CS164: Final Exam

Fall 2003

- Please read all instructions (including these) carefully.
- **Write your name, login, and circle the time of your section.**
- Read each question carefully and think about what's being asked. Ask the proctor if you don't understand anything about any of the questions. **If you make any non-trivial assumptions, state them clearly.**
- You will have 2 hours and 50 minutes to work on the exam. The exam is closed book, but you may refer to your two pages of handwritten notes. You may also use your cheat sheets from the midterms.
- Solutions will be graded on correctness and **clarity**, so make sure your answers are neat and coherent. Remember that if we can't read it, it's wrong!
- Each question has a relatively simple and straightforward solution. **We will deduct points if your solution is far more complicated than necessary.** Partial answers will be graded for partial credit.
- Turn off your cell phone.
- Good luck!

NAME and LOGIN: _____

Your SSN or Student ID: _____

Circle the time of your section: 9:00 10:00 11:00 2:00 3:00

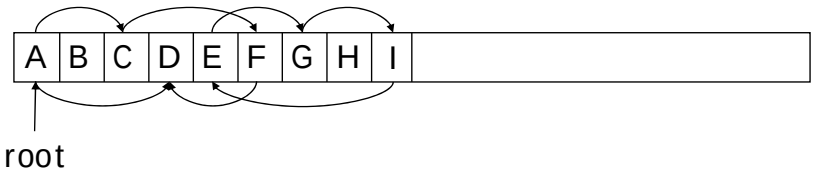
Q1 (30 points)	Q2 (45 points)	Q3 (25 points)	Q4 (50 points)	Total (**150** points)

1 Garbage Collection

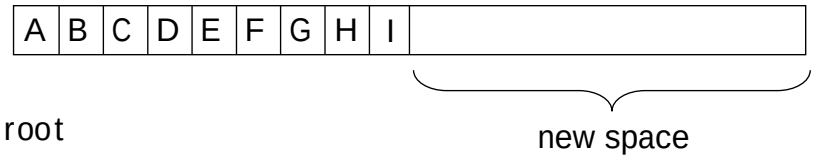
This question asks about the Stop&Copy garbage collector, and also about how to extend this collector with incremental collection.

The questions.

1. (8 points) The figure shows the state of the heap before garbage collection. Assume that the Stop&Copy collector copies objects in alphabetical order. Show the state of the heap immediately after the object *A* has been *both* copied and scanned.



Draw your answer here (don't draw pointer links that didn't change):



2. (8 points) Assume now that the Stop&Copy collector is *incremental*. That is, the main program can interleave its execution with the execution of the collector. In order for the incremental Stop&Copy collector to work correctly, the main program must be prevented from placing one of the following links. Which one?

   (a) from a copied object to a copied-and-scanned object, or

   (b) from an object in an old space to a copied-and-scanned object, or

   (c) from a copied-and-scanned object to an object in the old space, or

   (d) from a copied object to an object in the old space.

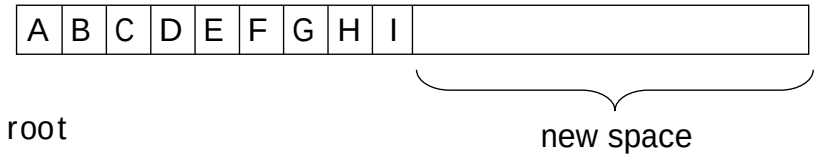
   Justify your answer by stating what would go wrong with the collection process if the main program indeed placed this link. Explain the problem with an **example**.

3. (10 points) To ensure that such a link is never placed by the main program, the various styles of incremental collectors use either read barriers or write barriers, or both. (Recall from the section that barriers are compiler-inserted statements performed before a read of (load) or a write to (a store) a pointer on the heap. Depending on the collector algorithm, the barrier's actions may copy an object and / or "fix" some pointer.)

   The question: Imagine that you want to build an incremental Stop&Copy collector that uses no write barriers, i.e., writes to pointers are not modified by the compiler at all. What checks and / or actions must the read barrier perform?

4. (4 points) Show the state of the heap after the main program reads a pointer to object F. Assume that before the read is performed, the heap looks as you showed in part 1 on the previous page.

   Draw your answer here:



2 Run-time Organization for Interfaces (a "project" question)

Imagine you want to add to Decaf the Java `interface` construct. To compile Java interfaces, you will have to design a different style of dynamic dispatch tables. This question asks you to design such a dispatch mechanism.

Background. If you never heard of Java interfaces, don't despair. We provide a sufficient background: Java interfaces provide a limited form of multiple inheritance. For the purpose of this question (we simplify Java unnoticeably), a Java interface is a type whose members are abstract methods. Except for the special class `Object`, each Java class `extends` exactly one class and `implements` zero or more interfaces. A class C is said to implement an interface I when C implements (that is, provides bodies of) all methods of I.

Example. The simple example below defines two interfaces (`IK` and `IM`) and three classes that implement (some) of the two interfaces.

```
public interface IK {
    abstract public void k();
}
public interface IM {
    abstract public void m();
}
public class A implements IK, IM {
    public void k() { System.out.println("A:k"); }
    public void m() { System.out.println("A:m"); }
}
public class B implements IM {
    public void k() { System.out.println("B:k"); }
    public void m() { System.out.println("B:m"); }
}
public class C extends B implements IK {
     public void k() { System.out.println("C:k"); }
}
```

Interface references. To make use of interfaces, Java introduces *interface references*, which are like object references, except that an interface reference of type I is allowed to point to objects of *any* type C such that the class C implements the interface I. For example, the interface reference `t` of type `IK` (introduced above) is allowed to point to an object of type `A` or `C` but not of type `B`, even though `B` happens to contain a method `k()`:

```
IK t;
t = new A();  // LEGAL
t = new B();  // ILLEGAL
t = new C();  // LEGAL

t.k();        // LEGAL (regardless of dynamic type of t)
t.m();        // ILLEGAL (regardless of dynamic type of t)
```

Compiling interfaces. Compiling Java interfaces boils down to supporting a new kind of method call, named *interface method call*. This call is like the (non-static) *virtual method call* that you implemented in PA5, except that the object on which you invoke a method comes from an interface reference, as opposed to an object reference.

Example (continued). The code below makes both kinds of calls.

```
B b; // b can point to any object of type B or subclass of B
IK ik; // ik can point to any object that implements interface IK

if (x==1) {
    b = new B();
    ik = new A();
} else {
    b = new C();
    ik = new C();
}
b.k();  // virtual call
ik.k(); // interface call
```

When `x==1`, the program outputs

```
B:k
A:k
```

Turn over.

The questions.

1. (2 point) Draw the class hierarchy for the example above. Don't include the Object class, but include the two interfaces and the `implements` edges.

2. (1 point) Indicate in the table below which interfaces are implemented by each class. Note that a class is considered to be implementing an interface if its superclass already implemented the interface.

3. (1 point) Indicate in the table below which methods can be invoked on an instance of each class. Use the notation `A:k` for the method `k` in the method `A`.

class	A	B	C
implemented interfaces			
invokable methods			

4. (2 point) What output does the example print when `x!=1`?

5. (2 point) Indicate the possible dynamic types of the object that can appear at each call statement.

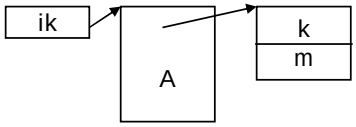
call statement	`b.k()` virtual call	`ik.k()` interface call
dynamic types of the object		

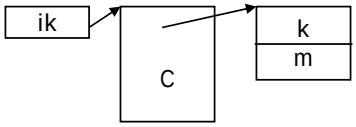
6. (3 point) Looking at the immediately preceding table, what is the key difference between the virtual-call dispatch tables of objects that may appear at the two kinds of calls?

7. (6 points) The figures below show the (traditional) dispatch tables for two executions of the example ($x = 1$ and $x \neq 1$). These dispatch tables are used in virtual calls, as in PA5. It seems that these tables could be used also in interface calls to decide which method to call. However, this impression is incorrect.

   Modify the example (on pages 4 and 5) so that the dispatch tables don't work in interface calls. Write your modification in the blank space below.

   To show why the tables don't work in general, modify also the dispatch tables shown below to reflect your modification to the example.





8. (4 points) Clearly explain the reason why interface method calls cannot rely on the traditional dispatch tables to decide which method to invoke.

Questions on this page will take you some time.

9. (14 points) Design run-time support for interface calls. Note that the new run-time support need not look like a dispatch table at all.

   What you need to draw: Using the running example above, show the data structures that your compiler would generate to perform the dispatch at an interface call.

   Note: To earn full credit, the space requirements of your run-time data structures must not be a function of the number of interfaces in the entire program, only a function of how many interfaces a class implements.

   Simplifications. (1) Assume interfaces cannot extend each other (as they can in Java). (2) Assume the program is completely available prior to the execution, as in Decaf (in contrast, in Java, classes are loaded and compiled during the execution).

10. (5 points) If your solution stores pointers (to methods or tables) in an array, give your rules for determining the index in the array where the pointer will be stored.

    (An example of a rule: in the case of "traditional" virtual dispatch tables, the rule is that the pointer to a method `f` is (a) placed at the same offset in the dispatch tables of all classes related by inheritance; and (b) the methods of a subclass are placed immediately below the offset of the superclass.)

11. (5 points) Show the code that your compiler would generate at an interface call. Use a pseudo-code, not an assembler. Your code should assume the run-time data structures you designed above.

3 Language Security

Imagine you want to extend your Decaf compiler with a security feature that thwarts the memory-bit-flip exploit discussed in the last lecture of the semester. Since this exploit relies on circumventing the Decaf/Java type checking, one prevention is to make the type system stronger. This question asks you develop such a prevention.

Background. While you don't need to remember (or even understand) all details of the exploit, you may find it helpful to look over the slides describing its steps. These slides are given on the last page of the exam.

How strong should your protection be? Before you start, we want to clarify that the solution to this question is very simple. **Warning:** since we are asking you to deal with an exploit that relies on memory errors, it may seem to you that your solution needs to either *correct* or at least *detect* all memory errors. In fact, it is expected that memory errors may crash the program even with your solution in place. Remember, a crash is better than when the attacker takes over your machine.

It is sufficient when your solution *prevents* the attack, by stopping the program. You don't even need to detect the attack (i.e., you don't need to flag with certainty that someone is attacking your machine).

As mentioned above, you are to prevent the exploit by strengthening the type system, that is, by performing some additional semantic checks. Such checks are of two kinds: *static* (compile-time) and *dynamic* (run-time).

Turn over.

The questions.

1. (7 points) Why don't Java type checks (static and dynamic) prevent the exploit?

2. (10 points) Design dynamic checks that prevent the exploit. Specifically, show what test you perform when an object field that stores a pointer is read (i.e., loaded from); and when it is written (i.e., stored to). If it helps, you can assume that local variables are always stored in registers, and that registers are not vulnerable to bit flips.

Reading from a reference variable p: (denote the value being read r)	
Writing to a reference variable p (denote the value being written r)	

3. (3 points) Circle the statement(s) in which dynamic checks will detect the security violation. (The program below refers to the example on slide 3 on the last page of the exam.)

```
B orig;
A tmp1 = orig.a1;
B bad = tmp1.b;
```

4. (5 points) Convince us that your dynamic checks are *sufficient* to prevent the attack.

4 A Language for Developing GUI Windows

This question comprehensively tests your knowledge from the entire semester. You will implement a small language for specifying GUI windows with dialog boxes and buttons. You can think of the language as a simple, convenient layer over a rather complicated GUI library.

Your test-taking strategy. Leave this problem until after you are done with the other questions. If you run out of time, answer as many parts as possible, while trying to convince us that you'd know how to finish each part.

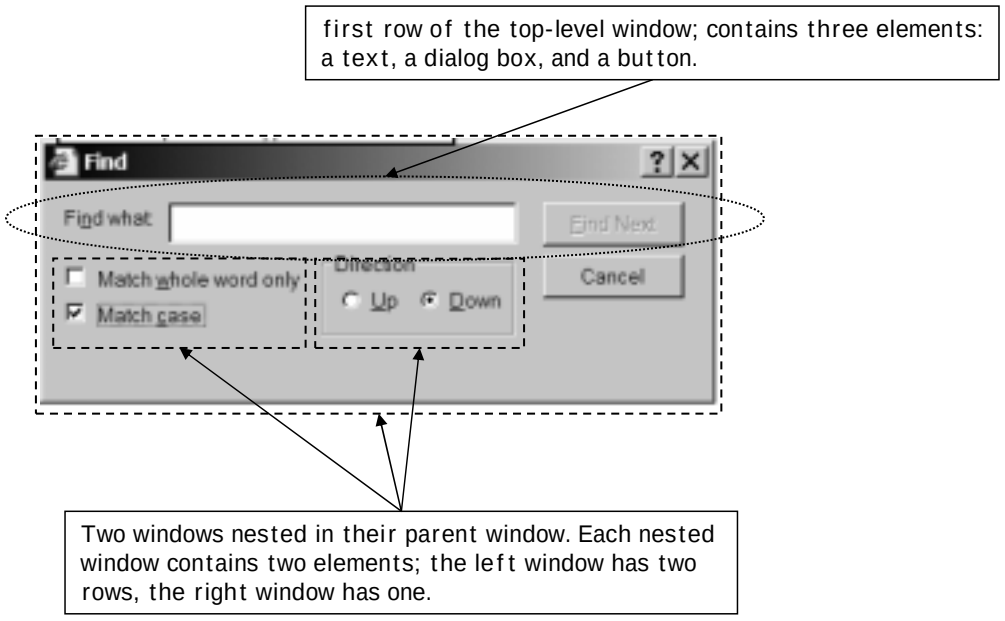
Assumptions. This question is intentionally specified in less detail than a typical exam question. Ask the proctors if you don't understand anything; however, if you merely need to make a design choice, then make the decision on your own, state your assumptions, and go on. We are prepared for the fact that there are multiple good solutions.

Now, the language you'll need to compile. The following program specifies a GUI window shown below it. (In other words, this is the window that will be drawn if you run the program.) The description of the language on the following page will help you understand the structure of the example program, which you should study very carefully before proceeding.

```
window "Find" {
    "Find what:"   <------------------>  ["Find next"],
    window "" {
         x "Match whole word only",
         X "Match case"
    }
    window framed "Direction" {
         o "Up"
         O "Down"
    }
    ["Cancel"]
}
```

An informal language specification. To understand the language you need to implement, it should be sufficient for you to study the example code on the previous page and read the bullets below. The annotated figure below may help, too.

- A window contains six kinds of elements:
  - text, denoted as string literals, i.e., *String*s, (in our program, there is only one text element, “Find what:”),
  - dialog boxes, denoted as <- - - -> where the length of the dashed part determines the size of the dialog box,
  - labeled buttons, denoted with [*String*],
  - labeled radio buttons, denoted as o *String*, (Capital O means that the button is initially selected),
  - labeled selections, denoted as x *String*, (capital X means that the button is initially selected),
  - nested windows, which can have recursively the same six kinds of elements.
- Windows have titles, given as *String*s, and can be optionally framed.
- Elements of a window are organized in rows. There can be zero or more rows in a window. Each row contains zero or more elements. Note that a comma token starts a new row of window elements.
- In the picture of the window, disregard the underlined letters and the fact that the “Find next” button is disabled. Your language doesn’t support these features.



The target language. You will compile the language into a sequence of calls to a hypothetical object-oriented GUI library. The running example could be compiled to the following code. Use these calls to implement your compiler.

```
// the constructor's argument is always the parent window;
Window top = new Window(null);    // top-level window is parentless
top.setTitle(''Find'');

// The first row of the top-level window

Text t = new Text(top);
t.setPosition(0,0);
// sets position within the parent window, given as x,y coord.
// position is relative to top left corner of parent window
// values are in percent of the parent size
t.setLabel("Find what:");

Dialog d = new Dialog(top);    d.setPosition(20,0);
d.setWidth(18*someConstant);  // there are 18 dashes in <--...-->

Button f = new Button(top);  f.setType(REGULAR_BUTTON);
f.setPosition(80,0);        f.setLabel("Find Next");

// Second row of the top level window
// Left nested window
Window w1 = new Window(top);
w1.setPosition(0,50);

Selection s1 = new Selection(w1);  s1.setPosition(0,0);
s1.setLabel("Match whole word only");

Selection s2 = new Selection(w1);  s2.setPosition(0,50);
s2.setLabel("Match case");
s2.setSelected(true);  // this selection is checked

// Right nested window
Window w2 = new Window(top); w2.setPosition(45,50);
w2.setTitle("Direction");    w2.setFramed(true);

Button r1 = new Button(w2);  r1.setType(RADIO);
r1.setPosition(0,0);         r1.setLabel("Up");

Button r2 = new Button(w2);  r2.setType(RADIO);
r2.setPosition(50,0);        r2.setLabel("Down");
r2.setSelected(true);  // this button is checked

// The very last element
Button c = new Button(top);  c.setType(REGULAR_BUTTON);
c.setPosition(80,50);        c.setLabel("Cancel");

// Finally, draw the entire window (it draws its subwindows, too, of course)
top.draw();
```

The questions

1. (5 points) Lexical Analysis. Identify the remaining lexical elements of the language.

lexeme (regular expression)	token (symbolic constant)	attribute, if any
“framed”	FRAMED	none (a keyword)

2. (12 points) Syntactic Analysis. Write a context-free grammar for the language.

3. (5 points) Abstract Syntax Tree. Design suitable AST nodes for the language.

node name	meaning of node	attributes	children nodes
Window	represents a window, which contains a sequence of rows	1) framed (boolean) 2) title (String)	a list of rows

4. (3 points) Draw the AST for the example program.

5. (5 points) Syntax Directed Translation. Write a syntax-directed translation for creating the AST. You can use a simple notation similar to the following:

$$E \rightarrow E + E \quad \text{if } E_2.\text{trans} == E_3.\text{trans then } E_1.\text{trans} := \textit{true}$$

.

6. (10 points) Write an interpreter that will traverse the AST and invokes GUI library calls to draw the window specified in the input program. You may want to traverse the AST in several passes, to first compute the sizes of windows and their position. You can assume that a suitable visitor support is available to you.

7. (10 points) Write a compiler that emits the code which, when executed, will draw the specified window. You may want to reuse some passes from your interpreter.