

Lean4

lecture 1

Quick Tour of Lean4

Metaprogramming

- Programming: Writing code to manipulate data
- Metaprogramming: Writing code to manipulate *code*
- Reflection: Programs examining their own code

Why Metaprogramming

- Extend Lean's syntax
- Create new tactics
- Interface with Lean's compiler
- Execute tactics based on data
- Inspect the content of compiled Lean

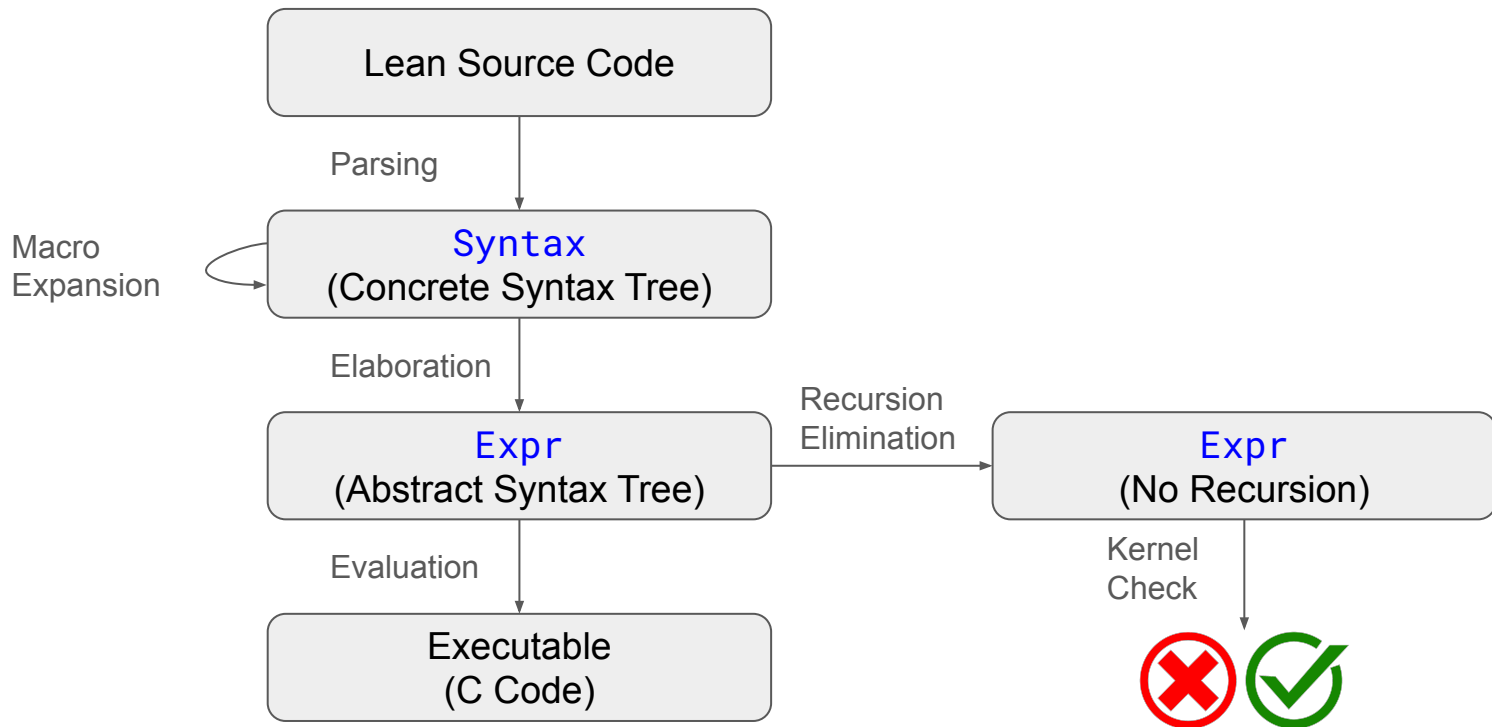
What's Unique about Lean?

Language	Meta-Level Language
C	C Macros (e.g. #define)
Rocq	Ltac, OCaml
Isabelle	SML, Scala
Agda	Haskell
Lean4	?

What's Unique about Lean?

Language	Meta-Level Language
C	C Macros (e.g. #define)
Rocq	Ltac, OCaml
Isabelle	SML, Scala
Agda	Haskell
Lean4	Lean4!

Lean Compiler Overview



Compiler Intermediates

```
--
```

```
Lean syntax trees.
```

```
-/
```

```
inductive Syntax where
```

```
...
```

```
--
```

```
Lean expressions.
```

```
-/
```

```
inductive Expr where
```

```
...
```

Compiler Intermediates

```
--  
Lean syntax trees.  
-/  
inductive Syntax where  
  ...
```

```
--  
Lean expressions.  
-/  
inductive Expr where  
  ...
```

De Bruijn Index

- Nameless variable representation: Replaces variable names with natural numbers indicating the number of binders in scope between def and use
- Easier to do capture-avoiding substitution
- Easier to check for alpha-equivalence
- Need additional context for free variables

$\lambda z. (\lambda y. y (\lambda x. x)) (\lambda x. z x)$ \longleftrightarrow $\lambda (\lambda 0 (\lambda 0)) (\lambda 1 0)$

$\lambda x. (\lambda y. x z)$ \longleftrightarrow $\lambda (\lambda 1 3)$
[z -> 3]

Locally Nameless

- Bound variables: De Bruijn Index
- Free variables: User-defined names

$\lambda x. (\lambda y. x z)$



$\lambda (\lambda 1 z)$

Lean's Expression Type

```
inductive Expr where
```

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Lean's Expression Type

inductive Expr where

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Bound variables: occurrences of variables where there's a binder above it.

The second **x** in `fun x => x + y`.

Lean's Expression Type

inductive Expr where

- | bvar : Nat → Expr
- | fvar : FVarId → Expr
- | mvar : MVarId → Expr
- | sort : Level → Expr
- | const : Name → List Level → Expr
- | app : Expr → Expr → Expr
- | lam : Name → Expr → Expr → BinderInfo → Expr
- | forallE : Name → Expr → Expr → BinderInfo → Expr
- | letE : Name → Expr → Expr → Expr → Bool → Expr
- | lit : Literal → Expr
- | mdata : MData → Expr → Expr
- | proj : Name → Nat → Expr → Expr

Free variables: occurrences of variables without an explicit binder. Can loop up it's type in [LocalContext](#).

The **y** in `fun x => x + y`.

Lean's Expression Type

`inductive Expr where`

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Meta variables: existential variables in Rocq. Placeholder values to be filled in during elaboration.

The `_` in `fun (x : _) => x + 1`.

Lean's Expression Type

inductive Expr where

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Polymorphic type universe levels

Lean's Universe Structure

sort	Prop (Sort 0)	Type (Sort 1)	Type 1 (Sort 2)	Type 2 (Sort 3)
type	True	Bool	Nat -> Type	Type -> Type 1
term	True.intro	true	fun n => Fin n	fun (⌊ : Type) => Type

Lean's Expression Type

inductive Expr where

| bvar : Nat → Expr

| fvar : FVarId → Expr

| mvar : MVarId → Expr

| sort : Level → Expr

| const : Name → List Level → Expr

| app : Expr → Expr → Expr

| lam : Name → Expr → Expr → BinderInfo → Expr

| forallE : Name → Expr → Expr → BinderInfo → Expr

| letE : Name → Expr → Expr → Expr → Bool → Expr

| lit : Literal → Expr

| mdata : MData → Expr → Expr

| proj : Name → Nat → Expr → Expr

Constants: things defined earlier
in some Lean document
(functions, theorems, etc.)

They may be polymorphic over a
list of universe levels

Lean's Expression Type

`inductive Expr where`

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Function applications

`f x y` becomes `app (app f x) y`

Lean's Expression Type

`inductive Expr where`

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Function abstraction

`fun (x : t) => e` becomes
`lam x t e`

`BinderInfo` specifies if the binder is
implicit or a typeclass argument

Lean's Expression Type

inductive Expr where

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Dependent arrow expression

$(x : t) \rightarrow e$ becomes
`forallE x t e`

Non-dependent arrows like $x \rightarrow e$ is
also encoded this way

Lean's Expression Type

inductive Expr where

- | bvar : Nat → Expr
- | fvar : FVarId → Expr
- | mvar : MVarId → Expr
- | sort : Level → Expr
- | const : Name → List Level → Expr
- | app : Expr → Expr → Expr
- | lam : Name → Expr → Expr → BinderInfo → Expr
- | forallE : Name → Expr → Expr → BinderInfo → Expr
- | letE : Name → Expr → Expr → Expr → Bool → Expr
- | lit : Literal → Expr
- | mdata : MData → Expr → Expr
- | proj : Name → Nat → Expr → Expr

Let expressions

let (x : t) := e; body
becomes
letE x t e body

Last boolean argument tells the
elaborator whether the let is
dependent or not

Lean's Expression Type

`inductive Expr where`

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Numeric and string literals

Exists mostly for performance reasons
so we don't represent `(1000 : Nat)`
as `Nat.succ (Nat.succ (Nat.succ ...))`

Lean's Expression Type

`inductive Expr where`

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Stores meta-data along with an expression

Lean's Expression Type

`inductive Expr where`

```
| bvar      : Nat → Expr
| fvar      : FVarId → Expr
| mvar      : MVarId → Expr
| sort      : Level → Expr
| const     : Name → List Level → Expr
| app       : Expr → Expr → Expr
| lam       : Name → Expr → Expr → BinderInfo → Expr
| forallE   : Name → Expr → Expr → BinderInfo → Expr
| letE      : Name → Expr → Expr → Expr → Bool → Expr
| lit       : Literal → Expr
| mdata     : MData → Expr → Expr
| proj      : Name → Nat → Expr → Expr
```

Projection: used to access fields in a structure

Redundant constructor, purely for performance

Examples in Lean Code