JavaScript

```
[] + []
{} + []
[] + {}
{} + {}
```

From *Wat:*

`https://www.destroyallsoftware.com/talks/wat`

Java

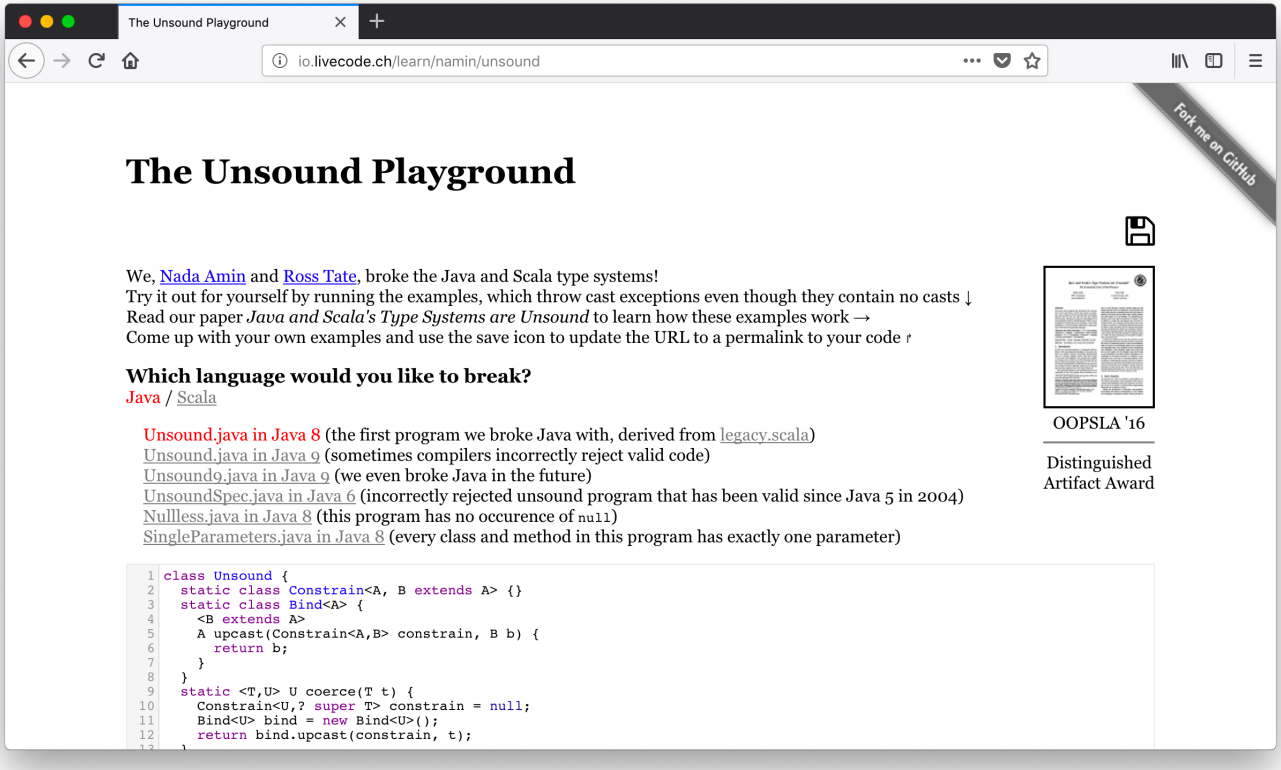
```
class A {
    static int a = B.b + 1;
}

class B {
    static int b = A.a + 1;
}
```

Python

```
a = [1], 2
a[0] += 3
```

Java and Scala



Nada Amin and Ross Tate:

`http://io.livecode.ch/learn/namin/unsound`

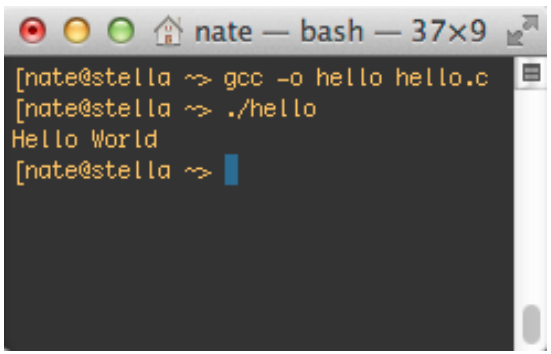
Semantics

Question: What is the meaning of a program?

Semantics

Question: What is the meaning of a program?

Answer: We could execute the program using an interpreter or a compiler, or we could consult a manual...



A6.7 Void

The (nonexistent) value of a `void` object may not be used in any way, and neither explicit nor implicit conversion to any non-void type may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only where the value is not required, for example as an expression statement (§A9.2) or as the left operand of a comma operator (§A7.18).

An expression may be converted to type `void` by a cast. For example, a void cast documents the discarding of the value of a function call used as an expression statement.

`void` did not appear in the first edition of this book, but has become common since.

...but none of these is a satisfactory solution.

A Tiny Language to Study Semantics

Functions from “math”:

$$f(x) = x + g(x) * 2$$

A Tiny Language to Study Semantics

Functions from “math”:

$$f(x) = x + g(x) * 2$$

Are like JavaScript functions that just contain a `return`:

```
function f(x) {
  return x + g(x) * 2;
}
```

A Tiny Language to Study Semantics

Functions from “math”:

$$f(x) = x + g(x) * 2$$

Are like JavaScript functions that just contain a `return`:

```
function f(x) {
  return x + g(x) * 2;
}
```

Also known as:

```
var f = (x => x + g(x) * 2);
```

A Tiny Language to Study Semantics

...but without the math, and without the names of functions.

```
x => x
```

A Tiny Language to Study Semantics

...but without the math, and without the names of functions.

```
x => x
f => x => f(x)
```

A Tiny Language to Study Semantics

...but without the math, and without the names of functions.

```
x => x
f => x => f(x)
```

a.k.a., `f => (x => (f(x)))`

A Tiny Language to Study Semantics

...but without the math, and without the names of functions.

```
x => x
f => x => f(x)
```

a.k.a., `f => (x => (f(x)))`

```
f => g => x => f(g(x))
```

Try it yourself in a JavaScript console!

The λ-calculus

JavaScript functions, but with a weird syntax:

```
x => x
```

$$\lambda x.\, x$$

The λ-calculus

JavaScript functions, but with a weird syntax:

`x => x`	$\lambda x.\, x$
`y => y + 1`	$\lambda y.\, y + 1$

The λ-calculus

JavaScript functions, but with a weird syntax:

`x => x`	$\lambda x.\, x$
`y => y + 1`	$\lambda y.\, y + 1$
`f(x)`	$f\, x$

The λ-calculus

JavaScript functions, but with a weird syntax:

`x => x`	$\lambda x.\, x$
`y => y + 1`	$\lambda y.\, y + 1$
`f(x)`	$f\, x$
`(x => x + 1)(4)`	$(\lambda x.\, x + 1)\, 4$

The λ-calculus

JavaScript functions, but with a weird syntax:

`x => x`	$\lambda x.\, x$
`y => y + 1`	$\lambda y.\, y + 1$
`f(x)`	$f\, x$
`(x => x + 1)(4)`	$(\lambda x.\, x + 1)\, 4$
`f => g => f(g(x))`	$\lambda f.\, \lambda g.\, f\, (g\, x)$

λ-terms

A λ-calculus program is always one of these three things:

- Any variable name is a λ-calculus program: for example, x, y, and z.
- If e is a λ-calculus program, then so is the *abstraction* $\lambda x.\ e$, for any variable x.
- If e_1 and e_2 are λ-calculus programs, then so is their *application* $e_1\ e_2$.

λ-terms

A λ-calculus program is always one of these three things:

- Any variable name is a λ-calculus program: for example, x, y, and z.
- If e is a λ-calculus program, then so is the *abstraction* $\lambda x.\ e$, for any variable x.
- If e_1 and e_2 are λ-calculus programs, then so is their *application* $e_1\ e_2$.

$$f \qquad g \qquad x$$

λ-terms

A λ-calculus program is always one of these three things:

- Any variable name is a λ-calculus program: for example, x, y, and z.
- If e is a λ-calculus program, then so is the *abstraction* $\lambda x.\ e$, for any variable x.
- If e_1 and e_2 are λ-calculus programs, then so is their *application* $e_1\ e_2$.

$$f \qquad g \qquad x$$

$$\lambda x.\ f \qquad \lambda f.\ x$$

λ-terms

A λ-calculus program is always one of these three things:

- Any variable name is a λ-calculus program: for example, x, y, and z.
- If e is a λ-calculus program, then so is the *abstraction* $\lambda x.\ e$, for any variable x.
- If e_1 and e_2 are λ-calculus programs, then so is their *application* $e_1\ e_2$.

$$\begin{array}{ccc} & f \quad\quad g \quad\quad x & \\ \lambda x.\ f & \lambda f.\ x & \lambda f.\ \lambda f.\ \lambda x.\ \lambda y.\ \lambda f.\ x \end{array}$$

λ-terms

A λ-calculus program is always one of these three things:

- Any variable name is a λ-calculus program: for example, x, y, and z.
- If e is a λ-calculus program, then so is the *abstraction* $\lambda x.\, e$, for any variable x.
- If e_1 and e_2 are λ-calculus programs, then so is their *application* $e_1\ e_2$.

$$f \qquad g \qquad x$$

$$\lambda x.\, f \qquad \lambda f.\, x \qquad \lambda f.\, \lambda f.\, \lambda x.\, \lambda y.\, \lambda f.\, x$$

$$\text{x} \quad \text{y} \qquad (\lambda f.\, x)\ (\lambda x.\, x)$$

A Grammar

A shorter way to define that *syntax:*

$$x \in \mathsf{Var}$$

$$e ::= x \mid e_1\ e_2 \mid \lambda x.\ e$$

Applied λ-calculi

Extend the grammar with math or something else that makes it useful:

$$x \in \text{Var}$$
$$n \in \mathbb{N}$$
$$e ::= x \mid e_1\ e_2 \mid \lambda x.\ e \mid n \mid e_1 + e_2 \mid e_1 * e_2$$

Informal Semantics

Look for a *reducible expression* and substitute the argument in for the variable in the function's body:

$$(\lambda x.\, x * 2)\ 21 \rightarrow 21 * 2$$

Informal Semantics

Look for a *reducible expression* and substitute the argument in for the variable in the function's body:

$$(\lambda x.\, x * 2)\ 21 \rightarrow 21 * 2 \rightarrow 42$$

Informal Semantics

Look for a *reducible expression* and substitute the argument in for the variable in the function's body:

$$(\lambda x.\, x * 2)\ 21 \rightarrow 21 * 2 \rightarrow 42$$

$$((\lambda x.\, x)(\lambda x.\, x + 1))\ 41 \rightarrow$$

Informal Semantics

Look for a *reducible expression* and substitute the argument in for the variable in the function's body:

$$(\lambda x.\, x * 2)\ 21 \rightarrow 21 * 2 \rightarrow 42$$

$$((\lambda x.\, x)(\lambda x.\, x + 1))\ 41 \rightarrow (\lambda x.\, x + 1)\ 41$$

Informal Semantics

Look for a *reducible expression* and substitute the argument in for the variable in the function's body:

$$(\lambda x.\, x * 2)\ 21 \to 21 * 2 \to 42$$

$$((\lambda x.\, x)(\lambda x.\, x + 1))\ 41 \to (\lambda x.\, x + 1)\ 41$$

$$((\lambda x.\, x)(\lambda x.\, x + 1))\ ((\lambda y.\, y)\ 41) \to$$

Informal Semantics

Look for a *reducible expression* and substitute the argument in for the variable in the function's body:

$$(\lambda x.\, x * 2)\ 21 \to 21 * 2 \to 42$$

$$((\lambda x.\, x)(\lambda x.\, x + 1))\ 41 \to (\lambda x.\, x + 1)\ 41$$

$$((\lambda x.\, x)(\lambda x.\, x + 1))\ ((\lambda y.\, y)\ 41) \to (\lambda x.\, x + 1)\ ((\lambda y.\, y)\ 41)$$

Substitution

Define a *value* to be a special kind of expression:

$$
\begin{aligned}
&x \in \mathrm{Var} \\
&n \in \mathbb{N} \\
&e ::= x \mid e_1\, e_2 \mid \lambda x.\, e \mid n \mid e_1 + e_2 \mid e_1 * e_2 \\
&v ::= \lambda x.\, e \mid n
\end{aligned}
$$

A redex is a function applied to a value. To evaluate it, substitute the argument for the variable in the body:

$$(\lambda x.\, e)\; v \to e\{x \mapsto v\}$$

This is a *call-by-value* semantics.

Example: Twice

Consider the function defined by $double\ x = x + x$.

Example: Twice

Consider the function defined by $double\ x = x + x$.

Now suppose we want to apply *double* multiple times:

Example: Twice

Consider the function defined by $double\, x = x + x$.

Now suppose we want to apply *double* multiple times:

$$quadruple\, x \quad = \quad double\, (double\, x)$$

Example: Twice

Consider the function defined by $double\, x = x + x$.

Now suppose we want to apply *double* multiple times:

$$\begin{aligned} quadruple\, x &= double\, (double\, x) \\ hexadecatuple\, x &= quadruple\, (quadruple\, x) \end{aligned}$$

Example: Twice

Consider the function defined by $\mathit{double}\, x = x + x$.

Now suppose we want to apply *double* multiple times:

$$\begin{aligned} \mathit{quadruple}\, x &= \mathit{double}\, (\mathit{double}\, x) \\ \mathit{hexadecatuple}\, x &= \mathit{quadruple}\, (\mathit{quadruple}\, x) \\ \mathit{256uple}\, x &= \mathit{hexadecatuple}\, (\mathit{hexadecatuple}\, x) \end{aligned}$$

Example: Twice

Consider the function defined by $\mathit{double}\, x = x + x$.

Now suppose we want to apply *double* multiple times:

$$\begin{aligned} \mathit{quadruple}\, x &= \mathit{double}\, (\mathit{double}\, x) \\ \mathit{hexadecatuple}\, x &= \mathit{quadruple}\, (\mathit{quadruple}\, x) \\ \mathit{256uple}\, x &= \mathit{hexadecatuple}\, (\mathit{hexadecatuple}\, x) \end{aligned}$$

We can abstract this pattern using a generic function:

$$\mathit{twice} \triangleq \lambda f.\, \lambda x.\, f\, (f\, x)$$

Example: Twice

Consider the function defined by $double\, x = x + x$.

Now suppose we want to apply *double* multiple times:

$$\begin{aligned} quadruple\, x &= double\, (double\, x) \\ hexadecatuple\, x &= quadruple\, (quadruple\, x) \\ 256uple\, x &= hexadecatuple\, (hexadecatuple\, x) \end{aligned}$$

We can abstract this pattern using a generic function:

$$twice \triangleq \lambda f.\, \lambda x.\, f\, (f\, x)$$

Now the functions above can be written as

$$\begin{aligned} quadruple &= twice\, double \\ hexadecatuple &= twice\, quadruple \\ 256uple &= twice\, hexadecatuple \\ & \quad (\text{or } (twice\, (\lambda x.\, twice\, x))\, double) \end{aligned}$$

Operational Semantics

We want to formalize our informal notion of the "step" arrow, $\rightarrow$.

$$(\lambda x.\, x + 2)((\lambda x.\, x * 2)\ 20) \rightarrow (\lambda x.\, x + 2)(20 * 2)$$

Operational Semantics

We want to formalize our informal notion of the "step" arrow, $\rightarrow$.

$$(\lambda x.\, x + 2)((\lambda x.\, x * 2)\ 20) \rightarrow (\lambda x.\, x + 2)(20 * 2)$$

We'll define it as a *function* from expressions to expressions. In math terms, if **Exp** is the set of all λ-calculus programs, it's a subset of the Cartesian product **Exp** $\times$ **Exp**.

Operational Semantics

We want to formalize our informal notion of the "step" arrow, $\rightarrow$.

$$(\lambda x.\, x + 2)((\lambda x.\, x * 2)\ 20) \rightarrow (\lambda x.\, x + 2)(20 * 2)$$

We'll define it as a *function* from expressions to expressions. In math terms, if **Exp** is the set of all λ-calculus programs, it's a subset of the Cartesian product **Exp** $\times$ **Exp**.

Notation: $e \rightarrow e'$
is a nicer way to write $(e, e') \in$ "$\rightarrow$"
which you might also think of as "$\rightarrow$"$(e) = e'$.

Operational Semantics

We want to formalize our informal notion of the "step" arrow, $\rightarrow$.

$$(\lambda x.\, x + 2)((\lambda x.\, x * 2)\ 20) \rightarrow (\lambda x.\, x + 2)(20 * 2)$$

We'll define it as a *function* from expressions to expressions. In math terms, if **Exp** is the set of all λ-calculus programs, it's a subset of the Cartesian product **Exp** $\times$ **Exp**.

Notation: $e \rightarrow e'$
is a nicer way to write $(e, e') \in$ "$\rightarrow$"
which you might also think of as "$\rightarrow$"$(e) = e'$.

Question: How should we define this relation? Remember that there are an infinite number of expressions and possible steps!

Inference Rules

Answer: Define it inductively, using inference rules:

$$\frac{\text{premise}_1 \qquad \text{premise}_2 \qquad \cdots}{\text{conclusion}}\ \textsc{Name}$$

Inference Rules

Answer: Define it inductively, using inference rules:

$$\frac{\text{premise}_1 \qquad \text{premise}_2 \qquad \cdots}{\text{conclusion}}\ \textsc{Name}$$

An inference rule defines an implication: if all the premises hold, then the conclusion also holds.

Formally, "$\rightarrow$" is the smallest relation that is closed under all the inference rules.

Call-by-Value λ-calculus Semantics

$$\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}\ \beta$$

Call-by-Value λ-calculus Semantics

$$\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}\ \beta$$

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2}$$

Call-by-Value λ-calculus Semantics

$$\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}\ \beta$$

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{v_1\, e_2 \to v_1\, e_2'}$$

Call-by-Value λ-calculus Semantics

$$\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}\ \beta$$

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{v_1\, e_2 \to v_1\, e_2'}$$

Key characteristics:

- Arguments evaluated fully before they are supplied to functions
- Evaluation goes from left to right
- We don't evaluate "under a λ"

Another Example Step

$$(\lambda x.\,(\lambda f.\,f)\;(\lambda y.\,y + 2)\;x)\;40 \rightarrow$$

Another Example Step

$$(\lambda x.\,(\lambda f.\,f)\ (\lambda y.\,y + 2)\ x)\ 40 \rightarrow (\lambda f.\,f)\ (\lambda y.\,y + 2)\ 40$$

Another Example Step

$$
\begin{aligned}
(\lambda x.\,(\lambda f.\,f)\ (\lambda y.\,y+2)\ x)\ 40 &\rightarrow (\lambda f.\,f)\ (\lambda y.\,y+2)\ 40 \\
&\rightarrow (\lambda y.\,y+2)\ 40 \\
&\rightarrow 40+2
\end{aligned}
$$

Rules for Math

$$\frac{m = n_1 + n_2}{n_1 + n_2 \rightarrow m}$$

Rules for Math

$$\frac{m = n_1 + n_2}{n_1 + n_2 \to m}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}$$

Rules for Math

$$\frac{m = n_1 + n_2}{n_1 + n_2 \to m}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \frac{e_2 \to e_2'}{n + e_2 \to n + e_2'}$$

Proof Tree for a Step

$$\frac{}{(\lambda x.\,(\lambda f.\,f)\ (\lambda y.\,y + 2)\ x)\ 40 \to (\lambda f.\,f)\ (\lambda y.\,y + 2)\ 40}$$

Another Step

$$\frac{}{((\lambda f.\, f)\ (\lambda y.\, y + 2))\ 40 \to (\lambda y.\, y + 2)\ 40}$$

Another Step

$$\frac{(\lambda f.\, f)\ (\lambda y.\, y + 2) \rightarrow \lambda y.\, y + 2}{((\lambda f.\, f)\ (\lambda y.\, y + 2))\ 40 \rightarrow (\lambda y.\, y + 2)\ 40}$$

Another Step

$$\frac{\dfrac{}{(\lambda f.\, f)\ (\lambda y.\, y + 2) \to \lambda y.\, y + 2}}{((\lambda f.\, f)\ (\lambda y.\, y + 2))\ 40 \to (\lambda y.\, y + 2)\ 40}$$

Chains of Steps

To “completely” evaluate a program, you probably need multiple steps. And each step needs its own proof tree.

$$\frac{}{(\lambda x.\,(\lambda f.\,f)\ (\lambda y.\,y+2)\ x)\ 40 \to (\lambda f.\,f)\ (\lambda y.\,y+2)\ 40}$$

$$\frac{\dfrac{}{(\lambda f.\,f)\ (\lambda y.\,y+2) \to \lambda y.\,y+2}}{((\lambda f.\,f)\ (\lambda y.\,y+2))\ 40 \to (\lambda y.\,y+2)\ 40}$$

$$\frac{}{(\lambda y.\,y+2)\ 40 \to 40+2}$$

A Second Look at λ-Calculus with Numbers

$$(\lambda x.\, x + 7)\ (31 + (\lambda y.\, y * 2)\ 2)$$

What could possibly go wrong?

A Second Look at λ-Calculus with Numbers

$$(\lambda x.\, x + 7)\ (31 + (\lambda y.\, y * 2)\ 2)$$

What could possibly go wrong?

$$(\lambda x.\, x + 7)\ (31 + (\lambda y.\, y * 2))$$

A Second Look at λ-Calculus with Numbers

$$(\lambda x.\, x + 7)\ (31 + (\lambda y.\, y * 2)\ 2)$$

What could possibly go wrong?

$$(\lambda x.\, x + 7)\ (31 + (\lambda y.\, y * 2)) \rightarrow ???$$

An Attempt to Rule Out Problems

We want to prevent stuff like $4 + \lambda x.\, x$. Maybe we can change the syntax...

An Attempt to Rule Out Problems

We want to prevent stuff like $4 + \lambda x.\, x$. Maybe we can change the syntax...

$$
\begin{aligned}
x &\in \mathsf{Var} \\
n &\in \mathbb{N} \\
e &::= x \mid e_1\, e_2 \mid \lambda x.\, e \mid a \\
a &::= n \mid a_1 + a_2 \mid a_1 * a_2
\end{aligned}
$$

Now, $4 + 2$ is an a expression, and $(\lambda x.\, x)\ 4$ is an e expression. But $4 + \lambda x.\, x$ is neither!

An Attempt to Rule Out Problems

We want to prevent stuff like $4 + \lambda x.\, x$. Maybe we can change the syntax...

$$
\begin{aligned}
x &\in \mathsf{Var} \\
n &\in \mathbb{N} \\
e &::= x \mid e_1\, e_2 \mid \lambda x.\, e \mid a \\
a &::= n \mid a_1 + a_2 \mid a_1 * a_2
\end{aligned}
$$

Now, $4 + 2$ is an a expression, and $(\lambda x.\, x)\ 4$ is an e expression. But $4 + \lambda x.\, x$ is neither! But what about...

$$(\lambda x.\, x + 2)\ 4$$

An Attempt to Rule Out Problems

We want to prevent stuff like $4 + \lambda x.\, x$. Maybe we can change the syntax...

$$
\begin{aligned}
x &\in \mathsf{Var} \\
n &\in \mathbb{N} \\
e &::= x \mid e_1\, e_2 \mid \lambda x.\, e \mid a \\
a &::= n \mid a_1 + a_2 \mid a_1 * a_2
\end{aligned}
$$

Now, $4 + 2$ is an a expression, and $(\lambda x.\, x)\ 4$ is an e expression. But $4 + \lambda x.\, x$ is neither! But what about...

$$(\lambda x.\, x + 2)\ 4 \qquad\qquad (\lambda f.\, f\ 2)\ (\lambda x.\, x + 4)$$

Type Systems

A *type system* is an “overlay” on an existing programming language that keeps you safe from some bad behavior.

Type Systems

A *type system* is an "overlay" on an existing programming language that keeps you safe from some bad behavior.
There are other ways to rule out bad behavior, but type systems strike a particular balance:

- They do the proof of safety for you, with minimal help from the programmer.
- Types are *compositional*.
- Type systems are usually *sound* but not *complete*.

Type Systems

A *type system* is an "overlay" on an existing programming language that keeps you safe from some bad behavior.
There are other ways to rule out bad behavior, but type systems strike a particular balance:

- They do the proof of safety for you, with minimal help from the programmer.
- Types are *compositional*.
- Type systems are usually *sound* but not *complete*.

Soundness means types will usually reject this kind of thing, even though it can never possibly cause a problem:

```
if (false) {
  2 + (x => x);
}
```

The Simply-Typed λ-Calculus

$$(\lambda x : \mathbf{int}.\, x + 2)\ 40$$

Simply-Typed Lambda Calculus

Syntax

expressions	$e ::= x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid n \mid e_1 + e_2 \mid \text{true} \mid \text{false}$
values	$v ::= \lambda x{:}\tau.\, e \mid n \mid \text{true} \mid \text{false}$
types	$\tau ::= \textbf{int} \mid \textbf{bool} \mid \tau_1 \to \tau_2$