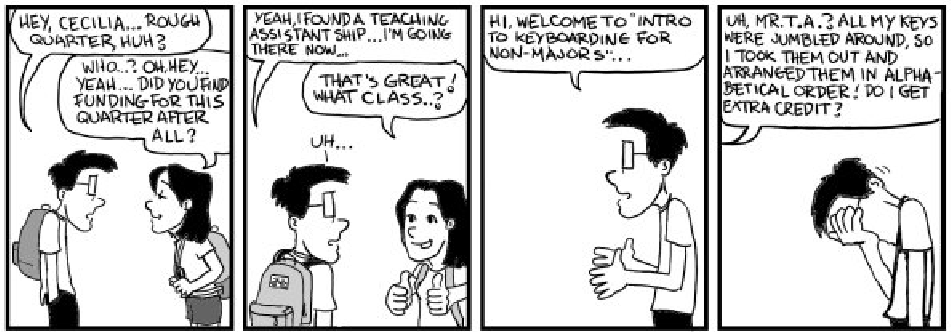
CS614: Advanced Compilers

Mid-Sem Exam (2 hours; 25 marks)

February 23rd, 2025

Instructions:

- Write neat, clear, short and TO-THE-POINT answers, with each major **Q** starting on a new page.
- In case you make an assumption, write it down before the corresponding answer.


HEY, CECILIA... ROUGH QUARTER, HUH?
WHO..? OH, HEY... YEAH... DID YOU FIND FUNDING FOR THIS QUARTER AFTER ALL?
YEAH, I FOUND A TEACHING ASSISTANTSHIP... I'M GOING THERE NOW...
THAT'S GREAT! WHAT CLASS..?
UH...
HI, WELCOME TO "INTRO TO KEYBOARDING FOR NON-MAJORS"...
UH, MR. T.A.? ALL MY KEYS WERE JUMBLED AROUND, SO I TOOK THEM OUT AND ARRANGED THEM IN ALPHABETICAL ORDER! DO I GET EXTRA CREDIT?

<u>Q1: EASY BUSY</u>

(A) Match the columns such that an entry in the second column *enables* an entry in the first column: [2]

(a) Type checking	(i) Liveness analysis
(b) Dead-code elimination	(ii) Constant folding
(c) Partial evaluation	(iii) Control-flow analysis
(d) Unreachable-code elimination	(iv) Symbol table

(B) An expression is *very busy* at a program point p if it is computed along each path from p (to the END) without getting invalidated. Write dataflow equations to compute very-busy expressions at each point in a program. Explicitly specify the analysis direction and the confluence operation. [2]

Now apply your dataflow equations to compute the set of very-busy expressions for the following program, in terms of IN and OUT sets for each statement. Write the sets clearly, preferably as a table. [2]

```
if (a > b) {
    x = b - a;
    y = a - b;
    z = a * b;
} else {
    y = b - a;
    x = a - b;
    a = x;
    z = a * b;
}
```

Given the set of very-busy expressions, what optimization can you perform? Write a short algorithm to perform that optimization as a compiler pass and show the result of applying it on the above program. [2]

Q2: JAYA JAVA

(A) Here is a (partial) context-free grammar that describes the structure of this question paper:

```
<Goal> --> <Header> <Instructions> <Question>* <Footer>
<Header> --> <Course> <Exam> DATE
<Course> --> <Code> <Title>
<Exam> --> <Name> <Duration> TOTAL_MARKS
<Instructions> --> <Instruction>* <Comic>
<Question> --> <SubQuestion>* <HRule>
<SubQuestion> --> <Text> MARKS
<Footer> --> <Activity> <Clever_Quote>
```

Similar to your assignments, write code for a `visit` method corresponding to each relevant non terminal, such that you can store the marks per question and check that they sum up to `TOTAL_MARKS`. Ignore the exact syntax of visitor pattern, add attributes (fields) as needed, and mention any assumptions that you make. [3]

(B) Recall how fields are laid out in Java (the jol examples). Assume the following number of bytes per type:

boolean	1 byte
char	2 bytes
int	4 bytes
long	8 bytes
reference	4 bytes

As the compiler, our goal is to minimize object sizes. We are free to order fields within a class but all the fields of a class can be laid out only after those of its parent. Object headers on our machine are 12 bytes, and objects can be laid out only at 8-byte multiples. Draw the object structure for the following classes, explicitly mentioning the amount of internal and external padding, and the total size of the object instances. [3]

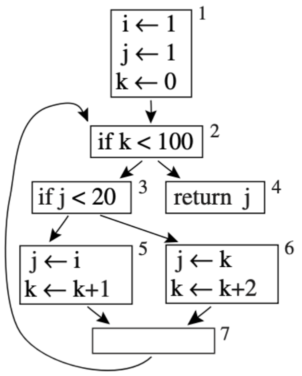
```
class C {
    boolean f1;
    int f2;
    E f3; // assume class E exists
}
```

```
class D extends C {
    boolean f2;
    long f4;
    char f5;
}
```

(C) Mention the five method-invocation Java bytecodes, and arrange them in the increasing order of expected run-time cost. Briefly justify your order. [3]

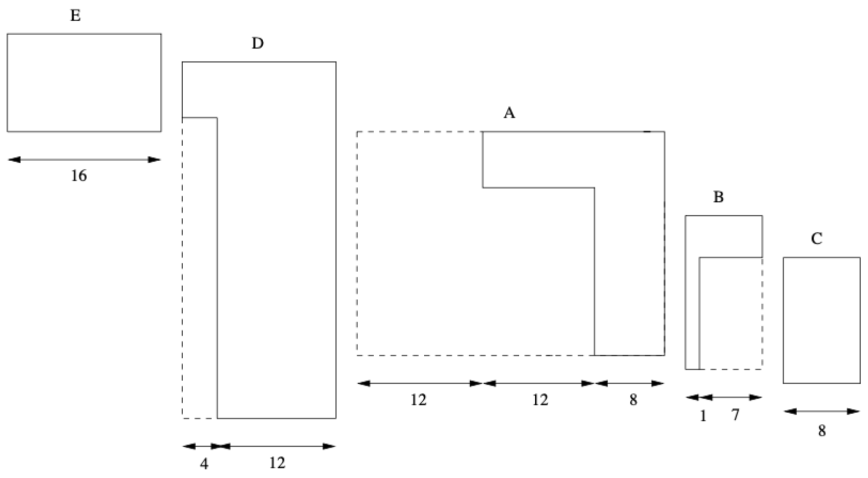
Q3: ASPECTO ABBREVIANTUM

(A) Convert the following control-flow graph to SSA form: [2]



(B) Draw the updated CFG (final one) after performing the following transformations over the SSA form in order: (sparse) conditional constant propagation, unreachable-code elimination, dead-code elimination. [3]

(C) Recall how we were able to reduce the required number of registers from four to two using Tallam-Gupta's bitwidth-aware register allocation (BARA), compared to standard graph coloring (GCRA), for the program in the class (bitwise live ranges obtained using DBS analysis are reproduced below). Also recall that we had discussed ways of reducing the number of required registers to even just one (the defragmentation approach, as well as Anirudh's idea of estimating maximum widths more precisely).



Kavya tries to perform BARA for his programs slated to run on *Mango Watch* (an upcoming smartwatch being developed by CS614 students), and concludes that set aside Anirudh's approach, even the original Tallam-Gupta algorithm that gives two registers does not terminate for his programs in a reasonable time. To mitigate this scalability challenge, Anamitra comes with a new proposal:

> Instead of maintaining bitwidth contributions with edges, maintain them with nodes. Each node has a label that tells the maximum bitwidth required for that variable across the entire program. Coalesce two nodes if their labels indicate that they can be put together in a 32-bit register.

Draw the interference graph using Anamitra's approach, and iteratively coalesce its nodes to determine the number of required registers. Finally, comment on the scalability-precision tradeoff among the various RA approaches we have learnt, so that it helps Team Kavya launch *Mango Watch* in the market before our neighbor surprises the world with a *DF*[1] *Watch*. [3]

HAPPINESS ISN'T A SLAVE TO MARKS

If you are enjoying this course, draw a smiley on the board while leaving :-) [0]

[1]If you can guess the correct full form of "DF", you will get a PC (not a watch).