CS 421 Spring 2012 Midterm 2

Tuesday, April 10, 2012

Name	
NetID	

- You have **70 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 eight pages to the exam. Please verify that you have all eight pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1a	15	
1b	15	
1c/d	15	
1e	10	
2	15	
3	15	
4	15	
Total	**100**	

1. This question is in several parts, all based on the following subset of OCaml, which we call MicroOcaml, or μOCaml:

```
type exp = Int of int | Var of string | App of exp * exp
         | Fun of string * exp | Binop of exp * binop * exp
```

Here are the evaluation rules for μOCaml, in the 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'')
$e' \Downarrow v'$
$e''[v'/a] \Downarrow v$

(a) (15 pts) Assume you are given function `subst: string → exp → exp → exp` (`subst` $x\ e\ e'$ $= e'[e/x]$), and `applyOp : binop → exp → exp → exp`. Write `reduce` for μOCaml:

```
reduce e = match e with

   Int i -> e


|  Fun(a,e) -> Fun(a,e)



|  Binop(e1,bop,e2) -> applyOp op (reduce e1) (reduce e2)




|  App(e1, e2) ->
      (match reduce e1 with
           Fun(x, e) -> reduce (subst x e2 e)
         | _ -> raise (TypeError "applying non-function"))
```

(b) (15 pts) Here are the rules for the environment-based evaluator:

(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 >

Assume you are given definitions for type `environment` and functions `fetch`: `string` → `environment` → `exp` and `extend`: `string` → `exp` → `environment` → `environment`. Also assume the abstract syntax has the additional constructor:

```
| Closure of exp * environment
```

Write `eval` for μOCaml (note that there is no `Rec` constructor in μOCaml, so no need to handle that case):

```
eval e env = match e with

   Int i -> e


|  Var a -> fetch a env



|  Fun(a,e) ->  Closure(e, env)



|  Binop(e1,bop,e2) ->  applyOp op (eval e1 env) (eval e2 env)



|  App(e1, e2) -> (match (eval e1 env) with
        Closure(Fun(x, e), env') -> eval e (extend x (eval e2 env) env')
     | _ -> raise (TypeError "applying non-function"))
```

(c) (8 pts) Suppose we add two new abstract syntax constructors:

```
| Fun2 of string * string * exp | Pair of exp * exp
```

`Fun2` represents a very simple form of pattern matching for pairs, and `Pair` is like `Tuple` from MP8, but only for constructing pairs. A function of the form "`fun (x,y) -> e`" would translate to abstract syntax Fun2(x,y,e), and a pair (e, e') would translate to Pair(e, e'). A Fun2 is applicable only to values of the form Pair(v, v').

We need three new rules for evaluation with pairs: two for the new abstract syntax operators, plus a new rule for `App` when the function is a `Fun2`. Fill in those rules for the substitution-based evaluator:

(Fun2) Fun2(a, b, e) $\Downarrow$ **Fun2(**a**,** b**,** e**)**

(Pair) Pair(e_1,e_2) $\Downarrow$ **Pair(**v_1**,** v_2**)**
$e_1 \Downarrow v_1$
$e_2 \Downarrow v_2$

(App2) $e\ e' \Downarrow v$
$e \Downarrow$ **Fun2(**a**,** b**,** e''**)**
$e' \Downarrow$ **Pair(**v_1**,** v_2**)**
$e''[v_1/a][v_2/b] \Downarrow v$

(d) (7 pts) Fill in those rules for the environment-based evaluator:

(Fun2) Fun2(a, b, e), $\rho \Downarrow <$**Fun2(**a**,** b**,** e**),** $\rho>$

(Pair) Pair(e_1,e_2), $\rho \Downarrow$ **Pair(**v_1**,**v_2**)**
e_1**,** $\rho \Downarrow v_1$
e_2**,** $\rho \Downarrow v_2$

(App2) $e\ e'$, $\rho \Downarrow v$
e**,** $\rho \Downarrow <$**Fun2(**a**,** b**,** e''**),** $\rho'>$
e'**,** $\rho \Downarrow$ **Pair(**v_1**,** v_2**)**
e''**,** $\rho'[a \mapsto v_1][b \mapsto v_2] \Downarrow v$

(e) (10 pts) Type judgments for μOCaml expressions, like those for MiniJava expressions, have the form $\Gamma \vdash e : \tau$, where Γ is a type environment giving types for the variables occurring free in e. Read this judgment as "e has type τ, if the variables occurring in e have the types given by Γ." Types are either "`int`" or a function from type τ to τ', written $\tau \to \tau'$. Here are the type rules for μOCaml, where we have included `Let` as well:

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

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

(Fun) $\Gamma \vdash \text{Fun}(a,e) : \tau \to \tau'$ if $\Gamma[a{:}\tau] \vdash e : \tau'$

(δ) $\Gamma \vdash e \; op \; e' :$ int if $\Gamma \vdash e :$ int and $\Gamma \vdash e' :$ int

(App) $\Gamma \vdash e \; e' : \tau'$ if $\Gamma \vdash e : \tau \to \tau'$ and $\Gamma \vdash e' : \tau$

(*Let*) $\Gamma \vdash \text{Let}(x, e, e') : \tau'$ if $\Gamma \vdash e : \tau$ and $\Gamma[x{:}\tau] \vdash e' : \tau'$

i. (5 pts) Here is a partial proof that `let x=3 in let y=x+1 in y*x` has type int. Fill in the missing lines in the proof, being sure to indent appropriately, and write in the blank lines the name of the rule used in every line. (Hint: it is just based on the abstract syntax constructor for the expression.)

Let	{} ⊢ `let x=3 in let y=x+1 in y*x` : int
Const	{} ⊢ `3` : int
Let	{x:int} ⊢ `let y=x+1 in y*x` : int
δ	{x:int} ⊢ `x+1` : int
Var	{x:int} ⊢ `x` : int
Const	{x:int} ⊢ `1` : int
δ	{x:int, y:int} ⊢ `y*x` : int
Var	{x:int, y:int} ⊢ `y` : int
Var	{x:int, y:int} ⊢ `x` : int

ii. (5 pts) To include pairs, we expand the set of types to include pairs, written $\tau^*\tau'$. Give type rules for the expressions involving pairs:

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

(Fun2) $\Gamma \vdash \text{Fun2(x,y,e)} : \tau_1 * \tau_2 \to \tau$ if $\Gamma[\text{x}{:}\tau_1][\text{y}{:}\tau_2] \vdash \text{e} : \tau$

2. (15 pts) (Higher-order functions) This problem uses a variant of the functional definition of environments from MP8. The abstract syntax above is extended by adding the constructor `Missing`:

```
type exp = Int of int | Var of string |  ... | Missing
```

(The precise constructors in `exp` don't matter for this problem.)

Define environments as functions:

```
type environment = string -> exp
```

Unlike in MP8, if you look up a variable in an environment that doesn't have that variable, it returns value `Missing`:

```
let emptyEnv = fun s -> Missing
```

(a) (7 pts) Define fetch and extend:

```
let fetch (s:string) (env:environment) : exp = env s
```

```
let extend (s:string) (e:exp) (env:environment) : environment =
      fun s' -> if s'=s then e else env s'
```

(b) (4 pts) Define `retract`, which *removes* a binding for a variable (so if we fetch `a` from `retract a env`, it will return `Missing`):

```
let retract (s:string) (env:environment) : environment =
      fun s' -> if s'=s then Missing else env s'
```

(c) (4 pts) Define `combine`, which combines two environments, giving precedence to the first for any names that are defined in both. That is, if `env = combine env1 env2`, then if we fetch variable `a` from `env`, it will return `Missing` if `a` is not defined in either `env1` or `env2`; the value from `env1` if it is defined only in `env1`; the value from `env2` if it is defined only in `env2`; and the value from `env1` if it is defined in both.

```
let combine (env1:environment) (env2:environment) : environment =
      fun s -> if env1 s <> Missing then env1 s else env2 s
```

3. (15 pts) For each of the following Java class definitions, fill in its "v-table" (virtual function table). Each entry should have the form "<function name> in <class name>", meaning this table entry points to the definition of <function name> given in <class name>. The functions in each table should appear in the correct order, as they would in a v-table for Java or C++. We have given the first one.

```
class B {
   void f() {}
   void g() {}
}
```

f in B
g in B

```
class C1 extends B {
   void h() {}
}
```

f in B
g in B
h in C1

```
class C2 extends B {
   void g() {}
}
```

f in B
g in C2

```
class D extends C1 {
   void i() {}
   void g() {}
}
```

f in B
g in D
h in C1
i in D

4. (15 pts) (MiniJava compilation) This is the compilation scheme for while statements in MiniJava (from MP 7 extra credit):

   while (e) S, $m \rightsquigarrow$ [JUMP m'] @ ils @ ile @ [CJUMP loc,$m+1$, m''], m''
   (where $m'' = m' + |ile| + 1$)
   S, $m+1 \rightsquigarrow ils$, m'
   e, loc $\rightsquigarrow ile$

   Note that it uses the non-short-circuit compilation scheme for the condition, e.

   (a) (10 pts) Give a compilation rule for the statement

   whilebreak { S_1 } (*cond*) { S_2 }

   which works like this: execute S_1, then test the condition; if the condition is false, terminate the entire whilebreak statement; if it is true, then execute S_2 and S_1 and test the condition again; and repeat. We have filled in some of it for you. (You do not need any more machine instructions than what are shown in the example above.)

   whilebreak $\{S_1\}$ (e) $\{S_2\}$, $m \rightsquigarrow$
   $il1$ @ il @ [CJUMP loc,$m''+1$,$m'+|il|+1$] @ il2 @ [JMP m], $m''+1$
   S_1, $m \rightsquigarrow il1$, m'
   e, loc $\rightsquigarrow il$
   S_2, $m'+|il|+1 \rightsquigarrow il2$, m''

   (b) (5 pts) Recall that in short-circuit evaluation, the compilation judgment for boolean expressions has the form:

   e, m, t, $f \rightsquigarrow_2 il$, m'

   which means that il is a code sequence that, when executed, will jump to t if e is true, and jump to f if e is false; furthermore, the code sequence starts at location m and ends at location $m'-1$.
   Give a compilation rule for the `whilebreak` statement using short-circuit evaluation.

   whilebreak S_1 (e) S_2, $m \rightsquigarrow il1$ @ il @ $il2$ @ [JMP m], $m'''+1$
   S_1, $m \rightsquigarrow il1$, m'
   e, m', m'', $m''' \rightsquigarrow_2 il$, m''
   S_2, $m'' \rightsquigarrow il2$, m'''