CS 335 - Homework 4

Deadline:

Due by 11:59 pm on Friday, March 4

How to submit:

Submit your files for this homework using `~st10/335submit` on nrs-labs, with a homework number of `4`

Purpose:

Learning a bit about Scheme `let/let*/letrec`, practicing thinking about static versus dynamic scoping, thinking a bit about binding times for various attributes of names

The Problems:

Problem 1

Start a Scheme file `335hw4-1-2.scm` or `335hw4-1-2.ss`. The first thing in this file should be comments containing at least:

- your name
- `CS 335 - Homework 4 - Problems 1, 2`
- and the last-modified date

Now follow that with a comment containing `Problem 1`, followed by your answers for this problem.

Problem 1 Background

Lisp/Scheme can have local variables within an expression, including a function body (again, that we typically do not change once we have set them... 8-)) with the help of three special expressions, `let/let*/letrec`

It is pretty straightforward, although a bit tricky in terms of parentheses:

```
(let ((name1 expr1)
      (name2 expr2)
      ...
      (name-n expr-n))
   expr-in-let1
   expr-in-let2
   ...
   expr-in-let-n
)
```

Within the expressions `expr-in-let1 .. expr-in-let-n`, the names `name1 .. name-n` will have as their values the values of the expressions `expr1 .. expr-n` respectively.

As a very simple example, consider:

```
(let ((x 1)
      (y 2)
      (z 3))
   (list x y z))
```

The value of this expression will be `(list 1 2 3)`.

Why do we need `let*` and `letrec`, then? For this reason: `let` behaves as if its assignments of values to names occurs in parallel -- an assignment from "earlier" in the `let` cannot be seen "later" in the `let`. (Thus, the local definitions are **independent** of one another.) The following fragment demonstrates this:

```
(define x 27)

(let ((x 1)
      (y x))
   (+ x y))
```

What do you think the value of this expression will be? Actually, it is 28 -- try it!

(Why? Because the assignments are done as if in parallel, independent of one another -- x is assigned 1, but y is assigned to the value of expression `x` in parallel -- it can't/doesn't know that a "local" x has been set to 1. It can only see the "external" x of 27, and so y is set to 27, and 1 + 27 is added to get 28.)

If you would like the assignments within to be able to depend on the previous assignments -- which is definitely stepping further away from functional programming, I think! -- you'd use `let*`, which does make these assignments sequentially, and so able to depend on earlier assignments within the same expression. That is,

```
(let* ((x 1)
       (y x))
   (+ x y))
```

...does have the value of 2 that you probably expected earlier. y can be assigned to the just-assigned local x.

That leaves `letrec` -- why would you ever need that? Why, for mutually recursive definitions:

```
(letrec
    ((local-even? (lambda (n) (cond ((= n 0) #t)
                                    (else (local-odd? (- n 1))))))
     (local-odd? (lambda (n) (cond ((= n 0) #f)
                                   (else (local-even? (- n 1)))))))
  (list (local-even? 23) (local-odd? 23)))
```

Because `local-even?`'s lambda expression refers to `local-odd?`, and `local-odd?`'s lambda expression refers to `local-even?`, the only way this can work is with `letrec`.

Now -- why would you even bother with these? Most commonly when the same expression is used several times within a function:

```
(define (delete item a-list)
    (cond
        ((null? a-list) a-list)
        ((list? (car a-list))
            (cons (delete item (car a-list))
                  (delete item (cdr a-list))))
        ((equal? item (car a-list))
            (delete item (cdr a-list)))
        (else
            (cons (car a-list)
                  (delete item (cdr a-list))))
    )
)
```

`let` allows us to rewrite this replacing repeated expressions with a local name instead:

```
(define (delete item a-list)
    (cond
        ((null? a-list) a-list)
        (else
            (let ((rest-result (delete item (cdr a-list))))
               (cond ((list? (car a-list))
                         (cons (delete item (car a-list))
                               rest-result))
                     ((equal? item (car a-list)) rest-result)
                     (else (cons (car a-list) rest-result))))
        )
    )
)
```

...but in a less-functional setting, it is very handy when doing various kinds of input/output to be able to give a name to a newly-read-in-value within a function, or to give a name to a file input stream or file output stream, for example. If you remember that `(read)` has as its result whatever the user types in at that point -- what if you'd like to, say, do one thing to the value if it is a number, and something else if it is not? Then `let` makes that easier:

```
(define (interactive-react)
   (display "type in something: ")
   (let ((input-val (read)))
      (cond
          ((number? input-val) (* input-val input-val))
          (else (list input-val input-val)))
   )
)
```

(This is not pure functional programming here -- but no-argument functions are allowed in R5RS

Scheme, and you can run the above function by simply typing:

```
(interactive-react)
```

...!)

See if you can read, and reason out, the value of the following expressions. For each part, write a comment giving what you *think* the value of this expression should be, and then type this expression into DrRacket and see if you determined it correctly. (Yes, it is OK to change your comment if you got it wrong -- BUT I am expecting that you will think about it until you see why that is its value...)

1 part a

```
(let ((stuff    '(3 4 5))
      (nonsense '("ha" "byte")))
   (cons (car nonsense) stuff))
```

1 part b

```
(let ((x 3))
   (list x
         (let ((x (* x x)))
            (+ x x))
         x
         (let ((x (+ x x)))
            (* x x))
         x)
)
```

1 part c

```
(let ((x 17))
     (list
         (let ((x 2) (y (- x 2)))
              (* x y))
         x
         (let* ((x 2) (y (- x 2)))
              (* x y))
         x
         (letrec ((frick (lambda (x)
                           (cond ((< x 0) 0)
                                 (else (+ x (frack (- x 1))))))))
                  )
                  (frack (lambda (y)
                           (cond ((< y 0) 0)
                                 (else (* y (frick (- y 1))))))))
                  ))
                 (list (frick 3) (frack 3)))
         x
     )
)
```

Problem 2

Put a comment containing `Problem 2`, followed by your answers for this problem.

With the help of `let`, `display`, and `read`, design a no-parameter function `get-type-string` that expects nothing, but it uses `display` to print a request to type something in to the screen and then it uses `let` to give a local name to what the user then enters, then allowing the function to return a string that gives the name of the type of the value that the user entered.

Your function should check for at least the following: if the typed-in value is boolean, a number, a list, a symbol, or a string. (Yes, there do happen to be built-in predicate functions for each of these types... 8-)) You may check for other types IF you would like -- and if the type is none of the types you've checked for, return the string `"other"`.

Being interactive, automated tests of this are tricky! Put in at least one call to your function after you define it -- I'll just have to trust that you've really run it multiple times, and made sure to have tested each of its branches.

`335hw4-1-2.scm` is now ready to submit.

Problem 3

For the rest of the problems in this assignment, create a file `335hw4-rest.txt`, and type in their answers there. Then consider the following fragment of code, written in pseudocode:

```
int x = 0;
int y = 0;

void function thing1()
{
    int x = 0;
    print "in thing1\n";
    x = 7;
    y = 8;
    thing2();
    print "in thing1: " + x + "\n";
    print "in thing1: " + y + "\n";
}

void function thing2()
{
    int y = 0;
    print "in thing2\n";
    x = 2;
    y = 108;
    print "in thing2: " + x + "\n";
    print "in thing2: " + y + "\n";
}

x = 50;
y = 60;
print "here\n";
thing1();
print "and now\n";
print "at end: " + x + "\n";
print "at end: " + y + "\n";
```

3 part a

Type `3 part a` into your file. Then, assume that the above fragment uses **static scoping**, and type in what this fragment would print to the screen in that case.

3 part b

Type `3 part b` into your file. Then, assume that the above fragment uses **dynamic scoping**, and type in what this fragment would print to the screen in that case.

(And, by the way -- if you were to study the little Perl static and dynamic scoping examples posted from Week 6 Lecture 2, I believe you could actually determine how to convert the above fragment into two little Perl fragments to actually double-check your answers. You don't have to, but you could. If you do, though, (1) be sure to first figure out what the answers should be, to see if you are getting these concepts, and (2) if the answers are different, be sure to figure out why...)

Problem 4

Consider static and dynamic binding times, as discussed in lecture. If we limit ourselves to just the options of static binding time or dynamic binding time (that is, without all of the static binding time subcategories discussed), for each of the following, put the part number followed by which binding time is most appropriate:

4 part a

What is the binding time for the type attribute of a local C++ pointer variable?

4 part b

What is the binding time for the value attribute of a local C++ pointer variable?

4 part c

What is the binding time for the location attribute of a local C++ pointer variable?

4 part d

Fictitious language ElmTree has as part of its language definition that `pi` is a reserved word, whose value is the value of pi to 48 places. What is the binding time for the value attribute of the ElmTree name `pi`?

4 part e

What is the binding time for the type attribute of a Scheme variable?

4 part f

What is the binding time for the location attribute of a C global variable?