UNIVERSITY OF LONDON

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 1997

BSc Honours Degree in Mathematics and Computer Science Part III
MSci 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 M3.38

LANGUAGE PROCESSORS

Friday, May 9th 1997, 2.00 - 4.00

Answer FOUR questions

For admin. only: paper contains 5 questions

1 A trainee programmer has written the following Turing program to locate a given character in an ordered array of characters using binary search:

```
type target : array 1 .. 7 of char

var table : target

procedure find ( key : char , low , high : int )
if high < low then
  put "key not found"
else
  var mid : int := ( low + high ) div 2
  if key > table ( mid ) then
    find ( key , mid + 1 , high )
  elsif key < table ( mid ) then
    find ( key , low , mid - 1 )
  else
●   put "key found at position " , mid
  end if
end if
end find
```

a Assuming that the array `table` contains the letters **abcdefg**, show the layout of store immediately before the execution of the **put** statement marked ● and resulting from the top-level call:

```
find ( 'b' , 1 , 7 )
```

b The project leader believes that using global variables makes programs run more slowly and asks the programmer to pass the array `table` to the procedure `find` using an extra parameter as follows:

```
procedure find ( table : target , key : ... )
```

Assuming that the recursive calls are changed to reflect the extra parameter, and that the top-level call is now written as:

```
find ( table , 'b' , 1 , 7 )
```

show the state of store again at ● and hence state whether the project leader's request was technically sound, taking into account both the instructions which are required to manipulate the stack and to access an element of the array.

c The project leader asks the programmer to make `table` a **var** parameter. Show the state of store again at ● following the top-level call of part b. Hence state whether this request is more technically sound than the previous one.

The three parts carry, respectively, 30%, 40%, 30% of the marks.

2 The syntax of **for**-loops and assignment statements in Turing is partially defined by the following grammar:

```
<forloop>     ::= for <ident> : <exp> .. <exp> <statseq> end for
<exp>         ::= <ident> | <value> | <exp> + <exp>
<statseq>     ::= <statement> <statseq> | <empty>
<statement>   ::= <forloop> | <assignment>
<assignment>  ::= <ident> := <exp>
<ident>       ::= a sequence of letters
<value>       ::= an integer
<empty>       ::= nothing at all
```

a Define a Miranda datatype suitable for representing the abstract syntax of the language fragment defined above.

b Write a Miranda expression to represent the abstract syntax of the following Turing program:

```
for i : 1 .. 10
  for j : i .. i + 10
     a := i + i
     b := a + j
  end for
end for
```

c Explain the semantics of **for**-loop evaluation and use Miranda to sketch an *interpreter* which implements them. Assume a data type `envir` for holding the values of variables and functions `lookup ::` <ident> `->` `envir` `->` <value> and `update ::` <ident> `->` <value> `->` `envir` `->` `envir` for retrieving and changing the values of variables held in it.

d Consider the statement `a := i + i` in the example program above and suggest (but *do not* program) an optimisation (rearrangement) of the abstract syntax tree which will speed up the interpretation of **for**-loops.

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

Turn over ...

3 A programming language provides Boolean values, variables and operations, but no comparison operations. In the compiler, Boolean expressions are represented as abstract syntax trees using the following Miranda datatypes:

```
name == [ char ]
exp ::= Val bool | Var name | Not exp |
        Or exp exp | And exp exp
```

a The target is a stack machine with logical operations and the following assembly language instructions:

```
inst ::= pushvar name   || push the value of a variable onto the stack
       | pushcon bool   || push the value of a constant onto the stack
       | logneg         || logical negation of the top stack element
       | logand         || logical and- of the top two stack elements
       | logor          || logical or- of the top two stack elements
```

Write a Miranda function `gena :: exp -> [ inst ]` to generate assembly language which will evaluate a Boolean expression on this machine.

b The target is a stack machine with conditional jumps (but *no* logical operations) and the following assembly language instructions:

```
inst ::= pushvar name   || push the value of a variable onto the stack
       | jump name      || branch unconditionally to a symbolic label
       | jumptrue name  || branch to a label if top stack element is True
       | label name     || defining occurrence of a symbolic label
```

Write a function `genb :: exp -> name -> name -> [ inst ]` to generate assembly language which tests the value of a Boolean expression and branches to one of two labels (the second and third parameters) as soon as the value of the expression is known to be `True`, and to the other as soon as it is known to be `False`. You may assume a "function" `unique -> name` which generates a unique character string each time it is applied.

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

4 The following grammar describes the syntax of postfix function application in the now-obsolete language POP-2:

```
<application>  ::= <arglist> . <ident> | . <ident>
<arglist>      ::= <arglist> , <argument> | <argument>
<argument>     ::= <number> | <ident> | ( <application> )
```

a Explain why the grammar is unsuitable for *recursive descent* parsing and convert it into a form which is suitable.

b Explain briefly what is meant by *ambiguity* in a grammar. Show by means of an example parse that the original grammar is ambiguous. Is your modified grammar from part a ambiguous?

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

```
token    ::= Dot | Comma | Open | Close
           | Num num | Id [ char ]
sentence == [ token ]
parser   == sentence -> sentence
```

The three parts carry, respectively, 20%, 30%, 50% of the marks.

5a Describe briefly how an *operator precedence* parser may be used to parse expressions. Explain the basic steps of the algorithm and the data structures used, but do not write any code.

b Explain briefly what is meant by an *operator grammar*.The following grammar describes the language of Miranda type expressions:

```
<typeexp>    ::= <type> -> <typeexp> | <type>
<type>       ::= <tupletype> | <listtype> | <simpletype>
<tupletype>  ::= ( <typeseq> , <typeexp> )
<typeseq>    ::= < typeseq> , <typeexp> | <typeexp>
<listtype>   ::= [ <typeexp> ]
<simpletype> ::= num | bool | char | ( <typeexp> )
```

Convert the grammar to operator form and derive the *precedence matrix.*

c Show the state of the input and the stacks during the operator precedence parse of the following Miranda type expression:

```
num -> ( num , [ char ] , num -> char ) -> bool
```

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

End of paper