Write the answers only in the space provided adjacent or after the question. Use the “blank” side(s) of each page for scratch work, especially to show evaluation details where appropriate. Be concise, precise, and legible.

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UID</td>
<td></td>
</tr>
<tr>
<td>Email</td>
<td></td>
</tr>
</tbody>
</table>
1 ADT Specification (3 + 2 + 3 + 2 pts)

A set is an unordered collection of values of the same type, with no significance attached to duplicates. That is, the list representation of a set \{1, 2, 3\} as [2, 3, 1], [1, 2, 2, 3], [3, 3, 1, 2, 1, 2, 3] are all equivalent, and creating any one of them is fine. The operations on the ADT ISet, the integer sets, are informally described below. (Assume Int is type name for integers and Bool is type name for booleans.)

- **empty**: Yields the empty set.
- **insert**: Takes an integer and a set as inputs, and yields the set resulting from introducing the integer into the set.
- **isEmpty**: Takes a set as input, and checks if it is empty.
- **isMember**: Takes an integer and a set as inputs, and checks if the integer belongs to the set.
- **union**: Takes two sets as inputs, and yields the set containing integers from both sets. (That is, union(\{1, 2\}, \{2, 3\}) = \{1, 2, 3\}.)
- **intersection**: Takes two sets as input, and yields the set containing integers common to both sets. (That is, intersection(\{1, 2\}, \{2, 3\}) = \{2\}.)
- **size**: Takes a set as input, and yields the number of distinct integer members. (That is, size(\{2, 3\}) = 2.)

1. Specify the signatures and classify the aforementioned operations on ADT ISet.
2. Write the axioms that `union` operation must satisfy.

3. Assuming that an `ISet` instance is represented using a Scheme list of integers, write a Scheme function (implementation) for carrying out the `intersection` operation. That is, assume `empty` is `()` and `insert` is `cons`. Trace your code for `(intersection '(1 1 3) '(1 3 6 3 7))` by writing out all but only the recursive calls made and results returned for these calls.
2 OOPL Code (2 + 6 + 2 pts)

Trace the execution of the following OOP program by providing (i) the initial state of the object in terms of its structure (class name, field names and field values) upon object initialization (e.g., O5->c1[x:0,y:0]) (ii) every method call and returned value if any (in one call per line format, identifying also the class of the method (e.g., send O5 m1() in c1 = 25), and (iii) the final state of the object. (Feel free to use the space to the right of the code below for your answer.)

(run
 "class c1 extends object
  field x
  field y
  method initialize ()
  begin set x = 111; set y = 222
  end
  method m1 () send self m2()
  method m2 () y
class c2 extends c1
  method m1 () send self m3()
  method m3 () super m1()
class c3 extends c2
  field y
  method initialize (xx)
  begin set y = xx;
    super initialize()
  end
  method m1 () 333
  method m4 (z) begin set x = +(y,z) end
let o3 = new c3 (444) in
begin
  send o3 m3();
  send o3 m4(555);
  o3
end"
)
3 Axiomatic Semantics (3 + 2 pts)

(a) Derive the weakest precondition:
\[ wp(\text{if } y < 0 \text{ then } x := y + 1 \text{ else } x := y;, \quad x > 0) = \]

(b) State the weakest precondition (assume: int x):
\[ wp(\text{while } (x \leq 5) x := x + 1;, \quad (x = 6) \text{ or } (x = 25)) = \]
Consider the following production rules for an (abstracted) expression grammar where $E$ is the non-terminal for expressions, $v$, $n$ and $o$ are terminals signifying variables, numbers and operators respectively.

$$
\begin{align*}
E & \rightarrow E \circ E \\
E & \rightarrow v \\
E & \rightarrow n
\end{align*}
$$

Our goal is to determine what expressions can be evaluated entirely at compile-time. In other words, expressions built entirely from operators and constants, and completely devoid of variables. (This helps in implementing constant folding optimization, which is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.)

- Is the grammar ambiguous? YES NO
- Should we rewrite the grammar to specify the solution? YES NO
- Write an attribute grammar to determine constant expressions. In particular, specify (i) attributes (with mode synthesized/inherited and data type), and (ii) attribute computation rules.
5 Modifying OOPL Interpreter (3 + 7 pts)

Read the given OOP Interpreter carefully. Note that it captures static scoping of free variables in a user-defined procedure by representing a closure via a procedure value rather than a record. In other words, it is similar to what you coded in Assignment 3. Answer the following questions, writing down the location of each change to be made to the code in terms of page number and line numbers, and the modified/replacement code.

- What is the scoping rule used for free variables of an instance method appearing in a class? How do we allow access to variables defined in the initial-env in an instance method? (In practice, this "global" environment contains predefined constants and functions (APIs).)
• The interpreter supports statically scoped free variables in a user-defined procedure. Make all the necessary changes to the interpreter code to support *dynamic scoping* instead.