Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.827 Multithreaded Parallelism: Languages and Compilers

Problem Set 4

Problem 1 Core pH: Applicative Interpreter

The following data types are used to represent a simple functional language. Our language consists of constants (integers and booleans), variables, applications, abstractions and let-blocks. The language syntax is very similar to that of λ_C; the main difference is that we distinguish various kinds of constants, and we use a list to represent all the bindings in a let (with the understanding that the order of list elements doesn't matter).

```
data Exp = IntConst Int
         | BoolConst Bool
         | Var Ident
         | App Exp Exp
         | Func Ident Exp
         | Cond Exp Exp Exp
         | Prim PrimId [Exp]
         | Let [(Ident, Exp)] Exp
         deriving (Eq,Show)

data Ident = Varid Int  deriving (Eq,Show)
data PrimId = Add | Sub | Mul | Div | Eq | Less | Greater
              deriving (Eq,Show)
```

Your task in this problem will be to write a simple applicative-order (call-by-value) interpreter for this small language.

Part a:

You should begin by writing a version of your interpreter for the pure λ calculus. This means your interpreter need only handle the `Func`, `Var`, and `App` disjuncts in the grammar given above. You should use the applicative interpreter discussed in lecture as a guide (this means you will need to come up with some sort of representation of substitution, or carry around an environment). You should evaluate terms to Weak Head Normal Form.

Translate the following term into a data structure (this is straightforward to do by hand) and show what your interpreter returns as a result when given this input:

```
(\a b -> b a) ((\x y -> x (x y)) (\u v -> v)) (\z -> z)
```

Part b:

Now add constants, primitives, and conditionals to your interpreter. Try it out on the following example:

```
    (\ a -> if (a*3 == 9) then (\b c -> b) else (\b c -> c)) 14 False True
```

Part c:

Finally, add recursive let-bindings to your interpreter. This is the most difficult part; it's very important that you be careful about the scoping of names. The crucial test of this will be whether mutually recursive functions are actually mutually recursive. If you are using environments for evaluation (and it's nearly impossible to do let-blocks without such a notion) you might find it easiest to create an extended environment with bindings for *all* the new values in a block, and evaluate each of those bindings non-strictly in the extended environment. Try this example:

```
    let odd  = \x -> if x==1 then oddresult else even (x - 1)
        even = \x -> if x==0 then evenresult else odd (x - 1)
        oddresult  = 17 + 13 * evenresult
        evenresult = 43
    in odd 327
```

Problem 2 Implementing Queues with M-Structures

In this problem, we will define a queue abstraction using M-Structures. A queue is represented as a product type consisting of fields for `front`, `qsize`, `buflen`, `buffer`. The `front`, `qsize`, and `buffer` fields are represented as M-Structures so that they can be mutated as the queue changes.

```
data MQueue a = MQueue (MCell Int)       -- front
                       (MCell Int)       -- qsize
                       Int               -- buflen
                       (MArray Int a)    -- buffer
```

The field `buflen` contains the length of the queue buffer, `qsize` contains the number of elements in the queue, and `front` contains the offset of the next element to be dequeued. We enqueue elements by "putting" them at offset `(front + qsize)` modulo `buflen` and increasing the `qsize`, and we dequeue elements by "taking" the element at offset `front`, increasing `front` by one (modulo buflen) and decreasing `qsize` by one.

The syntax for M-Structure fields in algebraic types is different in the compiler than in the pH book. As a reminder, for the current Linux compiler, there are extensive language notes in

`/mit/6.827/phc-new/phc/docs/language`

Part a:

Write a function `makeQueue` which takes one parameter, `buflen`, and returns an empty `MQueue` with a buffer of appropriate size.

Part b:

Write a function `enqueue` which takes an `MQueue` and an object and places the object in the queue. Analyze your function carefully to make sure that there are no race conditions[1] or deadlock possibilities! Also, be sure to generate an error if an attempt is made to enqueue an element into a queue that is already full.

Test your implementation of `enqueue` by writing another function, `testEnqueue`, that performs some number of enqueue operations in parallel (that is, there should be no barriers or data dependencies between them).

Part c:

In this part, you are to write the definition of `dequeue`, which takes a queue and returns the element pointed to by the queue's `front`, increments the queue's `front` and decrements the `qsize`. `dequeue` also takes in a default value to be returned in case the queue is empty.

Write an additional function, `testDequeue`, to test your implementation of `dequeue`.

Part d:

Write a function `testEnqueueDequeue` that takes two parameters, a queue and a number n, and performs n enqueues and dequeues in parallel.

[1]A race condition exists in a program when there is the opportunity that a particular ordering of events might lead to incorrect behavior.