new int[12]   (array 12 Num)   (array 5 (x : Num y : Num))

new int[n*2]   (array 5 Bool)   int[] a = {1, 2, 3};   (array 1 2 3)

int x = {1,2,3}[0];

new int[]{1,2,3}

```
expr := <number> | <name> | true | false
      |  (<name> <expr> <expr>)
      |  (if <expr> <expr> <expr>)
      |  (let (<name> <expr>) <expr>)
      |  (+ <expr> <expr>)
      |  (< <expr> <expr>)
      |  (dict <name> <expr> <name> <expr>)
      |  (get <expr> <name>)
      | (array <expr> <t>)
      | (index <expr> <expr>)

def := (def <name> (<name> : <t> <name> : <t>) : <t>
         <expr>)
```

```
type expr =
  | ENum of int
  | EBool of bool
  | EId of string
  | EIf of expr * expr * expr
  | ELet of string * expr * expr
  | EPlus of expr * expr
  | ELess of expr * expr
  | EApp of string * expr * expr
  | EDict of string * expr * string * expr
  | EGet of expr * string
  | EArray of expr * typ
  | EIndex of expr * expr
```

```
→ t := Num | Bool | (<name> : <t> <name> : <t>)
     (tarr <t> ~~<expr>~~)

prog := def ... <expr>
```

```
type typ = TNum | TBool | TDict of string * typ * string * typ
             | TArr of typ

type def =
  | DFun of string * string * typ * string * typ * typ * expr

type prog = def list * expr
```

But check out:
Liquid Haskell
Dependent Types
Dafny

Runtime Error
index out of bounds

~~$\Gamma \vdash e_2 < e_3 \quad \Gamma \vdash e_2 \geq 0$~~

$$\frac{\Gamma \vdash e_1 : Num}{\Gamma \vdash (array\ e_1\ \tau) : (tarr\ \tau\ \cancel{e_1})}$$

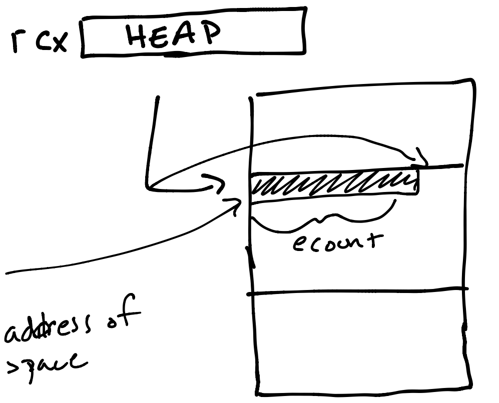
(like new $\tau[e_1]$ in Java)

$$\frac{\Gamma \vdash e_1 : (tarr\ \tau\ \cancel{e_3}) \quad \Gamma \vdash e_2 : Num}{\Gamma \vdash (index\ e_1\ e_2) : \tau}$$

(like $e_1[e_2]$ in Java)

```
section .text
global our_code_starts_here

our_code_starts_here:
    ; rdi holds heap pointer
    mov rcx, rdi

ret
```

```
int main(int argc, char** argv) {
  int64_t* HEAP = malloc(8 * 100000);

  int64_t result = our_code_starts_here( HEAP );

  print(result);
  return 0;
}
```



```
let rec e_to_is (e : expr) (si : int) (env : tenv) (defs : def list) =
  match e with
  | EArray (ecount, t) ->
    let ec_is = e_to_is ecount ... in
    ec_is @ [
      "mov rbx, rax";
      "mov rax, rcx";
      "mov [rcx], rbx";
      "mul rbx, 8";
      "add rcx, rbx";
      "add rcx, 8"; ]
  | EIndex (ea, ei) ->
    (* check index OOB *)
    let a_is = e_to_is ea ... in
    let i_is = e_to_is ei (si+1) ... in
    let store_a = ... in
    let check_oob = (check result of i_is against < 0 [underflow] / what's in memory at result from a_is)
    let lookup = (use i_is result as an offset from result a_is)
```

$\boxed{\text{rax}}$

return value

store length!

space for elts

space for length!

answer in rax is the start address of this space

clever tricks:
lea

Clean up!
- detag num
- check pos count

(ignore out-of-memory TODAY)

Mov rax, [rbx + 8*rax]

a_is result

i_is result, shifted to detag and +1 for length