UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 1998

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
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 Royal College of Science
Associateship of the City and Guilds of London Institute

PAPER 2.3 / MC2.3

COMPILERS

Thursday, May 14th 1998, 2.00 - 3.30

Answer THREE questions

For admin. only: paper contains 4 questions

1 Consider a simple programming language with record types. Here is an example program fragment:

```
type Occasion : record
                  date : int
                  duration : int
                end record

procedure MakeCleanUp(party, cleanup : pointer to Occasion)
  cleanup->time := party->date+1
  cleanup->duration := party->duration*2
end MakeCleanUp
```

a Explain what information will have to be maintained in the symbol table so that the compiler can generate correct code and check for errors.

b Consider the following addition to the code above:

```
type Event : record
               extends Occasion
               cause : array [1..10] of pointer to Event
               NoOfCauses : int
             end record

procedure Connect(e1, e2 : Event)
  e2->NoOfCauses := e2->NoOfCauses+1
  e2->cause[NoOfCauses] := e1
end Connect

var ball, washingup : pointer to Event
ball := new Event
washingup := new Event
ball->date := 2000
ball->duration := 365
MakeCleanUp(ball, washingup)
Connect(ball, washingup)
```

The `extends` keyword indicates that the `Event` record type includes all the fields of the `Occasion` type.

Explain why the line

```
  MakeCleanUp(ball, washingup)
```

is valid. How would a compiler check it?

c Recommend a garbage collection scheme for `Events`. Explain your choice.

(The three parts carry, respectively, 35%, 35% and 30% of the marks).

2 Consider an extremely simple programming language with "for" loops and assignment statements. An example program might be:

```
for i := 1 to 10 do
  for j := 100 to 110 do
    a := a - (i - j);
```

In Miranda, a suitable abstract syntax tree data type is as follows:

```
name == [char]
stat ::= For name num num stat | Assign name exp
exp ::= Const num | Var name | Minus exp exp
```

In Turing this AST could be represented using the following record types:

```
type stat :
    union kind : For..Assign of
        label For :
            ControlVariable : name
            StartValue, EndValue : int
            Body : pointer to stat
        label Assign :
            RHS : pointer to exp
    end union
type exp :
    union kind : Var..Minus of
        label Var :
            Name : name
        label Minus :
            Exp1, Exp2 : pointer to exp
    end union
```

a Using a Turing–like notation, sketch an interpreter for this language. You should assume that you have been given a procedure `SetVar(name,value)` and a function `GetVar(name)`, for maintaining a table of variables and their values.

b Suppose you have written a simple compiler for this language, which generates code for a conventional machine with plenty of registers (e.g. a 68000 or MIPS). Write down the assembler code which your compiler would produce given the example program shown at the start of the question (keep your answer straightforward; you can mention optimisations in the next part of the question).

c Identify *three* specific reasons why the compilation approach (i.e. generating machine code then executing it) should be more efficient than interpreting a program (as in part (a)).

(The four parts carry, respectively, 35%, 35%, and 30% of the marks).

Turn over ...

3a Consider the following procedure:

```
procedure SwapSections(m, n, p : int)
  var LeftSize, RightSize : int
  LeftSize := n-m
  RightSize := p-n
  if LeftSize > RightSize then
    SwapEquals(m, n, RightSize)
    SwapSections(m+RightSize, n, p)
  else if LeftSize < RightSize then
    SwapEquals(m, m+RightSize, LeftSize)
    SwapSections(m, n, m+RightSize)
  else
    SwapEquals(m, n, RightSize)
  end if
end SwapSections
```

Consider the execution of the statement "`SwapSections(0,6,9);`". Draw a sequence of diagrams showing the stack as it appears each time `SwapSections` is entered. Show all variables, parameters, return addresses and links, but do not attempt to show any details of the execution of `SwapEquals`.

b Assuming two-byte integers and four-byte pointers, how much memory space is needed by this program? Do not count any space needed by `SwapEquals`. How might this space requirement be reduced?

c The procedure `SwapEquals` is given below:

```
procedure SwapEquals(n, m, size : int)
  tmp : int
  for i = 0 .. size - 1
    tmp := A(n+i)
    A(n+i) := A(m+i)
    A(m+i) := tmp
  end for
end SwapEquals
```

`A` is a global array of bytes. The procedure swaps two equal-sized regions of `A`, starting at `n` and `m` respectively.

Sketch the 68000 code you would expect a good optimising compiler to produce for the three statements that make up the body of this loop. Explain each optimisation.

(The three parts carry, respectively, 45%, 20% and 35% of the marks).

4a Consider the following conditional statement:

```
if a and not (b or c) then e := 1 end if
```

Give *two* different reasons why short-circuiting of Boolean expressions might be *inefficient*?

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

```
procedure p()
  User code to perform action 1
  rescue
    User code to undo action 1       (if can't continue later)
  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 end if         (discover we can't continue)
end p
```

The `rescue` keyword introduces a block of statements, called a `rescue` clause, terminated by `end rescue`. 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
  end if
end p
```

Use Miranda (or some other convenient and clear notation) 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 (at most) only labels and jump instructions. Use the following abstract syntax tree:

```
statlist == [stat]
normalstatlist == [normalstat]
stat ::= Rescue normalstatlist | Raise | Other normalstat
```

You may assume the existence of functions `transOtherStat` and `transNormalStatlist` which generate code for ordinary statements.

(The three parts carry, respectively, 30% and 70% of the marks).

End of Paper