Constructive Logic (15-317), Fall 2016
Assignment 8, Part 1:

Contact: Oliver Daids (ojd@andrew.cmu.edu)

Due Tuesday, November 8, 2016, 1:30pm

This assignment is due at the beginning of class on the above date and must be submitted electronically via autolab. Submit your homework as a **tar** archive containing the file **hw8.pdf**.

After submitting via autolab, please check the submission's contents to ensure it contains what you expect. No points can be given to a submission that isn't there.

1 Cut!

Recall the cut primitive, `!`.

This primitive is always true, but has the side-effect of committing all decisions that led to evaluating it, preventing the backtracking mechanism from going back to before the cut.

Consider the following definition of `minimum/3`.

```
minimum(X, Y, Z) ;- X =< Y, Z = X.
minimum(X, Y, Z) :- X > Y.
```

If the `X =< Y` check succeeds, we might still evaluate the second clause if `Z = X` fails. This is unnnecessary work because `X > Y` is the exact opposite of `X =< Y`. We can solve this inefficiency by throwing in a cut to get the following:

```
minimum(X, Y, Z) ;- X =< Y, !, Z = X.
minimum(X, Y, Z) :- X > Y.
```

Now, when we try `X =< Y` and succeed, Prolog will no longer attempt to try the other definition even if `Z = X` fails.

However, cut is defined in terms of how it affects the evaluation of the program, and not in terms of the logical foundation Prolog originates from. Unsurprisingly, cuts can change the logical meaning of a Prolog program in unexpected ways. For example, we are able to define negation as the following:

```
\+(X) :- X, !, fail.
\+(X).
```

After seeing the second definition, one might reasonably assume that `\+(X)` should be true for all `X`. One might think that regardless of whether the first definition succeeds or not, the second definition should always accept. However, by inserting the cut, the first definition prevents the backtracking mechanism from trying the second definition if the evaluation mechanism fails (which it will, because we have the `fail` right after it). Unlike the previous **green** cut, where the cut had no effect on the logical meaning of the program, the use of cut here is fundamental to the behavior of `\+/1` and is therefore called a **red** cut.

Task 1 (2 points)**.** Prove or disprove that the following cut is a green cut.

```
foo(A) :- fail, !, fail.
foo(A).
```

Task 2 (2 points)**.** Prove or disprove that the following cut is a green cut.

```
member(X, [X|Xs]).
member(X, [_|Xs]) :- member(X, Xs), !.
```

Task 3 (2 points)**.** Prove or disprove that the following cut is a green cut.

```
fancy(1).
fancy(2).
fancy_member(X, [X|Xs]) :- fancy(X), !.
fancy_member(X, [_|Xs]) :- fancy_member(X, Xs).
```

2 Smooth Operator Operating Correctly

Task 4 (4 points)**.** Using the rules for explicit backtracking from class, derive a proof of $d / \top / \bot$

```
a.
b :- a
c :- a, fail; a.
d :- c, b, a.
```

3 Bringing them to terms with each other

Using the rules provided in class, find the most general unifier of the following terms or explain why it is not possible to find one.

Task 5 (2 points). $f(b, c, a)$ and $f(c, a, b)$

Task 6 (2 points). $f(x, x, g(x))$ and $f(y, h(z, a), w)$

Task 7 (2 points). $f(x, x, g(x))$ and $f(y, h(z, a), z)$

Task 8 (2 points). $f(y, f(x, y))$ and $f(f, y(y, x))$

Task 9 (2 points). $f(x, f(h(y), y, z), h(y))$ and $f(x, f(z, y, z), h(x))$