Homework 3
3-Satisfiability in Scheme

CS 355

Due Midnight, Monday 2/24/2014

1 Boolean Satisfiability

The *3-SAT* problem is a classic *decision problem* in computer science. You are given a set of boolean variables and a boolean expression written in *conjunctive normal form* (CNF) with 3 variables per clause. Is there some assignment of *true* and *false* to each variable so that the entire expression is true?

For example, say our variables are $\{A, B, C, D\}$ and our expression is

$$(D \lor A \lor \neg C) \land (\neg C \lor D \lor \neg A) \land (\neg C \lor D \lor \neg A) \land (B \lor \neg C \lor \neg D). \tag{1}$$

The following assignment of boolean variables

$$\{A = \text{true},\ B = \text{true},\ C = \text{false},\ D = \text{true}\} \tag{2}$$

"satisfies" the given expression. Note that above expression consists of a *conjunction* of 4 clauses. Each clause is a *disjunction* of thee terms where each term is a variable or a negation of a variable.

3-SAT is known to be *NP-complete* which means we can efficiently check to see if a solution is correct, but there is no tractable way to find a solution in general. Therefore, we typically use some sort of heuristic search method to find a solution (if there is one). Interestingly enough, 2-SAT (2 variables per clause) is solvable in polynomial time. You can read more about boolean satisfiability here:

`http://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

2 Scheme functions for solving 3-SAT

You are to write several top-level functions in Scheme as described below to help solve for instances of 3-SAT.

1. We consider a particular assignment of truth values to our given variables a *state* in a search graph. We will represent a state in Scheme as a list of pairs. Each pair is a list containing the variable name (a symbol in Scheme) and its corresponding truth value `#t` (true) or `#f` (false). Write a function `eval-var` that returns the value associated with a particular variable:

```
(define eval-var (lambda (var state) (...)))
```

For example

```
> (define state '((A #t) (B #f) (C #t) (D #f)))

> (eval-var 'A state)
#t
> (eval-var 'D state)
#f
```

2. We will represent a single *clause* in Scheme as a list of 3 elements. Each element is either a single variable name (i.e., an *atom*) or a list containing the symbol `not` followed by a variable name. Write the function `eval-clause` that evaluates a clause (i.e., returns `#t` or `#f`) for a given variables state:

   ```
   (define eval-clause (lambda (clause state) (...)))
   ```

   For example

   ```
   > (define state '((A #t) (B #f) (C #t) (D #f)))

   > (define clause '(A (not B) C))

   > (eval-clause clause state)
   #t
   ```

3. Write a function `get-vars` that returns a list of all the variables in the clause.

   ```
   (define get-vars (lambda (clause) (...)))
   ```

   You do not need to worry about duplicates in this case. For example

   ```
   > (get-vars '(A (NOT B) C))
   '(A B C)
   ```

4. Now write a function that returns all the variables contained in a list of variables.

   ```
   (define (lambda get-all-vars (clauses) (...)))
   ```

You should not have duplicate entries in the list, for example:

```
> (define clauses '((A (not B) C) (A (not B) (not C)) (A (not B) D)))
> (get-all-vars clauses)
'(C D B A)
```

The built-in **remove-duplicates** and **flatten** functions in Dr.Racket are handy here:

> (**remove-duplicates** '(A B B A))

>(**flatten** '(A B (C A) E)

5. Write a function `unsat-clauses` that returns all the unsatisfied clauses in an expressions for a given state:

```
(define unsat-clauses (lambda (clauses state) (...)))
```

For example

```
> (define state '((A #f) (B #t) (C #t) (D #f)))
> (define clauses '((A (not B) C) (A (not B) (not C)) (A (not B) D)))
> (unsat-clauses clauses state)
'((A (not B) (not C)) (A (not B) D))
```

6. Write a function `flip-var` that "flips" the "truthfulness" a particular variable in a state list.

```
(define flip-var (lambda (var state) (...)))
```

For example

```
> state
'((A #f) (B #t) (C #t) (D #f))
> (flip-var 'B state)
'((A #f) (B #f) (C #t) (D #f))
> (flip-var 'C state)
((A #f) (B #t) (C #f) (D #f))
```

7. We consider S' to be a *neighbor* to state S if S' can be created by flipping the state of exactly one variable in S. We consider S' to be a *better* neighbor if it generates less unsatisfied clauses than S in a given expression. Write the function `get-better-neighbor` that finds *some* neighbor to a given state that yields fewer unsatisfied clauses; This function actually returns a list which is the state of the better neighbor.

```
(define get-better-neighbor (lambda (clauses state vars num-unsat)(...)
      ))
```

Argument description:

`clauses` : boolean expression (list of clauses),

`state` : current state,

`vars` : list of variables used to generate neighbors *(hint: this list controls the recursion)*,

`num-unsat` : number of unsatisfied clauses generated by `state`.

If there are no better neighbors (i.e., we have reached the top of a "hill" or a "plateau") return #f.

```
> clauses
'((A  B C) (A (not B) (not C)) (A (not B) D))
> state
'((A #f) (B #t) (C #t) (D #f))
> (get-better-neighbor clauses state '(A B C)
   (length (unsat-clauses clauses state)) )
'(((A #t) (B #t) (C #t) (D #f)))
```

The above is actually a poor example since the returned neighbor is actually a solution and thus there are no corresponding unsatisfied clauses.

8. Now we put all the pieces together to perform *simple hill climbing* in search for a solution. Write the function `simple-hill-climb` that begins at a given start state and continually "climbs" by looking for better neighbors until it finds the solution or has visited a prescribed number of states.

```
(define simple-hill-climb (lambda (clauses state dist unsat)(...)))
```

Argument description:

`clauses` : list of clauses we are trying to satisfy,

`state` : starting state,

`dist` : number of states left to examine before giving up,

`unsat` : list of unsatisfied clauses generated by `state`. If this is empty, then `state` is a solution.

3 What to submit

Each function you write should have no *side effects* (i.e., each function returns freshly calculated values without altering any of the arguments or any global variables). There is no need to perform any iteration – recursion is all that is needed. Test each function individually as you write them.

- Create a Readme.txt file that includes information that give your name, email address, and a brief overview of the problem.
- Comments MUST be provided for each function. The comments must explain the algorithm used in each function. You will be given a test harness and some example 3-SAT problems of varying difficulty.
- Make sure your code works on a lab machine with Dr. Racket
- Archive the Scheme source code (and any supporting files) to your solution, and submit it electronically by midnight on the due date. Your source code should contain the eight top-level functions described in this document. Double check your submission after you submit your homework.

Note: A solution can be done with only the following special forms and built-in functions: *define, lambda, empty?, equal? cdr, car, cons, list?, if , not, and, remove-duplicates, flatten, list, -, and* >.