yz709-prolog-sup2

Question 1

Slides have the take(L1, X, L2) function that succeeds if List L2 is L1 minus element X. Implement a take_all(L1, X, L2) that succeeds if list L2 is L1 minus all the occurrences of element X. Can you do it with cut? And without it?

```
% take_all(+L1,+X,-L2) succeeds
% if List L2 is L1 minus all occurrence of element X.
take_all([],_,[]).
take_all([H|T],H,R) :- !,take_all(T,H,R).
take_all([H|T],X,[H|R]) :- take_all(T,X,R).
```

```
% take_all(+L1,+X,-L2) succeeds
% if List L2 is L1 minus all occurrence of element X.
take_all([],_,[]).
take_all([H|T],H,R) :- take_all(T,H,R).
take_all([H|T],X,[H|R]) :- H =\= X, take_all(T,X,R).
```

💡 Comments:

Question 2

9.2 (Shallow)

Review the example of the predicate whose logical interpretation is changed by the use of cut. Come up with your own example.

```
% p <=> c ⋁ (a ⋀ b)
p :- c.
p :- a, b.

% p <=> c ⋁ (¬c ⋀ a ⋀ b)
p :- !, c.
p :- a, b.
```

💡 Comments:

10.3 (Shallow)

State and explain Prolog’s response to the following queries, In those cases where Prolog says ’yes’ your answer should include the unified result for X.

> X = 1.

- X unifies with atom 1.

```
?- X = 1.
X = 1.
```

> not(X=1).

- For all X, we have X cannot unify with 1.

```
?- not(X = 1).
false.
```

> not(not(X=1)).

- There exists an X, such that X unifies with atom 1.

```
?- not(not(X = 1)).
true.
```

> not(not(not(X=1))).

- For all X, we have X cannot unify with 1.

```
?- not(not(not(X = 1))).
```

💡 Comments:

12.2 (Deeper)

Implement a relation choose(N,L,R,S) which chooses N items from L and puts them in R with the remaining elements in L left in S

```
% choose(+N,+L,-R,-S) succeeds if N items are chosen from L into R
% and S contains the rest.
choose(0,L,[],L).
choose(N,[H|T],[H|R],S) :- N > 0, N1 is N -1, choose(N1,T,R,S).
choose(N,[H|T],R,[H|S]) :- N > 0, choose(N,T,R,S).

?- choose(2,[1,2,3,4],L,R).
L = [1, 2],
R = [3, 4] ;
L = [1, 3],
R = [2, 4] ;
L = [1, 4],
R = [2, 3] ;
L = [2, 3],
R = [1, 4] ;
L = [2, 4],
R = [1, 3] ;
L = [3, 4],
R = [1, 2] ;
```

💡 Comments:

12.3 (Shallow)

Add additional clauses to the symbolic evaluator for subtraction and integer division (this is the // operator in Prolog i.e. 2 is 6//3) and then get the countdown game working.

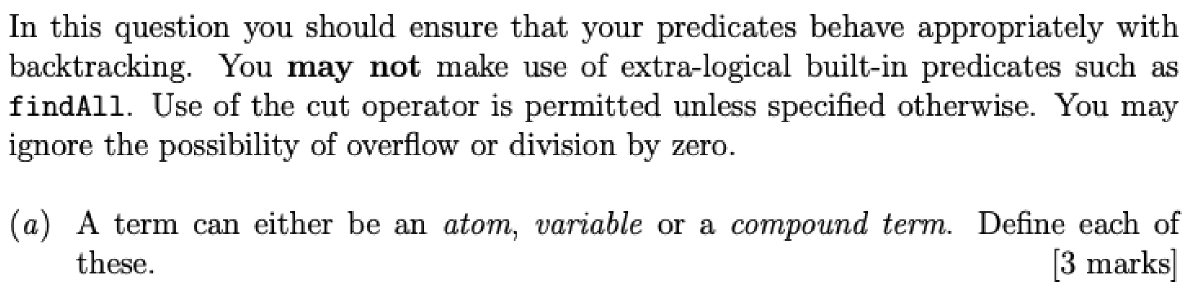
```
% choose(N,L,R,S) succeeds if R is the result of choosing N items from L
% and S is the remaining left in L
choose(0,L,[],L).
choose(N,[H|L],[H|R],S) :- N > 0, N1 is N - 1, choose(N1,L,R,S).
choose(N,[H|L],R,[H|S]) :- N > 0, choose(N,L,R,S).

% eval(A,B) succeeds if the symbolic expression A evaluates to B
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 + B1.
eval(minus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 - B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 * B1.
eval(divide(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 / B1.
eval(A,A) :- number(A).
```

```
% helper functions
notOne(A) :- eval(A,Av), Av =\= 1.
notZero(A) :- eval(A,Av), Av =\= 0.
isFactor(A,B) :- eval(A,Av), eval(B,Bv), 0 is Bv rem Av.

% arithop(A,B,C) succeeds if C is a valid combination of A and B
% e.g., arithop(A,B,plus(A,B)).
arithop(A,B,plus(A,B)).
arithop(A,B,minus(A,B)) :- eval(A,A1), eval(B,B1), A1 > B1.
arithop(A,B,minus(B,A)) :- eval(A,A1), eval(B,B1), B1 > A1.
arithop(A,B,mult(A,B)) :- notOne(A), notOne(B).
arithop(A,B,divide(A,B)) :- notOne(B), notZero(B), isFactor(B,A).
arithop(A,B,divide(B,A)) :- notOne(A), notZero(A), isFactor(A,B).

% countdown(L,T,S) succeeds if the header of L is the solution
% for the target symbolic expressions
countdown([Soln|_], Target, Soln) :- eval(Soln, Target).
countdown(L, Target, Soln) :- choose(2, L, [A,B], R),
                              arithop(A,B,C), countdown([C|R], Target, Soln).
```

Comments:

Question 3 Past paper questions

y2018p7q8


In this question you should ensure that your predicates behave appropriately with backtracking. You **may not** make use of extra-logical built-in predicates such as `findAll`. Use of the cut operator is permitted unless specified otherwise. You may ignore the possibility of overflow or division by zero.

(*a*) A term can either be an *atom*, *variable* or a *compound term*. Define each of these. [3 marks]

- Atom is any sequence of letters and digits that begin with a lower case letter or any sequence of characters enclosed by a single quote.
- Variable beginning with a capital letter, written as a sequence of letters and digits, can begin with an underscore.
- The compound term is consists of a functor followed by a sequence of arguments, and those arguments are wrapped inside parenthesis. e.g., `foo(a,b)`, we can also have syntactic sugar for infix operators, e.g., `X is 1 + 2` is actually `is(X, +(1,2))`.

(*b*) Euclid's algorithm for computing the greatest common divisor of two integers can be implemented in ML as:

```
fun gcd(a,0) = a
 |  gcd(a,b) = gcd(b, a mod b);
```

Provide an implementation in Prolog without using the cut operator.
[4 marks]

```
% gcd(+A,+B,-R) succeeds if R is gcd(A,B).
gcd(A,0,A).
gcd(A,B,R) :- B > 0, C is A mod B, gcd(B,C,R).
```

(*c*) We can represent fractions using the compound term `div/2`. For example `div(1,3)` represents $\frac{1}{3}$.

Implement a predicate `simplify` which transforms a fraction into its smallest exact representation. For example, `simplify(div(8,4),B)` should unify `B` with 2, and `simplify(div(4,8),A)` should unify `A` with `div(1,2)`. Your predicate should avoid unnecessary computation. [5 marks]

```
simplify(div(A,1),A).
simplify(div(A,B),div(A,B)) :- gcd(A,B,1).
simplify(div(A,B),R) :- gcd(A,B,GCD), GCD =\= 1, Av is A / GCD,
                      Bv is B / GCD, simplify(div(Av,Bv),R).
```

(*d*) We can also represent arithmetic expressions involving addition, subtraction, multiplication and division. For example, the expression $3\frac{5}{2-1} + 4$ is represented as `add(mul(3,div(5,sub(2,1))),4)`.

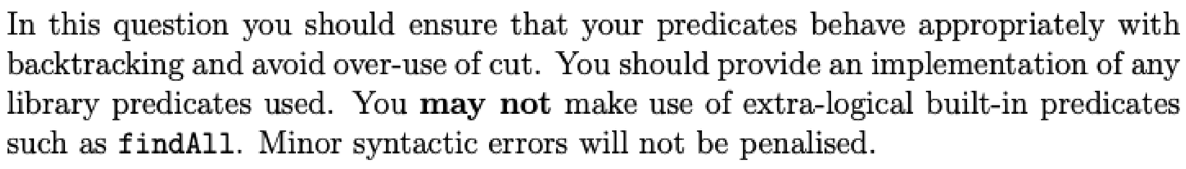
Implement a predicate `reduce` which reduces an arithmetic expression to its smallest exact representation e.g. `reduce(add(div(1,2),div(1,4)),A)` should unify `A` with `div(3,4)`. [8 marks]

```
% reduceUtil(A,B) succeeds if the symbolic expression A evaluates to B
reduceUtil(add(A,B),C) :- !, reduceUtil(A,div(A1,A2)), reduceUtil(B,div(B1,B2)),
              Num is A1 * B2 + B1 * A2, Denom is A2 * B2, C = div(Num, Denom).
reduceUtil(sub(A,B),C) :- !, reduceUtil(A,div(A1,A2)), reduceUtil(B,div(B1,B2)),
              Num is A1 * B2 - B1 * A2, Denom is A2 * B2, C = div(Num, Denom).
reduceUtil(mul(A,B),C) :- !, reduceUtil(A,div(A1,A2)), reduceUtil(B,div(B1,B2)),
              Num is A1 * B1, Denom is A2 * B2, C = div(Num, Denom).
reduceUtil(div(A,B),C) :- !, reduceUtil(A,div(A1,A2)), reduceUtil(B,div(B1,B2)),
              Num is A1 * B2, Denom is A2 * B1, C = div(Num, Denom).
reduceUtil(A,div(A,1)).
```

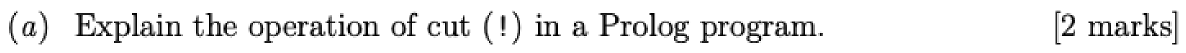
```
reduce(E,A) :- reduceUtil(E,B), simplify(B,A).
```

Comments:

y2016p3q7



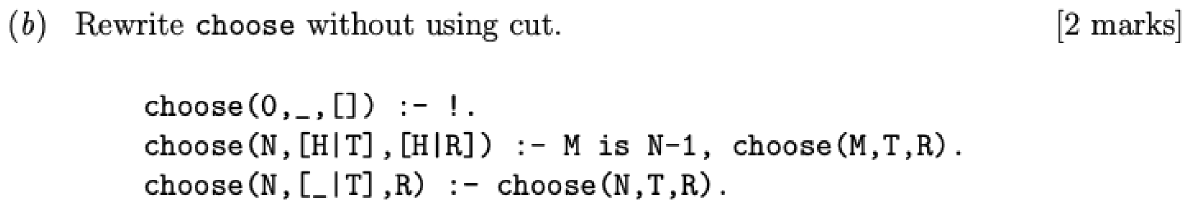
In this question you should ensure that your predicates behave appropriately with backtracking and avoid over-use of cut. You should provide an implementation of any library predicates used. You **may not** make use of extra-logical built-in predicates such as `findAll`. Minor syntactic errors will not be penalised.



(*a*) Explain the operation of cut (`!`) in a Prolog program. [2 marks]

- **Purpose**: closes choice points, prune search tree, prevent unwanted backtracking, avoid extra solutions or generating errors.
- Cut interpretations using **procedural box model**: (1) when you cross a cut operator, it commits you to the rule you are evaluating and any parts of the rule you have done so far; (2) removes all the choices that come before the cut in the rule body, removes the direct parent choice point that causes you to try that instance of the rule in the first place; (3) backtracking through a cut fails the entire procedure
- We have red cut that changes the logical meaning of the program and green cut that only boosts the program efficiency without changing the logical meanings.

```
% p <=> (a ⋀ b) ⋁ c                  % p <=> (a ⋀ b) ⋁ (¬a ⋀ c)
p :- a, b.                          p :- a, !, b.
p :- c.                             p :- c.
% p <=> a ⋁ b                        % p <=> a
p :- a.                             p :- !, a.
p :- b.                             p :- b.
```



(*b*) Rewrite `choose` without using cut. [2 marks]

```
choose(0,_,[]) :- !.
choose(N,[H|T],[H|R]) :- M is N-1, choose(M,T,R).
choose(N,[_|T],R) :- choose(N,T,R).
```

```
% choose(+N,+L,-R) succeeds if N items are chosen from List L into List R.
choose(0,_,[]).
choose(N,[H|T],[H|R]) :- N > 0, M is N - 1, choose(M,T,R).
choose(N,[_|T],R) :- N > 0, choose(N,T,R).
```

(*c*) Explain the operation of **not** (also written as \+) in a Prolog program. [1 mark]

- Negation by failure (`not(A)` or `\+(A)`) succeeds if `call(A)` fails, hence used to produce one solution semantics

```
% a program always fails, it is built-in called fail
fail :- !, 1 = 2
not(A) :- A, !, fail.
not(_).
% example: not(expensive(R))
% expensive(R) means ∃R. expensive(R)
% not(∃R. expensive(R)) = ∀R. ¬ expensive(R)
```

(*d*) Rewrite **chooseAll** without using **not** and cut (**!**). [10 marks]

```
chooseAll(N,L,Res) :- chooseAll(N,L,[],Res).
chooseAll(N,L,Seen,Res) :- choose(N,L,R),
                           not(member(R,Seen)), !,
                           chooseAll(N,L,[R|Seen],Res).
chooseAll(_,_,Res,Res).
```

- For each element in the list, we either choose it or not. Only append the `Picked` elements if the length of the `Picked` list is `N` as required.

```
% chooseAll(N,L,Res) succeeds if Res is a list of lists contains all possible
% ways of choosing N items from List L
chooseAll(N,L,Res) :- chooseAll(N,L,[],Res).
chooseAll(0,_,Picked,[Picked]).
chooseAll(N,[],_,[]) :- N > 0.
chooseAll(N,[H|T],Picked,Res) :- N > 0, M is N - 1,
    chooseAll(N,T,Picked,NCL), chooseAll(M,T,[H|Picked],CL), append(NCL,CL,Res).

% some tests
?- chooseAll(2,[1,2,3,4],R).
R = [[4, 3], [4, 2], [3, 2], [4, 1], [3, 1], [2, 1]] ;
?- chooseAll(2,[1,2,3,1,1],R).
R = [[1, 1], [1, 3], [1, 3], [1, 2], [1, 2], [3, 2], [1, 1], [1|...], [...|...]|...] ;
```

(*e*) What is *Last Call Optimisation* and why is it beneficial? [3 marks]

- **A space optimisation technique**: we jump to evaluating the next layer of a predicate rather than doing it with stack recursion.

- LCO is only possible if the **rule is determinate at the point where it is about to call the last goal in the body of the clause**, i.e., there is **no further searching needed** on account of determinacy in case the last call fails.

(*f*) Rewrite pos to enable Last Call Optimisation. [2 marks]

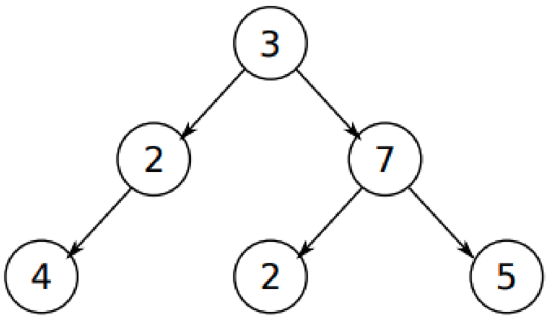
```
pos([],[]).
pos([H|T],[H|R]) :- H >= 0, pos(T,R).
pos([H|T],R) :- H < 0, pos(T,R).
```

```
% pos(L,R) succeeds if R contains all positive values from L
pos([],[]).
pos([H|T],[H|R]) :- H >= 0, !, pos(T,R). % no further searching needed
pos([_|T],R) :- pos(T,R).
```

💡 Comments: in part (d), I cannot deal with repeated elements, any tips on that?

y2014p3q8

You are asked to write a Prolog program to work with binary trees. Your code should not rely on *any* library predicates and you should assume that the interpreter is running without occurs checking.



(*a*) Describe a data representation scheme for such trees in Prolog and demonstrate it by encoding the tree shown above. [3 marks]

- Use a relation `treeNode(NodeValue, Lc, Rc)` to represent each Node, where `Lc` and `Rc` are relation `treeNode/3` as well, the leaf nodes are represented as `treeNode(LfValue, nil,nil)`.
- An empty tree is `nil`

```
treeNode(3,treeNode(2,treeNode(4,nil,nil),nil),
          treeNode(7,treeNode(2,nil,nil),treeNode(5,nil,nil))).
```

(*b*) Implement a Prolog predicate **bfs/2** which effects a *breadth-first* traversal of a tree passed as the first argument and unifies the resulting list with its second argument. For example, when given the tree shown above as the first argument the predicate should unify the second argument with the list `[3,2,7,4,2,5]`. [4 marks]

```
bfsUtil(nil,[]).
bfsUtil(treeNode(V,nil,nil),[V]).
bfsUtil(treeNode(V,Lc,nil),[V|LcRes]) :- bfsUtil(Lc,LcRes).
bfsUtil(treeNode(V,nil,Rc),[V|RcRes]) :- bfsUtil(Rc,RcRes).
bfsUtil(treeNode(V,Lc,Rc),Res) :- bfsUtil(Lc,LcRes), bfsUtil(Rc,RcRes), append([V|LcRes],RcRes,Res).

bfs(T,Res) :- bfsUtil(T,Res), !.
```

(*c*) Explain why the **bfs/2** predicate might benefit from being converted to use difference lists. [2 marks]

- Because we use the `append` program, which copies the list every time it is called, we can use a difference list to replace the `append` program and boost program efficiency and performance.

(*d*) Implement a new predicate **diffbfs/2** which makes use of a difference list to exploit the benefit you identified in part (c). Your predicate should take the same arguments as **bfs/2**. [6 marks]

- (1) write an answer doesn't require difference lists, (2) Rewrite with difference lists, (3) Rename variables to get rid of append

```
% after rewrite with difference lists
diffbfsUtil(nil,[]-[]).
diffbfsUtil(treeNode(V,nil,nil),[V|A]-A).
diffbfsUtil(treeNode(V,Lc,nil),[V|LcRes]-A) :- diffbfs(Lc,LcRes-A).
diffbfsUtil(treeNode(V,nil,Rc),[V|RcRes]-A) :- diffbfs(Rc,RcRes-A).
diffbfsUtil(treeNode(V,Lc,Rc),Res-A) :-
  diffbfsUtil(Lc,LcRes-B), diffbfsUtil(Rc,RcRes-C),
  append([V|LcRes]-B,RcRes-C,Res-A).

diffbfs(T,Res) :- diffbfsUtil(T,Res), !.

% after renaming, by unification append(A-B,B-C,A-C).
diffbfsUtil(nil,[]-[]).
diffbfsUtil(treeNode(V,nil,nil),[V|A]-A).
diffbfsUtil(treeNode(V,Lc,nil),[V|LcRes]-A) :- diffbfsUtil(Lc,LcRes-A).
diffbfsUtil(treeNode(V,nil,Rc),[V|RcRes]-A) :- diffbfsUtil(Rc,RcRes-A).
diffbfsUtil(treeNode(V,Lc,Rc),[V|LcRes]-A) :-
  diffbfsUtil(Lc,LcRes-RcRes), diffbfsUtil(Rc,RcRes-A).
```

```
diffbfs(T,Res) :- diffbfsUtil(T,Res), !.
```

(*e*) A friend observes that a clause in **diffbfs/2** will need to contain an empty difference list and proposes two possible ways of representing it, either **[]-[]** or **A-A**.

Consider your implementation of **diffbfs/2**. For each use of an empty difference list, justify your choice and explain what can go wrong using the alternative form. [2 marks]

- The empty list is only used when the tree is an empty tree, i.e., `T = nil`
- Uses `A-A`, the result is given out is a variable rather than an empty list `[]`, while using `[]-[]`, we terminate the list, it gives out `[]`.

```
% use []-[], i.e., diffbfsUtil(nil,[]-[]).
?- T = nil, diffbfs(T,Res-A).
T = nil,
Res = A, A = [].

% use A-A, i.e., diffbfsUtil(nil,A-A).
?- T = nil, diffbfs(T,Res-A).
T = nil,
Res = A.
```

(*f*) Is your implementation amenable to *last call optimisation* (LCO)? If so, explain why. If not, give details of the minimal changes you would make to make LCO possible. [3 marks]

- No, because in the rule `diffbfsUtil(treeNode(V,Lc,Rc),[V|LcRes]-A) :- diffbfsUtil(Lc,LcRes-RcRes), diffbfsUtil(Rc,RcRes-A).`, it is not deterministic at the point when it proved `diffbfsUtil(Lc,LcRes-RcRes)` is true, it creates a choice point, if `diffbfsUtil(Rc,RcRes-A)` returns false for the first try, then it backtracks to try out another possible rule.
- We can close this choice point by adding a cut between `diffbfsUtil(Lc,LcRes-RcRes)` and `diffbfsUtil(Rc,RcRes-A)`, thus the program will not generate unwanted backtracking.

💡 Comments: