Problem 1 - $\lambda$-term abbreviations

Give the fully expanded $\lambda$-terms for the following $\lambda$-term abbreviations as discussed in class (lecture 3, page 4).

1. $(fgx)v$ w
2. $\lambda xy.((\lambda z.x(yz))v)w$
3. $\lambda x.xx \lambda x.xx$

Problem 2 - $\beta$-reductions

Show all possible $\beta$-reduction sequences for the $\lambda$-term

$$(\lambda xyz.(xz)(yz)) (\lambda xy.x) (\lambda xy.x)$$

Clearly mark the redex at each step

Problem 3 - Normal forms

1. Give three distinct $\lambda$-terms that reduce to the same normal form
2. Give three distinct $\lambda$-terms that do not have a normal form
3. Give a $\lambda$-term for which the normal form and the head-normal form are distinct.

Problem 4 - Church-Rosser property

Using CR-I as discussed in class (see lecture 3, page 17), prove that if a normal form exists for $\lambda$-term $M$, it has to be unique.
Problem 5 - Programming in lambda calculus

Logical constants and operations can be represented in lambda calculus as follows:

true ≡ \( \lambda a. \lambda b. a \)  
false ≡ \( \lambda a. \lambda b. b \)  
not ≡ \( \lambda x. ((x \text{ false}) \text{ true}) \)  
and ≡ \( \lambda x. \lambda y. ((x \ y) \text{ false}) \)  
or ≡ \( \lambda x. \lambda y. ((x \text{ true}) \ y) \)

Give the lambda calculus implementation of the boolean operations:

1. implication (\( a \rightarrow b \))
2. exclusive or (\( a \text{ exor } b \))

Problem 6 - let and let* in TINY

\( x \in \text{Variables} \)
\( n \in \text{Integers} \)
\( c ::= n \mid \#t \mid \#f \mid + \mid - \mid * \mid / \)  constants
\( v ::= c \mid (\text{lambda} \ (x\ldots) \ e) \)  values
\( e ::= v \mid x \mid (e \ e_1 \ldots e_k) \mid (\text{if} \ e_1 \ e_2 \ e_3) \mid \)  expressions
\( \quad (\text{let} ((x_1 \ e_1) \ldots (x_n \ e_n)) \ e) \mid \)
\( \quad (\text{let*}((x_1 \ e_1) \ldots (x_n \ e_n)) \ e) \)
\( p ::= e \)  program

How can the let and let* constructs be expressed in the TINY language? In other words, can you rewrite a let and let* with TINY language constructs? If so, specify how to do this.

Note: You are not asked to implement an interpreter extension for let and let*, i.e., implement the constructs within the interpreter itself. Instead, you can think of a “pre-processor” that translates the two constructs into TINY.
Problem 7 – Closures

(define test
  (lambda()
    (let* ((a 10)
           (b 11)
           (c 12)
           (foo (lambda(y)
                  (let* ((a 2)
                         (b (+ a y))
                         (f (lambda(x) (+ (* a x) b))))
                        f)))
      ((foo 3) a)))) ;; (*1*)

(define run (test));; (*2*)

To specify a closure, please use the following notation:
<< lambda abstraction , environment >>
where environment is a finite mapping from variables to values.
1. Give the closure value for foo on line (*1*)
2. Give the closure value for (foo 3) on line (*1*)
3. Give the closure value of test on line (*2*)
4. What is the value of (test) on line (*2*)?

Problem 8 - Typing

In lecture 4, we introduced type inference rules for the polymorphic cons function and the polymorphic empty list nil, together with their representation as type expressions. Note that our lists must have elements of the same type, which is not what the corresponding Scheme functions require.

1. Give the corresponding type inferences rules for polymorphic car and cdr, and their type expressions.
2. Use the inference rules form the lecture and your inference rules to construct the type of
   (a) (car (cdr (cons 1 (cons 2 nil))))
   (b) (lambda (x) (cdr (car x)))
   (c) (lambda (x) (cons (cdr x) nil))
Problem 9 - Unification

Lecture 4 introduces a simple type expression language consisting of type expression with basic types (char, integer, typeError) and composite types (pointer, array). Assume that your type language also includes type variables, represented as single literals (e.g.: 'a', 'b', ...).

Sketch a UNIFY function that takes two type expressions as input and produces the most general unifier, or reports that such a unifier does not exist. Recursive data types are not allowed.

Examples:

UNIFY( char, pointer(a) ) = ‘does not unify’
UNIFY( pointer(a), pointer(pointer(b))) = { a->pointer(b) }
UNIFY( array(1...a, pointer(char)), array(1...100, pointer(b))) =
  { a->100, b->char }
UNIFY( pointer(a), a ) = ‘does not unify’