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

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE
UNIVERSITY OF LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2000

ISE PART I: M.Eng. and B.Eng. and ACGI

LANGUAGE PROCESSORS

Monday, May 15 2000, 2:00 pm

There are FIVE questions on this paper.

Answer THREE questions.

All questions carry equal marks.

Time allowed: 2:00 hours

Corrected Copy

Examiners: Mr R. Bailey

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

[10 marks

b For the following Turing program, show (as Haskell expressions) the different program representations produced during translation from source to the assembly language of an idealised stack machine. State any assumptions you make, including any Haskell data definitions needed to clarify the expressions.

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

[10 marks

2 The language D allows expressions containing integer and Boolean *Constants*, integer *Addition* operations (adding two integer values), Boolean *Equality* operations (comparing two values of the same type), and *Selection* operations (evaluating one of two expressions of the same type according to the value of a Boolean expression).

a Suggest a Haskell data type for representing the *Abstract Syntax* of expressions in D.

[2 marks

b Write an *Evaluator* which will evaluate semantically correct D expressions (*i.e.* with operands of the correct types).

[6 marks

c Suggest a Haskell data type for representing the *Type* of an expression, including the possibility that it may be incorrect.

[2 marks

d Write a *Type Checker* which will find the type of a D expression, or report that it is incorrect (but *do not report the reason*).

[6 marks

e D also allows variables in expressions. Explain what additional data types and data structures are needed in the Abstract Syntax, the Evaluator and the Type Checker, but *do not write any code.*

[4 marks

3a Describe the way in which run-time storage is managed in languages which allow recursive procedures. Why is a language which allows only local and global references simpler to compile than one which allows procedures to be nested arbitrarily? Describe how Non-Local, Non-Global variables are accessed in the latter type of language.

[8 marks

b For the following Haskell program, indicate the scope of the variables and draw a diagram showing the state of the stack at the point immediately preceding the evaluation of the expression marked ●.

```
elsie m n
  = m * lacie n
    where lacie n = | n > 0       = n * lacie ( n - 1 )
                    | otherwise = m + 5                 -- ●
elsie 4 3
```

[12 marks

4 The syntax of Boolean expressions in a language is defined as follows:

```
<exp>   ::= <exp> <op> <exp> | <term>
<op>    ::= and | or
<term>  ::= <ident> | not <exp> | ( <exp> )
<ident> ::= a sequence of letters
```

a Explain what is meant by *Ambiguity* in a grammar. Show by means of an example or otherwise that the grammar above is ambiguous.

[4 marks

b Explain why the grammar is unsuitable for *Recursive Descent* parsing.

[2 marks

c Transform the grammar so that it is suitable for recursive descent parsing

[4 marks

d Assuming the following definitions:

```
type Token    = [ Char  ]
type Sentence = [ Token ]
type Parser   = Sentence -> Sentence
```

use Haskell to write a recursive descent parser which will consume correctly formed expressions, and terminate with an error message otherwise. You may assume the existence of `isIdent :: Token -> Boolean` to identify an `<ident>` token.

[8 marks

e Is your transformed grammar from *c* above ambiguous? Justify your answer.

[2 marks

5a Explain briefly how an *Operator Precedence* parser works. Explain the basic steps of the algorithm and the data structures used, but do not write any code.

[6 marks

b Explain briefly what is meant by an *Operator Grammar*. A language allows *postfix* function applications with the syntax:

```
<app>     ::= <args> . <ident>
<args>    ::= <exp> | ( <exp> <explist> ) | <empty>
<explist> ::= , <exp> <explist> | <empty>
<exp>     ::= <id> | <app>
<id>      ::= a sequence of letters
<empty>   ::= nothing at all
```

Modify this grammar to make it into an operator grammar and write down a suitable *Precedence Matrix* for the language.

[8 marks

c Show the states of the input and of the stacks during the operator precedence parse of the following function application in the language:

```
( a , b . f , c ) . g . h
```

[6 marks