



# Theorembeweiserpraktikum

**Syntax & Macros**

Jakob von Raumer, Sebastian Ullrich | SS 2022



THEOREM PROVER

# It's a Date! – Second Attempt

final presentations: Monday, July 27, 10:30–12:00

# Syntax & Macros

```
...
syntax "[Com]" com* "]" : term

macro_rules
| `(`[Com|])           => `(Com.skip)
| `(`[Com|$c $cs*])    => `(Com.seq `[Com|$c] `[Com|$cs*])
| `(`[Com|if ($b) { $cts* }]) => `(`[Com|if ($b) { $cts* } else {}])
...

def example1 := `[Com|
  x := 8;
  y := 10;
  if (x < y) {
    x := x + 1;
  } else {
    y := y + 3;
  }
}]
```

What is even going on here?

# The Lean 4 Frontend Pipeline

- parser:  $\approx \text{String} \rightarrow \text{Syntax}$
- macro expansion:  $\text{Syntax} \rightarrow \text{MacroM Syntax}$ 
  - actually interleaved with elaboration
- elaboration
  - terms:  $\text{Syntax} \rightarrow \text{TermElabM Expr}$
  - commands:  $\text{Syntax} \rightarrow \text{CommandElabM Unit}$
  - universes:  $\text{Syntax} \rightarrow \text{TermElabM Level}$
  - tactics:  $\text{Syntax} \rightarrow \text{TacticM Unit}$

Every part of the pipeline is extensible by users

# Notations

```
infixl:65 " + " => HAdd.hAdd -- left-associative
infix:50 " = " => Eq -- non-associative
infixr:80 " ^ " => HPow.hPow -- right-associative
prefix:100 "-" => Neg.neg
postfix:max "-1" => Inv.inv
```

# Notations

```

infixl:65 " + " => HAdd.hAdd -- left-associative
infix:50 " = " => Eq         -- non-associative
infixr:80 " ^ " => HPow.hPow -- right-associative
prefix:100 "-"      => Neg.neg
postfix:max "-1"    => Inv.inv
  
```

These are just macros.

```

notation:65 lhs:65 " + " rhs:66 => HAdd.hAdd lhs rhs
notation:50 lhs:51 " = " rhs:51 => Eq lhs rhs
notation:80 lhs:81 " ^ " rhs:80 => HPow.hPow lhs rhs
notation:100 "-" arg:100 => Neg.neg arg
notation:1024 arg:1024 "-1" => Inv.inv arg
  
```

# Mixfix Notations

```
notation:max "(" e ")" => e  
notation:10  $\Gamma$  "  $\vdash$  " e " : "  $\tau$  => Typing  $\Gamma$  e  $\tau$ 
```

# Mixfix Notations

```
notation:max "(" e ")" => e
notation:10 "⊢" e " : " τ => Typing ⊢ e τ
```

```
notation:50 a:51 " = " b:51 " = " c:51 => a = b ∧ b = c
#reduce 1 = 2 = 3 -- 1 = 2 ∧ 2 = 3
```

Overlapping notations are parsed with a (local) “longest parse” rule

See also <https://leanprover.github.io/lean4/doc/syntax.html>



```
notation:max "(" e ")" => e
```

This is just a macro.

```
syntax:max "(" term ")" : term  
macro_rules ...
```

`term` is a *syntax category*

```
notation:max "(" e ")" => e
```

This is just a macro.

```
syntax:max "(" term ")" : term  
macro_rules ...
```

`term` is a *syntax category*

```
declare_syntax_cat index  
syntax ident : index  
syntax ident ":" term : index  
syntax term "<=" ident "<" term : index  
  
syntax "Σ" index ", " term : term
```

# More Syntax

```
syntax binderIdent      := ident <|> "_"  
syntax unbracketedExplicitBinders := binderIdent+ (" : " term)?  
  
syntax "begin " tactic,*,? "end" : tactic
```

<https://github.com/leanprover/lean4/blob/master/src/Init/Notation.lean#L40>

# Summary: Parsing

Each syntax category is

- a precedence (Pratt) parser composed of a set of leading and trailing parsers
- with per-parser precedences
- following the longest parse rule

# Summary: Parsing

Each syntax category is

- a precedence (Pratt) parser composed of a set of leading and trailing parsers
- with per-parser precedences
- following the longest parse rule

on the lower level: a combinatoric, non-monadic, lexer-less, memoizing recursive-descent parser

<https://github.com/leanprover/lean4/blob/master/src/Lean/Parser/Basic.lean#L7>

# Macros

```
notation:max "(" e ")" => e
```

This is just a macro.

```
syntax:max "(" term ")" : term
macro_rules
| `(($e)) => `($e)
```

# Macros

```
notation:max "(" e ")" => e
```

This is just a macro.

```
syntax:max "(" term ")" : term
macro_rules
| `(($e)) => `($e)
```

which can also be written as

```
macro:max "(" e:term ")" : term => `($e)
```

# Macros

```
notation: max "(" e ")" => e
```

This is just a macro.

```
syntax: max "(" term ")" : term
macro_rules
| `(($e)) => `($e)
```

which can also be written as

```
macro: max "(" e:term ")" : term => `($e)
```

or, in this case

```
macro: max "(" e:term ")" : term => pure e
```



# Quotations

```
`(let $id:ident $binders* $[: $ty?]? := $val; $body)
```

- has type `Syntax` in patterns
- has type `m Syntax` given `MonadQuotation m` in terms
- `id`, `val`, `body` have type `Syntax`
- `binders` has type `Array Syntax`
- `ty?` has type `Option Syntax`

# Quotations

```
`(let $id:ident $binders* $[: $ty?]? := $val; $body)
```

- has type `Syntax` in patterns
- has type `m Syntax` given `MonadQuotation m` in terms
- `id`, `val`, `body` have type `Syntax`
- `binders` has type `Array Syntax`
- `ty?` has type `Option Syntax`
- `ts` in `$ts,*` has type `SepArray`

# Quotations

```
`(let $id:ident $binders* $[: $ty?]? := $val; $body)
```

- has type `Syntax` in patterns
- has type `m Syntax` given `MonadQuotation m` in terms
- `id`, `val`, `body` have type `Syntax`
- `binders` has type `Array Syntax`
- `ty?` has type `Option Syntax`
- `ts` in `$ts,*` has type `SepArray`

`syntax foo := ...` introduces a new *antiquotation kind* `$e:foo`

`declare_syntax_cat index` introduces a new antiquotation kind `$e:index` and a new *quotation kind*

```
`(index|...)
```

# Macros

Macros are extensible

```
syntax ident "|" term : index
macro_rules
  | `(_big [$op, $idx] ($i:ident | $p) $F) => `(bigop $idx (Enumerable.elems _) (fun $i:ident => ($i:ident, $op, $p, $F)))
#check  $\sum i \mid \text{myPred } i, i+i$ 
#check  $\prod i \mid \text{myPred } i, i+i$ 
```

(*Beyond Notations* supplement,  
<https://github.com/leanprover/lean4/blob/master/tests/lean/run/bigop.lean>)

# Macros

Macros are extensible

```

syntax ident "|" term : index
macro_rules
  | `(_big [$op, $idx] ($i:ident | $p) $F) => `(bigop $idx (Enumerable.elems _) (fun $i:ident => ($i:ident, $op, $p, $F)))
#check  $\sum i \mid \text{myPred } i, i+i$ 
#check  $\prod i \mid \text{myPred } i, i+i$ 

```

(*Beyond Notations* supplement,  
<https://github.com/leanprover/lean4/blob/master/tests/lean/run/bigop.lean>)

The newest macro is tried first, absent specific priorities

```

macro_rules (priority := high) ...

```

# Examples: Simple Web Server

```
import Webserver

GET / => redirect "/greet/stranger"

GET /greet/{name} => write
  <html>
    <h1>Hello, {name}!</h1>
  </html>

def main : IO Unit := do
  let hIn ← IO.stdin
  let hOut ← IO.stdout
  Webserver.run hIn hOut
```

<https://leanprover.github.io/talks/PLDI20>

```
notation "const" e => fun x => e
```

“Of course”  $e$  may not capture  $x$

# Hygiene

```
notation "const" e => fun x => e
```

“Of course” *e* may not capture *x*

```
macro "myDef" ... : command => do  
  `(def helper := ...)
```

“Of course” *helper* may not be captured from outside

Macro *hygiene*: macro expansion should be *capture-avoiding*



# Hygiene

```
notation "const" e => fun x => e
```

“Of course” *e* may not capture *x*

```
macro "myDef" ... : command => do  
  `(def helper := ...)
```

“Of course” *helper* may not be captured from outside

Macro *hygiene*: macro expansion should be *capture-avoiding*

The same principle applies to tactics: variables generated by `induction` are not visible outside it!

# Example: Tactic Macros

```
macro "bonk" x:ident : tactic => `(induction $x <=> simp_all)

example {as bs cs : List  $\alpha$ } : (as ++ bs) ++ cs = as ++ (bs ++ cs) := by
  bonk as
```

# Example: Tactic Macros

```

syntax "trivial" : tactic

macro_rules | `(tactic| trivial) => `(tactic| assumption)
macro_rules | `(tactic| trivial) => `(tactic| rfl)
macro_rules | `(tactic| trivial) => `(tactic| contradiction)
macro_rules | `(tactic| trivial) => `(tactic| apply True.intro)
macro_rules | `(tactic| trivial) => `(tactic| apply And.intro <;> trivial)

```

```

macro:1 x:tactic " <;> " y:tactic:0 : tactic => `(tactic| focus ($x:tactic; allGoals $y:tactic))

```

```

syntax "repeat " tacticSeq : tactic
macro_rules
| `(tactic| repeat $seq) => `(tactic| try (($seq); repeat $seq))

```

<https://github.com/leanprover/lean4/blob/master/src/Init/Tactics.lean>

# Summary: Macros

Macros are syntax-to-syntax translations

- applied iteratively and recursively
- associated with a specific parser and tried in a specific order
- with “well-behaved” (hygienic) name capturing semantics