CSCI 4150 Introduction to Artificial Intelligence, Fall 1999
Assignment 6: out Tuesday October 26, due Thursday November 4

A Resolution-based Theorem Prover

In this assignment, you will implement a resolution-based theorem prover loosely based on the Otter theorem prover. There are three main parts of the theorem prover: unification, substitution, and resolution. I will provide code for unification (and other support code); you will write the substitution and resolution functions.

A few notes on this assignment:

- I will post a few hints for this assignment on the Assignment 6 web page on Thursday October 28.
- We will release at least part of our code testing program so that you can check your own code before turning it in. Of course, we will not release the actual test inputs we'll use, and the test cases we provide won't be extensive, but you can always add more of your own. This probably won't be released until next week.
- As usual, the assignment is to be turned in electronically. The directory for this assignment is `/dept/cs/ai/submit6`.
- There will be a checklist of procedures and data you are to turn in on the web page. The point breakdown for this assignment will be listed with the checklist.
- Please check the web site for updates and clarifications.

Representing first order logic in Scheme

I have written support code for this assignment for printing first order logic sentences among other things. In order to do so, I have assumed the following syntax for writing first order logic sentences in Scheme.

	Scheme representation	
FOL symbol	symbol	character
$\neg$	~	tilde
$\vee$	v	letter 'v'
$\wedge$	^	carat
$\rightarrow$	=>	equals, greater than

For this assignment, you will only be using sentences in conjunctive normal form (CNF), so you should only need the symbols `~` and `v`.

You also have:

- variables — must be a single letter (except for the letter 'v' because it is used as the "or" connective)
- predicates — can be any other valid Scheme symbol, but should generally consist of letters forming a word more than one character long
- False — the result of a contradiction

Sentences are specified in a prefix notation, just like scheme. Here are a few examples of FOL sentences translated into this Scheme syntax:

$\neg Likes(Alice, x) \vee Hates(x, Bob)$	`(v (~ (likes alice x)) (hates x bob))`
$Eats(John, z)$	`(eats john z)`
$\neg Single(x) \vee \neg Male(x) \vee Bachelor(x)$	`(v (~ (single x)) (~ (male x)) (bachelor x))`

Sentences

You will deal with three different "types" of sentences in doing this assignment. I have given them the following names:

- *(plain) sentence* — a regular FOL sentence as described above, assumed to be in CNF
- *derived sentence* — The sentences in this system will either have been given (i.e. premises), the negated goal (i.e. for doing proof by contradiction), or derived by use of the resolution rule (i.e. from two sentences using some substitution). We need to keep track of a sentence's derivation so that we can print out a proof at the end.

  The form of a derived sentence is one of the following:

  ```
  (<sentence> given)
  (<sentence> negated-goal)
  (<sentence> <sentence-symbol-1> <sentence-symbol-2> <substitution>)
  ```

  The following accessor function return parts of a derived sentence:

  - `(ds-sentence s)` — returns the sentence of derived sentence `s`
  - `(ds-derived? s)` — returns `#t` if s is not a "given" or "negated goal"
  - `(ds-ss1 s)` — returns the first sentence symbol of `s`
  - `(ds-ss2 s)` — returns the second sentence symbol of `s`
  - `(ds-sub s)` — returns the substitution of `s`

- *numbered sentence* — a derived sentence with an attached sentence symbol. All the sentences in the knowledge base and the set of support will be numbered sentences. A numbered sentence has the form:

  ```
  (<sentence-symbol> <derived-sentence>)
  ```

  The following accessor functions return parts of a numbered sentence:

  - `(ns-ss s)` — returns the sentence symbol of the numbered sentence `s`
  - `(ns-ds s)` — returns the derived sentence of the numbered sentence `s`

Sentence symbols

Sentence symbols are generated by the following functions:

- `(new-sentence-symbol)` — each call generates a new sentence symbol
- `(reset-sentence-counter)` — resets the internal counter used by `new-sentence-symbol`

The representation that I happen to have chosen for sentence symbols is a list of two elements: the first is the symbol `sentence` and the second is an integer. However, your program should only use the functions above to deal with sentence-symbols.

Examples

A derived sentence might look like this:

```
((v (~ (brother John y)) (siblings John y))
 (sentence 3)
 (sentence 5)
 ((x (sentence 5)) John))
```

The derived sentence is a list: the first element is a sentence, the second and third elements are sentence symbols (of the sentences used to derive this sentence), and the fourth element is the substitution used in the derivation of this sentence.

A numbered sentence might look like this:

```
((sentence 4) ((v (~ (rich x)) (~ (famous x)) (celebrity x))
                (sentence 1)
                (sentence 2)
                ()))
```

Substitutions

Substitutions are a set of variable and value pairs. In order to avoid conflicting variable names in two different sentences, we will attach sentence symbols to the variables in a substitution. The format of a substitution is:

```
(((<var-name> <sentence-symbol>) <value>) ... )
```

I.e. a substitution is a list of variable-value pairs. The first element of each is itself a list of two elements: the variable name and the sentence symbol. The second element is the value that will replace that variable.

For example, the following two numbered sentences

```
((sentence 3) ((likes John x) given))
((sentence 8) ((likes x Jane) given))
```

would unify with the substitution

```
(((x (sentence 3)) Jane) ((x (sentence 8)) John))
```

Other general support code

- `(print-sentence s)`

  This function will print any type of sentence (plain, derived, or numbered). If it is a numbered sentence, it will print the number first.

- `(simplify s)`

  This function will take any type of sentence and remove double negations and apply DeMorgan's laws to move "not" operators inwards. You should only need this function to remove double negations.

- `(variable? expression)`

  Returns true if `expression` is a variable, i.e. a single letter symbol which is not the letter 'v'.

Unification

There are two "front end" functions for unification:

```
(unify-terms x-ss x y-ss y)
```

The `unify-terms` procedure takes two sentences `x` and `y` which will generally be terms in a CNF sentence. For this procedure, you must specify the sentence symbols `x-ss` and `y-ss` separately. This is the procedure you should use in your resolution procedure.

```
(unify x y)
```

This procedure takes two (plain) sentences. The first (x) will be Sentence 1 and the second will be Sentence 2. This function is useful to experiment with the unification procedure because sentence symbols are assigned automatically.

The unification algorithm is based on the one in our text. The "occur-check" has been omitted, so this will make our theorem prover incomplete, but it should be sufficient for the proofs in this assignment.

1 Substitution

As you will recall, in using the generalized resolution inference rule, we may start with two sentences such as:

$$\neg Brother(x, y) \vee \neg Father(y, z) \vee Uncle(x, z)$$
$$\neg Uncle(a, b) \vee Knows(a, b)$$

Unify could return the substitution {x/a, z/b} when given the two "Uncle" terms. In order to form the sentence inferred by generalized resolution, we must do two things:

1. apply the substitution to both sentences
2. combine the two sentences, removing the terms than unified

For our example, the resulting sentence would be:

$$\neg Brother(a, y) \vee \neg Father(y, b) \vee Knows(a, b)$$

Substitutions always consist of "variable/value" pairs. Here the "value" just happens to be another variable; it could be any sentence.

For this problem, you are to write the function:

```
(subst ns sub omit)
```

which takes a numbered sentence `ns`, a substitution `sub`, and a term to be omitted `omit`. It should apply the applicable substitutions to the sentence, omit the omitted term, and return a plain sentence.

For example:

```
(subst '((sentence 1) (v (likes x y) (hates a b)))
        '(((x (sentence 1)) John))
        '(hates a b))
;Value: (v (likes john y))

(subst '((sentence 2) (v (eats john meat) (likes alice x)))
       '(((x (sentence 3)) fish))
       '(eats john meat))
;Value: (v (likes alice x))
```

Note that in the second example, there was no substitution made because the substitution is for the variable "x" in Sentence 3. It just happens that Sentence 2 also has an "x" variable.

Your procedure need only handle sentences in CNF.

2 Resolution

In this problem, you will implement a resolution-based theorem prover, loosely based on the Otter theorem prover as described in our text.

I am providing the top level procedure that is called to run the theorem prover:

```
(define (prove goal sos kb)
  (reset-sentence-counter)
  ; number all the sentences
  (let ((n-goal (number-sentence (simplify (list '~ goal))))
        (n-sos (map number-sentence sos))
        (n-kb  (map number-sentence kb)))
    (if (null? (resolution-prove (cons n-goal n-sos) n-kb))
        (begin
          (display "\nWas unable to prove: ")
          (print-sentence s)
          (display "\n\n")))))
```

This procedure simply numbers all the sentences given, negates the conclusion, adds it to the set of support, and calls the `resolution-prove` procedure.

For this problem, you will write the following procedures:

```
(resolution-prove sos kb)
(resolve-sentence s kb)
(resolve s k)
```

The `resolution-prove` procedure takes a set of support and a knowledge base, both lists of numbered sentences. It should do the following:

1. If the sos is empty, return `'()`.
2. Pick a sentence from the set of support. (You can simply take the first sentence.)
3. Infer new sentences with that sentence and sentences in the knowledge base using generalized resolution (using the procedure `resolve-sentence`).
4. Filter those sentences to remove any that are already in the knowledge base or the set of support (using the provided function `filter-new-sentences`).
5. Transform the derived sentences returned by `resolve-sentence` (and filtered in the previous step) into numbered sentences.
6. If any of the new unit sentences produce a contradiction, print the proof and return `#t`. You should use the provided functions `find-contradiction` and `print-proof`.
7. Otherwise, recursively call `resolution-prove` with the new sos and kb, i.e. with the selected sentence removed from the sos and added to the kb and with the new filtered sentences added to the sos.

The `resolve-sentence` procedure takes a single numbered sentence (from the sos) and a knowledge base. It should return a list of derived sentences that result from the sentence being resolved with the knowledge base (using the procedure `resolve`).

The `resolve` procedure takes two numbered sentences (one from the sos and one from the knowledge base) and returns a list of derived sentences that result from applying generalized resolution to these two sentences. This procedure should simplify any new sentence it produces.

We will test these procedures individually, so it is important that they return the proper types of sentences.

Support code

I am providing much of the support code for resolution. You should use these procedures as they may be integral to our testing process.

- `(filter-new-sentences sentences sos kb)`

  Takes a list of derived sentences, the sos, and the kb. Returns a list of sentences that are not duplicated in `sentences` and are not in the sos or the kb. Note that this function compares CNF sentences — the order of terms in the top level disjunction does not matter.

- `(find-contradiction new-numbered-sentences sos kb)`

  Takes a list of the new sentences (which must be numbered!), the sos, and the kb. It will select the unit sentences from the `new-numbered-sentences` and test for a (unit) contradiction with sentences in the sos and the kb. Returns `'()` if no contradiction is found. If a contradiction is found, it returns the contradiction which is a numbered sentence of the form:

  ```
  (<sentence-symbol> (false <sentence-symbol-1> <sentence-symbol-2>
                            <substitution>))
  ```

- `(print-proof contradiction sos kb)`

  Takes a contradiction, the sos, and the kb and prints out the proof, starting with the contradiction and working backwards to the given sentences.

3 Proofs

Do the following proofs using your implementation of the resolution-based theorem prover.

a. Solve Problem 4 from Assignment 5. Define the following variables to hold the arguments you pass the `prove` procedure: `veg-goal`, `veg-sos`, `veg-kb`.

b. TBA (to be announced) — see the web page