CS164: Midterm II

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.**
- Some questions span multiple pages. Be sure to answer every part of each question.
- You will have 1 hour and 20 minutes to work on the exam. The exam is closed book, but you may refer to your two pages of handwritten notes.
- 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.
- Write all of your answers in the space provided on the exam, and clearly mark your answers. Use the backs of the exam pages for scratch work. Do not use any extra scratch paper. Do not unstaple the exam.
- 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 (35 points)	Q3 (20 points)	Q4 (15 points)	Total (100 points)

1 Activation Records (30 points)

Consider the Java program on the following page:

- (3 points) What is the output of the program?

- (8 points) Draw the procedure-call stack at the point when the execution reaches the call to println (i.e., before the arguments to println are pushed on the stack). Draw on the following page.
  - Show, in detail, the content of each activation record (you don't need to show where the `args` argument of `main()` points to). Assume that arguments are pushed on the stack in reverse order (i.e., the first argument is pushed last).
  - Show the boundaries of each activation record.
  - For one activation record, show (i) which part of the activation record is created by the caller, (ii) which by the callee; and (iii) which could be created by either, depending on the calling convention.
  - Indicate where sp and fp registers point to when the execution stops at the call to println (assume that sp points to the top of the stack, as in x86).
  - Show where all control links point to (the control link is a pointer to the previous activation record).
  - Show where all return addresses point to (make them point to the Java source code).

- (5 points) The fp register has been introduced into processors to simplify code generation of accesses to the activation record.

  Question: In theory, the sp register could be used for this purpose. Why is it easier to use the fp register than the sp register? Answer this question by stating what information you would have to maintain during code generation to use sp for accessing the activation record.

```
class A {
   int data;
   A   a;
};
class Test {
   public static void g (A r, A s) {
      System.out.println(r.data + s.data);
   }
   public static void f (A x, A y) {
      A t = x.a;
      g(t, y);
   }
   public static void main (String[] args) {
      A p = new A();    // call this object P
      p.data = 1;
      A r = new A();    // call this object R
      r.data = 2;
      p.a = r;
      f(p, r);
   }
};
```

Optimizing the activation records. Assume that we restrict SkimDecaf and require that the programmer cannot write programs with recursion (i.e., procedures cannot call themselves, *directly* or *indirectly* through other procedures). (Recall also that SkimDecaf supports only static methods.) While some interesting programs (e.g., recursive graph traversals) cannot be written in such a restricted language, the language allows generating a more efficient code. Here's why:

- (6 points) One optimization is based on the observation that, without recursion, at any point of execution, each method can be active at most once. Hence, at any given moment, there is at most one active activation record for each procedure.

  This observation simplifies the implementation of activation records: because there is at most one copy of local variables and actual arguments for each procedure, they need not be stored on the stack; instead, they can be stored in static (global) memory at a fixed location. The benefit of this optimization is that there is less pushing and popping from the stack.

  Question: Which of the following are not needed when each procedure uses only one activation record that is located in a fixed memory location? Why are they still/no longer needed?

  - control link (i.e., the saved frame pointer)
  - return address.

- (8 points) In order to perform the above optimization, the compiler must verify that the programmer indeed did not use recursion (direct or indirect). Give a (simple) algorithm that your SkimDecaf compiler could use to detect the presence of recursion in the program. Your approach should perform a simple traversal of the AST representation of the program. Feel free to extend the AST as necessary; if you do, state clearly what links and/or nodes you added to the AST.

2 Global Dataflow Analysis. (35 points)

Uninitialized Variables. A programming language may want to guarantee that each local variable is initialized (typically to zero or null) by the compiler when it is declared. To provide this guarantee, a compiler for such a language must generate statements that initialize the variables.

Because initialization statements are costly, some languages (e.g., C) do not initialize variables. Instead, they assume that assigning a value to a variable before its first use is the responsibility of the programmer. To help the programmer find missing initializations, compilers for such languages analyze the program and tell the programmer when an expression uses a *potentially uninitialized variable.*

How do compilers determine if a statement, say $z := x + y$, uses an uninitialized variable? They use the following rule: x is potentially uninitialized in that statement if there is a path, in the CFG, *from* the beginning of the method *to* the statement such that the path does not contain an assignment into x. Same rule applies to y.

For example, in the statement $z = x + y$ below, x is potentially uninitialized but y is initialized.

```
static void f (int a) {
   int x, y, z;
   y = 1;
   if (a > 0) { x = 1; }
   z = x + y;
}
```

Why is global dataflow analysis imprecise? Because dataflow analysis does not analyze all aspects of the program, it only *approximates* what the program will do when it is executed. As a result, the analysis may think that a variable is uninitialized when, in fact, it will always be initialized.

- (4 points) Modify the program below in such a way that the following three conditions hold. The first condition refers to the analysis; the latter two refer to how the program behaves when executed:

  1. dataflow analysis will find x to be uninitialized at the statement S: $z = x + y$;
  2. S is not dead, i.e., there is a value of a for which the statement S will execute when $f(a)$ is invoked; and
  3. x is initialized whenever the execution reaches S, for each possible execution of the method f.

  That is, your task is to insert statement(s) and / or expression(s) to the program so that the analysis finds a mere approximation of the program's run-time behavior. (Note that *not all* empty lines and spaces need to be filled in.)

```
static void f (int a) {
   int x, y, z;
   y = 1;

   if (    ) {

         x = 1;
   }
   if (    ) {

         z = x + y;
   }
}
```

- (3 points) Why is dataflow analysis of the above program (after your edits) imprecise (i.e., the analysis says a variable may be uninitialized when it in fact will be initialized in all possible executions of the program)?

Computing the solution of the dataflow analysis. Many algorithms for performing dataflow analysis exist. The simple algorithm that we covered in class is *iterative*: It first initializes the solution of the analysis and then iteratively applies rules until the solution does not change.

Here are iteration rules for computing whether a variable x is uninitialized. As in the lecture, the solution is denoted as $I(p, x, in/out)$. The solution $I(...)$ is **true** if x is always initialized when the program reaches entry / exit of statement p. The solution is **false** if x may be *un*initialized when the program reaches entry / exit of statement p.

So, again: **true** means always initialized and **false** means maybe *un*initialized.

The following rules roughly correspond to the Rules 1–4 of liveness analysis (there are important differences, though).

1. $I(p, x, in) := \wedge \{I(s, x, out) \mid s$ is a predecessor of p in the CFG$\}$

   This rules describes how information is propagated across CFG *edges* (and hence also how it is merged when CFG edges meet). The following rules describe how CFG *nodes* are handled.

2. $I(start, x, in) :=$ **false**, where $start$ is the first statement of the method.

   This rule says that all variables are assumed to be uninitialized on the entry point of the method.

3. $I(p, x, out) :=$ **true** if x is assigned in node p.

   This rule says that x is initialized on the exit of its assignment.

4. $I(p, x, out) := I(p, x, in)$ if p does not assign x.

   This rule says that statements that do not assign x do not affect the solution; they merely propagate the information on whether x is initialized.

- (3 points) Do these rules describe a *forward* or a *backward* analysis? Explain why.

- (4 points) Explain why Rule 1 uses the logical *and* rather than the logical *or*).

- (6 points) The rules above describe the entire algorithm; the only missing part is how the $I(...)$ values should be initialized before the rules are applied. (The initial solution can be set either to **true** or to **false**.)

  In order to determine which of the two is the correct initialization, analyze variable x in the program below twice, once initializing $I(...)$ to **true**, once initializing to **false**.

  It is sufficient if you show the final solutions for x. You do not need to (but may) show the results of individual rule applications.

```
x = 0;
y = 1;
while ( f(x) ) {
      y = y + 1;
}
z = x + y;
```

  Draw two separate CFGs for the program, one for each analysis.

- (3 points) What is the correct way to initialize the solutions?

- (4 points) (a) Explain why incorrect initialization produced an imprecise result. (b) Is this imprecise result of the kind you'd prefer (see the question at the bottom of page 6).

Using the dataflow analysis for optimization. So far, we assumed that our *initialization analysis* is to be used for issuing warnings to the programmer in languages that *do not* perform initialization (like C / C++). The result of the analysis can, however, be also used for optimization of languages that *do* initialize all variables (like Java).

The optimization is simple. The informal optimization rule is: the compiler need not initialize a variable x if x is initialized by the programmer before its first use.

- (5 points) Which of the four rules precisely describes when the compiler-inserted initialization can be safely omitted?

  1. The initialization statement $x := 0$ need not be inserted into the beginning of the method if $I(p, x, in) =$ **true** at *every* node p that contains a use of x (e.g., $z := x + y$).
  2. The initialization statement $x := 0$ need not be inserted into the beginning of the method if $I(p, x, in) =$ **true** at at least *some* node p that contains a use of x (e.g., $z := x + y$).
  3. The initialization statement $x := 0$ need not be inserted into the beginning of the method if $I(p, x, in) =$ **false** at *every* node p that contains a use of x (e.g., $z := x + y$).
  4. The initialization statement $x := 0$ need not be inserted into the beginning of the method if $I(p, x, in) =$ **false** at at least *some* node p that contains a use of x (e.g., $z := x + y$).

  Assume, of course, that the analysis was computed before the compiler-inserted initialization statements were inserted.

- (3 points) In order for this optimization to be safe (which means that the compiler should never omit an initialization that may be necessary), the analysis must give conservative answers. What should the analysis answer (true or false) when it is not sure whether a variable x is initialized? Why?

3 Typechecking. (20 points)

You are given three pairs of similar typing rules (two pairs appear on this page, and one pair appears on the following page), and in each pair one rule is correct and the other incorrect. For each pair:

- (6 points) Identify which rule is correct and which is incorrect.
- (6 points) Say whether the incorrect rule affects *soundness* (allows unsafe programs to pass) or *completeness* (does not type check programs that are clearly type safe; that is, restricts the programmer), or both.
- (8 points) Give an example program that exposes the problem with the incorrect rule.

Assume that there is subtyping for the final pair of rules, and that there is no subtyping for the first two pairs.

$$\frac{O \vdash e_1 : int \qquad O \vdash e_2 : int}{O \vdash e_1 + e_2 : int}$$

$$\frac{O \vdash e_1 : int}{O \vdash e_1 + e_2 : int}$$

$$\frac{O \vdash e_0 : int \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash T_0\ x = e_0\ ;\ e_1 : T_1}$$

$$\frac{O \vdash e_0 : T_0 \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash T_0\ x = e_0\ ;\ e_1 : T_1}$$

$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \dots \\ O, M \vdash e_n : T_n \\ M(T_0, f) = (T'_1, T'_2, \dots, T'_n, T_r) \\ T_i = T'_i \text{ (for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0.f(e_1, \dots, e_n) : T_r}$$

$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \dots \\ O, M \vdash e_n : T_n \\ M(T_0, f) = (T'_1, T'_2, \dots, T'_n, T_r) \\ T_i \leq T'_i \text{ (for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0.f(e_1, \dots, e_n) : T_r}$$

4 DPAR parser generator. (15 points)

The following is a fragment of the starter kit for PA3. The grammar and actions below handle declarations of classes that contain no methods. Your task is to fill in the missing code. There are **six (6)** instances of missing code, each denoted with ".....".

```
PROLOGUE: [|
    private AST ast = new AST(new HashMap());
    private SimpleName makeSimpleName(int offset) {
         return ast.newSimpleName(((Token)peek(offset)).getLexeme()); }
|]

PROGRAM -> _ [| push(new ArrayList()); |] CLASSDECLLIST <EOF>;

CLASSDECLLIST   -> CLASSDECL
    [|
        ((List)peek(-1)).add(peek(0));
        .....;
    |]
  CDTAIL ;

CDTAIL -> CLASSDECLLIST | _ ;

CLASSDECL -> <CLASS> <IDENTIFIER>
    [|
        TypeDeclaration td = ast.newTypeDeclaration();
        td.setName(makeSimpleName(0));
        push(td);
    |]
  <LBRACE>
    [|
        push(((.....)peek(-1)).bodyDeclarations());
    |]
  FIELDDECLLIST <RBRACE>
    [|
        .....;
    |] ;

FIELDDECLLIST -> FIELDDECL
    [|
       ((List)peek(-1)).add((FieldDeclaration)peek(0));
       push(peek(-1));
    |]
  FIELDDECLLIST | _ ;

FIELDDECL ->  <INT> FIELDID <SEMICOLON>
    [|
        ((FieldDeclaration)peek(.....)).setType(ast.newPrimitiveType(PrimitiveType.INT));
        push(.....);
    |] ;

FIELDID ->  <IDENTIFIER>
    [|
        VariableDeclarationFragment vdf = ast.newVariableDeclarationFragment();
        vdf.setName((SimpleName)ast.newSimpleName(((Token)peek(.....)).getLexeme()));
        push(ast.newFieldDeclaration(vdf));
    |] ;
```