Programming Language Concepts

CSCI-344 **Recitation 4**
Term 20161

Scheme Programming (Higher-Order Functions and Operational Semantics)

1 Introduction

You will read, reason about, and write some simple higher-order functions on lists and you will explore some of the design space and consequences of the operational semantics for μScheme.

2 Description

- Consider the following functions:

```
(val flip (lambda (f) (lambda (x y) (f y x))))

(val mysteryA (flip +))
(val mysteryB (flip >))
(val mysteryC (o not ((curry <) 0)))
(define mysteryD (xs) (foldr cons '() xs))
(define mysteryE (xs) (foldl cons '() xs))
```

  Give an intuitive description of each function.

- Write functions `curry3` and `uncurry3`, which are like `curry` and `uncurry` from the μScheme Initial Basis but for 3-argument functions (rather than 2-argument functions).

- Consider the following functions:

```
(define all? (p? xs)
  (if (null? xs)
    #t
    (if (p? (car xs))
      (all? p? (cdr xs))
      #f)))

(define all?-via-foldr (p? xs)
  (foldr (lambda (x b) (if (p? x) b #f))
         #t
         xs))

(define all?-via-foldl (p? xs)
  (foldl (lambda (x b) (if (p? x) b #f))
         #t
         xs))
```

  Are there any predicates `p?` and lists `xs` such that `(all? p? xs)`, `(all?-via-foldr p? xs)`, and `(all?-via-foldl p? xs)` would return different result values?

  Are there any predicates `p?` and lists `xs` such that `(all? p? xs)`, `(all?-via-foldr p? xs)`, and `(all?-via-foldl p? xs)` would behave differently?

  Hint: Consider features of Impcore that remain in μScheme, but are avoided in disciplined functional programming.

- Use `foldr` or `foldl` to implement `filter`. Do any of the issues raised by the previous problem apply to your implementation of `filter-via-fold`?
- Recall that every Impcore program is also a μScheme program. Write an Impcore/μScheme program the behaves differently when executed by the Impcore interpreter and when executed by the μScheme interpreter. The ideal solution will have the following properties:
  - The differences should not be simply due to the echoing of the interpreter; rather, use `print` to output a final integer result.
  - The program will run without errors when executed by the Impcore interpreter and when executed by the μScheme interpreter.
- (Adapted from Exercise 44 of Chapter 2 from *Programming Languages: Build, Prove, Compare* (p. 210).)

  Recall the DefineOldGlobal and DefineNewGlobal rules from the operational semantics of μScheme:

$$\frac{x \in \operatorname{dom}\rho \qquad \langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \mathrm{VAL}(x, e), \rho, \sigma\rangle \to \langle \rho, \sigma'\{\rho(x) \mapsto v\}\rangle} \text{ (DefineOldGlobal)}$$

$$\frac{x \notin \operatorname{dom}\rho \qquad \ell \notin \operatorname{dom}\sigma \qquad \langle e, \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto \text{unspecified}\}\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \mathrm{VAL}(x, e), \rho, \sigma\rangle \to \langle \rho\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\}\rangle} \text{ (DefineNewGlobal)}$$

  - Prof. Ramsey writes "In both Scheme and μScheme, a `val` binding of a name that is already bound is equivalent to `set`." Give an alternative (but equivalent) DefineOldGlobal rule that makes Prof. Ramsey's statement manifestly correct (i.e., one that uses SET in a premise).
  - It would be much easier if `val` always created a new binding. Give a DefineGlobal rule to specify the operational semantics of such a `val`.
  - Write a μScheme program to detect whether `val` uses the Scheme semantics or the new semantics from the previous part.
  - Compare and contrast the two ways of defining `val`. Which design do you prefer, and why?

3 Requirements and Submission

You may use the reference interpreter (see Appendix A), but there may only be one active laptop in each group.

At the end of class, submit the group's solutions either as hard-copy or by e-mail to `hh@cs.rit.edu`; be sure to include the names of all group members in the submission.

A Interpreter

A reference μScheme interpreter is available on the CS Department Linux systems (e.g., `glados.cs.rit.edu` and `queeg.cs.rit.edu` and ICLs 1 and 2) at:

```
/usr/local/pub/mtf/plc/bin/uscheme
```

Use the reference interpreter to check your code.

B μScheme Initial Basis Functions

```
(define and (b c) (if b  c  b))
(define or  (b c) (if b  b  c))
(define not (b)   (if b #f #t))

(define max (x y) (if (> x y) x y))
(define min (x y) (if (< x y) x y))

(define mod (m n) (- m (* n (/ m n))))
(define gcd (m n) (if (= n 0) m (gcd n (mod m n))))
(define lcm (m n) (if (= m 0) 0 (* m (/ n (gcd m n)))))

(define atom? (x) (or (number? x) (or (symbol? x) (or (boolean? x) (null? x)))))

(define equal? (s1 s2)
  (if (or (atom? s1) (atom? s2))
    (= s1 s2)
    (and (equal? (car s1) (car s2)) (equal? (cdr s1) (cdr s2)))))

(define length (xs)
  (if (null? xs)
    0
    (+ 1 (length (cdr xs)))))

(define append (xs ys)
  (if (null? xs)
    ys
    (cons (car xs) (append (cdr xs) ys))))

(define revapp (xs ys)
  (if (null? xs)
    ys
    (revapp (cdr xs) (cons (car xs) ys))))

(define reverse (xs) (revapp xs '()))

(define o (f g) (lambda (x) (f (g x))))
(define curry   (f) (lambda (x) (lambda (y) (f x y))))
(define uncurry (f) (lambda (x y) ((f x) y)))
(define filter (p? xs)
  (if (null? xs)
    '()
    (if (p? (car xs))
      (cons (car xs) (filter p? (cdr xs)))
      (filter p? (cdr xs)))))
(define map (f xs)
  (if (null? xs)
    '()
    (cons (f (car xs)) (map f (cdr xs)))))
(define exists? (p? xs)
  (if (null? xs)
    #f
    (if (p? (car xs))
      #t
      (exists? p? (cdr xs)))))
(define all? (p? xs)
  (if (null? xs)
    #t
    (if (p? (car xs))
      (all? p? (cdr xs))
      #f)))
(define foldr (op zero xs)
  (if (null? xs)
    zero
    (op (car xs) (foldr op zero (cdr xs)))))
(define foldl (op zero xs)
  (if (null? xs)
    zero
    (foldl op (op (car xs) zero) (cdr xs))))
```