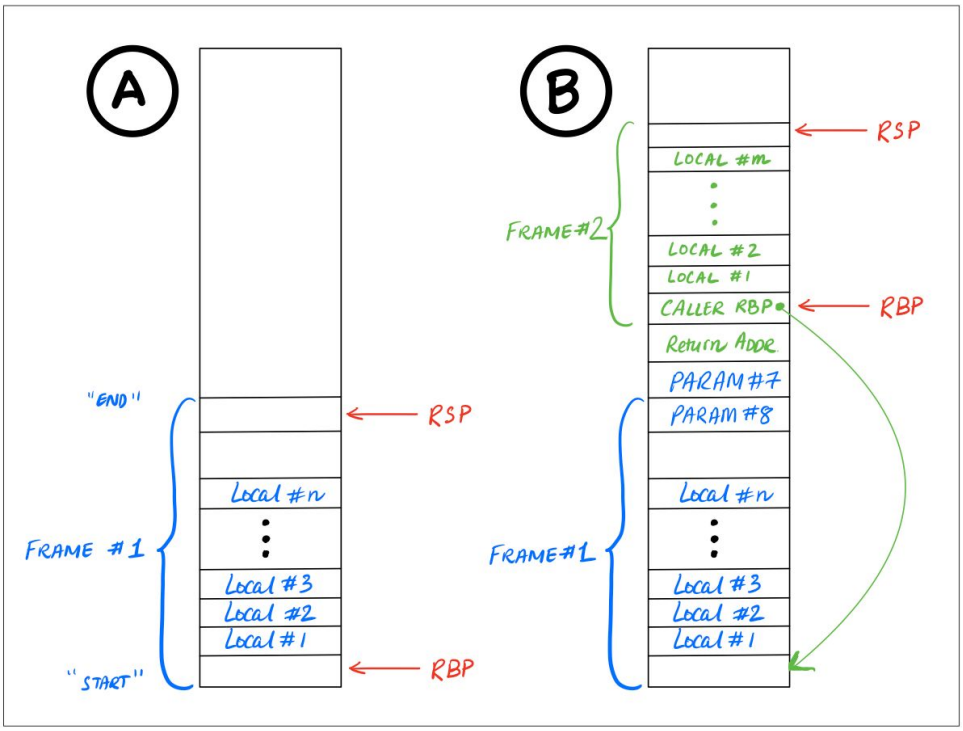
Calls: How much space for a stack frame?

```
fn compile_def_body(args: &[String], sp: usize, body: &Expr, count: &mut i32) -> String
{
  let fun_entry = compile_entry(body, sp);
  let body_code = compile_expr(body, &init_env(args), sp, count, "time_to_exit");
  let fun_exit  = compile_exit();

  format!("{fun_entry}
          {body_code}
          {fun_exit}")
}

fn compile_entry(e: &Expr, sp: usize) -> String {
  let vars = expr_vars(e) + sp;
  format!("push rbp
           mov rbp, rsp
           sub rsp, 8*{vars}")
}

fn compile_exit() -> String {
  format!("mov rsp, rbp
           pop rbp
           ret")
}
```



Calls: How much space for a stack frame?

```
fn expr_vars(e: &Expr) -> usize {
 match e {
  Expr::Num(_) | Expr::Var(_) | Expr::Input | Expr::True | Expr::False
     =>

  Expr::Add1(e) | Expr::Sub1(e) | Expr::Neg(e)  Expr::Set(_, e)
  | Expr::Loop(e) | Expr::Break(e) | Expr::Print(e) | Expr::Call1(_, e)
     =>

  Expr::Call2(_, e1, e2) | Expr::Let(_, e1, e2)
  | Expr::Eq(e1, e2) | Expr::Plus(e1, e2)
    =>

  Expr::If(e1, e2, e3)
    =>

  Expr::Block(es)
    =>

   }
}
```

Tail Calls

```
(defn (sum n acc)
  (if (= n 0)
    acc
    (sum (+ n -1) (+ acc n))))

(sum input 0)
```

```
(defn (fac n acc)
  (if (= n 0)
    acc
    (if (= n 2)
      (* 2 (fac (+ n -1) (* acc n)))
      (fac (+ n -1) (* acc n))
    )
  )
)
```

Which e can have tail call?

```
e ::= n
    | true
    | false
    | input
    | x
    | (add1 e)
    | (let (x e1) e2)
    | (+ e1 e2)
    | (= e1 e2)
    | (if e1 e2 e3)
    | (set x e)
    | (block e1...en)
    | (loop e)
    | (break e)
    | (print e)
    | (call1 e)
    | (call2 e1 e2)
```

Which calls are “tail-calls”?

```
fn compile_expr(e: &Expr,env: &Stack, sp: usize, count: &mut i32, tr: bool,...) -> String {
 match e {
   Add1(subexpr) => compile_expr(subexpr, env, sp, count, brk,                 , f) + ...,
   Plus(e1, e2)  => {
     let e1_code  = compile_expr(e1,  env,     sp, count, brk,                 , f);
     let e2_code  = compile_expr(e2,  env, sp + 1, count, brk,                 , f);
     ...
   }
   Eq(e1, e2)  => {
     let e1_code  = compile_expr(e1,  env,     sp, count, brk,                 , f);
     let e2_code  = compile_expr(e2,  env, sp + 1, count, brk,                 , f);
     ...
   }
   Let(x, e1, e2) => {
     let e1_code  = compile_expr(e1, env, sp,       count, brk,                , f);
     let e2_code  = compile_expr(e2, &newenv, sp+1, count, brk,                , f);
     ...
   }
   If(cnd, thn, els) => {
     ...
     let cnd_code = compile_expr(cnd, env, sp,      count, brk,                , f);
     let thn_code = compile_expr(thn, env, sp,      count, brk,                , f);
     let els_code = compile_expr(els, env, sp,      count, brk,                , f);
     ...
   }
   Set(x, e) => {
     let e_code   = compile_expr(e,    env, sp,     count, brk,                , f);
     ...
   }
   Block(es) => {
     let n = es.len();
     let e_codes: Vec<String> = es.iter().enumerate()
       .map(|(i, e)| compile_expr(e, env, sp, count, brk,                      , f))
       .collect();
     ...
   }
   Expr::Loop(e) => {
     ...
     let e_code = compile_expr(e, env, sp, count, &loop_exit,                 , f);
     ...
   }
   Break(e) => {
     let e_code = compile_expr(e, env, sp, count, brk,                        , f);
     ...
   }
   Print(e) => {
     let e_code = compile_expr(e, env, sp, count, brk,                        , f);
     ...
   }
   Call2(f, e1, e2) => {
     let e1_code = compile_expr(e1, env, sp,     count, brk,                  , f);
     let e2_code = compile_expr(e2, env, sp + 1, count, brk,                  , f);
     ...
   }
 }
}
```