

Exercise sheet 2 for 20 February 2002

2002-02-13

It is recommended that you hand in solutions to Exercises 2.1, 2.4, 2.5, and 2.6 after the exercise classes. If you solve more exercises, you are welcome to hand in those solutions also.

Exercise 2.1 Lecture 2 presents the 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.

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 2.2 Write a compiler (in SML, by modifying `scomp`) that generates stack machine code for the `Stack.seval` interpreter (written in Java) from the `expr` expression language. The compiler should output a sequence of integers representing the bytecode instructions to a file. An SML list `inss` may be output to the file called `fname` using this SML function:

```
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;
```

Exercise 2.3 Modify the stack machine interpreter in `Stack.java` to read the stack machine program, in the form of a sequence of integers, from a text file, and then execute it. 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 2.4 Extend the expression language with multiple simultaneous `let`-bindings, such as this (in concrete syntax):

```
let x1 = 5+7   x2 = 89 * 2 in x1 + x2 end
```

The corresponding abstract syntax might be

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

so that `Let` takes a list of bindings, where a binding is a pair of a variable name and an expression.

Revise the `eval` interpreter to work for the extended `expr` language. The idea is that all the right-hand side expressions should be evaluated, after which all the variables are bound to those values simultaneously. Hence

```
let x = 11 in let x = 22 y = x+1 in x+y end end
```

should compute $12 + 22$ because x in $x+1$ is the outer x (and hence is 11), and x in $x+y$ is the inner x (and hence is 22). Thus, in the let-binding

```
let x1 = e1 ... xn = en in e end
```

the scope of the variables $x1 \dots xn$ should be e , not $e1 \dots en$. In some languages it makes sense to let the scope of $x1 \dots xn$ be the right-hand sides $e1 \dots en$ in addition to e ; in this case the bindings are mutually recursive.

As an aside, observe that in SML, mutual recursion makes sense only for function declarations (not for arbitrary value declarations), such as these:

```
fun even n = if n=0 then true else odd (n-1)
and odd n = if n=0 then false else even(n-1);
```

However, mutually recursive bindings such as this

```
let x = y y = x + 1 in y end
```

make little sense in our expression language. One cannot find an integer value for y that satisfies $y = y + 1$.

Exercise 2.5 Revise the functions `closed : expr -> bool` and `freevars : expr -> string list` to work for the extended language.

Exercise 2.6 Revise the `expr-to-texpr` compiler `tcomp : expr -> texpr` for the extended language.

There is no need to modify the `texpr` language or the `teval` interpreter to accommodate multiple simultaneous let-bindings.

Exercise 2.7 Write a function `check : expr -> bool` for the extended `expr` language that returns true just when the expression is closed and no let-binding binds the same variable twice. For instance, it should return false on this expression

```
let x1 = 7
    x1 = 9
in x1 end
```

because `x1` is bound twice.

Exercise 2.8 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 2.9 Define functions similar to `even` and `odd` from Exercise 2.4, in Postscript. You may use that Postscript (like SML) defines constants `true` and `false`. Compute `even` of 10, 11, and 10001.

Exercise 2.10 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).