Type Checking
Part 2: OCaml Implementation

Speaker: David Walker
COS 326
Princeton University
Recall the OCaml Definition of Our Syntax

```
type t = IntT                  (* type int *)
  | BoolT                     (* type bool *)
  | ArrT of t * t             (* type t -> t *)

type x = string               (* variables *)
type c = Int of int | Bool of bool  (* integer and boolean constants *)
type o = Plus | Minus | LessThan (* operators *)

type e =                    (* expressions *)
  Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * t * e          (* t gives type of argument *)
  | Call of e * e
  | Let of x * e * e
```
(* abstract type of contexts *)
type ctx

(* empty context *)
val empty : ctx

(* update ctx x t: updates context ctx by binding variable x to type t *)
val update : ctx -> x -> t -> ctx

(* look ctx x: retrieves the type t associated with x in ctx
 * raises NotFound if x does not appear in ctx *)
exception NotFound
val look : ctx -> x -> t
(* const c is the type of constant c *)
let const (c : c) : t =
  match c with
  | Int i -> IntT
  | Bool b -> BoolT

(* op o = (t1, t2, t3) when o has type t1 -> t2 -> t3 *)
let op (o : o) : t =
  match o with
  | Plus -> (IntT, IntT, IntT)
  | ... 

(* use err s to signal a type error with message s *)
exception TypeError of string
let err s = raise (TypeError s)
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with
  | Const c -> const c
  | Op (e1, o, e2) ->
    let (t1, t2, t) = op o in
    (* op : t1 -> t2 -> t *)
    let t1' = check ctx e1 in
    let t2' = check ctx e2 in
    if (t1 = t1') && (t2 = t2') then
      t
    else
      err "bad argument to operator"

const(c) = t
G ⊢ c : t

optype(o) = (t1, t2, t3)
G ⊢ e1 : t1
G ⊢ e2 : t2
G ⊢ e1 o e2 : t3
let rec check (ctx : ctx) (e : e) : t =
  match e with
  | Var x ->
    begin
      try look ctx x with
      | NotFound -> err ("free variable: " ^ x)
    end
let rec check (ctx : ctx) (e : e) : t =
  match e with
  | Fun (x, t, e) ->
    check (update ctx x t) e

Notice that if we did not have the type t as a typing annotation we would not be able to make progress in our type checker at this point. We need to have a type for the variable x in our context in order to recursively check the expression e.
Function Typing

(* type check expression e in ctx, producing t *)

let rec check (ctx : ctx) (e : e) : t =
    match e with
    | Call (e1, e2) ->
        begin
            let t1 = check ctx e1 in
            match t1 with
            | ArrT (targ, tresult) ->
                let t2 = check ctx e2 in
                if targ = t2 then tresult
                else err "bad argument to function"
            | _,_ -> err "not a function in call position"
        end
Exercise: Other Rules

(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with
  | If (e1, e2, e3) -> ...
  | Let (x, e1, e2) -> ...