Homework 6 Notes

Robert Steele

2. The first step I took in implementing install-special-forms was to build the lookup table. The lookup table was based heavily off the lookup table from previous homework. It has get and put functions which allow us to insert new special forms and retrieve the lambda expression associated with the name of a special form. It also has a print function to use for testing. To help interact with this table I wrote two helper functions, lookup-action and insert-special-form (not to be confused with install-special-form). After these functions and the lookup table were working properly I wrote install-special-form which takes the name of the function and a lambda expression. I also wrote two helper functions, the function special-form? which takes a name and performs a table lookup returning true if the name exists in the table and false otherwise and the function defined? which checks the-global-environment to see if a name is already defined. With these I could implement install-special-form. Install-special-form throws errors if special-form? or defined? is true and otherwise makes a call to insert-special-form.

3. Implementing install-primitive-procedure was very simple. I started by looking at primitive-procedures. The definition primitive-procedures was a list of lists with each inner list made up of two values, the name of the primitive, and the procedure associated with that name. To initialize the-global-environment we extend the-empty-environment with a new frame (the first frame of the global environment) and give it a list of names and a list of procedures produced by mapping car and cadr to primitive-procedures respectively. The end result is the-global-environment which has one frame (a pair) containing two lists, the first of which contains the names of the primitive procedures and the second which contains the primitive procedures themselves. To implement install-primitive-procedure we needed to first initialize the-global-environment. Because we no longer have a primitive-procedures definition we replace the lists generated from primitive-procedures in setup-environment with ‘() (the empty list). Now that the-global-environment is initialized it’s trivial to install a new primitive. There’s already a function, add-binding-to-frame, that does this for us. All install-primitive-procedure needs to do to install a new procedure is parse a name and procedure from exp and make a call to install-primitive-procedure passing it the name, procedure, and the-global-environment. This will bind the procedure to the name in the global environment. We also had to check if the name being installed is already a special form (which can be done with special-form?) and throw an error if it was. We also had to check if the primitive already existed and if it did then update the binding. I wrote two helper functions for this which came in handy later. The first is has-binding-in-frame? which takes two

arguments, a name and a frame, and returns true if the name appears in the frame and false otherwise. The second is set-binding-in-frame! which takes three arguments, a name, a value, and a frame, and replaces the current binding in the frame with the new value. In our case this works because we can check if a frame (the first frame of the global environment) has a binding for our procedure and if it does we simply update the binding. If it does not then we create a new binding.

4. Implementing this step was really straightforward. The first thing we do in xeval is a lookup. If the lookup fails (there is no special form in the table with the name given) xeval continues testing for other cases. All we had to do was make sure that (lookup-action (type-of exp)) returned false instead of throwing an error when encountering a special form outside any enclosing parenthesis. Once this section returns false we can check for the case where exp is a special form outside any enclosing parenthesis. We test this with (is-special-form? exp). If this returns true we know we are looking at a special form without enclosing parentheses and we can return the correct message by string appending "Special form: " and ((symbol→string) exp).

5. This problem was probably the easiest portion given what we've already built. Since we already have the function is-special-form? we can simply add checks in eval-definition and eval-assignment to make sure the name of the exp is not already a special form. If it is then we throw an error.

6. This was definitely the most fun and satisfying portion of the homework. I started with the special forms defined? and locally-defined?. Their special form functions were eval-defined and eval-locally-defined respectively. The simplest of these two was eval-locally-defined. All I had to do was pull the name from exp and call (has-binding-in-frame? name (first-frame env)). This returns true if the local frame contains the binding of that name. Eval-defined was similarly straightforward except this time we have to check all enclosing environments until we find the name or reach the empty environment. So I defined an inner loop, which iterates through all the enclosing environments and checks if that frame contains the variable and if it does we return true and stop. If we reach the empty environment then we return false. Eval-make-unbound and eval-locally-make-unbound. As usual the more straightforward of the two was locally-make-unbound! since we only have to worry about the first frame. For these two functions I wrote another helper function remove-binding-from-frame! which takes a name and a frame and uses two variables defined in the function (temp-vars and temp-vals) to keep track of bindings that don't match the name. When we reach an instance of the name we simply don't keep track of it and go on to the next recursive call. We stop when we have reached the end of the lists in the frame and we set the frames vals and vars to temp-vals and temp-vars. For eval-make-unbound we do the same thing only this time we iterate over all the enclosing environments as well.

First we check to make sure we haven't reached the-empty-environment, then we check if the current frame has the binding we are looking for. If it does then we call remove-binding-from-frame! and finish. If it doesn't then we recurse to the next enclosing-environment and check that frame. If we do reach the-empty-environment then we simply finish and haven't changed anything. To test these four special forms I wrote the following function which can be easily loaded and run in the s450 prompt.

```
(define f
        (lambda ()
                (display "Entered outer lambda")
                (newline)
                (display "Defining 'a' and 'b'...")
                (newline)
                (define a 10)
                (define b 20)
                (display "'a' defined in first lambda? (expect true) ")
                (display (defined? a))
                (newline)
                (display "'a' locally defined in first lambda? (expect true) ")
                (display (locally-defined? a))
                (newline)
                (display "'b' defined in first lambda? (expect true) ")
                (display (defined? b))
                (newline)
                (display "'b' locally defined in first lambda? (expect true) ")
                (display (locally-defined? b))
                (newline)
                (display "Unbinding 'a'...")
                (make-unbound! a)
                (newline)
                (display "'a' defined in first lambda? (expect false) ")
                (display (defined? a))
                (newline)
                (display "'b' defined in first lambda? (expect true) ")
```

```
(display (defined? b))
(newline)
(display "Defining 'a' again...")
(define a 10)
(newline)
(display "'a' defined in first lambda? (expect true) ")
(display (defined? a))
(newline)
(display "Locally unbinding 'a'...")
(locally-make-unbound! a)
(newline)
(display "'a' defined in first lambda? (expect false) ")
(display (defined? a))
(newline)
(display "'b' defined in first lambda? (expect true) ")
(display (defined? b))
(newline)
(display "Defining 'a' again...")
(define a 10)
(newline)
(display "'a' defined in first lambda? (expect true) ")
(display (defined? a))
(newline)
(define inner-lambda
        (lambda ()
                (display "entered inner lambda")
                (newline)
                (display "'a' defined in second lambda? (expect true) ")
                (display (defined? a))
                (newline)
                (display "'a' locally-defined in second lambda? (expect false) ")
                (display (locally-defined? a))
                (newline)
```

```
                    (display "Locally unbinding 'a'...")
                    (locally-make-unbound! a)
                    (newline)
                    (display "'a' defined in second lambda? (expect true) ")
                    (display (defined? a))
                    (newline)
                    (display "Unbinding 'a'...")
                    (make-unbound! a)
                    (newline)
                    (display "'a' defined in second lambda? (expect false) ")
                    (display (defined? a))
                    (newline)
                    (display "Exiting inner lambda... ")
               )
          )
          (display "Entering inner lambda... ")
          (inner-lambda)
          (display "entered outer lambda")
          (newline)
          (display "'a' defined? (expect false) ")
          (display (defined? a))
          (newline)
          (display "Exiting outer lambda... ")
          'success!
     )
)
```