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

1st Practical Assignment in Artificial Intelligence (WS 2006/2007)
Solutions

Submit your solutions electronically to your tutor (via email). Groups of up to four persons are allowed.

Exercise 1.1 *(Three Kinds of Equality)* **(10 P)**

```
USER: (eq '(1 2 3) '(1 2 3))
> NIL
USER: (eql '(1 2 3) '(1 2 3))
> NIL
USER: (equal '(1 2 3) '(1 2 3))
> T
```

Explain the difference between `eq` and `equal`. Is there a difference between `eq` and `eql`? Look up a Lisp reference.

Solution:

`eq` implements equality on memory locations. Symbols are stored at the same memory locations, therefore (eq 'a 'a) evaluates to T (true). `eql` is generally the same as `eq`, however, on some Lisp implementations they work differently on numbers. For example, on some Lisp implementations, the following holds:

```
USER: (eq 1 1)
> NIL
USER: (eql 1 1)
> T
```

`equal` tests for structural equality between two Lisp objects.
Note: For some datatypes, the behaviour of `eq,eql` and `equal` is implementation-dependent.

Exercise 1.2 *(Two Kinds of List Operations)* **(10 P)**

Consider the following two snippets of Lisp code:

Example 1

```
USER:  (setq list1 '(1 2 3))
> (1 2 3)
USER:  (setq list2 '(4 5 6))
> (4 5 6)
USER: (append list1 list2)
> (1 2 3 4 5 6)
USER: (nconc list1 list2)
> (1 2 3 4 5 6)
```

Example 2

```
USER: (setq list1 '(1 2 3))
> (1 2 3)
USER: (setq list2 '(4 5 6))
> (4 5 6)
USER: (nconc list1 list2)
> (1 2 3 4 5 6)
USER: (append list1 list2)
> (1 2 3 4 5 6 4 5 6)
```

Why is that? Explain the difference between Example 1 and Example 2!

Solution:

`nconc` is a destructive operation on lists, which by appending a list to another list changes that list. Therefore, in Example 2, after calling nconc on list1 and list2, list1 is changed to the list (1 2 3 4 5 6). `append` does not change the lists which it receives as arguments.

Exercise 1.3 *(The Ackermann Function)* **(20 P)**

Write a Lisp function that implements the famous Ackermann function (published in 1928), known to be a non-primitive recursive function with an interesting runtime behaviour.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if m} > 0 \text{ and n} = 0 \\ A(m-1, A(m,n-1)) & \text{if m} > 0 \text{ and n} > 0 \end{cases}$$

Solution:

```
;;; The recursive Ackermann function
(defun ackermann (m n)
  (cond ((or (< m 0) (< n 0)) (error 'error ``not defined on negative arguments''))
        ((= m 0) (+ n 1))
        ((= n 0) (ackermann (- m 1) 1))
        ('T (ackermann (- m 1) (ackermann m (- n 1))))))
```

Exercise 1.4 *(Tree Recursion)* **(40 P)**

Let us represent trees in Lisp as nested lists (e.g., `'(1 (2 3))` for 

).

1. Write a Lisp function `treesum` that returns the sum of all numbers contained in a tree (i.e, a nested list). As a test, evaluate `(treesum '(4 (5 (6 7) (2 1)) 0))` and use `trace` to inspect the behaviour of the algorithm. (20 P)

   Solution:

   ```
   ; define the example tree
   (setq tree '(4 (5 (6 7) (2 1)) 0))

   ; function treesum
   (defun treesum (tree)
     (cond ((numberp tree) tree)
           ((consp tree)
            (+ (treesum (car tree)) (treesum (cdr tree))))
           (T 0)))
   ```

2. Write a recursive function that takes a tree as argument and returns it with the labels of its leaves squared. Use `dolist` to process lists. (10 P)

   Solution:

```
; recursive tree manipulation function using dolist
(defun treeman-do (l)
  ;Input: a list
  ;Value: the modified list
  (unless (listp l)
    (error "~S is not a list." l))
  (let ((ret nil))
    (dolist (x l ret)
      (if (listp x)
          (setq ret (append ret (list (treeman-do x))))
        (setq ret (append ret (list (* x x))))))))
```

3. Write a recursive function that takes a tree as argument and returns it with the labels of its leaves squared. Use `mapcar` to process lists. (10 P)

Solution:

```
; recursive tree manipulation function using mapcar
(defun treeman-map (l)
  ;Input: a list
  ;Value: the modified list
  (unless (listp l)
    (error "~S is not a list." l))
  (mapcar #'(lambda (x) (if (listp x)
                            (treeman-map x)
                          (* x x)))
          l))
```

Exercise 1.5 *(Closures)* **(20 P)**

Using `lambda`, we can define anonymous functions, called *closures*.

1. Write a function `make-adder` that takes as argument a number n and returns a closure, which takes as argument a number k and returns $n+k$. Try out your function with several examples (e.g., `(setq add7 (make-adder 7)) (funcall add7 2)`). (10 P)

Solution:

```
; function to construct adder functions
(defun make-adder (n)
  ;Input: a number n
  ;Value: a function that takes a number k as input and returns n+k
  (unless (numberp n)
    (error "~S is not a number." n))
  #'(lambda (k) (+ n k)))
```

2. Extend your function from exercise **??**.3 to take a closure as an additional argument. The closure takes one argument and is applied to the leaves of the tree. Try it out with `#'1+`, with `#'(lambda (x) (+ x x))`, and with `#'(lambda (x) (* x x))`. (10 P)

Solution:

```
; extended tree manipulation function
(defun treeman (l fun)
  ;Input: a list
  ;Value: the modified list
  (unless (listp l)
    (error "~S is not a list." l))
  (mapcar #'(lambda (x) (if (listp x)
                            (treeman x fun)
                          (funcall fun x)))
          l))
```

Exercise 1.6 *(Challenge)* **(extra 20 P)**

Write a function `memo`, which takes a function name as its only argument, and returns a "lambda" function (a closure) that behaves like the given function but uses a hash table to store previous function calls with their return values to speed up the execution. However, you do not need to retrieve the values of recursive function calls from the hash table. Try it out with the Ackermann function from exercise **??**. Consider the example:

```
(ackermann 3 7)
(ackermann 3 7) ;;; computes the return value twice

(setq memo-ackermann (memo 'ackermann))
(funcall memo-ackermann 3 7)
(funcall memo-ackermann 3 7) ;;; retrieves the return value from the hash table
                             ;;; (should be fast)
```

Hint: Make sure you use `#'equal` as the `:test` argument for the hash table. Functions (such as your lambda functions) can have an argument prefixed by the `&rest` key word, which serves to collect an arbitrary number of arguments to the function in a list:

```
USER: (defun foo (x y &rest z)
      (format 'T "These are x: ~S, y: ~S and z: ~S~%" x y z))
> FOO
USER: (foo 1 2 3 4 5 6)
> These are x: 1, y: 2 and z: (3 4 5 6)
```

The following example demonstrates the use of a Lisp hash table:

```
USER: (setq phonebook (make-hash-table :test #'equal))
> #<EQUAL hash-table with 0 entries @ #x2049f0e2>
USER: (gethash 'SCHMITZ phonebook)
> NIL
USER: (setf (gethash 'SCHMITZ phonebook) 3003003)
> 3003003
USER: (setf (gethash 'SCHULZ phonebook) 110)
> 110
USER: (gethash 'SCHMITZ phonebook)
> 3003003
```

Solution:

```
(defun memo (func)
  (let ((hashtable
              (make-hash-table :test #'equal)))
    #'(lambda (&rest arguments)
```

```
(let ((memo-result
          (gethash arguments hashtable 'MEMO-UNSET)))
  (if (equal memo-result 'MEMO-UNSET)
      (let ((computed-result
                (apply func arguments)))
        (setf (gethash arguments hashtable) computed-result)
        computed-result)
    memo-result)))))
```