

CMSC330 Fall 2009 Midterm #2 Solutions

1. (6 pts) Programming Languages

- a. (3 pts) Describe one design choice for *type declarations* for static types in a programming language, and list a programming language using this approach.

Manifest (explicit) – type specified in variable declaration (Java, C, Java)

Inferred (implicit) – not specified, determined by compiler from usage (OCaml)

- b. (3 pts) How can programmers write Java programs which (effectively) pass functions as arguments to other functions? Give a brief answer.

Pass an object implementing an interface with a known method.

2. (8 pts) Scoping

Consider the following OCaml code.

```
let app f y = let y = 5 in let x = 7 in let a = 9 in f y ;;  
let add x y = let incr a = a+y in app incr x ;;  
(add 2 4) ;;
```

- a. (2 pts) List the order the functions *add*, *incr*, and *app* are invoked in (add 2 4)
add, app, incr
- b. (3 pts) What value is returned by (add 2 4) with static scoping? Explain.
9, since for incr a = a+y: a=5 (let y = 5...in f y) and y=4 (add x y = ...) (add 2 4).
- c. (3 pts) What value is returned by (add 2 4) with dynamic scoping? Explain.
10, since for incr a = a+y: a=5 (let y = 5...in f y) and y=5 (let y = 5...in f y).

Explanation: The return value is determined by incr, when it is invoked as (f y) by app.

The value of incr a is a+y. The formal parameter a is bound to the value of the argument passed to incr. In app when (f y) is invoked, the argument y always has the value of 5, since the closest definition of y (both static & dynamic) is the let y = 5 in app. So a=5.

The value of y in a+y is more tricky. Within the body of incr, y is a free variable. So when incr is invoked, the value of y is determined based on the type of scoping.

For static scoping, y is bound to the closest lexical binding of y where it appears in the body of incr (i.e., let add x **y = let incr...). At runtime this is the 2nd argument to add, which for (add 2 4) is 4. So y=4, and a+y=9.**

For dynamic scoping, y is bound to the closest dynamic binding of y when incr is actually evaluated. Since incr is called after app, the closest binding of y is the binding in app (i.e., let y = 5). So y=5, and a+y=10.

3. (12 pts) Parsing

a. (5 pts) Compute First sets for S and A in the following grammar:

$$\begin{array}{ll} S \rightarrow Acdc & A \rightarrow bgS \\ S \rightarrow aAf & A \rightarrow \varepsilon \quad (* \text{ epsilon } *) \end{array}$$

$$\mathbf{First(S) = \{ a, b, c \}, First(A) = \{ b, \varepsilon \}}$$

b. (3 pts) Apply the algorithm discussed in class to transform the following grammar so that it can be parsed using a recursive descent parser.

$$\begin{array}{l} S \rightarrow Sb \\ S \rightarrow ac \end{array}$$

$$\begin{array}{l} S \rightarrow \mathbf{acL} \\ L \rightarrow \mathbf{bL} \mid \varepsilon \end{array}$$

c. (4 pts) Recursive Descent Parsing

Using pseudocode, write a recursive descent parser for the following grammar.

$$\begin{array}{l} S \rightarrow cbS \\ S \rightarrow \varepsilon \quad (* \text{ epsilon } *) \end{array}$$

```
parse_S() {  
    if (lookahead == "c") {  
        match("c"); match("b"); parse_S();  
    } else {  
        ;  
    }  
}
```

4. (16 pts) Lambda calculus

Evaluate the following λ -expressions as much as possible. Show each beta-reduction

a. (3 pts) $(\lambda x. \lambda y. y x) a (\lambda z. z) b$

$(\lambda x. \lambda y. y x) a (\lambda z. z) b$ // β -reduction: $x \rightarrow a$
 $\Rightarrow (\lambda y. y a) (\lambda z. z) b$ // β -reduction: $y \rightarrow (\lambda z. z)$
 $\Rightarrow ((\lambda z. z) a) b$ // β -reduction: $z \rightarrow a$
 $\Rightarrow a b$

b. (3 pts) $(\lambda y. \lambda x. x y) x a b$

$(\lambda y. \lambda x. x y) x a b$ // α -conversion: replace x with z
 $\Rightarrow (\lambda y. \lambda z. z y) x a b$ // β -reduction: $y \rightarrow x$
 $\Rightarrow (\lambda z. z x) a b$ // β -reduction: $z \rightarrow a$
 $\Rightarrow a x b$

c. (10 pts) Using encodings, show $2^*1 \Rightarrow^* 2$. Show each beta-reduction.

\Rightarrow^* indicates 0 or more steps of beta-reduction

You may assume $\lambda f. \lambda y. f (f y) \Rightarrow 2$

$M * N = \lambda x. (M (N x))$
 $1 = \lambda f. \lambda y. f y$
 $2 = \lambda f. \lambda y. f (f y)$
 $3 = \lambda f. \lambda y. f (f (f y))$
 $4 = \lambda f. \lambda y. f (f (f (f y)))$

$2^*1 \Rightarrow$

$\Rightarrow \lambda x. (2 (1 x))$ // replacing * w/ encoding
 $\Rightarrow \lambda x. (2 ((\lambda f. \lambda y. f y) x))$ // replacing 1 w/ encoding
 $\Rightarrow \lambda x. (2 (\lambda y. x y))$ // β -reduction: $1^{\text{st}} f \rightarrow x$
 $\Rightarrow \lambda x. ((\lambda f. \lambda y. f (f y)) (\lambda y. x y))$ // replacing 2 w/ encoding
 $\Rightarrow \lambda x. (\lambda y. (\lambda y. x y) ((\lambda y. x y) y))$ // β -reduction: $1^{\text{st}} f$ w/ $\lambda y. x y$
 $\Rightarrow \lambda x. (\lambda y. (\lambda y. x y) (x y))$ // β -reduction: $3^{\text{rd}} y \rightarrow y$
 $\Rightarrow \lambda x. \lambda y. (x (x y))$ // β -reduction: $2^{\text{nd}} y \rightarrow x y$
 $\Rightarrow \lambda f. \lambda y. f (f y)$ // α -conversion: replace x with f
 $\Rightarrow 2$ // i.e., is encoding for 2

5. (8 pts) OCaml Types and Type Inference

a. (2 pts) Give the value of the following OCaml expression. If an error exists, describe it.

let x y = y in 3 **Value = 3**

b. (3 pts) Give the type of the following OCaml expression

fun y -> [y 1] **Type = (int -> 'a) -> 'a list**

c. (3 pts) Write an OCaml expression with the following type

bool -> bool -> int **Code = (fun x y -> if (x && y) then 1 else 2)**
 OR = let f x y = if (x && y) then 1 else 2

6. (50 pts) OCaml Programming

a. (4 pts) Implement *lookup*

```
let rec lookup id lst = match lst with
  [] -> raise NotFound
  | (a,b)::t -> if (id=a) then b else (lookup id t)
```

b. (6 pts) Implement *remove* using fold & anonymous function

```
let remove id lst = fold (fun a (x,v) -> if (x=id) then a else ((x,v)::a)) [] lst
```

```
let rec remove id lst = match lst with // OR for partial credit
```

```
  [] -> []
  | (a,b)::t -> if (id=a) then t else (a,b)::(remove id t)
```

c. (14 pts) Implement *eval*

```
let rec eval st exp = match exp with
```

```
  Id str -> let n = (lookup str st) in (n, st)
  | Define (str, e) -> let (v, st2) = (eval st e) in (v, (str,v)::(remove str st2))
  | Num n -> (Val_num n, st)
```

d. (26 pts) Implement *eval* supporting updatable references

```
let rec eval st exp =
```

```
  match st with (bindings, memory, next_mem) ->
  match exp = match exp with
```

```
  ...
```

```
  | Ref e -> let (v, (b2,m2,n2)) = (eval st e) in
    ((Val_ptr n2), (b2, (n2,v)::m2, (n2+1)))
```

```
  | Deref e -> let (v,(b2,m2,n2)) = (eval st e) in
    ( match v with
```

```
      Val_num _ -> raise IllegalDeref
```

```
      | Val_ptr a -> let v2 = (lookup a m2) in (v2, (b2, m2, n2))
```

```
    )
```

```
  | Assign (lhs, rhs) -> ( let (lhs_v, (b2,m2,n2)) = (eval st lhs) in
    let (rhs_v, (b3,m3,n3)) = (eval (b2,m2,n2) rhs) in
```

```
      ( match lhs_v with
```

```
        Val_num _ -> raise IllegalDeref
```

```
        | Val_ptr a -> (rhs_v, (b3, (a,rhs_v)::(remove a m3), n3))
```

```
      )
```

```
    )
```

Note how whenever *eval* is called, it returns a pair (value, new state). The new state must be used the next time *eval* is called, and the most recent state returned by *eval*.