```
mov [rsp - 8], rdi

mov rax, 5
mov [rsp - 16], rax
mov rax, [rsp - 8]
mov [rsp - 24], rax
~~and rax, [rsp - 16]~~
~~and rax, 1~~
~~cmp rax, 1~~
~~jne near error_non_int~~
mov rax, [rsp - 24]
sub rax, 1
add rax, [rsp - 16]
jo near overflow_check
mov [rsp - 16], rax
mov rax, 3
mov [rsp - 24], rax
~~and rax, [rsp - 16]~~
~~and rax, 1~~
~~cmp rax, 1~~
~~jne near error_non_int~~
mov rax, [rsp - 24]
sub rax, 1
mov [rsp - 24], rax
mov rax, [rsp - 16]
sub rax, [rsp - 24]
jo near overflow_check
```

assume/require that input is Num

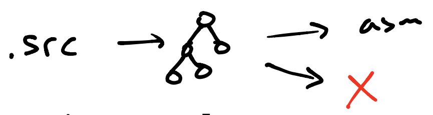
`(- (+ 2 input) 1)`

Can we get rid of all error checks in the generated code if we type-check first?

A: Yes
B: No

Can remove tag checks!
But not overflow.

Tag bits still useful for print, equals ...



Say we implement `calc_type : expr * typ_env -> typ`. Where do we use it?
What do `typ` and `typ_env` look like?

```
type typ = Num | Bool | Union typ * typ

type typ-env = (string * typ) list
```

$(+\ \overbrace{(\text{while} \ldots)}^{\text{type Bool eval to false}}\ 2)$

```
(let (x 0)
  (while (< x 10)
    (set x
      (+ x 1))))
```

```
let rec compile prog =
  let p = parse prog in        ← use calc-typ here!
  let _ ...   check p ..... in

    generate instructions
```

Variables Plan

```
(set x <expr>)
   ⇓
expr_is...
  mov [rsp-(stackloc i)], rax
```

While loops plan

```
(while cond
       body1
       body2
       body...)
```

```
start:
  COND
cmp ....
je end
  BODY
jmp start
end
```

```
start:
  BODY
  COND
cmp ....
je start
```

```
(def (g y)
  (+ y 1))

(def (f x)
  (+ (g (+ x 2)) 3))

(def (main input)
  (f (+ input 4)))
```

```
         name     arg        body
def =  strng * string  *  expr
prog = def list
```

calc_typ : expr * typ-env * def-env → typ

$$\frac{D;\ \Gamma \vdash e : \tau \qquad D[f] = (\tau, \tau_R)}{D;\ \Gamma \vdash (f\ \ e) : \underline{\tau_R}}$$

$D = \{ f : (\tau, \tau), \dots \}$

```
let compile (prog : strng) =
    let defs = parse prog in
    let defs-env = make_def-env defs
     check-defs defs defs-env

 ....

let check_def def   defs-env in
     ... FILL ...
```

```
(def (g y : Num) : Num
  (+ y 1))

(def (f x : Num) : Num
  (+ (g (+ x 2)) 3))

(def (main input : Num)
  (f (+ input 4)))
```

$$\frac{D;\ \{x : \tau_1\} \vdash e : \tau_2}{D; \{\} \vdash (\text{def}\ (f\ x : \tau_1) : \tau_2\ \ e) : (\tau_1, \tau_2)}$$

Exam Logistics

- Will post seating chart
- 45 min