UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 2002

BEng Honours Degree in Computing Part II
MEng Honours Degrees in Computing Part II
BEng Honours Degree in Information Systems Engineering Part II
MEng Honours Degree in Information Systems Engineering 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
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=I2.8

COMPILERS

Thursday 2 May 2002, 14:00
Duration: 90 minutes
(Reading time 5 minutes)

Answer THREE questions

Paper contains 4 questions
Calculators not required

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

1a Give a regular expression for each of the following:

i) All strings of decimal digits that represent even numbers.

ii) All strings of uppercase letters that begin and end in the letter `A`, including a single letter `A`.

iii) All strings of uppercase letters that either begin or end in the letter `A`. Exclude strings that begin and end in the letter `A`.

iv) All strings of uppercase letters that contain an even number of `A`'s.

b Give a formal definition for an LL(1) grammar. You need not give definitions for FIRST set and FOLLOW set.

c Consider the following grammar for an expression:

```
Expr      → Operand | List
List      → '[' Seq ']'
Seq       → Expr ',' Seq | Expr
Operand   → num | id
```

Assume that *num* and *id* are tokens (terminals) provided by the lexical analyser.

i) Transform the grammar to LL(1). Use BNF not EBNF for the transformed grammar.

ii) Derive the FIRST and FOLLOW sets for the non-terminals of your transformed grammar. Show your working.

iii) Using your definition in part b, show that your transformed grammar is LL(1).

iv) Construct the LL(1) parsing table for your transformed grammar.

The six parts a, b, c(i), c(ii), c(iii), c(iv) carry, respectively, 20%, 20%, 10%, 20%, 10%, 20% of the marks.

2a Give an AST class for Java's **new** expression:

```
NewExpr → new classidentifier '(' expressions ')'
```

Your class should include syntactic and semantic attributes as well as a check method that reports as many semantic errors as possible. Assume that classes have at most one declared constructor i.e. constructor overloading is not supported. State any additional assumptions that you make.

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

```
interface I {
    method p ( )
    method q ( )
}
class D implements I {
    I x
    method m ( ) { ... }
    method p ( ) { ... }
    method q ( ) { ... }
}
class E extends D {
    D y
    method p ( ) { ... }
    method r ( ) { ... }
}
D d = new D ( )
E e = new E ( )
d.x = e
e.y = d
```

c Design a runtime implementation for Java's **instanceof** operator which for

```
object instanceof class
```

returns true if *object* is an instance of class *class*. Give both the memory organisation and the code needed for implementing the operator.

The three parts carry, respectively, 50%, 25%, 25% of the marks.

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

1 Consider a simple programming language with assignment statements, arithmetic expressions, an if-then-else construct, and Boolean expressions. The abstract syntax for the language is represented by the following Haskell data type:

```
data AExp = Const Int | Var Name | Plus AExp AExp
data BExp = LessThan AExp AExp
data Stat = Assign Name AExp |
            If BExp Stat Stat  -- (then-branch, else-branch)
```

Consider code generation for a simplified 68000 instruction set, represented in Haskell as follows:

```
data Instruction =   Define [Char]          -- "label:"
                   | Bra [Char]             -- "bra label" (unconditional)
                   | Blt [Char]             -- "blt label" (conditional, <)
                   | Bge [Char]             -- "bge label" (conditional, >=)
                   | Mov operand operand    -- "mov.l xxx yyy" (yyy:=xxx)
                   | Add operand operand    -- "add.l xxx yyy" (yyy:=yyy+xxx)
                   | Cmp operand operand    -- "cmp.l xxx yyy"

data Operand =   Reg Register     -- addressing mode for access to a register
               | Abs [Char]       -- for referring to a variable at a named location
               | ImmNum Int       -- "#n"

data Register = D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 | A7
```

Assume that you have been given a code generation function "**transAExp**", which generates code for an arithmetic expression. Its type specification is as follows:

```
transAExp :: [Register] -> AExp -> [Instruction]
```

This function returns a list of instructions which evaluate the expression, using the registers given; the result is delivered in the first register in the list.

a Using Haskell (or some appropriate pseudo-code of your choice), write down a code generation function for "**Stat**" above. Note that you must handle the two cases (assignment, and if-then-else). Assume you have a function "newlabel" which returns a fresh label not used elsewhere.

b For the rest of this question, consider the same language as above, but with "**BExp**" extended as follows:

```
data BExp = LessThan AExp AExp | And BExp BExp | Not BExp
```

To handle this, define a function "**transBExp**". Your function should have the following type:

```
transBExp :: BExp -> Label -> Label -> [Instruction]
```

Your function should generate code which branches to the first label if the result of the expression is false, and to the second label if the result is true. Note that you must handle the three cases.

(The two parts carry, respectively, 50% and 50% of the marks).

2 Consider the following code fragment of C code:

```
int i;
int A[100][100];
for (i=0; i<100; i++) {
  A[i][i] = 0;
}
```

Suppose an optimising compiler represents this internally using the following intermediate code:

```
1:      mov #0,P0        ; pseudo-register P0 holds variable i
2:      bra L1
3:  L2:
4:      mul P1,P0,#400   ; P1 := P0*400: P1 is offset of A[i]
5:      mul P2,P0,#4
6:      add P3,P1,P2     ; P3 is offset of A[i][i]
7:      add P4,P3,#A
8:      mov #0,(P4)      ; A[i][i] = 0
9:      add P0,P0,#1     ; P0 := P0+1
10: L1:
11:     cmp P0,#100
12:     blt L2
```

a Recall the data-flow equations for live-range analysis:

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

Assume that LiveOut(9) = $\{P0\}$. Write down the sets

(i) LiveIn(9)
(ii) LiveOut(8)
(iii) LiveIn(8)
(iv) LiveOut(7)

b Draw a diagram showing the register interference graph for the example program.

c Use graph colouring to find an assignment of pseudo-registers (P0..P4) to a minimum number of physical registers (R0..R15).

d Recall that an induction variable is a variable which increases by a loop-invariant amount at each loop iteration. Write down all the induction variables in the example code above, together, in each case, with the loop-invariant increment.

e Using the information from part(d), or otherwise, write down an optimised version of the code above (you should find the code can be simplified substantially).

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

End of Paper