Harvey Mudd College
Computer Science 80
Logic for Computer Science
Spring Semester 2001

Propositional Logic: Implementing Resolution
Phase 1: Conversion to CNF
Due 11:00am, Friday March 30, 2001

The purpose of this project is to implement the first phase of a resolution refutation theorem prover. In particular, you will implement the conversion of a fomula to Conjunctive Normal Form. The project is to be implemented in either `rex` or `SML`. I have provided datatype/representation specifications below for both languages.

For this phase you are to implement two functions: `cnf`, and `cnf_list`. The function `cnf` takes a representation of a formula and converts it to conjunctive normal form, represented as a list of lists of literals.

The function `cnf_list` does the same, but for a list of formulas, representing a conjunction of those formulas. (Its output is, therefore, just the concatenation of the lists resulting from converting each of the formulas in its argument list individually.) The definition of `cnf_list` should be written in terms of `cnf`, and should be only a couple of lines long. The `cnf` function does almost all the work.

You will find it easier to implement `cnf` in terms of a group of support functions each of which implements one phase of the conversion.

Submission

You will submit your solution using `cs80submit` on turing.

The file you submit should contain well-organized and well-commented code for the `cnf` and `cnf_list` functions and its support functions. Do not include any other code. While you may want to put some testing code at the bottom of your file as you work on it, you must remove that part before making the final submission.

Implementation

In the notes, we gave an algorithm for converting a formula to clausal form:

1. Use logical equivalences to eliminate all connectives except $\neg$, $\wedge$, and $\vee$.
2. Use the DeMorgan laws on the result to move all the $\neg$ connectives deep into the formula, right next to atoms.
3. Use the distributive laws to create a conjunction of disjunctions of literals.
4. Convert the resulting CNF formula into clausal form (that is, to a list of lists of literals).

You may find it easier to combine steps 3 and 4 and apply the distributive laws to *clauses* rather than formulas. Or, you may, if you desire, take an entirely different approach. The only requirements are that the domain of your function be the well-formed-formula datatype appropriate to the programming language you choose, as specified below, and that your function produce the equivalent clausal form, specified as a list of lists of literals.

Rex Specification

If you are using `rex` then formulas are to be represented using strings and lists as follows:

- $\bot$ — `"false"`
- $\top$ — `"true"`
- p (a propositional letter) — `"p"`
- $\neg A$ — `["NOT",`A`]`
- $A \wedge B$ — `[`A`,"AND",`B`]`
- $A \vee B$ — `[`A`,"OR",`B`]`
- $A \Rightarrow B$ — `[`A`,"IMPLIES",`B`]`
- $A \equiv B$ — `[`A`,"EQUIV",`B`]`

You will find it useful during testing to define the following variables.

```
a = "a";
b = "b";
c = "c";
d = "d";
not = "NOT";
and = "AND";
or = "OR";
imp = "IMPLIES";
equ = "EQUIV";
```

and to similarly define variables for any other propositional letters you use in you tests. This way you can write `[a,or,[not,b]]` instead of `["a","OR",["NOT","b"]]`.

The file `/cs/cs80/rex/cnf_aux.rex` defines these and other shortcuts. It also defines several useful test cases. The file `/cs/cs80/rex/cnf_test.rex` tests your functions on those test cases. The file `/cs/cs80/rex/cnf_test.output` shows the result of running a sample solution on those test cases.

SML Specification

If you are using `SML` for your solution, you should use the following `datatype` to store formulas:

```
datatype wff = Bot
             | Top
             | Atom    of string
             | Not     of wff
             | And     of wff * wff
             | Or      of wff * wff
             | Implies of wff * wff
             | Equiv   of wff * wff;
```

The two required functions should have the type:

```
val cnf = fn : wff -> wff list list
val cnf_list = fn : wff list -> wff list list
```

The file `/cs/cs80/sml/wff.sml` contains the type definition above. You should include it (using `use`) at the beginning of your file.

During testing it will save you some typing if you make the following definitions:

```
val a = Atom "a";
val b = Atom "b";
val c = Atom "c";
val d = Atom "d";
```

and to similarly define variables for any other propositional letters you use in you tests. This way you can write `Or (a,Not b)` instead of `Or (Atom "a", Not (Atom "b"))`.

The file `/cs/cs80/sml/cnf_aux.sml` defines these and other shortcuts. It also defines several useful test cases. The file `/cs/cs80/sml/cnf_test.sml` tests your functions on those test cases. The file `/cs/cs80/sml/cnf_test.output` shows the result of running a sample solution on those test cases.