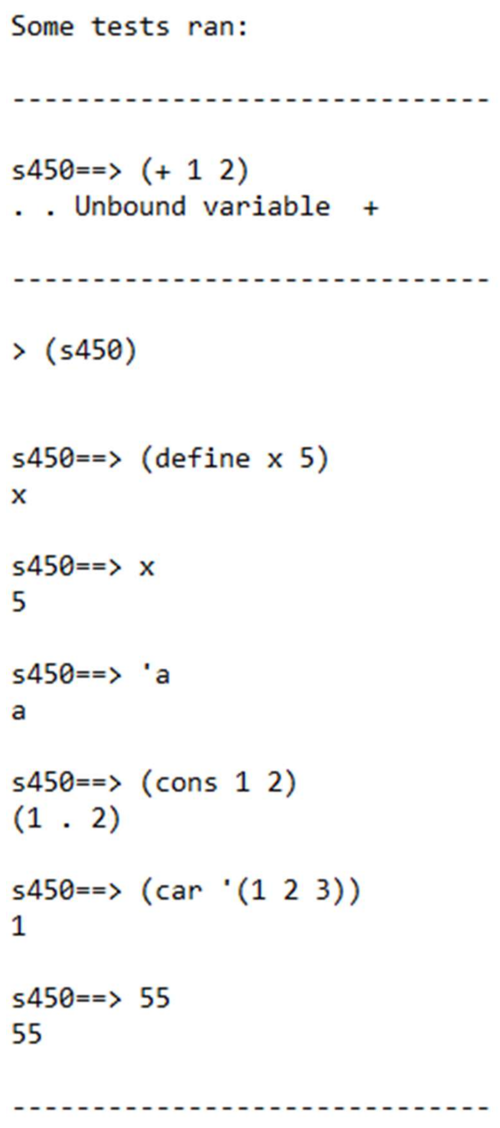
Question 1:

Played around with the default code. Tried out the preinstalled special forms and primitive procedures. Did not look too deep into how the code ran but just tested what it would output depending on the input. The preinstalled special forms and primitive procedures worked, anything not included would kick me out of the procedure.



```
Some tests ran:

------------------------------

s450==> (+ 1 2)
. . Unbound variable  +

------------------------------

> (s450)


s450==> (define x 5)
x

s450==> x
5

s450==> 'a
a

s450==> (cons 1 2)
(1 . 2)

s450==> (car '(1 2 3))
1

s450==> 55
55

------------------------------
```

Question 2:

In this portion I used the exact same table from HW4 for the special-forms-table. The lookup-action procedure is just lookup from HW4 but with the table already set to the special-forms-table. Same thing for install-special-form, it is just calling insert! from HW4. insert! Was modified a little so that when an existing entry is found in the table, it displays an error message instead of replacing the entry.

xeval was then rewritten exactly as the pseudocode described it.

```
let action = (lookup-action (type-of expression))
    if lookup succeeded
     invoke action, passing it the expression and the environment
     else cond ... ;; check for a few other types of expressions
```

Installation of new special forms was also written exactly the same as in the pdf.

```
==> (install-special-form ’set! (lambda (exp env) ... ) )
```

Question 3:

global environment:

```
 ____________________________________________
|                                            |
|  | car                                     |
|  |                                         |
|  | cdr       |                             |
|  |           |                             |
|  | cons      |          |                  |
|__|___________|__________|__________________|
   |           |          |
 *******    *******    ********
 * car *    * cdr *    * cons *
 *******    *******    ********
```

```
ORIGINAL WAY GLOBAL ENV IS INITIATED:
*** () is a list

(       ; GLOBAL ENV
        ; OTHER FRAMES WOULD BE HERE IF USER DEFINED PROCEDURES ARE CALLED

        (       ; FRAME
                ('car, 'cons, 'cdr)     ;LIST OF PRIMITIVE PROC NAMES
                (car, cons, cdr)        ;LIST OF PRIMITIVE PROC VALUES
        )
)

HOW PROFESSOR WANTS IT TO BE INTIATED:
*** () is a list

(       ; GLOBAL ENV
        ; OTHER FRAMES WOULD BE HERE IF USER DEFINED PROCEDURES ARE CALLED

        (       ; FRAME
                ()      ;LIST OF PRIMITIVE PROC NAMES
                ()      ;LIST OF PRIMITIVE PROC VALUES
        )
)
```

```
(install-primitive-procedure name action)     ; inserts name into LIST OF PRIMITIVE PROC NAMES and inserts
(install-primitive-procedure 'car car)        ; action into LIST OF PRIMITIVE PROC VALUES
(install-primitive-procedure 'cdr cdr)
(install-primitive-procedure 'cons cons)
```

This part took me a lot of time. I had to read and reread the pdf and the code many times. After tracing back the code starting from where the original primitive procedures were originally defined, I was able to figure out the general structure of the environment. This was sort of an eye-opening moment for me as it represented the "everything in scheme is a list" idea very well.

My pseudocode for the install-primitive-procedure is:

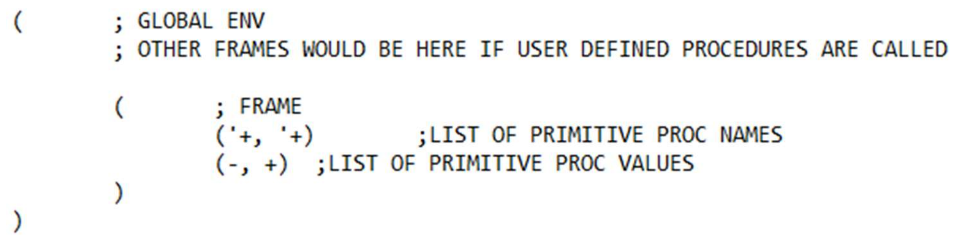
```
if name exists in special-forms-table
 throw error
 else scan the first frame in the global environment for the primitive function
      if primitive function is found, change associated value
      else insert new name and value into frame
```

The procedure actually works without checking for previous occurences of the primitive procedure being installed. For example:

```
> (install-primitive-procedure '+ +)
> (install-primitive-procedure '+ -)
```

would have + working like - without any problems. I did not get to the bottom of this, but I assume it is because of how the key value pairs are oriented in the frame. add-binding-to-frame! adds the key-value pair to the front of their respective lists so after the two statements above, the lists in the frame would

look like this:



```
(       ; GLOBAL ENV
        ; OTHER FRAMES WOULD BE HERE IF USER DEFINED PROCEDURES ARE CALLED

        (       ; FRAME
                ('+, '+)        ;LIST OF PRIMITIVE PROC NAMES
                (-, +)  ;LIST OF PRIMITIVE PROC VALUES
        )
)
```

I think that this will not cause any errors as when the evaluator looks or the body of the primitive procedure, it starts going through the list of already installed procedures from the left, thus when it lands on the first instance of the procedure in the list (which would be the latest installation of the procedure), it performs the operation based on the body linked to this procedure (which would be - for '+ since it was the last installation called for '+). Either way I scanned the frame for the procedure and replaced/updated it instead of just adding another one to the frame just in case.

Question 4:

This was just a matter of adding an additional condition to xeval. xeval first checks if the expression is a pair,

if it is not, check if it appears as a key in the table, if it does, display "Special Form: " and the expression.

Question 5:

To make sure than special forms cannot be overwritten, an if statement is added to definition to check whether the 2nd element in the expression is a key in the special-form-table, if it is then let the user know. Exactly the same was done with the eval-assignment procedure.

Question 6:

defined? loops through each fram in the environment and scans the car(list of defined variables) of that frame for the cadr(variable being looked up) of the expression. If found, return #t, otherwise, return #f.

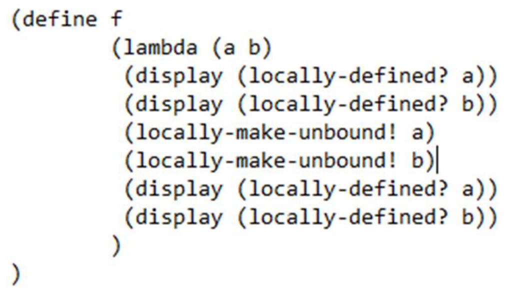
locally-defined? does the same thing as above but does not loop through the frames in the environment, only for the car(first frame) of the global environment.

locally-make-unbound! takes the car(list of variables) of the car(first frame) of the global environment and checks to see if the variable trying to be unbound is in the list, if it is, return it's index, otherwise return -1.

If index is not equalto -1, then set-car! the frame to itself minus the element at index and set-cdr! the frame to itself minus the element at index.

make-unbound! loops over all the frames in the environment and applies locally-make-unbound! to each frame.

Test cases for each of the 4 procedures:



```
(define f
        (lambda (a b)
         (display (locally-defined? a))
         (display (locally-defined? b))
         (locally-make-unbound! a)
         (locally-make-unbound! b)
         (display (locally-defined? a))
         (display (locally-defined? b))
        )
)
```

result was #t#t#f#f which should be correct as a and b defined in the frame and then unbound.

```
(define x 5)
(defined? x)
#t
```

which is correct as x is defined in the initial environment

```
(make-unbound x)
#t
```

I made the procedure return #t if the variable was found and unbound

```
(make-unbound y)        ; try to unbind a variable that was never defined
#f
```

I made the procedure return #f if the variable is not found and therefore cannot be unbound