Programming Languages and Environments – 2023/2024

Second Midterm June 3rd, 2024

Notes: The test is closed-book and lasts 1.5 hours. Answer on the corresponding answer sheet. You may use either pencil or pen.

Part 1

Version Number: 124862

Q-1 [*] (1 point) This question is about the classification of programming languages as declarative and/or imperative. **Select the correct option:**

A - The OCaml programming language is pure functional and it's a declarative language.

B - The C programming language is imperative and it is a declarative language.

C - Recursion is a flow control mechanism and it's imperative.

D - The use of state variables is the core mechanism of imperative languages.

Q-2 [*] (1 point) This question is about compiling and interpreting programs. **Select the correct option:**

A - The use of interpreted languages is necessarily linked to functional languages.

B - Intermediate languages allow the use of a set of optimizations in several languages.

C - Interpreting programs is always faster than compiling them.

D - Interpreting programs is always faster than executing the same compiled program.

Q-3 [*] (1 point) This question is about intermediate languages, interpreters and compilers. **Select the correct option:**

A - Compilers are programs that translate programs written in an intermediate language into machine language that is interpreted by a virtual machine.

B - A compiler *Just in Time* translates instructions in an intermediate language into machine code at load time.

C - An intermediate language is a low-level programming language interpreted by a stack machine.

D - A *Just in Time* compiler produces native code from source code at load time.

Q-4 [*] (1 point) This question is about mutability in OCaml. **Select the correct option:**

A - The OCaml programming language is pure functional and there are no state variables.

B - The use of references in OCaml can be complementary to functional programming, optimizing through techniques such as *memoization*.

C - The OCaml programming language uses state variables to implement evaluation of *Lazy* expressions.

D - Mutability in OCaml is done through global variables.

Q-5 [*] (1 point) This question is about execution support systems. **Select the correct option:**

A - A stack machine is characterized by using less memory to execute programs in an intermediate language.

B - The intermediate language of a stack machine has instructions with less parameters than a register machine.

C - A register machine is less efficient than a stack machine because it uses more complex instructions.

D - Recursion is a flow control mechanism only possible in a stack machine.

Q-6 [*] (1 point) This question is about side effects in programming languages. **Select the correct option:**

A - The side effects of functional languages make programs more efficient.

B - The concept of side effects corresponds to modifying the contents of state variables.

C - A programming language is pure if it has side effects.

D - Side effects are a mechanism that allows concurrent programming.

Part 2

This group is about algorithms on recursive algorithms on algebraic types. Consider the data type `'a tree` which represents a binary tree.

```
type 'a tree = | Leaf | Node of 'a * 'a tree * 'a tree
```

Q-7 [*] (3 points). Implement the 'mirror' function that inverts the tree with the following signature:

```
mirror: 'a tree -> 'a tree
```

as a result, consider the example,

```
let t = Node(1, Node(2, Leaf, Leaf), Node(3, Leaf, Leaf))
let _ = assert (mirror t = Node(1, Node(3, Leaf, Leaf), Node(2, Leaf, Leaf)))
```

Q-8 [**] (3 points). Implement the zip_tree function that joins two trees pairwise with the following signature:

```
zip_tree: 'a tree -> 'b tree -> ('a * 'b) tree
```

at each step, the zip_tree function joins the nodes of the two trees, and if at a certain point there is no possible pair, it terminates with an exception Expecting Node, found Leaf. As a result, consider the example,

```
let t = Node(1, Node(2, Leaf, Leaf), Node(3, Leaf, Leaf) ) in
let t' = Node(4, Node(5, Leaf, Leaf), Node(6, Leaf, Leaf) ) in
let t'' = zip_tree t t' in
assert (t'' = Node((1, 4), Node((2, 5), Leaf, Leaf), Node((3, 6), Leaf, Leaf)))
```

Q-9 [***] (3 points). Implement the function map2_with_or_without which applies a transformation function to each pair of nodes in two trees. The transformation function is designed as having optional parameters via the option type. If one of the trees is smaller than the other somewhere, the transformation function is applied using None instead, otherwise the constructor Some is used. The map2_with_or_without function has the following signature:

```
val map2_with_or_without : ('a option -> 'b option -> 'c) -> 'a tree -> 'b tree -> 'c tree
```

as a result, consider the example,

```
let z = function | Some x -> x | None -> 0 in
let f = (fun x y -> z x + z y) in
let t = Node(1, Node(2, Leaf, Leaf), Node(3, Leaf, Leaf) ) in
let t' = Node(4, Node(5, Node(6, Leaf, Leaf), Leaf), Leaf ) in
assert (map2_with_or_without f t t' = Node(5, Node(7, Node(6, Leaf, Leaf), Leaf), Node(3, Leaf, Leaf)))
```

Parte 3

This group is about the use of modules in OCaml.

Q-10 [*] (2 points). Consider the following signature of a module that implements a 24-hour clock and an example implementation using a register.

```
module type Clock = sig
  type time
  val compare : time -> time -> int
  val clock_of_seconds : int -> time
  val seconds_of_clock : time -> int
  val minutes_of_clock : time -> int
  val hours_of_clock : time -> int
end

module Date = struct
  type t = { hours : int; minutes : int; seconds : int }
  let compare d1 d2 =
    if d1.hours <> d2.hours then d1.hours - d2.hours
    else if d1.minutes <> d2.minutes then d1.minutes - d2.minutes
    else d1.seconds - d2.seconds
  let clock_of_seconds s = { hours = s / 3600; minutes = (s mod 3600) / 60; seconds = s mod 60 }
  let seconds_of_clock d = d.seconds
  let minutes_of_clock d = d.minutes
  let hours_of_clock d = d.hours
end
```

Implement a module ClockShort that implements the module signature Clock with an internal representation data type int that represents the number of seconds since midnight. **Give an example of using** the 'ClockShort' module that creates and compares two clocks, one reads 5h30 and the other reads 6h00, your example should say that the second goes ahead.

Part 4

This group is about the design of interpreters and compilers. Consider the following type for representing arithmetic expressions:

```
type ast =
  | Num of int
  | Add of ast * ast
  | Sub of ast * ast
  | Mul of ast * ast
  | Div of ast * ast
```

and the 'eval' function which evaluates an arithmetic expression:

```
let rec eval = function
  | Num n -> n
  | Add (a, b) -> eval a + eval b
  | Sub (a, b) -> eval a - eval b
  | Mul (a, b) -> eval a * eval b
  | Div (a, b) -> eval a / eval b
```

Q-11 [*] (1 point). Extend the ast type with the integer division remainder operation and the eval function to support this operation.

Q-12 [****] (2 points). **Extend the type** ast with the sum operation $(\sum_{i=1}^{n} E)$ **and extend the function** eval to support this operation. Consider the following examples, where you can use the variable Vari to represent the variable of the sum and Sum to represent the sum operation. The variable of the sum is fixed (the variable is "i" in the example above) and can occur in the internal expression of the sum, the first argument is the lower limit and the second argument is the upper limit. As a suggestion, you could also change the signature of the eval function with an integer value to serve as an accumulator. The aim is to make a pure function, with no side effects. The evaluation of the sum $\left(\sum_{i=1}^{10} 2i\right)$ should result in 110 and be represented as follows:

```
eval 0 (Sum (Num 1, Num 10, (Mul (Num 2, Vari))) = 110
```

and the evaluation of the sum $\left(\sum_{i=1}^{10} \left(\sum_{i=1}^{i} i\right)\right)$ should result in 440 and be represented as follows. Note that the variable i is used as the upper limit of the second summation and is local to the first summation. The representation in OCaml would be:

```
eval 0 (Sum (Num 1, Num 10, (Mul (Num 2, (Sum (Num 1, Vari, Vari)))))) = 440
```