Answers to the final examination of Erlang

Christian Rinderknecht

14 June 2007

1 Merging sorted lists

Question. Write a function `merge/2` which takes two lists of items increasingly sorted by `<` and returns a new sorted list containing the items of both inputs. For instance

```
> final:merge([3,5,8,8,10,12],[1,2,5,6,7,9,11]).
[1,2,3,5,5,6,7,8,8,9,10,11,12]
```

Make two versions: (1) one not tail recursive, (2) one tail recursive.

Answer.

```
-module(merge).
-compile(export_all).

% rev_append(List1,List2) reverses the list List1 and appends it to
% List2.
%
rev_append([],Suffix) ->
    Suffix;
rev_append([Item|Prefix],Suffix) ->
    rev_append(Prefix,[Item|Suffix]).

% Merging two sorted (<) lists (not tail-recursive)
%
merge([],List2) ->
    List2;
merge(List1,[]) ->
    List1;
merge([Item1|List1],[Item2|List2]) when Item1 < Item2 ->
    [Item1|merge(List1,[Item2|List2])];
merge([Item1|List1],[Item2|List2]) ->
    [Item2|merge([Item1|List1],List2)].

% Same but with tail recursion
%
merge_bis(List1,List2) when is_list(List1), is_list(List2) ->
```

```
    merge(List1,List2,[]).
merge([],List2,Acc) ->
    rev_append(Acc,List2);
merge(List1,[],Acc) ->
    rev_append(Acc,List1);
merge([Item1|List1],[Item2|List2],Acc) when Item1 < Item2 ->
    merge(List1,[Item2|List2],[Item1|Acc]);
merge([Item1|List1],[Item2|List2],Acc) ->
    merge([Item1|List1],List2,[Item2|Acc]).
```

2 Fun with binary trees

The atom `nil` denotes the empty binary tree and the triple `{Left, Root, Right}` is a non-empty whose root is `Root`, left subtree is `Left` and right subtree is `Right`.

2.1 Mirroring binary trees

Question. Write an Erlang function `mirror/1` which takes a binary tree and returns the same tree as in a mirror. Consider the facing example:

Answer.

```
-module(mirror).
-compile(export_all).

mirror(nil) ->
    nil;
mirror({Left,Root,Right}) ->
    {mirror(Right),Root,mirror(Left)}.
```

2.2 Summing

Question. Write a function `sum/1` which takes a binary tree whose nodes are integers and returns the sum of the nodes. In the facing example, the sum is 24.

Answer. The trap here is the empty tree. The sum of the nodes of an empty tree ought to be undefined. But, when following recursively the branches of tree, if we encounter the empty tree, then it can be an acceptable situation, since it means that the parent is actually a leaf of the original tree.

One way to handle the empty tree in these two different contexts consists in writing no clause for the empty tree and use an auxiliary function, *which is not exported* and which handles the empty tree in the recursion. Another technique consists in never accepting the empty tree and stop at the leaves.

```
-module(sum).
-compile(export_all).


sum({nil,Root,Right}) ->
    Root + sum(Right);
sum({Left,Root,nil}) ->
    sum(Left) + Root;
sum({Left,Root,Right}) ->
    sum(Left) + Root + sum(Right).

% With one accumulator
%
sum_bis({Left,Root,Right}) ->
    sum_bis({Left,Root,Right},0).
sum_bis(nil,A) ->
    A;
sum_bis({Left, Root, Right},A) ->
    sum_bis(Right, Root + sum_bis(Left,A)).

% Tail recursive version
%
sum_ter({Left,Root,Right}) ->
    sum_ter({Left,Root,Right},0,[]).
sum_ter(nil,Sum,[]) ->
    Sum;
sum_ter(nil,Sum,[Tree|Forest]) ->
    sum_ter(Tree,Sum,Forest);
sum_ter({Left,Root,Right},Sum,Forest) ->
    sum_ter(Left,Root+Sum,[Right|Forest]).
```