IS MSc Artificial Intelligence Programming II

Exercise 6
Issued: week 6

February 5, 2002

Everyone should attempt problems 1–5. The remaining problems are harder and are optional.

1. (a) Write a recursive procedure **count_down** which given an integer as argument counts down to 0 from that integer e.g.

```
count_down(5);
** 5
** 4
** 3
** 2
** 1
** 0
```

   (b) By adding one statement to count_down make a new procedure count_down_up which first counts down from the given number to 0 and then counts back up again to the number.

2. (a) Write both iterative and recursive procedures (named **bottle1** and **bottle2**) that take one argument, an integer, and print out the corresponding number of verses of the song about "green bottles" e.g.

```
bottle(5);
** [5 green bottles hanging on a wall]
** [5 green bottles hanging on a wall]
** [if 1 green bottle should accidently fall]
** [there would be 4 green bottles hanging on a wall]


** [4 green bottles hanging on a wall]
etc etc.
```

   (b) Write a recursive procedure called called **bottle_count** that takes a single integer as argument, corresponding to a number of bottles in question 2 and adds up the total number of bottles mentioned e.g in the example above there would be (5 + 5 + 1 + 4) + (4 + 4 + 1 + 3) + (3 + 3 + 1 + 2) + (2 + 2 + 1 + 1) + (1 + 1 + 1 + 0)

```
bottle_count(5)=>
** 45
```

3. Write a recursive procedure named **intersect** that takes two lists as arguments and returns another list which contains only those elements which occur in both the input lists. If there are no such elements the procedure should return [ ] . e.g.

```
intersect([tom dick harry],[mary susan jane])=>
** [ ]
intersect([a b c d e f],[e f g a])=>
** [a e f]
```

4. Use your **intersect** procedure to write a procedure called **set_difference** which given two lists as arguments returns a list of all those elements which are in one or other of the lists but not in both.
   Hint: You may find the built-in procedure **delete** useful.

5. Write a recursive procedure named **check** that takes an arbitrarily deeply nested list and a predicate as its two arguments. The procedure should return true if there is an item anywhere in any of the lists or sub-lists which satisfies the predicate and otherwise return <false>. e.g.

```
check([a [b c [[e f]]g] h i],isinteger)=>
** <false>
check([a [b c [[e f]]5] h i],isinteger)=>
** <true>
```

   What should happen if the predicate is **islist**?

6. Adjust your answer to the previous question to produce a recursive procedure named **censor**. It should take the same kind of arguments as check but rather than return simply <true> or <false>, it should return the nested list that it was given with any items satisfying the predicate replaced by the word "censored". e.g.

```
censor([a [b c [[e f]]g] h i],isinteger)=>
** [a [b c [[e f]] g] h i]
censor([a [b c [[99 f]]5] h i],isinteger)=>
** [a [b c [[censored f]] censored] h i]
```

   What should happen if the predicate is **islist**?

7. Write a procedure **substring_position(string1,string2)** which returns an integer **n** as its result if the string **string2** contains a substring starting at position **n** equal to **string1**, and <false> if no such position exists e.g.

```
substring_position('abc','xabdabcbd')=>
** 5
substring_position('abc','xabdacbab')=>
** <false>
```

   Hint: Use two nested loops.

8. Modify your **substring_position** procedure so that it returns *a list of all the positions* in **string2** where **string1** occurs as a substring.

PTO

9. Write a *recursive* procedure **list_palindrome**which takes a list as argument and returns<true> if the string is a palindrome (i.e is the same backwards as forwards), and <false> otherwise e.g.

   ```
   list_palindrome([hello there there hello])=>
   ** <true>
   string_palindrome([I was gone gone I])=>
   ** <false>
   ```

   Hint: You may find the built-in procedures **last** and **allbutlast** useful. Alternatively you can use the matcher.

10. Write an *iterative* procedure **string_palindrome** which takes a string as argument and returns <true> if the string is a palindrome (i.e is the same backwards as forwards), and <false> otherwise e.g.

    ```
    string_palindrome('abcba')=>
    ** <true>
    string_palindrome('ababbaba')=>
    ** <true>
    string_palindrome('abcca')=>
    <false>
    ```

    Do **not** use the built-in procedures **last** and **allbutlast**!

11. (**Hard**) Suppose there are ten people in a town, their names being:

    "a", "b", "c", "d", "e", "f", "h", "i", and "j"

    You are told that the following pairs of people talk to each other:

    [[a f] [f e] [g i] [h b] [c h] [j d] [g j] [b h] [d i]]

    Clearly "a", "e" and "f" form a 'sub-culture' whose members talk to each other but not to the members of any other sub-culture in town. How many sub-cultures are there? Define a procedure called **subculture** which when given a list of two element lists (like that above) groups the pairs together into larger lists, thus:

    ```
    subculture([[a b] [b c] [d e] [e f]])=>
    ** [[a b c] [d e f]]
    subculture([[a e] [b f] [c g] [d g]])=>
    ** [[a e] [b f] [c g d]]
    ```