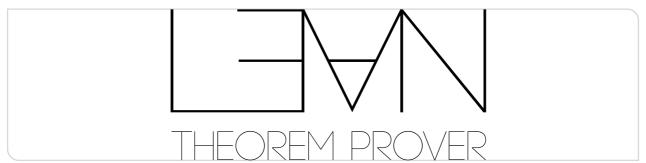




## Theorembeweiserpraktikum

Syntax & Macros

Jakob von Raumer, Sebastian Ullrich | SS 2021



# **Quick Project Remarks**



- prefer induction to manual recursion in tactic proofs
- prefer induction/cases on an inductive predicate to doing so on one of its parameters
- trust in simp/simp\_all (in terminal position!)
  - for goals directly solvable by combinations of rewriting, injection, ex falso, ...

#### We Have Dates!



- final presentations, part 1: Monday, July 12, 14:00–15:30
- final presentations, part 2: Friday, July 15, 14:00–15:30
- please tell us if your group *can't* make one of these dates!





```
syntax "`[Com|" com* "]" : term
macro rules
 | `(`[Com|]) => `(Com.skip)
 | `(`[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?





- parser: ≈ String → Syntax
- macro expansion: Syntax → MacroM Syntax
  - actually interleaved with elaboration
- elaboration
  - terms: Syntax → TermElabM Expr
  - commands: Syntax → CommandElabM Unit
  - universes: Syntax → TermElabM Level
  - tactics: Syntax → TacticM Unit

Every part of the pipeline is extensible by users





```
infix1:65 " + " => HAdd.hAdd -- left-associative
infix:50 " = " => Eq -- non-associative
infixr:80 " ^ " => HPow.hPow -- right-associative
prefix:100 "-" => Neg.neg
postfix:max "-1" => Inv.inv
```





```
infix1:65 " + " => 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
```





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





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

```
notation:50 a:51 " = " b:51 " = " c:51 => a = b \( \Lambda \) b = c
#reduce 1 = 2 = 3 -- 1 = 2 \( \Lambda \) 2 = 3
```

Overlapping notations are parsed with a (local) "longest parse" rule See also https://leanprover.github.io/lean4/doc/syntax.html

# **Syntax**



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

This is just a macro.

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

term is a *syntax category* 

# **Syntax**



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

This is just a macro.

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

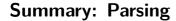
term is a *syntax category* 

```
declare_syntax_cat index
syntax ident : index
syntax ident ":" term : index
syntax term "≤" ident "<" term : index</pre>
syntax "∑" index ", " term : term
```





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





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



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

This is just a macro.



```
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)
```



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

This is just a macro.

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 se:index and a new quotation kind
`(index|...)
```



#### 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 \( \Si \) i | myPred i, i+i
#check \( \Si \) i | myPred i, i+i
```

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



#### 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 \Sigma i | myPred i, i+i #check \Pi i | 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**

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

# Hygiene



```
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!





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

example {as bs cs : List α} : (as ++ bs) ++ cs = as ++ (bs ++ cs) := by
bonk as
```





```
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/Notation.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