Answer Key – Computer Science & Engineering 505 Midterm

November 1, 1999
Open book & notes – 50 minutes – 10 points per question
50 points total

1. Consider the following program in an Algol-like language.

```
begin
integer g;
procedure clam(j,k: integer);
    begin
    print(j,k);
    g := g+1;
    print(j,k);
    end;
g := 1;
clam(2*g, g);
end;
```

What is the output if `j` and `k` are both passed by:

(a) call by value
2 1
2 1

(b) call by name
2 1
4 2

(c) call by reference
2 1
2 2

(Assume that having an expression as the actual parameter when passing by reference is legal. The compiler generates a temporary for the result of evaluating the expression, which is then passed by reference.)

2. Suppose that you write a random number generator in Algol-60, and use an `own` variable in the procedure `random` to hold the seed. Suppose also you want to be able to initialize the seed to a given value, so that the random number generator will generate the same values for each run of the program for testing purposes. How would you initialize the seed? Discuss the concept of `own` variables in light of your answer. If they worked well for this usage, say so; if they didn't work well, discuss alternatives.

   An `own` variable seems like it ought to be a good mechanism to hold the seed for the random number generator, since it will retain its value between invocations of the random number generator and is not visible outside the scope of the `random` procedure.

   Unfortunately, initializing the seed is not convenient. The seed can only be accessed from within the `random` procedure (or a procedure nested inside of `random`). There isn't a way to specify an initial value as part of the declaration. So the programmer would need to have a global boolean `initialized` flag, which is initially set to false. Within `random`, test `initialized`. If it is false, initialize the seed, and set `initialized` to true. However, using a global variable for `initialized` defeats the purpose of hiding the seed — we might as well just make the seed be a global.

   Note that we can't make the `initialized` flag be local to the `random` procedure, since otherwise we still have the same problem of initializing `initialized` — it must be global.

   This indicates that the concept of `own` variables has severe limitations, due to these initialization problems. A simple solution is to allow initial values to be specified in declarations. A comprehensive solution, available in object-oriented languages, is to define a random number generation object, whose constructor or initialization method sets the seed.

3. Consider the following linear programming problem.

   minimize x
   subject to
   $4 \leq x$
   $x \leq 10$

   As usual, x is constrained to be non-negative.

   We first eliminate the inequality constraints by adding slack variables s_1 and s_2:

   $4 + s_1 = x$
   $x + s_2 = 10$

   After the first phase of the simplex algorithm, we might obtain the following basic feasible solution:

   $s_1 = 6 - s_2$
   $x = 10 - s_2$

   (a) What is the solution given by this tableau? $x = 10, s_1 = 6, s_2 = 0$

   (b) What is the value of the objective function? 10

   (c) Starting from this tableau, show how you obtain an optimal solution.
   The objective function is $10 - s_2$. So we can decrease the value of the objective function by making s_2 positive. The value of s_2 is limited by both equations in the tableau, and $s_1 = 6 - s_2$ is more limiting. Solve this for s_2 and substitute in the other equation and objective function to get
   $s_2 = 6 - s_1$
   $x = 4 + s_1$
   The objective function is $4 + s_1$. Since the coefficient of s_1 is positive, increasing its value will just make the value of the objective function larger, so we're done. The solution is $x = 4, s_2 = 6, s_1 = 0$.

4. Our implementation of CLP($\mathcal{R}$) uses an incomplete solver. Call this one "Standard CLP($\mathcal{R}$)". Suppose that we had another implementation, called "Complete CLP($\mathcal{R}$)", that has a complete solver. Both Standard and Complete CLP($\mathcal{R}$) try rules in the same order, and select the leftmost literal in a derivation.

(a) Are there goals for which Standard CLP($\mathcal{R}$) returns an answer, and for which Complete CLP($\mathcal{R}$) says no? Why? Give an example if one exists.

Yes - if Standard CLP($\mathcal{R}$)'s solver answers unknown, but the constraints are in fact unsatisfiable, then Standard CLP($\mathcal{R}$) will say `maybe` and Complete CLP($\mathcal{R}$) will say `no`. An example of this occurs for the goal

```
X*X*X=10, X<0.
```

(b) Are there goals for which Complete CLP($\mathcal{R}$) returns an answer, and for which Standard CLP($\mathcal{R}$) says no? Why? Give an example if one exists.

No. If Complete CLP($\mathcal{R}$) returns an answer, since its solver is complete, the constraints can in fact be satisfied for some derivation. So Standard CLP($\mathcal{R}$)'s solver will answer either `satisfiable` or `unknown` for this same derivation — and so it will say `yes` or `maybe`.

(c) Suppose both Standard CLP($\mathcal{R}$) and Complete CLP($\mathcal{R}$) return an answer to a given goal (i.e. the final constraint store simplified with respect to the variables in the initial goal). Are these answers always equivalent with respect to the variables in the initial goal? Or are there cases in which they are not equivalent but one implies the other? Are there cases in which neither implies the other? Give examples.

See p 58 of the text for a definition of "equivalent with respect to the variables in the initial goal".

The answers are always equivalent with respect to the variables in the initial goal — if they were not, one of the solvers would be unsound. For example, for the query

```
X*X=4.
```

Standard CLP($\mathcal{R}$) will respond

```
4 = X * X

*** Maybe
```

while Complete CLP($\mathcal{R}$) will respond

```
X = 2

*** Yes
```

These answers are equivalent with respect to `X`.

5. Consider the following CLP($\mathcal{R}$) rules.

```
smallprime(2).
smallprime(3).
smallprime(5).
```

Show the derivation tree for the following goal.

```
smallprime(N), N>2.
```

I don't want to try and convince latex to lay out a tree, so here's a description. The root of the tree is the goal

$$\langle \text{smallprime(N)}, N > 2 \mid \text{true} \rangle$$

From the root node there are three branches, R_1, R_2, and R_3, where R_1, R_2, and R_3 are the three "smallprime" rules.

Here is the R_1 branch:

$$\begin{array}{c} \langle N = 2, N > 2 \mid \text{true} \rangle \\ | \\ \langle N > 2 \mid N = 2 \rangle \\ | \\ \langle \blacksquare \mid N = 2, N > 2 \rangle \end{array}$$

Here is the R_2 branch:

$$\begin{array}{c} \langle N = 3, N > 2 \mid \text{true} \rangle \\ | \\ \langle N > 2 \mid N = 3 \rangle \\ | \\ \langle \square \mid N = 3, N > 2 \rangle \end{array}$$

Here is the R_3 branch:

$$\langle N = 5, N > 2 \mid \text{true} \rangle$$
$$\Big|$$
$$\langle N > 2 \mid N = 5 \rangle$$
$$\Big|$$
$$\langle \square \mid N = 5, N > 2 \rangle$$