Constant Folding
Constant Propagation

```
let rec c_fold (e : expr) : expr =
  match e with
      (* If there are two numbers, produce a constant! *)
    | EPrim2(Plus, ENum(n1), ENum(n2)) -> ENum(n1 + n2)
      (* Otherwise, just recur *)
    | EPrim2(Plus, e1, e2) -> EPrim2(Plus, c_fold e1, c_fold e2)

    | EIf(EBool(true), thn, els)) -> c_fold thn
    | EIf(Ebool(false), thn, els)) -> c_fold els
    | EIf(cond, thn, els) -> EIf(c_fold cond, c_fold thn, c_fold els)

    | EPrim1(IsBool, ENum(n)) -> EBool(false)


    | EPrim2(TupGet, ETuple(tup_fields), ENum(n)) ->


    ... and many more, plus recursive cases
```

Dead code elimination

generate a better, shorter expression!

IDEA:
expr_has_no_errs_or_effects

if n < List.length tup_fields and n >= 0
and List.all tup_fields is_constant then
List.nth tup_fields n

(tup-get (tup 1 2 3) 0)

(tup-get (tup 1 (print 2)) 0) [effect chng]

(t-g (tup (tup 2) (tup 3)) 1) OK (addr chage)

(t-g (tup (> true 0) 3) 1) [answer change!]

A: Answer
B: Effects
C: Both

could this change answer or effects? (w/o is-constant)

(let (x (< true 0)) OK!
  (t-g (tup x 3) 1))

```
let rec c_prop (e : expr) : expr =
  match e with
    | ELet(x, e, body) ->
      begin match e with
        | ENum(n) -> replace (c_prop body) x e

        | EBool(b) ->

        | EId(x) ->

        | _ -> ELet(y, c_prop e, c_prop body)
      end
    ... other cases just recur ...
```

(let (x 5)
  (+ x 1))

⇓

(+ 5 1)

body (ENum(n))

```
let rec replace (e : expr) (x : string) (with : expr) : expr =
  match e with
    | EId(y) ->
    | EPrim2(O, e1, e2) ->
    | ELet(y, e, body) ->
... and other recursive cases ...
```

if x = y then with else e

EPrim2(o, replace e1 x with, replace e2 x with)

if x = y then ELet(y, replace e x with, body)
else replace in both e, body

(let (x 5)
  (let (y 10)
    (let (x 7)
      x)))

Don't replace!

```
let rec improve_expr ~~first_try~~ (e : expr) : expr =
  let folded = c_fold expr in
  let propped = c_prop folded in
  ~~propped~~
  if propped = e then e
  else improve_expr propped
```

c_fold

(let (x 3)
  (+ 1 x))

↓ c_prop

(+ 1 3)

(let (x (+ 1 2))
  (+ 1 x))

⇓ improve_expr_ft

A: (let (x 3)
     (+ 1 x))

B: 4

C: (+ 1 3)

D: Something else

Why does this terminate?

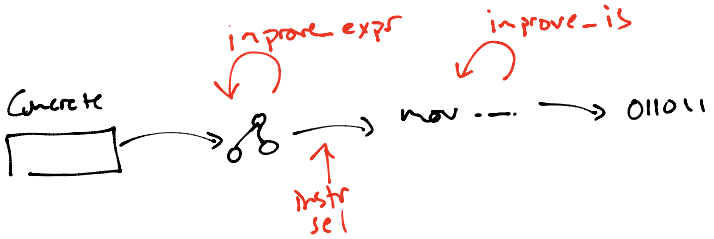
Lemma 1:
c_fold and c_prop make e smaller or leave it unchanged

Therefore, by induction on size(e),
it reaches a fixed point at some size

Case n = 1 → expr is a constant and does not change, so terminates

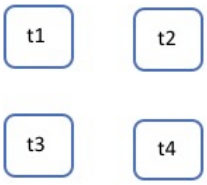
Case n > 1 — Assume terminates(e') where size(e') = n', n' < n (IH)

Case 1.1 → size(propped) = size(e) → by Lemma 1, e = propped
Case 1.2 → size(propped) < size(e) → by Inductive Hypothesis
Case 1.3 → size(propped > size(e) → Impossible by Lemma 1



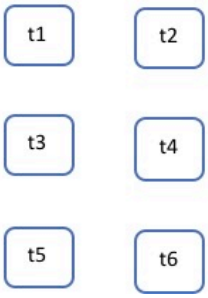
```
(def (append l1 l2)
  (if (= l1 false) l2
      (tup (tup-get l1 0) (append (tup-get l1 1) l2))))
```

explicit let var for every temporary




```
(def (append l1 l2)
  (let ((t1 (= l1 false)))
    (if t1 false
      (let ((t2 (tup-get l1 0))
            (t3 (tup-get l1 1))
            (t4 (append t3 l2)))
        (pair t2 t4)))))
```

```
(def (contains bst key)
  (let ((t1 (tup-len bst))
        (t2 (= t1 0)))
    (if t2 false
        (let ((k (tup-get bst 0))
              (t3 (= key k))
              (if t3
                  true
                  (let ((t4 (< key k)))
                    (if t4
                      (let ((t5 (tup-get bst 1)))
                        (contains t5 key))
                      (let ((t6 (tup-get bst 2)))
                        (contains t6 key)))))))))))
```



```
(def (insert bst key)
  (let ((t1 (tup-len bst 0))
        (t2 (= t1 0)))
    (if t2
        (let ((t3 (tup)) (t4 (tup))) (tup key t3 t4))
        (let ((k (tup-get bst 0))
              (t5 (= key k))
              (if t5
                  bst
                  (let ((t6 (< key k)))
                    (if t6
                        (let ((t7 (tup-get bst 1))
                              (t8 (insert t7 key))
                              (t9 (tup-get bst 2)))
                          (tup k t8 t9)))
                        (let ((t10 (tup-get bst 1))
                              (t11 (insert t10 key))
                              (t12 (tup-get bst 2)))
                          (tup k t11 t12)))))))))
```