# Midterm Exam #1 Pre-Posted Background

2013/10/08 50 mins

# 1 Eager Beaver [4 marks]

## 1.1 Pre-Posted Background

Eager and lazy evaluation differ in their treatment of actual parameter expressions at function call sites. In eager evaluation, the actual parameter is evaluated to a value before calling the function and then that value is bound to the function's formal parameter. In lazy evaluation, the unevaluated actual parameter expression is bound to the function's formal parameter. It is only evaluated later if actually used.

# 2 Non-Deterministic Numbers, Shaken and Stirred [28 marks]

## 2.1 Pre-Posted Background

Each of the following problems is based on our "non-deterministic numbers" language. However, each one can be completed *regardless* of whether you complete the other parts!

In several parts, we work with a new variant in our abstract syntax: `combineC`. The result of evaluating `(combineC <ast1> <ast2>)` should be a normalized non-deterministic number that includes all numbers from the value of `<ast1>` and all numbers from the value of `<ast2>`.

For example:

- `(combineC (numSetC (list 1)) (numSetC (list 2)))` evaluates to `(list 1 2)`
- `(combineC (numSetC (list 1 3 4)) (numSetC (list 3 5 7)))` evaluates to `(list 1 3 4 5 7)`
- `(combineC (numSetC empty) (numSetC (list 2 5)))` evaluates to `(list 2 5)`

Here's the implementation without these new pieces. The parser and all test cases have been left out as have the bodies of the `map2-non-deterministic` and `normalize-set` helper functions.

```
; consumes lon and produces the normalized set representing lon
; a normalized set is a sorted list of numbers with no duplicates
(define (normalize-set [lon : (listof number)]) : (listof number)
  ...)
```

```
; Consumes a binary numeric function and transforms it to work on
; non-deterministic numbers.  The new function operates on lists of
; numbers and gives back the (normalized) result of applying the old
; function to each pair of numbers where one comes from the first list
; and the other from the second.
(define (map2-non-deterministic [f : (number number -> number)])
  : ((listof number) (listof number) -> (listof number)) ...)

; ArithS AST
(define-type ArithS
  [numS (n : number)]
  [numSetS (ns : (listof number))]
  [plusS (l : ArithS) (r : ArithS)]
  [multS (l : ArithS) (r : ArithS)])

; parse s-expressions into ArithS ASTs
(define (parse [s-exp : s-expression]) : ArithS
  ...)

; ArithC AST
(define-type ArithC
  [numSetC (n : (listof number))]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])

; desugars the surface syntax to the core
(define (desugar [sast : ArithS]) : ArithC
  (type-case ArithS sast
    [numS (n) (numSetC (list n))]
    [numSetS (ns) (numSetC ns)]
    [plusS (l r) (plusC (desugar l) (desugar r))]
    [multS (l r) (multC (desugar l) (desugar r))]))

; NOTE: values are normalized sets of numbers.

; interpret (evaluate) the given well-formed arithmetic expression AST
(define (interp [ast : ArithC]) : (listof number)
  (type-case ArithC ast
    [numSetC (ns) (normalize-set ns)]
```

```
[plusC (l r) ((map2-non-deterministic +) (interp l) (interp r))]
[multC (l r) ((map2-non-deterministic *) (interp l) (interp r))]))
```

### 2.2 Additional Pre-Posted Background

Now, assume that we have added the following to the language:

1. `(min <expr>)` evaluates to an empty non-deterministic number if `<expr>` evaluates to an empty non-deterministic number and otherwise to a non-deterministic number containing only the *smallest* number in the result of `<expr>`. So, `(min (? 2 4 6 7))` evaluates to `(list 2)`.
2. `(max <expr>)` is like `(min <expr>)` except that it evaluates to the *largest* number where `min` yields the *smallest*. So, `(max (? 2 4 6 7))` evaluates to `(list 7)`.

## 3 Environments in Indiana, USA [6 marks]

This question will include a portion of an interpreter for DrRacket-style functions using environments, as presented in Chapter 6 of the textbook. Here's the interpreter code:

```
(define-type FunDefC
  [fdC (name : symbol) (arg : symbol) (body : ExprC)])

(define-type ExprC
  [numC (n : number)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [appC (f : symbol) (a : ExprC)]
  [idC (i : symbol)])

(define-type Binding
  [bind (name : symbol) (val : number)])

(define-type-alias Env (listof Binding))
(define mt-env : Env empty)
(define extend-env : (Binding Env -> Env) cons)

; parse the given program to an ExprC
(define (parse [sexp : s-expression]) : ExprC
  ...)

; look up the id in env, giving an error if id is not present
```

```
(define (lookup [id : symbol] [env : Env]) : number
  ...)

; look up the name f in fds, giving an ERROR if f is not present
(define (lookup-fun [f : symbol] [fds : (listof FunDefC)]) : FunDefC
  ...)

; interpret ast in the given environment with the given list of functions
(define (interp [ast : ExprC] [env : Env] [fds : (listof FunDefC)]) : number
  (type-case ExprC ast
    [numC (n) n]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (f a)
      (local [(define fun (lookup-fun (fdC-name fd) fds))]
        (interp (fdC-body fun)
                (extend-env (bind (fdC-arg fun) (interp a env fds)) mt-env)
                fds))]
    ;; DO NOT MODIFY THE idC CASE!
    [idC (i) (lookup i env)]))


; run the program sexp with the given list of function definitions
(define (run [sexp : s-expression] [fds : (listof FunDefC)]) : number
  (interp (parse sexp) mt-env fds))
```