School of Computer Science and Software Engineering

SAMPLE EXAMINATION 2010

CITS3242 PROGRAMMING PARADIGMS

SURNAME: ___________________________ STUDENT NO: __________
GIVEN NAMES: ______________________ SIGNATURE: __________

This paper contains: 10 pages (including the title page)
Time allowed: 2 Hours 10 minutes

This paper contains 6 questions, each worth 10 marks.
Candidates should attempt FIVE out of the six questions.

TOTAL MARKS: 50 marks

The actual exam will have the same format, and a very similar coverage of topics.

PLEASE NOTE

Examination candidates may only bring authorised materials into the examination room. If a supervisor finds, during the examination, that you have unauthorised material, in whatever form, in the vicinity of your desk or on your person, whether in the examination room or the toilets or en route to/from the toilets, the matter will be reported to the head of school and disciplinary action will normally be taken against you. This action may result in your being deprived of any credit for this examination or even, in some cases, for the whole unit. This will apply regardless of whether the material has been used at the time it is found.

Therefore, any candidate who has brought any unauthorised material whatsoever into the examination room should declare it to the supervisor immediately. Candidates who are uncertain whether any material is authorised should ask the supervisor for clarification.
Instructions

This paper contains 6 questions each worth 10 marks. Candidates should attempt FIVE out of the six questions, for a total of 50 marks.

If more than five questions are attempted (in the actual exam) only the first five will be marked, unless you clearly write “NOT ATTEMPTED” near the start of the answer that you wish to be excluded.
1. (a) Recall the F# type for binary trees introduced in lectures and labs.

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

The following function takes a tree of lists, and returns a list containing all elements in all lists in the tree, in left-to-right order.

```fsharp
let rec treeToListApp t acc =
  match t with
  | Leaf -> acc
  | Node (l, x, r) -> (treeToListApp l (x::acc)) @ (treeToListApp r [])

let rec treeToList t = treeToListApp t []
```

Explain why this code is inefficient when there are many elements in the tree.

(b) Write an efficient version of the function `treeToListApp` in the previous part.

(c) Consider the following F# code:

```fsharp
type point = float * float

type region = point -> bool // returns true for points in the region.

let rectangle (x1, y1) (x2, y2) : region = fun (x, y) ->
  (x>=x1) && (y>=y1) && (x<=x2) && (y<=y2)

let regionUnion reg1 reg2 pt = reg1 pt || reg2 pt
let regionInverse reg = reg >> not

let myRegion = rectangle (0.0, 0.0) (2.0, 3.0)
  |> regionInverse
  |> regionUnion (rectangle (1.0, 1.0) (4.0, 5.0))

let regionContains : region -> point -> bool = fun reg -> reg
```

Explain what `regionContains` does, and why it is so easy to implement. Your explanation should include examples based on applying `regionContains` to `myRegion` and a number of different points.

(d) Rewrite the definitions of `regionInverse` and `myRegion` above so that they do the same thing, but without using the the composition and pipeline operators: `>>` and `|>`. 
2.  

(a) Consider the following F# code that constructs the set of all “paths” in a tree. A path is a list of elements from the tree, starting at a leaf, with each element in one of the two children of the element that follows it, and the final element in path being the element at the top of the tree.

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

let rec allPathsHelp pathSofar = function
    | Leaf -> Set.singleton pathSofar
    | Node(l, x, r) ->
        let newPath = x::pathSofar
        Set.union (allPathsHelp newPath l) (allPathsHelp newPath r)
```

What main advantage or disadvantage does sharing of data structures lead to in the above code? Be sure to say what is shared, and how the sharing arises.  

(3)

(b) The function `allPaths` returns a set of immutable lists. Suppose we modify it to return a set of mutable lists, and then write code that uses these lists imperatively. What advantages or disadvantages might this imperative version have compared to the original, functional version?  

(3)

(c) Use sequence comprehension(s) and `allPaths` to write a function that takes a tree containing integers and finds the length of the path that contains the most even numbers. Hint: you may find some or all of the following useful.

```fsharp
% List.length  Seq.length  Seq.max
List.forall:('a -> bool) -> 'a list -> bool
```
3.

(a) What is *aliasing* in the context of imperative programs? Give an example of an imperative F# function definition that can lead to complicated behaviour when applied to arguments that involve aliasing.

(b) Recall the following example of “rolling our own” objects using F# features other than the built-in objects:

```fsharp
type point = {SetX : float -> unit;  // Interface
              Dist : unit -> float  }

let point initX initY = // Constructs point objects
    let x = ref initX // Instance variables (state)
    let y = ref initY
    let setX newX = (x:=newX) // Functions for the methods
    let dist () = sqrt (!x * !x + !y* !y)
    {SetX = setX; Dist = dist} // Package into a record object

let p = point 2.0 3.0 // Construct an object
p.SetX 4.0 // Call its methods
p.Dist() // YIELDS: val it : float = 5.0
```

Is it possible to “roll our own” objects in a roughly similar way using the features of Java, including Java objects? Explain how it can be done, or alternatively why it cannot be done. Are there any situations where this might be a desirable thing to do?

(c) Consider the following Prolog code:

```prolog
revAppend([],X,X).
revAppend([X|Y],Z,W) :- revAppend(Y,[X|Z],W).
```

What results does each of the following queries produce? Explain, showing the main steps taken in evaluating each query.

(i) \(-\) revAppend([1,2,3],[], ZS).
(ii) \(-\) revAppend(XS, YS, [1,2]).
(iii) \(-\) revAppend(XS, [2,3], [1,2,3]).
4.

(a) What does it mean to have multiple threads in a program, and why is this sometimes useful?  

(b) Consider the following code in which five clients each have a thread, and take turns in being “in charge” of some (simulated) shared resource. When a client takes charge, it will “use” the resource, then give it to the next client that asks for it.

```fsharp
open System.Threading
let mkThread f = Thread (ThreadStart f)
let startThread f = (mkThread f).Start()
let waitFor obj = ignore(Monitor.Wait obj)
let wakeWaiters obj = Monitor.PulseAll obj

let numClients = 5

type Client(clientID:int) =
    let clients:Client[ ] ref = ref Array.empty
    let inCharge = ref (if clientID=0 then true else false)
    let usingNow = ref false
    member this.ClientID = clientID
    member this.init(theClients) = clients:=theClients
                                startThread <| fun () ->
                                    for i in 1..10 do this.NewRequest()
    member this.NewRequest() = // Makes the client do a new request
        while(not (!inCharge)) do
            inCharge := !clients |> Array.exists (fun cl -> cl.CanIHaveIt())
        lock this <| fun() ->
            usingNow:=true
            printfn "Client %d took charge" clientID
            Thread.Sleep 100 // Only one client should do this at a time.
            usingNow:=false
            wakeWaiters this
    member this.CanIHaveIt() = // Returns true when giving to the caller
        lock this <| fun() ->
            while (!usingNow) do waitFor this
            let wasInCharge = !inCharge
            inCharge:=false
            wasInCharge

let clients = Array.map (fun cID -> Client(cID)) [|0..numClients-1|]
clients |> Array.iter (fun cl -> cl.init clients)
```
(i) Does the above code correctly prevent two different clients from reaching the `Thread.Sleep 100` at the same time? If yes, say why in detail. If no, say how the code should be modified to achieve this.

(ii) For the above code, including any modifications you made in the previous part, give an example of an interleaving between two threads that results in a deadlock.

(iii) Can increasing the granularity of locks avoid the possibility of deadlocks? What about decreasing the granularity? For each explain why or why not.
5.

(a) Consider the following code based on that presented in lectures, but with modifications near the end:

```csharp
type acctMsg = BalanceMsg of AsyncReplyChannel<int>
    | WithdrawMsg of int
    | DepositMsg of int
    | TransferMsg of account * int

and account(name:string) =
    let agent = MailboxProcessor.Start(fun inbox ->
        let balance = ref 1000
        let withdraw amt = balance := !balance - amt
        let rec loop() =
            async {
                let! msg = inbox.Receive()
                match msg with
                | BalanceMsg replyCh-> replyCh.Reply(!balance)
                | WithdrawMsg amt -> withdraw amt
                | DepositMsg amt -> balance := !balance + amt
                | TransferMsg (toAcct, amt) -> withdraw amt
                    toAcct.Debit amt
            return! loop()
        
        loop() )
    member this.Balance =
        agent.PostAndReply (fun replyChan -> BalanceMsg replyChan)
    member this.Name = name
    member this.Withdraw amount = agent.Post (WithdrawMsg amount)
    member this.Deposit amount = agent.Post (DepositMsg amount)
    member this.Transfer (toAcc:account) amount =
        agent.Post (TransferMsg (toAcc, amount))

let doTransfers (acc:account) toAcc () =
    for i in 1..100 do acc.Transfer toAcc 100

let acc1=account("Account1")
let acc2=account("Account2")
startThread (doTransfers acc1 acc2)
startThread (doTransfers acc2 acc1)
```
Give an example of a possible output from this program. Include a description of the sequence of messages that each agent/message-processor receives, and exactly what messages are in the queue each time a balance is retrieved.

(Hint: the arguments to each printfn are evaluated from left to right, and nothing is printed until all arguments are evaluated.)

(b) What is the difference between let and let! in an F# asynchronous computation? Why are both needed?

(c) What is Software Transactional Memory, and how does it prevent deadlocks?
6. (a) Recall the following representation of Booleans in the pure $\lambda$-calculus.

$$\text{True} = \textbf{fun} \; x \; y \rightarrow \; x \quad \text{False} = \textbf{fun} \; x \; y \rightarrow \; y \quad \text{if} = \textbf{fun} \; x \; y \; z \rightarrow \; x \; y \; z$$

The Boolean \textit{aand} function can be implemented using these as follows:

$$\textbf{fun} \; a \; b \rightarrow \; \text{if} \; a \; b \; \text{False}$$

Substitute in the definitions of \text{True}, \text{False} and \text{if} into the following $\lambda$-expression and then reduce the expression to normal form using normal order reduction. Show each step in the reduction.

$$(\textbf{fun} \; a \; b \rightarrow \; \text{if} \; a \; b \; \text{False}) \; \text{True}$$

(b) Explain what a \textit{most general unifier} of two types is, using the following two types as an example:

$$'a*'b \rightarrow 'b*'a$$
$$'c \rightarrow 'c$$

(c) Broadly outline the main steps in the algorithm for type inference presented in lectures. You do not need to include the details of how unification is done, nor the precise way that F# constructs like applications and functions are handled.

(d) Give the three main types of garbage collectors, as covered in lectures, and briefly explain the differences between them.