Paper Number(s): **ISE2.8**

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE
UNIVERSITY OF LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2001

ISE PART II: M.Eng. and B.Eng.

LANGUAGE PROCESSORS

Wednesday, 9 May 2:00 pm

There are FIVE questions on this paper.

Answer THREE questions.

Time allowed: 2:00 hours

Examiners: Bailey,R.

1a Explain briefly (100 words each) the functions of each of the following components of a language-processing system:

i. Lexical Analyser

ii. Syntax Analyser (Parser)

iii Semantic Analyser (Type-Checker)

iv. Evaluator

v. Decompiler

vi Code Generator *[6 marks*

b Explain with examples what data structures are used in the operation of each of the components in *a* above. *[8 marks*

c Draw diagrams showing how the components of *a* above can be combined together into:

i. A Compiler

ii. An Interpreter

iii A Source-to-Source Translator

In each case you should indicate clearly which of the data structures of *b* above are used to communicate between the components. *[6 marks*

2 In a fragment of a language, the abstract syntax trees for expressions are defined by the following Haskell data type:

```
data Exp = Plus  Exp  Exp
         | Var   Name
         | Const Int
         | Index Name Exp
```

For example, the expression x + 1 is represented as:

```
Plus ( Var "x" ) ( Const 1 )
```

The Index node is used to reference elements of one-dimensional arrays; thus the expression A [x + 1] is represented as:

```
Index "A" ( Plus ( Var "x" ) ( Const 1 ) )
```

a Use Haskell to sketch the design of a simple code generator for expressions represented using this data type. The output from your code generator should be for a *zero-address* (stack) machine. State *clearly* any assumptions you make about the target instruction set. *[10 marks*

b Now consider the same expression language augmented with function calls taking *a single* parameter. The modified abstract syntax tree is as follows:

```
data Exp = Plus  Exp   Exp
         | Var   Name
         | Const Int
         | Index  Name Exp
         | Call  Name  Exp
```

Show how your code generator would be extended to handle this. *[10 marks*

3 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 Haskell data type for representing the *Abstract Syntax* of expressions written in the language. *[4 marks*

b Write an *Evaluator* which will evaluate semantically correct expressions (*i.e.* having operands of the correct types) written in the language. *[7 marks*

c Suggest an appropriate Haskell data type for representing the *type* of value represented by an expression, including the possibility that it may be semantically incorrect. *[3 marks*

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). *[6 marks*

4 A functional language has the following partial grammar for expressions:

```
<exp>  ::= <var> | <cond>
<cond> ::= <exp> if <exp> else <exp>
```

a Explain why the grammar above is unsuitable for top-down parsing and transform it into a version which is suitable. *[4 marks*

b Explain why the original version of the grammar is ambiguous, justifying your explanation by showing a parse tree of an ambiguous sentence or by formally manipulating the grammar. *[5 marks*

c State whether your modified grammar from part a is ambiguous and justify your answer. *[6 marks*

d Explain a strategy which can be used in a top-down parser to overcome such ambiguities. *[2 marks*

e Suggest how the concrete syntax of conditional expressions in the language might be redesigned to remove the original ambiguity. *[3 marks*

5 Boolean expressions in a programming language are described by the following BNF grammar:

```
<bexp> ::= <bexp> and <bexp>  | <bexp> or <bexp>
         |  neg <bexp>        | ( <bexp> )
         |  <variable>        | <constant>
```

where **neg** has the highest priority and associates to the *right*, **and** has a higher priority than **or** (both of which associate to the *left*) and parentheses override these priorities in the conventional way. It is proposed to write an *Operator Precedence Parser* for the language.

a Write down the *Precedence Matrix* for this fragment of the grammar. *[8 marks]*

b Assuming the following declarations:

```
data Token     = And | Or | Neg | Open | Close | Term
data Tree      = Node Token [ Tree ] | Leaf
type Sentence = [ Token ]
type Stack a   = [ a ]
```

Write a precedence parsing function `parse :: Sentence -> Tree` which will construct abstract syntax trees from syntactically correct tokenised input sentences.

For simplicity, both the terminal symbol tokens `<variable>` and `<constant>` are represented by the single constructor Term. You may also assume the existence of a predicate `Lessthan :: Token -> Token -> Bool` giving `true` if the <· relation holds between its first and second arguments. *[12 marks]*