

COMPUTER SCIENCE TRIPOS Part IA – 2012 – Paper 1

2 Foundations of Computer Science (LCP)

*This question has been translated from Standard ML to OCaml*

second half of the course, in particular lectures 10–14

- (a) Write brief notes on `fun`-notation and curried functions in OCaml. Illustrate your answer by presenting the code for a polymorphic curried function `replicate`, which given a non-negative integer  $n$  and a value  $x$ , returns the list  $[x; \dots; x]$ . [6 marks]

---

*Answer:* The syntax of `fun`-notation is `fn x -> E`, denoting a function with argument  $x$  that returns when called the value of  $E$ . The point is the ability to express such a function without having to name it first. This notation, obviously, always yields expressions that have a function type derived from the types of  $x$  and  $E$ .

Given `fun`-notation, we can express curried functions, that is, functions that return another function as their result, and this function will typically make use of supplied argument. One advantage of currying is that it allows partial application: regarding such a returned function as useful in its own right.

OCaml provides a special syntax for writing curried functions, which is especially useful in the case of recursion. The function `replicate` can be declared as follows:

```
let rec replicate n x =
  if n = 0 then
    []
  else
    x :: replicate (n - 1) x
```

Its polymorphic type is `int -> 'a -> 'a list`.

---

- (b) Write brief notes on references in OCaml. Illustrate your answer by discussing (with the aid of a diagram) the effect of the following two top-level declarations:

```
let rlist = replicate 4 (ref 0) @ List.map ref [1; 2; 3; 4]
let slist = List.map (fun r -> ref !r) rlist
```

[6 marks]

---

*Answer:* The key concepts of references include the function `ref`, which creates a reference cell, `!`, which inspects a reference cell, and `:=`, which updates a reference cell with a given value. These cells are mutable, but the references to them are pure values, like everything else in OCaml.

The first declaration yields a list whose first four elements refer to a single shared reference cell containing zero. (The diagram should show this sharing.) The remaining four list elements refer to reference cells containing the integers 1 up to 4, respectively. The second declaration creates a new list referring to freshly-allocated (and therefore distinct) reference cells, the first four containing zero and the remaining four again holding the integers 1 up to 4. Students are expected to understand the use of `List.map` in this question.

---

- (c) The following three lines are typed at the OCaml top-level, one after the other. What value is returned in each case? Justify your answer clearly. [Note: Recall that an expression of the form  $v := E$  has type `unit`.]

— *Solution notes* —

```
List.map (fun r -> (r := !r + 1)) rlist
List.map (fun r -> (r := !r - 1; !r)) rlist
List.map (fun r -> (r := !r + 3; !r)) slist
```

[8 marks]

---

*Answer:* The values of the three lines are given as follows:

```
- : unit list = [(); (); (); (); (); (); (); (); ()]
- : int list = [3; 2; 1; 0; 1; 2; 3; 4]
- : int list = [3; 3; 3; 3; 4; 5; 6; 7]
```

The first one is trivial, for students who can remember that the only value of type `unit` is `()`. But students must also recognise that this first line has side effects: the first shared reference in `rlist` is increased by four (in four separate increments) while the remaining four references are increased by one.

For the second one, students need to understand the semi-colon notation, which in this case returns the contents of the reference at the given moment. The first four values of this result illustrate successive decrementing of the shared reference.

For the third one, students need to understand that all the references in `slist` are independent.

---