

* = tuple

fun interp INT of i = i

CIS 425, HW 3 - SML

| interp Plus $e_1 * e_2 = \text{interp}(e_1) + \text{interp}(e_2)$

Interpreter in SML

| interp Times $e_1 * e_2 = \text{interp}(e_1) * \text{interp}(e_2)$

Consider the following simple language E:

$E ::= \text{num} \mid E + E \mid E * E$

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 expressions¹:

$E ::= \text{NUM num} \mid \text{PLUS (E, E)} \mid \text{TIMES (E, E)}$

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

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

** Recursive*

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).

$\text{fun map } f \ x :: xs = f(x) :: \text{map } f \ xs$ **recursive*
 $\mid \text{map } f \ \text{last} = f(\text{last})$
 $[2, 2, 3, 4]$

1. Define map to work as expected on lists.

$\text{fun treemap } f \ \text{NIL} = \text{NIL}$

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

$\mid \text{treemap } f \ \text{LEAF } i = \text{LEAF } (f(i))$
 $\mid \text{treemap } f \ \text{CONS } (t_1, t_2)$

datatype tree = NIL | CONS of (tree * tree) | LEAF of int; $= \text{CONS}(\text{treemap } f \ t_1, \text{treemap } f \ t_2)$

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.

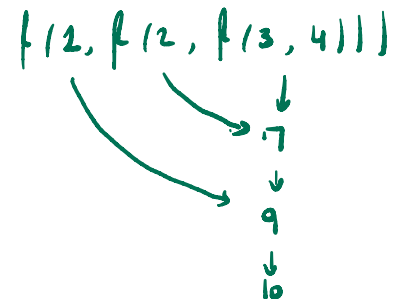
Reduce: $\text{func} \rightarrow \text{list} \rightarrow \text{singular}$

Tree map: $\text{func} \rightarrow \text{tree} \rightarrow \text{int}$

```

- 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

```



$[1, 2, 3, 4]$

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

Reduce
 $\text{fun reduce } f \ x :: xs = f(x, \text{reduce } f \ xs)$
 $\mid \text{reduce } f \ \text{last} = \text{last}$

Tree Reduce
 $\text{fun treereduce } f \ \text{NIL} = \text{NIL}$
 $\mid \text{treereduce } f \ \text{Leaf of } i = i$
 $\mid \text{treereduce } f \ \text{CONS of } (t_1, t_2) = f(\text{treereduce } f \ t_1, \text{treereduce } f \ t_2)$