# Progress on Developing a (Mini) Compiler for RESOLVE Theories

Daniel Welch

**RESOLVE 2023**

PennState
College of Information
Sciences and Technology

# Agenda

- Some goals

- The foundational language: "Math Entity Theory"
  - current syntax & classification system setup
  - implicit parameters

- Compilation pipeline
  - leveraging features of JDK 20 & 21 in the implementation
  - incremental compilation

# Agenda

- Some goals
- The foundational language: "Math Entity Theory"
  - current syntax & classification system setup

**Time permitting:** a short demo illustrating how Java 21 is utilized for representing ASTs

- Future work: mixfix syntax, the programming language, **L**anguage **S**erver **P**rotocol (**LSP**) support

# Goals

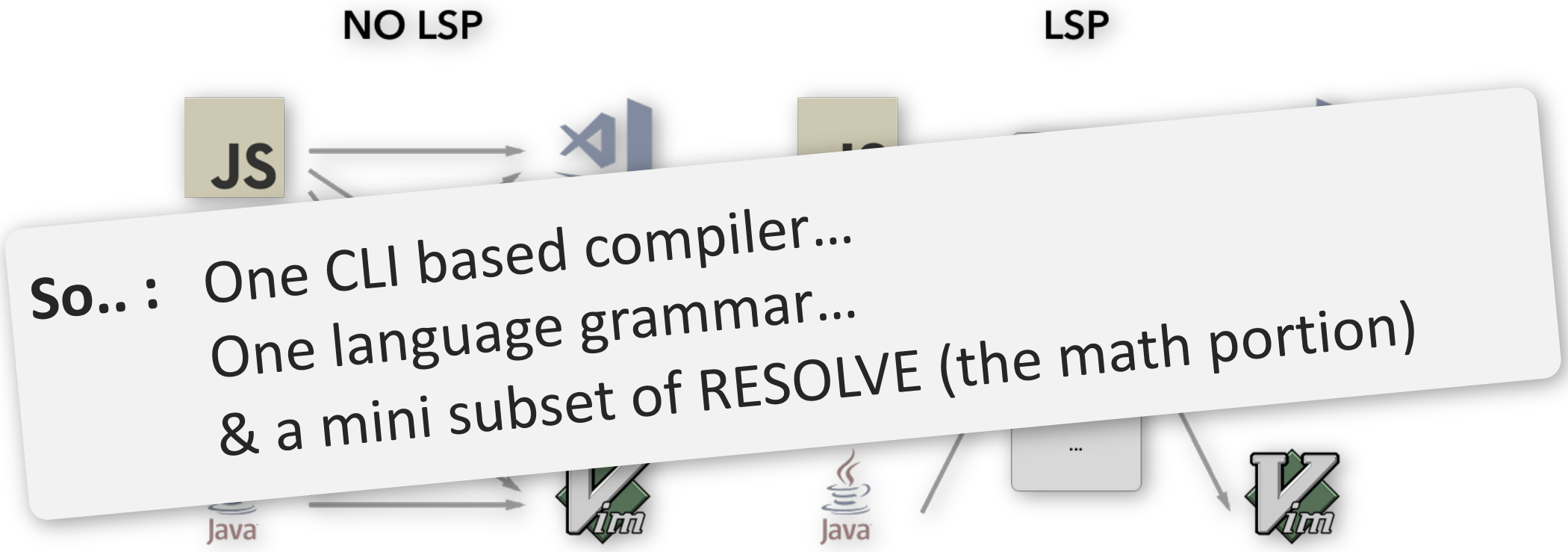
- **G1:** create a maintainable, reusable compiler/checker for RESOLVE mathematics and specifications
  - a type of 'sandbox' for experimentation with mechanically checkable notations for expressing mathematics
  - while doing this: come up with context formation rules & judgements for classification checking
- **G2:** support *incremental compilation* (more later)
  - the idea: semantic analysis (+ typechecking-/classification- checking) shouldn't have to be redone for everything every keystroke

# Goals

- **G3:** integration into editors: VSCode, Jetbrains Fleet, etc
  - or any other editor that supports the **L**anguage **S**erver **P**rotocol – **LSP**

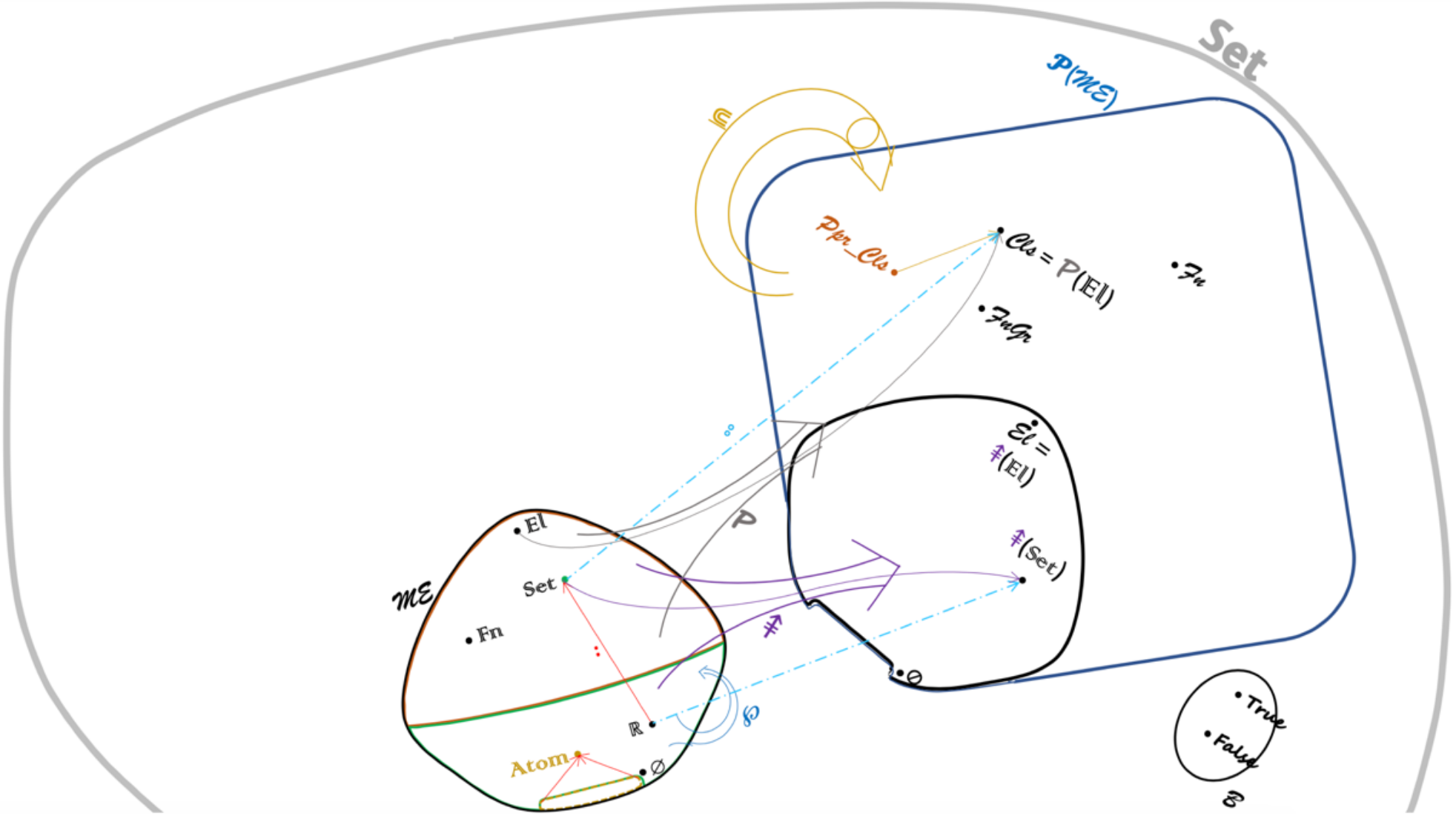
# Goals

- **G3:** integration into editors: VSCode, Jetbrains Fleet, etc
  - or any other editor that supports the **L**anguage **S**erver **P**rotocol – **LSP**

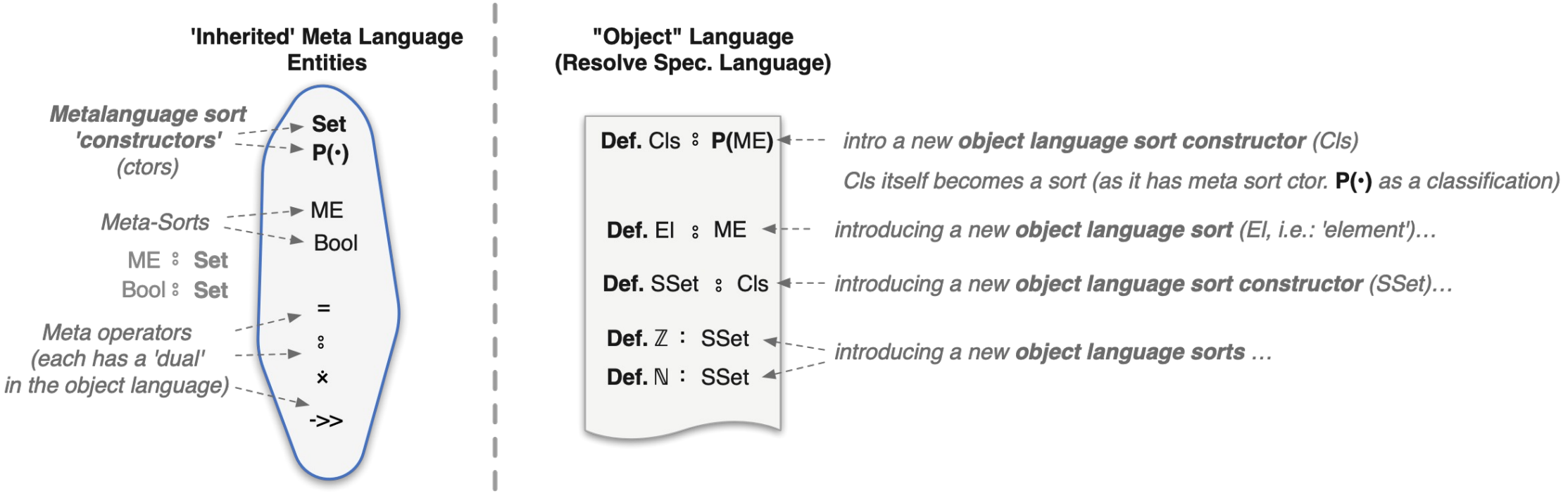



**So.. :** One CLI based compiler…
One language grammar…
& a mini subset of RESOLVE (the math portion)

# Math Entity Theory: The Big Picture



# Meta vs Object Language




**Some example categorical definitions for theories we'd like to express in our object language**

**Categorical Def. for** SStr ⁝ Cls , Λ : SStr , ext : SStr x El -> ( SStr ~ {Λ}) **is** … ← for heterogeneously-sorted strings

**Categorical Def. for** TTr ⁝ Cls , Ω : TTr , Jn : Str(TTr) x El -> ( TTr ~ {Ω}) **is** … ← *for heterogeneously-sorted trees*

# Math Language Constructs (Tentative)

- The math theory language targeted will support:
  - *Theory modules (+ theory extensions)*
  - *Definitions: Sort-introducing, categorical, inductive & constant*
  - *Recognitions* (for sub-sorting/sub-classifications)
- **A "Sequent" VC Conjecture construct that runs the prover**
- Potential syntax:

**Conj.** VCa1: givenExp1, …, givenExpN |- goalExp ;

(can only appear globally at the module level)

# Expressions

- Most type-theory based languages distinguish (syntactically) between expressions ***e*** denoting values and types $\boldsymbol{\tau}$

| | | | |
|---|---|---|---|
| $e ::= e :: \tau$ | annotated term[1] | $\tau ::= \alpha$ | base type |
| $\mid x$ | variable | $\mid \tau \to \tau'$ | function type |
| $\mid e\ e'$ | application | | |
| $\mid \lambda x \to e$ | lambda abstraction | | |

where α ∈ {**Bool**, **Unit**, **Nat**, ….}

- We make no syntactic distinctions between types and terms
  - similar to dependently typed languages – Lean, Agda, Coq, etc. (but without all the baggage)

# Expressions

- Most type-theory based languages distinguish (syntactically) between expressions ***e*** denoting values and types $\boldsymbol{\tau}$

| | | | |
|---|---|---|---|
| $e ::= e :: \tau$ | annotated term[1] | $\tau ::= \alpha$ | base type |
| $\mid x$ | variable | $\mid \tau \to \tau'$ | function type |
| $\mid e\ e'$ | application | | |
| $\mid \lambda x \to e$ | lambda abstraction | | |

where α ∈ {**Bool**, **Unit**, **Nat**, ….}

- We make no syntactic distinctions between types and terms
  - similar to dependent… …gda, Coq, etc.
    (but without all t… …guage)

$$ext: Str \times \mathbb{E}\mathbb{L} \to (Str \sim \{\Lambda\})$$

# Expressions (Implicit Slots)

- Like Lean and others, we also support *implicit classification parameters*:

```
def cmps {A B C : Type} (g : B → C) (f : A → B) (x : A) : C := g (f x)
```

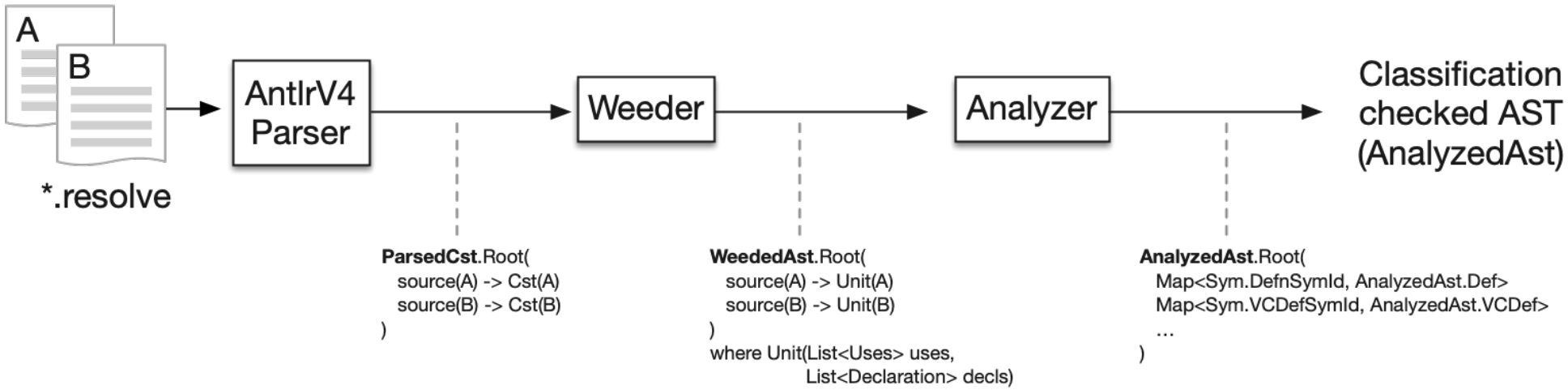
- Ours vary slightly (implicits are embedded within the constituent parts of a definition's schema):

```
Def. cmps (g : (B : SSet) → (C : SSet), f : (A : SSet) → B, x : A) : C = g(f(x))
```

- Variables bound left-to-right in the schema for the definition
  - normal and meta colon variable binders can be embedded in classificational expressions: ( id : *e* ) and ( id ⦂ *e* )
  - *e*'s classification needs to be compatible with the colon-symbol used

# Compilation Pipeline

- Usual "pipeline" architecture is used:




- Weeder: additional syntactic checks (ones that the parser can't pick up alone w/o semantic actions)
- Analyzer: builds scopes, resolves symbols, classification checking

# Work in Progress: Incremental Compilation

- The ASTs resulting from the analysis phase get cached into a "ChangeSet" data structure which keeps track of "fresh" and "stale" sources (in Maps)
  - goal: allow for "incremental compilation"
  - stale sources are re-analyzed in parallel; fresh sources can be reused

- The current implementation (which is not supposed to be the end goal) only caches files that are guaranteed to never change (i.e. a standard library)
  - so right now user source code is always recompiled, but not library code (once it has been compiled once, because the library code cannot change)

# Compilation Pipeline

- Each phase produces a different (immutable) AST/hierarchy
  - each AST holds more information the further along you get in the pipeline
- Each AST (except the concrete syntax tree produced by Antlr4) is modeled using Java record types + sealed interfaces
- Visitor pattern no longer needed due to (recent) addition of pattern matching and object deconstruction patterns
  - demo (time permitting)