f = lam <a, b, c> (x :: a, ...) body end

f <Num, String, Boolean>

t-arrow(t-var("a"), t-var("b"))

map :: (a -> b), List<a> -> List<b>

Str, Num
map<Num, Str>

data Expr:
  | e-lam(typvars :: List<String>,
          params :: List<Param>,
          body :: Expr,
          ret :: Type)
  | e-inst(e :: Expr, types :: List<Type>)

param(name, type)

data Type:
  | t-var(name :: String)
  | t-fun(args :: List<String>,
          body :: Type)

fun type-check(expr :: Expr, env :: TyEnv, types :: Set<String>):

```
| e-lam(tyvars, params, body, ret) =>
  new-tyvars = extend-vars(types, tyvars)
  new-env = extend-env(env, params)
  tb = tc(body, new-env, new-tyvars)
  when not(tb == ret): error end
t-fun(tyvars, t-arrow(params.map(get-type), ret))
```

```
fun id<a>(x :: a) -> a:
  x
end
y :: a = 10
t-fun([list: "a"],
  t-arrow(
    t-var("a"),
    t-var("a")))
```

```
| e-app(f, arg)
  ft = tc(f, env, types)
  at = tc(arg, env, types)
```

```
id(5)
id<Num>(5)
```

```
| e-inst(e, typ-args) =>
  et = tc(e, env, types)
  cases (Type) et:
    | t-fun(ty-params, tbody) => subst(tbody, typ-args, ty-params)
```

```
| e-plus(e1, e2) =>
  te1 = tc(e1 ...)
  what if tc1 is t-var("a")?
```

```
fun add1<a>(x::a)->Num
  -> x + 1
end
add<Num>(2)
add<Bool>(true)
```

Parametrically Polymorphic

```
fun map<a,b>(f::(a->b), l::List<a>) ->
                                List<b>:
```

How abstract is practical?

```
fun f<a>(x::a) -> Void:
  print(x)
end
```

Design Decision

```
fun f<a,b>(x::a, y::b) ->Bool:
  x == y
end
```

```
let id = lam(x): x end
```

```
id(1)
id("a")
```

ignored constraints (for now)

```
let id = lam<a>(x::a)->a:
           x
         end
id<Num>(1)
id<Str>("a")
```

$a \to a$

```
let id = lam<a>(x::a)->a: x end
if id<b>(1) == 1
   id<c>("a")
   id<d>(2)
```

Let-polymorphism