Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Letivo de 2021/2022
Linguagens e Ambientes Programação – Teste 1

08 de abril de 2022 às 18:30

Test with limited consultation and 1 hour and 30 minutes duration + 10 minutes tolerance

Nome: Num:

Notas: *This test paper consists of three groups of questions. Answer in the exam paper itself, using the front and the back. Normally, imperfect answers deserve some points. Fraud implies failure in the course.*

1. [3 points] Multiple choice. Wrong answers do not discount. Provide the answers here**:**

A	B

A) Interpretation overhead refers to all the management work that the interpreter needs to perform, in addition to the useful work that actually achieves the desired effects of the program's execution. The three points below describe the essential activities performed by interpreters. Which of these three points correspond to the aforementioned interpretation overload? Choose the correct letter option.

1. Performs the elementary language operations, eg change the value of a variable, call a function, make a decision. **2**. Executes the so-called "interpretation cycle" which consists of parsing the code and launching operations. **3**. Access to interpreter data structures, for example using a hashtable to discover a variable or a function with a given name.

a) 1, 2

b) 2, 3

c) 1, 3

d) 3

B) Consider scenarios **1** and **2** and then choose the letter option:

1. Prolog compiler, written in OCaml. This compiler is compiled using the **ocamlopt** native code compiler.
2. Prolog interpreter, written in OCaml. This interpreter is run using the OCaml interpreter **ocaml**.

a) The two implementations, **1** and **2**, seriously suffer because of the combined effect of the two levels of interpretation and compilation. In both cases, the execution speed is seriously affected.

b) Neither of the two implementations suffers from the combined effect of the two levels of interpretation and compilation. The effect on the speed of execution of the two levels is equivalent to that of only one level

c) In version **1**, Prolog's execution speed is extremely slow, because of the overhead associated with the two levels of compilation combined. Programs interpreted using version **2** of the Prolog implementation will run at the same speed as only one level of interpretation.

d) The version **2** implementation of Prolog is extremely slow, because of the overhead associated with the two levels of interpretation combined. Programs compiled using version 1 of the Prolog implementation will run at full speed, with no harm done because of the two levels of compilation.

2. [3 points] Which is the OCaml type of the following function, named **f**:

```
let f (a, b) x = (a (b x), b (a x))
```

3. **Linear paths** - A **linear path** is a non-empty sequence of points in a two-dimensional plane. Each point is an ordered pair of integers. Our linear paths are **directed**, which means they travel forward along the sequence, never backwards.

```
type point = int * int
type path = point list
```

The following function tests whether two points are adjacent:

```
let areAdjacent (x1, y1) (x2, y2) =
    abs(x2 - x1) <= 1 && abs(y2 - y1) <= 1
```

Here is an example of a complex path, with three discontinuities and two self-intersecting points (points **(4,4)** e **(0,8)**):

```
let example: path = [
        (0,4); (1,4); (2,4); (3,4); (4,4); (5,4); (6,4);                  (* line 1 *)
        (4,0); (4,1); (4,2); (4,3); (4,4); (4,5); (4,6);                  (* line 2 *)
        (0,0); (1,1); (2,2); (3,3); (4,4); (5,5); (6,6);                  (* line 3 *)
        (0,8); (1,8); (2,8); (2,9); (2,10); (1,10); (0,10); (0,9); (0,8)  (* square *)
]
```

A **segment** is a simple path with no discontinuities and usually no intersections. The only intersection allowed in a segment is the one that occurs when the two ends of the segment coincide. In this case, the segment is said to be **closed**..

The following functions allow accessing the start point and end point of a segment:

```
let first p = List.hd p
let last p = List.hd (List.rev p)
```

To solve complex problems, sometimes we need to decompose a path, breaking it at the discontinuity places and also in various intersection points. When breaking at a point of intersection, sometimes it is necessary to duplicate that point so as not to lose information about continuity. Study below, the sequence of seven segments that corresponds to the example path given above:

```
let segments: path list = [
        [(0,4); (1,4); (2,4); (3,4); (4,4)];   [(4,4); (5,4); (6,4)];
        [(4,0); (4,1); (4,2); (4,3); (4,4)];   [(4,4); (4,5); (4,6)];
        [(0,0); (1,1); (2,2); (3,3); (4,4)];   [(4,4); (5,5); (6,6)];
        [(0,8); (1,8); (2,8); (2,9); (2,10); (1,10); (0,10); (0,9); (0,8)]
]
```

Bibranched paths - A **bibranched path** is a non-empty binary tree of points in a two-dimensional plane. At each point there is the option to continue the path via two lanes (and if both lanes are used, this point is considered an intersection). There is also the option to continue the path using only one of the lanes, or even none of them (in the case of a discontinuity). The paths are **directed**, which means that they are always followed along down the tree, walking in the direction of the leaves, and never in the opposite direction.

```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree
type treePath = point tree
```

The following function obtains the root of a bibranched path:

```
let root t =
    match t with
    | Nil -> failwith "root: empty tree"
    | Node(x,_,_) -> x
```

Here are two examples of bi-branching paths:

```
let example2 =
    Node((0,0),
        Node((0,1),
              Node((0,2), Nil, Nil),
              Nil),
        Node((1,0),
              Node((2,0), Nil, Nil),
              Nil))

let example3 =
    Node((0,0),
        Node((0,1),
              Node((0,2), Nil, Nil),
              Nil),
        Node((2,2),
              Node((3,3), Nil, Nil),
              Nil))
```

In the following problems, show that you know how to use the inductive method and write, if possible, category 1 or 2 functions. You can also use helper functions written by you or belonging to the OCaml standard library. Do not use imperative mechanisms or reasoning or simulate imperative mechanisms or reasoning.

a) [2.5 points] Write a function **countDescontinuities** that, given a linear path, counts the number of discontinuities.

```
countDiscontinuities: path -> int
```

Example: `countDescontinuities example = 3`

b) [2.5 points] In a linear path, any consecutive duplicate points are useless and create complications. Write a function **clear** that, given a path, eliminates all consecutive duplicate points (and does not eliminate anything else).

```
clear: path -> path
```

Example: `clear [(0,0);(0,1);(0,1);(0,1);(0,2);(0,1)] = [(0, 0); (0, 1); (0, 2); (0, 1)]`

c) [2.5 points] Write a function **joinSegments** that converts a linear path represented by a sequence of segments into the corresponding normal linear path. You will need to join lists and also eliminate the artificial duplications present at the points of connectivity between segments.

```
joinSegments: path list -> path
```

Example: `joinSegments segments = example`

d) [2.5 points] Write a function **tree2path** to convert a bibranched path into the equivalent linear path. At each branching point, treat the left lane first and then the right lane. The result should not have useless points, but it may be necessary to duplicate some points (for example, the point **(0,0)**, in the two examples shown below).

```
tree2path: pathTree -> path
```

Examples:
```
tree2path example2 = [(0,0); (0,1); (0,2); (0,0); (1,0); (2,0)]
tree2path example3 = [(0,0); (0,1); (0,2); (0,0); (2,2); (3,3)]
```

e) [2 points] Write a function **`treeIsContinuous`** to check the continuity of a bibranched path. For a path to be continuous, there can be no discontinuities in descending paths. [The function `root` helps to simplify the code.] [The second example returns `false` because the points `(0,0)` and `(2,2)` are not adjacent.]

Examples: **`treeIsContinuous`** `example2 = true` **`treeIsContinuous`** `example3 = false`

f) [2 points] Write a function **`shortest`** to determine, in a continuous bibranched path, the shortest distance between two given points. What is at stake here is to traverse the tree in a descending direction, from the first point to the second point. If no such path exists, the result is **`max_int`**. Note that both the source point and the destination point can occur in the tree more than once.

```
shortest: treePath -> point -> point -> int
```

Examples: **`shortest`** `example2 (0,0) (0,2) = 2` **`shortest`** `example2 (0,1) (0,2) = 1`
`shortest` `example2 (1,0) (0,2) = max_int`