* = tuple

```
fun interp INT of i = i
  | interp Plus e1 * e2 = interp(e1) + interp(e2)
  | interp Times e1 * e2 = interp(e1) * interp(e2)
```

CIS 425, HW 3 – SML

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 [1]:

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

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

e1 e2

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

e1 e2

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
```

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, 2, 3, 4]

x :: xs

[1] Notice how SML notation differs for product **types** as opposed to product **values**. As you can check at the SML prompt, value pairs are separated by a comma. For example: $(1, 2)$. The types of such values on the other hand are separated by *. That is the value $(1, 2)$ has the type int * int, not (int, int).

fun map f x::xs = f(x) :: map f xs *Recursive

| map f last = f(last)

[1, 2, 3, 4]

1. Define map to work as expected on lists.

fun treemap f NIL = NIL

| treemap f LEAF i = LEAF f(i)

| treemap f CONS (t_1, t_2)

= cons(treemap (f, t_1), treemap(f, t_2))

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
```

ML Reduce for Lists and Trees

- Define the reduce function in ML to work as expected on lists. Recall that the reduce function takes in a function a list and an accumulator combines all the elements of the list using the given function.

Treemap: func → tree → int

Reduce: func → list → Singular

```
- fun add x y = x + y;
val add = fn : int -> int -> int
- val L = [1, 2, 3, 4, 5];
val L = [1,2,3,4,5] : int list
- reduce add L 0;
val it = 15 : int
```

f(1, f(2, f(3, 4)))

7

9

10

[1, 2, 3, 4]

- *John C. Mitchell,* problem 5.5 (use the definition of Tree given in problem 5.4)

Reduce

fun reduce f x::xs = f(x, reduce(f, xs))

| reduce f last = last

Tree Reduce

fun treereduce f NiL = NiL

| treerduce f Leaf of i = i

| treereduce f CONS of (t_1, t_2) = f(treereduce(f, t_1), treereduce(f, t_2))