

## Exercise sheet 2 for 19 February 2003

2003-02-12 \*

It is recommended that you hand in solutions to Exercises 2.1, 2.4, 2.5, and ?? after the exercise classes. If you solve more exercises, you are welcome to hand in those solutions also.

**Exercise 2.1** Define an SML function `linear : int -> int tree` so that `linear n` produces a right-linear tree with  $n$  nodes. For instance, `linear 0` should produce `Lf`, and `linear 2` should produce `Br(2, Lf, Br(1, Lf, Lf))`.

**Exercise 2.2** Lecture 2 presents an SML function `preorder : 'a tree -> 'a list` that returns a list of the node values in a tree, in *preorder* (root before left subtree before right subtree).

Now define a function `inorder` that returns the node values in *inorder* (left subtree before root before right subtree) and a function `postorder` that returns the node values in *postorder* (left subtree before right subtree before root):

```
inorder   : 'a tree -> 'a list
postorder : 'a tree -> 'a list
```

Thus if `t` is `Br(1, Br(2, Lf, Lf), Br(3, Lf, Lf))`, then `inorder t` is `[2, 1, 3]` and `postorder t` is `[2, 3, 1]`.

It should hold that `inorder (linear n)` is `[n, n-1, ..., 2, 1]` and `postorder (linear n)` is `[1, 2, ..., n-1, n]`, where `linear n` produces a right-linear tree as in Exercise ??.

Note that the postfix (or reverse Polish) representation of an expression is just a postorder list of the nodes in the expression's tree representation!

**Exercise 2.3** Extend the expression language `expr` from `sem2.sml` with multiple sequential let-bindings, such as this (in concrete syntax):

```
let x1 = 5+7   x2 = x1*2 in x1+x2 end
```

To evaluate this, the right-hand side expression `5+7` must be evaluated and bound to `x1`, and then `x1*2` must be evaluated and bound to `x2`, after which the let-body `x1+x2` is evaluated.

The new abstract syntax for `expr` might be

```
datatype expr =
  CstI of int
  | Var of string
  | Let of (string * expr) list * expr    (* CHANGED *)
  | Prim of string * expr list;
```

so that the `Let` constructor takes a list of bindings, where a binding is a pair of a variable name and an expression. The example above would be represented as:

```
Let ([("x1", ...), ("x2", ...)], Prim("+", [Var "x1", Var "x2"]))
```

Revise the `eval` interpreter from `sem2.sml` to work for the `expr` language extended with multiple sequential let-bindings.

**Exercise 2.4** Revise the function `closed1 : expr -> bool` to work for the language as extended in Exercise 2.4. Note that the example expression in the beginning of Exercise 2.4 is closed, but `let x1 = x1+7 in x1 end` is not — a variable cannot be used on the right-hand side of its own binding. (There *are* programming languages where a variable can be used in the right-hand side of its own binding, but `expr` is not such a language).

**Exercise 2.5** Revise the `expr-to-texpr` compiler `tcomp : expr -> texpr` from `sem2.sml` to work for the extended `expr` language. There is no need to modify the `texpr` language or the `teval` interpreter to accommodate multiple sequential `let`-bindings.

**Exercise 2.6** Write a compiler (in SML, by modifying `scomp` from `sem2.sml`) that generates stack machine code for the `Stack.seval` interpreter (written in Java) from the original `expr` expression language. The compiler should output a list of integers representing the bytecode instructions.

You may test the output of your compiler by typing in the numbers as an `int` array in the `Stack.java` interpreter. (Or you may solve Exercise ?? below to avoid this manual work).

**Exercise 2.7** Modify the compiler from Exercise ?? to write the lists of integers to a file. An SML list `inss` of integers may be output to the file called `fname` using this SML function (found in `sem2.sml`):

```
fun instofile (inss : int list) (fname : string) : unit =
  let val os = TextIO.openOut fname
      fun outn n = TextIO.output(os, " " ^ Int.toString n);
  in
    List.app outn inss;
    TextIO.closeOut os
  end;
```

Then modify the stack machine interpreter in `Stack.java` to read the sequence of integers from a text file, and execute it as a stack machine program. The name of the textfile may be given as a command-line parameter to the Java program. Reading from the text file may be done using the `StringTokenizer` class or `StreamTokenizer` class (see <http://www.dina.kvl.dk/~sestoft/programming/tekstfiler.pdf> or a suitable Java textbook, or an online Java tutorial).

It is essential that the compiler (in SML) and the interpreter (in Java) agree on the intermediate language: what integer represents what instruction.

**Exercise 2.8** Define Postscript functions corresponding to the SML functions

```
fun addSeven n = n + 7;
fun gauss n = n * (n + 1) div 2;
```

Then

- Evaluate and print `addSeven 20` by typing `20 addSeven =`.
- Evaluate and print `addSeven` for the numbers 0–12 by typing `0 1 12 { addSeven = } for`
- and similarly for the `gauss` function.

**Exercise 2.9** Define a version of the (naive) Fibonacci function

```
fun fib n = if n<2 then n else fib(n-1) + fib(n-2)
```

in Postscript. Compute Fibonacci of 0, 1, ..., 25.

**Exercise 2.10** Define functions similar to `even` and `odd` from `sem2.sml`, in Postscript. You may use that Postscript (like SML) defines constants `true` and `false`. Compute `even` of 10, 11, and 10001.

**Exercise 2.11** Write a Postscript program to compute the sum  $1 + 2 + \dots + 1000$ . It must really do the summation, not use the closed-form expression  $\frac{n(n+1)}{2}$  with  $n = 1000$ . (Trickier: do this using only a `for`-loop, no function definition).