CS143 Written Assignment 4

Due: June 4th, 2024 EOD

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

```
1  class Value {  -- tag: 1
2    value: Int;
3    value(): Int { value };
4    set(i: Int): SELF_TYPE { { value ← i; self; } };
5  };
6
7  class MulOp inherits Value {  -- tag: 2
8    operate(a: Int, b: Int): Int { a * b };
9  };
10 class AddOp inherits MulOp {  -- tag: 3
11   operate(a: Int, b: Int): Int { a + b };
12 };
13
14 class Stack {  -- tag: 4
15   head: Object;
16   tail: Stack;
17   get(): Object { head };
18   pop(): Stack { tail };
19   set(v: Object): SELF_TYPE { { head ← v; self; } };
20   push(s: Stack): SELF_TYPE { { tail ← s; self; } };
21 };
```

Listing 2: Main class

```
23 class Main {  -- tag: 5
24   stack: Stack;
25   reduce(): Int { let x : Object ← stack.get() in {
26     stack ← stack.pop();
27     case x of
28       op : MulOp ⇒ let
29           temp : Value,
30           lhs : Int,
31           rhs : Int in {
32         rhs ← reduce();
33         lhs ← reduce();
34         temp ← (new Value);
35         temp.set(op.operate(lhs, rhs)).value();
36       };
37       val : Value ⇒ val.value();
38     esac;
39   } };
40   init(num : Int): Object { {
41     -- computes 2 * num + 1
42     -- stack: [Value(2), Value(num), MulOp, Value(1), AddOp]
43     --     where Value(x) means a Value whose value is set to x
44     stack ← (new Stack).set((new Value).set(2)).push(stack);
45     stack ← (new Stack).set((new Value).set(num)).push(stack);
46     stack ← (new Stack).set((new MulOp)).push(stack);
47     stack ← (new Stack).set((new Value).set(1)).push(stack);
48     stack ← (new Stack).set((new AddOp)).push(stack);
49   } };
50   main() : Object { let
51       io : IO ← new IO,
52       num : Int ← io.in_int() in {
53     init(num);
54     io.out_int(reduce());
55     io.out_string("\n");
56   } };
57 };
```

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`:

Address	Line Number (Method)
0x8000	⟨line 17⟩ (`get`)
0x8008	⟨line 18⟩ (`pop`)
0x8010	⟨line 19⟩ (`set`)
0x8018	⟨line 20⟩ (`push`)

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`):

Address	Value	Meaning
⟨object `Main`⟩ + 0x0000	5	(class tag)
⟨object `Main`⟩ + 0x0008	4	(object size)
⟨object `Main`⟩ + 0x0010	0x8800	(dispatch ptr)
⟨object `Main`⟩ + 0x0018	`void`	(`stack`)

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.

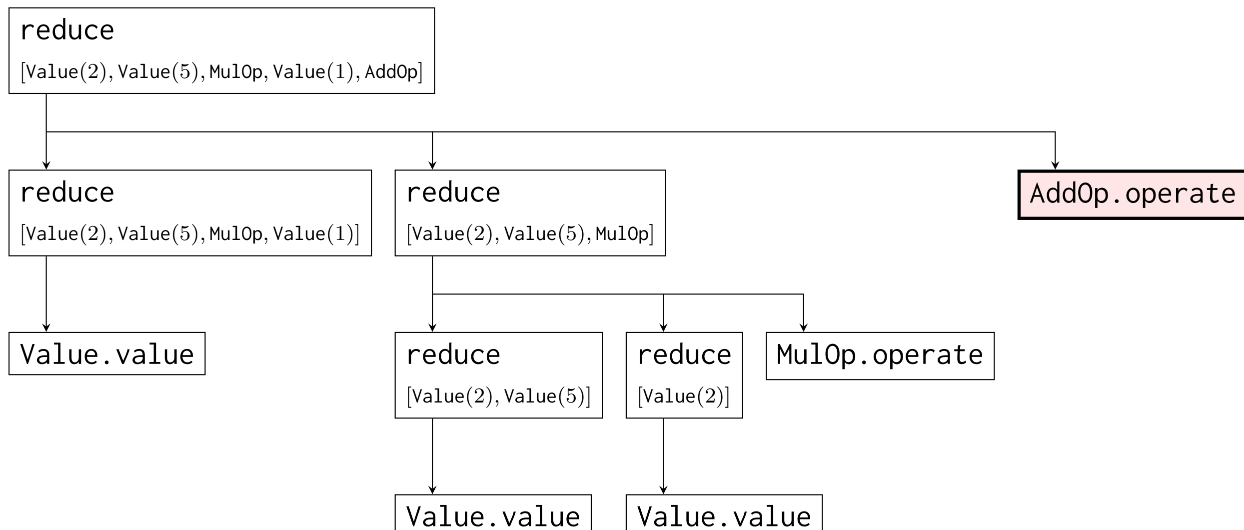
The following is a representation of the stack layout on entry to line 44, where the user as entered the value 5:

Address	Value	Meaning
0x7777fff8	0x7ffffff8	(saved frame pointer)
0x7777fff0	⟨object `Main`⟩	(argument 0 of `main`)
0x7777ffe8	0x2000	(return address of `main`)
0x7777ffe0	⟨object `IO`⟩	(local variable `io`)
0x7777ffd8	5	(local variable `num`)
0x7777ffd0	0x7777ffe8	(saved frame pointer)
0x7777ffc8	5	(argument 1 of `init`)
0x7777ffc0	⟨object `Main`⟩	(argument 0 of `init`)
0x7777ffb8	⟨line 53⟩	(return address of `init`)

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`.



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

$$\textbf{try } e_1 \textbf{ catch } e_2 \textbf{ yrt}$$
$$\textbf{throw}$$

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 accomodate this. Hint: introduce a new value $\perp$, 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.

$$\textbf{try } 2 + \textbf{throw catch } 3 \textbf{ yrt}$$

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

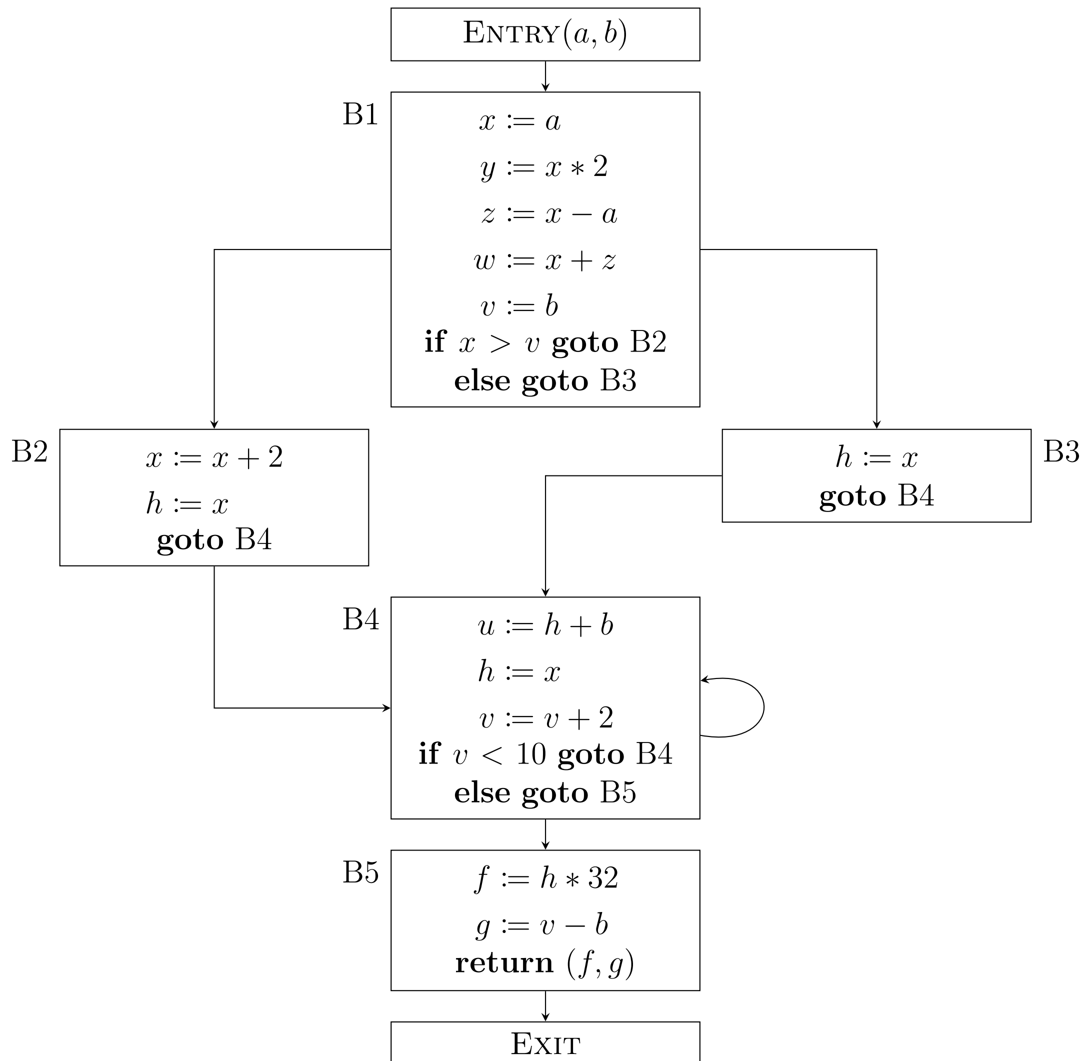
$$\dfrac{\dfrac{}{\vdash 2 \mapsto \text{Int}(2), S}\,[\text{Int}] \qquad \dfrac{\dfrac{}{\vdash 1 \mapsto \text{Int}(1), S}\,[\text{Int}]}{\vdash \sim 1 \mapsto \text{Int}(-1), S}\,[\text{Neg}]}{\vdash 2 * \sim 1 \mapsto \text{Int}(-2), S}\,[\text{Arith}]$$

   (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):

```
procedure CODEGEN(e, catchLabel)
    if e is try e1 catch e2 yrt then
        ourLabel ← fresh label
        ourEnd ← fresh label
        CODEGEN(e1, ourLabel)
        emit jmp ourEnd
        emit ourLabel:
        CODEGEN(e2, catchLabel)
        emit ourEnd:
    else if e is throw then
        emit jmp catchLabel
    else
        ...
    end if
end procedure
```

   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.

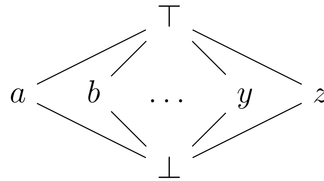


(a) Perform the following *local* optimizations on each basic block in order, and show the final result. You do not need to show the intermediate steps.

- 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.)



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

$$\top \wedge \top = \top \qquad \top \wedge a = a \wedge \top = a$$
$$a \wedge a = a \qquad a \wedge b = \bot \quad (a \neq b)$$
$$\bot \wedge \text{any} = \text{any} \wedge \bot = \bot$$

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 $\textsc{Out}[P_i, v]$), then the state of v on entry to B is

$$\begin{aligned} \textsc{In}[B, v] &= \textsc{Out}[P_1, v] \wedge \cdots \wedge \textsc{Out}[P_k, v] \\ &= \bigwedge_{\text{predecessors } P} \textsc{Out}[P, v]. \end{aligned}$$

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

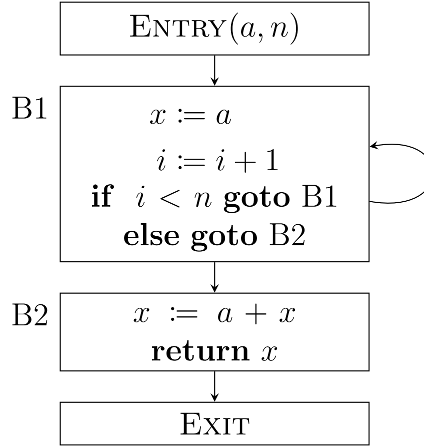
$$f_B(\textsc{In}[v]) = \textsc{In}[v], \text{ if } v \text{ is not modified in } B$$
$$f_B(\textsc{In}[v]) = \bot, \text{ if } v \text{ is set to something that is not a copy}$$
$$f_B(\textsc{In}[v]) = x, \text{ if } v \text{ is a copy of } x \text{ on exit from } B$$

[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 $B_1, \dots, B_n$, set of variables $\vec{v}$)
  assign IN[ENTRY, v] $\leftarrow \bot$ for every variable v
  assign OUT[B, v] $\leftarrow \top$ for every basic block B and variable v
  repeat
    for each basic block B **do**
      assign IN[B, v] $\leftarrow \bigwedge_{\text{predecessors } P}$ OUT[P, v], for each variable v
      assign OUT[B] $\leftarrow f_B$(IN[B])
    end for
  until no values have changed
  for each basic block B **do**
    for each variable v where IN[B, v] $\neq \bot$ **do**   ▷ note: IN[B, v] $\neq \top$
      temporarily insert $v :=$ 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:



We perform the dataflow process as follows: after initialization (note that IN is not initialized), we have

$$\begin{array}{llll} \text{OUT}[\text{ENTRY}, a] = \bot & \text{OUT}[\text{ENTRY}, i] = \bot & \text{OUT}[\text{ENTRY}, n] = \bot & \text{OUT}[\text{ENTRY}, x] = \bot \\ \text{OUT}[\text{B1}, a] = \top & \text{OUT}[\text{B1}, i] = \top & \text{OUT}[\text{B1}, n] = \top & \text{OUT}[\text{B1}, x] = \top \\ \text{OUT}[\text{B2}, a] = \top & \text{OUT}[\text{B2}, i] = \top & \text{OUT}[\text{B2}, n] = \top & \text{OUT}[\text{B2}, x] = \top \end{array}$$

Suppose we first visit B1. After we set IN[B1]:

$$\text{IN}[\text{B1}, a] = \bot \quad \text{IN}[\text{B1}, i] = \bot \quad \text{IN}[\text{B1}, n] = \bot \quad \text{IN}[\text{B1}, x] = \bot$$

We then compute $\textsc{Out}[\text{B1}]$ according to the transfer function. Since x is a copy of a, we set $\textsc{Out}[\text{B1}, a]$ to a. The full output of the transfer function is:

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

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

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

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

$$\begin{array}{llll} \textsc{In}[\text{B1}, a] = \bot & \textsc{In}[\text{B1}, i] = \bot & \textsc{In}[\text{B1}, n] = \bot & \textsc{In}[\text{B1}, x] = a \\ \textsc{Out}[\text{B1}, a] = \bot & \textsc{Out}[\text{B1}, i] = \bot & \textsc{Out}[\text{B1}, n] = \bot & \textsc{Out}[\text{B1}, x] = a \end{array}$$

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

$$\begin{array}{llll} \textsc{In}[\text{B1}, a] = \bot & \textsc{In}[\text{B1}, i] = \bot & \textsc{In}[\text{B1}, n] = \bot & \textsc{In}[\text{B1}, x] = a \\ \textsc{In}[\text{B2}, a] = \bot & \textsc{In}[\text{B2}, i] = \bot & \textsc{In}[\text{B2}, n] = \bot & \textsc{In}[\text{B2}, x] = a \end{array}$$

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

(after inserting copies)		(after copy propagation)		(copies removed)
$x \coloneqq a$		$x \coloneqq a$		
$x \coloneqq a + x$	$\Rightarrow$	$x \coloneqq a + a$	$\Rightarrow$	$x \coloneqq a + a$
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 $\textsc{Out}$ 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 algorithm in class, as this is not guaranteed to give a minimal result.