Daniel Opdahl
CS 370

PART A:

Because Scheme evaluates the arguments of a function call before the function itself, we can nest expressions inside of lambdas whose body is a single expression for begins that contain more than 1 subexpression. For example,

> (begin)

Contains zero subexpressions, and thus would be turned into my selected unspecified value:

> 'Nothing

For a begin with one subexpression,

> (begin (display 'a))

we take the subexpression and place it on its own:

> (display 'a)

For a begin with 2 or more subexpressions, we start to use nested lambda expressions. For example:

> (begin (display 'a) (display 'b))

Would be transformed into:

> ((lambda (moot) (display 'b)) (display 'a))

Where moot is a moot variable that is not used in the lambda expression. In this way, the result of evaluating > (lambda (moot) (display 'b)) is a procedure that can evaluate with > (display 'a) .
Every time we wish to add a subexpression to our begin expression (as long as the begin expression has 2 or more subexpressions), we simply take what we've already constructed and drop an expression of the form: > (lambda (moot) *subexpression*) in our most nested

lambda expression, in between the moot formal and the body of the lambda expression. For example, if we wanted to transform:

> (begin (display 'a) (display 'b)(display ‘c))

Into an expression without begins using my algorithm, it would proceed as follows:
Take out the first subexpression, in this case, it is > (display ‘a)

> (display ‘a)

Next, nest a lambda expression to the left of that subexpression as such:

> ((lambda (moot) (display 'b)) (display 'a))

Next, repeat, but place the lambda expression in between “(moot)” and “(display)”:

> ((lambda (moot) ((lambda (moot) (display 'c)) (display 'b))) (display 'a))

This technique will work for nested begins as well.

> (begin (display ‘a) (display (cons (begin ‘a) ‘())))

Becomes:

> ((lambda (moot) (display (cons 'a '()))) (display 'a))

And

> (begin (begin (begin )))

Becomes:

> ‘Nothing

And

> (begin (display 'a) (display (list (begin 'x 'y 'z))))

Becomes:

> ((lambda (moot) (display (list ((lambda (moot) ((lambda (moot) ‘z) ‘y)) ‘x)))) (display 'a))

PART B:

> (begin)

Becomes:

> ‘Nothing

Because there are 0 subexpressions in the begin expression.

> (begin x)

Becomes:

> x

Because there is exactly one subexpression in the begin expression.

> (begin y0 y1)

Becomes:

> ((lambda (moot) y1) y0)

Because there are more than one subexpressions in the begin expression we nest the subexpressions using a lambda expression.

> (begin z0 z1 z2)

Becomes:

> ((lambda (moot)((lambda (moot) z2) z1)) z0)

Because there are more than one subexpressions in the begin expression, we continue to nest the subexpressions using lambda expressions, building off of what we would have constructed for > (begin z0 z1).