CSSE 304 Assignment 11

This assignment has only four problems, but three of them are non-trivial. Start early!
A11a is an individual assignment.
You must work with your Interpreter project partner for A11b. One partner should submit the assignment to the Gradescope server. **Make sure to add your partner in Gradescope as you submit.**
No mutation is allowed in your code, except for problem 1c.

#1 (30 points) `define-syntax` exercises. Your code may not involve mutation. But the test cases, may include code that includes mutation

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

Example:
```
(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`. You may not use `or` in your expansion of `my-or`.

Example:
```
> (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. Of course += will do mutation.

Example:
```
(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`.

Example:
```
(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 before running the tests. Can you see why this is necessary?

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

The `chez-init.rkt` file should be in the parent folder of your code (Homework/chez-init.rkt). You can get it from: https://raw.githubusercontent.com/RHIT-CSSE/csse304/main/Homework/chez-init.rkt .Include the line `(require "../chez-init.rkt")` at the beginning of your code. This all should be handled automatically. Let us know if it doesn't work.

TO USE DEFINE-DATATYPE with the Gradescope Server:

The `chez-init.rkt` file is automatically loaded by the server for assignments that need it, so you should not have to do anything special. Just don't modify where `chez-init.rkt` is (i.e. it should be in the parent folder)

#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`.

> **On Piazza (Fall, 2016) a student wrote**: I'm confused on how we are supposed to do this problem. Aren't we given the tree as a list already, so we would just return the list? Or do we need to make it a tree first and then convert it back to a list?
>
> **This is my answer:**
>
> This question strikes at the heart of an issue that I mentioned several times in class this week: representation-dependent *vs.* representation-independent code. For this problem (and every problem that involves define-datatype), you should always write representation-independent code.
>
> It happens that the Chez Scheme representation of datatypes uses lists, and is in the same form as the output required by this problem. So a representation-dependent version of `bintree-to-list` could be just the identity procedure. But if we transported that code to another Scheme system's define-datatype implementation, it might not work.
>
> So I want you to use `cases` to write a representation-independent recursive implementation of `bintree-to-list`. I will not give any credit for an "identity" implementation.

#3. (40 points) `max-interior`. EoPL Exercise 2.25, page 50. The algorithm will be the same as in a previous assignment, but you must write it so that it expects its input to be an object of the `bintree` datatype. As before, you may not use mutation. As before, 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/csse304/202010/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?)))
```

Problem #4 description begins on the next page.

Problem #4 is HW11b.

HW11b is a with-your-team assignment. You should not begin it until teams are established. In Winter 2020-21, teams should be set by the end of Friday, January 8.

#4. (85 points)

- 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 `#t` and `#f`, and any other expression that evaluates to itself. Then make `parse-exp` recognize these literals.
- Make `parse-exp` bulletproof. I.e., 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 `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 that you give to `error` for this problem 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 your 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. **No credit for this part unless your unparsed-exp is representation-independent (using `cases` instead of `car`, `cadr`, etc.)**

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 uses (`error 'parse-exp …`) to flag the input as an error.

2. Call `(unparse-exp (parse-exp x))`, where x is a valid expression, and check to see if your code returns something that is `equal?` to 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 `error` with first argument `'parse-exp`, so that the output (in Racket) will begin with "error: 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.

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. **But all of 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. The second example is only a sample of one way that the**

expression could be parsed. You may parse it any way you wish, as long as it contains sufficient information so that unparsed takes it back to the original Scheme code.

```
> (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)
```

Piazza Q&A from previous terms:

Is it important for us to understand the code written in the init.ss file for any future assignments / exams?

the instructors' answer,
No. You should not have to look into the details of chez-init.ss ay all

the students' answer,
You should look through it and find cases as it is recommended that we use it on this homework assignment.

the instructors' response,
Not *recommended*. *Required* to use cases for A11. But you still don't have to look at the details of chez-init.ss

Syntax not reloading? Environment not recognizing updated define-syntax

As I'm changing and updating my code, I found my environment doesn't seem to update the syntax that I've modified so every time I change my code, I have to reload scheme. I went to Connor when I couldn't figure out why my my-or wasn't working and he says it's most likely an environment thing. So in case other people are having trouble debugging, a possible solution could be reloading scheme.

Claude Anderson

I expect you don't really have to reload Scheme, but you definitely have to reload the test code file every time you change and reload your solution code for problem 1.

And you ought to be able to explain why that is necessary. Define-syntax is quite different than define.

A11B error checking: Does the error message have to match the error messages in the test case?

1 / 1	(parse-exp (quote (lambda x)))	(*error* parse-exp Error in parse-exp: lambda expression missing

It passes the test case, but I'm not sure whether it is correct.

the students' answer,

It doesn't have to match the test case. As long as you throw errors of the form (eopl:error 'parse-exp …) when (and only when) you should be throwing errors, the wording of the message doesn't matter, though the assignment advises that "Error messages should be as specific as possible (that will help you tremendously when you write your interpreter in a later assignment)." In this particular case, specifying the lack of body expressions is a helpful step up from the test case's "incorrect length" error, although I'd recommend adding the expression in question to the message using "~s" like the test case does.

the instructors' answer,

It has to say ‘parse-exp. the rest of the message does not have to match.

A11b: Named Let

I'm currently with a TA, and he pointed out how none of the test cases check for named let, but the assignment does mention that we need to have it. So do we actually need to implement named let for A11b, or is it just something that's optional, but will come in handy later?

the instructors' answer,

If it is not in the test cases, you do not have to worry about it now. But you will have to parse it for one of the interpreter assignments, so you might as well do it now.

A11b

Does the parser need to translate syntactic sugar like named let and let* into core forms, or should the expression datatype be expanded to contain these (and have the interpreter deal with them)?

hw11

the instructors' answer,

You do not need to do this for A11. In fact, it will be difficult for you to write unparse and pass the A11b tests if you do the translation now. The main theme of A14 will be syntax expansion.