Principles of Programming Languages (CS329) **Quiz 2**
Computer Science and Engineering **2004-10-26**
Indian Institute of Technology Bombay **2 hours 30 minutes**

NAME _____ ROLL _____

This quiz has 5 printed page/s. Write your answer clearly in the spaces provided and on any last blank page. Do not attach rough work. Use the marks alongside each question for time management. You can use the FWH book and your own class notes only. Provide 1–2 sentences of informal justification to qualify for partial credit in case your final answer is wrong. **NOTE:** Illogical or incoherent answers are worse than wrong answers or even *no* answer, and may fetch negative credit.

1. Recall the following Java program that we studied in class. (`AException` and `BException` are trivial extensions of `java.lang.Exception`.)

```
public class Finally {
  public static void f1() throws AException, BException {
    try {
      System.out.println("f1 try block begin");
      throw new AException("from f1 try block");
      System.out.println("f1 try block end");
    }
    finally {
      System.out.println("f1 finally block begin");
      throw new BException("from f1 finally block");
      System.out.println("f1 finally block end");
    }
  }
  public static void main(String[] args) {
    try { f1(); } catch ( Exception anyx ) {
      System.out.println("main catch " + anyx);
      // insert short statement here
    }
  }
}
```

The output from running the above program is

```
f1 try block
f1 finally block
main catch iitb.BException: from f1 finally block
```

As you can see, throwing a new exception from within the `finally` block makes the JVM "forget" that an `AException` was first thrown.

Games Jostling, a co-inventor of Java, does not approve of silently getting rid of the `AException` and proposes that any applicable handler (as in `main`) be fired one for *each* exception thrown in the dynamic scope of a `catch`.

(a) By inserting a very short (hint: less than 10 characters) statement at the position indicated, show what is ill-specified about Jostling's proposal. Give an explanation in at most two sentences.

	2

(b) Write down standard semantics ($\mathcal{E}$) with static and dynamic environments, continuation/s and a store to support Java's specification (later `throw` masks earlier one) of `throw`, `catch` and `finally`. For simplicity, assume that (as in FLaX) there is no class hierarchy over exceptions. (This means in the code above, `main` must catch both an `AException` and a `BException` separately.)

	6

(c) Make minimal changes to $\mathcal{E}$ to support the *opposite* specification: all `throw`s after the *first* one are lost. Note that this does not mean that `finally` blocks are either ignored or continued to completion.

	3

2. `reverse(X,Z)` is true iff list `Z` is the reverse of list `X`. We start with the rule:

```
reverse(X, Z) :- reverse(X, [], Z).
```

`reverse(X,Y,Z)` is true iff list `Z` is the reverse of `X` appended to `Y`. The second argument acts as an accumulator. We add the rule

```
reverse([], Z, Z).
```

Now complete the last rule:

```
reverse([A|X], Y, Z) :- __________
```

with a brief justification.

	2

3. Here is a fragment of (incorrect) Prolog code to find a path (not necessarily the shortest one) from one given node to another given node. (`\=` means "not equal to".)

```
path(Graph,Start,Stop,Path):- path1(Graph,Start,Stop,[Start],Path).
path1(Graph,Stop,Stop,Path,Path).
path1(Graph,Start,Stop,CurrPath,Path):-
 Start\=Stop,
 edge(Graph,Start,Next),     (*)
 path1(Graph,Next,Stop,[Next|CurrPath],Path).
```

Here `edge` is the external relation that is populated from the given graph's node adjacency matrix.

(a) What is incorrect in the definition of `path1` above?

	1

(b) Fix the problem by adding a simple clause at the position marked `(*)` and define this clause in detail. | | 2 |

4. The relation `permute(X,Y)` is satisfied when `Y` is a permutation of `X` (or vice versa). The relation `var(X)` is a Prolog builtin which is satisfied when `X` is an unbound variable. Essentially, we want to evaluate `permute` when its first argument is known, and the second is the result. The second rule simply flips then two when this is not the case.

```
permute([],[]).
permute(X,Y) :- var(X), permute(Y,X).
permute(L1, [L2H | L2T]) :-
    not(var(L1)),
    append(L1F1, [L2H | L1F2], L1),
    append(L1F1, L1F2, L1S),
    permute(L1S, L2T).
```

Using `permute` we can write a very naive sorting routine:

```
sorted(X,Y) :- permute(X,Y), unknown(Y).
unknown([]).
unknown([_]).
unknown([A,B|C]) :- ________, unknown(______).
```

(a) Give the simplest possible completion of the code for `unknown`. Each blank is a single, simple expression. Explain why your solution works briefly. | | 2 |

(b) Given in input list with n elements, on an average, how long will this sorting routine take as an asymptotic function of n?

	2

Total: 20