1. Consider the RPN calculator program shown in Listings 1 and 2. For purposes of simplicity, we assume that all values of type \texttt{Int} cannot be \texttt{void}, so we can store them in a machine word instead of storing pointers. So for example, when passing \texttt{num} to \texttt{init}, the argument would be pushed directly onto the stack as a regular integer, rather than constructing a new \texttt{Int} object.

Suppose we are generating code for the AMD64 architecture, which has 8-byte words (so for this question, all pointers and integers are 8 bytes long), and the following calling convention, which resembles the convention given in lecture, is used:

Listing 1: RPN Calculator Program

```c
class Value { -- tag: 1
    value: Int;
    value(): Int { value };
    set(i: Int): SELF_TYPE { { value ← i; self; } }; // 5
};

class MulOp inherits Value { -- tag: 2
    operate(a: Int, b: Int): Int { a * b }; // 9
};

class AddOp inherits MulOp { -- tag: 3
    operate(a: Int, b: Int): Int { a + b }; // 12
};

class Stack { -- tag: 4
    head: Object;
    tail: Stack;
    get(): Object { head }; // 17
    pop(): Stack { tail }; // 18
    set(v: Object): SELF_TYPE { { head ← v; self; } }; // 19
    push(s: Stack): SELF_TYPE { { tail ← s; self; } }; // 20
};
```
Listing 2: Main class

class Main { -- tag: 5
  stack: Stack;
  reduce(): Int { let x : Object ← stack.get() in {
    stack ← stack.pop();
    case x of
      op : MulOp ⇒ let
        temp : Value,
        lhs : Int,
        rhs : Int in {
          rhs ← reduce();
          lhs ← reduce();
          temp ← (new Value);
          temp.set(op.operate(lhs, rhs)).value();
        };
      val : Value ⇒ val.value();
    esac;
  }
  init(num : Int): Object { { -- computes 2 * num + 1
    -- stack: [Value(2), Value(num), MulOp, Value(1), AddOp]
    -- where Value(x) means a Value whose value is set to x
    stack ← (new Stack).set((new Value).set(2)).push(stack);
    stack ← (new Stack).set((new Value).set(num)).push(stack);
    stack ← (new Stack).set((new MulOp)).push(stack);
    stack ← (new Stack).set((new Value).set(1)).push(stack);
    stack ← (new Stack).set((new AddOp)).push(stack);
  }
  main() : Object { let
    io : IO ← new IO,
    num : Int ← io.in_int() in {
      init(num);
      io.out_int(reduce());
      io.out_string("\n");
    }
  };
}
Calling convention:

- First, the frame pointer is pushed by the caller.
- Next, all arguments are pushed onto the stack in reverse order. Integers are stored directly as machine words. Since `self` is the first argument for all functions, it is always pushed last.
- The caller pushes the return address and jumps to the start of the function, using the `call` instruction.
- Now, the callee sets the frame pointer to the current stack pointer on entry to the function.
- Local variables are pushed onto the stack as necessary, following their scope in the program. For example, when calling `reduce()` when the top of the stack is a regular `Value`, only two values are pushed onto the stack (`x` and `val`), but when entering line 32, a total of 5 values are pushed onto the stack.
- When the callee returns, it resets the stack pointer to the frame pointer.
- The callee then pops the return address and jumps back to the caller at the return address, using the `ret` instruction.
- The caller restores the saved frame pointer.
(a) Suppose the dispatch tables are stored beginning at memory address 0x8000, in
the order Stack, Value, MulOp, AddOp, Main. Assume that Object has no
methods. The following is a representation of the dispatch table for Stack:

<table>
<thead>
<tr>
<th>Address</th>
<th>Line Number (Method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>(line 17) (get)</td>
</tr>
<tr>
<td>0x8008</td>
<td>(line 18) (pop)</td>
</tr>
<tr>
<td>0x8010</td>
<td>(line 19) (set)</td>
</tr>
<tr>
<td>0x8018</td>
<td>(line 20) (push)</td>
</tr>
</tbody>
</table>

Here, ⟨line x⟩ refers to the address of the generated code for line x. Also, the
ordering of the methods in the dispatch table are assumed to match the order
they are declared in the source code.

Using this format, provide dispatch tables for Value, MulOp, AddOp. (We will
assume for the remainder of the question that Main’s dispatch table is located at
0x8800.)

(b) Consider the state of the heap on entry to line 53. It is shown in the following
tables (ignoring IO):

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨object Main⟩ + 0x0000</td>
<td>5</td>
<td>(class tag)</td>
</tr>
<tr>
<td>⟨object Main⟩ + 0x0008</td>
<td>4</td>
<td>(object size)</td>
</tr>
<tr>
<td>⟨object Main⟩ + 0x010</td>
<td>0x8800</td>
<td>(dispatch ptr)</td>
</tr>
<tr>
<td>⟨object Main⟩ + 0x018</td>
<td>void</td>
<td>(stack)</td>
</tr>
</tbody>
</table>

Assume attributes are stored in the order they are declared in the program.

Since we do not know the precise heap layout, we specify heap addresses as an
offset from the start, according to the class name.

Give the heap layout after executing line 44. Hint: there is at most one object
for each class. When referring to other heap objects, you should reference them
by their class name, e.g. ⟨object Stack⟩.
(c) Consider the stack layout on entry to line 53. Suppose the return address of `main` is at address 0x2000, the initial value of the frame pointer is 0x7ffffff8, and the stack starts at address 0x77780000, so the first entry is at 0x7777fff8.

The following is a representation of the stack layout on entry to line 44, where the user entered the value 5:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7777fff8</td>
<td>0x7ffffff8</td>
<td>(saved frame pointer)</td>
</tr>
<tr>
<td>0x7777fff0</td>
<td>⟨object Main⟩</td>
<td>(argument 0 of <code>main</code>)</td>
</tr>
<tr>
<td>0x7777ffe8</td>
<td>0x2000</td>
<td>(return address of <code>main</code>)</td>
</tr>
<tr>
<td>0x7777ffe0</td>
<td>⟨object IO⟩</td>
<td>(local variable <code>io</code>)</td>
</tr>
<tr>
<td>0x7777ffd8</td>
<td>5</td>
<td>(local variable <code>num</code>)</td>
</tr>
<tr>
<td>0x7777ffd0</td>
<td>0x7777ffe8</td>
<td>(saved frame pointer)</td>
</tr>
<tr>
<td>0x7777ffe8</td>
<td>5</td>
<td>(argument 1 of <code>init</code>)</td>
</tr>
</tbody>
</table>

As shown above, return addresses specify the line number of the next instruction to be executed; since the caller needs to restore the frame pointer, this is the line number of the dispatch.

The call graph of `reduce` is shown in Figure 1. Give the stack layout on entry to line 11 (`AddOp.operate`), highlighted in red in the call graph. As before, you should refer to objects by their class name; you do not need to distinguish different objects of the same class. Hint: make sure to consider what the actual values of the local variables are.

Figure 1: Call graph for `reduce`, when `num = 5`. The stack is shown below each call to `reduce`.

```
reduce
[Value(2), Value(5), MulOp, Value(1), AddOp]

reduce
[Value(2), Value(5), MulOp]

reduce
[Value(2), Value(5)]

reduce
[Value(2)]

Value.value

Value.value

Value.value
```
2. Suppose we would like to add basic support for exception handling to Cool. We do this by introducing two new expressions:

\[
\text{try } e_1 \text{ catch } e_2 \text{ yrt throw}
\]

The behavior of these constructs is as follows: the `throw` expression produces an error and “halts execution”, resuming at the nearest `catch` expression. For `try` expressions, if \( e_1 \) executes normally without producing an error, its result is returned. Otherwise, if an error was produced, \( e_2 \) is executed and returned.

(a) Give the operational semantics for the `try` and `throw` expressions above. Also, show how the operational semantics for the \(+\) operator is changed to accommodate this. Hint: introduce a new value \( \bot \), which represents the result of an expression that produces an error.

(b) Show the evaluation of the expression below using operational semantics. You may omit the environments \( so, S, E \).

\[
\text{try } 2 + \text{ throw catch } 3 \text{ yrt}
\]

For reference, here is the proof tree for \( 2 \times \sim 1 \mapsto \text{Int}(-2)\):

\[
\begin{array}{c}
\vdash 2 \mapsto \text{Int}(2), S \\
\vdash 1 \mapsto \text{Int}(1), S \quad \text{[Int]} \\
\vdash \sim 1 \mapsto \text{Int}(-1), S \quad \text{[Neg]} \\
\vdash 2 \times \sim 1 \mapsto \text{Int}(-2), S \quad \text{[Arith]}
\end{array}
\]

(c) Simplicio wants to implement this feature using the following pseudocode (assume he has a separate scheme for method calls, which cannot be handled locally):

\[
\text{procedure CODEGEN}(e, \text{catchLabel})
\]

\begin{align*}
\text{if } e \text{ is } \text{try } e_1 \text{ catch } e_2 \text{ yrt then} & \\
\text{ourLabel} \leftarrow \text{fresh label} & \\
\text{ourEnd} \leftarrow \text{fresh label} & \\
\text{CODEGEN}(e_1, \text{ourLabel}) & \\
\text{emit jmp ourEnd} & \\
\text{emit ourLabel:} & \\
\text{CODEGEN}(e_2, \text{catchLabel}) & \\
\text{emit ourEnd:} & \\
\text{else if } e \text{ is throw then} & \\
\text{emit jmp catchLabel} & \\
\text{else} & \\
\ldots & \\
\text{end if} & \\
\end{align*}

What is wrong with his approach? Give an example of code that can produce undesired behavior and explain what happens.
3. Consider the following program, represented as a control-flow graph. $a$ and $b$ are the inputs to the program.

(a) Perform the following local optimizations on each basic block in order, and show the final result. You do not need to show the intermediate steps.

- Algebraic simplification (assume addition/subtraction is the fastest operation, followed by shifting, then multiplication.)
- Copy propagation
- Algebraic simplification again
- Constant propagation
- Algebraic simplification
(b) Notice that some optimization opportunities are missed by local optimization. We would like to perform global copy propagation by incorporating information from other basic blocks. When optimizing a basic block $B$, if we know that a variable $v$ is always a copy of another variable $u$ on entry to $B$, then we can use this information to propagate the copy to $v$. This analysis requires us to consider the program’s control flow. To do this, we store a mapping from each basic block and variable to a semilattice element\(^1\) representing this information. Then we can perform dataflow analysis, as with constant propagation, to compute a fixed point that consolidates the final state across all paths in the program.

We use the following semilattice structure (note: the convention used here is reversed from what is used in lecture; this follows what is used in the textbook.)

\[
\begin{array}{cccccc}
\top & & & & \downarrow \\
\text{a} & \bigtriangleup & b & \bigtriangledown & y & \downarrow \\
\downarrow & & & & & \downarrow \\
\bot & & & & & z
\end{array}
\]

, where $a, b, \ldots, y, z$ represent variables. We also define the meet (greatest lower bound) as follows:

\[
\begin{align*}
\top \land \top &= \top \\
\top \land a &= a \land \top = a \\
a \land a &= a \\
a \land b &= \bot \quad (a \neq b) \\
\bot \land \text{any} &= \text{any} \land \bot = \bot
\end{align*}
\]

The meet operator is used to combine information from a basic block’s predecessors: if we know the state of a variable $v$ on exit from every predecessor $P_1, \ldots, P_k$ of $B$ (and the corresponding semilattice elements are stored as $\text{Out}[P_i, v]$), then the state of $v$ on entry to $B$ is

\[
\text{In}[B, v] = \bigwedge_{\text{predecessors } P} \text{Out}[P, v].
\]

We also define a transfer function $f_B$ for each basic block as follows:

\[
\begin{align*}
f_B(\text{In}[v]) &= \text{In}[v], \text{ if } v \text{ is not modified in } B \\
f_B(\text{In}[v]) &= \bot, \text{ if } v \text{ is set to something that is not a copy} \\
f_B(\text{In}[v]) &= x, \text{ if } v \text{ is a copy of } x \text{ on exit from } B
\end{align*}
\]

\(^1\)The specific details of what a semilattice is are not relevant, except that you need a semilattice in order to perform dataflow, and that we have a meet (greatest lower bound).
To perform global copy propagation, we perform the following steps:

**procedure** CopyPropagation(CFG \( B_1, \ldots, B_n \), set of variables \( \vec{v} \))

assign \( \text{In}[\text{ENTRY}, v] \leftarrow \bot \) for every variable \( v \)
assign \( \text{Out}[B, v] \leftarrow \top \) for every basic block \( B \) and variable \( v \)
repeat
  for each basic block \( B \) do
    assign \( \text{In}[B, v] \leftarrow \bigwedge_{\text{predecessors } P} \text{Out}[P, v] \), for each variable \( v \)
    assign \( \text{Out}[B] \leftarrow f_B(\text{In}[B]) \)
  end for
until no values have changed
for each basic block \( B \) do
  for each variable \( v \) where \( \text{In}[B, v] \neq \bot \) do
    \( \triangleright \) note: \( \text{In}[B, v] \neq \top \)
    temporarily insert \( v := \text{In}[B, v] \) at the beginning of \( B \)
  end for
  perform local copy propagation
  remove the inserted copies
end for
end procedure

For example, for the following CFG:

```
ENTRY(a,n)
|
B1
x := a
i := i + 1
if i < n goto B1
else goto B2
|
B2
x := a + x
return x
|
EXIT
```

We perform the dataflow process as follows: after initialization (note that \( \text{In} \) is not initialized), we have

\[
\begin{align*}
\text{Out}[\text{ENTRY}, a] &= \bot & \text{Out}[\text{ENTRY}, i] &= \bot & \text{Out}[\text{ENTRY}, n] &= \bot & \text{Out}[\text{ENTRY}, x] &= \bot \\
\text{Out}[B1, a] &= \top & \text{Out}[B1, i] &= \top & \text{Out}[B1, n] &= \top & \text{Out}[B1, x] &= \top \\
\text{Out}[B2, a] &= \top & \text{Out}[B2, i] &= \top & \text{Out}[B2, n] &= \top & \text{Out}[B2, x] &= \top
\end{align*}
\]

Suppose we first visit \( B1 \). After we set \( \text{In}[B1] \):

\[
\begin{align*}
\text{In}[B1, a] &= \bot & \text{In}[B1, i] &= \bot & \text{In}[B1, n] &= \bot & \text{In}[B1, x] &= \bot
\end{align*}
\]
We then compute $\text{Out}[B1]$ according to the transfer function. Since $x$ is a copy of $a$, we set $\text{Out}[B1, a]$ to $a$. The full output of the transfer function is:

$$\text{Out}[B1, a] = \perp \quad \text{Out}[B1, i] = \perp \quad \text{Out}[B1, n] = \perp \quad \text{Out}[B1, x] = a$$

Next, we set $\text{In}[B2] = \text{Out}[B1]$, as B1 is the only predecessor of B2, and compute the transfer function:

$$\text{Out}[B2, a] = \perp \quad \text{Out}[B2, i] = \perp \quad \text{Out}[B2, n] = \perp \quad \text{Out}[B2, x] = \perp$$

In the next iteration, we must recompute the dataflow using the updated information from the previous iteration. In particular, since $\text{Out}[B1]$ has changed and B1 is a predecessor of B1, we should recompute $\text{In}[B1]$ and $\text{Out}[B1] = f_{B1}(\text{In}[B1])$. We now have:

$$\text{In}[B1, a] = \perp \quad \text{In}[B1, i] = \perp \quad \text{In}[B1, n] = \perp \quad \text{In}[B1, x] = a$$

Since $\text{Out}[B1]$ has not changed, $\text{In}[B2]$ and $\text{Out}[B2]$ are also unchanged, so we have converged. We can now use $\text{In}$ to perform constant propagation for each basic block. The results of dataflow as summarized below:

$$\text{In}[B1, a] = \perp \quad \text{In}[B1, i] = \perp \quad \text{In}[B1, n] = \perp \quad \text{In}[B1, x] = a$$

Using this, we find that B1 cannot be optimized, but we can optimize B2:

- (after inserting copies) $x := a$
- (after copy propagation) $x := a$
- (copies removed) $x := a + x \Rightarrow x := a + a$
- return $x$
- return $x$
- return $x$

Perform global copy propagation on the given CFG (without the local optimizations in part (a)). You do not have to show your work.

Hint: you should process blocks in the order given, as information propagate faster if the iteration order roughly follows the program’s control flow\(^2\). This can significantly reduce the number of iterations necessary for convergence.

Hint: you should only recompute the results for basic blocks for which $\text{Out}$ has changed for some predecessor. This can save a significant amount of work.

Note: you can also perform this optimization by inspection, if you are confident.

---

\(^2\)The optimal ordering for a forward dataflow problem is given by a reverse postorder (RPO) traversal of the CFG. For “rapid” dataflow problems (such as copy propagation, but not constant propagation), dataflow is guaranteed to terminate in $d + 2$ iterations, where $d$ is the ‘loop depth’ of the CFG, the maximum number of retreating edges in any cycle-free path.
4. Using the same CFG in Problem 3 (without any optimizations):

(a) Label each point in the program with the set of live variables.
(b) Show the interference graph. For ease of grading, also include a list of edges in
the form below:
   a - b;
   b - c;
(c) Give a minimal coloring for the interference graph. You should not use the algo-
   rithm in class, as this is not guaranteed to give a minimal result.