CS 421 Spring 2012 Final Exam

Wednesday, May 9, 2012

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 14 pages to the exam. Please verify that you have all 14 pages. (Page 14 contains no questions, but has definitions for several problems; you can tear it off for easier reference.)
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	17	
2	10	
3	10	
4	10	
5	4	
6	7	
7	9	
8	7	
9	10	
10	8	
11	8	
Total	**100**	

1. (17 pts) A *usemap* is a dictionary used to keep track of where variables are used; it map variables to lists of integers, representing line numbers in which the given variable is used. In this question, you will write several functions on usemaps, using two representations, a standard list-of-pairs representation and a functional representation. In each case, we will give you the representation, and the definition of `emptyusemap`, and you will have to define three functions:

   - `adduse` adds an additional integer to the list associated with a given variable, e.g.

     ```
     # let m1 = [("a", [4])];;
     # let m2 = adduse "b" 4 m1;;  // returns [("a", [4]); ("b", [4])]
     # let m3 = adduse "a" 5 m2;;  // returns [("a", [5; 4]); ("b", [4])]
     ```

   - `fetch` is a simple fetch operation on these tables:

     ```
     # fetch "a" m3;;  // returns [5; 4]
     ```

   - `removeuses` "zeroes out" the uses of a variable, that is, it replaces the list of uses associated with a given variable by the empty list:

     ```
     # let m4 = removeuses "a" m3;;  // returns [("a", []); ("b", [4])]
     ```

   (a) (6 pts) Define the three functions using the list-of-pairs representation:

   ```
   type usemap = (string * (int list)) list
   let emptyusemap = []

   let rec adduse (x:string) (u:int) (m:usemap) : usemap =
   ```

   ```
   let rec fetch (x:string) (m:usemap) : int list =
   ```

   ```
   let rec removeuses (x:string) (m:usemap) : usemap =
   ```

   (b) (6 pts) Now define the same functions, using the functional representation:

```
type usemap = string -> (int list)
let emptyusemap = fun s -> []

let rec adduse (x:string) (u:int) (m:usemap) : usemap =
```

```
let rec fetch (x:string) (m:usemap) : int list =
```

```
let rec removeuses (x:string) (m:usemap) : usemap =
```

(c) (5 pts) Given this abstract syntax for expressions:

```
    exp = Int of int | Var of string | Add of exp * exp
```

write the function `addexp`: `exp` → `int` → `usemap` → `usemap` which updates a usemap based on the variables in an expression. Specifically, `addexp` *e i m* modifies *m* by adding *i* to the list associated with each variable occurring in *e*. (The idea is that *i* is an integer identifying a statement that contains *e*, and the usemap is keeping track of all the statements where each variable occurs.) For example:

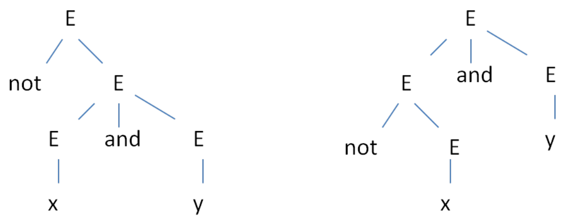
```
let m = [("a", []); ("b", [4])]
addexp (Add(Var "a", Var "b")) 5 m;;  // returns [("a", [5]); ("b", [5; 4])]
```

Don't worry if `addexp` adds the line number more than once. Although we've illustrated it using the list-of-pairs representation, your definition should just use the operations defined in question 1, so it will work for either representation.

```
let rec addexp (e:exp) (n:int) (m:usemap) : usemap =
```

2. (10 pts) Consider this grammar: $E \rightarrow$ `id` | `not` E | E `and` E

The following two trees for the sentence "`not x and y`" shows that the grammar is ambiguous:



(a) (8 pts) Show the shift-reduce parses for these two parse trees side-by-side. (Hint: they have the same number of steps, and are identical until line 4.) We have given the first line; recall that the effect of each action is shown on the following line.

	Parse tree on left			Parse tree on right		
	Action	**Stack**	**Input**	**Action**	**Stack**	**Input**
1.	Shift		`not x and y`	Shift		`not x and y`
2.						
3.						
4.						
5.						
6.						
7.						
8.						
9.	Accept	E	eof	Accept	E	eof

(b) (2 pts) If the tree on the right is the correct one (`not` has precedence over `and`), what action should be taken when `not` is the token closest to the top of the stack and `and` is the lookahead symbol?

3. (10 pts) Consider this grammar:

   $E \rightarrow$ (E) F | `id` F
   $F \rightarrow \epsilon$ | E

   (a) (5 pts) Calculate the FIRST sets for this grammar (remember for both FIRST and FOLLOW sets, column 3 is identical to column 2 - that is how we know the calculation is finished):

	0	1	2	3
E	{}			
F	{}			

   and FOLLOW sets:

	0	1	2	3
E	eof			
F	{}			

   (b) (5 pts) This grammar is LL(1): For the productions from E, the FIRST sets do not overlap; and for the productions from F, FIRST(E) does not overlap with FOLLOW(F). The top-down parser consists of mutually-recursive functions **`parseE`** and **`parseF`**, both of type **`token list`** $\rightarrow$ **`token list`**. Fill in the code for **`parseE`** (with tokens LPAREN, RPAREN, and IDENT); raise a SyntaxError where appropriate.

```
let rec parseE toklis = match toklis with

    LPAREN::toklis' ->











  | IDENT::toklis' ->



and parseF toklis = ... assume this is given ..
```

4. (10 pts) The evaluation rules for a subset of OCaml, in the substitution and environment models, are given on the last page of this exam. You may tear off that sheet for reference.

   We provide the outline of the evaluation of an expression (the same one) in each of the models; you are to fill in the blanks, and, in the parentheses on the left, give the name of the rule being used. We have included the first and last few lines of each evaluation. You may introduce abbreviations for long expressions or environments, but be sure to show this very clearly. The lengths of the blank lines is not significant, but their indentation level is.

   (a) (5 pts) Evaluation in substitution model

```
(App)        (fun a -> ( (fun h -> h a) (fun b -> b+1) ) ) 4 ⇓ 5
(Fun)            fun a -> ((fun h -> h a) (fun b -> b+1))
                     ⇓ fun a -> ((fun h -> h a) (fun b -> b+1))
(Const)          4 ⇓ 4
(App)            (fun h -> h 4) (fun b -> b+1) ⇓ 5
(Fun)                fun h -> h 4 ⇓ fun h -> h 4
(        )           ____________________________________________
(        )           ____________________________________________
(        )               ________________________________________
(        )               ________________________________________
(        )               ________________________________________
(Const)                      4 ⇓ 4
(Const)                      1 ⇓ 1
```

   (b) (5 pts) Evaluation in environment model. (Hint: you can use abbreviations $\rho_1 = \{a \mapsto 4,$ $h \mapsto <$ `(fun b -> b+1)` $, \{a \mapsto 4\} >\}$ and $\rho_2 = \{a \mapsto 4, b \mapsto 4\}$.)

```
(App)        (fun a -> ( (fun h -> h a) (fun b -> b+1) ) ) 4, ∅ ⇓ 5
(Fun)            fun a -> ((fun h -> h a) (fun b -> b+1)), ∅
                     ⇓ <fun a -> ((fun h -> h a) (fun b -> b+1)), ∅ >
(Const)          4, ∅ ⇓ 4
(App)            (fun h -> h a) (fun b -> b+1), {a ↦ 4} ⇓ 5
(Fun)                fun h -> h a, {a ↦ 4} ⇓ <(fun h -> h 4), {a ↦ 4} >
(        )           ____________________________________________
(        )           ____________________________________________
(        )               ________________________________________
(        )               ________________________________________
(        )               ________________________________________
(Var)                        b, ρ2 ⇓ 4
(Const)                      1, ρ2 ⇓ 1
```

5. (4 pts) To add pairs, we add two new expressions: `(`e`,` e'`)` and a pattern-matching version of `let`: `let (`x`,`y`) =` e `in` e'. In this version of `let`, e should evaluate to a value of the form `(`v`,` v'`)`, where v and v' are both values. We give the rule for pairs in each model; give the rules for this version of `let` in each model:

(a) (2 pts) **Substitution model**

(Pair) $(e, e') \Downarrow (v, v')$
$\quad e \Downarrow v$
$\quad e' \Downarrow v'$

(PairLet) `let (`x`,` y`) =` e `in` $e' \Downarrow v$

(b) (2 pts) **Environment model**

(Pair) $(e, e'), \rho \Downarrow (v, v')$
$\quad e, \rho \Downarrow v$
$\quad e', \rho \Downarrow v'$

(PairLet) `let (`x`,` y`) =` e `in` $e', \rho \Downarrow v$

6. (7 pts) To deal with assignable (ref) values in OCaml correctly, we need to adopt the two-level state model that we used for MiniJava. Recall that we first add a type of value called a "location" to the set of values that can appear in the environment (locations are written as ℓ), and then we add a persistent store (a map from locations to values, which we call η) to the state. Evaluations may change the store (not the environment), so it needs to be "threaded" through the evaluation. Therefore, evaluation judgments have the form:

$$e,\ (\rho, \eta)\ \Downarrow\ v,\ \eta'$$

meaning that when e is evaluated in environment ρ and store η, it produces a value v, and changes the store to η'. Here are some of the evaluation rules in this model (we diverge from OCaml only in that an assignment `e:=` e' returns the value of e'):

Two-level state model

(Const) Int i, $(\rho,\eta) \Downarrow$ Int i, η

(Var) a, $(\rho,\eta) \Downarrow \rho(a)$, η

(Assign) $e := e'$, $(\rho,\eta) \Downarrow v$, $\eta''[\ell \mapsto v]$
e, $(\rho,\eta) \Downarrow \ell$, η'
e', $(\rho,\eta') \Downarrow v$, η''

(Ref) ref e, $(\rho,\eta) \Downarrow \ell$, $\eta'[\ell \mapsto v]$
(ℓ a location not used in η')
e, $(\rho,\eta) \Downarrow v$, η'

(Fun) Fun(a,e), $(\rho,\eta) \Downarrow$ <Fun(a,e), ρ >, η

(Deref) !e, $(\rho,\eta) \Downarrow \eta'(\ell)$, η'
e, $(\rho,\eta) \Downarrow \ell$, η'

Give the evaluation rules for application, let, and sequence. In each case, we have filled in the lines for the appropriate number of sub-evaluations. Applications have three sub-evaluations: the function, the argument, and the body of the function, which we've called e''.

(App) $e\ e'$, $(\rho,\eta) \Downarrow v$, η'''

e, ____________________ $\Downarrow$ ____________________

e', ____________________ $\Downarrow$ ____________________

e'', ____________________ $\Downarrow$ ____________________

(Let) let x=e in e', $(\rho,\eta) \Downarrow v'$, η''

e, ____________________ $\Downarrow$ ____________________

e', ____________________ $\Downarrow$ ____________________

(Seq) $e; e'$, $(\rho,\eta) \Downarrow v'$, η''

e, ____________________ $\Downarrow$ ____________________

e', ____________________ $\Downarrow$ ____________________

7. (9 pts) FORTRAN has a control structure called the *arithmetic if*, which branches to one of three statements depending upon whether the value of an integer expression is positive, negative or zero. In this question, you will write a compilation scheme for this control structure.

   Our version of the arithmetic if is: `arithif (`e`)` S_{neg} S_{zero} S_{pos}. It evaluates e and executes one of the three statements, depending on whether e is negative, zero, or positive.

   Recall the basic compilation schemes for statements and expressions:

   (Stmt) $S,\ m \rightsquigarrow il,\ m'$ — S compiles to instruction list il, starting at location m and ending at $m' - 1$.

   (Expr) $e,\ loc \rightsquigarrow il$ — e compiles to instruction list il, which places the value of e in loc.

   You will need the following instructions: `LOADIMM(`*loc*`, `*k*`)` (load k into location loc). `EQUAL(`*tgt*`, `*loc*`1, `*loc*`2)` compares the values in locations $loc1$ and $loc2$ and places the resulting truth value (1 or 0) into location tgt. `LESSTHAN(`*tgt*`, `*loc*`1, `*loc*`2)` is similar to `EQUAL`, but tests whether the contents of $loc1$ are (strictly) less than the contents of $loc2$. `JUMP(`*m*`)` jumps to code location m. `CJUMP(`*loc*`, `*m*`, `*m'*`)` jumps to m if the value in loc is 1, or m' if it is zero.

   The compiled version of `arithif` starts with e, and then has five instructions to jump to the correct one of the three statements. We have filled in the first one and then left four blanks. The instructions you insert will need to store values in temporary locations; you can freely use names `t1`, `t2`, etc. for temporary locations.

   `arithif (`e`)` S_{neg} S_{zero} S_{pos}, $m \rightsquigarrow$

   il @ [LOADIMM t1,0; ______________________________;

   ______________________________;

   ______________________________;

   ______________________________]

   @ $il1$ @ [______________________________]

   @ $il2$ @ [______________________________] @ $il3$, m'''

   $e,\ loc \rightsquigarrow il$

   S_{neg}, ____________________ $\rightsquigarrow il1,\ m'$

   S_{zero}, ____________________ $\rightsquigarrow il2,\ m''$

   S_{pos}, ____________________ $\rightsquigarrow il3,\ m'''$

8. (7 pts) Fill in the virtual function tables (v-tables) for each of the following classes, using the format shown for the first one. Remember that the order of functions in a v-table is important.

```
class A {
String f() { ... }
}
```

f in A

```
class B extends A {
double g() { ... }
}
```


```
class C extends B {
double g() { ... }
String f() { ... }
}
```


```
class D extends C {
String f() { ... }
String h() { ... }
}
```


9. (10 pts) This question concerns the higher-order library function `fold_right`:

```
let rec fold_right f lis z = if lis=[] then z else f (hd lis) (fold_right f (tl lis) z)
```

which has type $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha\, list \rightarrow \beta \rightarrow \beta$.

Recall that `zip` takes two lists of the same length, and returns a list of pairs of the corresponding elements from the two lists, e.g. `zip ["a";"b";"c"] [1;2;3]` = `[("a",1);("b",2);("c",3)]`.

(a) (2 pts) The function `pos_names`: string list * int list → string list selects from its first argument those strings where the corresponding integer in the second argument is non-negative:

```
pos_names ["a";"b";"c"] [1;-2;3]    returns: ["a"; "c"]
```

Fill in the second argument of `fold_right` to get a definition of `pos_names`:

```
let pos_names lis1 lis2 = let zipped = zip lis1 lis2 in

   fold_right (__________________________________________________)  zipped []
```

(b) (2 pts) Similarly, write `sum_of_diffs`, which sums the differences of corresponding elements in two lists: `sum_of_diffs [10; 20; 30] [4;5;7]` returns 44 (6 + 15 + 23).

```
let sum_of_diffs lis1 lis2 = let zipped = zip lis1 lis2 in

   fold_right (__________________________________________________)  zipped 0
```

(c) (2 pts) We can also solve this by giving a two-list version of `fold_right`:

```
let rec fold_right2 f lis1 lis2 z =
    if lis1=[] then z
          else f (hd lis1) (hd lis2) (fold_right2 f (tl lis1) (tl lis2) z)
```

Give the type of `fold_right2`:

(d) (4 pts) Define `pos_names` and `sum_of_diffs` using `fold_right2` (without zipping the arguments):

```
let pos_names lis1 lis2 =

   fold_right2 (__________________________________________________)  lis1 lis2 []

let sum_of_diffs lis1 lis2 =

   fold_right2 (__________________________________________________)  lis1 lis2 0
```

10. (8 pts) The explicitly-typed polymorphic type system for OCaml is given on the last page of this exam. You may tear off that sheet so you can refer to it more easily.

    (a) (6 pts) Give the complete proof of the following judgment. We have filled in the first few lines. (The lengths of the lines are not significant, but their indentation level is.) In the parentheses on the left, enter the name of the rule used to prove that judgment. (*Hint:* You will eventually need type environment $\{g : \forall\alpha.\,(int \to \alpha) \to \alpha\}$, which you can abbreviate as Γ_1.)

```
(Let)          ∅ ⊢ let g:((int->alpha)->alpha) = fun f:(int->alpha) -> f 0 : int
                       in g[(int->int)->int] (fun x:int -> x+1) : int
(Fun)              ∅ ⊢ fun f:(int->alpha) -> f 0 : (int → α) → α
(App)                  {f : int → α} ⊢ f 0 : α
(Var)                      {f : int → α} ⊢ f : int → α
(Const)                    {f : int → α} ⊢ 0 : int

(__________)       ____________________________________________
(__________)           ____________________________________________
(__________)           ____________________________________________
(__________)               ____________________________________________
(__________)                   ____________________________________________
(__________)                   ____________________________________________
```

    (b) (2 pts) In question 5, we introduced a let expression that does pattern-matching for pairs: `let (`x`,`y`) = `e` in `e'. We have given the typing rule for pairs; give the typing rule for let expressions:

(Pair) $\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2$

$\Gamma \vdash e_1 : \tau_1$

$\Gamma \vdash e_2 : \tau_2$

(PairLet) $\Gamma \vdash$ `let (`x`,`y`) = `e` in `e' $: \tau$

____

____

11. (8 pts) Give loop invariants and termination functions for the following loops. (You do not have to prove anything.)

    (a) (4 pts)

    $y > 0 \wedge e = y \wedge p = 1$ {
    ```
    while(e > 0) { p = p * x; e = e - 1; }
    ```
    } $p = x^y$

    Invariant:

    Termination function:

    (b) (4 pts) Here, a is an n–element array:

    $max = 0 \wedge i = 1 \wedge n > 1$ {
    ```
    while(i != n) { if (a[i] > a[max]) max = i;
                    i = i+1; }
    ```
    } $\forall 0 \leq j < n.\, a[max] \geq a[j]$

    Invariant:

    Termination function:

Substitution model

(Const) Int x $\Downarrow$ Int x

(Fun) Fun(a,e) $\Downarrow$ Fun(a,e)

(δ) e *op* $e' \Downarrow v$ *OP* v'
 $e \Downarrow v$
 $e' \Downarrow v'$

(App) $e\ e' \Downarrow v$
 $e \Downarrow$ Fun(a, e'')
 $e' \Downarrow v'$
 $e''[v'/a] \Downarrow v$

Environment model

(Const) Int i, $\rho \Downarrow$ Int i

(Var) a, $\rho \Downarrow \rho(a)$

(δ) e *op* e', $\rho \Downarrow v$ *OP* v'
 e, $\rho \Downarrow v$
 e', $\rho \Downarrow v'$

(App) $e\ e'$, $\rho \Downarrow v$
 e, $\rho \Downarrow$ <Fun(a, e''), ρ' >
 e', $\rho \Downarrow v'$
 e'', $\rho'[a \mapsto v'] \Downarrow v$

(Fun) Fun(a,e), $\rho \Downarrow$ <Fun(a,e), ρ >

Explicitly-typed, polymorphic type system

(Const) $\Gamma \vdash$ Int i : int

(Var) $\Gamma \vdash a : \Gamma(a)$
 ($\Gamma(a)$ a type)

(Fun) $\Gamma \vdash$ fun a:τ `->` $e : \tau \rightarrow \tau'$
 $\Gamma[a{:}\tau] \vdash e : \tau'$

(δ) $\Gamma \vdash e \oplus e' : \tau''$
 $\Gamma \vdash e : \tau$
 $\Gamma \vdash e' : \tau'$

(App) $\Gamma \vdash e\ e' : \tau'$
 $\Gamma \vdash e : \tau \rightarrow \tau'$
 $\Gamma \vdash e' : \tau$

(True) $\Gamma \vdash$ true : bool

(False) $\Gamma \vdash$ false : bool

(PolyVar) $\Gamma \vdash a[\tau] : \tau$
 where $\tau \leq \Gamma(a)$
 ($\Gamma(a)$ a type scheme)

(Let) $\Gamma \vdash$ let a:τ `=` e `in` $e' : \tau'$
 $\Gamma \vdash e : \tau$
 $\Gamma[a{:}\mathrm{GEN}_\Gamma(\tau)] \vdash e' : \tau'$