

<b>Name (PRINTED):</b>	_____		
<b>University ID #:</b>	_____		
<b>Circle your TA's name:</b>	David	Brandon	
<b>Circle your discussion time:</b>	10:00	11:00	12:00

CMSC 330

Exam #2

Fall 2008

**Do not open this exam until you are told. Read these instructions:**

1. This is a closed book exam. **No notes or other aids are allowed.**
2. **You must turn in your exam immediately when time is called at the end.**
3. This exam contains 9 pages, including this one. **Make sure you have all the pages.** Each question's point value is next to its number. **Write your name on the top of all pages before starting the exam.**
4. In order to be eligible for as much partial credit as possible, show all of your work for each problem, and **clearly indicate** your answers. Credit **cannot** be given for illegible answers.
5. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.
6. If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.
7. If you need scratch paper during the exam, please raise your hand. Scratch paper must be turned in with your exam, with your name and ID number written on it. Scratch paper **will not** be graded.
8. Small syntax errors will be ignored in any code you have to write on this exam, as long as the concepts are correct.
9. The Campus Senate has adopted a policy asking students to include the following handwritten statement on each examination and assignment in every course: "*I pledge on my honor that I have not given or received any unauthorized assistance on this examination.*" Therefore, **just before turning in your exam**, you are requested to write this pledge **in full** and **sign it** below:

---



---

Good luck!

1	2	3	4	Total

1. [25 pts.] **Short Answer.**

- a. [4 pts.] List the free variables of the following OCaml function
- `f`
- :

```
let f a y =  
  let g z = a + b + z in  
  let w = (g a) * z in  
  w + b
```

**Answer:** `z, b`

- b. [4 pts.] Given an expression
- `e1; e2`
- in OCaml, the OCaml compiler will issue a warning if
- `e1`
- does not have type
- `unit`
- . Explain the rationale behind this warning.

**Answer:** The type `unit` (and corresponding value `()`) indicate an uninteresting result. If `e1` produces a non-`unit` value, that suggests the result may be meaningful. But `e1; e2` throws away the result of computing `e1`. Hence if the compiler sees that `e1` produces a meaningful result that is then discarded, it suggests you might have some bug in your program, because you're performing computation and then throwing away the result.

- c. [4 pts.] In class, I said that everyone basically agrees it is a bad idea to use call-by-name as a parameter passing mechanism in an imperative language. Explain why.

**Answer:** Under call-by-name, it is very hard to predict when a parameter will be evaluated, since it will not happen until that parameter is actually used. This makes it extremely difficult to determine the order that side effects occur in, which in turn makes it hard to predict the behavior of a program.

- d. [8 pts.] Apply beta reduction to the following lambda calculus term until no more reductions are possible. Show each individual step of beta reduction (i.e., don't skip steps).

$(\lambda x.\lambda y.\lambda z.z\ x\ y)\ (\lambda a.a)\ (\lambda b.\lambda c.b)\ (\lambda d.\lambda e.e)$

**Answer:**

$$\begin{aligned}
 (\lambda x.\lambda y.\lambda z.z\ x\ y)\ (\lambda a.a)\ (\lambda b.\lambda c.b)\ (\lambda d.\lambda e.e) &\rightarrow (\lambda y.\lambda z.z\ (\lambda a.a)\ y)\ (\lambda b.\lambda c.b)\ (\lambda d.\lambda e.e) \\
 &\rightarrow (\lambda z.z\ (\lambda a.a)\ (\lambda b.\lambda c.b))\ (\lambda d.\lambda e.e) \\
 &\rightarrow (\lambda d.\lambda e.e)\ (\lambda a.a)\ (\lambda b.\lambda c.b) \\
 &\rightarrow (\lambda e.e)\ (\lambda b.\lambda c.b) \\
 &\rightarrow (\lambda b.\lambda c.b)
 \end{aligned}$$

e. [5 pts.] Currying.

- i. Write a function `curry3` :  $((\text{'a} * \text{'b} * \text{'c}) \rightarrow \text{'d}) \rightarrow (\text{'a} \rightarrow \text{'b} \rightarrow \text{'c} \rightarrow \text{'d})$  whose input is a function that takes a tuple containing three arguments, and whose output is a curried version of that function. For example:

```
let f (x, y, z) = x + y + z
let g = curry3 f
g 1 2 3 (* returns 6 *)
```

**Answer:**

```
let curry3 f x y z = f (x, y, z)
```

- ii. Write a function `uncurry3` :  $(\text{'a} \rightarrow \text{'b} \rightarrow \text{'c} \rightarrow \text{'d}) \rightarrow ((\text{'a} * \text{'b} * \text{'c}) \rightarrow \text{'d})$  that is the inverse of `curry3`. For example:

```
let f x y z = x + y + z
let g = uncurry3 f
g (1, 2, 3) (* returns 6 *)
```

**Answer:**

```
let uncurry3 f (x, y, z) = f x y z
```

2. [25 pts.] **Language Features.**

- a. [16 pts.] Enter **Yes** or **No** in each of the cells in this table to indicate whether the listed language has the listed feature. For “static types,” we mean *mostly* having static types.

	Static types	Strong typing	Dynamic scoping	Call-by-value
Java	Yes	Yes	No	Yes
C	Yes	No	No	Yes
Ruby	No	Yes	No*	Yes
OCaml	Yes	Yes	No	Yes

\* - This answer is correct—although local Ruby variables are bound starting at the first assignment, they still exist only within the static scope of their assignment.

- b. [9 pts.] In lecture, we showed several times how to encode features of one language in another, e.g., using objects to represent closures, using closures to represent objects, encoding state in a purely functional language, and, of course, encoding booleans, pairs, integers, etc in lambda calculus. Yet despite the fact that all of these encodings exist, programmers still use many different languages. Briefly give **three different reasons** programmers might choose one language over another for a particular project.

**Answer:** There are a lot of possible answers; here are a few:

- The programmer may only know one language.
- There might be existing code in the language the programmer wants to reuse, use as a starting place, or use an API to.
- One language’s idioms might be particularly useful for solving a certain problem (e.g., systems programming in C, regexps in Ruby, pattern matching in OCaml).
- The programmer might be required to use the language by their employer/customer/professor.
- There might be performance requirements that are more easily satisfied in one language than another (e.g., pretty much anything versus Ruby).

3. [15 pts.] **Associative Lists.** Write the following functions on associative lists. You may **not** use any existing functions from the `List` module. Throughout, compare values using `=`.

- a. [5 pts.] `assoc : 'a -> ('a*'b) list -> 'b`. A call `assoc x l` should return the right component of the tuple that `x` is the left component of. If no such tuple exists in `l`, this function should raise the `Not_found` exception.

**Answer:**

```
let rec assoc x = function
  [] -> raise Not_found
| (a,b)::t -> if a=x then b else (assoc x t)
```

- b. [5 pts.] `remove : 'a -> ('a*'b) list -> ('a*'b) list`. A call `remove x l` should return a new list that is the same as `l`, except any tuples containing `x` as the left element should be removed. If `x` was not in the list, it is not an error—`remove` just returns a copy of the list in this case. If `x` occurs multiple times in `l`, all occurrences should be removed.

**Answer:**

```
let rec remove x = function
  [] -> []
| (a,b)::t -> if a=x then (remove x t) else (a,b)::(remove x t)
```

- c. [5 pts.] `upd : ('a*'b) list -> 'a -> 'b -> ('a*'b) list`. A call `upd l x v` should return a new list that is the same as `l`, except that `x` should now be associated with `v`. There should be exactly one association with `x` in the output list.

**Answer:**

```
let upd l x v = (x,v)::(remove x l)
```

4. [35 pts.] **Evaluating Three-Address Code** In this problem, you will write an evaluator for *three-address code*, which is a common representation used inside of compilers.
- a. [15 pts.] Consider the following OCaml types, which represent one statement of three-address code:

```

type operand = EInt of int | EVar of string
type operator = OAdd | OSub | OMul
type stmt =
  ESkip
  | ECopy of string * operand
  | EBinOp of string * operand * operator * operand
type env = (string * int) list

```

Here are some examples of source-level instructions and their representation in this data type.

Source	stmt representation
x = 3	ECopy("x", EInt 3)
x = y	ECopy("x", EVar "y")
x = y + 2	EBinOp("x", EVar "y", OAdd, EInt 2)
x = a * b	EBinOp("x", EVar "a", OMul, EVar "b")

The statement `ESkip` does nothing, i.e., the program state is the same before and after it is evaluated. The type `env` is the type of environments, which track the values of variables (which for this question can only be integers).

Write a function `eval_stmt : env -> stmt -> env` that, given an `env` and a statement, returns the new environment updated according to the statement. For example,

```
eval_stmt [("x", 2)] (ECopy("y", EVar "x")) = [("y", 2); ("x", 2)]
```

You don't need to worry about accessing undefined variables, or the order variables appear in the environment. Each variable should appear at most once in the environment (and you can assume that is the case for the initial `env`). You can use the functions you wrote in the previous problem, and you may continue your answer at the top of the next page.

**Answer:**

```

let val_in_env e = function
  EInt n -> n
  | EVar x -> assoc x e

let rec eval_stmt e = function
  ESkip -> e
  | ECopy (x, o) = upd e x (val_in_env e o)
  | EBinOp (x, o1, op, o2) =
    let v1 = val_in_env e o1 in
    let v2 = val_in_env e o2 in
    let v = (match op with
      OAdd -> (v1 + v2)
      | OSub -> (v1 - v2)
      | OMul -> (v1 * v2)) in
    upd e x v

```

*You may continue your answer to part a here*

- b. [5 pts.] Write a function `eval_stmts : env -> stmt list -> env` that, given an `env` and a list of statements, returns the new environment updated according to the sequence of statements. Statements should be evaluated in order, from the front of the list to the back. For example,

```
eval_stmts [("x", 2)] [ECopy("x", EInt 3); ECopy("y", EVar "x")]
```

would return `[("y", 3); ("x", 3)]`. **Your function must be tail recursive.** You can, of course, call any functions you have previously defined on the exam.

**Answer:**

```
let rec eval_stmts e = function
  [] -> e
| h::t -> eval_stmts (eval_stmt e h) t
```

- c. [15 pts.] Finally, we will use the following types to represent a program as a control flow graph:

```

type branch =
  BJump of int
| BISZero of string * int * int
| BExit
type basic_block = stmt list * branch
type cfg = (int * basic_block) list

```

Here a `basic_block` is a list of statements followed by a `branch`, which is either an unconditional jump `BJump i` to another `basic_block i` (we'll use integers to index `basic_blocks` in a program); a conditional jump `BISZero (x, t, f)`, which gets the current value of `x`, and if it is 0, jumps to `t`, and otherwise jumps to `f`; or `BExit`, which exits the program. A `cfg` (or control-flow graph) is a list mapping integers to `basic_blocks`.

Write a function `eval_cfg : env -> cfg -> int -> env` such that `eval_cfg e c i` evaluates `c`, starting in the environment given by `e`, and evaluation starts by evaluating the block indexed by `i`. After it evaluates that block, it should either stop (if the branch at the end of the block is `BExit`) or continue by evaluating the block indicated by the branch. This function returns the final environment when `BExit` is reached. You don't need to worry about jumps to undefined indexes. **Your function must be tail recursive.** And as before, you can call any functions you have previously defined on the exam.

**Answer:**

```

let rec eval_cfg e c i =
  let (stmts, b) = assoc i c in
  let e' = eval_stmts e stmts in
  match b with
  | BExit -> e'
| BJump i' -> eval_cfg e' c i'
| BISZero (x, t, f) ->
  if (assoc x e' = 0)
  then eval_cfg e' c t
  else eval_cfg e' c f

```