Assignment #3

Sample Solutions

Question 1: Programming the Invincible Tic-Tac-Toe Player

Before you tackle the World Isola Championships, let us consider a simpler game: Tic-Tac-Toe. The complete code for playing a game human vs. computer can be found in the file TicTacToe.hs (or TicTacToe.fr if you prefer) on the course homepage. Well, the code runs but the computer player is not very smart, to say the very least. It simply puts its next piece into the first (in reading direction) open square on the board.

(a) Your first task is to make the program smarter. Implement the minimax algorithm to improve the `findBestMove` function. This function receives the current state of the game and the number of moves to look ahead. It returns the move, i.e., an integer between 0 and 8, that the computer should play. You do not have to implement alpha-beta pruning.

When we look ahead using minimax, it is important that we do not just compute the best expected score but also figure out which move guarantees that move. Therefore, our recursive minimax function should not only return score information but also information about the corresponding moves. In the example code, this is accomplished by the `projectMoves` function returning a pair (move, score). Note that at the end we only need the move information for the first move to be made (at depth 0). Therefore, at each level, we only return the move information for that very level. It is also important to stop looking further ahead from a state where the game is over, i.e., one player has won or it is a tie. This would not make any sense and could lead to incorrect results.

```
findBestMove :: State -> Int -> Int
findBestMove state lookAhead = fst $ projectMoves lookAhead 0 state

projectMoves :: Int -> Int -> State -> (Int,Int)
projectMoves maxDepth currDepth state
    | maxDepth == currDepth = (-1, score state)
    | isWinner (pl1 state) = (-1, 1000)
    | isWinner (pl2 state) = (-1, -1000)
    | null moves = (-1, 0)
    | pl1Turn state = getMaxScore moves $ map (projectMoves maxDepth (currDepth + 1)) states
    | otherwise = getMinScore moves $ map (projectMoves maxDepth (currDepth + 1)) states
    where states = map (playTurn state) $ moves
          moves = genMoves state

getMaxScore :: [Int] -> [(Int, Int)] -> (Int, Int)
getMaxScore moves stateScores = maximumBy (comparing snd) currMoves
    where currMoves = zip moves (map snd stateScores)

getMinScore :: [Int] -> [(Int, Int)] -> (Int, Int)
getMinScore moves stateScores = minimumBy (comparing snd) currMoves
    where currMoves = zip moves (map snd stateScores)
```
(b) Bonus Question: Once the computer player is playing better, further improve its play by coming up with a more powerful evaluation function. Ideally, your program should be invincible if it looks only 5 moves ahead.

Obviously, there are many good solutions. In this example, those lines where a player has one piece and the two other squares are empty, the player scores one point. If the player has two pieces and the third square is empty, it counts three points. And if a player has three pieces... well, then that player has won and we return a large number. All these scores are added for the Max player, and then those for the Min player are subtracted. This function is definitely better than the one presented in the book, but there could still be better ones. Here it is:

```haskell
score :: State -> Int
score state = sum $ map scoreLine winLines
  where scoreLine line = lineScore (count (pl1 state) line) (count (pl2 state) line)
    count list1 list2 = length $ intersect list1 list2
    lineScore 1 0 = 1
    lineScore 2 0 = 3
    lineScore 3 0 = 500
    lineScore 0 1 = -1
    lineScore 0 2 = -3
    lineScore 0 3 = -500
    lineScore _ _ = 0
```

You can find the complete sample program on the course homepage as TicTacToe_minimax.hs.

**Question 2: Algorithmic Inference – Sledgehammer Style (required for CS670, bonus question for CS470)**

You now have all the tools you need to build your sledgehammer style inference algorithm. If your hypothesis is given by a string h (multiple hypotheses can be combined into one using AND connectives) and your conclusion is described by a string c, then your function

```
infer h c
```
will output either “valid” or “not valid for...” For example:

```
*Main> infer “P AND Q” “P”
valid
```

```
*Main> infer “P OR Q OR S” “P AND Q”
not valid for P, NOT Q, NOT S
```

Below is a sample solution. It makes use of the powerSet function that we developed earlier. The idea is to form the power set of our set of all variables in the given expression. Then we define that all variables in a given subset, i.e., a given element of the power set, are true and the remaining ones are false. This way we can easily create all possible interpretations (combinations of truth values).

Another useful trick is to connect the hypotheses and conclusion with an implication; this way we do not have to work with two expressions but just have to check whether this single expression is always true, regardless of the truth values of the variables. Well, and finally we need a function that traverses the expression tree to create a list of all variables.
infer :: [Char] -> [Char] -> [Char]
infer hypo conc = test interprets
where test [] = "valid"
    test (i:is)
        | eval exp $ vars i = test is
        | otherwise          = "not valid for " ++ i
    interprets = map (varList allVars) (powerSet allVars)
    allVars = getVars exp
    exp = parse $ "(" ++ hypo ++ ") IMP (" ++ conc ++ ")"

varList :: [Char] -> [Char] -> [Char]
varList allVars trueVars = init $ concat $ map vList allVars
where vList v
    | v `elem` trueVars = [v] ++ ","
    | otherwise         = "NOT " ++ [v] ++ ","

getVars :: Exp -> [Char]
getVars exp = sort $ nub $ gVars exp
where gVars (NOT exp)   = gVars exp
     gVars (AND e1 e2) = gVars e1 ++ gVars e2
     gVars (OR  e1 e2) = gVars e1 ++ gVars e2
     gVars (IMP e1 e2) = gVars e1 ++ gVars e2
     gVars (VAR v)     = [v]
     gVars _           = []

powerSet :: [a] -> [[a]]
powerSet [] = [[]]
powerSet (x:xs) = powerSet xs ++ map (x:) (powerSet xs)

Question 3: Resolving Problems – Paper and Pencil Style

Are the following arguments valid or not? Use resolution or resolution refutation to find out. Proceed in four steps:

- Extract the propositions from the argument and name them with single letters.
- Describe the hypotheses and the conclusion in propositional calculus.
- Convert these expressions into conjunctive normal form (CNF) suitable for resolution or resolution refutation (whichever method you decide to use).
- List the steps of the resolution (refutation) and state the result, that is, whether the argument is valid or not.

If you wrote the “infer” function, use it to verify your result (and the function).

a) If Kevin Flynn loses his disc or his AI system (CLU) cannot be turned off, Kevin Flynn is trapped on the grid. If CLU has no “off” switch, it cannot be turned off. Kevin Flynn does not lose his disk. CLU has no “off” switch. Therefore, Kevin Flynn is trapped on the grid.

b) Anne goes to the library on Saturday or on Sunday. John also goes to the library on Saturday or Sunday. Therefore, both Anne and John will be at the library on Saturday, or both of them will be at the library on Sunday.

c) If the AI lecture is boring, I will fall asleep. If I fall asleep, I will hit my head on the desk. If I hit my head on the desk, I will go to the hospital. I did not go to the hospital. Therefore, the AI lecture is not boring.

d) Jennifer is either watching a soccer game or a baseball game, or both. Whenever she watches a soccer game, she wears a Beckham t-shirt. Whenever she watches a baseball game, she wears a Red Sox scarf. Therefore, Jennifer is wearing a Beckham t-shirt and a Red Sox scarf.
a) If Kevin Flynn loses his disc or his AI system (CLU) cannot be turned off, Kevin Flynn is trapped on the grid. If CLU has no “off” switch, it cannot be turned off. Kevin Flynn does not lose his disk. CLU has no “off” switch. Therefore, Kevin Flynn is trapped on the grid.

Propositions:
D = Kevin Flynn loses his disc,
S = AI system cannot be turned off,
T = Kevin Flynn is trapped on the grid,
O = CLU has no “off” switch,

<table>
<thead>
<tr>
<th>Hypotheses</th>
<th>Propositional Calculus</th>
<th>CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(D OR S) IMP T</td>
<td>NOT D OR T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOT S OR T</td>
</tr>
<tr>
<td>2</td>
<td>O IMP S</td>
<td>NOT O OR S</td>
</tr>
<tr>
<td>3</td>
<td>NOT D</td>
<td>NOT D</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>

CNF expression for resolution refutation:
(NOT D OR T) AND (NOT S OR T) AND (NOT O OR S) AND (NOT D) AND O AND (NOT T)

Resolution on T:
(NOT D OR T) AND (NOT S) AND (NOT O OR S) AND (NOT D) AND O

Resolution on O:
(NOT D OR T) AND (NOT S) AND (S) AND (NOT D)

Resolution on S:
(NOT D OR T) AND FALSE AND (NOT D) which is always false, so the conclusion is valid.

b) Anne goes to the library on Saturday or on Sunday. John also goes to the library on Saturday or Sunday. Therefore, both Anne and John will be at the library on Saturday, or both of them will be at the library on Sunday.

Propositions:
A = Anne goes to the library on Saturday,
B = Anne goes to the library on Sunday,
J = John goes to the library on Saturday,
Q = John goes to the library on Sunday,

<table>
<thead>
<tr>
<th>Hypotheses</th>
<th>Propositional Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A OR B</td>
</tr>
<tr>
<td>2</td>
<td>J OR Q</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(A AND J) OR (B AND Q)</td>
</tr>
</tbody>
</table>

CNF expression for resolution refutation:
(A OR B) AND (J OR Q) AND (NOT A OR NOT J) AND (NOT B OR NOT Q)
Resolution on A:
(B OR NOT J) AND (J OR Q) AND (NOT B OR NOT Q)
Resolution on J:
(B OR Q) AND (NOT B OR NOT Q)
Resolution on B:
Q OR NOT Q which is always true so the conclusion was invalid.

c) If the AI lecture is boring, I will fall asleep. If I fall asleep, I will hit my head on the desk. If I hit my head on the desk, I will go to the hospital. I did not go to the hospital. Therefore, the AI lecture is not boring.

Propositions:
B = AI lecture is boring,
S = I fall asleep,
D = I hit my head on the desk,
H = I go to the hospital,

<table>
<thead>
<tr>
<th>Hypotheses</th>
<th>Propositional Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B IMP S</td>
</tr>
<tr>
<td>2</td>
<td>S IMP D</td>
</tr>
<tr>
<td>3</td>
<td>D IMP H</td>
</tr>
<tr>
<td>4</td>
<td>NOT H</td>
</tr>
<tr>
<td>Conclusion</td>
<td>NOT B</td>
</tr>
</tbody>
</table>

CNF expression for resolution refutation:
(NOT B OR S) AND (NOT S OR D) AND (NOT D OR H) AND (NOT H) AND B
Resolution on B:
S AND (NOT S OR D) AND (NOT D OR H) AND (NOT H)
Resolution on S:
D AND (NOT D OR H) AND (NOT H)
Resolution on D:
H AND (NOT H) which is always false, therefore the conclusion is valid.

d) Jennifer is either watching a soccer game or a baseball game, or both. Whenever she watches a soccer game, she wears a Beckham t-shirt. Whenever she watches a baseball game, she wears a Red Sox scarf. Therefore, Jennifer is wearing a Beckham t-shirt and a Red Sox scarf.

Propositions:
S = Jennifer is watching a soccer game,
B = Jennifer is watching a baseball game,
T = she wears a Beckham t-shirt,
R = she wears a Red Sox scarf,
<table>
<thead>
<tr>
<th>Hypotheses</th>
<th>Propositional Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S OR B</td>
</tr>
<tr>
<td>2</td>
<td>S IMP T</td>
</tr>
<tr>
<td>3</td>
<td>B IMP R</td>
</tr>
<tr>
<td>Conclusion</td>
<td>T AND R</td>
</tr>
</tbody>
</table>

CNF expression for resolution refutation:
(S OR B) AND (NOT S OR T) AND (NOT B OR R) AND (NOT T OR NOT R)

Resolution on R:
(S OR B) AND (NOT S OR T) AND (NOT B OR NOT T)
Resolution on B:
(S OR NOT T) AND (NOT S OR T)
Resolution on S:
T OR NOT T which is always true so the conclusion was invalid.
Question 4: Algorithmic Inference – Refutation Style

It should also be possible to implement the resolution refutation algorithm in Haskell without too much effort. The greatest problem is getting the logical expression into CNF. To make your life easier, I already came up with functions to remove all IMPs, limit the score of the NOTs, turn everything into CNF (i.e., prevent any AND from being a child of an OR), make a list of clauses, and then eliminate any double-listed variables in each clause. The function textToCNF applies all of these functions in sequence:

```haskell
textToCNF xs = map nub $ listCNF $ makeCNF $ limitNOT $ removeIMP $ parse xs
```

If we consider the example with Gary, once we added the negated conclusion for resolution refutation, we can get the whole thing into CNF using this function:

*Main> textToCNF "(I OR A) AND (I IMP C) AND NOT C AND NOT A"
[[I,A],[NOT I,C],[NOT C],[NOT A]]
it ::= [[Exp]]

Your task is to write a function “inferRR” that gets the same input as “infer” from Question 2, uses resolution refutation and either outputs “valid” or “not valid: remainder:…”

You can test your function with the arguments in Question 3.

As we now know, this problem is a bit tricky, and you will receive full credit if you only solved this question the way we initially discussed it. I wrote some code that actually performs a depth-first search to discover any possible solutions. I did not have much time to write or test the code. For example, a function that simplifies clauses and eliminates them if they contain both a positive and a negative literal of the same variable would be useful. Please let me know if you find any problems with the code. It does not print the remainder in case of an invalid argument, because various remainders can be discovered during the search.

The resolve function receives a CNF expression and the indices of two of its clauses plus the name of a variable to be (potentially) resolved upon. If this resolution is possible, it returns the resulting CNF. Otherwise, it returns an empty list. resoSearch performs depth-first search for generating an empty (false) expression. The empty clause is not actually generated, but the function isSolved checks whether the expression contains two single-literal clauses where one is the negation of the other. In that case, the expression must resolve to false, and we are done – the argument is valid. Here is the code:

```haskell
inferRR :: [Char] -> [Char] -> [Char]
inferRR hypo conc = if resoSearch $ textToCNF $ "(" ++ hypo ++ ") AND NOT (" ++ conc ++ ")" then "invalid" else "valid"

resoSearch :: [[Exp]] -> Bool
resoSearch cnf
  | isSolved cnf = False
  | otherwise = and $ map resoSearch newCNFs
where newCNFs = filter (not . null) $ map (resolve cnf) resInfo
      resInfo = [(i, j, v) | i <- [0..length cnf - 2], j <- [i+1..length cnf - 1], v <- cnf !! i]
```
resolve :: [[Exp]] -> (Int, Int, Exp) -> [[Exp]]
resolve cnf (i, j, v)
  | or $ map (checkLits v) (cnf !! j) = resolvedCNF cnf (i, j, absVar v)
  | otherwise = []
where checkLits x (NOT y) = x == y
checkLits (NOT x) y = x == y
checkLits _ _ = False
absVar (NOT (VAR v)) = VAR v
absVar (VAR v) = VAR v

resolvedCNF :: [[Exp]] -> (Int, Int, Exp) -> [[Exp]]
resolvedCNF cnf (i, j, v) = resolvedClause:remainingClauses
  where resolvedClause = filter notResVar ((cnf !! i) ++ (cnf !! j))
    notResVar exp = (exp /= v && exp /= (NOT v))
    remainingClauses = [cnf !! c | c <- [0..length cnf - 1], c /= i && c /= j]

isSolved :: [[Exp]] -> Bool
isSolved cnf = or $ map (\(i, j) -> isFalse (cnf !! i, cnf !! j)) indexPairs
  where indexPairs = [(i, j) | i <- [0..length cnf - 2], j <- [i+1..length cnf - 1]]
    isFalse ([u], [NOT v]) = u == v
    isFalse (NOT u, [v]) = u == v
    isFalse _ = False

You can find the code on the course homepage as infer.hs.