ECE650 Written Assignment 2:
Boolean logic, normal forms, proof systems, solvers, parsers, grammars

Question 1: Short answer questions (15 points)

Please provide short and correct answers to the following questions. You will get points only if you provide appropriate justification. Otherwise you will get 0 points. Each sub-question is worth 5 points.

1. Consider a proof system P for Boolean logic such that given any formula F (valid or otherwise) in conjunctive normal form, P produces a proof of F. That is, P asserts that F is a theorem and hence valid. Is P a sound proof system? Is P a complete proof system? (State the definitions of soundness and completeness of proof systems in order to justify your answers.)

   - Soundness means that given a proof system P and an arbitrary formula F. P is sound would mean that it proved F to be valid, then F is absolutely valid.
   - Completeness means that given a proof system P and any arbitrary formula F. P is complete if F is valid, and P can indeed construct a proof for it, and if it fails to do so, then P is incomplete.
   - In this case, P is NOT sound but complete. Because as stated in the question, P can produce a proof of any formula F, whether it's valid or not. So if P could produce proof for invalid formula F, then it's not sound. On the other hand, since P could produce proof for any formula F, it means that it could produce proof for all valid formulas, whenever F is valid, P is able to construct a proof for it, and this means that P is complete.
   - Hence, as a result, P is complete but NOT sound.

2. Consider the semantic argument method discussed in class for Boolean logic. Alice implemented this method correctly (sound and complete) on her computer using recursion. Her friend Bob asked for the source code and saw some opportunities to optimize it. In the process, he accidentally commented out the recursive case for negation. Is Bob's implementation sound and complete? If yes, argue why Bob's system is both sound and complete. If not, provide an argument as to why Bob's system is unsound or incomplete. (The only change made by Bob you need to worry about is the accidental removal of the recursive case for negation.)

   You can assume that Alice's implementation of the semantic argument method has an input-processing step that performs the following steps in the order given below:

   - Checks if inputs are well-formed. If they are not well-formed, rejects such inputs.
   - For any well-formed input formula F, the input-processing step applies negation to F and converts $\neg F$ to NNF. Let us call the resultant formula G. The semantic argument method is then called on G.

   - In this case, Bob's implementation is sound but incomplete. Since the recursive case for negation is commented out, it means that the proof rule 1 – negation rule is no longer there. When the rule could be applied, given input ¬F, the method could try to deduce F and find a contradiction. However, since Bob's implementation still uses Alice's code, so it should still be sound. Although with the rule no long in use now, the proof system may not be able to deal with formulas that contain negations, and in the worst case, the proof could not be produced. But when it can come up with a proof, then it must mean the input formula is valid. However, for certain formulas, formulas that involves negation, since there's no proof rule for negation, Bob's implementation is not able to deal with the problem and provide proof for these certain formulas even if they are indeed valid. Hence, Bob's system is sound but incomplete.

3. Are context-free languages closed under infinite union (we define this as L = $L_1 \cup L_2 \cup L_3 \ldots$)? If yes, provide a proof. Else, provide a counter-example.
   - No, they are not closed under infinite union. Say we have language $\{a^n b^n c^n | n \geq 1\}$, and let L = $L1 \cup L2 \cup L3 \ldots$. In this case, L1 can be {abc}, L2 can be{aabbcc}, L3 can be {aaabbbccc}. In this case, all the individual L1, L2, L3, …. are context-free. However, the resultant L =$\{a^n b^n c^n | n \geq 1\}$ is not context-free. Hence it's not closed under infinite union.

Question 2: Semantics of Boolean Formulas (25 points)

Write a recursive program Eval that takes as input a Boolean formula F as a graph (defined in class) and a complete assignment A to all variables in F, and correctly evaluates the truth value of F on A and returns it. Please only provide the pseudo-code for Eval in an imperative language like C/Java, and describe its operation in detail. Assume that input formulas are syntactically correct (aka, *well-formed*).

For this one, assume the structure of graph is similar to that in PA1, and F is the pointer node that points to the root node of graph.

```
Eval (GraphNode* F, Assignment A){
        node = *F
        If node == NULL{
                return ;
        }
        If (node.content == "^"){
                lNode = Eval(node.left, A);
                rNode = Eval(node.right, A);
                return lNode && rNode;
        } else if (node.content == "v"){
                lNode = Eval(node.left, A);
                rNode = Eval(node.right, A);
                return lNode || rNode;
        } else if (node.content == "¬"){
                return !Eval(node.left, A);
        }else if (node.content == "→"){
                lNode = Eval(node.left, A);
                rNode = Eval(node.right, A);
                return !lNode || rNode;
```

```
        }else if (node.content == "↔"){
                lNode = Eval(node.left, A);
                rNode = Eval(node.right, A);
                return (lNode && rNode) ||
                        (!lNode && !rNode);
        }else if (node.type == "variable"){
                If A[node.content] != NULL{
                        return A[node.content];
                } else {
                        return error;
                }
        }else if (node.type == "constant"){
                return node.content;
        }else{
                return error;
        }
}
```

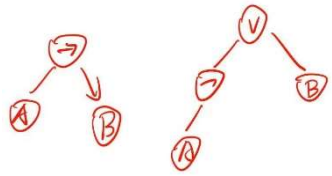
By assumption this eval function takes as input the pointer to the root node of the graph and the assignment map. With the pointer (which is assigned to node), first need to check if it's null, if so just return. And if node's content is one of "and", "or", "not", "implies", "iff", then need to perform corresponding recursive calls. For and, just return value of left child and right child. For or, just return value of left child or right child. For not, just return value of not left child. For implies, just return value of not left child or right child, and for iff, need to return left and right child, or not left child and not right child. And if the node is of type variable, then need to find its corresponding value inside the provided assignment map. Otherwise, if node's type is of constant, then it's value is either 1 or 0, and since it already has the value, we could just return it.

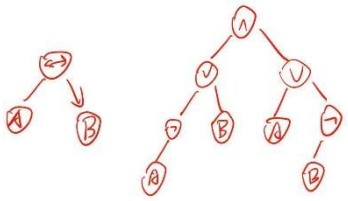
Question 3: Negation Normal Form (25 points)

Write a recursive program NNF() that takes as input a Boolean formula F as a graph (defined in class), and outputs an equivalent formula G such that $F \iff G$ and G is in NNF. Explain why your algorithm isequivalence-preserving.

For this one, assume the structure of graph is similar to that in PA1, and F is the pointer node that points to the root node of graph.

```
NNF1(Graphnode* F){
    Graphnode node = *F
    If (node.content == "¬"){
        NNF1(node.left);
    }else if (node.content == "^" || node.content == "v"){
        NNF1(node.left);
        NNF1(node.right);
    }else if (node.content == "→"){
        Graphnode* temp = node.left;
        node.content = "v";
        Graphnode leftSub = new Graphnode("¬");
        node.left = leftSub;
        node.left.left = *temp;
        NNF1(node.left);
        NNF1(node.right);
    }else if (node.content == "↔"){
        Graphnode* lTemp = node.left;
        Graphnode* rTemp = node.right;
        Graphnode lOr = new Graphnode("v");
        Graphnode rOr = new Graphnode("v");
        node.left = lOr;
        node.right = rOr;
        Graphnode lNot = new Graphnode("¬");
        Graphnode rNot = new Graphnode("¬");
        node.left.left = lNot;
        node.right.right = rNot;
        node.left.left.left = lTemp;
        node.left.right = rTemp;
        node.right.left = lTemp;
        node.right.right.left = rTemp;
        NNF1(node.left);
        NNF1(node.right);
    }else if (node.type == "constant" || node.type == "variable"){
        return;
```





```
        }else{
            return error;
        }
    }
    NNF2(Graphnode* F){
        Graphnode node = *F;
        If (node.content == "¬"){
            If (node.left.content == "¬"){
                return NNF2(node.left.left);
            }else if (node.left.content == "^"){
                node.left.content = "v";
                node.left.left.content = ! node.left.left.content;
                node.left.right.content = ! node.left.right.content;
                NNF2(node.left);
                NNF2(node.right);
            }else if (node.left.content == "v"){
                node.left.content = "^";
                node.left.left.content = ! node.left.left.content;
                node.left.right.content = ! node.left.right.content;
                NNF2(node.left);
                NNF2(node.right);
            }
        }
    }
```

For this, NNF1 first traverse through the graph and convert the graph to have only "and", "or", and "not" operators, and eliminates the "implies" and "iff" operators. Then, NNF2 can be called and used to traverse the graph again to push the negations inside clauses with the use of De Morgan's law. With these 2 recursive calls, the input formula could be transformed to NNF. And it's equivalence preserving because in NNF1, A → B is transformed to ¬A v B, ↔ is transformed to A → B ^ B → A. Later in NNF2, it only uses De Morgans law to push the negations in. Hence, it's equivalence preserving.

Question 4: DPLL SAT Solvers (20 points)

Provide the pseudo-code for the DPLL SAT solver discussed in class. Informally argue as to why the DPLL SAT solver is a decision procedure for the Boolean satisfiability problem. (In order to show that a program is a decision procedure for the Boolean satisfiable problem, you have to show two properties: First, you have to show that given a satisfiable input, the procedure will eventually find a satisfying assignment for it and terminate. Second, you have to show that given an unsatisfiable input, the procedure will eventually prove that it is unsatisfiable and terminate.)

```
bool DPLL(CNF φ, AssignMap A)
{
  1. φ' = BCP(φ,A)
  2. if(φ' = ⊤) then return SAT;
  3. else if(φ' = ⊥) then return UNSAT;
  4. p = choose_var(φ');
  5. if(DPLL(φ',A[p ↦ ⊤])) then return SAT;
  6. else return (DPLL(φ',A[p ↦ ⊥]));
}
```

-This is the pseudo-code for DPLL, it calls BCP in the first line. And here, in addition to doing unit resolution, BCP also detects whether clauses are being satisfied by the partial assignment A. In step 4, we can see the choose_var method that chooses a variable and assign it a value in step 5 and 6. With each new assignment, BCP will try to simplify the formula whenever it can. Since BCP can be described as unit resolution until saturation, we can know that all possible unit resolutions that can be applied have been applied. And in the end, when the formula is really simplified, it will determine the right value to return (whether it's SAT or UNSAT). And all possible assignments could be covered. For some formulas, DPLL could take exponential time. And in the worst case, it will have to use brute force to try all possible assignments for the variables. In this way, it makes sure that every possible assignment s covered, and by doing so, it's basically replicating a truth table, and it goes through every possible assignments. By doing so, it makes sure that given a satisfiable input, it will find the satisfying assignment and return the correct value and halt. It also makes sure that if given an unsatisfiable input, by trying all possible assignments, if none of them works with the output of BCP, then it would return UNSAT and terminate.

Question 5: Recursive-descent Parsers (15 Points)

Each of the sub-questions below are worth 5 points. Please provide detailed, correct and complete answers to get full points.

1. Why do recursive-descent parsers have to backtrack? What does backtracking mean? What are the consequences of backtracking in terms of performance? Describe performance in terms of worst-case time complexity. (The time taken by a parser to parse a string can be measured in terms of the size of the strings and number of productions in the corresponding grammar.)
   - Recursive descent parsers parse from start symbol of grammar to the string. However, the parser doesn't know which grammar rule to apply when parsing, and it have to try the rule one by one to see which rule match with the string. And when the chosen rule doesn't match with string, it will have to backtrack to the point where no conflicts appeared, and continue to try new rules from that point.
   - Backtracking means jump back to the point where things are fine, where no conflicts/problems appeared, and in this way, could then try new things and keep moving forward.
   - The consequences
   - The performance in terms of worst-case time complexity is bad. Since the parser doesn't know which grammar rule to apply, it'll have to try them one by one, and in the worse case, it would try all grammar rules and keep backtracking until it finds the right rule to apply. And in this case,

2. Explain, using an example, why left-recursion is a problem for recursive-descent parsers.
   - Left recursion is a problem for recursive-descent parsers because that it could lead to infinite recursion, which means it may never terminate. A simple example would be S → Sa. In this case, left recursion would lead to infinite recursion.
3. Consider the following grammar:

$$S \rightarrow AB$$
$$A \rightarrow S \mid \epsilon$$
$$B \rightarrow \epsilon$$

   Is the grammar as given left-recursive? If yes, then how would you rewrite it get rid of the left-recursion? If no, explain why not?

   - The given grammar is left-recursive because $S \rightarrow^{+} SB$.
   - In order to get rid of left recursion, need to rewrite it as:

$$S \rightarrow \beta S'$$
$$S' \rightarrow AB \mid \epsilon$$
$$A \rightarrow S \mid \epsilon$$
$$B \rightarrow \epsilon$$

Question 6: Tseitin Transformation and Conjunctive Normal Form (+2 extra credit)

Using structural induction prove that the input and output formulas of the Tseitin transformation algorithm are equisatisfiable to each other. Give an argument as to why the Tseitin transformation is a polynomial time algorithm. (Either provide an original proof or if you use a resource, then explain the proof in your own words and cite the resource.)