CS:4420 Artificial Intelligence
Spring 2018

Homework 3

Due: Friday, Mar 9 by 11:59pm

This is a programming assignment in F# to be done *individually*. Download the accompanying F# source file `hw3.fsx` and enter your solutions in it where indicated. When you are done, submit it through the Assignments section of ICON with the same name. Make sure you *write your name in that file* where indicated.

Each of your answers *must be free of static errors*. You may receive no credit for problems whose code contains syntax or type errors.

Pay close attention to the specification of each problem and the restrictions imposed on its solution. Solutions ignoring the restrictions may receive only partial credit or no credit at all.

You are welcome to use as needed your own helper methods or those provided in `hw3.fsx`. You are allowed in this problem to use any F# library methods you want. Those for lists may be most helpful.[1] Use recursion and pattern matching to write clean code.

1 Propositional Logic in F#

In this problem we model the language of propositional logic in F#. In particular, we will use the following types:

```
type id = int

type interpr = id list

type prop = True
          | False
          | P of id
          | Not of prop
          | Or of prop * prop
          | Xor of prop * prop
          | And of prop * prop
          | Impl of prop * prop
          | Iff of prop * prop

type answer = Yes | No
```

[1] `https://msdn.microsoft.com/library/a2264ba3-2d45-40dd-9040-4f7aa2ad9788`

The union type `prop` models propositional sentences (or, more concisely, *propositions*). Propositional symbols (variables) are encoded as terms of the form (`P` n) where n is an integer identifier (id). More complex propositions are constructed using the unary constructor `Not` and the binary constructors `Or`, `Xor`, `And`, `Impl`, and `Iff` standing respectively for the propositional connectives $\vee$, $\oplus$, $\wedge$, $\Rightarrow$, and $\Leftrightarrow$. The type `interpr` implements propositional interpretations as a F# list of propositional symbol ids (with no repetitions). A propositional symbol (`P` n) is meant to be true in an interpretation i if and only if n occurs in i.

Using these types, implement the methods below.

1. Write a function `meaning: interpr -> prop -> bool` that takes an interpretation `i` and a proposition `p`, and returns the F# Boolean value `true` if the interpretation satisfies the proposition according to the semantics of propositional logic, and returns the value `false` otherwise.

   For example, `meaning [1; 2] (And (P 1, P 2))` is `true`, while `meaning [1] (And (P 1, P 2))` is `false`.

2. The code in `hw3.fsx` defines a method `isValid: prop -> answer` that takes a proposition `p`, and returns `Yes` if `p` is valid and `No` otherwise. To do that, `isValid` generates all possibles interpretations of `p`'s symbols and checks that each of those interpretations satisfies `p`. It uses both the function `meaning` above and the function `vars: prop -> id list` that takes a proposition `p` and returns a list, with no repetitions, of all the ids of the propositional symbols in `p`. Implement `vars`.

   For example, `vars (And (P 1, Not(P 2)))` is `[1; 2]` (`[1; 2]` is also acceptable because the order of the ids does not matter).

3. Write a function `isUnsat: prop -> answer` that takes a proposition `p`, and returns `Yes` if `p` is unsatisfiable in propositional logic and returns `No` otherwise.

4. Write a function `entails: prop list -> prop -> answer` that takes a list `ps` of propositions and a proposition `p`, and returns `Yes` if the set of propositions in `ps` entails `p` in propositional logic, and returns `No` otherwise.

5. Write a function `areEquiv: prop -> prop -> answer` that takes two propositions `p1` and `p2`, and returns `Yes` if `p1` and `p2` are equivalent in propositional logic and returns `No` otherwise.

6. We saw in class that most logical connectives are redundant in propositional logic since propositions containing them are equivalent to propositions not containing them. Write a function `normalize: prop -> prop` that takes a proposition `p` and returns an *equivalent* proposition `q` whose connectives are from the restricted set $\{\textbf{True}, \neg, \vee\}$.[2]

   Hint: Use pattern matching *and recursion*.

2 Rule Soundness

If you were unsure about the soundness of the inference rules described in Chap. 7.5 of the textbook and in the related course noted, you could use some of the code implemented in the previous section

[2] Note that $\varphi_1 \oplus \varphi_2$ is equivalent to $\neg(\varphi_1 \Leftrightarrow \varphi_2)$.

to verify their soundness mechanically. How would you do that? Explain that in an F# comment.

3 Formula Conversions

The language of propositional logic can be defined in various ways depending on the choice of basic connectives. One alternative definition uses the logical constants **True**, **False** and the ternary (prefix) logical connective **ite**. The semantics of **ite** is specified by the following truth-table:

	p	q	r	$\mathbf{ite}(p, q, r)$
1.	*false*	*false*	*false*	*false*
2.	*false*	*false*	*true*	*true*
3.	*false*	*true*	*false*	*false*
4.	*false*	*true*	*true*	*true*
5.	*true*	*false*	*false*	*false*
6.	*true*	*false*	*true*	*false*
7.	*true*	*true*	*false*	*true*
8.	*true*	*true*	*true*	*true*

A possible F# type for propositions in such a language is the following:

```
type prop2 = TT                              // constant for true
           | FF                              // constant for false
           | V of id                         // propositional symbol
           | ITE of prop2 * prop2 * prop2
```

1. Write a method `convert: prop -> prop2` that takes a proposition `p` in the language of Section 1 (i.e., a proposition of type `prop`) and returns an *equivalent* proposition in the language of this section (i.e., a proposition of type `prop2`).

   Hint: For each constructor of type `prop`, the proposition built by that constructor is logically equivalent to a proposition built by an expression of type `prop2` that contains at most two occurrences of `Ite`.

2. Write a method `meaning2: interpr -> prop2 -> bool` with the same behavior as the method `meaning: interpr -> prop -> bool` from Section 1, but applying to `prop2` propositions.

4 Constraint Modeling in Propositional Logic

Consider again the n-rook problem seem in Homework 2 and recall the constraints that n rooks must be placed on the board, with no two rooks on the same row or column. This constraint satisfaction problem can be expressed in propositional logic by using n^2 propositional symbol $P_{i,j}$ for each $i, j = 1, \ldots, n$, expressing that board position (i, j) is occupied by a rook[3].

[3] With $P_{i,j}$ being true exactly when position (i, j) is occupied.

1. Devise a set of propositions that precisely models the 3-rook problem. The set should be such that it is satisfied by a complete interpretation of its symbols if, and only if, that interpretation describes a possible solution of the 3-rook problem.

   Write down your propositions as F# values of type `prop` from Section 1 and collect them into an F# list. Represent the symbols $P_{1,1}, P_{1,2}, \ldots, P_{2,1}, \ldots$ respectively as the propositions `(P 11)`, `(P 12)`, ..., `(P 21)`, ...

   Make sure that your answer is a valid F# value of type `prop list`. Add appropriate explanations in F# comments on what each formula in the list is supposed to model.

   Hint: `Xor` is your friend.

   To help you verify the correctness of your formalization you can use the method `findModel: prop list -> interpr option` included in `hw3.fsx` which, given a list `ps` of propositions returns one interpretation, if it exists, that satisfies every proposition in `ps`.

2. Assuming that the list in the previous question is correct, write down the simplest possible proposition that, when added to the list, effectively makes it model the 2-rook problem—in the sense that any possible solution of the 2-rook problem can be extracted from a satisfying interpretation for the new list.

3. [**Optional, extra credit**] Examine the code of `findModel`. Briefly explain in an F# comment what it does and what kind of search strategy it follows.