Proof Assistant Programming Language Grammar

Lexical Grammar

Keywords

```
theorem  | lemma      | definition | axiom      | proof      | qed
let      | in         | match      | with       | if         | then
else     | forall     | exists     | lambda     | fun        | type
inductive| coinductive| record     | module     | import     | export
admit    | sorry      | exact      | apply      | intro      | intros
destruct | induction  | rewrite    | reflexivity| symmetry   | transitivity
assumption| constructor| case      | split      | left       | right
trivial  | auto       | simp       | ring       | field      | omega
```

Identifiers

```
identifier ::= letter (letter | digit | '_' | '\'')*
letter     ::= 'a'..'z' | 'A'..'Z'
digit      ::= '0'..'9'
```

Literals

```
number     ::= digit+ ('.' digit+)?
string     ::= '"' (char | escape)* '"'
char       ::= any_unicode_except_quote_backslash
escape     ::= '\' ('n' | 't' | 'r' | '\' | '"')
```

Operators and Symbols

```
logical_op ::= '∧' | '∨' | '->' | '<->' | '~'
equality   ::= '=' | '==' | '!=' | '<>'
comparison ::= '<' | '>' | '<=' | '>='
arithmetic ::= '+' | '-' | '*' | '/' | '^' | 'mod'
structural ::= '::' | '++' | '@'
special    ::= '|' | ':' | ';' | ',' | '.' | '?' | '!'
brackets   ::= '(' | ')' | '[' | ']' | '{' | '}'
```

Syntactic Grammar

Program Structure

```
program ::= declaration*

declaration ::=
  | theorem_decl
  | lemma_decl
  | definition_decl
  | axiom_decl
  | inductive_decl
  | record_decl
  | module_decl
  | import_decl
```

Type System

```
type ::=
  | identifier                    (* type constructor *)
  | type '->' type                (* function type *)
  | type '*' type                 (* product type *)
  | type '+' type                 (* sum type *)
  | 'forall' identifier ':' type ',' type  (* dependent type *)
  | '{' identifier ':' type '|' prop '}'   (* subset type *)
  | type identifier*              (* type application *)
  | '(' type ')'                  (* parentheses *)
  | 'Prop'                        (* proposition *)
  | 'Type' number?                (* universe *)
  | 'Set'                         (* computational type *)

prop ::=
  | identifier                    (* atomic proposition *)
  | prop '/\' prop                (* conjunction *)
  | prop '\/' prop                (* disjunction *)
  | prop '->' prop                (* implication *)
  | prop '<->' prop               (* biconditional *)
  | '~' prop                      (* negation *)
  | 'forall' identifier ':' type ',' prop  (* universal quantification *)
  | 'exists' identifier ':' type ',' prop  (* existential quantification *)
  | term '=' term                 (* equality *)
  | term '<' term                 (* ordering *)
  | '(' prop ')'                  (* parentheses *)
```

Terms and Expressions

```
term ::=
    | identifier                (* variable *)
    | number                    (* numeric literal *)
    | string                    (* string literal *)
    | 'fun' identifier+ '=>' term (* lambda abstraction *)
    | term term                 (* application *)
    | 'let' identifier '=' term 'in' term  (* local binding *)
    | 'match' term 'with' match_case* 'end'  (* pattern matching *)
    | 'if' prop 'then' term 'else' term      (* conditional *)
    | term ':' type             (* type annotation *)
    | '(' term ')'              (* parentheses *)
    | constructor term*         (* constructor application *)
    | term '.' identifier       (* field access *)
    | '{' field_assign* '}'     (* record construction *)

match_case ::= '|' pattern '=>' term

pattern ::=
    | identifier                (* variable pattern *)
    | '_'                       (* wildcard *)
    | constructor pattern*      (* constructor pattern *)
    | '(' pattern ')'           (* parentheses *)
    | pattern '::' pattern      (* cons pattern *)

field_assign ::= identifier '=' term (';' field_assign)*
```

Declarations

```
theorem_decl ::=
    'theorem' identifier param* ':' prop proof_body

lemma_decl ::=
    'lemma' identifier param* ':' prop proof_body

definition_decl ::=
    'definition' identifier param* ':' type '=' term

axiom_decl ::=
    'axiom' identifier ':' prop

param ::= '(' identifier+ ':' type ')'

proof_body ::=
    | ':=' proof_term
    | 'proof' tactic* 'qed'
    | 'proof' 'admit'
    | 'proof' 'sorry'
```

Inductive Types

```
inductive_decl ::=
    'inductive' identifier param* ':' type '='
    constructor_decl ('|' constructor_decl)*

constructor_decl ::= identifier ':' type

record_decl ::=
    'record' identifier param* ':' type '=' '{'
    field_decl (';' field_decl)*
    '}'

field_decl ::= identifier ':' type
```

Proof Terms and Tactics

```
proof_term ::=
    | term                          (* direct proof term *)
    | 'exact' term                  (* exact proof *)
    | 'apply' term 'to' term*       (* function application *)
    | 'intro' identifier*           (* introduce assumptions *)
    | 'split'                       (* split conjunction *)
    | 'left' | 'right'              (* choose disjunction *)
    | 'reflexivity'                 (* reflexivity of equality *)
    | 'assumption'                  (* use assumption *)

tactic ::=
    | 'intro' identifier*
    | 'intros'
    | 'apply' term
    | 'exact' term
    | 'split'
    | 'left' | 'right'
    | 'destruct' term ('as' pattern)?
    | 'induction' term ('as' identifier)?
    | 'rewrite' ('<-' | '->') term ('in' identifier)?
    | 'case' term
    | 'reflexivity'
    | 'symmetry'
    | 'transitivity' term?
    | 'assumption'
    | 'trivial'
    | 'auto'
    | 'simp' ('[' identifier* ']')?
    | 'ring'
    | 'field'
    | 'omega'
```

```
  | 'admit'
  | 'sorry'
```

Module System

```
module_decl ::=
  'module' identifier '=' '{' declaration* '}'

import_decl ::=
  'import' module_path ('as' identifier)?

module_path ::= identifier ('.' identifier)*
```

Comments

```
line_comment  ::= '--' any_char* newline
block_comment ::= '(*' (any_char | block_comment)* '*)'
```

Precedence and Associativity

Operator Precedence (highest to lowest)

1. Function application (left associative)
2. Field access `.` (left associative)
3. Unary operators `~` (right associative)
4. Multiplicative `*`, `/`, `mod` (left associative)
5. Additive `+`, `-` (left associative)
6. Cons `::` (right associative)

7. Comparison `=`, `<>`, `<`, `>`, `<=`, `>=` (non-associative)
8. Conjunction `/\` (right associative)
9. Disjunction `\/` (right associative)
10. Implication `->` (right associative)
11. Biconditional `<->` (right associative)

Type Precedence

1. Type application (left associative)
2. Product type `*` (right associative)
3. Sum type `+` (right associative)
4. Function type `->` (right associative)

Grammar Rules Summary

Core Language Features

- **Dependent types**: Types can depend on values
- **Propositions as types**: Curry-Howard correspondence
- **Pattern matching**: Structural decomposition of data
- **Higher-order functions**: Functions as first-class values
- **Polymorphism**: Parametric and ad-hoc polymorphism
- **Module system**: Namespace management and abstraction

Proof Features

- **Interactive tactics**: Step-by-step proof construction
- **Proof terms**: Direct proof objects

- **Automated tactics**: Decision procedures and automation
- **Inductive reasoning**: Structural and well-founded induction
- **Equality reasoning**: Rewriting and substitution

Safety Features

- **Total functions**: All functions must terminate
- **Universe hierarchy**: Prevents paradoxes
- **Strict positivity**: Ensures consistency of inductive types
- **Termination checking**: Structural recursion requirements