UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 2003

BEng Honours Degree in Computing Part II
MEng Honours Degrees in Computing Part II
BSc Honours Degree in Mathematics and Computer Science Part II
MSci Honours Degree in Mathematics and Computer Science Part II
BSc Honours Degree in Mathematics and Computer Science Part III
MSci Honours Degree in Mathematics and Computer Science Part III
PhD
for Internal Students of the Imperial College of Science, Technology and Medicine

This paper is also taken for the relevant examinations for the Associateship of the City and Guilds of London Institute
This paper is also taken for the relevant examinations for the Associateship of the Royal College of Science

PAPER C221=MC221

COMPILERS

Tuesday 13 May 2003, 10:00
Duration: 120 minutes

Answer THREE questions

Paper contains 4 questions
Calculators not required

Section A *(Use a separate answer book for this Section)*

1 Here are the data-flow equations for live variable analysis:

$$\begin{aligned} \text{LiveOut}(n) &= \cup_{s \in succ(n)} \text{LiveIn}(s) \\ \text{LiveIn}(n) &= uses(n) \cup (\text{LiveOut}(n) - defs(n)) \end{aligned}$$

a What do the elements of the LiveIn and LiveOut sets represent?

b When solving these equations iteratively, what should the initial value be for each LiveIn(n) and LiveOut(n) set?

c How big could the LiveIn and LiveOut sets become?

d What information is needed for each node in the control flow graph?

e Is this a forward or a backward data-flow problem?

f Given the solution to the equations (i.e. LiveOut(n) and LiveIn(n) for each node), give an algorithm for building the interference graph required for graph-colouring register allocation.

g In a Java just-in-time compiler (JIT), it is essential to minimise the time spent on code generation. Is graph colouring a good approach to register allocation in a JIT? Justify your answer carefully.

(The seven parts carry, respectively, 15%, 10%, 10%, 10%, 10%, 25% and 20% of the marks).

2a Write down a *simple* fragment of C or Java whose control flow graph includes a node which does not dominate its successor. Illustrate your answer by drawing the control flow graph.

b Write down a *simple* fragment of C or Java whose control flow graph includes two back-edges which both point to the same node. Illustrate your answer by drawing the control flow graph.

c A recently proposed programming language (called Alef) has an unusual construct, intended for handling errors. Consider the following example:

```
PROCEDURE p();
BEGIN
  Some user code to perform action 1;
  RESCUE
    Some user code to undo action 1;    (if can't continue later)
    RETURN;
  END RESCUE;
  User code to perform action 2;
  RESCUE
    User code to undo action 2;         (if can't continue later)
  END RESCUE
  IF error THEN RAISE ENDIF             (discover we can't continue)
  ...
END p;
```

The idea is that where code for some action appears, the **RESCUE** clause provides a way of undoing it should an error occur later. The **RAISE** statement causes each of the **RESCUE** clauses to be executed in turn, back to the beginning of the procedure.

Thus the example above is equivalent to the following:

```
PROCEDURE p();
BEGIN
  User code to perform action 1;
  User code to perform action 2;
  IF error THEN
    User code to undo action 2;
    User code to undo action 1;
    RETURN;
  ENDIF
  ...
END p;
```

Use Haskell (or some convenient and clear pseudocode) to write a detailed design for a *simple* code generator for statement lists in this language. The target machine is a 68000, although you will need only labels and jump instructions. Use the following abstract syntax tree:

```
type statlist == [stat]
data stat = Rescue Statlist | Raise | Other Normalstat
```

You may assume the existence of a function **transOtherStat** which generates code for ordinary statements.

(The three parts carry, respectively, 20%, 20% and 60% of the marks).

Section B *(Use a separate answer book for this Section)*

3a Give a DFA for strings of letters that contain the substring `'cat'`. Your DFA should accept words like `abdicate` and `desiccate`, but fail on `boccaccio`. Use the label **not a** for any letter except **a**; the label **not c** for any letter except **c**, etc.

b For this question your objective is to show that the following grammar is LR(1) but not LALR(1).

```
S'  →  S
S   →  Xa | bXc | Yc | bYa
X   →  d
Y   →  d
```

i) Give the DFA of LR(1) items for the grammar.

ii) Construct the parse table from the DFA of LR(1) items. Explain why the grammar is LR(1).

iii) Indicate how the LALR(1) DFA and parse table for this grammar differs from the LR(1) DFA and parse table, and explain why the grammar is not LALR(1).

The four parts a, b(i), b(ii), b(iii) carry, respectively, 20%, 30%, 30%, 20% of the marks.

4a Draw a fully labelled diagram showing clearly the memory layout after execution of the following:

```
interface X { method p, method q }
interface Y { method r }
interface Z { method s }

class A {
    A g
    method m ( ) { ... }
    method n ( ) { ... }
}

class B extends A implements X {
    method m ( ) { ... }
    method p ( ) { ... }
    method q ( ) { ... }
}

class C extends B implements Y, Z {
    int h
    method p ( ) { ... }
    method r ( ) { ... }
    method s ( ) { ... }
}

B b = new B ( )
C c = new C ( )
b.g = c
c.g = b
c.h = 2
X x = c
```

b Given the declarations above, show how the following statements could be implemented in assembly language (e.g. Pentium or 68000). Assume that *c* and *x* are assigned as above and mapped to registers. You can describe your implementation in English if you prefer but be as precise as assembly language.

```
x.q ( )
c = (B) x
```

c Outline how reference-counting garbage collection works. Include an example when explaining the case of cyclic data structures. Outline how object reference assignment could be implemented in reference-counting garbage collection, e.g.

```
ObjRefP = ObjRefQ
```

d Compare and contrast the performance of mark-sweep garbage collection with reference-counting garbage collection.

The four parts carry, respectively, 30%, 20%, 30%, 20% of the marks.