Types
What do we do about errors?

In our big step semantics, we can describe situations where errors arise. But we won’t track errors, since that requires representing the program context (hard 😞).

When we hit an error, we just abandon the derivation.
What if we could catch errors early?

This program contains an error:

```
( racket ... )
( more racket ... )
( lots of racket ... )
...
(+ 1 #t)
```

But we won’t know about it until we get here!

What if we could catch the error before the program runs?
Idea: give each expression in our language a type label.

(racket ... )
(more racket ... )
(lots of racket ... )
...
(+ 1 #t)
(+ :number → number → number 1 : number #t : bool)

clash!
Types

A type is...

✦ a set of values that **share some property**
✦ a **promise to produce** a member of a certain set of values
✦ a **prediction about the value** an expression will yield
Why types?

- Help catch certain kinds of errors
- Help with documentation
- Compilers can exploit them to make code go faster
A type judgment is a rule for determining the type of an expression.

We use the following notation for a type judgment:

\[ \Gamma \vdash e : t \]

This rule is pronounced “\( \Gamma \) proves that \( e \) has type \( t \)” or “\( e \) types to \( t \) in environment \( \Gamma \)”
Typing numbers, Booleans, and addition

\[ \Gamma \vdash e : \text{number} \] \hspace{1cm} \[ \Gamma \vdash e : \text{bool} \]

\[ \Gamma \vdash e_1 : \text{number} \quad \Gamma \vdash e_2 : \text{number} \]

\[ \Gamma \vdash (+ e_1 e_2) : \text{number} \]

Credit: presentation of type-checking follows Krishnamurthi (2007)
Typing functions

Functions are a little harder.

We’ll consider only anonymous functions, since we don’t have `define` or any local binding constructs.

Also, we’ll only consider 1-parameter functions.

```
(lambda (x) e)
```

parameter  body
Typing functions

The body is the easy part. We can just recursively type-check it.

\[ \Gamma \vdash \text{lambda (x) e} : \text{tau} \]

functions have arrow types
Typing functions

But what can we do about the parameter?
We don’t know anything about it!

\[(\text{lambda} \ (x) \ e)\]
Typing functions

Solution: we have to assume that the program comes with type annotations on all function parameters.

\[
\Gamma(x \vdash \text{tau}_1) \quad \Gamma \vdash e : \text{tau}_2
\]

\[
\Gamma \vdash (\text{lambda } (x : \text{tau}_1) \ e) : \text{tau}_1 \rightarrow \text{tau}_2
\]
Typing variables

Our last kind of value is variables.

Typing variables is easy. We will assume that the environment records their type.

\[
\Gamma(e) = \text{tau} \quad \Rightarrow \quad \Gamma \vdash e : \text{tau}
\]
What about function application?
We have two expressions: the function and its argument.

\(((\text{lambda} \ (x : \text{number} \ ) \ (+ \ x \ 5)) \ 10)\)
Typing function application

What about function application?
We have two expressions: the function and its argument.

$$((\text{lambda } (x : \text{number}) (+ x 5)) \ 10)$$

We can recursively type check those:

$$\Gamma \vdash e_1: \text{tau1} \rightarrow \text{tau2} \quad \Gamma \vdash e_2: \text{tau3}$$

$$\Gamma \vdash (e_1 \ e_2): ???$$
Typing function application

We also need to make sure that the function and its argument are type-compatible!

\[ \Gamma \vdash e_1 : \text{tau1} \quad \rightarrow \quad \text{tau2} \quad \Gamma \vdash e_2 : \text{tau3} \]

\[ \Gamma \vdash (e_1 \ e_2) : \text{tau2} \quad \textit{if} \quad \text{tau1} = \text{tau3} \]
Practice:

Work on type judgment rules for number comparison:

1) =

2) equal?
Evaluating our type judgment rules

How do we evaluate our type-checking system?
Evaluating our type judgment rules

How do we evaluate our type-checking system?

Idea #1: Try out some programs
Let’s check some programs!

\[
\begin{align*}
((\text{lambda} \ (x : \text{number}) \ (+ \ x \ 5) \ 10) \\
((\text{lambda} \ (x : \text{number}) \ (+ \ x \ 5) \ #t) \\
((\text{lambda} \ (x : \text{number}) \ (+ \ #f \ 5) \ 10)
\end{align*}
\]