

## Exercise sheet 8 for 10 April 2002

2002-04-03

Do exercises 8.1, 8.2, 8.6, and 8.7. Hand in the solutions.

**Exercise 8.1** Write a tail-recursive version `leni : int list -> int -> int` and a CPS (continuation-passing style) version `lenc : int list -> (int -> int) -> int` of the list length function `len`:

```
fun len []      = 0
  | len (x::xr) = 1 + len xr;
```

**Exercise 8.2** Write a continuation-passing version `revc : 'a list -> ('a list -> 'a list) -> 'a list` of the list reversal function `rev`:

```
fun rev []      = []
  | rev (x::xr) = rev xr @ [x];
```

This should be called as `revc xs id`, where `id = fn v => v` is the identity function. Then observe that the continuation `k` can always be represented by a function of the form `fn u => u @ r`. Use this to obtain an efficient tail-recursive version `revi : 'a list -> 'a list -> 'a list` of the list reversal function.

**Exercise 8.3** Write a tail-recursive version `prodi : int list -> int -> int` and a continuation-passing version `prodc : int list -> (int -> int) -> int` of the list product function `prod`:

```
fun prod []      = 1
  | prod (x::xr) = x * prod xr;
```

**Exercise 8.4** Optimize the tail-recursive version and the CPS version of the `prod` function above. The idea is that both versions could terminate as soon as they encounter a zero in the list (because any list containing a zero will have product zero).

**Exercise 8.5** Write more examples using exceptions and exception handling in the small functional and imperative languages implemented in `cont/fun.sml` and `cont/imp.sml`, and run them using the given interpreters.

**Exercise 8.6** What statements are in tail position in the simple imperative language implemented by `coExec1` in file `cont/imp.sml`? Intuitively, the last statement in a statement block `{ ... }` is in tail position provided the entire block is. Can you argue that this is actually the case, looking at the interpreter `coExec1`?

**Exercise 8.7** In the `coExec1` version of the imperative language in file `cont/imp.sml` (with `Throw` but not `TryCatch`), add an `EThrow` expression to the expression abstract syntax, and rewrite the expression interpreter `eval` in continuation-passing style. Its return type should be `answer` as for `coExec1`, and it should take a normal continuation of type `(int -> answer)` as argument, but no error continuation. An error continuation is not needed when we do not implement exception handling.

Your interpreter should be able to execute `run1 ex4` and `run1 ex5` where

```
val ex4 =
  Block[If(EThrow (Exn "Foo"), Block[], Block[])];

val ex5 =
  While(EThrow (Exn "Foo"), Block[]);
```

**Exercise 8.8** (Project) Write a program to transform programs into continuation-passing style, using the Danvy and Filinski 1992 presentation (which distinguishes between administrative redexes and other redexes).

**Exercise 8.9** (Project) Implement a subset of the language `Icon` (<http://www.cs.arizona.edu/icon/>) using continuations.

**Exercise 8.10** (Somewhat hairy project) Extend a higher order functional language with the ability to capture the current (success) continuation, and to apply it. See papers by Danvy, Malmkjær, and Filinski. It would be a good idea to experiment with `call-with-current-continuation` in Scheme first.