Concrete syntax matters, actually

Slim Lim

Notion

PLAIT Lab, UC Berkeley

December 2025

Wadler's Law?

Wadler's Law (1996)

> *In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.*
>
> 0. *Semantics*
> 1. *Syntax*
> 2. *Lexical syntax*
> 3. *Lexical syntax of comments*

Wadler's Law (1996)

> *In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.*
>
> 0. *Semantics*
> 1. *Syntax*
> 2. ***Lexical syntax***
> 3. *Lexical syntax of comments*

Wadler's Law (1996)

> *In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.*

0. *Semantics*
1. *Syntax*
2. ***Lexical syntax***
3. *Lexical syntax of comments*

Also called:
concrete syntax,
surface syntax

What's the difference?

What's the difference?

- **Semantics**: Greek

$$\frac{\delta, \delta'; \pi_{m+2}(\Theta); \sigma \vdash e_i \Downarrow \theta_i \qquad \bar{e} = e_1, \ldots, e_n}{\delta, \delta'; \Theta; \sigma \vdash e(\bar{e})\ \mathrm{FAIL}}$$

- **Syntax**: Latin

```
data Expr = Let Name Expr Expr
          | Fun Name Expr
          | App Exp Expr
```

What's the difference?

- **Semantics**: Greek

$$\frac{\delta, \delta'; \pi_{m+2}(\Theta); \sigma \vdash e_i \Downarrow \theta_i \qquad \bar{e} = e_1, \ldots, e_n}{\delta, \delta'; \Theta; \sigma \vdash e(\bar{e})\ \mathrm{FAIL}}$$

- **Syntax**: Latin

```
data Expr = Let Name Expr Expr
          | Fun Name Expr
          | App Exp Expr
```

Hope this helps!

Ok but actually

Consider a simple program:

```
let n = 3 in
  max n 0
```

Semantics: example

$$\boxed{\texttt{let } x \texttt{ = 2 + 2 in } N}$$

- **Substitution**: replace occurrences of x with $2 + 2$, or 4?

$$\texttt{let } x \texttt{ = } M \texttt{ in } N \longmapsto \underbrace{N[x := M]}_{\text{substitution}}$$

- **Environment**: partial function σ maps name x to $2 + 2$, or 4?

$$\langle \sigma, \texttt{let } x \texttt{ = } M \texttt{ in } N \rangle \longmapsto \langle \underbrace{\sigma[x := M]}_{\text{extend env}}, N \rangle$$

Semantics: finding meaning

- Many different genres:
  - **Denotational** (Scott, Strachey)
  - **Axiomatic** (Hoare triples, pre- and post-conditions)
  - **Operational** (reduction rules, evaluation contexts)
  - ...

Semantics

- They make the language what it is!
- But **not the point** of this talk

Syntax: concrete vs. abstract

```
let n = 3 in
  max n 0
```

Syntax: concrete vs. abstract

```
let n = 3 in
  max n 0
```

- Local **binding**
  - **Variable** n
  - Numeric **literal** 3
  - Function **application**
    - **Function** max
    - **Variable** n
    - Numeric **literal** 0

One AST, many concrete possibilities

- Local **binding**
  - **Variable** n
  - Numeric **literal** 3
  - Function **application**
    - **Function** max
    - **Variable** n
    - Numeric **literal** 0

```
let n = 3 in
  max n 0
```

```
(let ([n 3])
  (max n 0))
```

```
const n = 3;
max(n, 3)
```

...*many* concrete possibilities

```
my $n = 3;
max($n, 3)
```

```
max n 0
  where n = 3
```

```
With[{n = 3},
  Max[n, 0]
]
```

```
(let ([n 3])
  (max n 0))
```

```
n ← 3
n ⌈ 0
```

```
const n = 3;
max(n, 3)
```

See also The Next 700 Programming Languages (Landin 1966)

Concrete syntax: examples

- **Keyword naming**
  - `const`, `let`, `var`, `my`
- **Sigils/operators**
  - `x = v`, `x := v`, `x ← v`
- **Block demarcation**
  - `{...}`
  - `BEGIN...END`
  - Significant indentation
  - Parens

and more...

Wadler's Law, again

> *In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.*
>
> 0. *Semantics*
> 1. *Syntax*
> 2. *Lexical syntax*
> 3. *Lexical syntax of comments*

Wadler's Law, again

> *In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.*
>
> 0. *Semantics*
> 1. *Syntax*
> 2. *Lexical syntax*
> 3. *Lexical syntax of comments*

Implication: priority is **backwards**

i.e. concrete syntax **exponentially less important**

We often dismiss concrete syntax

- Sometimes it's explicit: **"it's *just* syntax"**
  - Others absorb and uncritically repeat this view
- Even when we care, we might **unconsciously devalue** it
  - "Apologies for bikeshedding, but…"

Problems

Too often, concrete syntax decisionmaking is:

1. **Idiosyncratic**
2. **Under-documented**
3. **Under-researched**

And that is a shame!

Why does this matter?

Concrete syntax is the **foremost user interface** for most programming languages.

Why does this matter?

Concrete syntax is the **foremost user interface** for most programming languages.

also software libraries, mathematical theories (notation), etc.

Programming languages have user interfaces

Not a new idea

> *[m]illions for compilers, but* ***hardly a penny for understanding human programming language use.*** *Now, programming languages are obviously symmetrical, the computer on one side, the human on the other. In an* ***appropriate science of computer languages****, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are* ***easy or productive to use****.*

John Pane (1985), via Newell and Card, via Felleisen, emphasis mine

Not a new idea

- Notation as a tool of thought (Iverson 1979)
- Cognitive Dimensions of Notations (Green 1989)
- Human Language Interface (Felleisen 2003)
- Good Ideas, Through the Looking Glass (Wirth 2006)

But little research on syntax itself

- Graphical user interfaces (GUIs)
  - **Visual languages** (Alice, Scratch, Max/MSP, Quartz Composer)
  - **GUI environments, IDE extensions** for textual programming languages
- Error messages

But little research on syntax itself

- Graphical user interfaces (GUIs)
  - **Visual languages** (Alice, Scratch, Max/MSP, Quartz Composer)
  - **GUI environments, IDE extensions** for textual programming languages
- Error messages

Both important, but **metatextual**!

What about everything else?

Expanding our concept of UI

```
my $n = 3;
max($n, 3)
```

```
With[{n = 3},
  Max[n, 0]
]
```

```
n ← 3
n ⌈ 0
```

```
let n = 3 in
  max n 0
```

```
(let ([n 3])
  (max n 0))
```

```
const n = 3;
max(n, 3)
```

Expanding our concept of UI

- Text is here to stay
- Choosing concrete syntax is unavoidable
- **Legitimize thinking about the program itself as UI**

Expanding our concept of UI

> *By **relieving the brain** of all unnecessary work, a good notation sets it free to **concentrate on more advanced problems**, and in effect increases the mental power of the race.*

A. N. Whitehead, emphasis mine

Our focus

- **Textual**, **general-purpose** programming languages
- For **experienced users** (not exclusively, but at least)
- **Focus on the language itself**, not the programming environment
    - Basic affordances: syntax highlighting, LSP

Caveats

- GUIs **no less valid**, just relatively better-studied!
- **Hard to decouple**: system = notation + environment (Green)

> *If we have function keys to generate syntactic constructions, for example, **which is the 'notation'**—the **keys we press**, or the **words we see**? Various factors will determine the user's view, such as **prior experience**; the **units operated upon by the editor** [...] and whether a simple mapping can be perceived between [...] a **function key** [...] generating a simple **indivisible unit** of a few words.*

T.R.G. Green (1989), emphasis mine

Syntax mediates semantics

Potentially counterintuitive

Syntactic

$\vdash \varphi$

"can be proved"

Semantic

$\models \varphi$

"is modeled by"

Example: Propositional logic

The following definition is **purely syntactic**:

$$\begin{aligned} \varphi := \ & \dots \\ & \mid \varphi_1 \wedge \varphi_2 \\ & \mid \varphi_1 \to \varphi_2 \\ & \mid \dots \end{aligned}$$

Example: Propositional logic

The following definition is **purely syntactic**:

$$\begin{aligned}\varphi &:= \dots \\ &\mid \varphi_1 \wedge \varphi_2 \\ &\mid \varphi_1 \to \varphi_2 \\ &\mid \dots\end{aligned}$$

"Just symbols," but if I went onto **define implication using** $\wedge$, you would probably hate me

What does this operator mean?

```
x >>= y
```

What does this operator mean?

- **Functional programmers**: monadic bind

```
m >>= f >>= g >>= h
```

- **C-style programmers**: bitwise right shift assignment

$$[\![\texttt{x >>= 1}]\!] \approx [\![\texttt{x = x >> 1}]\!]$$

$\approx$ *assuming* **x** *is scalar variable, no side effects, etc.*

- **JavaScript programmers**: maybe no idea?

Human factors broach the divide

Despite best intentions, we are **pareidolic** creatures

- Prior background (both depth and nature)
- Other syntactic choices

Question

If symbol perception is all relative, **why design concrete syntax**?

Syntax mediates semantics

> *It has become fashionable to regard notation as a secondary issue depending* ***purely on personal taste****. This could partly be true; yet the choice of notation* ***should not be considered arbitrary****. It has consequences and* ***reveals the language's character****.*

Niklaus Wirth (2006), emphasis mine

Tales from the other side

1. Different kinds of sugar
2. Symbols and whitespace
3. Names matter

Two kinds of syntactic sugar

Syntactic sugar

- Coined by Landin in 1964
- Formalized through **macro extensibility** by Felleisen (1991)
- Describes **syntactic niceties** (e.g. `let`-binding) built on top of a smaller **core language** (e.g. applicative expressions)

```
let x = v in …
```

```
(\x -> …) v
```

Example 1: Definitions in Scheme

Top-level `define`

```
(define x 1)
(define y (+ 4 x))

> (+ x y)
6
```

Functions are just bound `lambdas`

```
(define x 1)
(define y (+ 4 x))
(define f (lambda (n) (* n 2)))

› (f (+ x y))
12
```

Sussman form shorthand

```
(define f (lambda (n) …))
```

becomes

```
(define (f n) …)
```

Sussman form shorthand

```
(define f (lambda (n) (* n 2)))
```

becomes

```
(define (f n) (* n 2))
```

Sussman form shorthand

```
(define fact
  (lambda (n)
    (if (zero? n) 1
        (* n (fact (- n 1))))))
```

becomes

```
(define (fact n)
  (if (zero? n) 1
      (* n (fact (- n 1)))))
```

Sussman form

- **Terser**, less nesting
- **Visually distinguishes** top-level function bindings
- But **hides the simplicity** of first-class functions
    - Easier to understand recursion without the sugar

Example: JavaScript inheritance

Example: JavaScript inheritance

- Pre-2015: **prototypal inheritance**

```
function Parent() {}
function Child() { Parent.call(this) }
Child.prototype = Object.create(Parent.prototype)
Child.prototype.constructor = Child
```

- Post-2015: **class-based inheritance**

```
class Parent {}
class Child extends Parent {
  constructor() { super() }
}
```

ES2015 `class` syntax

- Mostly engine-level syntactic sugar over prototypes
- **Intentionally obscures** prototypal semantics, **redirecting mental model** to classes
  - Don't need to understand prototypes to use `class`—you're often better off without!

A tale of two sugars

“Mystifies” functions for visual efficiency

```
(define (fact n)
  (if (zero? n) 1
      (* n
         (fact (- n 1)))))
```

Redirects mental model to classes

```
class Parent {}
class Child extends
          Parent {
  constructor() {
    super()
  }
}
```

A tale of two sugars

Both **obscure the core language** (syntactic abstraction).

- Can permit **intentional redirection** of programmer mental model
- Or introduce **incidental opacity** or makes the language seem more complicated than it is
  - But could **still be worthwhile** on balance (e.g. terseness, visual distinction)

A tale of two sugars

Both **obscure the core language** (syntactic abstraction).

- Can permit **intentional redirection** of programmer mental model
- Or introduce **incidental opacity** or makes the language seem more complicated than it is
  - But could **still be worthwhile** on balance (e.g. terseness, visual distinction)

Worth considering when defining your own

Names matter

TypeScript

- **Subtyping**: types form a lattice over $<:$ relation
  - $\top$ is the **universal supertype**

$$\bot <: \texttt{"hello"} <: \texttt{string} <: \top$$

- **Gradual typing**: typed-untyped codebase interaction
  - Dyn is the **dynamic type**, an "escape hatch"

$$\bot <:> \mathsf{Dyn} <:> \top$$

Question

How to express **any possible type**?

$$\mathrm{isString} : ??? \rightarrow \mathsf{boolean}$$

Question

How to express **any possible type**?

$$\mathrm{isString} : \top \rightarrow \mathsf{boolean}$$

Writing the program

How to express **any possible type**?

```
function isString(x: /* ?? */): boolean
```

Writing the program

How to express **any possible type**?

```
function isString(x: any): boolean
```

Writing the program

How to express **any possible type**?

```
function isString(x: any): boolean
```

Problem: any is Dyn, not ⊤!

Dyn vs. $\top$

$$\bot \ <:> \ \textbf{Dyn} \ <:> \ \top$$

Dyn is unsound: breaks typing guarantees, causes major incidents

So how do we write ⊤ actually?

```
function isString(x: unknown): boolean
```

Vernacular misconceptions

> ***any*** *(adj.)*
>
> *1. one or some indiscriminately of whatever kind [...]*
>
> *2. unmeasured or unlimited in amount, number, or extent*

source: Merriam-Webster

But wait, it gets worse

	Dynamic	Top
TypeScript (JS), **Luau** (Lua)	`any`	`unknown`
Flow (JS)	`any`	`mixed`
mypy, Pyre (Python)	`Any`	`object`
Sorbet (Ruby)	`untyped`	`anything`[1]
Hack (PHP)	`dynamic`	`mixed`
Elixir (Erlang)	`dynamic`	`any`[2]
Typed Racket (Racket)	-[3]	`Any`
Scala, Kotlin	-	`any`
Swift	-	`Any`[4]

Developers, developers, developers!

- **Half the languages** use any for Dyn, and the **other half** for $\top$
  - First group includes JavaScript and Python-based
  - Second group includes Scala, Kotlin, Swift
- Languages that don't use any all have **different names for** $\top$
  - `unknown`, `mixed`, `object`, `anything`

Researchers, researchers, researchers!

- **Modern, research-based PLs** like Luau (Roblox) are adopting the same names
  - Luau follows TypeScript's naming exactly, despite ~no linguistic heritage
- **Names change during implementation**: Elixir's paper uses `term` for $\top$, but the documentation uses `any`
  - Trivial-seeming changes can affect careful work

Of note: Stefik & Siebert (2013), An Empirical Investigation into Programming Language Syntax

Punctuation & whitespace matter

Or, how I became an η-expansion scrooge

Background

- It's the early 2010s, and everyone is talking about **Node.js**, the new server-side JavaScript runtime
- JavaScript uses the **reactor pattern** to perform **non-blocking I/O**

Async in JavaScript: abridged history

1. **Continuation-passing style** (“callback hell”)
2. **`Promise`** chaining
3. **`async/await`** (today’s world)

Continuation-passing style

```
runA(function (a) {
  runB(a, function (b) {
    runC(b, function (c) {
      …
    })
  })
})
```

Promise chaining

```
runA()
  .then(runB)
  .then(runC)
  .then(…)
```

async/await

```
const a = await runA()
const b = await runB(a)
const c = await runC(b)
…
```

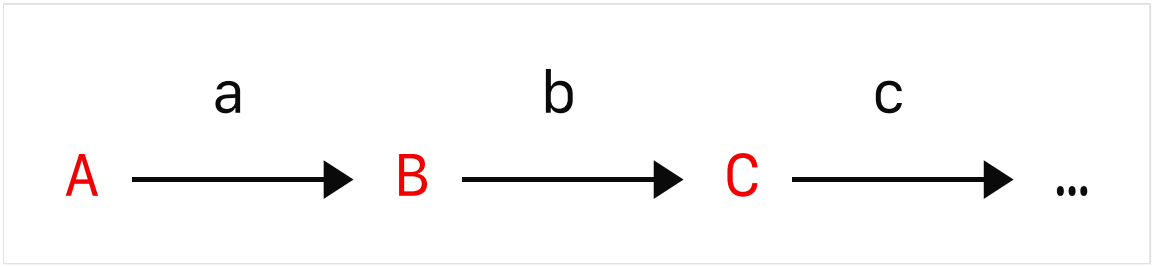
Evolution

```
runA(function (a) {
  runB(a, function (b) {
    runC(b, function (c) {
      …
    })
  })
})
```

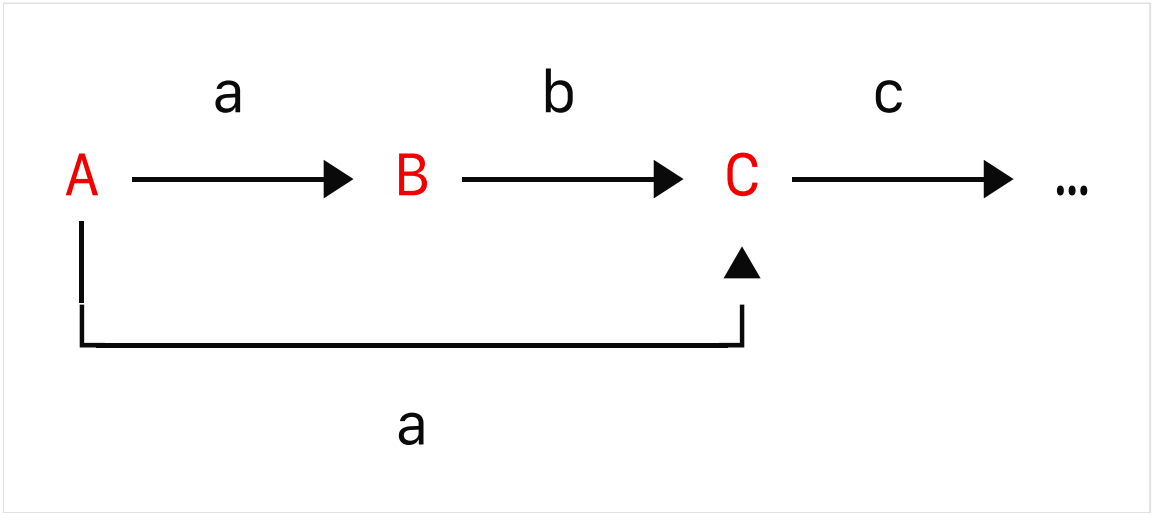
```
runA()
  .then(runB)
  .then(runC)
  .then(…)
```

```
const a = await runA()
const b = await runB(a)
const c = await runC(b)
…
```

Previous dataflow



Alternate dataflow



Callbacks and `async/await`

```
runA(function (a) {
  runB(a, function (b) {
    runC(a, b, function (c) {
      …
    })
  })
})
```

```
const a = await runA()
const b = await runB(a)
const c = await runC(b)
…
```

Promise chaining

```
runA()
  .then(runB)
  .then(b => runC(??, b)) // Missing `a`
  .then(…)
```

Promise chaining

```
runA()
  .then(runB)
  .then(runC)
  .then(…)
```

Promise chaining

```
runA()
  .then(a => [a, runB()])
  .then(([a, b]) => runC(a, b))
  .then(…)
```

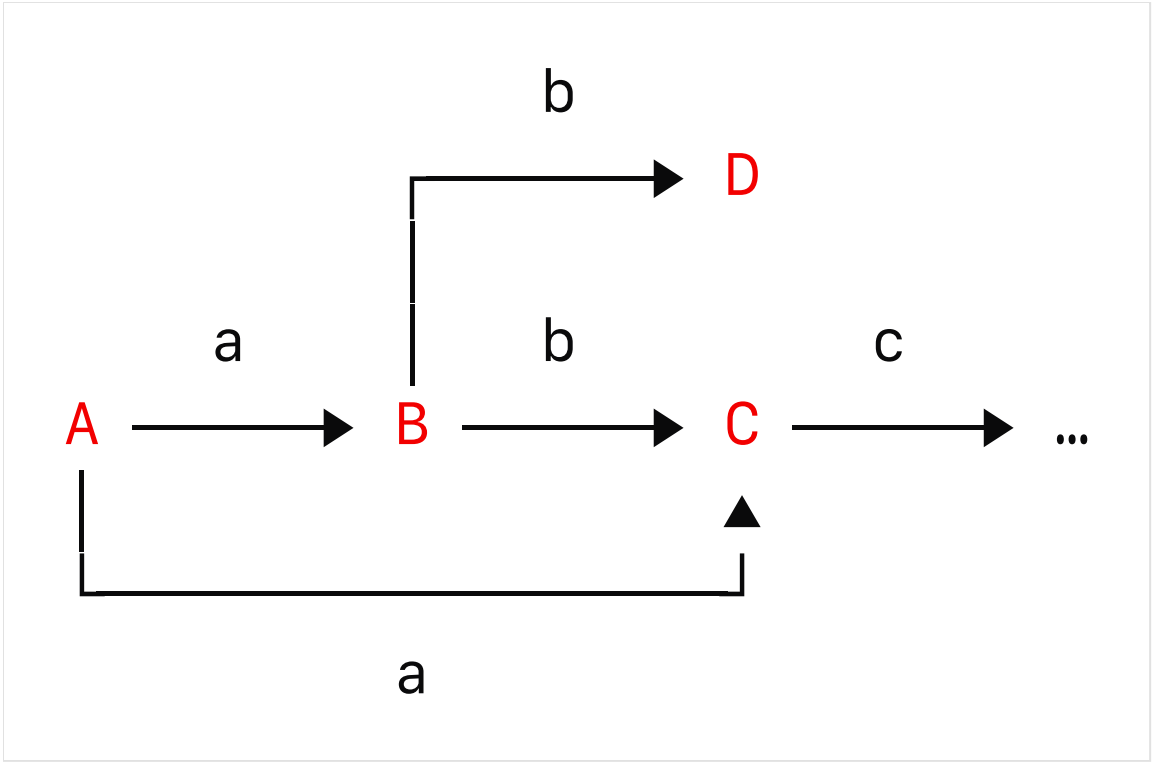
Promise chaining

```
runA()
  // Must return Promise<…>
  .then(a => [a, runB()])
  .then(([a, b]) => runC(a, b))
  .then(…)
```

Promise chaining

```
runA()
  // Must return Promise<…>
  .then(a => Promise.all([a, runB()]))
  .then(([a, b]) => runC(a, b))
  .then(…)
```

Add an extra call D



Promise chaining

```
runA()
  .then(a => Promise.all([a, runB()]))
  .then(([a, b]) => Promise.all(
    [runC(a, b), runD()]
  ))
  .then(([c, _] => …)
```

`Promise` chaining

```
runA()
  .then(a => Promise.all([a, runB()]))
  .then(([a, b]) => Promise.all(
    [runC(a, b), runD()]
  ))
  .then(([c, _] => …)
```

(This is my η villain origin story btw)

Issues with `Promise`-chaining

- Super fluent syntax for **linear dataflow**
- **Immediately falls apart** with the slightest branching
  - aka real world
- **Lost the ability to shadow** from earlier callbacks

Nested scopes

```
runA(function (a) {
  runB(a, function (b) {
    runC(a, b, function (c) {
      …
    })
  })
})
```

```
do
  a <- runA
  b <- runB a
  c <- runC a b
```

So why did people hate callbacks?

Indentation & closing delimiters

```
runA(function (a) {
    runB(a, function (b) {
        runC(a, b, function (c) {
            …
        })
    })
})
```

Indentation & closing delimiters

```
runA(function (a) {
    runB(a, function (b) {
        runC(a, b, function (c) {
            …
        })
    })
})
```

This is all **completely incidental**, yet resulted in a **loss of expressive fluency** in async-handling constructs!

Concrete vs. abstract syntax

```
runA(function (a) {
  runB(a, function (b) {
    runC(a, b, function (c) {
      …
    })
  })
})
```

```
do
  a <- runA
  b <- runB a
  c <- runC a b
```

Moral of the story

Languages can go through **tremendous semantic changes** based on **completely incidental** syntax

What now?

Problems

Too often, concrete syntax decisionmaking is:

1. **Idiosyncratic**
2. **Under-documented**
3. **Under-researched**

And that is a shame!

First steps

1. **Idiosyncratic** → take these questions seriously (and rigorously)
2. **Under-documented** → document in archival/semi-archival media, including when decisions are unprincipled
3. **Under-researched** → we have tremendous opportunity to study these questions…more on this soon :)

Conclusions

If we take seriously the idea of programming languages as user interfaces, then **concrete syntax matters just as much as semantics**.

Concrete syntax shapes the way people understand semantics, and in turn **shapes the semantics themselves**.

1. Uses `any` for union types ↩
2. Paper by Castagna et al. 2023 uses `term`, but the documentation uses `any` ↩
3. Uses module boundary system instead of creating a dynamic type ↩
4. Not truly $\top$, but a type-erased existential box that requires casting or narrowing to use ↩