1. (15 points) We parsed a let expression into a tree datatype that I call \textit{let-exp}. Suppose we have defined procedures

- constructor: \texttt{(new-let-exp ids vals body)} creates a new let-exp tree
- recognizer: \texttt{(let-exp? tree)} says if a tree node is a let-exp tree
- getters: \texttt{(getIds tree)} \texttt{(getVals tree)} \texttt{(getBody tree)} get the data out of a let-exp tree

Procedure \texttt{(eval-exp tree env)} is a big cases statement that breaks down a parse tree according to its node type. Write the case of \texttt{eval-exp} for a let-exp node. Then give a sentence in English that says how we evaluate a let expression.
2. (10 points) We talked in class about how to implement dynamic scoping; your MiniScheme interpreter implements static (also called lexical) scoping. Give any example that evaluates differently under dynamic scoping than under static scoping.
3) (15 points) You can answer each of the following questions in one sentence.

   a. What is call-by-value? Name any programming language that implements call-by-value.

   b. What is call-by-name?

   c. What is call-by-reference?
4. (20 points) Suppose you have a binary tree (every node has at most 2 children) where the internal nodes are symbols and the leaves are numbers, as in

```
  A
 / \ 
B   C
|   |
D   |
|   23
|   |
|   14
|   |
|   7
```

Suppose we have two datatypes defined:

- **treeNode** for internal nodes:
  - constructor (*new-tree-node* *left-child* *right-child*)
  - recognizer (*tree-node?* *tree*)
  - getters (*leftChild* *tree*) (*rightChild* *tree*)

- **leafNode** for the leaves:
  - constructor (*new-leaf-node* *val*)
  - recognizer (*leaf-node?* *tree*)
  - getter (*Val* *tree*)

Finally, suppose that a tree with only one child has the other child null.

Write procedure (*sum* *tree*) that returns the sum of the values of the leaves of such a tree.
5. (15 points) Here is the expression datatype we used for MiniScheme C:

(define parse
  (lambda (exp)
    (cond
      [(number? exp) (lit-exp exp)]
      [(symbol? exp) (varref-exp exp)]
      [(pair? exp) (app-exp (parse (car exp)) (map parse (cdr exp)))]
      [else (error 'parse "Invalid syntax ~s" exp)])
))

Write an unparse procedure for MiniScheme C. This takes a parse tree and returns the expression that was parsed to make the tree. For example (unparse (parse '(+ 3 (* 4 5)))) returns (+ 3 (* 4 5)) You can make up any names you like for the getter functions for the tree datatypes.
6. (10 points) Here is a grammar for a calculator language:

\[
E ::= E + T \mid E - T \mid T \\
T ::= T \ast F \mid T / F \mid F \\
F ::= \text{number} \mid (E)
\]

Add to this grammar an exponentiation operator \(^\), so \(3^4\) is 81. I want \(^\) to have higher precedence than + - \(*\) or / and to be right-associative, so \(2^3^4\) is the same as \(2^81\).

Note to the 2016 class: we spent more time on grammars in 2014 than we did this year.
7. (15 points) Remember the let* expression. This evaluates its bindings one at a time, so the expression  
(let* ([x 5] [y x]) (+ x y)) evaluates to 10. Write a function foo that takes such a let* expression and produces an equivalent let expression. 

(foo '(let* ([x 5] [y x]) (+ x y))) 
should return 

(let ([x 5]) (let ([y x]) (+ x y))) 
This lets you add let* expressions to MiniScheme by adding just one line to your parser.
You can use this page as extra space for any question.

Please write and sign the Honor Pledge before you hand in the exam.