Principles of Programming Languages, Spring 2015
Assignment 6

Imperative Programming

Submission Instructions

1. Submit an zipped file named id1_id2.zip where id1 and id2 are the IDs of the students responsible for the submission (or id1.zip for one student in the group).
2. The contract must be included for every implemented procedure.
3. Use exact procedure and file names, as your code is tested automatically.
4. Answer the theoretical questions in the file ex6.pdf.

Imperative Programming:

1. Mutable data and local state.
2. environment based interpreter and compiler for imperative programming.

Question 1: Theoretical Questions:

1. Can the substitution model support Box type? explain.
2. Can the substitution model support `set!` primitive? explain.
3. Is the special operator `begin` useful in pure functional programming (i.e. without any kind of side effects) ?
4. Explain the difference between *object identity* to *state equality* in imperative programming. What are Racket's procedures for testing identity and state equality?
5. Explain why imperative programming is prefered over functional programming when it comes to iterations.
6. Explain how does the implementation of our imperative interpreter support mutations.

Question 2: Adding Profiling Capabilities to the Imperative Interpreter:

An important part of writing code is to understand how it behaves. In order to do so, many of the programming languages used today has a code profiling tool called *profiler*.
Java, for example, has a built in profiler which the user can use to monitor memory allocations, see the structure of the stack, count the number of times a method was invoked and more.
You can read more here http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html.

We wish to implement a very basic profiler for the imperative interpreter that will keep track of how many times a procedure was applied. To do so, we will keep a counter for each procedure and increment it each time the procedure is applied.

The interpreter needs to support the following operations:

initialize-counter:

```
; Signature: initialize-counter( Procedure )
; Type: [Scheme Procedure -> Scheme Procedure]
; Purpose: Initialize the counter for the given procedure.
```

Remarks:

- `initialize-counter` can receive both primitive and composite procedures.
- `initialize-counter` returns the procedure.
- In case that `initialize-counter` was provided anything other than procedure, it will return an error.

Examples:

```
>(derive-eval '(define a +))
'ok
>(derive-eval '(initialize-counter a))
'(primitive #<procedure:+>...)
>(derive-eval '(define sqr (initialize-counter (lambda(x) (* x x)))))
'ok
>(derive-eval '(define b 3))
'ok
>(derive-eval '(initialize-counter b))
eval-initialize-counter: expression is not a procedure: 3
```

get-counter:

```
; Signature: get-counter( Procedure )
; Type: [Scheme Procedure -> Number]
; Purpose: Return the number of times the provided procedure was applied.
```

Remarks:

- `get-counter` returns the number of times the procedure was applied.
- If `get-counter` receives as parameter a procedure which was not initialized, it will return an error.
- In case that `get-counter` was provided anything other than procedure, it will return an error.

Examples:(continue from above)

```
>(derive-eval '(+ 1 b))
4
>(derive-eval '(get-counter a))
1
>(derive-eval '(get-counter *))
eval-get-counter: procedure was not initialized: (primitive #<procedure:*>...)
```

show-all-counters:

```
; Signature: show-all-counters()
; Type: [Empty -> List(Pair(Scheme Procedure, Number))]
; Purpose: Return a list of all procedures that their counter was
; initialized. Each element in the list is a pair where the head is the
; procedure and tail is the counter for that procedure.
```

Remarks:

- Only one instance of a procedure in the list returning from `show-all-counters`.

Examples:(continue from above)

```
> (derive-eval '(show-all-counters))
'((#0=(procedure
       (x)
       ((* x x))
       (#&((sqr a car cdr cons null? + * / > < - = list box unbox set-box!)
              (#0#
              #1=(primitive #<procedure:+> #&1)
              (primitive #<procedure:car> #&#f)
              (primitive #<procedure:cdr> #&#f)
              (primitive #<procedure:cons> #&#f)
              (primitive #<procedure:null?> #&#f)
              #1#
              (primitive #<procedure:*> #&#f)
              (primitive #<procedure:/> #&#f)
              (primitive #<procedure:>> #&#f)
              (primitive #<procedure:<> #&#f)
              (primitive #<procedure:-> #&#f)
              (primitive #<procedure:=> #&#f)
              (primitive #<procedure:list> #&#f)
              (primitive #<procedure:box> #&#f)
              (primitive #<procedure:unbox> #&#f)
              (primitive #<procedure:set-box!> #&#f))))
       #&0)
   .
   0)
  (#1# . 1))
```

Instructions

We leave the implementation up to you, but keep it mind that:

- Two names of the same procedure will both increase the counter (in the example about both application of ‘a’ and ‘+’ will increase the counter), so think carefully where the state of the counter should be kept.
- You will need to use imperative programming of course.
- The `show-all-counters` procedure needs another data structure.

Question 3 mutable lists:

(During this question, you can safely assume no negative indices will be provided.)

Part 1:

Create a mutable linked list ADT, supporting the following operations:

```
; Signature: make-mlist()
; Type: [Empty -> MList]
; Purpose:   Creates an empty mutable list.

; Signature: mlist?( value )
; Type: [T -> Boolean]
; Purpose:   Validates that the passed value is indeed an mlist.
; Example:   (mlist? (make-mlist)) --> #t
; Example:   (mlist? 42) --> #f

; Signature: mlist-void?( mlist )
; Type: [MList -> Boolean]
; Purpose: Tests the passed mlist for emptyness.
; Example: (mlist-void? (make-mlist)) --> #t
; Example: (define m (make-mlist)) (mlist-add m 1) (mlist-void? m) --> #f

; Signature: mlist-add!(list, item)
; Type: MList*T -> T
; Purpose: Adds the item to the end of mlist and returns the item.
; Example (mlist-add mlist 8) -> 8

; Signature: mlist-size(mlist)
; Type: MList->Integer
; Purpose: Return the number of items in mlist.
; Example: (mlist-size (make-mlist)) --> 0

; Signature: mlist-get-index( mlist, index )
; Type: MList*Integer->T|Void
; Purpose: Return the item at the passed index (0-based), or (void), if no such
item exists.

; Signature: mlist-set-index!( mlist, index, item )
; Type: MList*Integer*T->T|Void
; Purpose: Set the item at the passed (0-based) index to the new item. Return
the item, or void, if no such item exists.

; Signature: mlist-replace-where!(mlist, predicate, new-item)
; Type: MList*[T->Boolean]*T->Void
; Purpose: Replace all items in the list on which pred returns true with
new-item.
; Example: (define m (make-mlist))
```

```
;           (mlist-add m 'secret1)
;           (mlist-add m 3)
;           (mlist-add m 'secret2)
;           (mlist-replace-where! m symbol? 'redacted)
;            --> m is now ('redacted, 3, 'redacted) (actual representation may
vary)

; Signature: mlist-delete-index(mlist, index)
; Type: MList*Integer->void
; Purpose: Remove the item at the passed (0-based) index from mlist. If no such
item exists, do not throw an error.
```

Part 2:

Implement an iterator for the list you've implemented in part 1. The iterator has to support the following operations:

```
; Signature: make-iterator(mlist)
; Type: MList->Iterator
; Purpose: Create an iterator that will go over the values in mlist.

; Signature: iterator-has-next?(itr)
; Type: Iterator->Boolean
; Purpose: Test if itr has any more values

; Signature: iterator-next!(itr)
; Type: Iterator->T
; Purpose: Advance itr to its next value, and return its current one.
```

Sample code:

```
> (define ml (make-mlist))
(mlist-add! ml 0)
> 0
(mlist-add! ml 1)
> 1
(mlist-add! ml 2)
> 2
(mlist-add! ml 3)
> 3
(define itr (make-iterator ml))
> (iterator-has-next? itr)
#t
> (iterator-next! itr)
0
> (iterator-next! itr)
1
> (iterator-next! itr)
2
> (iterator-next! itr)
```

```
3
>  (iterator-has-next? itr)
#f
```