Tail Calls

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

local vars

```
fun_start_sum:
  push rbp
  mov rbp, rsp
  sub rsp, 8*3
fun_body_sum:
  mov rax, [rbp - 8*-2]
  mov [rbp - 8*1], rax
  mov rax, 0
  cmp rax, [rbp - 8*1]
  mov rax, 1
  jne eq_exit_2
  mov rax, 3
eq_exit_2:
  cmp rax, 1
  je label_else_2
  mov rax, [rbp - 8*-3]
  jmp label_exit_2
label_else_2:
  mov rax, [rbp - 8*-2]
  mov [rbp - 8*1], rax
  mov rax, -2
  add rax, [rbp - 8*1]
  mov [rbp - 8*1], rax
  mov rax, [rbp - 8*-3]
  mov [rbp - 8*2], rax
  mov rax, [rbp - 8*-2]
  add rax, [rbp - 8*2]
```

sum 1000 0
→ sum 999 1000
→ sum 998 1000+998
⋮
→ sum 0 1000+...+1

// mov rax, n

n-1

acc
n+acc
n

① No impl
② Mutual Rec
③

foo e_1 e_2 e_3 e_4

<e_1>
mov [rbp-8·sp], rax
<e_2>
mov [rb-8·(sp+1)], rax
⋮

Call

```
push rax
mov rcx, [rbp - 8*1]
push rcx
call fun_start_sum
add rsp, 8*2
```

```
mov [rbp + 24], rax
mov rcx, [rbp - 8*1]
mov [rbp + 16], rcx
jmp fun_body_sum
```

Tail Call

```
label_exit_2:
  mov rsp, rbp
  pop rbp
  ret
```

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 f e)
    | (call2 f e1 e2)
```

block

(foo 10)

e_2

(+ 10 (let (x 5) (foo ...)))

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,      false      , f) + ...,
   Plus(e1, e2)  => {
     let e1_code  = compile_expr(e1,  env,     sp, count, brk,      false      , f);
     let e2_code  = compile_expr(e2,  env, sp + 1, count, brk,      false      , f);
     ...
   }
   Eq(e1, e2)  => {
     let e1_code  = compile_expr(e1,  env,     sp, count, brk,      false      , f);
     let e2_code  = compile_expr(e2,  env, sp + 1, count, brk,      false      , f);
     ...
   }
   Let(x, e1, e2) => {
     let e1_code  = compile_expr(e1, env, sp,       count, brk,      false      , f);
     let e2_code  = compile_expr(e2, &newenv, sp+1, count, brk,      tr         , f);
     ...
   }
   If(cnd, thn, els) => {
     ...
     let cnd_code = compile_expr(cnd, env, sp,      count, brk,      false      , f);
     let thn_code = compile_expr(thn, env, sp,      count, brk,      tr         , f);
     let els_code = compile_expr(els, env, sp,      count, brk,      tr         , f);
     ...
   }
   Set(x, e) => {
     let e_code   = compile_expr(e,    env, sp,     count, brk,      false      , 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, tr && i == n-1 , f))
       .collect();
     ...
   }
   Expr::Loop(e) => {
     ...
     let e_code = compile_expr(e, env, sp, count, &loop_exit, false      , f);
     ...
   }
   Break(e) => {
     let e_code = compile_expr(e, env, sp, count, brk,        false      , f);
     ...
   }
   Print(e) => {
     let e_code = compile_expr(e, env, sp, count, brk,        false      , f);
     ...
   }
   Call2(f, e1, e2) => {
     let e1_code = compile_expr(e1, env, sp,     count, brk,  false      , f);
     let e2_code = compile_expr(e2, env, sp + 1, count, brk,  false      , f);
     ...
   }
 }
}
```

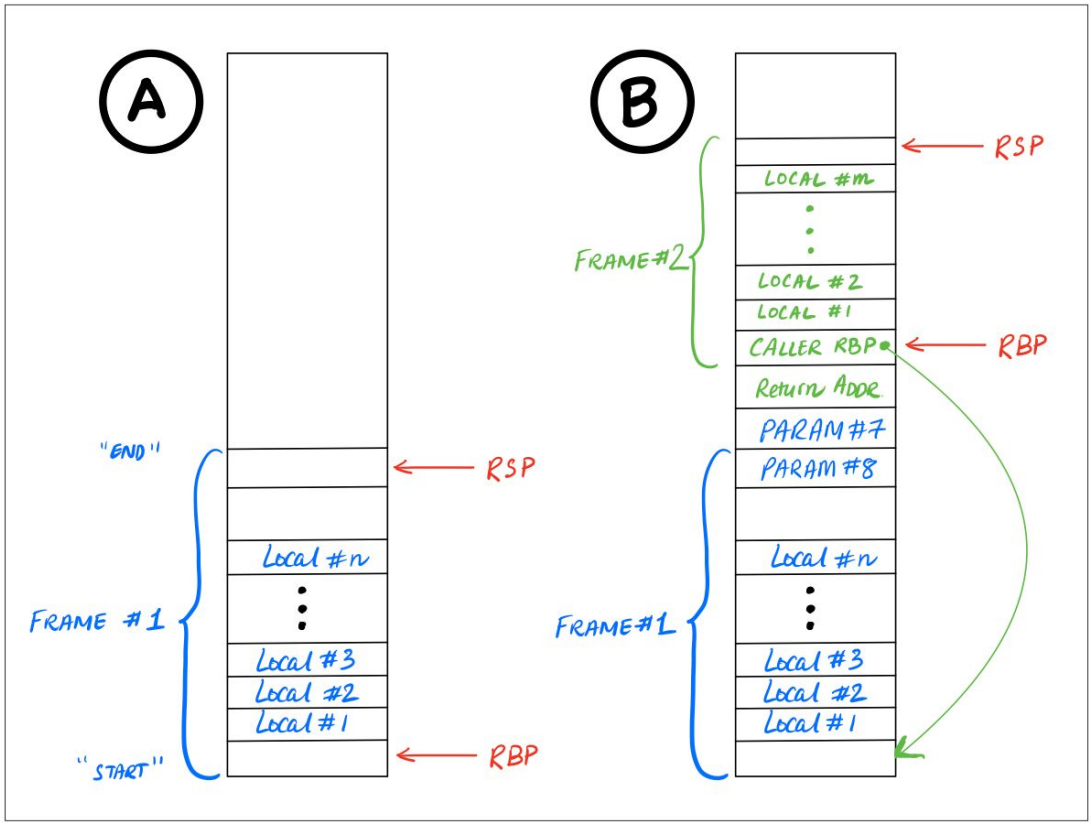
(1) false (2) true (3) tr

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")
}
```

push-pop-frame-dance

push pop - fram- undance



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
     => 0

  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_vars(e)

  Expr::Call2(_, e1, e2) | Expr::Let(_, e1, e2)
  | Expr::Eq(e1, e2) | Expr::Plus(e1, e2)
     => max( vars(e1), 1+ vars(e2) )

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

  Expr::Block(es)
     => max( vars(e1) ... vars(en) )

   }
}
```

(+ e1 e2)

(let (x1 1) (x2 2) (x3 3) ... (x100 100) x100)

(let (x1 1) (x2 2) (x3 3) ... (x100 100) x100)

Because Recycle!

Next: structured data!

```
expr := ... | (vec <expr> <expr>) | nil
            | (vec-get <expr> 0) | (vec-get <expr> 1)
```

```
(defn (head l) (vec-get l 0))

(defn (tail l) (vec-get l 1))

(defn (inc xs)
 (if (= xs nil)
   nil
   (vec (+ (head l) 1) (inc (tail l)))))

(inc (vec 10 (vec 20 nil)))
```

```
(defn (sum lst)
 (let (total 0)
   (loop
     (if (= lst nil) (break total)
       (block
         (set! total (+ total (head lst)))
         (set! lst (tail lst)))))))

(sum (vec 1 (vec 2 (vec 3 nil))))
```

```
use std::env;
#[link(name = "our_code")]
extern "C" {
    #[link_name = "\x01our_code_starts_here"]

    fn our_code_starts_here(input : i64                    ) -> i64;
}

#[no_mangle]
#[export_name = "\x01snek_print"]
fn snek_print(val : i64) -> i64 {
  if val == 3 { println!("true"); }
  else if val == 1 { println!("false"); }


  else if val % 2 == 0 { println!("{}", val >> 1); }









  else {
    println!("Unknown value: {}", val);
  }
  return val;
}

fn parse_arg(v : &Vec<String>) -> i64 {
  if v.len() < 2 { return 1 }
  let s = &v[1];
  if s == "true" { 3 }
  else if s == "false" { 1 }
  else { s.parse::<i64>().unwrap() << 1 }
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let input = parse_arg(&args);









    let i : i64 = unsafe { our_code_starts_here(input, buffer) };

    snek_print(i);
}
```

```
enum Expr {
    ...
    Vec(Box<Expr>, Box<Expr>),
    Nil,
    Get(Box<Expr>, usize)
}
```