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:

\[
( \text{racket ... } ) \\
( \text{more racket ... } ) \\
( \text{lots of racket ... } ) \\
\ldots
\]

\[
(+ 1 \#t)
\]

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!
A type is…

- a set of values that \textit{share some property}
- a \textit{promise to produce} a member of a certain set of values
- a \textit{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
What types?

In theory we could make every value its own type. But this is equivalent to not having any types: it doesn’t help us reason about errors like (+ 1 #t): + would have to accept an infinite number of individual number types.

In practice, type systems often reflect categories of values that share implementation-independent properties.
Categorizing values by their properties

**Integers:**
- Can be added and subtracted
- Can be multiplied and divided
- Produce an integer when 1 is added to them

**Lists:**
- Can be concatenated
- Can have items appended to them
- Have a first item

These properties are implementation independent.
In **static type checking**, a program is checked for type errors before it is run.

This catches errors as early as possible.
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 \( \Gamma \)"
# A subset of Racket

## Values
- numbers
- Booleans
- functions

## Operators
- +
- -
- and
- or
- function application
- if

**Note:** we’re omitting `let`, `letrec`, and `define`
Typing numbers and Booleans

\[ \Gamma \vdash e : \text{number} \quad \Gamma \vdash e : \text{bool} \]

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

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

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

Write the type judgment for and statements.
Exercise: typing **and**

Write the type judgment for **and** statements.

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

\[
\Gamma \vdash (\text{and } e_1 \ e_2) : \text{bool}
\]
Exercise: typing and

Write the type judgment for and statements.

\[ \Gamma \vdash e_1 : boolean \quad \Gamma \vdash e_2 : bool \]

\[ \Gamma \vdash (\text{and } e_1 \ e_2) : bool \]

Is this accurate for Racket?
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.

\[(\text{lambda } (x) \; e)\]

\[\Gamma \vdash x : ?? \quad \Gamma \vdash e : \tau\]

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

functions have arrow types
Typing functions

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

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

\[\Gamma \vdash \ x:?? \quad \Gamma \vdash \ e: \tau\]

\[\Gamma \vdash (\text{lambda} \ (x) \ e):?? \rightarrow \tau\]
Typing functions

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

\[(\lambda (x:\text{tau}_1) \ e)\]

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

\[\Gamma \vdash (\lambda (x:\text{tau}_1) \ e): \text{tau}_1 \rightarrow \text{tau}_2\]
Our last kind of **value** is variables.

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

\[
\Gamma(e) = \tau \\
\Gamma \vdash e: \tau
\]
Typing function application

What about function application?

We have two expressions: the function and its argument.

```
((lambda (x) (+ x 5)) 10)
```
Typing function application

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

\[
((\text{lambda } (x) (+ x 5)) 10)
\]

We can recursively type check those:

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_3
\]

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

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

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

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