UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 2001

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

Wednesday 21 March 2001, 10:00
Duration: 180 minutes

Answer TWO questions

Paper contains 2 questions
Calculators not required

1 This question concerns the game of Mastermind in which a person tries to work out secret code represented as a collection of coloured pegs. The exercise requires you to build some of the supporting functions that would be required in a full Haskell implementation of the game.

> **The Game** is generally played with two people. The first sets a secret (hidden) code using a specified number of coloured pegs. In this version of the game there are three colours: red (R), blue (B) and green (G) and the secret code comprises an arbitrary number, n say, of these. Duplicates are allowed so, for n=3 for example, valid codes include RRG, RBG, BBB and so on. The second player then tries to guess the code and each guess is marked using *black* and *white* scoring sticks. If they guess a colour which is correct and in the correct position they score one black stick; if they guess a colour which is correct but in the wrong place they score one white stick. Black sticks are scored first and the corresponding pegs in both the secret code and the guess are eliminated before scoring the white sticks. For example, if the code is RGR and the guess is RBB the score is 1 black and 0 white, written (1,0). A guess of RBG would score (1,1), RRR (2,0) and so on.

The colours can be represented as an enumerated data type in Haskell thus:

```
data Colour = R | B | G deriving ( Eq, Ord, Show )
```

The secret code and each guess can both be represented as a list of colours and the result of a given guess as a pair comprising the guess and its corresponding score, a score being the count of the number of black and white sticks respectively. Thus:

```
type Colours = [ Colour ]

type Score = ( Int, Int )

type Result = ( Colours, Score )
```

a Define a function `colour :: Int -> Colour` that will convert a given `Int` to a `Colour`, assuming the mapping 0 -> red (R), 1 -> blue (B) and 2 -> green (G). You may assume the precondition that the given integer is either 0, 1 or 2.

b Define a function `base3 :: Int -> Int -> [ Int ]` which given two numbers `x` and `n` will convert `x` to its base 3 representation (this is like binary except each digit may assume the value 0, 1 or 2). For example, given the integer `32` the function should return the list `[1,0,1,2]` i.e. $1\times3^3+0\times3^2+1\times3^1+2\times3^0$. The final list should contain exactly `n` digits so, for example, `base3 0 4` should deliver `[0,0,0,0]`. A precondition is that `n` is at least as large as the number of digits required to represent `x`.

c Define a function `blacks :: Colours -> Colours -> Int`, which given a guess and a secret code will determine the number of black sticks the guess should score. For example, given the Haskell lists `[R,B,G]` and `[R,R,G]` representing the guess and the secret respectively, the result should be `2`. A precondition is that the two lists are of the same length.

d If `s` is the secret code and `g` is a guess then the number of pegs in `g` that score neither white nor black can be computed using Haskell's list difference operator, viz. `g \\ s`. (The `\\` operator is defined in the List module, but this has been imported for you via the `import` statement in the given template.) If `|l|` denotes the number of elements in a list `l` (the `length` function in Haskell) then the number of pegs which *do* score either white or black

is `n-|g\\s|` where `n=|g|` is the number of pegs in the guess (and code). This must equal the sum of the number of black (`b`) and white (`w`) pegs so the number of white pegs is given by `w = n-|g\\s|-b`. Using this, and the function `blacks` from part c define a function `score :: Colours -> Colours -> Result` which given a guess and a secret code will return the correct `Result` for the guess. For example, `score [G,G,R] [B,G,G]` should return `( [G,G,R], (1,1) )`.

e Using the functions `base3` and `colour` define a function `allGuesses :: Int -> [ Colours ]` which given an integer `n` will generate all possible guesses of length `n` using the three colours R, B and G (including duplicates). For example, `allGuesses 2` should generate, in some order, the list `[ [R,R], [R,B], [R,G], [B,R], [B,B], [B,G], [G,R], [G,B], [G,G] ]`. Hint: each guess will be a colour encoding of the base-3 representation of an integer between 0 and 3^n-1 inclusive. Thus, to generate the above list you would first need to build the list of two-digit base 3 numbers, i.e. `[ [0,0], [0,1], [0,2], ..., [2,2] ]` and then map the digits to colours.

f Suppose we (ultimately) want the computer to guess the secret code. A simple way to construct a next guess is to try all possible guesses (part e) and strike out those that are inconsistent with the given list of scores. It works like this: take each possible guess, assume it is the same as the secret code and then re-score each of the previous guesses; if the scores produced are the same as those previously recorded then the guess is consistent. Thus, define a function `consistent :: Colours -> [ Result ] -> Bool` which given a guess and the list of results from all previous guesses returns `True` if the guess would have produced the observed list of results; `False` otherwise. A precondition is that the list of results is correct with respect to the secret code.

g Using the function `consistent` from part f define a function `strike :: [ Colours ] -> [ Result ] -> [ Colours ]` to implement the striking-out described in part f. Again, a precondition is that each (previous) guess has been correctly scored. For example, if the secret code is `[R,G,R]` and the guesses `[R,B,B]` and `[R,G,G]` have been scored correctly (`(1,0)` and `(2,0)`) respectively) then the expression `strike ( allGuesses 3 ) [ ([R,B,B],(1,0)), ([R,G,G],(2,0)) ]` should yield the list `[ [R,R,G], [R,G,R] ]`, i.e. the only two codes that can possibly be correct given the list of `Results`.

Note: The seven parts carry 2, 3, 4, 4, 3, 6 and 3 marks respectively. An additional 5 marks will be awarded for the appropriate use of Haskell's higher-order functions and list comprehensions.

2 This question is concerned with analysing Prolog representations of simple English sentences conforming to the following grammar:

```
a sentence comprises a nounphrase followed by a verbphrase;
a nounphrase comprises a qualifier followed
    either by a noun
    or else by an adjective followed by a noun;
a verbphrase comprises either a verb
    or else a verb followed by an adverb.
```

The words allowed in the sentences, together with their types, are determined by this set of Prolog facts:

```
qualifier(a).       noun(cat).      adjective(black).
qualifier(the).     noun(dog).      adjective(white).

verb(runs).         adverb(quickly).
verb(walks).        adverb(slowly).
```

Open the file **prolex.pl** already set up for you. It includes the above facts and also defines various predicates `tester1(Z)`, `tester2`, etc. that you can query to test the code you create. **Do not alter the given code in this file.**

The final state of the file should contain **all** the programs created by you.
You can test them using either the suggested queries or your own queries.
Read the **Hints for Survival** set out on the last page of this question.

a One way of representing a sentence is by a list term containing the words in their intended order, as in the example `[a, cat, runs, slowly]`.

Create in prolex.pl a non-recursive program defining the predicate snlist(Z) which holds when list term Z represents a sentence conforming to the given grammar.

You can, if you wish, test this first program by executing the pre-defined query

```
?- tester1(Z).
```

Each solution should print a list term representing some sentence.

b Any list can be represented by a set of facts stating notional positions between which members occur. For example, `[a, cat, runs, slowly]` can be represented by

```
item(a, 1, 2).          "a"    occurs between positions 1 and 2
item(cat, 2, 3).        "cat" occurs between positions 2 and 3
item(runs, 3, 4).         etc.
item(slowly, 4, 5).
```

Create in prolex.pl a program defining the predicate transf(Z) which, given **any** list term Z (not necessarily representing a sentence), generates its fact-oriented representation. As a guide, you might first write the following clause:

```
transf(Z) :- transf1(Z, 1).      % to initiate a recursion starting with position 1
```

and then write a recursive `transf1` program that calls `assert` to generate `item` facts.

You can, if you wish, test all your code so far by executing the pre-defined query

```
?- tester2.
```

Each solution should print both representations of some list representing a sentence.

c Create in prolex.pl a non-recursive program defining the predicate snpos(J, K) which holds when the words between positions J and K in a fact-oriented representation form a sentence. This program must use no list terms at all, instead relying only upon clauses in this style:

```
snpos(J, K) :- nounphrasepos(J, M), verbphrasepos(M, K).
nounphrasepos(J, K) :- qualifierpos(J, M), nounpos(M, K).
:
nounpos(J, K) :- item(cat, J, K).
:
etc.
```

Note that the most basic phrase-types should be defined directly in terms of item, as shown above in the clause defining nounpos.

You can, if you wish, test all your code so far by executing the pre-defined query

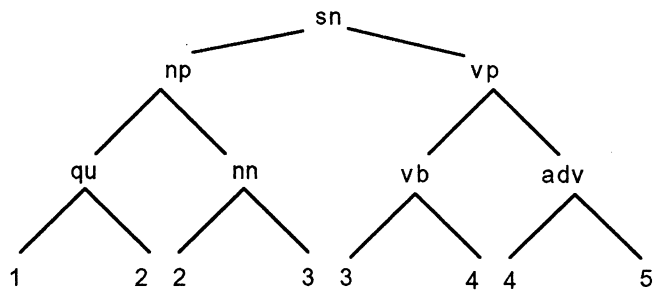
```
?- tester3.
```

Each solution should print both representations of a list together with one or more pairs (J, K) of positions between which a sentence occurs. Note that the example shown in part b would yield two such pairs (1, 4) and (1, 5).

d Create in prolex.pl a non-recursive program defining the predicate sntree(J, K, T) which holds when a sentence occurs between positions J and K in a fact-oriented representation and T is a tree representing the structure of that sentence. Here is an example in which the tree T would be

```
sn(np(qu(1, 2), nn(2, 3)), vp(vb(3, 4), adv(4, 5)))
```

```
item(a, 1, 2).
item(cat, 2, 3).
item(runs, 3, 4).
item(slowly, 4, 5).
```



The leaf nodes on the tree's frontier are positions (numbers), whilst the allowed non-leaf nodes and their interpretations are as follows:

sn	sentence	nn	noun
np	nounphrase	adj	adjective
vp	verbphrase	vb	verb
qu	qualifier	adv	adverb

As a guide to the writing of this program, it might begin as follows:

```
sntree(J, K, sn(T1, T2)) :- nptree(J, M, T1), vptree(M, K, T2).
nptree(J, K, np(qu(J, M), nn(M, K))) :- qualifierpos(J, M), nounpos(M, K).
etc.
```

You can, if you wish, test all your code so far by executing the pre-defined query

```
?- tester4.
```

Each solution should print some list in both representations together with one or more trees representing sentences occurring in that list.

e Create in prolex.pl a Prolog program defining the predicate frontier(T, F) which, given **any** tree T whose leaves are numbers, returns a list F representing the frontier of T. For example:

```
if      T is      sn(np(qu(1, 2), nn(2, 3)), vp(vb(3, 4), adv(4, 5)))
then    F is      [1, 2, 2, 3, 3, 4, 4, 5]
```

Note—to find the frontier of a compound tree, first find all its subtrees' frontiers and then append them together. The base case defines the "frontier" of a number N to be [N]. This program needs to use the Prolog primitives =.. and findall.

You can, if you wish, test all your code so far by executing the pre-defined query

```
?- tester5.
```

Each solution should print some list in both representations together with one or more trees (and their frontiers) representing sentences occurring in that list.

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

Hints for Survival

Take care to avoid trivial syntax errors such as mismatched brackets; mis-spelled predicate symbols; illegal spaces between predicate symbols and left brackets; or the omission of **full-stops** from the ends of clauses or queries.

After entering or editing material in prolex.pl, always **resave** and **recompile** the file before attempting new querying. Do not delete programs already completed.

If your programs compile correctly but compute wrong results, debug them by inserting appropriate writenl calls enabling you to print out appropriate terms at intermediate stages in the execution.

Useful Prolog primitives include the following:

append(X, Y, Z)	appending list Y onto list X gives list Z
member(X, Y)	X is a member of list Y
X is E	binds X to the numeric value of the arithmetic expression E
number(X)	tests whether X is a number
compound(X)	tests whether X is a compound term
X =.. L	given X, returns L as the list whose first member is the principal functor in X and whose remaining members are the functor's arguments, if any e.g. if X is f(a, b) then L is [f, a, b]
findall(T, P, L)	returns L as the list of all values of term T for which the call-term P is true
assert(F)	adds (invisibly) a dynamic fact F
retractall(item(_, _, _))	deletes all previously asserted 3-ary item facts (your programs should not need to use this, but you can call it at any time if you want to)