CSSE 304 Assignment 11 Updated for Spring, 2016.

This assignment has only four problems, but three of them are non-trivial. Start early! **This is an individual assignment. No input error-checking is required.** You may assume that all arguments have the correct form.

No mutation is allowed in your solutions, except in problem 1.

Abbreviation for the textbook: EoPL - *Essentials of Programming Languages*, 3rd Edition.

#1 (30 points) `define-syntax` exercises

(a) Extend the definition of `my-let` produced in class to include the syntax for named `let`. This should be translated into the equivalent `letrec` expression.

```
(my-let fact ([n 5]) (if (zero? n) 1 (* n (fact (- n 1)))))  ➔ 120
```

(b) Suppose that `or` was not part of the Scheme language. Show how we could add it by using `define-syntax` to define `my-or`, similar to `my-and` that we defined in class. This may be a little bit trickier than `my-and`; the trouble comes if some of the expressions have side-effects; you want to make sure that no expression gets evaluated twice. In general, your `my-or` should behave just like Scheme's `or`.

```
  > (begin (define a #t)
           (define x (my-or #f (begin (set! a (not a)) a) #t (set! a (not a))))
           (list a x))
  (#f #t)
```

(c) Use define-syntax to define += , with behavior that is like += in other languages.

```
  (begin (define r 4) (define y (+ 6 (+= r 3))) (list r y)) ➔ (7 13)
```

(d) Recall that `(begin e1 … en)` evaluates the expressions `e1 … en` in order, returning the value of the last expression. It is sometimes useful to have a mechanism for evaluating a number of expressions sequentially and returning the value of the *first* expression. I call that syntax `return-first`. Use `define-syntax` to define `return-first`.

```
  > (define a 3) (begin a (set! a (+ 1 a)) a)  ➔ 4
  > (define a 3) (return-first a (set! a (+ 1 a)) a) ➔ 3
```

A note on testing problem 1 offline. Defining new syntax is very different than defining a procedure. Every time you reload your code for problem 1 into Scheme, you must subsequently reload the test code file. Can you see why this is necessary?

#2. (10 points) `bintree-to-list`. EoPL Exercise 2.24, page 50. This is a simple introduction to using `cases` and the `bintree` datatype (`bintree` definition is given on page 50). See notes below on using `define-datatype` and `bintree`.

#3. (40 points) `max-interior`. EoPL Exercise 2.25, page 50. The algorithm will be the same as before, but you will write it so that it expects its input to be an object of the `bintree` datatype. As before, you may not use mutation. You may not traverse any subtree twice (such as by calling `leaf-sum` on each interior node). You may not create an additional non-constant-size data structure that you then traverse to get the answer. Think about how to return enough info from each recursive call to be able to compute the answer for the parent node without doing another traversal.

Code to use for #2 and #3: Copy this code to the beginning of your file, or get it from http://www.rose-hulman.edu/class/csse/csse304/201630/Homework/Assignment_11/11.ss

```
;;  Binary trees using define-datatype
(load "chez-init.ss")   ; chez-init.ss should be in the same folder as this code.

;; from EoPL, page 50
(define-datatype bintree bintree?
  (leaf-node
   (num integer?))
  (interior-node
   (key symbol?)
   (left-tree bintree?)
   (right-tree bintree?)))
```

#4. (85 points) You should use the definitions in http://www.rose-hulman.edu/class/cs/csse304/201630/Homework/Assignment_11/parse.ss as a starting point.

Details of the exercise:

- Copy the given code; then modify the `expression` datatype, `parse-exp,` and `unparse-exp` so that they work for all of the expressions that were legal for the `occurs-free` and `occurs-bound` exercises in Assignment 10, and also for `letrec` and named `let`.
- Allow multiple bodies for `lambda`, `let` (including named `let`), `let*`, and `letrec` expressions. Also allow `(lambda x lambda-body …)` (note that the `x` is not in parentheses) or an improper list of arguments in a `lambda` expression, such as `(lambda (x y . z) …)`.
- Add `if` expressions, with and without the "else" expression;
- Add `set!` expressions.
- Expand the `expression` datatype to include `lit-exp`, which will be the parsed form for numbers, strings, quoted lists, symbols, the two Boolean constants, and any other expression that evaluates to itself. Then make `parse-exp` recognize these literals.
- Make `parse-exp` bulletproof. Add error checking to your `parse-exp` procedure. It should "do the right thing" when given *any* Scheme data as its argument. Error messages should be as specific as possible (that will help you tremendously when you write your interpreter in a later assignment). Call the `eopl:error` procedure (same syntax as Chez Scheme's `errorf`, whose documentation can be found at http://www.scheme.com/csug8/system.html#./system:s2) ; the first argument to `eopl:error` must be `'parse-exp`. This will enable the grading program to process your error message properly, i.e. to recognize that the error is caught and the error message is generated by <u>your</u> program rather than by a built-in procedure.
- Modify `unparse-exp` so it accepts any valid `expression` object produced by `parse-exp`, and returns the original concrete syntax expression that produced that parsed expression.
  Suggestion: when you modify or add a case to `parse-exp`, go ahead and make the corresponding change to `unparse-exp` and test both.

The grading program will have two kinds of tests for this problem:

1. Call `parse-exp` with an argument that is not a valid expression, then check to make sure that your program flags it as an error using `(eopl:error 'parse-exp …)`.

2. Call `(unparse-exp (parse-exp x))`, where x is a valid expression, and check to see if you get back the original expression. I will never directly compare the output of your `parse-exp` to any particular answer, since you have some leeway in what your parsed expressions look like. **Note:** It is possible to "pass" these tests by simply defining both procedures to be the identity procedure, so that you do not parse at all. This is clearly unacceptable.

Below or on the next page are some examples of what `parse-exp` might do. Your results from `parse-exp` do not have to be identical to mine, except that the error cases must call `eopl:error` with first argument `'parse-exp`, so that the output (in *Chez* Scheme) will begin with "Error in parse-exp". **The second example is not a sample test case**, since I stated that I will not call this procedure this way; it is simply intended to show what your procedure might produce. There is another example in the PowerPoint slides from the day when we introduced parsing (I think it will be Day 17 in Spring, 2016).

The output that I show in the second example is from *Chez* Scheme, where constructors based on `define-datatype` are transparent (you can see the contents of what they produce). The *Chez* Scheme records are much nicer for debugging than in other Scheme systems where records are opaque. **Your code that uses records must be representation-independent; you must use `cases` rather than `car` and `cadr` to access the fields of a record.**

```
> (parse-exp '(let ((w x y)) z))
Error in parse-exp: Invalid concrete syntax (let ((w x y)) z).

> (parse-exp '(lambda x (if (< x (* x 2)) #t "abc")))
(lambda-exp
  variable
  (x)
  ((if-exp
     (app-exp
       (var-exp <)
       ((var-exp x)
        (app-exp (var-exp *) ((var-exp x) (lit-exp 2)))))
     (lit-exp #t)
     (lit-exp "abc"))))
```

Note that the outer list surrounding the if-exp is because a lambda (or a let, let*, letrec) can have multiple bodies. This one has only one body, thus we have a list of one expression. Your output does not have to be the same as mine.

The symbol **'variable** in the parsed expression is to indicate that when this code is executed, it produces a procedure that can take a variable number of arguments because in the original code it is (lambda x …) rather than (lambda (x) …). This is one of several ways that you might handle that special case. Another approach is to have a separate data-type variant, **lambda-exp-variable**.

```
> (unparse-exp
    (parse-exp
      '((lambda (x)
           (if x 3 4))
         5)))
((lambda (x)
   (if x 3 4))
 5)
```

TO USE DEFINE-DATATYPE with petite *Chez* Scheme on your computer:

The `chez-init.ss` file should be in the same folder as your code. You can get it from: http://www.rose-hulman.edu/class/csse/csse304/201630/Homework/Assignment_11/chez-init.ss .
Include the line `(load "chez-init.ss")` at the beginning of your code. If your chez-init file is in a different location, you can do something like

```
(load "C:\Users\me\Documents\csse304\chez-init.ss")
```

TO USE DEFINE-DATATYPE with the Grading Server:

The `chez-init.ss` file is automatically loaded by the serve for assignments that need it, so you should not have to do anything special.