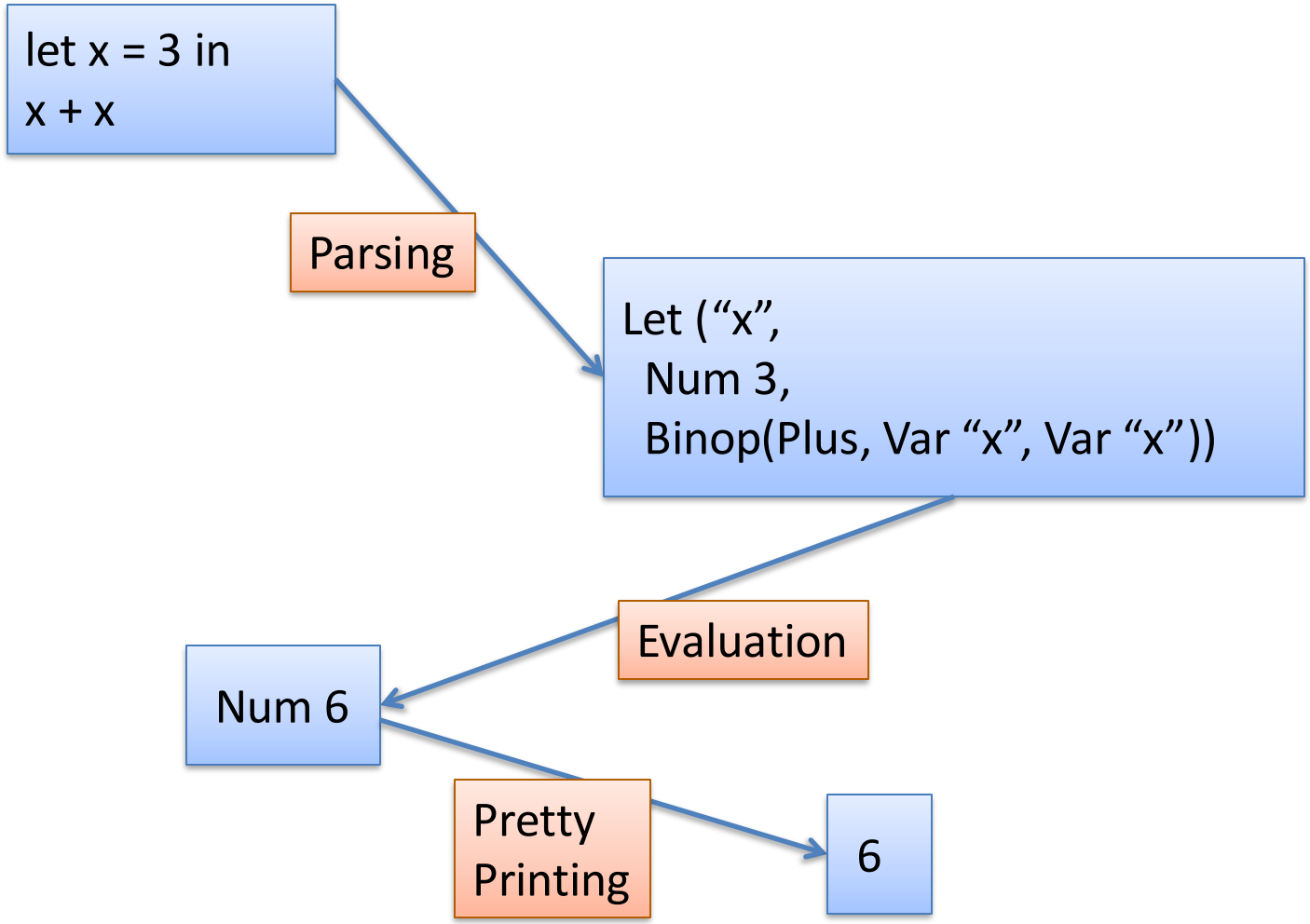
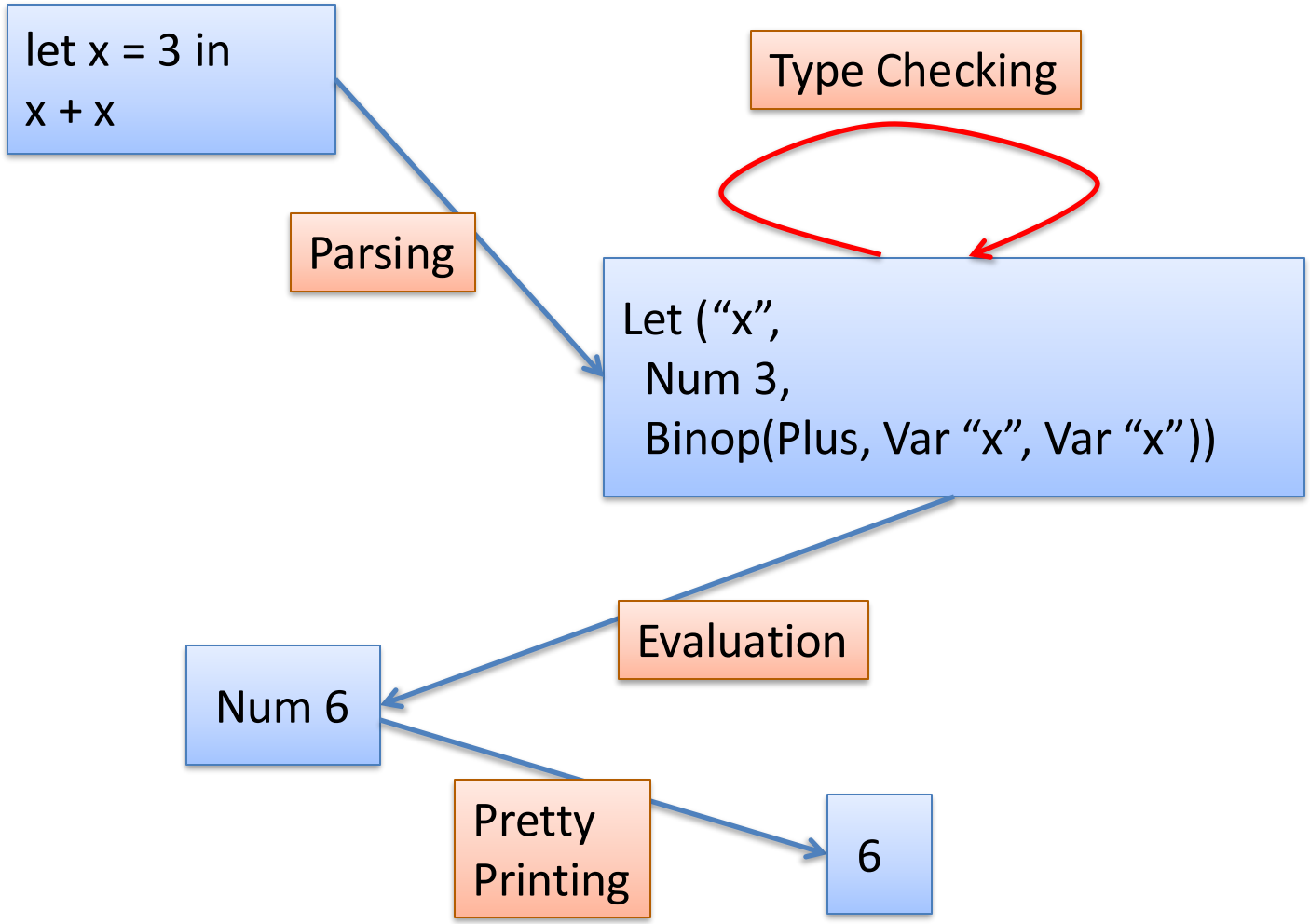
Type Checking

COS 326
David Walker
Princeton University

Implementing an Interpreter



Implementing an Interpreter



SYNTAX

Language Syntax

```
type t = IntT | BoolT | ArrT of t * t



type x = string   (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
    Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
```

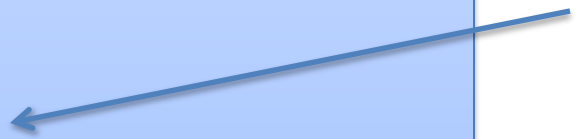
Language Syntax

```
type t = IntT | BoolT | ArrT of t * t



type x = string   (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
    Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
```



Notice that we require
a type annotation here.

We'll see why this is required
for our type checking algorithm later.

Language Syntax (BNF Definition)

```
type t = IntT | BoolT | ArrT of t * t



type x = string   (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
    Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
```

```
t ::= int | bool | t -> t

b       -- ranges over booleans
n       -- ranges over integers

x       -- ranges over variable names
c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

JUDGEMENTS AND PROOFS

Judgement

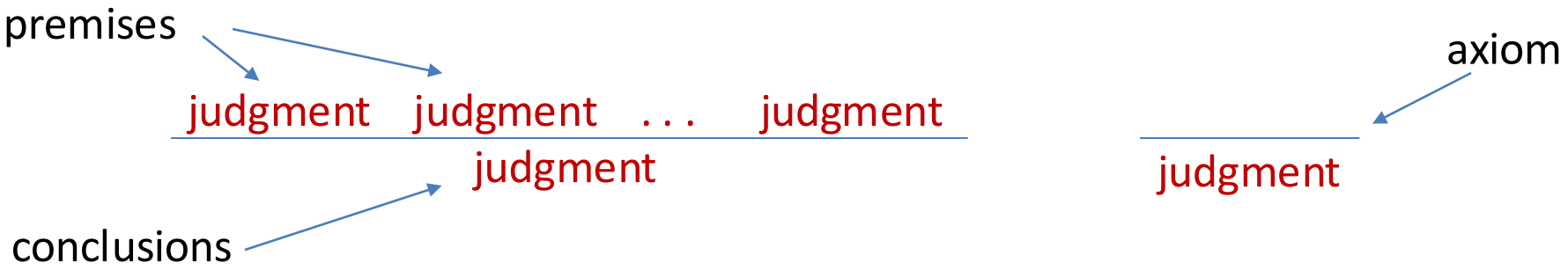
A *judgement* is a “claim” or an “assertion” or a “property” or a “relationship” – a statement that may or may not be true.

eg:	e1 --> e2	A judgement stating that expression e1 evaluates to e2 in a single execution step.
eg:	e1 ⇓ v	A judgement stating that expression e1 fully evaluates to the value v

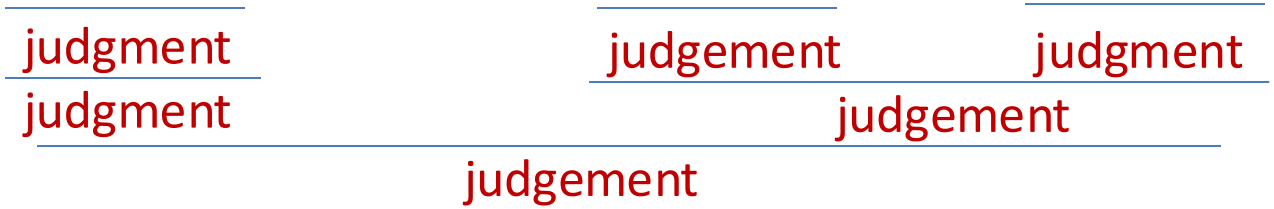
A *valid judgement* is a judgement that we have a proof of.

Recall Inference Rule Notation

An *inference rule* is a rigorous way to explain how to draw new conclusions from existing knowledge – a means of obtaining a proof for some judgement, often by using pre-existing proofs



A *proof* is a stack (a tree really) of inference rules with axioms at the top:



Recall Inference Rule Notation

An example inference rule for evaluation of function application

$$\frac{e1 \Downarrow \lambda x.e \qquad e2 \Downarrow v2 \qquad e[v2/x] \Downarrow v}{e1\ e2 \Downarrow v}$$

In English:

> "if e1 evaluates to a function with argument x and body e
> and e2 evaluates to a value v2
> and e with v2 substituted for x evaluates to v
> then e1 applied to e2 evaluates to v"

And we were also able to translate each rule into 1 case of a function in OCaml. Together all the rules formed the basis for an interpreter for the language.

TYPING RULES

The typing judgement

This notation:

$$G \vdash e : t$$

is read in English as "e has type t in context G." It is going to define how type checking works.

It describes a relation between three things – a type checking context G, an expression e, and a type t.

We are going to think of G and e as given, and we are going to compute t. The typing rules are going to tell us how.

Typing Contexts

What is the type checking context G?

Technically, I'm going to treat G as if it were a (partial) function that maps variable names to types. Notation:

G(x) -- look up x's type in G

G,x:t -- creates context G' which is the same as

G extended so that x maps to t

When G is empty, I'm just going to omit it. So I'll sometimes just write: |- e : t

Example Typing Contexts

Here's an example context:

x:int, y:bool, z:int

Think of a context as a set of "assumptions" or "hypotheses"

Read it as the assumption that "x has type int, y has type bool and z has type int"

In the subsitution model, if you assumed x has type int, that means that when you run the code, you had better actually wind up substituting an integer for x.

Typing Contexts and Free Variables

One more bit of intuition:

If an expression e contains free variables x, y, and z then we need to supply a context G that contains types for at least x, y and z. If we don't, we won't be able to type check e.

For example, this judgement:

$$|- x + y : int$$

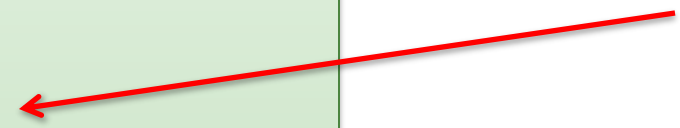
won't be a valid typing judgement because free variables x and y aren't assigned types by the context.

Type Checking Rules

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```



Goal: Give rules that define the relation "G |- e : t".

To do that, we are going to give one rule for every sort of expression.

(We can turn each rule into a case of a recursive function that takes an expression as an input and implement rules pretty directly.)

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for constant booleans:

$$\frac{}{G \vdash b : bool}$$

English:

"boolean constants b *always* have type bool, no matter what the context G is"

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for constant integers:

$$\frac{}{G \vdash n : int}$$

English:

"integer constants n *always* have type int, no matter what the context G is"

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for operators:

$$\frac{G \vdash e1 : t1 \qquad G \vdash e2 : t2 \qquad optype(o) = (t1, t2, t3)}{G \vdash e1\ o\ e2 : t3}$$

where

optype (+) = (int, int, int)
optype (-) = (int, int, int)
optype (<) = (int, int, bool)

English:

"e1 o e2 has type t3, if e1 has type t1, e2 has type t2 and o is an operator that takes arguments of type t1 and t2 and returns a value of type t3"

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for variables:

$$\frac{G(x) = t}{G \mid\!- x : t}$$

English:

"variable x has the type given by the context"

Note: this is rule explains (part) of why the context needs to provide types for all of the free variables in an expression

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for if:

$$\frac{G \vdash e1 : bool \quad G \vdash e2 : t \quad G \vdash e3 : t}{G \vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : t}$$

English:

"if e1 has type bool
and e2 has type t
and e3 has (the same) type t
then e1 then e2 else e3 has type t "

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for functions:

$$\frac{G, x{:}t \vdash e : t2}{G \vdash \lambda x{:}t.e : t \to t2}$$

English:

"if G extended with x:t proves e has type t2 then λx:t.e has type t -> t2 "

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for function call:

$$\frac{G \vdash e1 : t1 \to t2 \qquad G \vdash e2 : t1}{G \vdash e1\ e2 : t2}$$

English:

"if G extended with x:t proves e has type t2 then λx:t.e has type t -> t2 "

Typing Contexts and Free Variables

```
t ::= int | bool | t -> t

c ::= n | b
o ::= + | - | <

e ::=
  c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
```

Rule for let:

$$\frac{G \vdash e1 : t1 \qquad G,x:t1 \vdash e2 : t2}{G \vdash \text{let } x = e1 \text{ in } e2 : t2}$$

English:

"if e1 has type t1
and G extended with x:t1 proves e2 has type t2
then let x = e1 in e2 has type t2 "

A Typing Derivation

A typing derivation is a "proof" that an expression is well-typed in a particular context.

Such proofs consist of a tree of valid rules, with no obligations left unfulfilled at the top of the tree. (ie: no axioms left over).

$$G \vdash \lambda x{:}int.\ x + 2 : int \to int$$

A Typing Derivation

A typing derivation is a "proof" that an expression is well-typed in a particular context.

Such proofs consist of a tree of valid rules, with no obligations left unfulfilled at the top of the tree. (ie: no axioms left over).

$$\frac{G, x{:}int \vdash x + 2 : int}{G \vdash \lambda x{:}int.\ x + 2 : int \to int}$$

common error – the following is wrong:

$$\frac{G \vdash x : int \qquad G \vdash x + 2 : int}{G \vdash \lambda x{:}int.\ x + 2 : int \to int}$$

this rule suggests we *check* that the parameter x has the right type. That's not something we can check. It is something we *assume*.

A Typing Derivation

A typing derivation is a "proof" that an expression is well-typed in a particular context.

Such proofs consist of a tree of valid rules, with no obligations left unfulfilled at the top of the tree. (ie: no axioms left over).

$$\dfrac{\dfrac{G,x{:}int(x) = int}{G, x{:}int \vdash x : int} \qquad \dfrac{}{G,x{:}int \vdash 2 : int}}{\dfrac{G, x{:}int \vdash x + 2 : int}{G \vdash \lambda x{:}int.\ x + 2 : int \to int}}$$

Key Properties

Good type systems are *sound*.

In other words, if the type system says that e has type t then e should have "well-defined" evaluation (ie, our interpreter should not raise an exception part-way through because it doesn't know how to continue evaluation).

Also, if e has type t and it terminates and produces a value, then it should produce a value of that type. eg, if t is int, then it should produce a value with type int.

Soundness = Progress + Preservation

Proving soundness boils down to two theorems:

Progress Theorem:

If |- e : t then either:

(1) e is a value, or

(2) e --> e'

Progress says that well-typed programs are not *immediately* stuck

Preservation Theorem:

If |- e : t and e --> e' then |- e' : t

Preservations says that well-typed programs continue to be well-typed after each execution step

See COS 510 for proofs of these theorems.
But you have most of the necessary techniques:
Proof by induction on the structure of ... various inductive data types. :-)

The typing rules also define an algorithm for ... type checking ...

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
```

Signature for Context Operations

```
(* 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
```

Auxiliary Functions

```
(* 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)
```

Simple Rules

```
(* 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"
```

$$\frac{const(c) = t}{G \vdash c : t}$$

$$\frac{optype(o) = (t1, t2, t3) \quad G \vdash e1 : t1 \quad G \vdash e2 : t2}{G \vdash e1\ o\ e2 : t3}$$

Simple Rules

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

$$\frac{}{G \vdash x : G(x)}$$

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Fun (x,t,e) ->
        ArrT t (check (update ctx x t) e)
```

$$\frac{G, x{:}t \vdash e : t2}{G \vdash \lambda x{:}t.e : t \rightarrow t2}$$

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
```

$$\frac{G \vdash e1 : targ \rightarrow tresult \qquad G \vdash e2 : targ}{G \vdash e1\ e2 : tresult}$$

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) -> ...
```