Advanced Functional and Logic Programming

Labworks 2

October 31, 2018

Deadline: November 15, 2018.

1. Write a tail-recursive version of the function `fact`, where `(fact n)` computes the factorial of `n`.

2. If `n` is a positive integer then `(sum-digits n)` computes the sum of its digits. For example

```
> (sum-digits 243)      > (sum-digits 5)
9                       5
```

   (a) Write a recursive definition of `sum-digits`.

   (b) Write a tail-recursive definition of `sum-digits`.

3. The function call `(increasing-digits? n)` expects a positive integer `n` as input, and should return `#t` if `n` is a sequence of digits in increasing order (e.g., `5`, `579`, `445677`), and `#f` otherwise.

   Write a recursive, preferably tail-recursive, definition of `increasing-digits?`

4. Write down a tail recursive definition of the function `alternating01` such that `(alternating01 l)` returns `#t` if `l` is a list of alternating `0` and `1`, and `#f` otherwise. For example:

```
> (alternating01 '())          > (alternating01 '(1 0 1))
#t                              #t
> (alternating01 '(0 1))       > (alternating01 '(0 1 1 0))
#t                              #f
```

5. The *deep reverse* of a list is the list produced by reversing the elements of all sublists of a list. Implement recursively `(deep-reverse l)` which returns the deep revers of list `l`. For example:

```
> (deep-reverse '((a b) (2 (3 4))))
'(((4 3) 2) (b a))
```

6. Define recursively the function `(deep-symbol->string l)` which replaces all symbols that occur in list `l`, at any depth, into strings. Use the predefined function `symbol-string` which converts a symbol into the corresponding string, and the predicate `symbol?` which recognises symbols. For example:

```
> (deep-symbol->string '(a b (ab a (c 1 2) ())))
 '("a" "b" ("ab" "a" ("c" 1 2) ()))
```

7. Write a tail-recursive version of the function `power-sum` such that the function call `(power-sum x n)` computes the sum $1 + x + \ldots + x^n$ for any number x and integer $n \geq 0$. In the base case when $n = 0$, the function call should return 1.

8. Consider the set of nested lists defined inductively by

   ⟨*nlist*⟩ `::= null | (cons` ⟨*symbol*⟩ ⟨*nlist*⟩`)`
   `| (cons` ⟨*number*⟩ ⟨*nlist*⟩`) | (cons` ⟨*nlist*⟩ ⟨*nlist*⟩`)`

   and the set of substitutions defined recursively by

   ⟨*subst*⟩ `::= null | ((cons (list` ⟨*symbol*⟩ ⟨*value*⟩`))` ⟨*subst*⟩`)`

   where ⟨*value*⟩ can be any value. Define recursively the following functions:

   (a) The recogniser functions `nlist?` for ⟨*nlist*⟩, and `subst?` for substitutions.

   (b) The function `(removeAll` *nlst s*`)` which removes all occurrences of symbol *s* from the nested list *nlst*. For example,

   ```
   (removeAll '(1 s (a s b) s c) 's)
   ```

   should return `'(1 (a b) c)`.

   (c) The function `(get` *subst s*`)` which returns the value paired with symbol *s* in the substitution *subst*, if it exists, and *s* is *s* is not paired with any value in *subst*. For example:

   ```
   > (get '((a 4) (b 6)) 'b)      > (get '((a 4) (b 6)) 'c)
   6                               'c
   ```

   (d) The function `(substitute` *nlst subst*`)` which computes the nested list in which every symbol *s* from *nlst* is replaced with `(get` *subst s*`)`. For example:

   ```
   > (substitute '(s (a ((b)) s (c)) '((s 1) (b 3)))
   '(1 (a ((3)) 1 (c))
   ```

   (e) The function `(countsym` *nlst*`)` which counts the number of symbol occurrences in the nested list *nlst*. For example:

```
> (countsym '((s (1 a) b (c d) s)))
6
```

9. Consider the set of lists of symbols defined inductively by

   ⟨*slist*⟩`::= null | (cons` ⟨*symbol*⟩ ⟨*slist*⟩`)`

   Define recursively the following functions:

   (a) `(slist? sl)` returns `#t` if `sl` is a list of symbols, and `#f` otherwise.

   (b) `(remove-all sl s)` removes all occurrences of symbol `s` from the list `sl`.

   (c) `(remove-first sl s)` removes the first occurrence of symbol `s` from the list of symbols `sl`.

10. Define the function `(notate` *nlst*`)` which replaces every occurrence of a symbol s in a nested list *nlst* with the list `(list` s d`)`, where d is the nesting depth of the occurrence of s in *nlst*. For example:

```
> (notate '(a b (() (a))))        > (notate '(1 b (b (b))))
'((a 1) (b 1) (() ((a 3))))      '(1 (b 1) ((b 2) ((b 3))))
```

11. Consider the set of binary trees defined inductively by

    ⟨*btree*⟩ `::=` ⟨*number*⟩ `| (list` ⟨*symbol*⟩ ⟨*btree*⟩ ⟨*btree*⟩`)`

    The size of a binary tree $bt \in \langle btree \rangle$ is the number of numbers and symbols that occur in *bt*. The *depth* of a binary tree is the number of nodes along the longest path from the root to a leaf node.

    (a) Define recursively the following functions:

        i. `(btsize` *bt*`)` which computes the size of the binary tree *bt*.

        ii. `(btdepth` *bt*`)` which computes the depth of the binary tree *bt*.

    (b) Prove by induction the following facts:

        i. `(btsize` *bt*`)` is an odd number, whenever *bt* is a binary tree.

        ii. If *bt* is a binary tree which is not a number, then the number of numbers in *bt* is an even number.