Two additional requirements for homework assignment 5. Note that your score for the assignment 5 will largely depend on correctly implementing these two requirements. If you do not correctly implement both of these two requirements, your score for the assignment will be very low. If you correctly implement one of the two requirements, your score will be reasonably high. For this additional requirements, you can change envlookup and eval-under-env's existing two cases. However, you should be able to implement the requirements without changing those code.

(1) Implement global variable binding (glet) that overrides any existing binding in closures. The AST of glet is equivalent to the ast of mlet. Calling a closure within glet overrides any binding of free variables within the closure. The semantic of glet is similar to mlet except that it overrides the variable bindings for closure invocations. Checkout the following example.

```
(mlet "x" (int 42)
   (mlet "fun_a"
         (fun "funname" "arg1"
            (add (var "x") (var "arg1")))   ; mlet for var “fun_a” completed
      (glet "x" (int 10)
          (call (var "fun_a") (int 1)))))
```

In the above example, “fun_a” function is a closure with variable “x” bound to int 42. However, when calling the function, glet overrides the binding of variable “x”, and thus (int 11) is returned instead of (int 43). Check out the following examples.

```
(glet "x" (int 42)
   (mlet "fun_a”
         (fun "funname" "arg1"
            (add (var "x") (var "arg1")))   ; mlet for var “fun_a” completed
      (glet "x" (int 10)
          (call (var "fun_a") (int 1)))))
; ⇒ this example also returns (int 11)
```

```
(mlet "x" (int 42)
   (mlet "fun_a"
         (fun "funname" "arg1"
            (add (var "x") (var "arg1")))   ; mlet for var “fun_a” completed
      (mlet "x" (int 10)
          (call (var "fun_a") (int 1)))))
; ⇒ this example returns (int 43)
```

(2) Implement mutable integer array. You can use num-array, num-array?, num-array-at, and num-array-set for creating AST for array related expressions. Then when you evaluate an

expression including an array, you use num-array-object?, array-length, make-array-object, and set-array-val function. Check out the attached file for the source code of these functions and structure. Note that you have to use make-array-object to create an actual array object if you see num-array in the AST. The expression num-array-set returns the value that is being set.

e.g. (mlet “array1” (num-array 10)
        …)

In the above example, you have to create an array of 10 elements and then bind it to the variable “array1”.

e.g. (num-array-set (var “array-var”) 10 (int 42))

In the above example, you set the 10’th element of the num-array object referenced by “array-var” to be (int 32). The above expression should return (int 42). More examples are shown in the provided racket source code (in the comments).