Harvey Mudd College
Computer Science 80
Logic for Computer Science
Fall Semester 2002

Propositional Logic: Implementing Resolution
Phase 1: Conversion to CNF
Due 5:00pm, Wednesday, November 27, 2002

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 `SML`.

For this phase you are to implement two functions: `cnf`, and `cnf_list`. The function `cnf` takes a datatype representation of a well-formed 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 any 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 or comment-out that part before making your submission. Failure to do so will result in a grade reduction due to the additional work required on the part of the graders.

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 `wff` datatype, as specified below, and that your function produce the equivalent clausal form, specified as a list of lists of literals.

SML Specification

The following `datatype` is used to represent 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
```

You should **not** make this type definition yourself. Rather, to launch a version of sml that has this type pre-defined, use the executable at `/cs/cs80/sml/sml-cs80`.

Support Code Provided

To save you typing during testing, this executable also contains the following definitions of the form:

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

This way you can write `Or (a,Not b)` instead of `Or (Atom "a", Not (Atom "b"))`. Additional pre-defined atoms are `p`, `q`, `r`, `s`, `hj`, `bij`, and `bd`.

In addition to these definitions, the executable provides the function `parse` of type `string -> wff`. This function can be used in your testing to simplify entering test cases. The parser uses the strings `=>` and `<=` for $\Rightarrow$, and `=` for $\equiv$. The conjunction, $\wedge$, can be written `^` or `/\`, and disjunction, $\vee$, as `V` or `\/`. Negation, $\neg$, is written `~`, and $\top$ and $\bot$ are written `T` and `F`, respectively.

Because upper-case "V" is used as an operator, this character cannot appear in propositional variable names. Other than that restriction, variable names can be arbitrary alphanumerical strings with dashes and underscores allowed as well. To increase readability, any of the bracketing characters, `{}`, `()`, or `[]` can be used, in matched pairs. The parser also assumes a default precedence in the absence of bracketing. Here is a sample of the parser in use:

```
- parse "(hj => [bij => {(hj => (bij => bd)) => bd}])";
val it =
  Implies (Atom "hj",
           Implies (Atom "bij",
                    Implies (Implies (Atom "hj",
                                      Implies (Atom "bij",Atom "bd")),
                             Atom "bd"))): wff
```

The file

`/cs/cs80/sml/cnf_test.sml`

provides several useful test cases. The file

`/cs/cs80/sml/cnf_test.output`

shows the result of running a sample solution on those test cases.