

## Exercise sheet 8 for 2 April 2003

2003-03-26

Do exercises 8.1, 8.2, 8.6, 8.7, and 8.8. Hand in the solutions.

**Exercise 8.1** Write a tail-recursive version `leni : int list -> int -> int` and a continuation-passing style (CPS) 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** The `coExec1` version of the imperative language interpreter in file `cont/imp.sml` supports a *statement* `Throw` to throw an exception. This `Throw` statement is similar to `throw` in Java. Add an *expression* `EThrow` to the expression abstract syntax to permit throwing exceptions also inside an expression, as in SML's `raise` expression. You will need to rewrite the expression interpreter `eval` in continuation-passing style.

The return type of your new expression interpreter should be `answer` as for `coExec1`, and it should take a normal continuation of type `(int -> answer)` as argument, where `answer` is the exact same type used in the `coExec1` statement interpreter. (Like `coExec1`, your new expression interpreter need not take an error continuation, because we do not intend to 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** (1) Using the Icon-style mini-language and interpreter in `cont/backtrack.sml`, write an expression that produces and prints the values 2 4 6 8. Write an expression that produces and prints the values 3 5 7 9. Write an expression that produces and prints the values 21 22 31 32 41 42.

(2) The language (like real Icon) has no Boolean values. Instead, failure is used to mean `false`, and success means `true`. For instance, the less-than comparison operator (`<`) behaves as follows: `3 < 2` fails, and `3 < 4` succeeds (once) with the value 4. Similarly, thanks to backtracking, `3 < (1 to 5)` succeeds (twice) with the values 4 and 5. Use this to write an expression that prints the least multiple of 7 that is greater than 50.

(3) Extend the abstract syntax with unary (one-argument) primitive functions, like this:

```
datatype expr =
  ...
  | Prim1 of string * expr
```

Extend the interpreter `eval` to handle such unary primitives, and define two such primitives: (a) define a primitive `"sqr"` that computes the square  $x \cdot x^2$  of its argument  $x$ ; (b) define a primitive `"even"` that fails if its argument is odd, and succeeds if it is even (producing the argument as result). For instance, `square(3 to 6)` should succeed four times, with the results 9 16 25 36, and `even(1 to 7)` should succeed three times with the results 2 4 6.

(4) If you have time, define a unary primitive `"multiples"` that succeeds infinitely many times, producing all multiples of its argument. For instance, `multiples(3)` should produce 3, 6, 9, ... Note that `multiples(3 to 4)` would produce multiples of 3 forever, and would never backtrack to the subexpression `(3 to 4)` to begin producing multiples of 4.

**Exercise 8.9** (For adventurous Java hackers:) Implement abstract syntax and an interpreter for a backtracking Icon-style language subset, in the spirit of `cont/backtrack.sml`, but do it in Java. You can draw some inspiration from method `fact` in `cont/Factorial.java`. The result will probably appear rather incomprehensible, but it should not be too hard to write if you take a systematic approach.

**Exercise 8.10** (Four-week project) Implement a subset of the language Icon (<http://www.cs.arizona.edu/icon/>). This involves deciding on the subset, writing lexer and parser specifications, and writing an extended interpreter in the style of `cont/backtrack.sml`. The interpreter must at least handle assignable variables.

**Exercise 8.11** (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 8.12** (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.