HW2

Due date: 11/7

Turn in your source code for green-starred problems.

1 [Insertion sort, Lecture on Scheme, p.27]

a) Suppose that the local function insert is called with
(insert 3 '(1 2 4 5)) ⇒ (1 2 3 4 5)
Draw a diagram showing the underlying structures of the argument list (1 2 4 5) and the resulting list (1 2 3 4 5). (10%)
Hint: Beware of sharing.

b) Based on part a), what are the worst-case time and space complexities of the function isort? (10%)

2* [Metaprogram]

The pow function given in the lecture generates clumsy code: (10%)

(pow 1) ⇒ (lambda (x) (* x 1))

(pow 5) ⇒ (lambda (x) (* x (* x (* x (* x (* x 1))))))

Write a Scheme function power that does the same program specialization, except that it generates clean code:

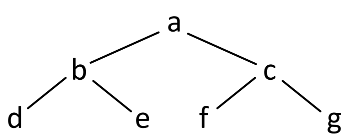
(power 1) ⇒ (lambda (x) x)

(power 5) ⇒ (lambda (x) (* x x x x x))

3* [Binary tree, Accumulator-passing style]

In Scheme, binary trees may be represented by lists – an empty list () represents an empty tree and a 3-element list (root left-subtree right-subtree) represents a non-empty tree.

For example, the binary tree



is represented by the list

(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))

Consider now the inorder traversal

```
(define inorder
   (lambda (bt)
      (if (emptytree? bt)
         '()
         (append (inorder (ltree bt)) (cons (root bt) (inorder (rtree bt)))))))
```

with the help functions

```
(define root car)
(define ltree cadr)
(define rtree caddr)
(define emptytree? null?)
```

Rewrite the function inorder in accumulator-passing style (APS) to eliminate the use of append. (10%)

Requirement

Name the function inorderAPS and define it as follows:

```
(define inorderAPS
   (lambda (bt)
      ; define here a local recursive function to traverse the binary tree bt in APS
   )
)
```

Sample run

```
>     (define tree '(a (b (c () ()) (d () ())) (e (f () ()) (g () ()))))
>     (inorderAPS tree)
      (c b d a f e g)
```

4* [Binary tree, Continuation-passing style] (10%)
Rewrite the function inorder of Problem 3 in continuation-passing style (CPS). But, this time the inorder traversal shall be abandoned in case the binary tree contains the symbol * and return the symbol bomb! immediately.

Hint

Use the function eq? to heck if two symbols are identical.

(eq? 'a '*) ⇒ #f

(eq? '* '*) ⇒ #t

Requirement

Name the function inorderCPS and define it as follows:

```
(define inorderCPS
  (lambda (bt)
    ; define here a local recursive function to traverse the binary tree bt in CPS
  )
)
```

Sample run

```
>   (define tree '(a (b (c () ()) (d () ())) (e (f () ()) (g () ()))))
>   (inorderCPS tree)
    (c b d a f e g)
>   (define bombtree '(a (b (c () ()) (d () ())) (e (* () ()) (g () ()))))
>   (inorderCPS bombtree)
    bomb!
```

5* Redo Problem 4, but this time uses call/cc. (10%)

Sample run

Let inorderCC be the function that uses call/cc

```
>   (define tree '(a (b (c () ()) (d () ())) (e (f () ()) (g () ()))))
>   (inorderCC tree)
    (c b d a f e g)
>   (define bombtree '(a (b (c () ()) (d () ())) (e (* () ()) (g () ()))))
>   (inorderCC bombtree)
    bomb!
```

6 [Fixed-point combinator]

a) Let $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ and $G = \lambda y.\lambda f.f(yf)$
Show that YG is a fixed point combinator with lazy evaluation. (10%)

b) Let $Y = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$ and $G = \lambda y.\lambda f.f(\lambda z.yfz)$
Show that YG is a fixed point combinator with eager evaluation. (10%)

c)* Use YG of part b) to redefine the named recursive function inorder of Problem 2 as an unnamed function in Scheme. (10%)

7 [SKI combinators]

Recall the SKI compilation algorithm given in the lecture:

compile x	⇒	x, if x is a variable, built-in constant, or built-in function
compile (e1 e2)	⇒	compile e1 (compile e2)
compile λx.e	⇒	abstract x (compile e)

abstract x x	⇒	I
abstract x y	⇒	K y, if y is a variable (≠ x), built-in constant, or built-in function
abstract x (e1 e2)	⇒	S (abstract x e1) (abstract x e2)

a) Compile the following λ-expression to code consisting of S, K, I. (10%)
λx.λy.yx

b) Let *exp* be the compiled code of part a), reduce the expression (10%)
exp 2 + 3

c)* Write the functions compile and abstract in Scheme. (20%)

Notes

1) To reduce the number of parentheses, ML-style notations are used in the lecture note. However, in Scheme, the compiled code must be fully parenthesized, e.g.

```
(compile '(lambda (x) ((c+ x) x)))       ⇒  ((s ((s (k c+)) i)) i)
(compile '((lambda (x) ((c+ x) x)) 2))   ⇒  (((s ((s (k c+)) i)) i) 2)
```

where c+ denotes the curried addition function defined below.
Together with the definitions of S, K, and I,

```
(define S (lambda (x) (lambda (y) (lambda (z) ((x z)(y z))))))
(define K (lambda (x) (lambda (y) x)))
(define I ((S K) K))
```

the compiled code is ready for evaluation in Scheme

```
(eval (compile '((lambda (x) ((c+ x) x)) 2)))    ⇒  4
```

2) For simpility, we assume that there are only numeric constants and two built-in functions c+ and c*.

```
(define c+ (lambda (x) (lambda (y) (+ x y))))
(define c* (lambda (x) (lambda (y) (* x y))))
```

3) To check if the expression expr is a variable (e.g. x, y), built-in constant (e.g. 2, 3), or built- in function (e.g. c+, c*), do this:
   (or (symbol? expr) (number? expr))
   N.B. Variables (e.g. x, y) and built-in functions (i.e. c+, c*) are symbols.

4) You also need the function eq? to check if two symbols are identical.
   For example, to determine if the expression expr is a λ-expression of the form (lambda (x) e), do this:
   (eq? (car expr) 'lambda)

Sample run

For testing purpose, define

```
(define compose '(lambda (f) (lambda (g) (lambda (x) (f (g x))))))
(define sq '(lambda (x) ((c* x) x)))
(define db '(lambda (x) ((c+ x) x)))
(define expr `(((,compose ,sq) ,db) 5))
```

Thus, expr is the expression that applies the composition of sq and db to 5:

```
((((lambda (f) (lambda (g) (lambda (x) (f (g x))))) (lambda (x) ((c* x) x)))
   (lambda (x) ((c+ x) x)))
  5)
```

Next, compile expr and run it:

```
(compile expr)  ⇒
(((((s
        ((s (k s))
          ((s ((s (k s)) ((s (k k)) (k s)))) ((s ((s (k s)) ((s (k k)) (k k)))) ((s (k k)) i)))))
      ((s
          ((s (k s)) ((s ((s (k s)) ((s (k k)) (k s)))) ((s ((s (k s)) ((s (k k)) (k k)))) (k i)))))
        ((s (k k)) (k i))))
    ((s ((s (k c*)) i)) i))
   ((s ((s (k c+)) i)) i))
  5)
```

```
(eval (compile expr))  ⇒  100
```