

Fall 2004 - Programming Languages Qualifier

Computer Science Department
University of Massachusetts Lowell
Lowell, MA 01854

Sep. 18, 2004.

There are 8 problems on this exam. You should write the answers to problems 1–4 in one blue book and the answers to problems 5–8 in the other blue book.

1. Programming Language Concept Distinctions (30 pts)

Briefly describe the similarities and/or differences between the following concepts:

- (a) *l*-values and *r*-values.
- (b) lists and sequences (aka streams)
- (c) lexical scope and dynamic scope
- (d) operational semantics and denotational semantics.
- (e) preconditions and postconditions.
- (f) structures, functors, and signatures (in SML).
- (g) partial correctness and total correctness.
- (h) normal-order and applicative-order evaluation.
- (i) parametric polymorphism and subtype polymorphism.
- (j) definite Horn clauses and literals.

2. Unification (5 pts).

Indicate whether the following types unify in ML. If they do have a most general unifier, state it. If not, why not?

- (a) `'a` and `int`
- (b) `'a -> bool` and `int * bool`
- (c) `'a -> 'a` and `int -> 'b`
- (d) `('a * 'b) -> ('a * bool)` and `(int * bool) -> 'c`
- (e) `'a -> 'b` and `('a -> int) -> ('b -> bool)`

3. Hoare axiomatization (10 pts).

(a) (6 pts) Fill in the blanks by applying the assignment axiom and composition rules:

i. { } $x := z; y := x$ { $y=0$ }

ii. { } $x := z; y := x-y$ { $y \geq 0 \wedge t=5$ }

(b) (4 pts) The following version of the assignment axiom is *not* sound:
{ p } $x := e$ { $p \wedge x=e$ }

Give an example showing how it goes wrong.

4. This (self) and super (10 pts)

Given the following uSmalltalk code, what is printed in response to the last 8 lines?

If unfamiliar with uSmalltalk, the comments should tell you what is going on. "self" in Smalltalk is called "this" in some other OO languages. "super" is called super in most OO languages. If you know Java or C++ you should be able to do this problem. (in C++ think "virtual member function" when you see "method").

```
(class A Object () ; define class A inheriting from Object
  (method whatis () (isa self)) ; method whatis passes "isa" to self (this)
                                ; this.isa () for Java programmers
                                ; this->isa () for C++ programmers
  (method isa () #A)) ; method isa returns symbol A (prints as A)

(class B A () ; define class B inheriting from A
  (method isa () #B) ; method isa returns symbol B (prints as B)
  (method bypass () (isa super))) ; method bypass passes "isa" to super
                                   ; super.isa () for Java programmers
                                   ; super::isa () for some C++ implementations

(class C B () ; define class C inheriting from B
  (method isa () #C) ; method isa returns symbol C (prints as C)
  (method bypass () (whatis super)) ; method bypass invokes super's whatis.
  (method whatis () #Cish)) ; method whatis returns symbol Cish

(class D B () ; define class D inheriting from B
  (method isa () #D)) ; method isa returns symbol D

(val a (new A)) ; create a new object a, instance of class A
(val b (new B)) ; create a new object b, instance of class B
(val c (new C)) ; create a new object c, instance of class C
(val d (new D)) ; create a new object d, instance of class D

(whatis a) ; prints:
(whatis b) ; prints:
(whatis c) ; prints:
(whatis d) ; prints:

(bypass a) ; prints:
(bypass b) ; prints:
(bypass c) ; prints:
(bypass d) ; prints:
```

5. Fixed points, Lambda calculus (10 pts).
 (2 pts) What is (define) a fixed point of a function f ?
 (2 pts) What is (define) a fixpoint combinator?
 (6 pts) Show that the following lambda-calculus term is a fixpoint combinator:
 $(\lambda d.\lambda f.f((d d) f))(\lambda d.\lambda f.f((d d) f))$
6. Control (10 pts).
 (3 pts) What is a continuation?
 (3 pts) What is an exception?
 (4 pts) Explain briefly how to use exceptions in ML.
7. Operational semantics of Prolog (10 pts).

Assume that prolog has a predicate `print(X)` that prints what the interpreter loop would print for `X` and then always succeeds.

Suppose we define `backprint` as

```
backprint(_).
backprint(X) :- print(X).
```

what will `print` and `backprint` print (in order) if you execute the query

```
member(X, [1,2]), print(trying(x, X)), backprint(failed(x, X)),
member(Y, [3,2]), print(trying(y, Y)), backprint(failed(y, Y)),
X > Y.
```

Write out all printed text.

(`member(X, [1,2])` first unifies `X` with 1. If the computation backtracks into `member` it unifies `X` with 2. Once `member` reaches the end of the list then `member` fails. `X > Y` succeeds if `X` and `Y` are both instantiated to numbers and the number for `X` is greater than the number for `Y`.)

Suppose we define `backprint` as

```
backprint(_).
backprint(X) :- print(X), fail.
```

now what will `print` and `backprint` print (in order) if you execute the query

```
member(X, [1,2]), print(trying(x, X)), backprint(failed(x, X)),
member(Y, [3,2]), print(trying(y, Y)), backprint(failed(y, Y)),
X > Y.
```

8. ML and Scheme programming. (15 pts)

In Prolog we could define `reverse` as follows:

```
append([], X, X).
append([X|Y], Z, [X|W]) :- append(Y, Z, W).
reverse([], []).
reverse([X|Y], W) :- reverse(Y, Z), append(Z, [X], W).
```

This version of `reverse` requires $O(N^2)$ time because of its use of `append`

A faster version of `reverse` in Prolog could be written using the equivalent of an *accumulating parameter*.

```
reverse([], []).
reverse([X|Y], Z) :- aux_reverse([X|Y], [], Z).
aux_reverse(..., ..., ...) :- code deliberately omitted
```

The example shows one of the drawbacks of non-block-structured languages (i.e., those languages that do not allow the definition of local functions): in order to implement an efficient version of `reverse` we must clutter the global (relation) name-space with `aux_reverse`.

- (a) (3 pts: ML) Write an ML function `append` that takes two lists and appends the second one to the first.
- (b) (3 pts: ML) Write an ML function `reverse` that takes a list as argument and produces the reversed list without using accumulating parameters. (It does not have to define `append` internally.)
- (c) (3 pts: ML) Rewrite `reverse` so that it runs in linear time, by using a local function, `aux_reverse`, with an accumulating parameter.
- (d) (3 pts: Scheme) Repeat part b) in Scheme. You can assume the existence of an `append` function in Scheme.
- (e) (3 pts: Scheme) Repeat part c) in Scheme