Solutions to In-Class Problems — Week 2, Mon

**Problem 1.** Two Boolean formulas $F_1(x_1, \ldots, x_n)$ and $F_2(x_1, \ldots, x_n)$ are equivalent iff they yield the same truth value for all truth assignments to the variables $x_1, \ldots, x_n$.

(a) Describe an infinite set of equivalent Boolean formulas.

**Solution.** Define the formulas $F_i$ as $F_0 ::= x_0$ and $F_{i+1} ::= (F_i \land x_0)$ for all $i \geq 0$. These formulas are all equivalent, since for any truth assignment they all have the same truth value as $x_0$.

(b) How many equivalence classes are there of formulas with (at most) variables $x_1, \ldots, x_n$?

**Solution.** Formulas are equivalent iff their truth tables agree, so there are as many equivalence classes as there are truth tables. Given $n$ variables, there are $2^{2^n}$ possible truth tables. To see this, think of a truth table as having a row for each possible truth assignment. A truth assignment consists of a True or False value for each variable, so there are $|\{T,F\}|^n = 2^n$ possible truth assignments. Then, a truth table consists of an assignment of True or False to each truth assignment, so with $2^n$ truth assignments there are $2^{2^n}$ possible truth tables, giving $2^{2^n}$ equivalence classes of formulas.

**Problem 2.** A Scheme expression satisfies the “Variable Convention” if no variable identifier is bound more than once, and no identifier has both bound and unbound occurrences. For example, the expression

```
(let ((x 2) (y 5))
  (+ ((lambda (x) (+ x 1)) 3) ((lambda (z) (+ x y z 11)) 99) z)).
```

violates the Variable Convention because $x$ is bound twice—once by let and once by lambda, and also because $z$ has both a bound and an unbound occurrence.

Copyright © 2003, Prof. Albert Meyer.
Any expression can be slightly modified to satisfy the Convention solely by adding integer suffixes to some of the bound identifiers—in a way that preserves all the binding structure and all the computational behavior of the original expression.

For example, by adding suffix 0 to the x’s and z’s bound by the lambda’s, we obtain an equivalent expression which satisfies the Variable Convention:

\[
\text{(let ((x 2) (y 5))}
\text{ (+ ((lambda (x0) (+ x0 1)) 3) ((lambda (z0) (+ x y z0 11)) 99) z)).}
\]

Show how to add such suffixes to the identifiers in

\[
(a \ b \ c \ d \ e
\text{(let ((a e) (b c))}
\text{ (a b c d e)}
\text{ (letrec ((a c)(c b))}
\text{ (a b c d e))))))
\]

to obtain an equivalent expression satisfying the Variable Convention. (See the Scheme reference manual to find out the scoping rules for letrec.)

SOLUTION:

\[
(a \ b \ c \ d \ e
\text{(let ((a0 e) (b0 c))}
\text{ (a0 b0 c d e)}
\text{ (letrec ((a1 c0)(c0 b0))}
\text{ (a1 b0 c0 d e))))))
\]

Problem 3. (a) Define a Scheme procedure self-compose which, given a one-parameter procedure argument, \(f\), returns a procedure that computes \((f \circ f)\), that is, the composition of \(f\) with itself. For example, the Scheme expressions

\[
\text{(define (self-compose f) <your definition>)}
\text{(define (s n)(* n n))}
\text{((self-compose s) 3)}
\]

would return the integer 81.

SOLUTION:

\[
\text{(define (self-compose f) (lambda (x) (f (f x)))})
\]
Solutions to In-Class Problems — Week 2, Mon

(define (s n) (* n n))

((self-compose s) 3)
;Value: 81

(b) What should (((self-compose self-compose) s) 3) return? Explain.

SOLUTION:
Reasoning using an informal Substitution Model:

( ((self-compose self-compose) s) 3)
= ( (((lambda (x) (self-compose (self-compose x)))) s) 3)
= ( (self-compose (self-compose s)) 3)
= ( (self-compose fourth-power) 3)
= (fourth-power (fourth-power 3))
= (16th-power 3)
= 43046721

Problem 4. Define a Scheme procedure abc-strings which applied to any positive integer argument, \( n \), will print out all the strings of length \( n \) over the alphabet \( \{a, b, c\} \) in alphabetical order.

SOLUTION: There are many nice ways to do this. Here’s one:

(define (print-abc n)
  (let ((putsuffix
         (lambda (pre)
           (lambda (post)
             (string-append pre post))))))
  (letrec ((abc-list (lambda (n)
                      (if (zero? n)
                          (list "")
                          (let ((n-1-list (abc-list (- n 1))))
                           (append
                            (map (putsuffix "a") n-1-list)
                            (map (.putsuffix "b") n-1-list)
                            (map (putsuffix "c") n-1-list))))))
    (for-each (lambda (str) (begin (display str) (display " ")))
      (abc-list n)))))

(print-abc 3)
aaa aab aac aba abb abc ac aca acb acc baa bab bac bba bbb
bbc bca bcb bcc caa cab cac cba cbb cbc cca ccb ccc
;Value: #\[unspecified-return-value\]