on lambdas is mapLam, which applies a function just to the strings that occur in an expression, and returns an AST of the same shape as its argument. This is analogous to map on lists. E.g.

```
mapLam (fun s -> s^"2") (Fun ("x", App (Fun ("y", App (Var "x", Var "z")), Var "x")));;
 = Fun ("x2", App (Fun ("y2", App (Var "x2", Var "z2")), Var "x2"));
```

Define mapLam using reduce.

```
let mapLam f = reduce (fun s -> Var (f s))
                 (fun s e -> Fun(f s,e))
                 (fun e e' -> App(e,e'));
```

5. (8 pts) A grammar is given below, where terminals are written in boldface.

> Exp -> Decl **in** Exp | **int** | **fun id ->** Exp
> Decl -> **let id =** Exp

Assume the tokens are defined as follows:

> type token = IN | INT of int | FUN | ID of string | ARROW | LET | EQ

Write a recursive descent parser for this language. A skeleton code is given for you below. This parser is a "recognizer:" it does not return an AST but instead a (token list) option. (Recall that the definition of option: type α option = Some α | None.)

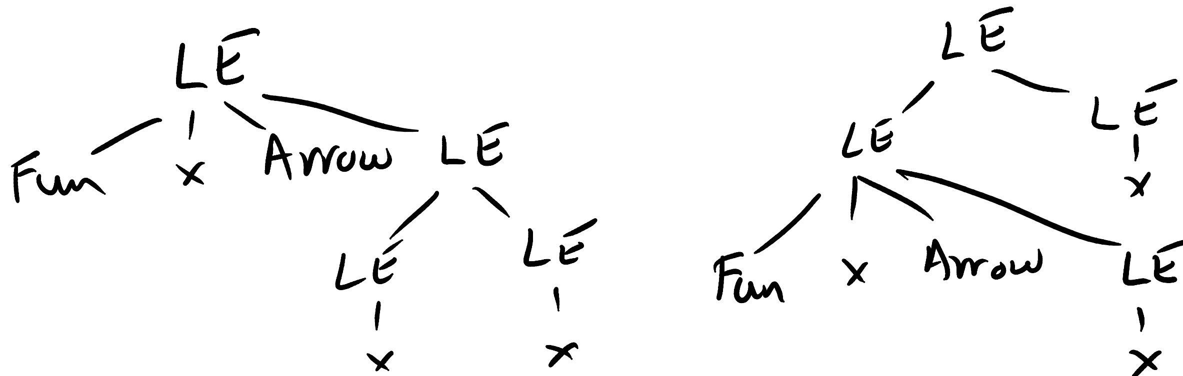
```
let rec parseDecl toklist =
        match toklist with
              LET::ID _::EQ::rest -> parseExp rest
              | _ -> None

and parseExp toklist =
        match toklist with
             INT n :: rest -> Some rest
             | FUN::ID _::ARROW::rest -> parseExp rest
             | _ -> (match parseDecl toklist with
                        Some (IN :: rest) -> parseExp rest
                      | _ -> None);;
```

6. (8 pts) Assume the tokens in the following grammar are Ident, Fun (keyword "fun"), and Arrow ("->"). This is a grammar for the lambda calculus ("LE" is for "Lambda Expression"):

LE -> Ident | Fun Ident Arrow LE | LE LE

(a) This grammar is ambiguous. There are exactly two parse tree for expression "fun x -> x x". Show both of them:



(b) Given this grammar, ocamlyacc will report a conflict, but will nonetheless produce a parser, in which any shift-reduce conflicts will be resolved by shifting. Give the sequence of shift-reduce steps that ocamlyacc will do when parsing "fun x -> x x". Among all the actions, there is one shift action that is taken because of the rule just stated, i.e. where there is a shift/reduce conflict; reducing at that step is possible and would lead to the other parse tree; circle this one shift action.

	Action	Stack	Input
1.	Sh		fun x -> x x
2.	**Sh**	fun	x -> x x
3.	**Sh**	**fun x**	**-> x x**
4.	**Sh**	**fun x ->**	**x x**
5.	**Red**	**fun x -> x**	**x**
6.	**Sh** (circled)	**fun x -> LE**	**x**
7.	**Red**	**fun x -> LE x**	
8.	**Red**	**fun x -> LE LE**	
9.	**Red**	**fun x -> LE**	
10.	Accept	LE	

7. (6 pts) Write the following APL expressions. You can use either APL or APL-in-OCaml notation, as you prefer. The APL cheat sheet is given at the end of the exam.

(a) Given a vector *v*, compute the vector of differences of consecutive elements of *v*. For instance, given the input vector [1; 4; 2; 9; 5], it should return the vector [-3; 2; -7; 4]. (Hint: You'll need to use the array subscripting operation @@.)

v[ι(ρv - 1)] - v[ι(ρv - 1) + 1]

(b) Given a number *n*, return an *n*-by-*n* matrix containing the numbers 1 through *n*^2 in reverse order, in row-major order. For example, if n is 2, the result is the matrix

4 3
2 1 .

(2ρn) ρ (n * n + 1 – ι(n * n))

8. (6 pts) This question concerns the definition of "large" values – in this case, a dictionary value – using higher-order functions in OCaml. A dictionary which associates keys of type α with values of type β, can be expressed as a function from α to β option. Define

```
type 'a 'b dictionary = 'a -> 'b option;;

type ‘a option = Some ‘a | None;;  (* reminder *)
```

A dictionary d maps k to v when m k = Some v, and contains no value for k when d k = None.

a) Define emptydict, the dictionary that contains no key-value pairs.

let emptydict = fun k -> None;;

b) Define the function insert d a b, which inserts the mapping of a to b into the dictionary d, overwriting any previous value associated with the key a.

let insert d a b = fun a' -> if a=a' then Some b else d a';;

9. (4 pts) We want to define the same representation for dictionaries in Java, using function objects, that we used in OCaml in the previous question. Complete the definition of insert in the following Java implementation of dictionaries, where the keys are of type int and the values are of type String. Use null to represent the value None.

```
interface IntStringDict {
        String apply (int x);
}

class DictOps {

   static IntStringDict emptydict = new IntStringDict(){
                                        String apply (int x) { return null; }
                                   };

   static IntStringDict insert (final IntStringDict d, final int a, final String b) {

        return new IntStringDict () {
            String apply (int x) {
                    if(x == a) return b;
                    else return d.apply(x);
                }
            };

        }
}
```

10. (12 pts) Operational semantics. The rules for OS_{clo} are given at the end of the exam. Instead of asking you to write one large proof tree, we're asking for several small ones. For problems a-c, fill in the complete proof trees (we've given skeletons), and write, on each line, the name of the axiom or rule of inference used in that line.

(a)

______________________(Var) **__________________________(Const)**

{x → 4}, x⇓ 4 **{x → 4}, 1 ⇓ 1** **(δ)**

{x → 4}, x+1 ⇓ 5

(b)

_______________________________(Abs) **________ __(Const)** **____________(Var)**

{y → 4}, fun x -> y⇓ < fun x -> y, {y → 4}> **{y → 4}, 3 ⇓ 3** **{y → 4, x→ 3}, y ⇓ 4**

{y → 4}, (fun x -> y) 3 ⇓ 4

(c) For this problem, we abbreviate value <fun x->y, {y → 3}> by the symbol κ.

__________________(Var) **________________(Const)** **__________________(Var)**

{g → κ}, g ⇓ κ **{g → κ}, 4 ⇓ 4** **{y → 3, x → 4}, y ⇓**

__(App)

{g → κ}, g 4 ⇓ 3

(d) Give an operational semantics rule for let expressions:

η, e1 ⇓ v' **η[x→v'], e2 ⇓ v** (Let)

η, let x=e1 in e2 ⇓ v

11. (10 pts) Give a proof in the T_{OCaml} system (see rules at back of exam) of the judgment:

$\varnothing \vdash$ let f = fun x -> true in f (f 3): bool

Hint: You can abbreviate the type environment {f : $\forall\alpha.\ \alpha \rightarrow$bool} by η_1. We recommend that you turn the paper sideways (landscape mode) to do this problem.

- (Const): $\{x:\alpha\} \vdash$ true : bool
- (Abs): $\varnothing \vdash$ fun x -> true : $\alpha \rightarrow$ bool
- (Var): $\eta_1 \vdash$ f : int $\rightarrow$ bool
- (Const): $\eta_1 \vdash$ 3 : int
- (App): $\eta_1 \vdash$ f 3 : bool
- (Var): $\eta_1 \vdash$ f : bool $\rightarrow$ bool
- (App): $\eta_1 \vdash$ f (f 3) : bool
- (Let): $\varnothing \vdash$ let f = fun x -> true in f (f 3) : bool

12. (8 pts) For these two program fragments, give an invariant for each loop. Keep in mind that the invariant must be true upon entering the loop (so the statements preceding the loop must make it true); it must, when combined with the negation of the loop condition, imply the post-condition; and it must, of course, be invariant.

(a) A is the program:

```
i = n;
p = 1;
while (i>0)  {
   p := p*2;
   i := i-1;
}
```

and the Hoare formula for the program as a whole is: n >= 0 { A } $p = 2^n$. Give the loop invariant.

$i \geq 0$ & $p = 2^{n-i}$

(b) B is the program

```
x := x0; s := 0;
while (x != []) {
    s := s + hd(x);
    x := tl(x);
}
```

and the Hoare formula for the program as a whole is: true {B} $s = \sum_{i=0}^{|x0|-1} hd\,(tl^i(x0))$.

$s = \sum_{i=0}^{|x0|-|x|-1} hd\,(tl^i(x0))$

(c) (4 EC pts) C is the program:

```
i := 0;
while (i<a.length)  {
   j := i+1;
   while ( j<a.length)  {
      if (a[i] > a[j])
           s := a[i]; a[i] := a[j]; a[j] := s;
      j++;
   }
   i++;
}
```

The overall Hoare formula is: $0 \leq |a|$ {C} $\forall k.\ 0 \leq k \leq |a|-2 \Rightarrow a[k] \leq a[k+1]$. ($|a|$ denotes the length of a. With indexing from zero, the valid indices of a are 0, …, $|a|-1$, so the post-condition asserts that the entire array is sorted. Give an invariant for the *outer* loop.

i ≤ a.length & ∀m. 0≤m<i-1 ⇒ a[m] ≤ a[m+1]
& ∀m,n. 0≤m<i & i≤n<|a| ⇒ a[m] ≤ a[n;]

APL Reference

Operation	*Expression*	*Value*
Sample data	`A ; a 2,3-matrix`	1 2 3 4 5 6
	`V ; a 3-vector`	2 4 6
	`C ; a logical 2-vector`	1 0
	`D ; a logical 3-vector`	1 0 1
Arithmetic	`A *@ A`	1 4 9 16 25 36
	`V -@ (newint 1)`	1 3 5
Relational	`A >@ (newint 4)`	0 0 0 0 1 1
Reduction	`!+ V`	12
	`maxR A`	3 6
Compression	`D % V`	2 6
	`C % A`	1 2 3 (a 1,3-matrix)
Shape	`shape A`	2 3
Ravelling	`ravel A` `ravel (newint 1)`	1 2 3 4 5 6 1
Restructuring	`rho (shape A) V`	2 4 6 2 4 6
	`rho (shape V) C`	1 0 1
Catenation	`A ^@ C`	1 2 3 4 5 6 1 0
Index generation	`indx (newint 5)`	1 2 3 4 5
Transposition	`trans A`	1 4 2 5 3 6
Subscripting	`V @@ (indx (newint 2))`	2 4
	`A @@ (newint 1)`	1 2 3 (a 1,3-matrix)
	`(trans A) @@ (indx (newint 2))`	1 4 2 5

Rules of T_{OCaml}

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

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

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

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

Rules of OS_{clo}

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

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

(δ) $$\frac{\eta, e_1 \Downarrow v_1 \quad \eta, e_2 \Downarrow v_2 \quad v = v_1 \oplus v_2}{\eta, e_1 \oplus e_2 \Downarrow v}$$

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