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

Homework 5, Nov 26, 2000

This homework is due by 18:00 on Tuesday, Dec 5. 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.

Operations on lists

1. Use `equal?` to write a procedure `member` that is like `memq` (see page 144 in the book) except that it can find any item, not just a symbol, in a given list. For example,

```
==> (member '(apple pear) '(x (apple sauce) y apple pear))
()

==> (member '(apple sauce) '(x (apple sauce) y apple pear))
((apple sauce) y apple pear)
```

2. The procedure `map` is actually provided in SCHEME in a more general form. It takes as arguments one procedure of n arguments, followed by n lists, ($n > 0$). Assuming the lists are of equal length, say m, it returns a list of m elements, such that the i-th element is the result of applying the procedure to the i-th elements of all the lists. For example:

```
==> (map (lambda (x y z) (+ x (* y z))) '(2 3) '(4 5) '(6 7))
(26 38)
```

Write a procedure `map-all` that implements this general form of map. You will have to use the dotted tail notation (see homework 4). Use the version of map that we saw in class (which takes one procedure and one list). You may also use the standard procedure `(apply proc arg-list)`, which applies the procedure to the list of arguments. For example:

```
==> (apply + '(2 3 4))
9
```

3. Consider the procedure `(apply-resp procs items)`, which takes a list of 1 argument procedures `procs`, and a list of elements `items`. Assuming both lists are of equal length, it applies the i-th procedure on the i-th element, and forms a list of the same length as a result. For example:

```
==> (apply-resp (list sqrt square cube) '(4 3 2))
(2 9 8)
```

a. Write the procedure using the general version of map from the previous exercise. **b.** Write the procedure without using map.

4. Write a procedure `deep-reverse` which recursively reverses an entire list structure. For example,

```
==> (deep-reverse '((1 2 (3 4)) (5 6)))
((6 5) ((4 3) 2 1))
```

Note the difference between `deep-reverse` and `reverse` that we saw in class:

```
==> (reverse '((1 2 (3 4)) (5 6)))
((5 6) (1 2 (3 4)))
```

5. Implement a procedure `for-each` which takes as arguments a procedure and a list of elements. `for-each` applies the procedure to each of the elements in turn, from *left to right*. The values returned by applying the procedure to the elements are not used at all – `for-each` is used with procedures that perform an action, such as printing. For example,

```
==> (for-each (lambda (x) (newline) (display x))
              (list 57 321 88))
57
321
88
```

`for-each` should always return the value `true`. Note that `for-each` is provided in SCHEME. The value returned is undefined, so different implementations of SCHEME may not agree.

6. Redefine `count-leaves` as an accumulation:

```
(define (count-leaves t)
   (accumulate <??> <??> (map <??> <??>)))
```

7. Write a procedure `(tree-map proc tree)`, which creates a new tree with the same structure as the original, but with values that are the result of applying `proc` to the respective values in `tree`. To make sure your procedure works correctly, show that you can implement `scale-tree` as follows (compare to the version of `scale-tree` that used `map`, and to `scale-list` that used `map`).

```
(define (scale-tree tree factor)
   (tree-map (lambda (x) (* x factor)) tree))
```

8. The `accumulate` procedure is also known as `fold-right`, because it combines the first element of the sequence with the result of combining all the elements to the right. More specifically if `(op` x y`)` returns the element $x \circ y$, then `(accumulate op` e `(list` a_1 a_2 `...` a_n`) )` returns the element $a_1 \circ (a_2 \circ (\ldots (a_n \circ e) \ldots))$.

The `fold-left` procedure combines the elements from left to right. More specifically, `(fold-left op` e `(list` a_1 a_2 `...` a_n`) )` returns the element $((\ldots(((e \circ a_1) \circ a_2) \circ a_3) \circ \ldots) \circ a_n)$.

For example:

```
==> (fold-right / 3 '(1 2 3 4))
1.125
==> (fold-left / 3 '(1 2 3 4))
0.125
```

Give two implementations of `fold-left`. The first implementation should be similar to the implementation of `fold-right`, i.e., `accumulate`. Obtain the second implementation by completing the following code:

```
(define (fold-left op init lst) (fold-right <??> init <??>) )
```

9. **a.** Complete the following definitions of `reverse` in terms of `fold-left` and `fold-right`:

```
(define (reverse sequence)
  (fold-right (lambda (x y) <??>)
              nil
              sequence))

(define (reverse sequence)
  (fold-left (lambda (x y) <??>)
             nil
             sequence))
```

b. What is the orders of growth of the running time of the 3 versions of `reverse`: The original one we wrote - see below, the one using `fold-right` and the one using `fold-left`. You should count the number of `cons` operations done.

Here is the original `reverse`:

```
(define (reverse lst)
   (cond ((null? lst) lst)
         (else (append (reverse (cdr lst))
                       (list (car lst))))))
```

10. **a.** Write a procedure `(insertR new old lst)` that inserts the symbol `new` to the right of every appearance of symbol `old` in the list `lst`.

b. Write a procedure `(insertR* new old tree)` that inserts the symbol `new` to the right of every appearance of symbol `old` in the tree `tree`.

Symbolic Differentiation

11. Extend the basic symbolic differentiation package which you saw in class according to the following instructions. You will have to start from the code given in class, (including the improved version of `make-sum` and `make-product` that produce simplified expressions), and modify it. You may get the original code from the course web page. There is a link in the homework area to a scheme file. Just follow the link, use "save as" and then use the resulting file as your starting point.

a. Extend the symbolic differentiation package so that it could also differentiate expressions involving subtractions and divisions. Use the following rules:

$$(f(x) - g(x))' = f'(x) - g'(x) \ ,$$

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{g(x) \cdot f'(x) - f(x) \cdot g'(x)}{(g(x))^2} \ .$$

Do this by adding new clauses to the `deriv` procedure and defining appropriate procedures `make-sub`, `sub?`, `sub-arg1`, and `sub-arg2` for subtraction, `make-division`, `division?`, `div-arg1`, `div-arg2` for division. Use the symbols $-$ and / to denote subtraction and division. For example, you should get these (or equivalent) results:

```
==> (deriv '(/ 1 (* 2 x)) 'x)
(/ -2 (* (* 2 x) (* 2 x)))
==> (deriv '(/ x y) 'x)
(/ y (* y y))
```

b. Extend the package so that it could also differentiate expressions involving sines and cosines. Use the rules:

$$(\sin(f(x)))' = \cos(f(x)) \cdot f'(x)$$

$$(\cos(f(x)))' = -\sin(f(x)) \cdot f'(x) \ .$$

Do this by adding new clauses to the `deriv` procedure and defining appropriate procedures `make-sin`, `sin?`, `sin-arg`, `make-cos`, `cos?`, and `cos-arg` For example, you should get the following (or equivalent) result:

```
==> (deriv '(/ (sin (* x (cos x))) (* 3 x)) 'x)
(/ (- (* (* 3 x) (* (cos (* x (cos x))) (+ (* x (* -1 (sin x))) (cos x))))
      (* (sin (* x (cos x))) 3))
   (* (* 3 x) (* 3 x)))
```

c. Implement the derivative of tan by transforming $tan(\alpha)$ into $\frac{sin(\alpha)}{cos(\alpha)}$. Do this by adding a new clause to the `deriv` procedure and defining appropriate procedures `make-tan`, `tan?`, and `tan-arg`. For example, you should get this (or an equivalent) result.

```
==> (deriv (make-tan 'x) 'x)
(/ (- (* (cos x) (cos x)) (* (sin x) (* -1 (sin x)))) (* (cos x) (cos x)))
```

Sets as Ordered Lists

12. Give an implementation of `adjoin-set` using the ordered representation. Show how to take advantage of the ordering to produce a procedure that requires on the average about half as many steps as with the unordered representation.

13. Give a $\Theta(n)$ implementation of `union-set` for sets represented as ordered lists.

Sets as Binary Trees

14. Each of the following two procedures converts a binary tree to a list.

```
(define (tree->list-1 tree)
  (if (null? tree)
      nil
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1 (right-branch tree))))))
```

```
(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list (right-branch tree)
                                          result-list)))))
  (copy-to-list tree nil))
```

a. Do the two procedures produce the same result for every tree? Justify your answer.

b. Do the two procedures have the same order of growth in the number of steps required to convert a balanced tree with n elements to a list? If not, which one grows more slowly?