UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 1996

BSc Honours Degree in Mathematics and Computer Science Part III
MSc Degree in Computing Science
for Internal Students of the Imperial College of Science, Technology and Medicine

This paper is also taken for the relevant examinations for the
Diploma of Membership of Imperial College
Associateship of the Royal College of Science

PAPER M338

LANGUAGE PROCESSORS
Wednesday, May 8th 1996, 2.00 - 4.00

Answer THREE questions

For` admin. only: paper contains
5 questions
4 pages (excluding cover page)

1a Draw a diagram showing the structure of a typical compiler. Describe the function of each component of the diagram.

b For the following program, show (as Miranda expressions) the different program representations produced during translation from source to the assembly language of an idealised stack machine having an operation to print the top of stack.

```
var alpha : int := 20
var beta : int := 30
alpha := alpha + 2 * beta
put alpha , beta
```

State any assumptions you make, including any Miranda data definitions needed to clarify the expressions.

2 A language provides expressions which consist of integer and Boolean *constants*, a dyadic *addition* operation to add two integer values, a dyadic *comparison* operation to compare two operands of the same type for equality (yielding a Boolean result), and a triadic *conditional* operation to evaluate one of two expressions of the same type according to the value of a Boolean expression.

a Suggest an appropriate Miranda data type for representing the abstract syntax of expressions written in the language.

b Write an *interpreter* which will evaluate semantically correct expressions (*i.e.* having operands of the correct types) written in the language.

c Suggest an appropriate Miranda data type for representing the *type* of value represented by an expression, including the possibility that it may be *semantically incorrect.*

d Write a *type-checker* which will determine the type of value represented by an expression, or report that it is semantically incorrect (but *do not report the reason*).

e It is proposed to introduce *variables* into the language. State *briefly* what additional data types and data structures are required in your abstract syntax, interpreter and type-checker, but *do not write any code.*

The five parts carry, respectively, 15%, 35%, 10%, 25%, 15% of the marks.

3 The syntax of Boolean expressions in a programming language is defined by the following BNF productions:

```
<exp> ::= <exp> or <ter> | <ter>
<ter> ::= <ter> and <pri> | <pri>
<pri> ::= true | false | not <pri> |
          <var> | ( <exp> )
<var> ::= a sequence of letters
```

a Suggest a suitable *abstract syntax* for representing Boolean expressions, and write down the abstract syntax tree for the expression:

```
not ( a or b and c or d and not e and f )
```

b Write an *interpreter* which will evaluate Boolean expressions represented as abstract syntax trees, defining any additional data structures required for the purpose.

c Consider the evaluation of the above expression in the cases when the variable a has the value **true** and when the variable d has the value **false**; hence show how your interpreter can be optimised to avoid unnecessary computation in such cases.

d Describe the optimisation of *constant-folding* and write an optimisation function which will remove redundant operations from your abstract syntax trees.

e Show how the interpreter optimisation introduced in part *c* can be combined with constant-folding to allow your optimisation function to remove further redundant operations from your abstract syntax trees.

The five parts carry, respectively, 15%, 25%, 15%, 25%, 20% of the marks.

Turn over ...

4 Consider the following grammar for simple English sentences capable of describing the results of competitions:

```
<S>   ::= <NP> <VP> | <S> and <S>
<VP>  ::= beat <NP>
<NP>  ::= <N> | <NP> and <NP>
<N>   ::= Ali | Bev | Jim | Zoe
```

a Modify the grammar to make it suitable for *recursive descent* parsing.

b Write down the FIRST*, LAST* and FOLLOWER sets for the above grammar, and hence or otherwise show that there is a context clash and that the grammar is ambiguous.

c Use Miranda to write a recursive-descent parser which will consume sequences of tokens representing sentences in simple English, returning any unconsumed tokens. Your parser should produce an error message if the sentence was syntactically incorrect. You may use the following definitions:

```
word     == [ char ]
sentence == [ word ]
parser   == sentence -> sentence

parseN :: parser
parseN ( "Ali" : s ) = s
parseN ( "Bev" : s ) = s
parseN ( "Jim" : s ) = s
parseN ( "Zoe" : s ) = s
parseN   s           = error "bad noun"
```

The three parts carry, respectively, 15%, 40%, 45% of the marks.

5 An expression language for defining strings is described by the following BNF grammar:

```
<exp> ::= <number> * <exp> | <exp> + <exp> |
          <exp> & | ( <exp> ) | <string>
```

where **&** has the highest priority, ***** the next highest, and **+** the lowest, with parentheses overriding the priorities in the usual way. Both **+** and **&** associate to the *left*, but ***** associates to the *right*. It is proposed to write an *operator precedence parser* for the language.

a Write down the *precedence matrix* for the grammar above.

b Using Miranda, write an operator precedence parser and a suitable reduction function which will convert a list of tokens into an abstract syntax tree. Tokens are defined by:

```
token ::= Mul | Add | And | Open | Close |
          Num num | Str [ char ]
```

so that (*e.g.*) the program **(3 * thing &)** would be tokenised as:

```
[ Open , Num 3 , Mul , Str "thing" , And , Close ]
```

Abstract syntax trees are defined by:

```
ast ::= Exp token [ ast ] | Val token
```

Hence the example program above would have the abstract syntax tree:

```
Exp Mul [ Val ( Num 3 ) ,
          Exp And [ Val ( Str "thing" ) ] ]
```

Operator priorities are defined by:

```
rel ::= Gt | Eq | Lt | Err
```

and may be determined in your parser using the following function:

```
prio :: token -> token -> rel
prio t1 t2 = Gt  , if t1 ⋖ t2
           = Eq  , if t1 ≐ t2
           = Lt  , if t1 ⋗ t2
           = Err , otherwise
```

The two parts carry, respectively, 40%, 60% of the marks.

End of paper