UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 2000

BEng Honours Degree in Computing Part I
MEng Honours Degrees in Computing Part I
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 City and Guilds of London Institute

PAPER C120

DECLARATIVE PROGRAMMING

Friday 5 May 2000, 14:00
Duration: 90 minutes
(Reading time 5 minutes)

Answer THREE questions

Paper contains 4 questions

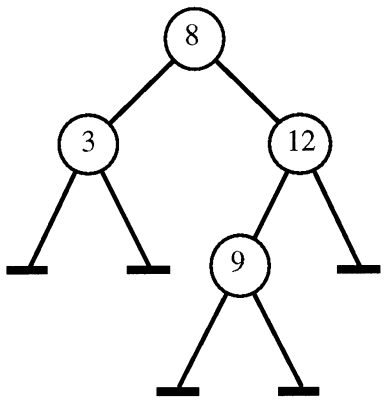
1a Explain, with examples, the difference between *polymorphism* and *overloading* in Haskell.

b Explain, with examples, how a Haskell `data` statement can be used to introduce new user-defined data types. For such data types explain the role played by

(i) Constructor functions
(ii) Pattern matching

c (i) Define a *binary tree* data type `Btree` with two constructors suitable for storing and accessing items of arbitrary type held in the internal nodes of the tree.

(ii) An *ordered* binary tree satisfies the property that for every internal node all the elements of the left sub-tree are smaller than or equal to that at the node, and all the elements in the right sub-tree are greater than that at the node. Write down a Haskell expression which represents the following ordered tree:

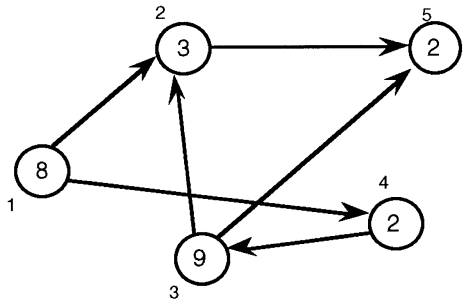


(iii) Define a function `union` which will combine two ordered binary trees to produce a new ordered tree containing all the elements present in the original trees. The individual trees contain no duplicate elements but they may share elements in common; your function should remove any such duplicates in the combined tree.

(iv) What is the most general type of your `union` function in part iii?

Note: the three parts carry 20%, 20% and 60% of the marks respectively. For part c. the four sections are weighted 1:1:3:1 respectively.

2 A *directed acyclic graph* comprises a set of *nodes*, each with a unique identifier and an associated node value, and a set of directed *edges*, each connecting a source node to a destination node. An example of a directed graph is shown below. The node values (here these are integers) are shown inside the nodes and the node identifiers are written next to the node. For example, node 3 contains the integer value 9 and has edges connecting it to nodes 2 and 5. Note that in general the node values may be of any type and that the graph contains no cycles (every node satisfies the property that there is no path through the graph from that node back to itself).



Directed graphs like this can be represented in Haskell as a list of pairs, each pair containing the identifier of a node together with a list of the identifiers of the other nodes to which that node is connected, thus:

```
type Graph a = [ ( Int, ( a, [Int] ) ) ]
```

a Define a function `nedges :: Graph a -> Int` which returns the total number of edges in a given graph. Given the above graph, for example, the function should return 6.

b Define a function `nodes :: Graph a -> [ ( Int, a ) ]` which returns the list of node identifiers, and their associated values, in a given graph. Given the above graph, for example, the function should return the list of pairs [(1,8), (2,3), (3,9), (4,2), (5,2)] in some order.

c Define a function `getedges :: Graph a -> Int -> [ Int ]` which given a graph and the identifier of a node in the graph returns a list containing the identifiers of the nodes to which the given node is connected. Given the above graph and the node identifier 3, for example, the function should return the list [2,5] in some order.

d Define a function `reachable :: Graph a -> Int -> Int -> Bool` which given a graph and the identifiers of a target node and a source node respectively, will return `True` if the target node can be reached from the source node by traversing the edges of the graph in some order; `False` otherwise. For example, given the above graph and the identifiers 3 (target) and 1 (source) the function should return `True` by virtue of the path 1→4→3. However, given the identifiers 4 and 5, for example, it should return `False`.

Credit will be awarded for clarity and conciseness, and the appropriate use of higher-order functions throughout.

Note: the four parts carry 15%, 20%, 30% and 35% of the marks respectively.

Turn over...

3 a `[X|T]` is the list pattern for a list of at least one element. Give the list pattern for a list that has:

(i) at least one element, where that element is itself a list of at least one element
(ii) at least two elements
(iii) at least three elements, where the second element is the empty list.

b The following is a Prolog program which defines the relation number_on(N,L) which holds when N is a number on a list L:

```
number_on(N,[N|T]) :- number(N).
number_on(N,[_|T]) :- number_on(N,T).
```

For example, `number_on(X,[1,2,[3],4])`

has the answers `X=1, X=2, X=4` but not the answer X=3 because 3 is not at the top level on the list, it is on a sublist.

Give a Prolog program for the relation number_somewhere_on(N,L) which holds when N is a number on L, or is a number somewhere on a sublist of L

For example, the query:

`number_somewhere_on(E,[1,[2,[3,4],5],0,-3,[],9,[[3]])`

should have the answers: `E=1, E=2, E=3, E=4, E=5, E=0, E=-3, E=9, E=3` given in that order.

c. Assume that a Prolog database has facts for the relations:

part(P)	P is a part item
supplies(M,P):	manufacturer M supplies part P
located_in(M,T)	manufacturer M is located in town T
direct_component_of(P1,P2)	part P1 is a direct component of part P2

Making use of *findall*, *forall*, and *not* where appropriate, define the following relations in terms of the above data base relations:

(i) component_of(P1,P2) P1 is a direct component part of P2 or there is a direct component P of P2 of which P1 is a component

(ii) supplies_parts(M,L) manufacturer M supplies all and only those parts on the list L.

(iii) supplies_all(M,P) manufacturer M supplies part P and all its components

(iv) atomic_part(P) P is a part with no direct component

The three parts carry respectively 15%,30%,55% of the marks.

4 a A *tree* is an atom, or of the form tree(T1,T2) where T1 and T2 are trees. Thus, tree(tree(a,b),c) is a tree. The leaf profile of such a tree is the list of atoms appearing at its tips, in left to right order. Thus, [a,b,c] is the leaf profile of tree(tree(a,b),c) and of tree(a,tree(b,c)). The leaf profile of a tree that is an atom A, is the list [A].

Consider the relation:

leaf_profile(T,L) L is the list of atoms which is the leaf profile of tree T.

i) Give a two-clause recursive definition of this relation that uses the app relation defined by:

```
app([],Y,Y).
app([U|X],Y,[U|Z]) :- app(X,Y,Z).
```

You can also use the Prolog primitive, atom(A), for testing if a term is an atom.

ii) Give a three-clause recursive definition of the relation that does not use the app relation. Your definition should only use recursive calls to leaf_profile and the atom test primitive.

[Hint: treat separately the case when the left subtree is an atom.]

b Suppose a Prolog data base contains facts for the relations:

student(S)	S is the name of a student
takes(S,C)	S takes course C
lectures(L,C)	L lectures on course C
lecturer(L)	L is a lecturer

Using *forall, not findall* if appropriate:

(i) Write a single Prolog clause defining a 0-arity predicate check1 which holds if and only if there is no student who does not take at least one course.

(ii) Write a single Prolog clause defining a 0-arity predicate check2 which holds if and only if every course being taken by at least one student is has at least one lecturer.

(ii) Write a single Prolog clause defining a 0-arity predicate check3 which holds if and only if every lecturer lectures on at least one course that has at least 10 students. (For you answer to this part you can make use a relation length(L,N) which holds when N is the length of list L without defining it.)

(iv) Define the relation

untaught_by(S,L) there is no course C that S takes that is lectured on by L

The two parts of this question carry equal marks