```
enum Type { Num, Bool, Any, Nothing }
struct TyEnv {




}
fn calc_type(e : &Expr<()>, type_env: &TyEnv) ->                              {
    match e {
        Expr::Num(_) =>

        Expr::True =>

        Expr::Add(e1, e2) =>



        Expr::Let(x, ex, eb) =>




        Expr::Id(x) =>


        Expr::Set(x, e) =>



        Expr::Call2(f, e1, e2) =>





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





        Expr::Cast(t, e) =>




    }
}
```

```
Γ <number> : Num
```

```
Γ true : Bool
```

```
Γ input : Any
```

```
Γ (op e1 e2) : Num
  when Γ e1 ≤ Num and Γ e2 ≤ Num
  and op is +, -, *
```

```
Γ x : T
  when Γ(x) = T
```

```
Γ (let (x ex) eb) : T
  when Γ e : T1 and Γ[x : T1] e : T
```

```
Γ (set! x e) : T
  when e : T
   and Γ(x) ≤ T
```

```
Γ (f e1 e2 ...) : T
  when (fun (f (x1 : T1) (x2 : T2) ...) -> T e)
   and e1 ≤ T1, e2 ≤ T2, ...
```

```
Γ (if e1 e2 e3) : T1 ∪ T2
  when Γ e2 : T1 and Γ e3 : T2 and Γ e1 : Bool
```

```
Γ (cast T e) : T
  when Γ e : T'
```

```
enum Type { Num, Bool, Any, Nothing }
struct TyEnv {





}

fn calc_type(e : &Expr<()>, type_env: &TyEnv) ->                           {
        …
        Expr::Loop(e) =>













        Expr::Break(e) =>
















    }
}
```

```
Γ (loop e) : T1 ∪ T2 ∪ ... ∪ Tn
  when Γ1 e1 : T1, Γ2 e2 : T2, ... Γn en : Tn
   and e1, e2, ... en are (break e)
       subexpressions of e not nested in another break
   and Γ1, Γ2, ..., Γn
       are the environments for the corresponding en

Γ (break e) : Nothing
  when Γ e : T
```