Continuations – Theory

PART A

1.
   a. DEFINEGLOBAL rule :

$$\frac{\begin{array}{c}\ell \notin \mathrm{dom}\ \sigma \\ \langle e, \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto \text{unspecified}\}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \text{VAL}(x, e), \rho, \sigma\rangle \longrightarrow \langle \rho\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\}\rangle} \ (\text{DEFINE GLOBAL})$$

   b. µScheme program that detects whether val uses the Scheme semantics or the new semantics :

```
val x = 5
val f = ( lambda ( ) x)

val x = 15
(f)
```

Using the above program, we can differentiate between the old and new opsem. This program first sets a variable x to 5, it then defines a variable f to be an (anonymous) function lambda that returns the value in x. Then we set variable x to 15 and call f.

Using the old opsem, calling f would return 15, as the same variable x that was set to 5 initially gets reset to 15 before calling f.

Using the new opsem, calling f would return 5. This is because `(val x = 15)` would create a NEW variable x that would store 15 (in another location) and when we call f, it would remember the instance of x when it was defined, hence it would use the old variable x and return 5.

   c. With the new semantics where `val` always creates a new binding, variables are treated as immutable by default. It may result in more memory usage and potentially less efficient code because new bindings are created instead of modifying existing ones. I prefer Scheme's semantics for `val` because it allows us to modify existing bindings. It allows for in-place modification, which can be efficient in certain scenarios.