COMPUTER SCIENCE 320 (FALL TERM, 2005)
CONCEPTS OF PROGRAMMING LANGUAGES

Solutions for Mid-Term Examination

THURSDAY, OCTOBER 20, 2005

Problem 1. ***Higher-Order Functions.***

Part (a) [4 points] Define a function `curry` that takes as input a function f of two simultaneous arguments. The expression `(((curry` f`)` x`)` y`)` should evaluate to the same value as `(`f x y`)`. For example, `(((curry +) 7) 5)` should evaluate to `12`.

```
(define (curry f) (lambda (x) (lambda (y) (f x y))))
```

Part (b) [4 points] Define a function `double` which takes as input a function f of one argument, and returns as output a function that applies f twice. For example, if `inc` is a function that adds `1` to an integer input, then `(double inc)` should be a function that adds `2`.

```
(define (double f) (lambda (x) (f (f x))))
```

For parts (c), (d) and (e), consider the procedure:

```
(define (f g) (g 2))
```

Part (c) [4 points] What value is returned by evaluating `(f square)`?

4

Part (d) [4 points] What value is returned by evaluating `(f (lambda (z) (* z (+ z 1))))`?

6

Part (e) [4 points] What happens if we evaluate `(f f)`? Explain precisely.

The function `f` takes its argument `g` and applies it to 2. So, if we evaluate `(f f)`, i.e., the argument `f` is now substituted for `g` in the expression `(g 2)`, this means that 2 (the argument of `g`) will be applied to 2. But this will cause an error, because `(2 2)` cannot be evaluated.

Problem 2. ***Recursion on Lists.*** The procedure `remove` consumes two arguments: a symbol `s` and a list of symbols `los`. It returns the list obtained from `los` by removing all occurrences of `s`, if any. For example,

`(remove 'aa '(b aa ab aa bb))` returns `(b ab bb)`.

A definition of `remove` follows:

```
(define (remove s los)
            (if (null? los)
                '()
                (if (eq? (car los) s)
                    (remove s (cdr los))
                    (cons (car los) (remove s (cdr los))))))
```

Part (a) [5 points] In the definition of `remove`, if the inner `if`'s two alternative are switched — equivalently, if the inner `if`'s test is changed to `(not (eq? (car los) s))` — what function does the resulting procedure computes? Give your answer in no more than 2 sentences.

The resulting procedure returns the list obtained from `los` by removing all occurrences of symbols *not equal to* `s`. For example, using the altered `remove`, `(remove 'aa '(b aa ab aa bb))` returns `(aa aa)`.

Part (b) [5 points] In the definition of `remove`, if the inner `if`'s second alternative `(cons (car los) (remove s (cdr los)))` is replaced by `(remove s (cdr los))`, what function does the resulting procedure computes? Give your answer in no more than 2 sentences.

The resulting procedure returns the empty list `()`.

Part (c) [10 points] Define another procedure `remove-first` by making the smallest possible change in the definition of `remove`. The function computed by `remove-first` is similar to the function computed by `remove` except that it only removes the first occurrence of `s` in `los`.
For example, `(remove-first 'aa '(b aa ab aa bb))` returns `(b ab aa bb))`.

```
(define (remove-first s los)
   (if (null?  los)
        '()
        (if (eq?  (car los) s)
             (cdr los)
             (cons (car los) (remove-first s (cdr los))))))
```

Problem 3. ***Abstract Data Types. [20 points]*** Suppose we want to define the *abstract data type* (ADT) of natural numbers (the non-negative integers), based on the following representation of the natural numbers:

`0`	represented by	`()`	(the empty list)
`1`	represented by	`(#t)`	(the list with one occurrence of `#t`)
`2`	represented by	`(#t #t)`	(the list with two occurrences of `#t`)
`3`	represented by	`(#t #t #t)`	(the list with three occurrences of `#t`)
etc.			

Part (1) [10 points] For this representation of the natural numbers, define the constant `zero`, the one-argument predicate `iszero?` (which tests whether the input argument is zero), and the one-argument procedures `succ` (which returns the successor of its argument) and `pred` (which returns the predecessor of its argument, if this argument is not zero, and is not specified to return anything, if this argument is zero):

```
(define zero '())
(define iszero?  null?)
(define succ (lambda (x) (cons #t x)))
(define pred (lambda (x) (cdr x)))
```

Part (2) [10 points] Based on these definitions we implement the two-argument procedure `plus` as follows:

```
(define  plus
         (lambda (x y)
            (if (iszero? x)
                 y
                 (succ (plus (pred x) y)))))
```

In the same style in which `plus` is defined above, give a Scheme implementation of the two-argument procedure `mult` on the natural numbers, which multiplies its two input arguments and returns the value resulting from the multiplication. You can use any of `zero`, `iszero?`, `succ` and `pred`, as well as `plus`:

```
(define mult
  (lambda (x y)
    (if (iszero?  x)
         zero
         (plus y (mult (pred x) y)))))
```

Problem 4. ***Side Effects [20 points]*** Here is the transcript of a Scheme session. Fill in the blanks saying what values are returned.

```
1 ]=> (define (last-pair x)
         (if (null? (cdr x))
             x
             (last-pair (cdr x))))
;Value: last-pair
1 ]=> (define (append! x y)
         (begin (set-cdr! (last-pair x) y)
        x))
;Value: append!
1 ]=> (define x (list 'a 'b))
;Value: x
1 ]=> (define y (list 'c 'd))
;Value: y
1 ]=> (append x y)

;Value: (a b c d)

1 ]=> (reverse x)

;Value: (b a)

1 ]=> (append! x y)

;Value: (a b c d)

1 ]=> (reverse x)

;Value: (d c b a)

1 ]=> (append! x y)

;Value: (a b c d c d c d c d c d c d c d ... forever.
```

Problem 5. ***Call-by-value versus Call-by-name.*** Consider the following program, where `bindpar` is the "parallel binding" construct, just as the `let` in Scheme:

```
;; test program written in INTEX+bind
(program (a b)
  (bindpar ( (a (* a b))
```

```
        (b (+ a b))
        (c (mod a b)) )
      (bindpar ( (a (- a b))
                 (b (div a b)) )
               (+ a b))))
```

We execute the above program on the arguments 3 and 5. Determine the number of times which each of the 5 binary operators `+`, `-`, `*`, `div` and `mod`, is performed, and also determine the final value of the program — when using:

Part (a) [10 points] Call-by-value evaluation:

`+` : 2 times.
`-`: 1 time.
`*`: 1 time.
`div`: 1 time.
`mod`: 1 times.
`final value`: 8

Part (b) [10 points] Call-by-name evaluation:

`+` : 3 times.
`-`: 1 time.
`*`: 2 times.
`div`: 1 time.
`mod`: 0 times.
`final value`: 8