Coláiste na Tríonóide, Baile Átha Cliath
Trinity College Dublin
Ollscoil Átha Cliath | The University of Dublin

Faculty of Engineering, Mathematics and Science

School of Computer Science and Statistics

Integrated Computer Science Programme **Trinity Term 2017**
B.A. (Mod.) Computer Science and Business
B.A. (Mod.) Computer Science & Language
Mathematics
Year 3 Annual Examinations

Symbolic Programming

Monday 8th May 2017 **RDS Main Hall** **09:30 – 11:30**

Dr Tim Fernando

Instructions to Candidates:

Attempt *two* questions. All questions carry equal marks. Each question is scored out of a total of 50 marks.

You may not start this examination until you are instructed to do so by the Invigilator.

Materials permitted for this examination:

Non-programmable calculators are permitted for this examination – please indicate the make and model of your calculator on each answer book used

1. (a) Specify Prolog's response to the following queries (taking for granted the built-in predicates `member`, `setof`, `findall` and `number`).

   (i) `member(0,[1,2]).` False as 0 is not in the list [1,2]

   (ii) `member(X,X).` X = [X|_8282] This instantiates X to a list with X as the head.

   (iii) `setof(X,member(X,X),L).` STACK OVERFLOW Repeatedly tries to find the occurrences of X in member(X) which is unlimited
   % setof(X, Y, L) find's all X's that satisfy Y and returns them in L

   (iv) `findall(X,\+member(X,X),L).` L = []
   % Findall is like setof except when it fails instantiates L to []

   (v) `number(X).` False as X is a variable and not a number.

   (vi) `[a|[b,c]] = .(a,.(b,X)).` Error as the first term is not a valid list, otherwise it would return X=C

   [12 marks]

   (b) What is an anonymous variable, and why is it also called a singleton variable? The anonymous variable in prolog is the _ symbol and it is used in predicate and logic when you do not care about a value. Say if you only cared about the head of a list you could just do [H|_]. It is also called the singleton variable as this is an error that warns you that a variable is named and not used, instead it should be replaced with _.

   [6 marks]

   (c) Define a binary predicate `lessSome(List1,List2)` that is true exactly if `List1` and `List2` are lists of numbers, and some member of `List1` is less than some member of `List2`.

```
lessSome(L1, L2) :- isNumberList(L1), isNumberList(L2), hasLess(L1,L2).
isNumberList([]).
isNumberList([H|T]) :- number(H), isNumberList(T).
isLessThan(_, []) :- false.
isLessThan(X, [H|T]) :- X < H; isLessThan(X, T).
hasLess(_, []) :- false.
hasLess([H1|T1], L2) :- isLessThan(H1, L2); hasLess(T1, L2).
```

   [8 marks]

   (d) Define a binary predicate `lessAll(List1,List2)` that is true exactly if `List1` and `List2` are lists of numbers, and every member of `List1` is less than every member of `List2`. same as (C) except remove false from isLessThan and hasLess

```
isLessThan(_, []).
hasLess(_, []).
```

   [10 marks]

   (e) This question is about 3-ary predicates that, to a first approximation, are true of three lists when the third list is the union of the first two lists. The simplest example is the predicate `append(List1,List2,List3)` that is true precisely when `List3` is `List1` followed by `List2`. (Note that append is sometimes called `concatenate`.)

   (i) Define `append(List1,List2,List3)` so that, for example,

```
?- append([1,2,3],[2],L).
L = [1,2,3,2].
```

```
append([], L, L).
append([H1|T1], L2, [H1|T3]) :- append(T1, L2, T3).
```

   [4 marks]

(ii) To remove duplications in `List3` (and ensure `List3` has no repeating members), we might use `setof` two different ways as follows:

```
union1([],[],[]).
union1(L1,L2,L3) :-
        setof(X,(member(X,L1);member(X,L2)),L3).

union2([],[],[]).
union2(L1,L2,L3) :-
        setof(X,(member(X,L1);member(X,L2)),U),
        setof(X,member(X,L3),U).
```

Union 1 works well, it first checks that X is a member of L1, if that doesn't succeed it checks if it is a member of L2 and returns the list L3 of all members
Union 2 also works well but in this predicate you would have to define the union as L3 when calling it as it depends on L3 to be a list and not a variable.

Explain how `union1` and `union2` differ by giving examples of queries handled properly by one but not the other, and vice versa.

[5 marks]

(iii) What unary predicate `p(List)` can be used below to combine the predicates `union1` and `union2` as follows.

```
union(L1,L2,L3) :-  (p(L3),!,union1(L1,L2,L3));
                    union2(L1,L2,L3).
```

union(L1, L2, L3) :-
(var(L3), !, union1(L1, L2, L3));
union2(L1, L2, L3)

Are there terms t, t', t'' such that the Prolog interpreter says yes (or `true`) to `append`(t, t', t''), but complains that the query `union`(t, t', t'') results in an error? Explain.

[5 marks]

2. (a) Define a binary predicate `sum(List,Sum)` that is true exactly if `List` is a non-empty list of numbers that add up to `Sum`. For example,

```
?- sum([1,2],X).
X = 3
```

```
sum(List, Sum) :- validList(List), sumOfList(List,0, Y), Y =:= Sum.
validList([]).
validList([H|T]) :- number(H), validList(T).
sumOfList([], Acc, Acc).
sumOfList([H|T], Acc, Sum) :- NewAcc is Acc + H, sumOfList(T, NewAcc, Sum).
```

[5 marks]

A tail-recursive predicate is a predicate where the recursive call is the last function invoked in the evaluation of the body of the function

(b) The predicate `length(List,N)` below computes the length of a list.

```
length([],0).
length([_|T],N) :- length(T,M), N is M+1.
```

What is a *tail-recursive* predicate, and how can we redefine the predicate `length(List,N)` so that it becomes tail-recursive.

B

[6 marks]

(c) Define a predicate `split(+Number,+List,?Small,?Big)` that is true exactly whenever

- `Number` is a member of `List`
- `List` is a list of numbers
- `Small` is the list of all numbers in `List` smaller than `Number`
- `Big` is the list of all numbers in `List` bigger than `Number`.

For example,

```
?- split(2,[1,2,3,0,5,7],Small,Big).
Small = [1,0], Big = [3,5,7].
```

```
split(N, L, S, B) :- member(N, L), isNumberList(L), isSmall(S,L,N), isBig(B,L,N).
isNumberList([]).
isNumberList([H|T]) :- number(H), isNumberList(T).
isSmall(SmallList, List, Number) :- findall(X, (member(X,List), X < Number), SmallList).
isBig(BigList, List, Number) :- findall(X, (member(X,List), X > Number), BigList).
```

[15 marks]

(d) Define a predicate `median(List,Median)` that holds precisely when

- `List` is a list of odd length where each member of `List` is a number that occurs exactly once in `List`, and
- `Median` is the median of `List` — i.e., there are as many members of `List` that are smaller than `Median` as there are members of `List` that are bigger than `Median`.

```
median(L,M) :- odd(L), noOnce(L), isMedian(L,M).
odd(List) :-
    length(List,Len),
    \+ mod(Len,2) =:= 0.
isMedian(List,Median) :-
    findall(X, (member(X, List), X < Median), SmallerList),
    findall(Y, (member(Y, List), Y > Median), BiggerList),
    length(SmallerList, SLLen),
    length(BiggerList, BLLen),
    SLLen =:= BLLen.
noOnce(List) :-  noOnceAcc(List,[]).
noOnceAcc([H|T], Acc) :- \+member(H, Acc), append([H], Acc, NewAcc), noOnceAcc(T, NewAcc).
noOnceAcc([], _).
```

[10 marks]

(e) Define a predicate `remove(X,List,Rest)` that is true exactly when `X` is a member of `List` and `Rest` is the list resulting from removing `X` from `List`. For example,

```
?- remove(2,[1,2,3,0,5,7],Rest).
Rest = [1,3,0,5,7].
```

[7 marks]

(f) Use the predicate `remove(X,List,Rest)` to define a predicate `permute(+List1,?List2)` that is true exactly when `List2` is a permutation of `List1` (that is, `List2` differs from `List1` at most by a reordering of its members).

[7 marks]

```
remove(X,L,R) :- member(X,L), removeAcc(X,L,[],R).
removeAcc(_,[],Acc,Acc).
removeAcc(Element, [H|T], Acc, CleanList) :-
   H \== Element,
   append([H], Acc, NewAcc),
   removeAcc(Element, T, NewAcc, CleanList).
removeAcc(Element, [_|T], Acc, CleanList) :- removeAcc(Element, T, Acc, CleanList).
```

```
insert(X, L1, L) :- remove(X, L, L1). % what do you remove from L to give L1
permute([X], [X]).
permute([H|T], L) :-
  permute(T, T1),
  insert(H, T1, L).
```

3. This question is about regular expressions over the alphabet $\{0,1\}$. An example, with alternation (or choice) written | (also sometimes written +), is 0(0|11)*11 which picks out the set of strings of the form

$$0^{n_1}1^{2m_1}0^{n_2}1^{2m_2}\ldots 0^{n_k}1^{2m_k}$$

for some positive integer k, and positive integers $n_1, m_1, n_2, m_2, \ldots, n_k, m_k$. For example, the smallest string in this set is 011, which we shall represent in Prolog as the list `[0,1,1]`.

(a) Define a DCG that generates the aforementioned set of strings so that for example,

```
?- s([0,1,1,0,0,0,1,1],L).
L=[0,0,0,1,1] ? ;
L=[] ? ;
no.
```

[10 marks]

(b) Write out the DCG in part (a) as ordinary Prolog clauses, making the difference lists explicit. What are difference lists and why are they useful?

[10 marks]

(c) To generalize the construction of the DCG in part (a) to arbitrary regular expressions over the alphabet $\{0,1\}$, let us agree to use the binary functors `c`, `a` and `k` for concatenation, alternation and Kleene star (respectively) so that for example, 0|11 can be encoded as `a(0,c(1,1))`, and (0|11)* can be encoded as `k(a(0,c(1,1)))`. For completeness, let us use the constant `e` for the empty set (consisting of no strings), and `n` for the set consisting (solely) of the string `[]` of length 0. Now, the idea is to add an argument to the symbol `s` in the part (a), which we can fill by any regular expression over $\{0,1\}$ (under the encoding above) so that, for example,

```
?- s(c(1,1),L,[]).
L=[1,1] ? ;
no.
?- s(a(0,c(1,1)),L,[]).
L=[0] ? ;
L=[1,1] ? ;
no.
```

```
?- s(k(a(0,c(1,1))),[0,1,1],T).
T=[0,1,1] ? ;
T=[1,1] ? ;
T=[] ? ;
no.
```

Define a DCG for this 3-ary predicate s.

[15 marks]

(d) A regular expression such as 0*1*, encoded above as `c(k(0),k(1))`, has infinitely many strings, not all of which may appear as Prolog answers the query below.

```
?- s(c(k(0),k(1)),L,[]).
L=[] ? ;
L=[1] ? ;
L=[1,1] ? ;
L=[1,1,1] ? ;
...
```

Missing from the enumeration above is `[0,1]` even though

```
?- s(c(k(0),k(1)),[0,1],[]).
yes.
```

Describe concisely how to revise the predicate s to a predicate sr so that for any regular expression R and any string x in R, we need only type ; enough times, as the Prolog interpreter processes the query `sr(`R`,L)` before L is set to x. For example, the string `[0,0,0,1,1]` should be bound to L at some finite point below.

```
?- sr(c(k(0),k(1)),L).
L=[] ? ;
...
L=[0,0,0,1,1]
```

[15 marks]