Universität des Saarlandes
Fachrichtung 6.2 – Informatik
J. Siekmann, S. Autexier, K. Fischer, E. Melis, C. Zinn
M. Schiller, E. Schulz, R.Vollmann

3rd Practical Assignment in Artificial Intelligence (WS 2006/2007)

Issued: November 28, 2006 Due: December 12, 2006

Submit your solutions electronically to your tutor (via email) in a loadable text file. Groups of up to four persons are allowed. Include your names, matriculation numbers, and your group id.

This exercise sheet asks you to implement the resolution algorithm for first-order-logic. The resolution algorithm requires some preliminaries like substitution and unification. We will provide the preliminaries as a downloadable lisp-file at `http://www.ags.uni-sb.de/~omega/teach/KI0607/material/resolution-template.lisp`.
We consider a language for first-order-logic:

$$
\begin{aligned}
TERM \;::=\; & (\texttt{VAR}\ x) && (1)\\
\mid\; & (\texttt{CONST}\ c) && (2)\\
\mid\; & (\texttt{APPLY}\ f\ \ TERM^{+}) && (3)
\end{aligned}
$$

$$
ATOMICSENTENCE \;::=\; (\texttt{PREDICATE}\ p\ \ TERM^{+}) \qquad (4)
$$

$$
\begin{aligned}
SENTENCE \;::=\; & ATOMICSENTENCE && (5)\\
\mid\; & (\texttt{IMPLIES}\ SENTENCE\ \ SENTENCE) && (6)\\
\mid\; & (\texttt{AND}\ SENTENCE\ \ SENTENCE) && (7)\\
\mid\; & (\texttt{OR}\ SENTENCE\ \ SENTENCE) && (8)\\
\mid\; & (\texttt{EQUIV}\ SENTENCE\ \ SENTENCE) && (9)\\
\mid\; & (\texttt{FORALL}\ (\texttt{VAR}\ x)\ SENTENCE) && (10)\\
\mid\; & (\texttt{EXISTS}\ (\texttt{VAR}\ x)\ SENTENCE) && (11)\\
\mid\; & (\texttt{NEG}\ SENTENCE) && (12)
\end{aligned}
$$

This means, a $TERM$ can be either a variable, a constant or a function applied to at least one argument (where each argument is a $TERM$). Note that `VAR`, `CONST` and `APPLY` are three (3) specific Lisp symbols where x, c and f stand for Lisp symbols naming the particular variable, constant or function. An atomic sentence consists of a predicate applied to at least one argument. A sentence consists of either one atomic sentence, or a negated sentence, or a connective applied to two sentences, or a quantifier that binds a variable and applies to a sentence. A $LITERAL$ is an atomic sentence or the negation of an atomic sentence.

Our unification algorithm will take two literals $LITERAL_1$ and $LITERAL_2$ and returns a substitution that makes $LITERAL_1$ and $LITERAL_2$ equal. If there is no such substitution, this algorithm will return the Lisp symbol `FAIL`.
We have implemented the unification algorithm as a function named `KI-UNIFY` which takes two arguments $LITERAL_1$ and $LITERAL_2$ and returns either the Lisp symbol `FAIL` or a substitution $SUBST$ (as described above).
Examples:

```
> (KI-UNIFY '(PREDICATE P (APPLY KNOWS (CONST JOHN) (VAR X)))
            '(PREDICATE P (APPLY KNOWS (VAR Y) (APPLY MOTHER (VAR Y)))))
(((VAR X) (APPLY MOTHER (CONST JOHN))) ((VAR Y) (CONST JOHN)))


> (KI-UNIFY '(PREDICATE P (APPLY WITH (VAR X) (CONST WEST-GERMANY)))
            '(PREDICATE P (APPLY WITH (CONST EAST-GERMANY) (VAR X))))
FAIL
```

A substitution ($SUBST$) is a list of pairs associating variables with terms:

$$SUBST \quad ::= \quad (((\texttt{VAR}\ x)\ TERM)^*) \tag{13}$$

Note that NIL (the empty list) is a substitution.
We have implemented a lisp function named `KI-SUBST` which expects two arguments. The first argument is a substitution $SUBST$ and the second argument is a literal $LITERAL$. The function `KI-SUBST` returns a literal which is the result of replacing the variables in the given $LITERAL$ with the values given by $SUBST$.
Example:

```
> (KI-SUBST '(((VAR X) (CONST JOHN)) ((VAR Y) (APPLY MOTHER (VAR Z))))
            '(PREDICATE P (APPLY KNOWS (VAR X) (VAR Y))))
(PREDICATE P (APPLY KNOWS (CONST JOHN) (APPLY MOTHER (VAR Z))))
```

In the following exercises, you have to implement the resolution algorithm for first-order logic (see the Russell/Norvig book on p. 295 *ff.*, and the lecture slides). You can assume that your algorithm receives sentences in conjunctive normal form (CNF) (i.e. a conjunction of clauses, where each clause is a disjunction of literals). **Use lists of lists of literals to represent the conjunction of the disjunction of literals (e.g.** $(A \vee B) \wedge (A \vee \neg C)$ **will be represented as** $((A\ B)\ (A\ (\texttt{NEG}\ C)))$**).**

Exercise 3.1 **(50 P)**
In this exercise, you will implement some intermediate functions which will be used by your resolution-implementation in the following exercise.

1. Write a lisp function `resolve`, which takes two clauses as arguments and returns a list of all the clauses which can be obtained from the application of the resolution inference rule to these clauses (e.g. the application of the inference rule for the two clauses `(A B)` `((NEG A) (NEG B))` results in the two clauses `(B (NEG B))` and `(A (NEG A))`. (15 P)

2. Write a lisp function `factorize`, which takes a clause as argument and returns a factorized clause where multiple occurrences of unifiable literals are sorted out (i.e. if A and B unify with the mgu Θ then we apply Θ on the entire clause and replace the occurrences of $\Theta(A)$ and $\Theta(b)$ by one single literal $\Theta(A)$). (15 P)

3. For technical reasons we need that in the resolve function the two input-clauses don't share common variables.
   (a) Write a function that receives a clause as an input and returns a list with all variables that occur in that clause. (10 P)
   (b) Write a function which takes two clauses and returns a clause which is the same as the second clause but where the common variable names of the two clauses are replaced by fresh variables (i.e. variables which don't occur in the knowledge base). (10 P)

Exercise 3.2 **(30 P)**

Write a lisp function `resolution`, which takes a list of clauses, and returns `PROOF` when the empty clause is derived. You can adapt the resolution algorithm in Russell/Norvig on p. 216 to first-order logic by using the two functions from the above exercises. In the *resolution-template.lisp* file you will find a function `clause-equal` which determines when two clauses can be considered equal, and which will be helpful together with the Lisp function `union`.

Exercise 3.3 **(20 P)**

Assume we want to describe the world in the `AIchhörnchen` environment. We represent squares by constants, and define a binary predicate *adjacent*, such that *adjacent(i,j)* if and only if i and j are adjacent. Furthermore, we write *squirrel(i)* if the squirrel is at square i, *reach(j)* if the squirrel can reach the field j, *tree(k)* if there is a tree on field k, and *disallowed(l)* if the squirrel may not go to field l.

1. Square b is adjacent to square a.
2. There is a tree on square b.
3. The squirrel is on square a.
4. Whenever a square i is adjacent to a square j, then j is also adjacent to i.
5. Whenever we have two adjacent squares, and the squirrel is at the first square, then the squirrel can reach the adjacent square.
6. Whenever a square is reachable for the squirrel and there is a tree on that square, the squirrel is disallowed to go there.

- Formalize the sentences in first-order logic. (5 P)
- Formalize the sentences as Lisp S-expressions in the syntax introduced in the beginning of this exercise sheet. (5 P)
- Either generate the CNF by hand or use some code which will be provided by us that normalizes your formulae. Use your implementation of resolution to show that *disallowed*(b)! (10 P)

Exercise 3.4 **(Extra 20 P)**

Write a lisp function `resolutionSOS`, same as in the previous exercise, but implement this time the *set of support* strategy (described in the Russell/Norvig book on p. 305). For this strategy, a subset of the sentences is called *set of support*. Each application of resolution combines one sentence from the set of supports with another sentence and adds the resulting resolvent into the set of supports.

Note: It is possible to find complete implementations of resolution theorem provers on the internet (for example `http://www.cs.technion.ac.il/~shaulm/software/classic-rtp.html`). However, only solutions that follow the instructions on this exercise sheet will be given points.