

1. [20 pts.] **Short Answer.**

- a. [4 points] Consider the following program, written in a C-like language:

```
void f(integer x, integer y) {
    y = y+1;
    x = y;
}
void main() {
    integer x = 0;
    integer y = 10;
    f(x,y);
    printf("x=%d, y=%d\n", x, y);
}
```

Assume `x` is always call-by-reference. What the does the program print if `y` is call-by-value? What is printed if `y` is call-by-reference?

Answer: Under call-by-value, the program prints `x=11 y=10`. Under call-by-reference, the program prints `x=11 y=11`.

- b. [5 pts.] Consider the Ruby expression
- `a+4`
- , without knowing anything else about object
- `a`
- . What does the “+” refer to? What sequence of events happens when this expression is evaluated?
- Note: This is intended to be an easy question—we’re not looking for anything deep here.*

Answer: The “+” refers to the “+” method of `a`. When this expression is evaluated, the integer 4 is evaluated, and then it is passed as a parameter to the method.

- c. [4 pts.] PL/1 is an imperative procedure-oriented language from the 1970s. The first PL/1 compiler did not implement a stack, and instead, each activation record was implemented as an `allocate()` and a `free()` of the storage from the heap as the procedure was entered and exited, respectively. Other data could also be allocated on the same heap by the programmer. In two sentences, describe two potential advantages or disadvantages of this implementation.

Answer:

- Very slow procedure activation and return due to need to call the allocator.
 - Increased memory pressure on the heap, which could cause fragmentation if the program allocates heap objects as well.
 - Activation records are likely to be different sizes, hence they will not be reused nearly as efficiently as the stack.
 - Hard to detect infinite recursion, because the stack is mixed up with the heap.
 - By the same token, the stack can grow as much as possible given the memory space.
 - Support for higher-order functions may be easier, because local variables are all on the heap.
- d. [4 pts.] A *real-time* program is one where various actions the program carries out must occur within a certain amount of time. For example, if a car's braking system were controlled by software, then if the software didn't do its job in time, you might crash. Why is an implementation that uses mark-and-sweep or stop-and-copy garbage collection not a good idea in a real-time program? Is reference counting a better solution? Explain your answer.

Answer: Garbage collection causes execution to halt, perhaps for a long period, while the garbage collector reclaims unused space. This may be a significant problem in a realtime program that needs to respond quickly to events.

Reference counting may be a better solution because its cost is spread out among all the operations in the program. However, reference counting can still have significant pauses, because removing the last reference to a large data structure could cause the whole structure to be collected.

- e. [3 pts.] Fill in the following blanks with the correct language from the following list. Write one answer per blank. Languages may be used zero or more times in different answers.

ALGOL, BASIC, C, CLU, COBOL, FORTRAN, LISP, PL/1

- i. _____ was designed mostly for scientific computing
- ii. _____ was designed mostly for business computing
- iii. _____ used garbage collection to reclaim heap storage
- iv. _____ included exception handling
- v. _____ is considered the first high-level language
- vi. _____ was originally described using a BNF grammar

Answer:

- i. FORTRAN or PL/1 or ALGOL
- ii. COBOL or PL/1
- iii. LISP (or CLU)
- iv. CLU or PL/1
- v. FORTRAN
- vi. ALGOL

2. [10 pts.] **OCaml Types.** Write down the type the OCaml interpreter will assign to each of the following functions. If the OCaml interpreter would use polymorphic types like 'a, you can pick any mutually-consistent set of polymorphic type variables. (E.g., 'a->'a and 'b->'b are the same.)

a. `let f x y = x::y`

Answer: 'a -> 'a list -> 'a list

b. `let f (x, y) z = z x y`

Answer: 'a * 'b -> ('a -> 'b -> 'c) -> 'c

c. `let f x y = x (x y)`

Answer: ('a -> 'a) -> 'a -> 'a

d. `let rec f x y = match x with [] -> y | (_::xs) -> f xs (y+1)`

Answer: 'a list -> int -> int

e. `let f a b c = b a c`

Answer: 'a -> ('a -> 'b -> 'c) -> 'b -> 'c

3. [20 pts.] **Formal Languages.**

- a. [4 pts.] Give a DFA for the set
- $R = \{a^m b^n c^p d^q \mid m, n, p, q \text{ are all } \geq 1\}$
- .

Answer:

- b. [4 pts.] Give a context-free grammar for
- R
- with
- $m = n$
- and
- $p = q$
- .

Answer:

$$S \rightarrow XY$$

$$X \rightarrow aXb \mid ab$$

$$Y \rightarrow cYd \mid cd$$

- c. [4 pts.] Give a context-free grammar for
- R
- with
- $m = p$

Answer:

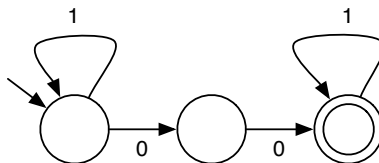
$$S \rightarrow XD$$

$$X \rightarrow aXc \mid aBc$$

$$B \rightarrow bB \mid b$$

$$D \rightarrow dD \mid d$$

- d. [4 pts.] Let L be the language recognized by the following DFA over the alphabet $\{0, 1\}$.



The complement of language L is defined as $\neg L = \{s \mid s \notin L\}$. Draw a DFA that accepts $\neg L$, and briefly describe a general procedure for complementing a DFA.

Answer: To complement a DFA, just switch the accepting and non-accepting states, being sure to account for the implicit dead state.

- e. [2 pts.] Give the DFA with the fewest states that accepts $\neg L \cup L$, i.e., strings that are either in $\neg L$ or in L .

Answer:

- f. [2 pts.] Does the same procedure you described in part 3d work for complementing an NFA? Justify your answer by either arguing it does or showing by counterexample it does not.

Answer:

4. [8 pts.] **Lambda calculus**

- a. [4 pts.] Reduce the following term until no more reductions are possible:
- $(\lambda z.z) (\lambda x.x x) (\lambda x.x y)$

Answer:

$$\begin{aligned}
(\lambda z.z) (\lambda x.x x) (\lambda x.x y) &\rightarrow (\lambda x.x x) (\lambda x.x y) \\
&\rightarrow (\lambda x.x y) (\lambda x.x y) \\
&\rightarrow (\lambda x.x y) y \\
&\rightarrow y y
\end{aligned}$$

- b. [4 pts.] Suppose we define the following combinators:

- $true = \lambda x.\lambda y.x$
- $false = \lambda x.\lambda y.y$
- $not = \lambda x.x false true$

Show that $not(not x) \rightarrow \dots \rightarrow x$ where x is either *true* or *false*. *Hint: To save writing, first reduce $not(not x)$ as much as possible, and then set x to either *true* or *false*.*

Answer:

$$\begin{aligned}
not(not x) &= not((\lambda x.x false true) x) \\
&\rightarrow not(x false true) \\
&= (\lambda x.x false true) (x false true) \\
&\rightarrow x false true false true
\end{aligned}$$

So then

$$\begin{aligned}
not(not true) &= true false true false true \\
&= (\lambda x.\lambda y.x) false true false true \\
&\rightarrow false false true \\
&= (\lambda x.\lambda y.y) false true \\
&\rightarrow true
\end{aligned}$$

and

$$\begin{aligned}
not(not false) &= false false true false true \\
&= (\lambda x.\lambda y.y) false true false true \\
&\rightarrow true false true \\
&= (\lambda x.\lambda y.x) false true \\
&\rightarrow false
\end{aligned}$$

5. [8 pts.] **Operational Semantics.** Consider the following language, which is lambda calculus to which we've added booleans as primitives:

$$e ::= x \mid e e \mid \lambda x.e \mid \text{true} \mid \text{false} \mid \text{not } e$$

We would like to give an operational semantics for this language that defines reductions of the form $A; e \rightarrow v$, meaning in environment A (which binds variables to values), expression e evaluates to value v . The values v are given by

$$v ::= (A, \lambda x.e) \mid \text{true} \mid \text{false}$$

Here true and false are the appropriate boolean values, and $(A, \lambda x.e)$ is a closure with environment A and function $\lambda x.e$.

Here is the start of an operational semantics:

$$\frac{}{A; x \rightarrow A(x)} \quad \frac{}{A; \lambda x.e \rightarrow (A, \lambda x.e)} \quad \frac{}{A; \text{true} \rightarrow \text{true}}$$

$$\frac{A; e_1 \rightarrow (A', \lambda x.e) \quad A; e_2 \rightarrow v \quad A', x : v; e \rightarrow v'}{A; e_1 e_2 \rightarrow v'}$$

- a. [4 pts.] What is the parameter passing mechanism represented by these semantics? In order to receive credit for your answer, justify it by referring to the rules.

Answer: This is call-by-value, because the expression e_2 is evaluated to a value v before the function is invoked.

- b. [4 pts.] Write down operational semantics rules for false and not , following the pattern above. (Don't worry about the fonts.)

Answer:

$$\frac{}{A; \text{false} \rightarrow \text{false}}$$

$$\frac{A; e \rightarrow \text{true}}{A; \text{not } e \rightarrow \text{false}}$$

$$\frac{A; e \rightarrow \text{false}}{A; \text{not } e \rightarrow \text{true}}$$

6. [8 pts.] **Threading in Java.** In this question, you will write Java code for a synchronization construct called a *barrier*. A barrier is created with a certain number of parties n . When a thread calls `await()`, it *enters the barrier* and blocks until a total of n parties have entered the barrier. When the n th party enters the barrier, all the threads waiting at the barrier wake up and unblock, and the n th thread continues without blocking. A barrier may also be *reset* so that it starts fresh in counting up to n .

Write a class `Barrier` that implements this behavior. Your class should have a constructor `Barrier(int n)` to create a barrier with n parties, a method `void await()`, and a method `void reset()`.

Answer:

```
public class Barrier {
    private int n, cur;

    public void Barrier(int n) {
        this.n = n; cur = 0
    }

    public synchronized void await() {
        cur++;
        while (cur < n)
            wait();
        notifyAll();
    }

    public synchronized void reset() {
        cur = 0;
    }
}
```

7. [16 pts.] **Boolean expression.** A *boolean expression* is a boolean function built from operators like *and* and *not*. Examples are $(x_1 \wedge x_2) \wedge x_3$ or $\neg x_1 \wedge x_3 \wedge x_4$. (Recall that \wedge means *and*.) Here is an OCaml data type for abstract syntax trees for boolean expressions:

```
type expr =
  False
  | True
  | Var of string
  | And of expr * expr
  | Not of expr
  | Forall of string * expr
```

Here `False`, `True`, `And`, and `Not` represent the usual boolean values and operators. In this AST, boolean expressions may include variables, given by AST node `Var`. We will use an associative list of the following type to track the values of variables:

```
type assn = (string * bool) list
```

Here if an `assn` contains the pair (x, b) , then that assignment gives the variable x the value b .

The last expression represents \forall , where the expression $\forall x.e$ is true if e is true under all possible assignments to x , i.e., if e is true for $x = \text{true}$ and $x = \text{false}$.

- a. [4 pts.] Write a function `free : expr -> string list` that returns a list of free variables in the expression. Just as in a programming language, the free variables of a boolean expression are those not bound by `Forall`. For example, `free (Var "x") = ["x"]` and `free (Forall ("x", And (Var "x", Var "y")))` = ["y"].

Answer:

```
let rec free = function
  False -> []
  | True -> []
  | Var v -> [v]
  | And (e1, e2) -> (free e1) @ (free e2)
  | Not e -> free e
  | Forall (x, e) -> List.filter (fun y -> not (y = x)) (free e)
```

- b. [6 pts.] Write a function `eval : assn -> expr -> bool` that evaluates the boolean expression on the given variable assignment. For example, given the OCaml value `And(And(Var "x1", Var "x2"), Var "x3")` and the assignment `[("x1", true); ("x2", false); ("x3", true)]`, the `eval` function should return `false`. You may assume that any variable encountered during evaluation is either given a value by the assignment or bound by an enclosing `Forall`.

Answer:

```
let rec eval env = function
  | False -> false
  | True -> true
  | Var v -> List.assoc env v
  | And (e1, e2) -> (eval env e1) && (eval env e2)
  | Not e -> not (eval env e)
  | Forall (x, e) -> (eval (x, true)::env e1) && (eval (x, false)::env e1)
```

- c. [4 pts.] Write a function `taut : expr -> bool` that returns true if and only if the expression is a tautology (i.e., is always true under any assignment to the free variables). You may use functions you wrote for parts a and b.

Answer:

```
let taut e =
  let vars = free e in
  let taut' env = function
    [] -> eval env e
    | (x::xs) ->
      let t = taut' ((x, true)::env) xs in
      let f = taut' ((x, false)::env) xs in
      t && f
  in
  taut' [] vars
```

- d. [2 pts.] Consider the following grammar for a subset of boolean expressions, where we've chosen *prefix* notation, so that the operator appears first, followed by its arguments.

$$expr \rightarrow \text{true} \mid \text{false} \mid \text{and } expr \ expr \mid \text{not } expr$$

Is this grammar ambiguous? Justify your answer.

Answer: No. One way to see this is to observe that in a predictive parser, we always know which production to choose based on the lookahead.

8. [10 pts.] **Language Comparison.** *Interactive fiction* is a computer gaming genre based on text input and output. The game presents the player with text descriptions of the world, and the player moves around the world and interacts with it by typing in a limited vocabulary of noun-verb commands. For example, here is a brief transcript of playing the game Zork, which was developed in the late 1970's. The text entered by the player is preceded by the > prompt, and everything else is displayed by the game.

```
Welcome to Zork (originally Dungeon).  This version created 11-MAR-91 (PHP mod 03-AUG-05)
...
You are in an open field west of a big white house with a boarded front door.
There is a small mailbox here.

> open mailbox
Opening the mailbox reveals:
A leaflet.

> take leaflet
Taken.

> read leaflet
Welcome to Zork (originally Dungeon)!
...
```

Suppose you were asked to write a program that implements an interactive fiction game. Out of Ruby, OCaml, or Java, which language(s) would you choose for your implementation, and why? You will receive two points for each advantage or disadvantage you list for using the language for this application. We will subtract points if you say something that is wrong. Thus we recommend keeping your answer to roughly one paragraph, or two at most. You may continue on the next page if you are feeling expansive.

Name: _____

You may continue your answer here