1 Making Primitive Interpreter General and Robust (5 + 5 pts)

Consider the interpreter given in the file 3-5.scm. In this exercise, we generalize one primitive operation, and make another operation more robust by checking for potential errors.

1. Does the call (run "+(1,2,3,4)") return an error? If so, explain the problem. If not, what value does the call (run "+(1,2,3,4)") return?
   Modify the interpreter so that the outcome of running variable-arity calls such as (run "+(1,2,3,4)") is the same as that in Scheme such as for (+ 1 2 3 4).

2. What is the value of (run "sub1(12,3,4)") returned by the code in 3-5.scm?
   Modify the interpreter so that it behaves correctly for a single numeric argument, but returns 0 otherwise (that is, whenever the number of arguments is different from 1, or the argument is of incompatible type).

2 Adding a New Construct to the Language ([3 + 5] + 2 pts)

Consider the interpreter given in the file 3-5.scm. In this exercise, we explore adding and-construct to the interpreter with the following syntax and semantics described only informally.

\[ \text{and} \ (\text{exp1} \ \text{exp2} \ ... \ \text{expn}) \]

The and-construct begins with the keyword “and” and consists of parenthesis delimited, blanks separated, zero or more expressions. The meaning of this construct is obtained by evaluating each expression for a boolean value and then returning true iff all the expressions return true.

Discuss, and then make, all necessary changes to the interpreter to extend it with and-construct. Specifically, clearly locate the lines you are deleting/modifying/inserting using the line numbers given. Eventually, your modified interpreter should run programs such as

\( \text{(run "let x = 0 y = 1 in and (x 3 y)"}) \)

What additional test cases would you consider to improve faith in the correctness of your code?
3 Calculating using Axiomatic Semantics (3 + 3 pts)

Determine the following weakest preconditions. (Assume that all variables are of \textit{integer} type.)

\[
\text{wp( } \{ \text{if } i > j \text{ then } i := i - j \text{ else } j := i; \}, \ i = j) = ?
\]

\[
\text{wp( } \{ \text{while } i > 0 \text{ do } i := i - j; \}, \ (i = 0) \land (j = 2)) = ?
\]

4 ADT Specification (3 + 5 + 1 pts)

A \textit{sequence} is an ordered collection of values of the same type, possibly with duplicates. You are required to specify the generic ADT $\textbf{Seq}$ that supports the following operations: \textit{empty}, \textit{insert}, is\textit{Empty}, \textit{length}, and \textit{drop}. Informally,

- \textit{empty}: the empty sequence.
- \textit{insert}: Takes a sequence and a value as input, and yields the sequence resulting from introducing one occurrence of the value at the beginning of the sequence.
- is\textit{Empty}: Takes a sequence as input, and checks to see if it is empty.
- \textit{length}: Takes a sequence as input, and yields the number of values it contains.
- \textit{drop}: Takes a sequence and a number as input, and yields the sequence resulting from eliminating the given number of values from the beginning of the sequence. (That is, \textit{drop}([],5) = [], \textit{drop}([1,11,2,22,3,33],4) = [3,33], \textit{drop}([a,b,c],1) = [b,c], etc.)

1. Specify the signatures and classify the aforementioned operations on ADT $\textbf{Seq}$.
2. Give an algebraic specification of the semantics of ADT $\textbf{Seq}$.
3. Verify your specification by tracing the simplification of the term:

\[
\text{drop(insert(insert(insert(empty,3),4),5)), 2).
\]