Solutions

CMPT 383 Midterm 1, Fall 2019

Last name *exactly as it appears on student card*									
First name *exactly as it appears on student card*									
SFU Student #									
SFU email *ends with* `sfu.ca`									

This is a **closed book exam**: notes, books, computers, calculators, electronic devices, etc. are **not permitted**. Do not speak to any other students during their exam or look at their work. If you have a question, please remain seated and raise your hand and a proctor will come to you.

	Out of	***Your Mark***
Racket and EBNF	15	
You Be the Interpreter!	10	
Short Answer	10	
Total	35	

Racket and EBNF

Consider the following Racket-like language for prefix expressions involving just + and *. Both + and * have exactly 2 arguments, and numbers and symbols (as permitted by Racket's `number?` and `symbol?` functions) are considered legal expressions. Here are some example expressions:

Legal	Not Legal
`4.5`	`"4.5"`
`cat`	`(cat)`
`(+ 4 -2)`	`(+ 4)`
`(* a (* 2 2))`	`(a * (* 2 2))`
`(+ (* a b) (+ 4 mouse))`	`(+ 2 (* a b) (+ 4 mouse))`

We'll call this language **AM** for short.

a) (5 marks) Write an **EBNF grammar** that describes the syntax of **AM**. Use the Go-style EBNF notation as used in lectures and in the notes.

```
expr   =  number
        | symbol
        | "(" op expr expr ")" .
op     = "+" | "*" .

number = a valid Racket number according to number? .
symbol = a valid Racket symbol according to symbol? .
```

b) (10 marks) Write a Racket function called `(am-eval e)` that evaluates any valid **AM** expression `e`. If `e` doesn't contain any variables, then return the number that it evaluates to. If `e` contains one, or more, variables, then return the symbol `'unknown`. You can assume `e` is a valid **AM** expression.

For example:

```
> (am-eval '(+ (* 2 3) (+ 4 5)))
15

> (am-eval '(+ (* 2 3) (+ (* 4 y) 5)))
'unknown
```

You can use any standard Racket functions, and you can write helper functions if needed. In addition to correctness, the following will also be considered by the markers:

- Is the syntax correct? Are relevant language features used appropriately?
- Is there any unnecessary code?
- Is the code readable? For example, is standard Racket-style indentation and spacing used?

```
;; am-eval1 recursively evaluates e, checking for unknown values as it goes
(define (am-eval1 e)
  (cond [(number? e) e]
        [(symbol? e) 'unknown]
        [else (match e
                [`(,op ,a ,b) (let ([aval (am-eval1 a)]
                                    [bval (am-eval1 b)])
                                (if (and (number? aval)
                                         (number? bval))
                                    ((eval op) aval bval)
                                    'unknown))]
                [_ (error "invalid expression")])]))

;; in am-eval2 the standard Racket eval function is applied
;; to expressions that don't have any variables

(define (is-var? x)
  (and (symbol? x)
       (not (equal? x '+))    ;; + and * are symbols, and so
       (not (equal? x '*))))  ;; they are special cases

(define (am-eval2 e)
  (cond [(number? e) e]
        [(symbol? e) 'unknown]
        [(ormap is-var? (flatten e)) 'unknown]
        [else (eval e)]))
```

You Be the Interpreter!

(10 marks) For each of the following Racket expressions, write down on the right what that expression on the left evaluates to when it is typed into the DrRacket interpreter. If something other than a value is returned, then describe what happens using one or two words (e.g. for an error you can write "error").

Note that in the sample solutions, step-by-step derivations are given to help explain what is going on. Correct answers do not require the steps, just the final correct value.

`(- (+ 10 1 2) (* 2 1 0) 2)`	`(- (+ 10 1 2) (* 2 1 0) 2)`
	`= (- 13 0 2)`
	`= 11`

`(rest (rest (first '((1 2) (3) (4 5 6)))))`	`(rest (rest (first`
	`'((1 2) (3) (4 5 6)))))`
	`= (rest (rest '(1 2))`
	`= (rest '(2))`
	`= '()`

`(cons (list (+ 5 2)) '(a b))`	`(cons (list (+ 5 2)) '(a b))`
	`= (cons (list 7) '(a b))`
	`= (cons '(7) '(a b))`
	`= '((7) a b)`

`(cons 'a (append '(b) (cons 'c '())))`	`(cons 'a (append '(b)`
	`(cons 'c '())))`
	`= (cons 'a (append '(b) '(c)))`
	`= (cons 'a '(b c))`
	`= '(a b c)`

`(let ([a 3] [b 3]) (+ (if (< a b) 1 2)`
`(if (> a b) 3 4)))`
	`(let ([a 3] [b 3])`
	`(+ (if (< a b) 1 2)`
	`(if (> a b) 3 4)))`
	`= (+ (if (< 3 3) 1 2)`
	`(if (> 3 3) 3 4))`
	`= (+ 2 4)`
	`= 6`

`(apply map (list even? '(5 6 7 8)))`	`(apply map (list even?`
	`'(5 6 7 8)))`
	`= (apply map '(<even?-proc>`
	`'(5 6 7 8)))`
	`= (map <even?-proc> '(5 6 7 8))`
	`= '(#f #t #f #t)`

`((lambda (x) (cons x `(list ,x))) 'a)`	`((lambda (x) (cons x `(list ,x))) 'a)` `= (cons 'a `(list 'a))` `= '(a list a)`

`(apply (first '(+ *)) '(1 2 3))`	`(apply (first '(+ *)) '(1 2 3))`
	`= (apply '+ '(1 2 3))`
	`= Error, '+ is a symbol, not a function`
	
	In contrast, note that using `list` instead of a quote gives a different result:
	
	`(apply (first (list + *)) '(1 2 3))`
	`= (apply (first (list + *)) '(1 2 3))`
	`= (apply <proc+> '(1 2 3))`
	`= (<proc+> '(1 2 3))`

`(foldr append '() (map list '(a b c)))`	`(foldr append '() (map list '(a b c)))`
	`= (foldr append '() '((a) (b) (c))`
	`= (append '(a) (append '(b)`
	`(append '(c) '())))`
	`= (append '(a) (append '(b) '(c)))`
	`= (append '(a) '(b c))`
	`= '(a b c)`

`((lambda (x) (x x)) (lambda (x) (x x)))`	It helps to first make this definition:
	
	`(define F (lambda (x) (x x))`
	
	For example, calling `(F 'a)` results in error because Racket tries to evaluate `('a 'a)`, but `'a` is not a function.
	
	The question asks you to evaluate `(F F)`:
	
	`(F F)`
	`= ((lambda (x) (x x)) F)`
	`= (F F)`
	`= ((lambda (x) (x x)) F)`
	`= (F F)`
	…
	
	It expands forever, and so the expression is an infinite loop that never returns a value.
	
	`(F F)` is an interesting expression: it is an infinite loop that doesn't use loops or recursion!

Short Answer

a) (2 marks) Describe in brief, clear English the difference between calling a *function* and calling a *macro* in Racket.

A function call evaluates its arguments and passes the resulting values to the function. A macro call passes its arguments to the functions *without* first evaluating them.

b) (1 mark) Name two different standard Racket forms that are macros.

`if`, `cond`, `and`, `define`, `match`, `lambda`, `let`, etc.

c) (1 mark) Racket's macros are *hygienic*. What does that mean?

Local variables in macros are automatically renamed (e.g using `gensym`) to be a unique name that is different than any other variable in the current environment. This avoids clashes with the names of any passed-in variables.

d) Consider the following code:

```
(define x 1)
(define f (lambda (x) (g 3)))
(define g (lambda (y) (+ x y)))
```

i) (1 mark) Using *static scoping*, what does `(f 6)` evaluate to?

4

ii) (1 mark) Using *dynamic scoping*, what does `(f 6)` evaluate to?

9

e) (2 marks) Write a *continuation passing style* (*CPS*) version of Racket's `cons` function called `cons-c`.

```
(define (cons-c x y k) (k (cons x y)))
```

f) (2 marks) Implement the `(curry3 f)` function that returns the *curried* version of any 3-argument function `f`.

```
(define (curry3 f)
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (f x y z)))))
```

Blank for scratch work

Blank page for scratch work