Lean4

lecture 2

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
    ...
```

Parsing

```
inductive Syntax where
  | missing : Syntax
  | node (kind : SyntaxNodeKind) (args : Array Syntax) : Syntax
  | atom : String → Syntax
  | ident : Name → Syntax

def ParserFn :=
  ParserContext → ParserState → ParserState

structure Parser where
  info : ParserInfo
  fn : ParserFn
```

Parsing

```
inductive Syntax where
  | missing : Syntax
  | node (kind : SyntaxNodeKind) (args : Array Syntax) : Syntax
  | atom : String → Syntax
  | ident : Name → Syntax
```

`Syntax` is a n-ary tree of atomic tokens and identifiers.

```
def ParserFn :=
  ParserContext → ParserState → ParserState

structure Parser where
  info : ParserInfo
  fn : ParserFn
```

Parsing

```
inductive Syntax where
  | missing : Syntax
  | node (kind : SyntaxNodeKind) (args : Array Syntax) : Syntax
  | atom : String → Syntax
  | ident : Name → Syntax
```

```
def ParserFn :=
  ParserContext → ParserState → ParserState

structure Parser where
  info : ParserInfo
  fn : ParserFn
```

Primitive parsers can be built out of functions that consume the raw text and returns syntax trees.

Primitive Parsers gives full flexibility
But are tedious to write

Context Free Grammar

```
E = ( E )
  | numbers
  | E ^ E
  | E * E
  | E + E
  | -E
```

Context Free Grammar

```
E = ( E )
    | numbers
    | E ^ E
    | E * E
    | E + E
    | -E
```

Each line separated by a “|” represent a production rule

Context Free Grammar

```
E = ( E )
  | numbers
  | E ^ E
  | E * E
  | E + E
  | -E
```

Capital letters represent non-terminals: things that can be expanded with production rules

Context Free Grammar

```
E = ( E )
  | numbers
  | E ^ E
  | E * E
  | E + E
  | -E
```

Numbers and symbols are terminals: things that cannot be expanded further with production rules

Context Free Grammar

```
E = ( E )
  | numbers
  | E ^ E
  | E * E
  | E + E
  | -E
```

This is ambiguous! How do we start parsing this expression?

-(1 + 1 + 2) * 3 * 2 ^ 3 ^ 2

Operator Precedence Grammar

```
E = ( E[0] )               [50]
  | numbers                [50]
  | E[31] ^ E[30]          [30]
  | E[20] * E[21]          [20]
  | E[10] + E[11]          [10]
  | -E[39]                 [40]
```

Operator Precedence Grammar

```
E = ( E[0] )            [50]
  | numbers             [50]
  | E[31] ^ E[30]       [30]
  | E[20] * E[21]       [20]
  | E[10] + E[11]       [10]
  | -E[39]              [40]
```

Each production rule and non-terminal now get a precedence value

Operator Precedence Grammar

```
E = ( E[0] )            [50]
  | numbers             [50]
  | E[31] ^ E[30]       [30]
  | E[20] * E[21]       [20]
  | E[10] + E[11]       [10]
  | -E[39]              [40]
```

We dictate a non-terminal N with precedence n can only be expanded with a production rule R with precedence r when r >= n

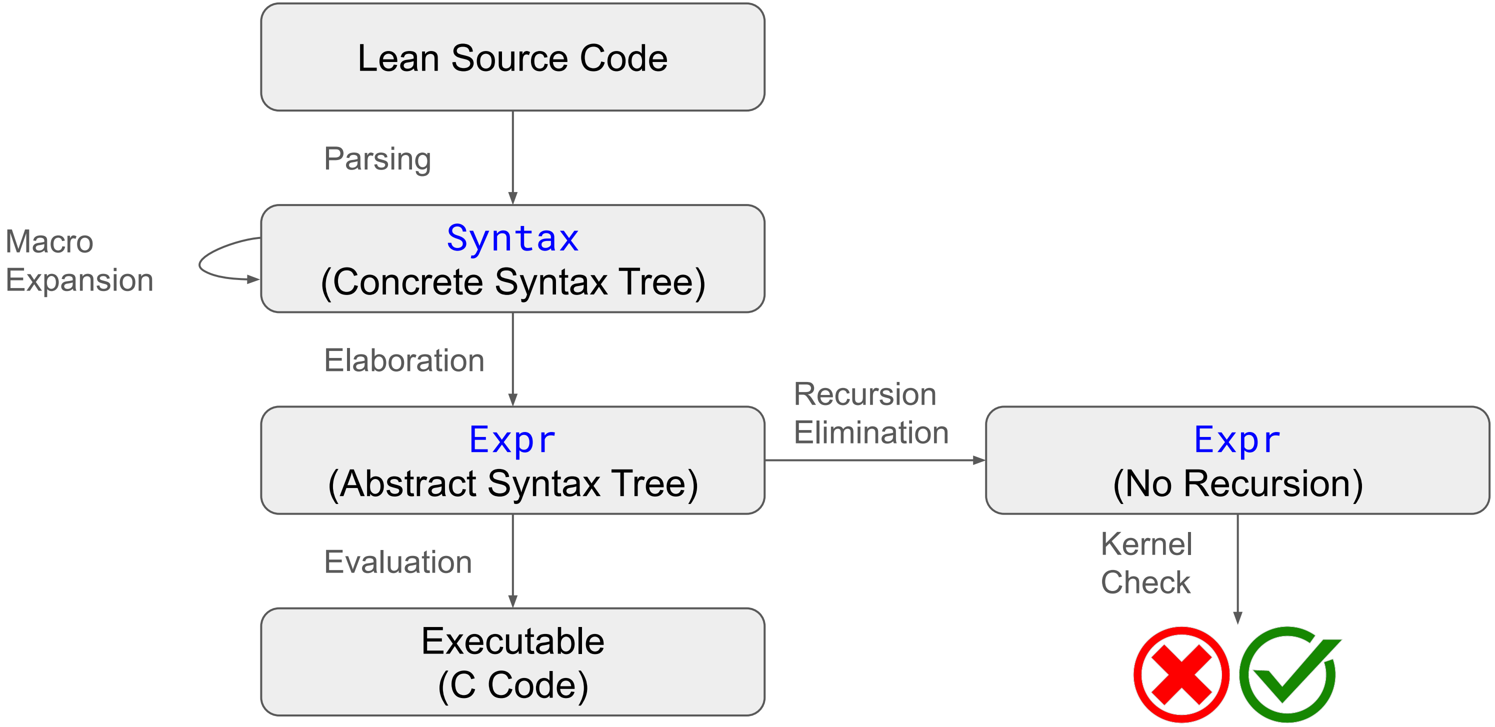
Operator Precedence Grammar

```
E = ( E )
  | numbers
  | E[31] ^ E[30]      [30]
  | E[20] * E[21]      [20]
  | E[10] + E[11]      [10]
  | -E[39]             [40]
```

By default, non-terminals get 0 and productions get maximal precedence value

Examples in Lean Code

Lean Compiler Overview



Metaprogramming

- Want to programmatically manipulate Lean expressions
- Monad zoo:
  - `CoreM`: gives access to the environment, including imports, declarations, options etc.
  - `MetaM`: gives access to the metavariable context, including currently declared meta-variables and their assignments (if any)
  - `TermElabM`: gives access to various information used during elaboration
  - `TacticM`: gives access to the list of current goals
  - `MacroM`: used for macro expansions, very limited in capabilities

Metaprogramming

- Want to programmatically manipulate lean expressions
- Monad zoo:
  - `CoreM`: gives access to the environment, including imports, declarations, options etc.
  - `MetaM`: gives access to the metavariable context, including currently declared meta-variables and their assignments (if any)
  - `TermElabM`: gives access to various information used during elaboration
  - `TacticM`: gives access to the list of current goals

Each monad above strictly increase in capabilities
e.g., `TacticM` can do everything `TermElabM` can do and more

  - `MacroM`: used for macro expansions, very limited in capabilities

Examples in Lean Code

Type Unification

- Determine whether two expressions are equal
- Assign meta-variables knowing that two expressions have to be equal
- Determine universe level meta-variables.
- Determine type class instances

`isDefEq`

- The main API for doing type unification
- It determines whether two expressions are definitionally equal
- `Lean.Meta.isDefEq : Expr -> Expr -> MetaM Bool`
  - Meta-level function used in elaboration
  - Will assign meta-variables based on a depth argument
- `Lean.Kernel.isDefEq`
  - Kernel-level function
  - The kernel does not support meta-variables
  - Rarely needed in meta-programming

State Management

- Remember `Lean.Meta.isDefEq` will modify the meta-variable state!
- `Lean.withoutModifyingState`
  - Use this to execute a block of meta-level code without modifying the state

Proof State

- Each goal is a meta-variable
  - `MetavarDecl`
    - Stores all information about a meta-variable
- Hypotheses in scope are stored inside a local context
  - `LocalContext`
    - An array of free variables in the current context that can appear in the goal
  - `LocalDecl`
    - A free variable that can appear in current goal

Finally, let’s write some tactics!

Some interesting applications of meta-programming

Canonical

- A tactic that exhaustively searches for proof terms
- Search engine implemented in parallel Rust
- Builds a canonical proof in Lean using meta-programming
- https://github.com/chasenorman/CanonicalLean

Alloy

- Library to embed external C FFI code directly in Lean
- https://github.com/tydeu/lean4-alloy

```
alloy c extern def myAdd (x y : UInt32) : UInt32 := {
  return x + y;
}
```

```
LEAN_EXPORT uint32_t _alloy_c_l_myAdd ( uint32_t x , uint32_t y ) {
  return x + y;
}
```

Plausible

- A property testing framework for Lean 4 that integrates into the tactic framework
- https://github.com/leanprover-community/plausible

```
import Plausible

example (xs ys : Array Nat) : xs.size = ys.size → xs = ys := by
  /--
  ===================
  Found a counter-example!
  xs := #[0]
  ys := #[1]
  guard: 1 = 1
  issue: #[0] = #[1] does not hold
  (0 shrinks)
  -------------------
  -/
  plausible

#eval Plausible.Testable.check <| ∀ (xs ys : Array Nat), xs.size = ys.size → xs = ys
```

How to Learn More

- Metaprogramming in Lean 4
  - https://leanprover-community.github.io/lean4-metaprogramming-book/
- Lean Language Reference
  - https://lean-lang.org/doc/reference/latest/
- Read standard library code
  - https://github.com/leanprover/lean4