Assignment Part 9 – Scheme

# Program Testing Procedure

Run the shaker sort function with various inputs and observe the output. In all cases, the resulting list should be sorted in ascending order.

|  |  |  |
| --- | --- | --- |
| **Test name** | **Test input** | **Expected output** |
| Empty list | () | () |
| 1 element | (42) | (42) |
| 2 elements, in order | (1 2) | (1 2) |
| 2 elements, reverse order | (2 1) | (1 2) |
| Many elements, in order | (1 2 3 4 5 6 7 8 9 10) | (1 2 3 4 5 6 7 8 9 10) |
| Many elements, reverse order | (10 9 8 7 6 5 4 3 2 1) | (1 2 3 4 5 6 7 8 9 10) |
| Many elements, random order | (5 2 3 7 1 7 10 4 9 8) | (1 2 3 4 5 6 7 8 9 10) |

# Weekly Question

Scheme performs I/O via “ports”, which can be thought of as streams or handles. The port must be closed, so their lifetime is significant. This breaks regularity, as the lifetime of values/objects in functional languages is usually irrelevant (the concept of lifetime is not really needed). After being closed, ports can no longer be used, which also somewhat breaks regularity.

# Reflection

Scheme’s use of lists and atoms for both data and computations violates the syntactic consistency principle, reducing the readability of the program. It can be ambiguous whether any given piece of code is intended to be data or a computation.

Scheme’s use of lists for everything also violates the labelling principle. Computational expressions depend on the order of items in the list to be correct, with no alternatives available (no keyword arguments, for example).

Scheme’s ability to treat data and code the same allows first-class functions, which complies with the abstraction principle. The single\_pass function can be reused for both the forward and reverse sorting passes since the comparison function can be passed as a function argument.