

CMSC330 Fall 2009 Midterm #2

Name _____

Discussion Time (circle one): 10am 11am 12pm 1pm 2pm 3pm

Do not start this exam until you are told to do so!

Instructions

- You have 75 minutes to take this midterm.
- This exam has a total of 100 points, so allocate 45 seconds for each point.
- This is a closed book exam. No notes or other aids are allowed.
- If you have a question, please raise your hand and wait for the instructor.
- Answer essay questions concisely using 2-3 sentences. Longer answers are not necessary and a penalty may be applied.
- In order to be eligible for partial credit, show all of your work and clearly indicate your answers.
- Write neatly. Credit cannot be given for illegible answers.

	Problem	Score
1	Programming Languages	/6
2	Scoping	/8
3	Parsing	/12
4	Lambda Calculus	/16
5	OCaml Types & Type Inference	/8
6	OCaml Programming	/50
	Total	/100

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.

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

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)`

- b. (3 pts) What value is returned by `(add 2 4)` with static scoping? Explain.

- c. (3 pts) What value is returned by `(add 2 4)` with dynamic scoping? Explain.

3. (12 pts) Parsing

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

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

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.

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

c. (4 pts) Recursive Descent Parsing

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

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

Use the following utilities:

lookahead	Variable holding next terminal Lookahead == "\$" when at end of input
match (x)	Function to match next terminal to x
error ()	Reports parse error for input

parse_S() {

4. (16 pts) Lambda calculus

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

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

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

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 $\lambda f. \lambda y. f (f y) \Rightarrow 2$

$2 * 1 \Rightarrow$

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

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 =**

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

fun y -> [y 1] **Type =**

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

bool -> bool -> int **Code =**

6. (50 pts) OCaml Programming

In this problem, you will implement an interpreter for some features of a small programming language with integers and updatable references (which support OCaml's `ref`, `!`, and `:=` operators). You are not allowed to use any OCaml library functions in your solutions.

First you need to design an abstraction for storing bindings between symbols and their values in the top-level environment. You decide to implement the environment as a list of pairs which can be accessed using the two functions *lookup* and *remove*.

a. (4 pts) Implement *lookup*

The function *lookup* has type: `'a -> ('a * 'b) list -> 'b`

Given a key x with type `'a` and a list of pairs lst with type `('a * 'b) list`, invoking *lookup* x lst returns either y (where y is the value associated with x in lst), or *lookup* throws the exception *NotFound* (remember that OCaml syntax for throwing exceptions is `raise <ExceptionName>`). You do not need to catch the exception. You may assume there will be at most one value associated with x . Use `=` to compare keys. For example:

```
lookup "bar" [ ("foo", 1) ; ("bar", 2) ]      (* returns 2 *)
lookup 4 [ (3,"foo") ; (4,"bar") ; (4,"baz") ] (* returns "bar" *)
lookup 5 [ (3,"foo") ; (4,"bar") ]           (* raise NotFound *)
```

b. (6 pts) Implement *remove*

The function *remove* has type: `'a -> ('a * 'b) list -> ('a * 'b) list`

Given a key x with type `'a` and a list of pairs lst with type `('a * 'b) list`, invoking *remove* x lst returns a new list where *all* bindings associated with x have been removed. Use `=` to compare keys. The order of bindings in the new list does not matter.

Implement *remove* using `fold` and an anonymous function.

let rec fold f a l = match l with [] -> a (h::t) -> fold f (f a h) t
--

Examples:

```
remove "bar" [ ("foo", 1) ; ("bar", 2) ]      (* returns [ ("foo", 1) ] *)
remove 4 [ (3,"foo") ; (4,"bar") ; (4,"baz") ] (* returns [ (3,"foo") ] *)
```

c. (14 pts) Implement *eval*

The function *eval* has type: `state -> expr -> (value, state)`

Now we can begin implementing the interpreter, starting with just integers and variable bindings. Abstract syntax trees representing expressions in this language are given by the OCaml datatype *expr*:

<pre>type expr = Id of string Define of string * expr Num of int</pre>	<pre>type value = Val_num of int</pre>	<pre>type state = (string * value) list</pre>
--	--	---

Values in this language are given by type *value*. For now the only values are integers.

We will represent the state of the program as type *state*. For now the only state is the list of bindings in the top-level environment. The initial environment is [].

The semantics of these expressions are:

- This language has no closures and no local environments, just a global, top-level list of bindings. `Id` and `Define` behave just like the top-level environment in Scheme.
- The expression `(Id "foo")` looks up the identifier `foo` in the global, top-level environment and returns its value. If there is no such identifier, then it throws the exception `NotFound`.
- The expression `(Define ("foo", e))` adds or replaces the binding of `foo` in the top-level environment with whatever `e` evaluates to, and returns the value of `e`. You must ensure *at most one binding exists* for each identifier in the top-level environment.

Example Code	Returns
<code>let st0 = [] ;;</code>	<code>(* initial state is [] *)</code>
<code>let (v,st1) = eval st0 (Num 1) ;;</code>	<code>(Val_num 1, [])</code>
<code>let (v,st2) = eval st1 (Define ("foo", (Num 2))) ;;</code>	<code>(Val_num 2, [("foo", Val_num 2)])</code>
<code>let (v,st3) = eval st2 (Define ("foo", (Num 3))) ;;</code>	<code>(Val_num 3, [("foo", Val_num 3)])</code>
<code>let (v,st4) = eval st3 (Id "foo") ;;</code>	<code>(Val_num 3, [("foo", Val_num 3)])</code>

let rec eval st exp =

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

The function *eval* has type: `state -> expr -> (value, state)`

Now we can extend the interpreter to support updatable references. Updatable references require adding a *memory* to the program state. We can add a memory easily by treating it as a binding of addresses (ints) to values. We add additional types to *expr* and *value* as follows:

type <i>expr</i> = Id of string Define of string * <i>expr</i> Num of int Ref of <i>expr</i> Deref of <i>expr</i> Assign of <i>expr</i> * <i>expr</i>	type <i>value</i> = Val_num of int Val_ptr of int	type <i>state</i> = ((string * <i>value</i>) list, (int * <i>value</i>) list, int)
---	---	--

The semantics of these expressions are:

- The expression (Ref *e*) "allocates" a fresh (i.e., new & unused) memory cell at address *n*, binds address *n* to the value *e* evaluates to, and returns a pointer *p* to address *n*. You must ensure that *at most one binding exists* for each address.
- The expression (Deref *e*) evaluates *e* to produce a pointer *p*, and then returns the value at the address pointed to by *p*. If *e* does not have value Val_ptr *p*, throw the exception IllegalDeref.
- The expression (Assign (*e*₁, *e*₂)) evaluates *e*₁ to produce a pointer *p*, and updates *p*'s value in memory to be whatever value *e*₂ evaluates to, then returns the value of *e*₂. If *e*₁ is not a pointer, throw the exception IllegalDeref.

With these changes, values can now be either integers or pointers.

- Val_num *n* represents the integer *n*
- Val_ptr *p* represents a pointer to the address *p*. (We could use any denumerable set as addresses, but here we've chosen to use integers. This is not C, though – there is no way to cast an integer to a pointer or vice versa)

Now the state of the program is a tuple (*d*, *m*, *n*) where

- *d* is the list of bindings in the top-level environment created with Define
- *m* is the memory, binding addresses (ints) to values
- *n* is the next available memory location

Initially, the state of the program starts as ([], [], 0). Defines add to *d*. Refs add to *m* and increment *n*. Since at most one binding exists for each identifier in *d*, and at most one binding for each address in *m*, the order of pairs in the list of bindings & list of memory values does not matter.

Note: For this part of the problem, you may assume that the cases for Id, Define, and Num have already been implemented properly for the extended interpreter. Your interpreter only needs to handle the cases for Ref, Deref, and Assign.

Example Code

```
let st0 = ([],[],0) ;;  
let (v,st1) = eval st0 (Define ("foo", Ref (Num 5))) ;;  
let (v,st2) = eval st1 (Deref (Id "foo")) ;;  
let (v,st3) = eval st2 (Assign (Id "foo", Num 6)) ;;  
let (v,st4) = eval st3 (Define ("bar", Ref (Id "foo"))) ;;
```

Returns

```
(* initial state is ([],[],0) *)  
(Val_num 5, ([("foo", Val_ptr 0)],[(0, Val_num 5)],1)  
(Val_num 5, ([("foo", Val_ptr 0)],[(0, Val_num 5)],1)  
(Val_num 6, ([("foo", Val_ptr 0)],[(0, Val_num 6)],1)  
(Val_ptr 1, ([("bar", Val_ptr 1);("foo", Val_ptr 0)],  
             [(1, Val_ptr 0); (0, Val_num 6)],2)
```

let rec eval st exp =