TEL AVIV UNIVERSITY
Department of Computer Science
0368.1105 – Extended Introduction to Computer Science
Fall Semester, 2000/2001

Homework 4, Nov 19, 2000

This homework is due by 18:00 on Thursday, Nov 30. Write your name, id and the **Tirgul group number** on top of the first page. Keep a copy of the homework. Please answer all questions that require you to write a procedure by submitting the procedure itself, and the result of testing your code on the computer (a printout). Try to pick at least one meaningful example of an execution to show that the procedure works. There is a lot of work so we suggest that you start working right away.

Pairs and lists

1. Give combinations of cars and cdrs that will pick 5 from each of the following lists:

```
(1 3 3 4 5 6 7)
((1) (3 3 4) (5 6) 7)
(1 (3 (3 (4 (6 (5 (7)))))))
(1 ((3 3 ((4)) 4) 5) 7)
```

2. Suppose we define `x` and `y` to be the two lists:

```
(define x (cons 10 20))
(define y (list 11 22 (list 100 200)))
```

What result is printed by the interpreter in response to evaluating each of the following expressions:

```
(cons x y)
(list x y)
(append x y)
(append y x)
```

3. Draw the box-and-pointer diagrams for the results of the following expressions:

```
(list 1)
(list (cons (list 1 2) (list 4 5)))
(list (cons (list 1 2) (cons 2 3)) 4 4 (list 1 2 3))
(cons (list (cons (cons 3 4) 9) (cons 5 6) 7) 8)
```

4. Given the following alternative definition of cons, write the appropriate definitions of car and cdr.

```
(define (cons x y)
  (lambda (m) (m x y)))
```

5. Show that we can represent pairs of non negative integers using only numbers and arithmetic operations if we represent the pair `(a . b)` as the integer $2^a 3^b$. For example `(5 . 4)` is represented by the number $2^5 3^4 = 2592$. Give the corresponding definitions of the procedures cons, car and cdr.

Dotted Tail Notation - procedures with varying number of parameters

In order to define procedures with a varying number of parameter, we use the dotted tail notation. For example, given the definition

```
(define (f x y . z) <body> )
```

the procedure f may be called with 2 or more parameters. For a call `(f 1 2 3 4 5 6)` when evaluating the body of f, x will be 1, y will be 2, and z will be the list of all the other arguments, `(3 4 5 6)`. So in general, `(car z)` will be the next argument, if it exists - that is, if `(null? z)` is false.

Given the definition

```
(define (g . w) <body> )
```

the procedure g may be called with 0 or more parameters. For a call `(g 1 2 3 4 5 6)` when evaluating the body of g, w will be `(1 2 3 4 5 6)`.

The unsugared versions of these procedures will be:

```
(define f (lambda (x y . z) <body> ))
(define g (lambda w <body> ))
```

(To understand the structures, draw box-and-pointer representations).

As an example, we could define the procedure `+`, using a procedure `add` that adds 2 numbers.

```
(define (+ . x)
  (define (add-list lst)
    (if (null? lst)
        0
        (add (car lst) (add-list (cdr lst)))))
  (add-list x))
```

6. Use the dotted tail notation to write two procedures `divide` that behaves exactly like the primitive `/`, and `less-than` that behaves exactly like the primitive `<`. First, you must examine the behavior of `/` and `<` carefully on 0, 1 and more arguments using the interpreter. Your procedures may call the primitive `/` and `<` *only* on 2 arguments, just as `+` in the example above calls `add` on 2 arguments. Show enough different examples of running your procedures to convince the reader that all the different cases are handled correctly.

7. Use the dotted tail notation to write a procedure `divi?` that takes one or more integers and returns a list of all the arguments which are divisible by the first argument.

for example:

```
=> (divi? 1 2 3 4 5 6 7)
(1 2 3 4 5 6 7)
=> (divi? 2 3 4 5 6 7)
(2 4 6)
```

List structures

8. Write a version of `reverse` that runs in linear time.

9. Write a combiner procedure that takes as arguments a binary operation and two lists of numbers and returns a third list whose i-th element is the result of applying the binary operation to the respective i-th elements of the two lists. You can assume that the two input lists are of the same length (that is, there is no need to compare their length).

For example:

```
=> (combiner + (list 1 2 3 8) (list 1 1 1 1))
(2 3 4 9)
```

10. Write an accumulate procedure that takes 4 arguments: (1) a binary operation (that specifies how the current term is to be combined with the accumulation of the rest of the list), (2) an initial value that should be used when the list elements run out, (3) a term procedure that is applied to each list element just prior to its accumulation with the rest of a list, and (4) a list.

The returned value is the result of combining (via the binary operator) the results of applying term to each of the list elements.

For example:

```
=> (accumulate + 0 cube (list 1 2 2))
17
=> (accumulate * 1 cube (list 1 2 2))
64
```

11. Use the accumulate and combiner procedures (from the previous questions), and only them, to implement the following operations on lists of numbers (think of them as vectors):

(+list v w) - returns a list which is the sum of lists v and w, (v+w for vectors v and w).

(-list v w) - returns v-w.

(*list v w) - returns the dot product v and w (sum of element products).

(=list v w) - returns true iff v=w .

You can assume that v and w are lists of the same length.

12. Write a procedure `(remove-duplicates lst)` that returns a list containing the elements of `lst`, but with duplicates removed. You may assume that the input is a list of integers. For example:

```
(remove-duplicates (list 1 2 3 1 2 3)) => (1 2 3)
(remove-duplicates (list 1 1 1)) => (1)
```