**Universität des Saarlandes**
**Fachrichtung 6.2 – Informatik**
J. Siekmann, S. Autexier, C. Benzmüller, C.E. Brown
C. Hahn


# 2nd Practical Assignment in Artificial Intelligence (SS 2005)
# Solutions

- **Please submit your solution by sending an email to your tutor. The e-mail address of your tutor can be found on the webpage.**
- **Indicate the name and matriculation number of each student working in your project.**
- **Make sure to include comments in your code. In order to obtain full credit, your code must be documented and comprehensible.**

**Exercise 2.1** **(100 P)**

In this exercise, you have to implement an unification algorithm which has been specified in the lecture using Lisp. In general, this unification algorithm should take two terms $TERM_1$ and $TERM_2$ and return a substitution that would make $TERM_1$ and $TERM_2$ look the same. If there is no such substitution, this algorithm should return the Lisp symbol `FAIL`. Formally, a $TERM$ has one of three forms:

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

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.

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

$$SUBST ::= ((\texttt{VAR}\ x)\ TERM)^* \quad (4)$$

Note that $NIL$ (the empty list) is a substitution.

1. Implement a lisp function named `KI-PRINT-TERM` which prints a term in the notation from the slides. Namely, (`VAR` $x$) should be printed as $x$, (`CONST` $c$) should be printed as $c$ and (`APPLY` $f\ TERM_1\ \cdots\ TERM_n$) should be printed as $f(t_1, \ldots, t_n)$ where $t_i$ is the printed version of $TERM_i$. (The return value of this function is not important.) (10 P)

   Hint: The Lisp function `format` could be helpful.

   Example:

```
> (KI-PRINT-TERM '(APPLY KNOWS (VAR X) (CONST JOHN)))
KNOWS(X,JOHN)
```

2. Implement a lisp function named `KI-PRINT-SUBST` which prints a substitution in the notation `{VAR/TERM,...}`. (The return value of this function is not important.) (10 P)

   Examples:

```
> (KI-PRINT-SUBST NIL)
{}
```

```
> (KI-PRINT-SUBST '(((VAR X) (CONST JOHN))))
{X/JOHN}
```

```
> (KI-PRINT-SUBST '(((VAR X) (CONST JOHN)) ((VAR Y) (APPLY MOTHER (VAR Z)))))
{X/JOHN,Y/MOTHER(Z)}
```

3. Implement a lisp function named `KI-SUBST` which expects two arguments. The first argument will be a substitution $SUBST_1$ and the second argument will be a term $TERM_1$. The function `KI-SUBST` should return a term $TERM_2$ which is the result of replacing the variables in $TERM_1$ with the values given by $SUBST_1$. (30 P)

   Example:

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

4. Implement the unification algorithm as a function named `KI-UNIFY` which takes two arguments $TERM_1$ and $TERM_2$ and returns either the Lisp symbol `FAIL` or a substitution $SUBST$ (as described above). (50 P)

   Examples:

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

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

Hint: Rules for unification are in the slides. There is also a unification algorithm in the book (Figure 9.1, page 278). Be careful! The unification algorithm in the book assumes the inputs are terms or *lists* of terms. Also, the rules in class are written for two sets of equations (pairs of terms) $U, E$.

***Solution:***

```
; Author: Chad E Brown
; Date: May, 2005

; testing functions
(defun variable? (x)
  (and (consp x)
       (eq (car x) 'VAR)))

(defun constant? (x)
  (and (consp x)
       (eq (car x) 'CONST)))

(defun compound? (x)
  (and (consp x)
       (eq (car x) 'APPLY)))

; Term Printing Function
(defun ki-print-term (trm)
  (format t "~d~%" (ki-print-term-1 trm)))

(defun ki-print-term-1 (trm)
  (cond ((variable? trm) (format nil "~d" (cadr trm)))
((constant? trm) (format nil "~d" (cadr trm)))
((compound? trm) (format nil "~d(~d~d)" (cadr trm)
 (ki-print-term-1 (caddr trm))
 (ki-print-args (cdddr trm))))
(t (error "~d is not a term" trm))))

(defun ki-print-args (args)
  (if args
      (format nil ",~d~d"
      (ki-print-term-1 (car args))
      (ki-print-args (cdr args)))
    ""))

(defun ki-print-subst (sub)
  (if sub
      (format t "{~d~d}~%"
      (ki-print-subst-0 (car sub))
      (ki-print-subst-1 (cdr sub)))
    (format t "~%")))

(defun ki-print-subst-0 (pair)
  (format nil "~d/~d"
  (ki-print-term-1 (car pair))
  (ki-print-term-1 (cadr pair))))

(defun ki-print-subst-1 (sub)
```

```
  (if sub
      (format nil ",~d~d"
      (ki-print-subst-0 (car sub))
      (ki-print-subst-1 (cdr sub)))
    ""))

; Substitution Function
(defun KI-SUBST (sub trm)
  (cond ((variable? trm)
 (let ((a (assoc trm sub :test #'equal)))
   (if a
       (cadr a)
     trm)))
((constant? trm)
 trm)
((compound? trm)
 (cons 'APPLY
       (cons (cadr trm)
     (mapcar #'(lambda (arg)
 (ki-subst sub arg))
     (cddr trm)))))
(t (error "~d is not a term" trm))))

; Unification Function, calling from Russell/Norvig, 2nd Ed, Fig 9.1, p. 278
(defun KI-UNIFY (trm1 trm2)
  (RN-UNIFY trm1 trm2 nil))

(defun list? (x)
  (and (consp x)
       (not (variable? x))
       (not (constant? x))
       (not (compound? x))))

; Unification Function from Russell/Norvig, 2nd Ed, Fig 9.1, p. 278
; x, y can either be (VAR -), (CONST -), (APPLY ---) or it is a "list" of terms
; theta - substitution
(defun RN-UNIFY (x y theta)
  (cond ((eq theta 'FAIL)
 'FAIL)
((equal x y)
 theta)
((variable? x)
 (rn-unify-var x y theta))
((variable? y)
 (rn-unify-var y x theta))
((and (compound? x) (compound? y))
 (rn-unify (cddr x) (cddr y) theta))
((and (list? x) (list? y))
 (rn-unify (cdr x) (cdr y)
   (rn-unify (car x) (car y) theta)))
(t
```

```
 'FAIL)))

(defun rn-unify-var (var x theta)
  (let ((a (assoc var theta :test #'equal)))
    (if a
(rn-unify (cadr a) x theta)
      (let ((b (assoc x theta :test #'equal)))
(if b
    (rn-unify var (cadr b) theta)
  (let ((foo (ki-subst theta x)))
    (if (occur-check? var foo)
'FAIL
      (cons (list var foo)
    (mapcar #'(lambda (p)
(list (car p)
      (ki-subst (list (list var foo)) (cadr p))))
    theta)))))))))

(defun occur-check? (var x)
  (cond ((variable? x) (equal var x))
((constant? x) NIL)
((compound? x) (occur-check-l? var (cddr x)))))

(defun occur-check-l? (var args)
  (if args
      (or (occur-check? var (car args))
  (occur-check-l? var (cdr args)))
    NIL))
```

---