

Exercise sheet 5 for 12 March 2003

2003-03-05

Exercises 5.1 and 5.2 are intended to illustrate the difference between Standard ML and Java programming style; the latter exercise uses higher-order functions. The exercises look a bit overwhelming, but that's mostly because of the amount of explanation. Do those exercises (skipping subquestions as indicated) if you feel that you need to strengthen your Standard ML skills. In this case, also do one or two of the next exercises.

Exercises 5.3, 5.4, 5.5 illustrate typical higher-order functions in Standard ML. Do all three exercises if you do not do 5.1 or 5.2. Consider doing also Exercise 5.6 which is about a practical application of higher-order functions.

The later exercises are about the implementation of the higher-order functional language.

Exercise 5.1 The purpose of this exercise is to contrast the Standard ML and Java programming styles, especially as concerns the handling of lists of elements. The exercise asks you to write functions that merge two sorted lists of integers, creating a new sorted list that contains all the elements of the given lists.

(A) Implement an SML function

```
merge : int list * int list -> int list
```

that takes two sorted lists of ints and merges them into a sorted list of ints. For instance, `merge ([3,5,12], [2,3,4,7])` should give `[2,3,3,4,5,7,12]`.

(B) Implement a similar Java method

```
static int[] merge(int[] xs, int[] ys)
```

that takes two sorted arrays of ints and merges them into a sorted array of ints. The method should build a new array, and should not modify the given arrays. Two arrays `xs` and `ys` of ints may be built like this:

```
int[] xs = { 3, 5, 12 };
int[] ys = { 2, 3, 4, 7 };
```

(C) The goal of this exercise is to use `LinkedLists` of `Integers` instead of arrays of ints as in part (B). This appears to be rather cumbersome — at least if you try to use iterators to traverse the lists — so do not waste too much time on this. But if you feel like it, then go ahead: Implement a Java method

```
static LinkedList merge(List xs, List ys)
```

that takes two sorted lists of `Integer` objects and merges them into a sorted `List` of `Integer` objects. The method should build a new `LinkedList`, and should not modify the given `Lists`. The interface `List` and the class `LinkedList` are from the `java.util` package.

Two `Lists` `xs` and `ys` of `Integer` objects may be built like this:

```
LinkedList xs = new LinkedList();
xs.addLast(new Integer(3));
xs.addLast(new Integer(5));
xs.addLast(new Integer(12));
LinkedList ys = new LinkedList();
ys.addLast(new Integer(2));
ys.addLast(new Integer(3));
ys.addLast(new Integer(4));
ys.addLast(new Integer(7));
```

Exercise 5.2 This exercise is similar to the preceding one, but now you should handle sorted lists of arbitrary element type.

(A) Write an SML function

```
mergep : 'a list * 'a list * ('a * 'a -> int) -> 'a list
```

so that `mergep(xs, ys, cmp)` merges the two sorted lists `xs` and `ys`. The third argument `cmp : 'a * 'a -> int` is a comparison function so that `cmp(x, y)` returns a negative number if `x` is less than `y`, zero if they are equal, and a positive number if `x` is greater than `y`.

For instance, with the integer comparison function

```
fun icmp (x, y) = if x<y then ~1 else if x>y then 1 else 0
```

the call `merge ([3,5,12], [2,3,4,7], icmp)` should return `[2,3,3,4,5,7,12]`.

Define a string comparison function `scmp` and write a call to the `mergep` function that merges these lists of strings:

```
ss1 = ["abc", "apricot", "ballad", "zebra"]
ss2 = ["abelian", "ape", "carbon", "yosemite"]
```

Using this function for comparing integer pairs

```
fun pcmp ((x, y1), (y, y2)) =
  if x<y then ~1 else if x=y then icmp(y1,y2) else 1
```

write a call to the `mergep` function that merges these lists of integer pairs:

```
ps1 = [(10, 4), (10, 7), (12, 0), (12, 1)]
ps2 = [(9, 100), (10, 5), (12, 2), (13, 0)]
```

(B) Write a similar Java method

```
static Object[] mergep(Comparable[] xs, Comparable[] ys)
```

that merges two sorted arrays `xs` and `ys` of `Comparable` objects. A `Comparable` object `x` has a method `int compareTo(Object y)` so that `x.compareTo(y)` returns a negative number if `x` is less than `y`, zero if they are equal, and a positive number if `x` is greater than `y`. Since class `Integer` implements the `Comparable` interface, your `mergep` method will be able to merge sorted arrays of `Integer` objects.

As in (A) above, show how to call the `mergep` method to merge two arrays of `Strings`. Class `String` also implements `Comparable`.

As in (A) above, show how to call the `mergep` method to merge two arrays of `IntPair` objects, representing pairs of ints. You will need to define a class `IntPair` so that it implements `Comparable`.

(C) Write a Java method

```
static Object[] mergec(Object[] xs, Object[] ys, Comparator cmp)
```

that merges two sorted arrays `xs` and `ys`. The `Comparator` interface (from package `java.util`) describes a method `int compare(Object x, Object y)` so that `cmp.compare(x, y)` returns a negative number if `x` is less than `y`, zero if they are equal, and a positive number if `x` is greater than `y`.

Show how to call the `mergec` method to merge two arrays of `Integers`.

Exercise 5.3 Define the following polymorphic SML functions on lists using the `foldr` function for lists:

- `filter : ('a -> bool) -> ('a list -> 'a list)`
where `filter p xs` applies `p` to all elements `x` of `xs` and returns a list of those for which `p x` is true.
- `mapPartial : ('a -> 'b option) -> ('a list -> 'b list)`
where `mapPartial f xs` applies `f` to all elements `x` of `xs` and returns a list of the values `y` for which `f x` has form `SOME y`.
Thus `mapPartial (fn i => if i>7 then SOME(i-7) else NONE) [4, 12, 3, 17, 10]` should give `[5, 10, 3]`.

Exercise 5.4 Define the polymorphic SML function `tmap : ('a -> 'b) -> ('a tree -> 'b tree)` so that `tmap f t` creates a new tree of the same shape as `t`, but in which the value of a node is `f v` if the value of the corresponding node in the old tree is `v`.

Exercise 5.5 Define the following polymorphic SML functions on trees using the `tfold` function from the lecture:

- `preorderl : 'a tree -> 'a list`
- `inorderl : 'a tree -> 'a list`
- `postorderl : 'a tree -> 'a list`
- the above function `tmap`
- the function `depth : 'a tree -> int` from exercise sheet 1.

Recall that the preorder, inorder, and postorder traversals were defined in exercise sheet 2.

Exercise 5.6 This exercise is about using higher-order functions for production of HTML code. This is handy when writing Web scripts in Standard ML (as in ML Server Pages, see <http://ellemore.dina.kvl.dk/~sestoft/msp/index.msp>) or when generating static webpages from database information.

(A) Write an SML function

```
htmlrow : int * (int -> string) -> string
```

that builds one row of a numeric HTML table. For example,

```
htmlrow (3, fn j => Int.toString(j * 8))
```

should produce the string

```
"<TR><TD ALIGN=RIGHT>0<TD ALIGN=RIGHT>8<TD ALIGN=RIGHT>16"
```

Write an SML function

```
htmltable : int * (int -> string) -> string
```

that builds an HTML table. For example,

```
htmltable (3, fn i => String.concat["<TD>", Int.toString i,
                                     "<TD>", Int.toString(i*8)]);
```

should produce the string

```
"<TABLE>\n<TR><TD>0<TD>0\n<TR><TD>1<TD>8\n<TR><TD>2<TD>16\n</TABLE>"
```

The `\n` means newline as in Java. Similarly,

```
htmltable (10, fn i => htmlrow(10, fn j => Int.toString((i+1)*(j+1))));
```

should produce a 10-by-10 multiplication table in HTML.

(B) Implement methods similar to `htmlrow` and `htmltable` in Java. (This is cumbersome).

Exercise 5.7 Add anonymous functions, similar to SML's `fn x => ...`, to the higher-order functional language:

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

Anonymous functions give rise to non-recursive closures of the form

```
datatype value =
  ...
  | Clo of string * expr * vfenv (* (x, body, bodyenv) *)
withtype vfenv = (string, value) env
```

For instance, the expression `fn x => 2*x` might evaluate to this closure:

`Clo("x", Prim("*", [CstI 2, Var "x"]), [])` in the empty environment `Env.empty`.

Similarly, the expression `let y = 22 in fn z => z+y end` might evaluate to this closure:

`Clo("z", Prim("+", [Var "z", Var "y"]), [y ↦ 22])`.

Extend the evaluator in file `fun/hofun.sml` to interpret such anonymous functions.

Exercise 5.8 Extend the lexer and parser specification to permit anonymous functions. The concrete syntax may be as in SML: `fn x => expr`, where `x` is a variable.

Exercise 5.9 Write an SML function `check : expr -> bool` that checks that all variables and function names are defined when they are used, and returns `true` if they are. This checker should accept the higher-order language. That is, in the abstract syntax `Call(e1, e2)` for a function call, the expression `e1` can be an arbitrary expression and need not be a variable name.

The `check` function needs to carry around an 'environment' to know which variables are bound. This environment can just be a simple list of the bound variables.

Exercise 5.10 Add mutually recursive function declarations in the higher-order functional language:

```
datatype expr =
  ...
  | Letfun of fundef list * expr
  | ...
where type fundef = string * string * expr
```

Exercise 5.11 Design a concrete syntax for the higher-order functional language extended with lists, extend the lexer and parser specifications, and write some example programs in concrete syntax. (This is especially interesting if you have already implemented lists in the functional language abstract syntax, as suggested in an Exercise from sheet 4).

Exercise 5.12 Implement some higher-order functions on lists as examples. You may get inspiration from the structure `List` in Moscow ML or Standard ML Basis Library.