1 Interpreter in Racket

Consider the following simple language E :

```
E ::= num | E + E | E * E
```

where num is any integer. How do we represent expressions defined by this grammar in Racket? We use an integer to represent itself and a list to represent a compound expression. The first element in the list will be a symbol, either + or *, and the other two will be the operands. In other words, we can think of the grammar in terms of Racket expressions:

```
E ::= num | (+ E E) | (* E E)
```

In order to form a list in Racket without treating it as a function call, you prefix it with "*quote*". So the expression $3 + (4 * 5) + 6$ might be written as

```
(quote (+ 3 (+ (* 4 5) 6)))
```

An example of a function on E is as follows:

```
(define (desc e)
  (cond
    ((number? e) "Number")
    ((eq? (car e) '+) "Plus")
    ((eq? (car e) '*) "Times")
    (else (error "Error: Not a valid E term"))))
```

What is the result of the following function calls:

- `(desc 1)`

  Solution: "Number" □

- `(desc '(+ 1 2))`

  Solution: "Plus" □

- `(desc '(* 1 2))`

  Solution: "Times" □

- `(desc '(/ 1 2))`

  Solution: Error: Not a valid E term □

Write a function *interp* that accepts as input an expression E, interprets it, and returns the integer result. On ill-formed input (that is, input that does not conform to the grammar for E), *interp* must raise an error. For example:

```
> (interp 1)
1
> (interp '(+ 1 2))
3
> (interp '(+ (+ 1 2) 3))
6
> (interp '(+ (+ 1 2) (* 3 4)))
15
> (interp '(cons 1 2))
Error: Not a valid E term
```

(Important: You may not use eval to write *interp*.)
You may use this skeleton code to get started :

```
(define (interp e)
  (cond
    ((number? e) ________)
    ((eq? (car e) '+) ________)
    ((eq? (car e) '*) (* (interp (car (cdr e))) _______))
    (else (error "Error: Not a valid E term"))))
```

Solution:

```
(define (interp e)
  (cond
    ((number? e) e)
    ((eq? (car e) '+) (+ (interp (car (cdr e))) (interp (car (cdr (cdr e))))))
    ((eq? (car e) '*) (* (interp (car (cdr e))) (interp (car (cdr (cdr e))))))
    (else (error "Error: Not a valid E term"))))
```

□

1.1 Syntax Manipulation

We have now seen Racket's "quotation" mechanism. Consider the following example:

```
(quote (+ (+ 2 3) 1))
```

From what we have already seen, we expect the *cadr* of this expression to be (+ 2 3). Now suppose we would like to return the list (+ 5 1). That is, we want to evaluate only the *cadr* of our expression. Racket provides us a mechanism, "quasiquote" allows us to do just that:

```
(quasiquote (+ (unquote (+ 2 3)) 1))
```

That is, as intended, this will return the expression

```
`(+ 5 1).
```

Write a translator function called TR, with one argument E, such that

```
(TR E)
```

produces a new expression E' such that every occurrence of the symbol + in E is replaced with the symbol *, and every occurrence of the symbol * in E is replaced with the symbol +. For example,

```
(TR '(+ 1 2))
```

returns the expression:

```
'(* 1 2)
```

and

```
(TR '(+ (* 2 3) (* (+ 1 2) (* 8 8))))
```

produces

```
'(* (+ 2 3) (+ (* 1 2) (+ 8 8)))
```

Use Racket's quasiquote mechanism in returning the result.

You can run the result of your translator using the eval function:

```
> (eval (TR '(+ (* 2 3) (* (+ 1 2) (* 8 8)))))
90
```

Again, you may use the following skeleton :

```
(define (TR e)
  (cond
    ((number? e) ________)
    ((eq? (car e) '+) ________)
    ((eq? (car e) '*) (quasiquote (+  (______) (unquote (TR (car (cdr (cdr e))))))))
    (else (error "Error: Not a valid E term"))))
```

Solution:

```
(define (TR e)
 (cond
   ((number? e) e)
   ((eq? (car e) '+) `(* ,(TR (car (cdr e))) ,(TR (car (cdr (cdr e))))))
   ((eq? (car e) '*) `(+ ,(TR (car (cdr e))) ,(TR (car (cdr (cdr e))))))))
```

□

1.2 Map in Racket

As in most functional languages (including SML and Haskell), map is a built-in higher-order function which takes two arguments, a function F and a list L, and returns a similar list with F applied to each element in L. For example:

```
> (define (square x) (* x x))
> (define L '(1 2 3 4 5))
> (map square L)
'(1 4 9 16 25)
> (map square empty)
```

Write a *map* function that takes a function and a list and maps the function onto each of the terminal elements of that list. You may use provided skeleton.

```
> (define (square x) (* x x))
> (define L '(1 2 3 4 5))
> (map square L)
'(1 4 9 16 25)

(define (map f x)
  (cond
    ((null? x) ________)
    ((pair? x) (cons ___________))))
```

Solution:

```
(define (map f x)
  (cond
    ((null? x) empty)
    ((pair? x) (cons (f (car x)) (map f (cdr x))))))
```

□

2 Interpreter in SML

Consider the following simple language E:

```
E ::= num| E + E | E * E
```

where num is any integer. We represent terms of E by using a corresponding datatype:

```
datatype E = NUM of int | PLUS of E * E | TIMES of E * E
```

Specifically, we use the NUM constructor to represent an integer and PLUS or TIMES to represent a compound expression. In other words, we can think of the grammar in terms of SML expressions:

```
 E ::= NUM num | PLUS (E, E) | TIMES (E, E)
```

(Careful! The typeof a pair in SML is written with a * (meant to suggest the Cartesian product), but a pair value is written with a comma (much as in Python). Hence, as you can check yourself at the SML prompt, the value $(1, 2)$ has the type int * int. This is why the datatype declaration has asterisks but the example expressions have commas.)

So the expression $3 + (4 * 5) + 6$ might be written as

```
PLUS (NUM 3, PLUS (TIMES (NUM 4, NUM 5), NUM 6)
```

Write a function interp that accepts as input a program written in E, interprets it, and returns the integer result. For example:

```
- interp (NUM 1);
val it = 1 : int
- interp (PLUS (NUM 1, NUM 2));
val it = 3 : int
- interp (PLUS (PLUS (NUM 1, NUM 2), NUM 3));
val it = 6 : int
- interp (PLUS (PLUS (NUM 1, NUM 2), (TIMES (NUM 3, NUM 4))));
val it = 15 : int
```

Solution:

```
datatype E = NUM of int | PLUS of E * E | TIMES of E * E

fun interp (NUM n)            = n
  | interp (PLUS (e1, e2))   = (interp e1) + (interp e2)
  | interp (TIMES (e1, e2))  = (interp e1) * (interp e2)
```

□

3 Map on Lists and Trees

As in most functional languages (including Javascript and Haskell), map is a built-in higher-order function which takes two arguments, a function F and a list L, and returns a similar list with F applied to each element in L. For example:

```
- fun square x = x * x;
val square = fn : int -> int
- val L = [1,2,3,4,5];
val L = [1,2,3,4,5] : int list
- map square L;
val it = [1,4,9,16,25] : int list
```

1. Define map to work as expected on lists. *Solution:*

   ```
   fun map(f, l) = case l of
         []       => []
        |(x::xs) => (f x) :: map(f, xs)
   ```

   or

   ```
   fun map(f, [])     =  []
     | map(f, x::xs) = (f x) :: map(f, xs)
   ```

   □

2. Suppose we have this datatype for ML-style nested lists (i.e. trees) of integers:

   ```
   datatype tree = NIL | CONS of (tree * tree) | LEAF of int;
   ```

   Write a treemap function that takes a function and a tree and maps the function onto each of the terminal elements of that list.

```
- fun square x = x * x;
val square = fn : int -> int
- Control.Print.printDepth := 100; (* do this or the next output will be garbled *)
val it = () : unit
- val L = CONS (CONS (LEAF 1, LEAF 2), CONS (CONS (LEAF 3, LEAF 4), LEAF 5));
val L = CONS (CONS (LEAF 1,LEAF 2),CONS (CONS (LEAF 3,LEAF 4),LEAF 5)) : tree
- treemap square L;
val it = CONS (CONS (LEAF 1,LEAF 4),CONS (CONS (LEAF 9,LEAF 16),LEAF 25)) : tree
```

Solution:

```
fun treemap (F, NIL)               = NIL
  | treemap (F, (CONS (L, R)))     = CONS (treemap (F, L), treemap (F, R))
  | treemap (F, (LEAF x))          = LEAF (F x);
```

□

4 ML Reduce for Lists and Trees

- Define the reduce function in ML to work as expected on lists. *Solution:*

```
fun reduce (f, l, zero) = case l of
                   []       => zero
                 | (x::xs) => f (x, reduce (f, xs, zero))
```

or

```
fun reduce (f, [], zero)          = zero
  | reduce (f, (x::xs), zero) = f (x, reduce (f, xs, zero))
```

□

- Problem 5.5

Solution:

```
fun reduce (f, tr)
= case tr of
  LEAF i => i
| NODE (a,b) => f (reduce (f,a),reduce (f,b))
```

or

```
fun reduce (f, LEAF i)       = i
  | reduce (f, NODE (a,b)) = f (reduce (f,a),reduce (f,b))
```

□