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

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

Procedure \(\text{eval-exp tree env}\) is a big cases statement that breaks down a parse tree according to its node type. Write the case of eval-exp for a let-exp node. Then give a sentence in English that says how we evaluate a let expression.

\[
\begin{align*}
\text{eval-exp (let-exp? tree) (getIds tree) (getVals tree) (getBody tree))}
\end{align*}
\]

We evaluate a let expression in environment \textit{env} by evaluating its body in the environment we get from extending \textit{env} with bindings of the let’s \textit{ids} to the values of the corresponding expressions.
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.

(let ([x 5])
     (let ([f (lambda (y) (+ x y))])
         (let ([x 25])
             (f 6)))))

With static binding this produces 11, with dynamic binding 31.
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.

In call-by-value the argument values are passed to the procedure. Scheme, Java, C, etc. do this.

b. What is call-by-name?

In call-by-name the text (or parse trees) for the arguments are passed to the procedure.

c. What is call-by-reference?

In call-by-reference the addresses of the arguments, which must be variables, are passed to the procedure.
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

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.

Assuming all of that is already written, write a procedure sum that sums the leaves of such a tree.

(define sum (lambda (tree)
  (cond
    [(null? tree) 0]
    [(leaf-node? tree) (Val tree)]
    [else (+ (sum (leftChild tree)) (sum (rightChild tree)))])))
5. (15 points) Here is the expression datatype we used for MiniScheme C:

```scheme
(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.

```scheme
(define unparsed (lambda (tree)
    (cond
      [(lit-exp? tree) (lit-value tree)]
      [(varref-exp? tree) (var-symbol tree)]
      [(app-exp? tree) (cons (unparse (proc-part tree)) (map unparsed (args-list tree)))])
  ))
```
6. (10 points) Here is a grammar for a calculator language:

```
E ::= E+T | E-T | T
T ::= T*F | T/F | F
F ::= number | (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.

```
E ::= E+T | E-T | T
T ::= T*F | T/F | F
F ::= G^F | G
G ::= number | (E)
```
7. (15 points) Remember the let* expression. This evaluates its bindings one at a time, so the expression
\((\text{let*} \ ((x \ 5) \ [y \ x]) \ (+ \ x \ y))\) evaluates to 10. Write a function foo that takes such a let* expression and produces an equivalent let expression.

\((\text{foo} \ '(\text{let*} \ ((x \ 5) \ [y \ x]) \ (+ \ x \ y)))\)

should return
\((\text{let} \ ((x \ 5)) \ (\text{let} \ ([y \ x]) \ (+ \ x \ y)))\)
This lets you add let* expressions to MiniScheme by adding just one line to your parser.

\[(\text{define} \ \text{foo} \ \text{(lambda} \ \text{(let*-exp)})
\quad (\text{let} \ ((\text{bindings} \ (\text{cadr \ let*-exp})))
\quad \quad (\text{body} \ (\text{caddr \ let*-exp})))
\quad \quad (\text{bar} \ \text{bindings} \ \text{body}))))\]

\[(\text{define} \ \text{bar} \ \text{(lambda} \ \text{(bindings} \ \text{body})
\quad \text{(cond}
\quad \quad [[\text{null?} \ \text{bindings}} \ \text{body}]
\quad \quad \quad \quad \text{[else} \ (\text{list} \ \text{`let} \ \text{(list} \ \text{(car} \ \text{bindings}) \ \text{(bar} \ \text{(cdr} \ \text{bindings}) \ \text{body}))])])\]
You can use this page as extra space for any question.

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