

Assignment 2: Theorem Prover

CS 730/730W/830, Spring 2012

Electronic submission of milestone due at **11:30pm on Sun, Feb 26**

Full electronic submission due at **11:30pm on Sun, Mar 4**

Hardcopy submission due at **1:10pm in class on Mon, Mar 5**

Overview

You will write a resolution refutation theorem prover for first-order logic (without equality). Given a first-order theory and query, your program will either report a series of proof steps, report that the query is false, or proceed to derive new conclusions forever. Pages 255 and 347 of the textbook might be helpful. You might also want to check out the handout on the course web page for a description of the unification algorithm. You should implement the ‘set of support’ strategy (p. 355 in the textbook). As part of the milestone submission, you will also encode a small knowledge base.

Input

We provide a converter from first-order logic (without equality) to CNF, so you may assume that all input will be in Skolemized CNF. Standard input will contain the theory (one clause per line), followed by the line `--- negated query ---`, followed by the query (on one or more additional lines). The query will already be negated. You do not need to handle equality specially. The goal of your prover is to find a contradiction between the theory and the negated query by deriving the empty clause. This proves the query.

Example input:

```
-Human(x1) | Mortal(x1)
Human(Socrates)
Animal(F2(x2)) | Loves(F1(x2), x2)
-Loves(x2, F2(x2)) | Loves(F1(x2), x2)
--- negated query ---
-Mortal(Socrates)
```

You may assume that all variables start with lowercase letters and that all predicates, functions, and constants are capitalized. A little more precisely, the CNF syntax you should accept is:

clause	→	literal clause
		literal
literal	→	predicate
		- predicate
predicate	→	capitalized-name (term-list)
term-list	→	term , term-list
		term
term	→	capitalized-constant-name
		capitalized-function-name (term-list)
		lowercase-variable-name

Because this is CNF, all variables are universally quantified and variables in different clauses might coincidentally have the same names.

Output

Your program should write only the following to standard output: a numbered list of the input clauses, followed by the proof’s resolution steps (one per line), followed by the total number of resolutions performed:

```
1: -Human(x1) | Mortal(x1)
2: Human(Socrates)
3: Animal(F2(x2)) | Loves(F1(x2), x2)
4: -Loves(x2, F2(x2)) | Loves(F1(x2), x2)
```

```

5: -Mortal(Socrates)
1 and 2 give 6: Mortal(Socrates)
5 and 6 give 7: <empty>
1646 total resolutions

```

Note the special printing of the empty clause. If you detect that no proof exists, print `No proof exists.` instead of the resolution steps. The other parts of the output should be the same.

Execution

As in assignment one, please submit `make.sh` and `run.sh` scripts. On the course web page, we supply some utility programs and input files:

`wffs-to-cnf` A CNF converter. Accepts first-order logic sentences on standard input, one per line, followed by the line `--- query ---`, followed by a query (on its own line). Writes CNF to standard output, along with the negated query. The accepted syntax is:

```

sentence  →  predicate
            |  ( sentence connective sentence )
            |  quantifier lowercase-variable-name sentence
            |  - sentence
predicate  →  capitalized-name ( term-list )
term-list  →  term , term-list
            |  term
term       →  capitalized-constant-name
            |  capitalized-function-name ( term-list )
            |  lowercase-variable-name
connective →  <-> | -> | & | |
quantifier →  forall | exists

```

Note that this means every expression with a connective needs to be surrounded by its own parentheses! The simplifier also takes a `-noquery` flag, in which case it does not take a query separator line and will not negate the last formula. Lines starting with `#` are ignored as comments.

`X.wffs` and `Y.cnf` sample input

`proof-validator` takes your program's name (which should be `run.sh`) as its first argument and your program's arguments as its remaining arguments. Passes standard input to your program. Runs your program, parses its output, and verifies that its proof (if any) is valid.

`fol-prove-reference` a sample solution

Intermediate Milestone

Write a KB of at least 10 first-order formulae and at least 4 queries (at least one of which is not provable). Submit a trace of the reference solution answering the queries.

Implement a CNF parser and printer. Submit a transcript of your code reading the output of the CNF simplifier on your sample KB and printing it out.

Implement a basic resolution algorithm. Write very simple prover that does not attempt unification and that only tries a single resolution step. In other words, you can assume that the input is two clauses (one per line), then try to resolve them (without attempting unification). Use the same output format as the full solver: if you produce the empty clause, print the resolution steps, otherwise print that no proof was found. Your milestone should work with the validator. Submit a transcript of your program working on an example, in addition to your source code. No need to turn in hardcopy.

Write-up

In your final submission, include a brief write-up answering the following questions:

1. Describe any implementation choices you made that you felt were important. Mention anything else that we should know when evaluating your program.
2. What can you say about the the time and space complexity of your program?
3. What suggestions do you have for improving this assignment in the future?

Your hardcopy submission should consist of the writeup and a transcript of your code solving the example cases. No need to include source code—we will print it from your electronic submission.

Graduate Extensions

Those in 830 must implement two extensions: preferring smaller clauses during resolution (a generalization of unit preference) and answer literals (for getting constructive proofs). If the user wants a constructive proof, they will include `Ans(x)` (for some variable `x`) in the negated query that you receive as input.

Evaluation

If you are in 730W, 2 points of your grade will depend on how clean, elegant, and readable your code is. Otherwise, we will look at your code mainly to try to give you partial credit. For the milestone:

- 0** no KB or code does not build or does not work with validator
- 1** something significant not right
- 2** looks good

For the final submission:

Unification 5

Resolution 5

Parsing 3

Variable renaming 1

Search 1

Writeup 2

Output format 1

For graduate students:

Answer literal 3

Light clauses 1

The emphasis will be on correctness. We won't be measuring efficiency to the same degree as with assignment 1, so there won't be an extra credit contest for this assignment.

Design Suggestions

Feel free to use `lex` and `yacc` for your parser if you know how to use them. I implemented my CNF parser by writing something like following functions:

parse s Takes a string, returns a sentence

parse-sentence s i Takes a string and an index. Tries to parse a sentence starting from the index in the string. Returns the sentence and the index after the last character consumed.

parse-literal s i Takes a string and an index. Tries to parse a literal starting at the index in the string. Returns the literal and the index after the last character consumed.

parse-terms s i Takes a string and an index. Tries to parse a list of terms starting at the index in the string. Returns the terms and the index after the last character consumed.

parse-term s i Takes a string and an index. Tries to parse a single term starting at the index in the string. Returns the term and the index after the last character consumed.

next s i Takes a string and an index. Returns the index of the first non-whitespace character at or after the index.

token-end s i Takes a string and an index. Returns the index of the first non-name character at or after the index.

Note that sentences to be resolved should be standardized apart, and that two sentences might resolve on more than one pair of complementary literals at once.

Use the example KBs that we distribute to test your code. The examples in the book also make good test cases. You probably want to start by printing out all the resolutions that you do. Once the prover is working, you can implement the necessary record-keeping.