EECS 490: Final Exam

Total time: 2 hours
Total points: 60

This is an open book exam.

Problem 1

For the following problems, you may use the function member? that we studied in class as a helper function. The function member? is reproduced below:

```
(define member?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      (else (or (eq? (car lat) a) (member? a (cdr lat)))))))
```

Problem 1a (3 Points)

Write a function set?. The function set? takes one argument: a list of atoms *lat*. set? returns true iff no atom occurs more than once in *lat*; otherwise set? returns false. The following examples illustrate the intended behaviour.

Input	Output
(set? '(apples peaches apples plums))	#f
(set? '(apples peaches pears plums))	#t

Problem 1b (3 Points)

Write a function makeset. The function makeset takes one argument: a list of atoms *lat*. makeset returns *lat* with duplicates removed. The following examples illustrate the intended behaviour.

Input	Output
(makeset '(apples peaches apples plums))	(apples peaches plums)
(makeset '(apples peaches pears plums))	(apples peaches pears plums)
(makeset '(apple peach pear peach plum apple lemon peach))	(apple peach pear plum lemon)

Problem 1c (3 Points)

Write a function intersect. The function intersect take two sets *set1* and *set2* as arguments and returns the intersection of the two sets. The following example illustrate the intended behaviour.

Input	Output
(intersect '(stewed tomatoes and macaroni) '(macaroni and cheese))	(macaroni and)

Problem 1d (3 Points)

Write a function intersectall. The function intersectall takes a non-empty list of sets *l-set* as argument and returns the intersection of all the sets in the list. You may use any of the functions you wrote above as helper functions. The following examples illustrate the intended behaviour.

Input	Output
(intersectall '((a b c) (c a d e) (e f g h a b)))	(a)
(intersectall '((6 pears and) (3 peaches and 6 peppers) (8 pears and 6 plums)))	(6 and)

Problem 2 (8 Points)

Recall Church booleans and Church numerals from Chapter 5 of the Pierce book.

- true = \t. \f. t
- false = \t. \f. f

- c0 = \s. \z. z
- c1 = \s. \z. s z
- c2 = \s. \z. s(s z)
- c3 = \s. \z. s(s(s z))
- ...

Write a combinator iseven n that returns the Church boolean true if n is an even number and returns the Church boolean false if n is an odd number. You may assume that n is a Church numeral.

Problem 3

Consider the language of boolean and arithmetic expressions discussed in class (Figures 3-1 and 3-2 in the Pierce book). Suppose we add a new syntactic form:

- decrement t by twice t

Suppose we also add new evaluation rules:

-
  ```
  t2 ---> t2'
  -------------------------------------------------------
  decrement t1 by twice t2 ---> decrement t1 by twice t2'
  ```
- decrement t1 by twice 0 ---> t1
- decrement t1 by twice (succ nv1) ---> decrement (pred (pred t1)) by twice nv1

Problem 3a (7 Points)

Does the termination theorem discussed in class (Theorem 3.5.12 in the Pierce book) remain valid for the new language?

If the theorem is no longer valid, then present a counter example. If the theorem is still valid but its proof is different, then present the new proof.

Problem 3b (3 Points)

Which of the other theorems discussed in class (Theorems 3.5.4, 3.5.7, 3.5.8, 3.5.11 in the Pierce book) remain valid?

You do not have to prove or disprove the theorems. Just state which theorems remain valid and which ones do not.

Problem 4

Consider the subset of the Java bytecode language discussed in class (Figures 3 and 4 in the POPL 1998 paper by Stata and Abadi). Your goal for this problem is to extend the language to support the following two instructions:

- goto L
- add

The instruction goto L jumps to label L. The instruction add pops the top two elements off the stack, adds them, and pushes the result onto the stack.

Problem 4a (4 Points)

Extend the operational sematics (dynamic semantics) of the language (Figure 3) to support the new instructions.

Problem 4b (4 Points)

Extend the typing rules (static sematics) of the language (Figure 4) to support the new instructions.

Problem 5

Consider the language of Featherweight Java discussed in class (Figures 19-1, 19-2, 19-3, 19-4 in the Pierce book). Note that unlike Java, Featherweight Java has no *null* pointers. The goal of this problem is to add null pointers to Featherweight Java. The following examples illustrate the intended semantics.

```
class A extends Object {
  B b;
  A (B b) { super(); this.b = b; }
  A setB(B b) { return new A(b); }
  B getB() { return this.b; }
}

class B extends Object {
  B () { super(); }
  B id() { return this; }
}
```

Input	Output
new A(new B())	new A(new B())
new A(new B()).getB()	new B()
new A(new B()).getB().id()	new B()
new A(null)	new A(null)
new A(null).getB()	null
new A(null).getB().id()	error (null pointer dereferencing)
new A(new B()).setB(null)	new A(null)
new A(new B()).setB(null).getB()	null
new A(new B()).setB(null).getB().id()	error (null pointer dereferencing)
null	null
(A) null	null

Problem 5a (3 Points)

Extend the syntax of Featherweight Java (Figure 19-1 in the Pierce book) to support null pointers.

Problem 5b (3 Points)

Extend the evaluation rules (Figure 19-3 in the Pierce book) to support null pointers.

Problem 5c (3 Points)

Extend the typing rules (Figures 19-1 and 19-4 in the Pierce book) to support null pointers.

Problem 5d (3 Points)

What can you say about well typed programs in the new language? Modify the statements of the progress and preservation theorems (Theorems 8.3.2 and 8.3.3 in the Pierce book) so that they hold for the new language.

You only have to state the theorems. You do not have to prove the theorems.

Problem 6 (10 Points)

Recall the verfication conditions for some simple commands discussed in class:

VC(skip, B) =	**B**
VC(c1; c2, B) =	**VC(c1, VC(c2, B))**
VC(x:=e, B) =	**B[x→e]**
VC(if b then c1 else c2, B) =	**(b ⇒ VC(c1, B)) ∧ (¬b ⇒ VC (c2, B))**
VC(while$_I$ b do c, B) =	**I ∧ (∀ x_1..x_n. I ⇒ (b ⇒ VC(c, I) ∧ ¬b ⇒ B)),** ***where x_1..x_n are variables modified in c***

Find an appropriate loop invariant and prove the partial correctness assertion for the following factorial program, using the verification condition rules discussed in class.

{n = x, x > 0, fact = 1} while (n > 0) do (fact := n × fact; n := n − 1) {fact = x!}

That is, compute the verification condition before the while loop, and confirm that the given precondition **{n = x, x > 0, fact = 1}** implies the verification condition you computed.

bchandra@eecs.umich.edu