Syntax Extensibility in Lean 4

Sebastian Ullrich | 2023/06/21

Towards a Fully Extensible Frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

Towards a Fully Extensible Frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

- extensible syntax from simple mixfix notations to character-level parsing

Towards a Fully Extensible Frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

- extensible syntax from simple mixfix notations to character-level parsing
- extensible semantics from simple syntax sugars to type-aware elaboration

Towards a Fully Extensible Frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

- extensible syntax from simple mixfix notations to character-level parsing
- extensible semantics from simple syntax sugars to type-aware elaboration
- extensible tooling with access to frontend metadata
  - concrete syntax tree
  - elaboration annotations

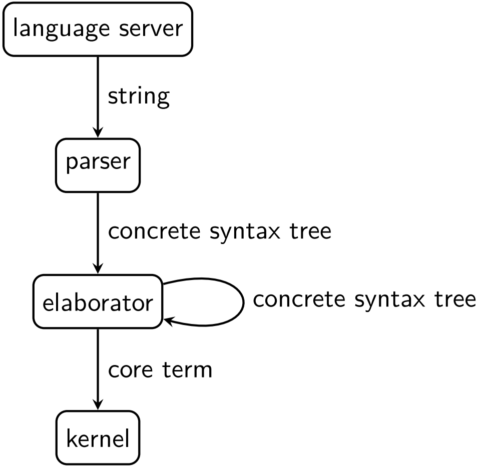
Towards a Fully Extensible Frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

- extensible syntax from simple mixfix notations to character-level parsing
- extensible semantics from simple syntax sugars to type-aware elaboration
- extensible tooling with access to frontend metadata
  - concrete syntax tree
  - elaboration annotations

Non-goal: extensible type theory

Frontend: Overview



Concrete Syntax Tree

provide

- precise source locations
- whitespace and comments
- erroneous input

for

- code editors
- documentation generators
- code formatters
- refactoring tools
- better LaTeX highlighting...

Extensible Concrete Syntax Tree

```
inductive Syntax where
  | atom   (info : SourceInfo) (val : String)
  | ident  (info : SourceInfo) (rawVal : Substring) (val : Name) (preresolved : List Syntax.Preresolved)
  | node   (info : SourceInfo) (kind : SyntaxNodeKind) (args : Array Syntax)
  | missing

inductive SourceInfo where ...

abbrev SyntaxNodeKind := Name
```

```
a -> b
```

```
(Term.arrow `a "->" `b)
```

Parser

- Lean 3: basic lexer, LL(1) recursive descent parser
- Isabelle: basic lexer, Earley parser for arbitrary context-free grammars, delimited terms

Parser

- Lean 3: basic lexer, LL(1) recursive descent parser
- Isabelle: basic lexer, Earley parser for arbitrary context-free grammars, delimited terms
- Lean 4: arbitrary, character-based parser; combinators including Pratt parser and longest-prefix matching

Parser

- Lean 3: basic lexer, LL(1) recursive descent parser
- Isabelle: basic lexer, Earley parser for arbitrary context-free grammars, delimited terms
- Lean 4: arbitrary, character-based parser; combinators including Pratt parser and longest-prefix matching
    - problem: monadic parser combinators allocate like crazy, lexing and parsing should be cached

Parser State

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

structure ParserContext where  -- simplified
  input    : String
  fileName : String
  fileMap  : FileMap
  env      : Environment
  prec     : Nat
  -- ...

structure ParserState where
  pos      : String.Pos
  stxStack : SyntaxStack
  cache    : ParserCache
  errorMsg : Option Error
  -- ...
```

Syntax Stack

```
def nodeFn (k : SyntaxNodeKind) (p : ParserFn) : ParserFn
```

```
nodeFn `Term.arrow (identFn ≫ symbolFn "->" ≫ identFn)
```

```
  [..., `a, "->", `b]
↝ [..., (Term.arrow `a "->" `b)]
```

Token Caching

Cache last “token” read

```
def tokenFn (expected : List String := []) : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then s.mkEOIError expected
  else
    let tkc := s.cache.tokenCache
    if tkc.startPos == i then
      let s := s.pushSyntax tkc.token
      s.setPos tkc.stopPos
    else
      let s := tokenFnAux c s
      updateTokenCache i s
```

Token Caching

Token set not currently extensible except for constant-length symbols

```
private def tokenFnAux : ParserFn := fun c s ⇒
  let input := c.input
  let i     := s.pos
  let curr  := input.get i
  if curr == '"' then
    strLitFnAux i c (s.next input i)
  else if curr == '\'' && getNext input i ≠ '\'' then
    charLitFnAux i c (s.next input i)
  else if curr.isDigit then
    numberFnAux c s
  else if curr == '`' && isIdFirstOrBeginEscape (getNext input i) then
    nameLitAux i c s
  else
    let (_, tk) := c.tokens.matchPrefix input i
    identFnAux i tk .anonymous c s
```

Token Caching

Token set not currently extensible except for constant-length symbols

```
private def tokenFnAux : ParserFn := fun c s ⇒
  let input := c.input
  let i     := s.pos
  let curr  := input.get i
  if curr == '”' then
    strLitFnAux i c (s.next input i)
  else if curr == '\'' && getNext input i ≠ '\'' then
    charLitFnAux i c (s.next input i)
  else if curr.isDigit then
    numberFnAux c s
  else if curr == '`' && isIdFirstOrBeginEscape (getNext input i) then
    nameLitAux i c s
  else
    let (_, tk) := c.tokens.matchPrefix input i
    identFnAux i tk .anonymous c s
```

Plan for unblocking incompatible lexical syntax: store `tokenFnAux` in parser context, making it replaceable

Token Caching

```
def identFn : ParserFn := fun c s ⇒
  let initStackSz := s.stackSize
  let iniPos := s.pos
  let s      := tokenFn ["identifier"] c s
  if !s.hasError && !s.stxStack.back.isIdent then s.mkErrorAt "identifier" iniPos initStackSz else s
```

Deterministic Parsing

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

structure ParserInfo where
  collectTokens : List Token → List Token
  collectKinds  : SyntaxNodeKindSet → SyntaxNodeKindSet
  firstTokens   : FirstTokens

abbrev Token := String
```

following [Swierstra **and** Duponcheel 1996]

Collected token set currently global

Pratt Parser

Token-indexed precedence parsing with local longest-match semantics

```
def prattParser (tables : PrattParsingTables) ... : ParserFn

structure PrattParsingTables where
  leadingTable    : TokenMap (Parser × Nat)  -- e.g. `postfix:100 "-"`
  leadingParsers  : List (Parser × Nat)
  trailingTable   : TokenMap (Parser × Nat)  -- e.g. `infix:65 "-"`
  trailingParsers : List (Parser × Nat)      -- e.g. `syntax term term : term`
```

parses syntax of the shape

```
  leading trailing*
```

where all parsers' precedences are at least the current precedence level

Parser Caching

Like Packrat parsing [Ford 2002]

```
def withCacheFn (parserName : Name) (p : ParserFn) : ParserFn := fun c s ⇒ Id.run do
  let key := ⟨c.toCacheableParserContext, parserName, s.pos⟩
  if let some r := s.cache.parserCache.find? key then
    ...
```

Changing the uncacheable parser context flushes the cache

Pratt Parsing

Lean 3/[Pratt 1973]: *tokens* annotated (globally) with precedence, parsing of hole chains trailing parsers as long as next token precedence higher than hole precedence

```
notation a `-`:65 b:65 := has_sub.sub a b
```

Lean 4: *syntax* annotated with precedence, syntax fits into hole with equal or lower precedence

```
notation:65 a:65 "-" b:66 => Sub.sub a b
notation:max "(" e:0 ")" => e
```

Actual Stdlib Parsing

Syntax categories are Pratt parsers extensible via attributes

```
initialize registerParserCategory `term ...

def term (rbp : Nat := 0) : Parser :=
  categoryParser `term rbp

@[termParser] def anonymousCtor := node `Term.anonymousCtor (
  "⟨" >> sepBy term ", " >> "⟩")

def optIdent : Parser := optional (atomic (ident >> " : "))
@[termParser] def «if»  := node `Term.if (
  "if " >> optIdent >> term >> " then " >> term >> " else " >> term)
```

Actual Stdlib Parsing

Syntax categories are Pratt parsers extensible via attributes

```
declare_syntax_cat term

syntax "⟨" (sepBy term ", ") "⟩" : term

syntax optIdent := (try (ident " : "))?
syntax "if " optIdent term " then " term " else " term : term
```

ParserDescr

A deep embedding of `Parser` used for bootstrapping and to avoid compile-time dependencies

```
inductive ParserDescr where
  | const  (name : Name)
  | unary  (name : Name) (p : ParserDescr)
  | binary (name : Name) (p₁ p₂ : ParserDescr)
  | node (kind : SyntaxNodeKind) (prec : Nat) (p : ParserDescr)
  | cat (catName : Name) (rbp : Nat)
  | parser (declName : Name)
  | ...
```

Embedding Languages in Action

```
declare_syntax_cat jsxElement
declare_syntax_cat jsxChild

syntax jsxAttrName := rawIdent <|> str
syntax jsxAttrVal := str <|> group("{" term "}")
syntax jsxSimpleAttr := jsxAttrName "=" jsxAttrVal
syntax jsxAttrSpread := "[" term "]"
syntax jsxAttr := jsxSimpleAttr <|> jsxAttrSpread

syntax "<" rawIdent jsxAttr* "/>" : jsxElement
syntax "<" rawIdent jsxAttr* ">" jsxChild* "</" rawIdent ">" :
↪  jsxElement

syntax jsxText      : jsxChild
syntax "{" term "}" : jsxChild
syntax jsxElement   : jsxChild

scoped syntax:max jsxElement : term
```

```
macro_rules
  | `(<$n $attrs* />) ⇒ do
    let kind := quote (toString n.getId)
    let attrs ← translateAttrs attrs
    `(Html.element $kind true $attrs #[])
  | `(<$n $attrs* >$children*</$m>) ⇒ ...
```

```
def classInstancesToHtml (className : Name) : HtmlM Html
↪  := do
  pure
    <details «class»="instances">
        <summary>Instances</summary>
        <ul id={s!"instances-list-{className}"}
        ↪  class="instances-list"></ul>
    </details>
```

https://github.com/leanprover/doc-gen4

Conclusion

Arbitrarily extend the Lean grammar using combinator, Pratt, Packrat parsers

Extend Lean with other languages... with some current caveats