CS143 Final
Spring 2022

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, some with multiple parts. You have 180 minutes to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason other than to access the class webpage.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

SUNET ID: _____

NAME: _____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

Problem	Max points	Points
1	20	
2	20	
3	20	
4	20	
5	20	
TOTAL	100	

1. Local Optimization

Consider programs that consist solely of two types of instructions:

- Load: `x = y` for variable `x` and variable or constant `y`.
- Add: `x = y + z` for variable `x` and variables or constants `y` and `z`.

Assume that for each program, the only live variable on exit is the variable on the left-hand-side of the last instruction.

For each problem, give the program with fewest instructions that satisfies the stated conditions. You should apply the requested optimizations in the given order. For each optimization, you must continue to apply it until no further application is possible. We write CSE for common subexpression elimination, CP for copy propagation, and DCE for dead code elimination.

(a) Give a program with 5 or fewer instructions, where applying optimizations (i) results in a final program with more instructions than applying optimizations (ii). Write the two final programs obtained by applying (i) and (ii), respectively.

(i) CP $\rightarrow$ DCE.

(ii) CSE $\rightarrow$ CP $\rightarrow$ DCE.

Answer:

A minimal program has 3 instructions:

```
a = 1 + 2
b = 1 + 2
c = a + b
```

Applying (i) results in a final program with 3 instructions:

```
a = 1 + 2
b = 1 + 2
c = a + b
```

Applying (ii) results in a final program with 2 instructions:

```
a = 1 + 2
c = a + a
```

(b) Give a program with 6 or fewer instructions, where applying optimizations (ii) results in a final program with more instructions than applying optimizations (iii). Write the two final programs obtained by applying (ii) and (iii), respectively.

(iii) CP $\rightarrow$ CSE $\rightarrow$ CP $\rightarrow$ DCE.

Answer:

A minimal program has 4 instructions:

```
a = 1
b = 1 + 2
c = a + 2
d = b + c
```

Applying (ii) results in a final program with 3 instructions:

```
b = 1 + 2
c = 1 + 2
d = b + c
```

Applying (iii) results in a final program with 2 instructions:

```
b = 1 + 2
d = b + b
```

2. Generational Garbage Collection

In this problem, we will design parts of a *generational garbage collector*. The garbage collector of Cool uses this technique, as does most modern garbage-collected languages.

The generational hypothesis states that most allocated objects become dead quickly, while a few live for a long time. We can exploit this by dividing memory into two separate garbage-collected regions, called the *young generation* and the *old generation*. The high level algorithm is as follows:

1. Allocate new objects in *young*.
2. Whenever *young* is exhausted, garbage collect and move live objects into *old*.
3. When (and only when) *old* is exhausted, garbage collect *old*.

This algorithm can achieve better performance because most garbage collection happens on *young*, and when collecting *young* the algorithm does not traverse through *old*. Less often, a full collection on *old* occurs. In the following questions, you will flesh out the details.

Each of your answers must be justified. A couple of sentences will do. There is no need to write pseudocode or detailed proofs in any of your answers unless you find it helpful.

(a) For best performance, which garbage collection strategy (mark and sweep, stop-and-copy, or reference counting) should be used when collecting *young*?

Answer:

Stop-and-Copy. We need to move the objects into another memory region anyway, and since the work is proportional to live objects only we take advantage of the generational assumption. Other methods will do extra work processing dead objects.

(b) When collecting *young* objects, we first consider traversing the object graph starting with the objects reachable from registers and the stack. Describe why this strategy is incorrect and may collect objects that are still alive.

Answer: This strategy does not take into account that old objects may point to new objects. Although an older object cannot point at a newer object when it is created, because of attribute mutation it may later do so.

(c) In order to address the issue you found in the previous sub-question, the runtime must maintain an additional set of root pointers during program execution. Explain why it is necessary and sufficient to notify the GC every time an assignment to an attribute is made (as you did in PA4 using `_GenGC_Assign`), and explain what `_GenGC_Assign` must check regarding the current memory locations of the source and destination objects.

Answer:

The GC must keep track of *old* $\rightarrow$ *young* pointers. The `_GenGC_Assign` function checks whether the destination lives in *old* and the value lives in *young*. If so, then the value is added as a new GC root. No other language construct can create such pointers.

(d) We notice that a sizable fraction of objects that die quickly are nevertheless moved into *old* because they happen to be alive during their first garbage collection. How can we modify the *young* space and its collection procedure to reduce this problem?

Answer:

There are at least two solution to this problems:

- mark each pointer with one or more extra bits to record how many collections it has survived; move those that have survived two or more collections into *old* while scavenging *young*.
- explicitly subdivide *young* into two regions: *nursery* and *young*. During garbage collection, promote objects that survive the *nursery* to *young* and objects that survive *young* to *old*. This technique can also be applied with more than two regions.

3. Language Design

To enhance user privacy, a company that uses Cool decides to let programmers clear sensitive data when it is no longer needed. Their idea is to introduce a new expression:

$$\begin{aligned} \textit{expr} ::= & \ldots \\ & \mid \textbf{clear } \textit{expr} \end{aligned}$$

The **clear** expression works as follows. Given an expression **clear** e, the system first evaluates e. If e evaluates to `void`, then it does nothing; otherwise, it sets all attributes in the object to the default value for the type (e.g., `void` for objects, 0 for Int, etc.), thus destroying the sensitive data. The return value of **clear** e should be the value of e after the clearance is performed, and the static type of **clear** e is the same as that of e.

(a) Write down the type rule for the **clear** expression.

Answer:

$$\frac{O, M, C \vdash e : T}{O, M, C \vdash \textbf{clear } e : T} \text{ [Clear]}$$

Note: In hindsight, we aught to have forbidden **clear** from being used with Int, String, and Bool, since they are supposed to be immutable. Answers that explicitly remove primitive types are also correct.

(b) Write one or more formal *operational* rules for **clear** that match the above description of what the expression should do, in a style similar to the Cool Reference Manual. You may use without definition any shorthand defined in the Cool manual. In particular, the default value of type T is denoted D_T. Also, as an additional shorthand, you can access information about types and locations of attributes using the syntax:

$$v = X(a_1{:}\,T_1 = l_1, \ldots, a_n{:}\,T_n = l_n).$$

Answer:

$$\frac{so, S_1, E \vdash e \mapsto \text{void}, S_2}{so, S_1, E \vdash \textbf{clear } e \mapsto \text{void}, S_2} \text{ [Clear-Void]}$$

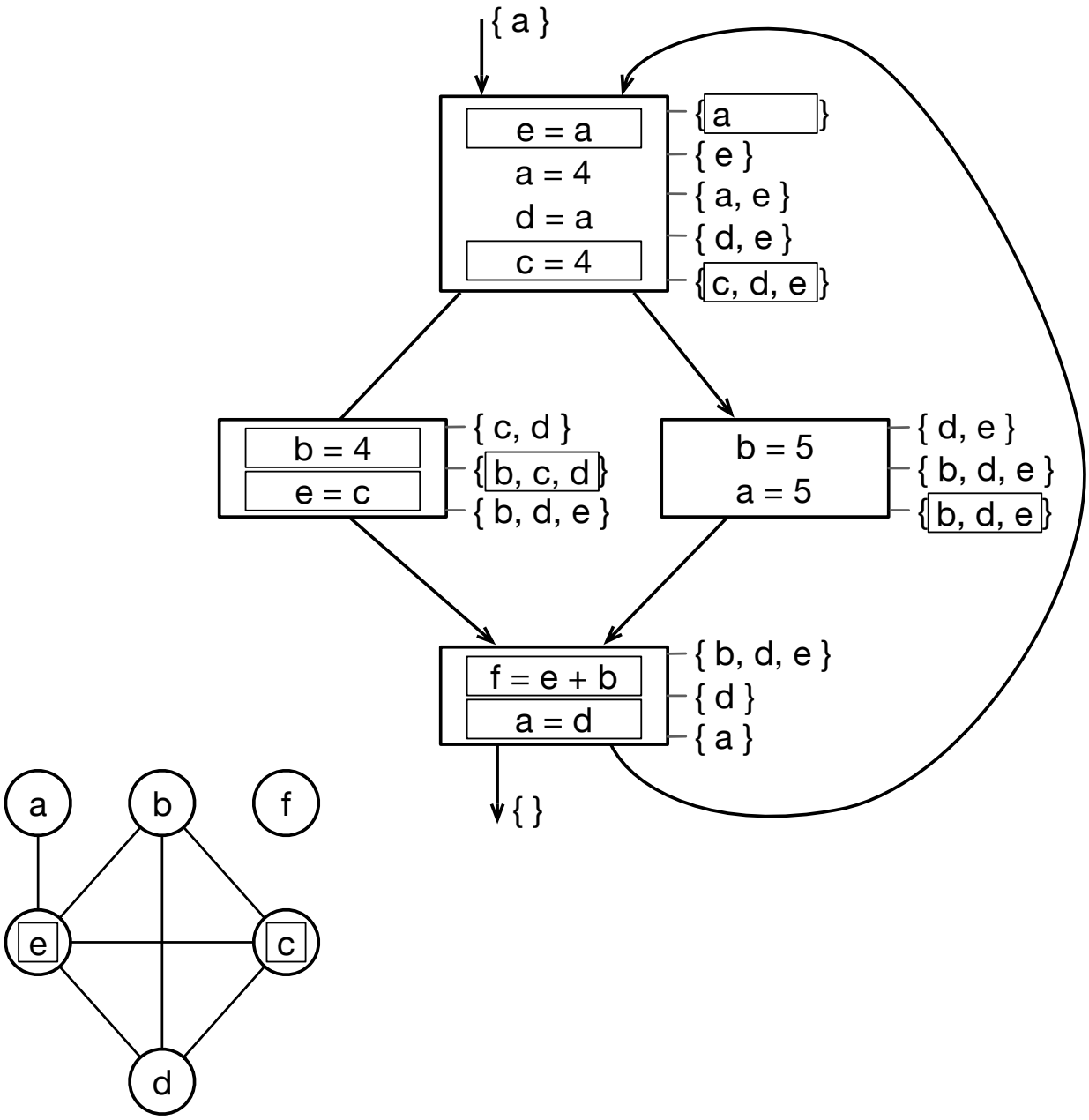
$$\frac{\begin{array}{c} so, S_1, E \vdash e \mapsto v, S_2 \\ v = T_0(a_1{:}\,T_1 = l_1, \ldots, a_n{:}\,T_n = l_n) \\ S_3 = S_2[D_{T_1}/l_1] \ldots [D_{T_n}/l_n] \end{array}}{so, S_1, E \vdash \textbf{clear } e \mapsto v, S_3} \text{ [Clear-Class]}$$

(c) Suppose Cool is implemented using a stop-and-copy garbage collector. How effective is the `clear` operator in destroying sensitive data?

Answer: The **clear** expression makes sensitive data no longer accessible to the Cool program and zeros out their references. But if an earlier stop-and-copy pass moved copied the object to the other half of the memory space, then it is possible for attributes to still be stored in the old memory.

4. Dataflow Analysis and Register Allocation

(a) You spilled coffee on your control-flow graph annotated with live variable sets and on your register inference graph. Please reconstruct the missing values by filling in the missing six statements, four live variable sets, and two register inference graph nodes. The missing statements are of the form `x = c`, `x = y`, or `x = y + z`, where `x`, `y`, and `z` are one of the variables `a`–`f` and c is a constant,



(b) What is the minimum number of register needed to avoid any register spills?

Answer:

Four registers (there is a clique $\{b, c, d, e\}$ of size four).

(c) Provide a register allocation by listing, for each register, the variables that share it.

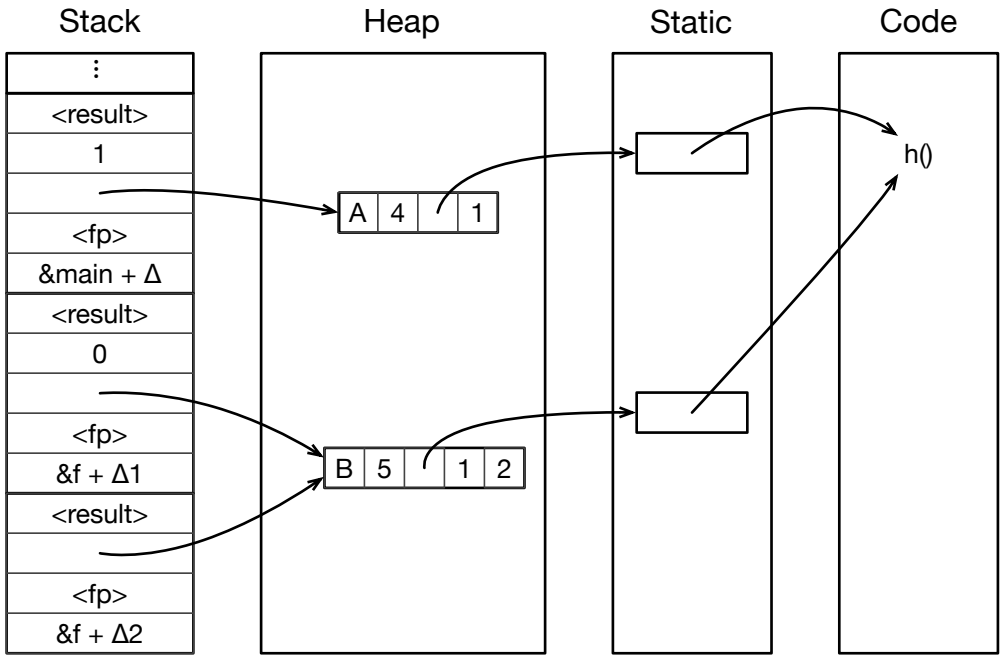
Answer:

Several register allocations are possible as long as the nodes in the clique $\{b, c, d, e\}$ are in different registers. One answer is:

```
r1: b,a
r2: c,f
r3: d
r4: e
```

5. Code Generation and Runtime Organization

Write a Cool program that at some point in its execution results in the following data on the stack, heap, static data, and code segments. Note that `<result>` denotes space set aside for the result of a method and that `&main + Δ` denotes some address inside the main function.



Answer:

```
class A {
    a : Int <- 1;
    h() : Object { self };
};

class B inherits A {
    b : Int <- 2;
};
```

```
class Main {
    main() : Object {
        f(1, new A)
    };

    f(i : Int, a : A) : Object {
        if i = 1 then f(0, new B) else g(a)
    };

    g(a : A) {
        a
    }
};
```