Theorem: for any t ∈ exptree

eval (t) = intC(stackmc [] compile(t))

where intC : bigint → int { inv mk_big }
is a conversion function from bigints to int.

where eval, stackmc & compile functions are as specified in a1.ml.

Let's define a function 'size: exptree → int'

```
let size t = match t with
    N a      -> 1
  | fun2(a,b) -> 1 + (size a) + size b)
  | fun1(a)   -> 1 + size(a) ;;
```

Proof by induction over size.

Base Case: i. let size t = 1

⇒ t = N a for some a ∈ ℤ integers.

Now eval (N a) = a

compile (N a) = [Const mk_big (a)]

where mk_big : int → bigint is defined in a0.ml.

stackmc [] [const mk_big(a)] = mk_big(a).

intC (mk_big(a)) = a.

Induction hypothesis: let the above theorem satisfies for all t s.t. size t ≤ k

Induction step: let's take a tree t of size t = k+1

There arise 2 cases.

Property: In our implementation of stackmc [] [l1 @ l2 @ FUN]

~~= stackmc~~

let L1 = stackmc [] [l1]
L2 = stackmc [] [l2]

such that both are defined & exist.

then, stackmc [] [l1 @ l2 @ FUN] = stackmc [L2 :: L1] [FUN];

= ~~FUN(~~ FUN (L1, L2)

where FUN is ~~a binary arg operator~~ of type
: bigint x bigint → bigint.

also, if stackmc [] [l1] exists

then stackmc bl [l1 @ ls]

= stackmc l1 :: bl [ls]

This can be easily verified by marking the start of bl. ~~with a~~ ~~with some~~

Now bl = [] :: bl

and stackmc [] [l1] exists

⇒ the evaluation never pops more elements than it pushes so otherwise it would cause error.

and returns result as the head of the bigint list.

So if bl is ~~indep~~ unaffected by l1.

~~Now~~

Induction Step: let's take a tree t with size t = k+1

Here arise 2 cases.

Case 1:

$t = fun2(t1, t2)$

fun2 is of type: exptree * exptree → exptree.

also size $t = 1 + (\text{size } t1) + (\text{size } t2) = k+1$

$\Rightarrow (\text{size } t1) + (\text{size } t2) = k$

$\Rightarrow (\text{size } t1), (\text{size } t2) \leq k$

$\Rightarrow$ by induction hypothesis

eval t1 = intC (stackmc [] [Compile t1])

& eval t2 = intC (stackmc [] Compile t2)

now eval t = fun2, (eval t1, eval t2)

↑

Computes fun2 operation on eval t1 & eval t2

compile t = compile t1 @ compile t2 @ [FUN 2]

↓

FUN 2 is bigint version of fun2

now stackmc [] compile t

= stackmc [] (compile t1 @ compile t2 @ [FUN])

now stackmc [] compile t1 & stackmc [] compile t2 exist

$\Rightarrow$. It becomes FUN2 (stackmc [] compile t1, stackmc [] compile t2)

↓

Computes FUN2 on values,

for eg. let's takes fun 2 = plus.

then eval t = eval t1 + eval t2

stackmc [] compile t = stack [] compile t1 + stackmc [] compile

intC (stackmc [] compile t) = intc (stack [] compile t1)

+ intC (stackmc [] compile t2)

= eval t1 + eval t2.

Case 2: t = fun1 (t1) & size t = 1 + size t1 = k+1
∴ size t1 = k
↓ satisfies theorem

fun1 is of type : exptree → exptree

eval t = {fun1, eval t1}

↓

fun1 on int eval t1

compile t = compile t1 @ [FUN1]

=> stackmc [] compile t = stackmc [] {compile t1 @ FUN1}

= intC (stackmc [] compile t1 FUN1, stackmc [] compile

= eg. let fun1 = abs

then abs (eval t1) = intC (stackmc [] compile t1 ABS,

=> {fun1, eval t1} = intC (FUN1, stackmc [] compile t1)

Hence, Proved.