ASSIGNMENT 2
DUE OCTOBER 19TH (OR 22ND), 2007

1. This question is concerned with combinatorial generation, and in particular the cool-lex algorithm for generating every \((s,t)\)-combination and every binary string of length \(n\).

   In Prolog an \((s,t)\)-combination is a list that has length \(n = s + t\) and contains \(s\) zeros and \(t\) ones. In the cool-lex ordering of \((s,t)\)-combinations the first list is \([1,1,...,1,0,0,...,0]\) (ie \(1^t0^s\)), the last list is \([0,1,1,...,1,0,0,...,0]\) (ie \(01^t0^{s-1}\)), and each subsequent list is obtained by removing the leftmost entry and reinserting it immediately to the right of the leftmost remaining ...,0,1,... In the case that there is no leftmost ...,0,1,... then the leftmost element is reinserted as the rightmost element.

   The program we developed in class, combo.pl, as well as sample output, are included in Appendix 1.

   In Prolog a binary string of length \(n\) is a list that has length \(n\) and contains zeros and ones. In the cool-lex ordering of binary strings of length \(n\) the first list is \([1,1,...,1]\) (ie \(1^n\)), the last list is \([0,1,1,...,1]\) (ie \(01^{n-1}\)), and each subsequent list is obtained by removing the leftmost entry and reinserting it immediately to the right of the leftmost remaining ...,0,1,... In the case that there is no leftmost ...,0,1,... then the leftmost element is reinserted as the rightmost element and its value is flipped (0 becomes 1, and 1 becomes 0).

   Your task is to implement the cool-lex ordering of binary strings in a file binary.pl. You may develop your own code or you may start with the existing code in combo.pl and follow the suggested steps below. In either case provide a listing of your program code, as well as sample output, are included in Appendix 1. [20 marks]

   i) Write a predicate \(\text{flip}(X,Y)\) that is true iff \(Y = 1 - X\).
   ii) Modify the predicate \(\text{continue}(B)\) to test against the last binary string of length \(n\) instead of the last \((s,t)\)-combination. (Hint: combo.pl uses \(\text{all01}\); you may want to create a slightly different predicate.)
   iii) Modify the predicate \(\text{next}([H|T],X)\) to give the next binary string in cool-lex order instead of the next \((s,t)\)-combination.

   BONUS. Implement the cool-lex ordering of balanced parenthesis and present your solution in a similar manner. Balanced parenthesis are strings with an equal number of left parenthesis and right parenthesis together with the property that no prefix contains more right parenthesis than left parenthesis. For example, we say that \((())())\) is a valid balanced parenthesis string, while \(()()()())\) is invalid. Balanced parenthesis are often represented by binary strings where 1 corresponds to ( and 0 corresponds to ).

   In Prolog a balanced parenthesis string of length \(2n\) is a list of length \(2n\) containing \(n\) zeros and \(n\) ones with the property that no prefix contains more ones than zeros. In the cool-lex ordering of balanced parenthesis the first list is \([1,1,...,1,0,0,...,0]\) (ie \(1^n0^n\)), the last list is \([1,0,1,1,...,1,0,0,...,0]\) (ie \(01^{n-1}0^{n-1}\)), and each subsequent list is obtained by removing the second leftmost entry and reinserting it immediately to the right of the leftmost remaining ...,0,1,... if that is valid, or otherwise immediately to the left of the leftmost remaining ...,0,1,... In the case that there is no leftmost ...,0,1,... then the leftmost element is reinserted as far to the right as is valid.

   For example, the listing when \(n = 4\) is

   \[
   11110000, 11100010, 11001010, 10101010, 11010010, 10110010, 11100100, ..., \\
   11001100, 10101100, 11010100, 10101000, 11011000, 11011000, 10111000.
   \]

   Provide the program code and sample output for \(n = 5\). For those who are curious, balanced parenthesis are counted by the Catalan Numbers and are equivalent to dozens of different objects including binary trees (search wikipedia for Catalan Number.) [10 marks]
2. This question is concerned with binary decision diagrams, and their use in solving difficult problems like independent set. During the initial parts of the question you will be focused on the graph $G$ that appears on the left side of Figure 1. To make these initial parts easier we will give sample answers for the example graph $G'$ that appears on the right hand side of Figure 1.

![Figure 1](image1.png)

**Figure 1.** Your graph $G$ is on the left, and the example graph $G'$ is on the right.

An *independent set* in a graph is a set of vertices that have no edge between them. The *unreduced independent set BDD* for a graph is an unreduced BDD that represents the independent sets of the graph. Proposition $i$ represents whether or not the vertex $v_i$ is in the independent set of the graph.

Each part is worth [5 marks].

a) Draw your unreduced independent set BDD $B$ for your graph $G$. Use the proposition order 1, 2, 3, 4 so that proposition 1 is at the root of your BDD. Figure 2 gives the example for graph $G'$.

![Figure 2](image2.png)

**Figure 2.** The example unreduced BDD $B'$.

b) Enter your BDD into a file. Below is how the example can be done.

```prolog
get_bdd(example, 
    bdd(1, 
        bdd(2, 
            bdd(leaf, t, x), 
            bdd(leaf, t, x) 
        ), 
        bdd(2, 
            bdd(leaf, t, x), 
            bdd(leaf, f, x) 
        ) 
    ) 
).
```

Use `consult` to load in your BDD and then display your BDD by using the `write_bdd` command. Below is how the example can be done.

```prolog
50 ?- get_bdd(example, Bexample), write_bdd(Bexample).
[1] v1  
[2] v2  
[3] t   
  [3]  
[4] v2  
  [3]  
[5] f   
```
Bexample = bdd(1, bdd(2, bdd(leaf, t, x), bdd(leaf, t, x)), bdd(2, bdd(leaf, t, x), bdd(leaf, f, x)))

Provide output that shows the result of write_bdd as well as how your BDD was input.

c) Use the three reduction rules to reduce your BDD by hand. During each step write out the current BDD and indicate which of the reductions rules you are applying. We will refer to your reduced BDD as R. Figure 3 shows how to reduce the example BDD.

![Figure 3](https://via.placeholder.com/150)

**Figure 3.** Creating the example reduced BDD $R'$ from the example unreduced BDD $B'$.

d) Instead of reducing your unreduced BDD by hand, this time use the command reduce and output the result using write_bdd. Provide your output. Below is how the example can be done

?- get_bdd(example,Bexample), reduce(Bexample,Rexample), write_bdd(Rexample).

```
[1] v1
[2] t
[3] v2
     [2] f
[4] f
```

Rexample = bdd(1, bdd(leaf, t, x), bdd(2, bdd(leaf, t, x), bdd(leaf, f, x)))

e) Take the output of write_bdd from the last question and draw it as a BDD. How does this reduced BDD compare to the reduced BDD that you constructed by hand? Explain why they are different or why they are the same.

f) In an independent set BDD what does the number of satisfying assignments mean in terms of the original graph? What is the value of numSAT for your unreduced BDD? Below is how the example can be done

?- get_bdd(example,Bexample), numSAT(Bexample,2,N).

N = 3

g) Currently the numSAT predicate only works correctly for unreduced BDDs. For example, here is the output for the example reduced BDD

?- get_bdd(example,Bexample), reduce(Bexample,Rexample), numSAT(Rexample,2,N).

N = 2

Recursively the current predicate works as follows. If the two children have $s_1$ and $s_2$ satisfying assignments starting from their respective nodes, then the parent is said to have $s_1 + s_2$ satisfying assignments. However, in reduced BDDs there can be variables that are skipped over so this is not the correct calculation. Figure 4 illustrates the issue for two versions of the example BDD, where the numbers next to each node are the number of satisfying assignments starting from that node. Now let us consider how to fix the problem. Suppose the predicate numbers in the two children are $n_1$ and $n_2$ and the predicate number of the parent is $n$. Then what is the correct number of satisfying assignments for the parent in terms of $s_1$, $s_2$, $n_1$, $n_2$, and $n$? Your expression may assume that neither of the children are leaves.

h) BONUS: Modify numSAT so that it correctly computes the number of satisfying assignments. To do this correctly you should first reconsider your answer to the previous question. In particular, what is the correct expression when one of the children is a leaf. The answer depends on the total number of predicates, $NumPred$, in the formula that the BDD is representing. Provide your code and output that
Figure 4. Illustrating why numSAT gives an incorrect value for the reduced version of the example BDD.

shows that the predicate has been modified correctly. For output you may want to consider using get\_bdd and get\_reduced for sample BDDs.
APPENDIX 1

Note: Your output does not need to be in columns.

all0([]).
all0([H|T]) :-
    H = 0,
    all0(T).

has01([A,B|T]) :-
    ( A=0, B=1 ->
        true
    ;
        has01([B|T])
    ).

decreasing([]).
decreasing([H|T]) :-
    ( H=1 ->
        decreasing(T)
    ;
        all0(T)
    ).

continue([H|T]) :-
    ( H=1 ->
        true
    ;
        has01(T)
    ).

next([H|T], X) :-
    continue([H|T]),
    ( decreasing(T) ->
        append(T,[H],X)
    ;
        append(D,[0,1|C],T),
        decreasing(D),
        append(D,[0,1,H|C], X)
    ).

cool(A,C) :-
    next(A,C).
cool(A,C) :-
    next(A,B),
    cool(B,C).

?- cool([1,1,1,0,0,0],X).
X = [1, 1, 0, 0, 0, 1] ;
X = [1, 0, 0, 0, 1, 1] ;
X = [0, 0, 0, 1, 1, 1] ;
X = [0, 0, 1, 0, 1, 1] ;
X = [0, 1, 0, 1, 1, 1] ;
X = [0, 0, 1, 0, 1, 1] ;
X = [1, 0, 0, 1, 1, 0] ;
X = [0, 0, 1, 1, 1, 0] ;
APPENDIX 2

children(B,B1,B2) :-
   arg(2,B,B1),
   arg(3,B,B2).

numSAT(B,NumPred,Count) :-
   ( arg(1,B,leaf) ->
     ( arg(2,B,t) ->
       Count is 1
     ;
       Count is 0
     )
   ;
   children(B,B1,B2),
   numSAT(B1,NumVar,Count1),
   numSAT(B2,NumVar,Count2),
   Count is Count1 + Count2
 ).