Formalising Extensible Grammars in Lean 4

Litu Zou

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2023

Abstract

With the increasing popularity of domain-specific languages (DSL), extensible syntax has been introduced in some modern programming languages like Lean 4 and Racket, allowing developers to dynamically extend program syntax and naturally express their program logic in their custom syntax. However, developers often struggle with infinite parsing and syntactic ambiguity that arise from their ill-designed syntax, because this syntax extension feature is not yet formalised. Based on previous work on the verified PEG parser generator, we introduce a formalised syntax extension specification that not only automatically verifies the wellformedness of the given grammar but also guarantees parsing termination and determinism. To achieve this, we implement a PEG parser generator in Lean 4 that respects parsing termination and determinism, and then we extend the scope of formalism from static PEG grammars to extensible grammars. The parser generator we introduced in this thesis will empower developers to confidently implement their own DSL that always guarantees parsing termination and determinism.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Litu Zou*)

Acknowledgements

I would like to express my sincere gratitude to Dr. Tobias Grosser and Dr. Andrés Goens for their invaluable guidance, patience, and support throughout this project. Their expertise in programming languages, as well as their constructive criticism, have helped me to develop my research ideas and produce a more comprehensive thesis. I would not have gained new perspectives on programming languages without their support.

I would also like to thank my parents for their love, encouragement, and unwavering support throughout my university life. Their patience and understanding have been a source of inspiration and motivation for me.

Table of Contents

Chapter 1

Introduction

Parsing is an algorithmic process to convert a string of tokens into a syntactic structure of a program based on a formal specification called grammar[5]. Parser is a critical piece of infrastructure within the compiler. Its usage ranges from interpreting a programming language to unpacking data from a stream of web traffic. As such, the ambiguity in grammar specification or the lack of input validations for parser implementation would bring vulnerability and error to be exploited. These problems are increasingly prominent since extensible grammars were introduced allowing users to define their own syntax to the existing grammars. A parser should not only ensure the core syntax is well defined but also actively forbid users from introducing ill-designed grammars from the extensible syntax.

Traditionally, a parser for a static grammar is easy to implement and verify because these grammars are typically instances of Context-Free Grammar (CFG) or Parsing Expression Grammar (PEG). There has been some work in the past to address these problems such as the formalising a subset of Parsing Expression Grammar (PEG) to ensure parsing termination[2]. However, the emerging concept of extensible syntax brings another dimension of complexity to the grammar and the formalism on such type of syntax is limited as of today. Consequently, the parser is unable to determine if the user supplies a wellformed grammar or not. An extensible syntax, adopted in some programming languages like Racket[6] and Lean 4[14], provides an interface for users to define the syntax of their domain-specific languages (DSL) that can be parsed from the same built-in parser. Until now, these programming languages only provide limited mechanism to prove the termination and determinism of the user-defined syntax. For example, the following user-defined syntax in Lean 4 results into stack overflow from within the Lean 4 parser.

```
declare_syntax_cat foo
declare_syntax_cat bar
syntax foo : bar
syntax bar : foo
syntax "[bar|" bar "]" : term
-- the following input string results into infinite
parsing from Lean 4 and stack overflow
#check [bar| ]
```

Although the above example of infinite recursion is simple enough to be mitigated, in practice, the programming language designers may easily embed infinite syntactic recursion and syntactic ambiguity into their language designs. If there is no working mechanism to detect these problems in the grammar before parsing, unexpected behaviours and vulnerabilities can be accidentally produced from within the language compiler. Analogous to the ownership system in Rust which mitigates many common programming mistakes and enables "Fearless concurrency"[10] in developing multithreaded programs, a formal check on grammar's termination and determinism mitigates many common mistakes in the programming language design and ultimately empowers the language designers to confidently write a practical and (more importantly) verified programming language that always satisfies the parsing termination and determinism.

With this formal mechanism, a "wellformed" extensible grammar particularly in a theorem proving language like Lean 4 also creates opportunities of verifying grammars of existing programming language. CompCert[11], for example, is an on-going project with a vision to create a formally verified C compiler using computer-assisted mathematical proofs. Most compiler infrastructures of CompCert has been verified. However, its parser is not yet formally verified as of today. If the C language syntax is formalised in an extensible grammar and properties of parsing termination and determinism can be checked at compile time, the parser for C language can also be fully verified. In fact, there is an ongoing research project[1] to use Lean 4 syntax extension to parse C language. If this syntax extension feature is verified, that would essentially prove the completeness of that C language parser.

In this thesis, we define a notion of a wellformed grammar that supports syntax extensions and preserves the parsing termination and determinism. This thesis also introduces an algorithmic mechanism in Lean 4 to check the user-defined syntax without employing the built-in parser.

In the first part (Chapter 3), we define the parsing expression grammar and explain the notions and properties of a wellformed PEG grammar with relevant implementation in Lean 4. We then explain the abstract syntax tree (AST) as an intermediate representation (IR) of the program, formally define the wellformed AST, and explain the sufficient conditions for the wellformness. In the second part (Chapter 4), we formally construct a parser generator function that always guarantees termination using the definitions and properties of wellformed grammar and AST. Finally, we define the extensible grammar specification and describe one possible implementation for extensible grammar. We also include a brief introduction of Lean 4 programming language in the background chapter (Chapter 2) to help readers understand our Lean 4 implementation.

1.1 Related Work

A parser is an important part of a compiler which directly exposes to users' untrusted interactions. Compiler designers also want to know if their grammar is correct during

[1]The project can be found in this GitHub repository https://github.com/opencompl/C-parsing-for-Lean4

the compiler development. As such, there are some work in the past bringing formalism to syntax grammars and their relevant parser implementations.

- Blaudeau and Shankar[2] developed a wellformed subset of the PEG grammar formalism with defined constraints and proved the parsing termination is guaranteed with this grammar. The paper also outlined the algorithmic strategy to check the behaviours of a given syntax grammar without running a parser. The soundness and completeness properties are also proven for the paper's reference (not optimised) PEG parser and packrat parser. This thesis will extend the idea of PEG grammar formalism to the scope of extensible grammar.
- Grimm[8] presented a parser generator implemented in Java that supports easily extensible syntax. The module system introduced in this paper allows developers to easily organise, modify, and compose large-scale syntactic specifications.
- Ullrich and de Moura[14] introduced a novel hygienic macro expansion system specifically built for Lean 4. Unlike more restrictive mechanisms in other theorem proving languages, Lean 4 improved its macro expansion algorithm to resolve the hygienic issues in the tactic languages and enhanced the type-awared elaborations. With this macro expansion system, Lean 4 has successfully removed a significant amount of syntactic redundancy from the existing tactics.
- Hutton and Meijer[9] introduced a monadic perspective of parser combinators. It outlined the fact that many grammars including the complex ones can be simplified into some compositions of parser combinators. The use of monad not only makes the creation of new parser easier but also improves the readability and extensibility of a parser.
- Christiansen[4] introduced an adaptive extension to attribute grammars, which uses the declarative method to describe the semantics of an extensible grammar. It also addresses the syntactic ambiguity in the user-defined syntax.

1.2 Contributions

This thesis mostly follows from the previous work on formalising a wellformed subset of PEG grammars. Blaudeau and Shankar[2] implemented the formalism in a proof assistant language called Prototype Verification System (PVS) and the source code has been published in a GitHub repository[2]. We adopt the same formalism from the previous work and implement it in Lean 4[3]. Although both languages are proof assistant languages, their underlaying logic constructions are different: Lean 4 is based on the calculus of inductive construction[12] (similar to languages like Coq and Agda) whereas PVS is based on the higher-order logic[13]. As such, the definitions and proof methodologies of relevant theorems in Lean 4 are different from those in the PVS implementation. For example, many property definitions in the PVS implementation are encoded as boolean values whereas in our implementation these are defined as inductive propositions.

[2]The repository is called *PVSPackrat* and can be found in https://github.com/SRI-CSL/PVSPackrat
[3]The source code can be found in https://github.com/lituzou/ExtParser

In addition to translating previous work into Lean 4, this thesis also identifies a potential improvement to the PEG grammar formalism which efficiently checks the pattern wellformness of a given grammar. At the later part of the thesis, we extend the formalism from the previous work to extensible grammars by introducing the extensible grammar specification and outlining a possible implementation for the extensible grammar. This thesis is so far the first attempt in applying the formalism of PEG grammars to extensible grammars.

Chapter 2

Background

2.1 The Lean 4 Programming Language

Lean 4 is an open source proof assistant language developed by Microsoft Research. It shares some similarities with other languages like Coq and Agda as they are all based on the dependent types theory and the calculus of inductive construction[12]. It features many distinguishing features such as tactic and syntax extensions[14]. We introduce some syntax of Lean 4 in the remainder of this section. The full documentation and tutorial of the Lean 4 programming language can be found in the official website[1].

2.1.1 Basic syntax

Definitions of variables and functions are started using the `def` keyword. A theorem starts from the `theorem` keyword and follows with the theorem description and its proof.

```
-- Define a natural numbered variable m with value equal to 42
def m : Nat := 42

-- Define a function which adds one to a natural number
def add_one (x : Nat) : Nat := x + 1

-- Evaluate the result of applying add_one to m, which is 43
#eval add_one m

-- A theorem showing (add_one m) is equal to 43
theorem double_m : add_one m = 43 := by simp
```

2.1.2 Inductive types

Inductive types are defined by `inductive` keyword. We use the definition of natural number type `Nat` as an example. A natural number is either zero or a succession of

[1]The official website of Lean 4 is https://leanprover.github.io/documentation/

another natural number.

```
-- Definition of natural number
inductive Nat where
| zero : Nat
| succ : Nat → Nat

-- A natural number 3 uses exactly three succ constructors
example three : Nat := Nat.succ (Nat.succ (Nat.succ Nat.zero))
```

2.1.3 Pattern matching

An inductive type can be pattern matched using the `match` keyword. It should be noted that all cases of induction must be considered in the match expression. We use the addition function of a pair of natural numbers as an example. Also, the patterns in the match are ordered. If multiple patterns are matched to the given arguments, the first matched pattern is used.

```
-- Addition of two natural numbers
def Nat.add (x y : Nat) : Nat := match x, y with
| x, zero => x
| x, succ y => succ (add x y)
```

2.1.4 Tactic

There are two main approaches in constructing proofs. We can directly inductive constructors to construct value of the proof described in the theorem. Alternatively, we can instruct the compiler to produce the proof using a type of commands called *tactic*. Tactic commands apply on the current tactic state of the proof and automatically generate the proof of the goals described in the tactic state. Tactic mode can be entered using the keyword `by`.

```
-- Prove (a + 1) is equal to (Nat.succ a) by the simp tactic
theorem add_one_eq_succ (a : Nat) : a + 1 = Nat.succ a := by simp
```

2.1.5 Syntax extension

One major distinguishing feature of Lean 4 from some theorem proving languages like Agda is the ability to extend and customise the Lean 4 built-in syntax. A syntax node can be declared using the keyword `declare_syntax_cat` and defined using the keyword `syntax`. Syntax cannot be directly evaluated so a set of translation rules can be defined to transform the user-defined syntax into Lean 4 commands. These rules can be declared using the keyword `macro_rules`.

We use the arithmetic expression as an example. For simplicity, an arithmetic expression `arith` can be a number, an addition of a pair of arithmetic expressions, a subtraction of a pair of arithmetic expressions, or a bracket-enclosed expression. We treat the arithmetic expression as a term in Lean 4 and define the equivalent macros for `arith`.

```
-- Definition of arithmetic expression
syntax num : arith
syntax arith "-" arith : arith
syntax arith "+" arith : arith
syntax "(" arith ")" : arith
syntax "[Arith|" arith "]" : term

-- equivalent macro rules for arith terms
macro_rules
| `([Arith| $x:num]) => `($x)
| `([Arith| $x:arith + $y:arith]) => `([Arith| $x] + [Arith| $y])
| `([Arith| $x:arith - $y:arith]) => `([Arith| $x] - [Arith| $y])
| `([Arith| ($x:arith)]) => `([Arith| $x])

#eval [Arith| (15 - 4) + 3] -- returns 14
```

2.2 Parsing Expression Grammar

2.2.1 Grammar definition

A grammar of any kinds (PEG, CFG, etc.) is a tuple $G := (N, T, P)$ consisting sets of non-terminals (N), terminals (T) as well as syntax production rules (P)[3]. In Lean 4, non-terminals are typically referred to syntax categories. A syntax production rule consists of two items, one is the non-terminal $A \in N$ and another is a valid expression $\delta \in \Delta_{N,T}$ with respect to current non-terminals N and terminals T. Here we will use $A \longmapsto \delta$ to express the production rule.

2.2.2 PEG definition

The following inductively defines the set of PEG grammar expression $\Delta_{N,T}$ with respect to non-terminals N and terminals T[7]:

$$
\begin{array}{rcll}
\Delta_{N,T} & := & \varepsilon & \text{empty expression} \\
& | & [\cdot] & \text{any character} \\
& | & [a] & \text{a terminal where } a \in T \\
& | & A & \text{a non-terminal where } A \in N \\
& | & e_1; e_2 & \text{a sequence where } e_1, e_2 \in \Delta_{N,T} \\
& | & e_1 / e_2 & \text{a prioritised choice where } e_1, e_2 \in \Delta_{N,T} \\
& | & e* & \text{a greedy repetition where } e \in \Delta_{N,T} \\
& | & !e & \text{a not-predicate } e \in \Delta_{N,T}
\end{array}
$$

In addition to these basic definitions, for the convenience, we also define some types of PEG expression:

- $[\text{"}s\text{"}]$ the string, equivalent to $[c_1]; [c_2]; \ldots; [c_n]$, where s is a sequence of tokens $s = c_1 c_2 \ldots c_n$.

Parsing expression grammars (PEG) are designed to have similar level of expressiveness to context-free grammar (CFG). Both types of grammars support terminal symbols, non-terminals, expression sequences ($e_1; e_2$) and repetition operators ($e*$). However, unlike the choice operator ($A|B$) in CFG, PEG uses the prioritised choice (A/B) where expression B can be matched only if the match on expression A is failed. In addition, the repetition operator in PEG ($e*$) is greedy as it tries to match as many expressions to e as possible until it fails. The not-predicate operator $!e$ in PEG which looks ahead the tokens without consuming them is also not available in CFG. This allows PEG to match some non-context-free grammars like $A^nB^nC^n$.

One important property of PEG is the fact that PEG grammars are free of ambiguity[7], meaning that there is at most one way to parse the string of tokens with respect to the given grammar. In the later chapter, we use this property to construct an abstract syntax tree (AST) which respects the given grammar and proves the uniqueness of the AST.

2.2.3 Problem of termination and PEG grammar properties

Although PEG grammars are unambiguous, it does not guarantee parsing termination. One obvious non-terminating example is the repetition of empty expression $\varepsilon*$ since it will repeat matching indefinitely without consumption of input tokens. In addition, left recursion may also introduce non-termination such as $A \longmapsto A; [a]$.

To solve this problem, Blaudeau and Shankar[2] define some properties of PEG grammars which are essential for checking parsing termination.

- *fail* means a given grammar expression can fail. This property is needed because the not-predicate succeeds if the expression fails.
- *success without consumption* means a given grammar expression can succeed without consuming any token. Examples include empty expression ε and greedy repetition $e*$.
- *success with consumption* means a given grammar expression can succeed by consuming one or more tokens.

Aside from the definitions of properties, they also introduce an algorithm to compute these properties for any grammar[2]. This algorithm is recursive but they have shown the number of iterations required for this algorithm is finite for finite non-terminals.

With the PEG grammar properties, they also define notions of grammar wellformness by introducing structural and pattern wellformness.

Structural Wellformness Every star operator can only be applied to grammar expressions which cannot succeed without consuming any token. All grammar expressions without star operator are structurally wellformed.

Pattern Wellformness A grammar is pattern wellformed when there exists a particular order of non-terminal such that the PEG expression of each non-terminal can only uses a strictly smaller non-terminal unless at least one token is consumed beforehand.

If a grammar is wellformed in both aspects, Blaudeau and Shankar[2] shows the parser will always terminate on any input and return a unique parse tree upon successful parsing.

Chapter 3

PEG Formalism in Lean 4

3.1 Parsing Expression Grammar

3.1.1 PEG syntax

The PEG syntax is defined in Lean 4 as an inductive datatype shown below. It captures all the primitives required to build a valid PEG expression.

```
inductive PEG (n : Nat) where
    | ε
    | any
    | terminal (c : Char)
    | nonTerminal (vn : Fin n)
    | seq (p1 p2 : PEG n)
    | prior (p1 p2 : PEG n)
    | star (p : PEG n)
    | notP (p : PEG n)
deriving DecidableEq, Repr
```

Note that every PEG type in the implementation is dependent on a natural number `n`, which characterises the finite cardinality of the non-terminal set N. Although we can use any set (including those with infinite cardinality), we can safely assume N is finite since N is usually defined by the programming language developer. In the `nonTerminal` constructor, it uses a finite natural number which is strictly less than `n` to represent a non-terminal in N. Intuitively, one can regard such finite natural number as an index to N in an array form and there exists a bijective relationship between a non-terminal and a finite natural number. In the later sections, we denote N as a finite natural number representing a defined non-terminal.

For practical compilers with static grammars, it usually involves lexical analysis (performed by lexer or tokeniser) to convert a string of individual characters into a string of tokens before feeding it to a parser because it is more performant to parse. However, in the scope of extensible grammar, this is no longer feasible as the set of all possible tokens can be increased from the user-defined syntax and it is impossible for lexer to know the user-defined tokens before parsing. For that reason, we skip lexical

analysis by treating each raw character as a terminal. Although the resulting parser is less performant compared to the practical one, we find it easier to formalise. In our implementation, we use `Char` as the terminal type for simplicity. Other types can also represent the terminal type as long as they satisfy decidable equality property `DecidableEq`.

With this construct, we can easily define the production rules for grammars `GProd` in the implementation. We require a function which takes a non-terminal (which is defined as `Fin n`) and returns the corresponding PEG expression with type `PEG n`. We also require `n` to be a positive natural number because it is not possible to have a grammar with zero non-terminal.

```
structure GProd (n : Nat) where
    pos_n : 0 < n
    f : Fin n → PEG n
```

We also define a subterm proposition for a pair of PEG expressions. For any PEG expressions $P, Q \in \Delta$, P is said to be a subterm of Q (or $P \sqsubseteq Q$) if and only if P is equal to a subtree of Q's expression tree. The rules of subterm are listed below:

- The subterm operator is reflexive, meaning $P \sqsubseteq P$ for all $P \in \Delta$.
- The first and second parts of a sequence expression are the subterms to the original sequence. $e_1 \sqsubseteq e_1; e_2$ and $e_2 \sqsubseteq e_1; e_2$.
- The first and second branches of a prioritised choice expression are the subterms to the original expression. $e_1 \sqsubseteq e_1 / e_2$ and $e_2 \sqsubseteq e_1 / e_2$.
- The expression within greedy repetition and not-predicate is the subterm to the original expression. $e \sqsubseteq e*$ and $e \sqsubseteq !e$.

In the implementation, it is defined as a inductive proposition `PEG.le`. They are clearly reflexive and transitive (the proofs are easy to derive so they are not listed in this paper).

```
inductive PEG.le : PEG n → PEG n → Prop where
    | refl       : le p p
    | seq_left   : le e p1 → le e (.seq p1 p2)
    | seq_right  : le e p2 → le e (.seq p1 p2)
    | prior_left : le e p1 → le e (.prior p1 p2)
    | prior_right : le e p2 → le e (.prior p1 p2)
    | star       : le e p → le e (.star p)
    | notP       : le e p → le e (.notP p)
```

This order relation is also an instance of typeclass `LE.le`. That means one may use $\leq$ operator to express this order relationship between PEG expressions.

3.1.2 PEG Grammar Properties

In the implementation, we uses `PropF`, `Prop0` and `PropS` for *fail*, *success without consumption* and *success with consumption* respectively. In contrast to the PVS implementation which explicitly uses `Bool` type for these properties, this implementation

uses an inductive `Prop` type to represent these properties, making it easier to infer the grammar structure behind the properties. Note that all propositions are mutually inductive propositions.

The full property descriptions of `PropF`, `Prop0` and `PropS` in Lean 4 are shown below:

3.1.2.1 *Fail* Property

The rules for the *fail* property can be summarised in the following cases.

- For any character, failure occurs when there is no more token to be consumed.
- For a terminal symbol, failure occurs when the next token does not match the expected token.
- For a non-terminal symbol, failure occurs exactly when the corresponding PEG expression may fail, because there is exactly one production rule for each non-terminal.
- For a sequence of PEG expressions e_1, e_2, failure occurs when e_1 fails or e_1 succeeds (with or without consumption) but e_2 fails.
- For a prioritised choice of PEG expressions e_1, e_2, failure occurs when both e_1 or e_2 fail to be parse.
- For a not-predicate with respect to a PEG expression e, failure occurs exactly when e succeeds (with or without consumption)

The detailed definition in the implementation is shown below.

```
inductive PropF : GProd n → PEG n → Prop where
    | any : PropF Pexp any
    | terminal : ∀ (c : Char), PropF Pexp (terminal c)
    | nonTerminal : ∀ (vn : Fin n), PropF Pexp (Pexp.f vn) → PropF Pexp
   (nonTerminal vn)
    | seq_F : ∀ (e1 e2 : PEG n), PropF Pexp e1 → PropF Pexp (seq e1 e2)
    | seq_0F : ∀ (e1 e2 : PEG n), Prop0 Pexp e1 → PropF Pexp e2 →
   PropF Pexp (seq e1 e2)
    | seq_SF : ∀ (e1 e2 : PEG n), PropS Pexp e1 → PropF Pexp e2 →
   PropF Pexp (seq e1 e2)
    | prior : ∀ (e1 e2 : PEG n), PropF Pexp e1 → PropF Pexp e2 → PropF
   Pexp (prior e1 e2)
    | notP_0 : ∀ (e : PEG n), Prop0 Pexp e → PropF Pexp (notP e)
    | notP_S : ∀ (e : PEG n), PropS Pexp e → PropF Pexp (notP e)
```

3.1.2.2 *Success Without Consumption* Property

The rules for *success without consumption* property can be summarised in the following cases.

- For an empty expression, it always succeeds without consuming any token.

- For a non-terminal symbol, this property is satisfied exactly when the corresponding PEG expression succeeds without consumption.
- For a sequence of PEG expressions e_1, e_2, it may succeed without consumption when both e_1 and e_2 succeed without consumption.
- For a prioritised choice of PEG expressions e_1, e_2, this properties is satisfied when e_1 succeeds without consumption or e_2 succeeds without consumption in case where e_1 fails.
- For a greedy repetition of a PEG expression e, this property is satisfied if e may fail.
- For a not-predicate with respect to a PEG expression e, it does not consume tokens upon failure in e.

The detailed definition in the implementation is shown below.

```
inductive Prop0 : GProd n → PEG n → Prop where
    | ε : Prop0 Pexp ε
    | nonTerminal : ∀ (vn : Fin n), Prop0 Pexp (Pexp.f vn) → Prop0 Pexp
    (nonTerminal vn)
    | seq : ∀ (e1 e2 : PEG n), Prop0 Pexp e1 → Prop0 Pexp e2 → Prop0
    Pexp (seq e1 e2)
    | prior_0 : ∀ (e1 e2 : PEG n), Prop0 Pexp e1 → Prop0 Pexp (prior e1
    e2)
    | prior_F0 : ∀ (e1 e2 : PEG n), PropF Pexp e1 → Prop0 Pexp e2 →
    Prop0 Pexp (prior e1 e2)
    | star : ∀ (e : PEG n), PropF Pexp e → Prop0 Pexp (star e)
    | notP : ∀ (e : PEG n), PropF Pexp e → Prop0 Pexp (notP e)
```

3.1.2.3 *Success With Consumption* Property

The rules for *success with consumption* property can be summarised in the following cases.

- For any character, it always consumes a token upon success.
- For a terminal symbol, it always consumes a token upon success.
- For a non-terminal symbol, this property is satisfied exactly when the corresponding PEG expression succeeds with consumption.
- For a sequence of PEG expressions e_1, e_2, it may succeed with consumption when both e_1 and e_2 succeed and one of these must succeed with consumption.
- For a prioritised choice of PEG expressions e_1, e_2, this properties is satisfied when e_1 succeeds with consumption or e_2 succeeds with consumption in case where e_1 fails.
- For a greedy repetition of a PEG expression e, this property is satisfied if e may succeed with consumption.

The detailed definition in the implementation is shown below.

```
inductive PropS : GProd n → PEG n → Prop where
    | any : PropS Pexp any
    | terminal : ∀ (c : Char), PropS Pexp (terminal c)
    | nonTerminal : ∀ (vn : Fin n), PropS Pexp (Pexp.f vn) → PropS Pexp
   (nonTerminal vn)
    | seq_S0 : ∀ (e1 e2 : PEG n), PropS Pexp e1 → Prop0 Pexp e2 →
   PropS Pexp (seq e1 e2)
    | seq_0S : ∀ (e1 e2 : PEG n), Prop0 Pexp e1 → PropS Pexp e2 →
   PropS Pexp (seq e1 e2)
    | seq_SS : ∀ (e1 e2 : PEG n), PropS Pexp e1 → PropS Pexp e2 →
   PropS Pexp (seq e1 e2)
    | prior_S : ∀ (e1 e2 : PEG n), PropS Pexp e1 → PropS Pexp (prior e1
   e2)
    | prior_FS : ∀ (e1 e2 : PEG n), PropF Pexp e1 → PropS Pexp e2 →
   PropS Pexp (prior e1 e2)
    | star : ∀ (e : PEG n), PropS Pexp e → PropS Pexp (star e)
```

3.1.3 Grammar Property Computation

3.1.3.1 Motivation

For any given grammar, it is sometimes not possible to decide whether a property holds for a given PEG expression or not. One obvious example is the cyclic reference of non-terminals.

$$A \longmapsto B$$
$$B \longmapsto A$$

According to the definitions of three properties, non-terminal inherits the properties from the corresponding PEG expression. In the example above, we are unable to decide the property by induction, meaning the decidability of these properties cannot be computed.

We take an alternative way of property computation. Instead of proving the decidability of a property, we try to prove the reachability of a property. That means a property of a PEG can be unknown initially and satisfied later as we know more about properties in the grammar. To facilitate this idea, we introduce `Maybe` type in Lean 4 as a wrapper of any property.

```
inductive Maybe (p : α → Prop) (a : α) where
    | found : p a → Maybe p a
    | unknown
```

In this `Maybe` type, you may construct unknown for any properties (without any proofs) or introduce a proof to demonstrate such property is satisfied.

With that, we can formally introduce the predicate of these properties which takes a non-terminal and returns a tuple consisting `Maybe` types of three grammar properties. This predicate is important as it demonstrates the parsing behaviours of a given grammar.

$$P : N \to (\mathbb{M}_F, \mathbb{M}_0, \mathbb{M}_S)$$

We also define $\mathbb{P}$ to be the set of all possible predicates, in which the order of predicates $P, Q \in \mathbb{P}$ is defined as follows

$$P \leq Q \iff \forall A \in N, \begin{cases} P(A)(1) \leq Q(A)(1) \\ P(A)(2) \leq Q(A)(2) \\ P(A)(3) \leq Q(A)(3) \end{cases}$$

It can be easily proved that this order of predicates is both reflexive and transitive.

The computation process can be summarised as follows:

1. Create an unknown predicate where all properties are unknown for any non-terminal.
2. Compute the properties of all non-terminals one by one based on the known properties in the predicate and the inductive definition of the grammar properties.
3. Update the predicate accordingly during computation.
4. Check if the predicate has been updated during computation. If so, go back to step 2 and recompute the predicate. Otherwise, we reach a fixpoint and the computation process can be terminated.

Although this computation process may not discover all the properties of a given grammar, this is good enough because it is sufficient to avoid infinite parsing by considering the reachable known properties of a given grammar.

It is also shown the number of iterations in the computation process is at most $3 \times |N|$. This is because there are $3 \times |N|$ entries in the predicate and it is not possible to change properties from known to unknown during computation (this process is monotonic).

3.1.3.2 Formalising Computation Process - Single Iteration

Several definitions regarding the computational process is defined below:

Definition 3.1.1 *The grammar property function g takes a PEG expression $G \in \Delta$ and a predicate from $\mathbb{P}$ then returns the relevant properties of G.*

$$g : \Delta \times \mathbb{P} \to (\mathbb{M}_F, \mathbb{M}_0, \mathbb{M}_S)$$

This function corresponds to the second step in the computation process. In the implementation, this function is called `g_props`

```
def g_props {Pexp : GProd n} (G : PEG n) (P : PropsTriplePred Pexp) :
    PropsTriple Pexp G
```

Definition 3.1.2 *The property augmentation function ρ takes a non-terminal $A \in N$ and a predicate from $\mathbb{P}$ then returns a new predicate which reflects the property changes for the PEG expression of A.*

$$\rho : N \times \mathbb{P} \to \mathbb{P}$$
$$(A, P) \mapsto \begin{cases} \rho(A,P)(A) = g(P_{exp}(A), P) \\ \rho(A,P)(B) = P(B) \quad (A \neq B) \end{cases}$$

This function corresponds to the third step in the computation process.

To formally show the bound of iteration steps for this algorithm, several theorems are given below.

Theorem 3.1.3 (Order Preservation of g) *Given any $P, Q \in \mathbb{P}$ and any grammar expression $G \in \Delta$, if $P \leq Q$, then we have $g(G,P) \leq g(G,Q)$.*

Proof: We can easily induct on the PEG expression definition of G and apply the hypothesis $P \leq Q$ as well as the definition of function g to each individual cases. For cases with sub-expressions, we can simply apply the same theorem on the sub-expressions. The following code outlines the implementation of the proof.

```
theorem g_props_growth : ∀ {Pexp : GProd n} {G : PEG n} {P Q :
    PropsTriplePred Pexp}, P ≤ Q → g_props G P ≤ g_props G Q := by
intro Pexp G P Q hpq
cases G with
| ε => apply PropsTriple.le_refl
| any => apply PropsTriple.le_refl
| terminal c => apply PropsTriple.le_refl
| nonTerminal vn => exact g_props_growth_nonterminal hpq
-- induct on each sub-expression in the following cases
| seq e1 e2 => ... -- sequence case
| prior e1 e2 => ... -- prioritised choice case
| star e => ... -- greedy repetition case
| notP e => ... -- not-predicate case
```

Although the order of predicates can be preserved via function g, a predicate P may not grow monotonically over function ρ. One obvious example is when a non-terminal can be expanded into an empty expression $P_{exp}(A) = \varepsilon$ and all properties for non-terminal A are proven $P(A) = (known, known, known)$. We have $P \not\leq \rho(A,P)$ because $g(\varepsilon, P) = (unknown, known, unknown)$.

However, by placing a new constraint to the predicate set $\mathbb{P}$, we can ensure ρ is monotonic over the constraint set.

Definition 3.1.4 *A predicate is said to be coherent if the known properties do not contradict with the corresponding PEG expression. We denote this predicate set as coherent predicate set $\mathbb{C}$.*

$$\mathbb{C} = \{P \in \mathbb{P} \quad | \quad \forall A \in N, P \leq \rho(A,P)\}$$

In the implementation, the coherent predicate is encoded as a structure.

```
structure CoherentPred (Pexp : GProd n) where
    pred : PropsTriplePred Pexp
    coherent : ∀ (i : Fin n), pred i ≤ g_props (Pexp.f i) pred
```

With this definition, we can formally implement the property augmentation function ρ, in which the function is called `g_extend`.

```
def g_extend {Pexp : GProd n} (a : Fin n) (P : CoherentPred Pexp) :
    CoherentPred Pexp
```

Several theorems regarding function ρ can be proven with the coherent predicate assumption.

Theorem 3.1.5 (Growth of ρ) *Given any coherent predicate $P \in \mathbb{C}$ and any non-terminal $A \in N$, $P \leq \rho(A,P)$*

```
theorem g_extend_growth1 : ∀ {Pexp : GProd n} (a : Fin n) (P :
    CoherentPred Pexp), P ≤ g_extend a P
```

Proof: This theorem is very easy to prove. We can simply apply the definition of function ρ and the definition of a coherent predicate.

Theorem 3.1.6 (Order Preservation of ρ) *Given two coherent predicates $P,Q \in \mathbb{C}$ and any non-terminal $A \in N$, if $P \leq Q$, we have $\rho(A,P) \leq \rho(A,Q)$.*

```
theorem g_extend_growth2 : ∀ {Pexp : GProd n} (a : Fin n) (P Q :
    CoherentPred Pexp), P ≤ Q → g_extend a P ≤ g_extend a Q
```

Proof: This theorem is also very easy to prove by directly applying the definition of function ρ and the definition of a coherent predicate.

With the property augmentation function for single non-terminal ρ, we formally define the recursive function r describing the entire computational process of grammar properties.

$$r : N \times \mathbb{C} \to \mathbb{C}$$
$$(A,P) \mapsto \begin{cases} r(A+1, \rho(A,P)) & (A \neq \max(N)) \\ \rho(A,P) & (A = \max(N)) \end{cases}$$

In the implementation, this function is called `recompute_props`.

```
def recompute_props {Pexp : GProd n} (a : Fin n) (P : CoherentPred Pexp)
    : CoherentPred Pexp :=
    match Nat.decEq a.val.succ n with
    | isTrue _ => g_extend a P
    | isFalse hne =>
      have _ : n - a.val.succ < n - a.val := Nat.sub_succ_lt_self n
    a.val a.isLt; -- prove termination
      recompute_props (Fin.inbound_succ a hne) (g_extend a P)
termination_by recompute_props a P => n - a.val
```

Similar to function ρ, we prove similar theorems for the recursive function r. This can be broken down in three theorems.

Theorem 3.1.7 (Growth of r) *Given a coherent predicate $P \in \mathbb{C}$ and any non-terminal $A \in N$, we have $P \leq r(A,P)$.*

```
theorem recompute_lemma1 : ∀ {Pexp : GProd n} (a : Fin n) (P :
    CoherentPred Pexp), P ≤ recompute_props a P
```

Proof: We induct on non-terminal A. In the base case where $A = \max(N)$, it directly follows from the growth of ρ. Given an induction hypothesis where $P' \leq r(A+1,P')$

for all $P' \in \mathbb{C}$ and $A \neq \max(N)$, we apply the transitive property between $P \leq \rho(A,P)$ and $\rho(A,P) \leq r(A+1, \rho(A,P))$. These two inequalities follow from the growth of ρ and the inductive hypothesis.

Theorem 3.1.8 (Order Preservation of r) *Given two coherent predicates $P,Q \in \mathbb{C}$ and any non-terminal $A \in N$, if $P \leq Q$, we have $r(A,P) \leq r(A,Q)$.*

```
theorem recompute_lemma2 : ∀ {Pexp : GProd n} (a : Fin n) (P Q :
    CoherentPred Pexp), P ≤ Q → recompute_props a P ≤ recompute_props a
    Q
```

Proof: Once again, we induct on non-terminal A. In the base case where $A = \max(N)$, it directly follows from the order preservation of ρ. Given an induction hypothesis where $P' \leq Q' \implies r(A+1,P') \leq r(A+1,Q')$ for all $P', Q' \in \mathbb{C}$ and $A \neq \max(A)$, we first show $\rho(A,P) \leq \rho(A,Q)$ using the order preservation of ρ and directly apply the induction hypothesis.

Theorem 3.1.9 (Order of r over non-terminal) *Given a coherent predicate $P \in \mathbb{C}$ and any non-terminal $A \neq \max(N)$, we have $r(A+1,P) \leq r(A,P)$*

```
theorem recompute_lemma3 : ∀ {Pexp : GProd n} (a : Fin n) (P :
    CoherentPred Pexp), (hne : ¬(a.val.succ = n)) → recompute_props
    (Fin.inbound_succ a hne) P ≤ recompute_props a P
```

Proof: We first rewrite the right hand side of the inequality according to the definition of function r into

$$r(A+1,P) \leq r(A+1, \rho(A,P))$$

Then we apply both growth of ρ and order preservation of r to prove the above inequality.

3.1.3.3 Formalising Computation Process - Multiple Iterations

The function r basically augments a given predicate over a range of non-terminal up to the cardinality of non-terminals. Intuitively, $r(0,P)$ augments a given coherent predicate over all the non-terminals. By applying r multiple times with $A = 0$, we can eventually reach a point where P is no longer updated. We call such set of resulting predicates as fixpoints $\mathbb{F}$.

$$\mathbb{F} = \{P \in \mathbb{C} \quad | \quad P = r(0,P)\}$$

In the implementation, this fixpoint is characterised as a structure:

```
structure Fixpoint (Pexp : GProd n) where
    coherent_pred : CoherentPred Pexp
    isFixed : recompute_props (Fin.mk 0 Pexp.pos_n) coherent_pred =
    coherent_pred
```

We claim that a fixpoint can be reached from any coherent predicate by taking at most $3 \times |N|$ steps. This is because the number of known properties is non-decreasing via function r according to the growth of r and the maximum number of known properties for a predicate with non-terminals N is exactly $3 \times |N|$.

We formally define the terminating function `compute_props` in the implementation using process given above.

```
def compute_props {n : Nat} {Pexp : GProd n} (P : CoherentPred Pexp) :
    Fixpoint Pexp :=
    let fin_zero : Fin n := Fin.mk 0 Pexp.pos_n;
    let new_P : CoherentPred Pexp := recompute_props fin_zero P;
    have le_pred : P ≤ new_P := recompute_lemma1 fin_zero P;
    match Fin.decEq P.count_found new_P.count_found with
    | isTrue h => {coherent_pred := P, isFixed := by {
        apply Eq.symm;

        apply CoherentPred.eq_of_le_with_same_count P new_P le_pred h;
      }}
    | isFalse h =>
      have _ : 3 * n + 1 - (new_P.count_found).val < 3 * n + 1 -
    (P.count_found).val := by
      {
        have g : P.count_found < new_P.count_found := by
        {
          match Nat.eq_or_lt_of_le (CoherentPred.count_growth le_pred)
    with
          | Or.inl g => exact absurd (Fin.eq_of_val_eq g) h;
          | Or.inr g => exact g
        }
        have lem : ∀ {a b c : Nat}, b < a → c < a → b < c → a - c < a
    - b := by
        {
          intro a b c hba hca hbc;
          induction hbc with
          | refl => rw [Nat.sub_succ]; apply Nat.pred_lt; apply
    Nat.sub_ne_zero_of_lt hba;
          | step _ ih =>
            rw [Nat.sub_succ]; apply Nat.lt_trans; apply Nat.pred_lt;
    apply Nat.sub_ne_zero_of_lt;
            apply Nat.lt_of_succ_lt hca; apply ih; exact
    Nat.lt_of_succ_lt hca;
        }
        apply lem;
        exact P.count_found.isLt;
        exact new_P.count_found.isLt;
        exact g;
      }
      compute_props new_P
termination_by compute_props n Pexp P => 3 * n + 1 - P.count_found
```

We finish up by stating the important theorem about a fixpoint.

Theorem 3.1.10 (Fixpoint Theorem) *Recomputing the properties on a fixpoint always return the same predicate. In mathematical terms, for any fixpoint $P \in \mathbb{F}$ and any non-terminal A,*

$$P = \rho(A, P)$$

```
theorem Fixpoint.no_growth : ∀ {Pexp : GProd n} (a : Fin n) (P :
    Fixpoint Pexp), P.coherent_pred = g_extend a P.coherent_pred
```

Proof: This can be done by proving $P \leq \rho(A,P)$ and $\rho(A,P) \leq P$. The first inequality simply follows from the growth of ρ.

The second inequality is less trivial. We can prove some inequalities $\rho(A,P) \leq r(A,P)$ and $r(A,P) \leq r(0,P)$, then the second inequality can be restructured into a chain the inequalities.

$$\rho(A,P) \leq r(A,P) \leq r(0,P) = P$$

where the last inequality follows from the definition of a fixpoint P.

3.1.4 Wellformed Grammar

With the essential properties defined in previous sections, we can define the **wellformness** of a given grammar.

Definition 3.1.11 *A PEG grammar is **wellformed** when its expression structure guarantees the parsing termination.*

Such wellformness can be achieved by enforcing two properties on the grammars, which are structural and pattern wellformness.

Structural Wellformness concerns only on the use of *star* operator in the PEG expression. Every star operator can only be applied to grammar expressions which cannot succeed without consuming any token. All grammar expressions without star operator are structurally wellformed.

The definition of structural wellformness in the implementation is shown below:

```
inductive StructuralWF (Pexp : GProd n) : PEG n → Prop where
    | ε : StructuralWF Pexp ε
    | any : StructuralWF Pexp any
    | terminal : ∀ (c : Char), StructuralWF Pexp (terminal c)
    | nonTerminal : ∀ (vn : Fin n), StructuralWF Pexp (nonTerminal vn)
    | seq : ∀ (e1 e2 : PEG n), StructuralWF Pexp e1 → StructuralWF Pexp
   e2 → StructuralWF Pexp (seq e1 e2)
    | prior : ∀ (e1 e2 : PEG n), StructuralWF Pexp e1 → StructuralWF
   Pexp e2 → StructuralWF Pexp (prior e1 e2)
    | star : ∀ (e : PEG n), StructuralWF Pexp e → ¬IsKnown (getProp0
   Pexp e) → StructuralWF Pexp (star e)
    | notP : ∀ (e : PEG n), StructuralWF Pexp e → StructuralWF Pexp
   (notP e)
```

Pattern Wellformness concerns on the relation of non-terminals in the grammar. A grammar is pattern wellformed when there exists a particular order of non-terminal such that the PEG expression of each non-terminal can only uses a strictly smaller non-terminal unless at least one token is consumed beforehand.

The definition of pattern wellformness in the implementation is shown below:

```
inductive PatternWF {p : Fin n → Fin n} (Pexp : GProd n) (σ : Bijective
    p) (A : Fin n) : PEG n → Prop where
    | ε : PatternWF Pexp σ A ε
    | any : PatternWF Pexp σ A any
    | terminal : ∀ (c : Char), PatternWF Pexp σ A (terminal c)
    | nonTerminal : ∀ (B : Fin n), p B < p A → PatternWF Pexp σ A
    (nonTerminal B)
    | seq : ∀ (e1 e2 : PEG n), PatternWF Pexp σ A e1 → (IsKnown
    (getProp0 Pexp e1) → PatternWF Pexp σ A e2) → PatternWF Pexp σ A
    (seq e1 e2)
    | prior : ∀ (e1 e2 : PEG n), PatternWF Pexp σ A e1 → PatternWF Pexp
    σ A e2 → PatternWF Pexp σ A (prior e1 e2)
    | star : ∀ (e : PEG n), PatternWF Pexp σ A e → PatternWF Pexp σ A
    (star e)
    | notP : ∀ (e : PEG n), PatternWF Pexp σ A e → PatternWF Pexp σ A
    (notP e)
```

The essential part here is the strictly less condition in the non-terminal case, this ensures only smaller non-terminal (for a particular order) is employed in the non-terminal expression.

Another interesting case is the sequence operator over two PEG expression. When the first expression of the sequence may succeed without consuming tokens, we require both expressions to be pattern wellformed. Otherwise, we only require the first expression to be pattern wellformed because by then the sequence is guaranteed to succeed with token consumption.

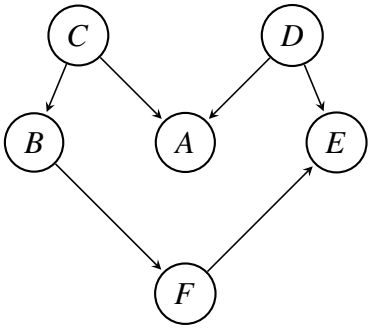

Figure 3.1: Example of the directed acyclic graph. The topological order is *CDBFAE*

One graphical intuition of pattern wellformness is by building a dependency graph for non-terminals. We can build a directed graph consisting nodes representing all non-terminals. There is an directed edge from node A to node B if and only if the PEG expression for non-terminal A may employ non-terminal B without consuming any token. This edge also constitutes an order between non-terminals A and B, meaning B is strictly smaller than A for a particular order.

A pattern wellformness means a particular order is consistent with the directed graph representation. Since an order is transitive, the graph must not contain a cycle. If there exists a cycle in the directed graph, that means there exists an input which results into infinite parsing.

The order of non-terminals can be found by simply enumerating all possible permutations of finite non-terminals. However, there is a more efficient way of finding such order. We can first generate a dependency graph for non-terminals, then use topological sort to find the order if the graph is acyclic (see Figure 3.1). The exact implementation of such algorithm is beyond the scope of this project.

A **wellformed** grammar must be structurally wellformed and pattern wellformed of a particular order. The definition of wellformed grammar in the implementation is shown below:

```
structure Wellformed_GProd (n : Nat) where
    Pexp : GProd n
    p : Fin n → Fin n
    σ : Bijective p
    structural : StructuralWF_GProd Pexp
    pattern : PatternWF_GProd Pexp σ
```

Any grammar with this grammar structure is guaranteed to have parsing termination.

3.2 Abstract Syntax Tree

In a compiler, abstract syntax tree (AST) is the first intermediate representation (IR) after parsing. Depending on different compiler implementations, the abstract syntax tree may have different formats. In this project, we choose to define the abstract syntax tree that inherits all the PEG expression operators and represents the full trace of parsing (whether this parse is successful or not). Defining AST in this way not only makes such representation as generic as possible but also allows us to explicitly observe and prove the relevant properties of the parser based on the full computational path encoded in the AST.

To begin, we first define the basic structure of an AST.

3.2.1 Definition of Abstract Syntax Tree

Because an AST must fully reflect the syntactic structure of the input which is finite in nature, an AST depends on the type of terminal T, the type of non-terminal N and an size of an input string $b \in \mathbb{N}$. Every tree node regardless of the node type contains the start and end of the input string $s, e \in [0, b)$ consumed by the parse tree. The range of input string follows inclusion start exclusion end rule. The constructors of an AST is listed below:

- $skip(s, e, G)$ where $G \in \Delta$. **Skip tree** represents the part of an grammar which is skipped during parsing. An example is the prioritised choice. If the first branch is successfully parsed, the second branch is then skipped. Currently, there is no condition for the start and end of the input string for the skip tree, but we later require $s = e$ because skip tree do not consume any input. We also define the set of skip trees as $\mathcal{S}$.
- $\varepsilon(s, e)$. **Empty expression tree** represents the empty expression in PEG.

- $any(s,e,x)$. **Any character tree** represents the any operator in PEG and $x \in T$ is the consumed character.
- $terminal(s,e,a,x)$. **Terminal tree** represents the terminal operator. $a \in T$ is the expected character and $x \in T$ is the consumed character. In the successful scenario, the tree must have $a = x$. In the failed case, we store why this parse fails by storing $a \neq x$.
- $nonTerminal(s,e,A,T)$. **Non-terminal Tree** represents the non-terminal operator. $A \in N$ is the expected non-terminal to be consumed and $T \notin \mathcal{S}$ is the parse tree for non-terminal A.
- $seq(s,e,T_1,T_2)$. **Sequence expression tree** represents the sequence operator in PEG where T_1 and T_2 are the parse trees consumed in sequence. It should be noted that $T_1 \notin \mathcal{S}$, but T_2 has no such constraint because the first part of a sequence can be failed.
- $prior(s,e,T_1,T_2)$. **Prioritised choice tree** represents the prior operator in PEG where T_1 and T_2 are the parse trees for each branch. It should be noted that $T_1 \notin \mathcal{S}$ because the first branch should not be skipped in favour of the second branch.
- $star(s,e,T_0,T_S)$. **Greedy repetition tree** represents the greedy repetition operator in PEG where $T_0 \notin \mathcal{S}$. To maintain the structure of greedy repetition, we must enforce T_S to be either greedy repetition tree or skip tree. We define the set of greedy repetition trees as $\mathcal{G}$ and enforce $T_S \in \mathcal{S} \cup \mathcal{G}$.
- $notP(s,e,T)$. **Not-predicate tree** represents the not-predicate operator in PEG where $T \notin \mathcal{S}$.

In the implementation, the AST definition is encoded as a base inductive datatype called `PreAST` and a proposition `PreAST.IsValid`. Some helper propositions are included to check if a tree is a skip tree or a greedy repetition tree (this corresponds to $\mathcal{S}$ and $\mathcal{G}$).

The base datatype `PreAST` has the definition shown below

```
inductive PreAST (n : Nat) (b : Nat) where
    | skip (s e : Fin b) (G : PEG n)
    | ε (s e : Fin b)
    | any (s e : Fin b) (x : Char)
    | terminal (s e : Fin b) (a x : Char)
    | nonTerminal (s e : Fin b) (A : Fin n) (T : PreAST n b)
    | seq (s e : Fin b) (T1 T2 : PreAST n b)
    | prior (s e : Fin b) (T1 T2 : PreAST n b)
    | star (s e : Fin b) (T0 TS : PreAST n b)
    | notP (s e : Fin b) (T : PreAST n b)
```

Relevant propositions for a valid AST is shown below

```
inductive PreAST.SkipPreAST : PreAST n b → Prop where
    | skip : SkipPreAST (.skip s e G)

inductive PreAST.StarPreAST : PreAST n b → Prop where
    | star : StarPreAST (.star s e T0 TS)
```

```
inductive PreAST.IsValid : PreAST n b → Prop where
    | skip : IsValid (.skip s e G)
    | ε : IsValid (.ε s e)
    | any : IsValid (.any s e x)
    | terminal : IsValid (.terminal s e a x)
    | nonTerminal : IsValid sub_T → ¬SkipPreAST sub_T → IsValid
    (.nonTerminal s e A sub_T)
    | seq : IsValid T1 → IsValid T2 → ¬SkipPreAST T1 → IsValid (.seq
    s e T1 T2)
    | prior : IsValid T1 → IsValid T2 → ¬SkipPreAST T1 → IsValid
    (.prior s e T1 T2)
    | star : IsValid T0 → IsValid TS → ¬SkipPreAST T0 → (SkipPreAST
    TS ∨ StarPreAST TS) → IsValid (.star s e T0 TS)
    | notP : IsValid sub_T → ¬SkipPreAST sub_T → IsValid (.notP s e
    sub_T)
```

3.2.2 Outcome of Abstract Syntax Tree

An AST may encode both successful branch and failure branch. One example is the prioritised choice tree where the first branch failed but the second branch succeeds. We must define both success and failure scenarioes for each AST constructor. For each constructor, the success and failure conditions are listed below:

- $skip(s,e,G)$. Skip tree does not directly constitute either success and failure.
- $\varepsilon(s,e)$. The tree is successful if no token is consumed and $s = e$. There is no failure scenario for this tree.
- $any(s,e,x)$. The tree is successful if $s+1 = e$ or failed if $s = e = b-1$ (meaning nothing is consumed at the end of input string).
- $terminal(s,e,a,x)$. The tree is successful if $s+1 = e$ and $a = x$. This tree has two failure scenarioes. The first case is when it reaches the end of input string $s = e = b-1$ and the second case is the expected token does not match the consumed token ($s+1 = e$ and $a \neq x$).
- $nonTerminal(s,e,A,T)$. The tree is successful if T is successful. Similarly, the tree is failed if T is failed.
- $seq(s,e,T_1,T_2)$. The tree is successful if T_1 and T_2 are successful. The tree is failed if either T_1 is failed or successful T_1 is followed by failed T_2.
- $prior(s,e,T_1,T_2)$. The tree is successful if either T_1 is successful or failed T_1 is followed by successful T_2. The tree is failed if both T_1 and T_2 are failed.
- $star(s,e,T_0,T_S)$. The tree is successful if T_0 is failed or both T_0 and T_S are successful. There is no failure scenario for this tree.
- $notP(s,e,T)$. The tree is successful if T is failed. The tree is failed if T is successful.

In the implementation, the successful and failed ASTs are encoded as two mutual propositions over `PreAST`.

```
mutual
inductive PreAST.SuccessAST : PreAST n b → Prop where
    | ε : s = e → SuccessAST (.ε s e)
    | any : s.inbound_succ h = e → SuccessAST (.any s e x)
    | terminal : s.inbound_succ h = e → a = x → SuccessAST (.terminal
   s e a x)
    | nonTerminal : SuccessAST T → SuccessAST (.nonTerminal s e A T)
    | seq : SuccessAST T1 → SuccessAST T2 → SuccessAST (.seq s e T1 T2)
    | prior_S : SuccessAST T1 → SuccessAST (.prior s e T1 T2)
    | prior_FS : FailureAST T1 → SuccessAST T2 → SuccessAST (.prior s
   e T1 T2)
    | star_F : FailureAST T0 → SuccessAST (.star s e T0 TS)
    | star_SS : SuccessAST T0 → SuccessAST TS → SuccessAST (.star s e
   T0 TS)
    | notP : FailureAST T → SuccessAST (.notP s e T)

inductive PreAST.FailureAST : PreAST n b → Prop where
    | any : s = e → Fin.IsMax e → FailureAST (.any s e x)
    | terminal_mismatch : s.inbound_succ h = e → a ≠ x → FailureAST
   (.terminal s e a x)
    | terminal_empty : s = e → Fin.IsMax e → FailureAST (.terminal s e
   a x)
    | nonTerminal : FailureAST T → FailureAST (.nonTerminal s e A T)
    | seq_F : FailureAST T1 → FailureAST (.seq s e T1 T2)
    | seq_SF : SuccessAST T1 → FailureAST T2 → FailureAST (.seq s e T1
   T2)
    | prior : FailureAST T1 → FailureAST T2 → FailureAST (.prior s e
   T1 T2)
    | notP : SuccessAST T → FailureAST (.notP s e T)
end
```

To ensure no AST is both successful and failed, we prove the following theorem that successful ASTs and failed ASTs are mutually exclusive.

Theorem 3.2.1 *Assuming an AST is valid, if that AST is successful, it cannot be failed.*

```
theorem PreAST.SuccessAST.ne_failure : ∀ {T : PreAST n b},
    PreAST.IsValid T → SuccessAST T → ¬FailureAST T
```

Proof: This theorem is simple to prove. We basically induct on the constructors of the successful AST and use the definition of failed AST to create contradiction.

Corollary 3.2.1.1 *Assuming an AST is valid, if that AST is failed, it cannot be successful.*

```
theorem PreAST.FailureAST.ne_success : ∀ {T : PreAST n b},
    PreAST.IsValid T → FailureAST T → ¬SuccessAST T
```

Proof: This corollary is the contrapositive to the previous theorem.

3.2.3 Wellformed Abstract Syntax Tree

With the notion of failure and success for the ASTs, we can say that AST is meaningful if such AST is either successful or failed. In the later session, we will prove the AST produced from the parser is always meaningful.

Definition 3.2.2 *A meaningful AST is a AST that can be determined to be either successful or failed.*

```
def PreAST.IsMeaningful (T : PreAST n b) : Prop := SuccessAST T ∨
    FailureAST T
```

A wellformed abstract syntax tree must corresponds the real computational path from the input. With this objective in mind, we impose more conditions to ensure the abstract syntax tree is wellformed. Depending on the different AST constructors, the conditions for a wellformed tree is listed below.

- $skip$. Skip tree along is not wellformed.
- $\varepsilon, any, terminal$. Empty expression trees, any character trees and terminal trees are wellformed if they are meaningful.
- $nonTerminal(s,e,T)$. Non-terminal trees are wellformed if the subtree T is wellformed and the declared bound is equal to the bound of subtree, meaning $s = s(T)$ and $e = e(T)$.
- $seq(s,e,T_1,T_2)$. For sequence expression trees to be wellformed, the first subtree T_1 must be wellformed and the bounds of two subtrees T_1 and T_2 must form a partition between s and e (i.e. $s = s(T_1)$, $e(T_1) = s(T_2)$ and $e(T_2) = e$). In addition, if T_1 is a failed tree, the second part of the sequence is never visited and the second subtree must be a skip tree $T_2 \in \mathcal{S}$ which has an empty bound $s(T_2) = e(T_2)$.
- $prior(s,e,T_1,T_2)$. For prioritised choice trees to be wellformed, the first subtree T_1 must be wellformed and both subtrees T_1 and T_2 must start from s (i.e. $s = s(T_1)$ and $s = s(T_2)$). In addition, if the first branch of the prioritised choice is satisfied, the first subtree must be successful and ended at e (i.e. $e = e(T_1)$) whereas the second subtree must be a skip tree with empty bound. If the first branch is not satisfied, the first subtree must be a failed tree whereas the second subtree must be wellformed and ended at e (i.e. $e = e(T_2)$).
- $star(s,e,T_0,T_S)$. For greedy repetition trees to be wellformed, the subtree T_0 must be wellformed and the bounds of two subtrees T_1 and T_2 must form a partition between s and e (i.e. $s = s(T_0)$, $e(T_0) = s(T_S)$ and $e(T_S) = e$). In addition, if the subtree T_0 is successful, the recursive subtree T_S must be wellformed. If the subtree T_0 is failed, the recursive subtree must be a skip $T_S \in \mathcal{S}$ and the entire tree must have empty bound $s = e$.
- $notP(s,e,T)$. For not-predicate trees to be wellformed, the subtree must be wellformed and have equal starting point $s = s(T)$. Since not-predicate does not consume any token, we require $s = e$.

In the implementation, the wellformness is defined as an inductive proposition over a base AST `PreAST`. The definition is shown below:

```
inductive PreAST.IsWellformed : PreAST n b → Prop where
    | ε :           IsMeaningful (.ε (n := n) (b := b) s e)
                    → IsWellformed (.ε (n := n) (b := b) s e)
    | any :         IsMeaningful (.any (n := n) (b := b) s e x)
                    → IsWellformed (.any (n := n) (b := b) s e x)
    | terminal :    IsMeaningful (.terminal (n := n) (b := b) s e a x)
                    → IsWellformed (.terminal (n := n) (b := b) s e a x)
    | nonTerminal : ∀ {sub_T : PreAST n b},
                    s = sub_T.start → e = sub_T.end
                    → IsWellformed sub_T
                    → IsWellformed (.nonTerminal s e A sub_T)
    | seq_F :   ∀ {T1 T2 : PreAST n b},
                s = T1.start → T1.end = T2.start → e = T2.end
                → T2.start = T2.end
                → IsWellformed T1 → FailureAST T1 → SkipPreAST T2
                → IsWellformed (.seq s e T1 T2)
    | seq_S :   ∀ {T1 T2 : PreAST n b},
                s = T1.start → T1.end = T2.start → e = T2.end
                → IsWellformed T1 → SuccessAST T1 → IsWellformed T2
                → IsWellformed (.seq s e T1 T2)
    | prior_S : ∀ {T1 T2 : PreAST n b},
                s = T1.start → s = T2.start → s = T2.end → e = T1.end
                → IsWellformed T1 → SuccessAST T1 → SkipPreAST T2
                → IsWellformed (.prior s e T1 T2)
    | prior_F : ∀ {T1 T2 : PreAST n b},
                s = T1.start → s = T2.start → e = T2.end
                → IsWellformed T1 → FailureAST T1 → IsWellformed T2
                → IsWellformed (.prior s e T1 T2)
    | star_S :  ∀ {T0 TS : PreAST n b},
                s = T0.start → T0.end = TS.start → e = TS.end
                → IsWellformed T0 → SuccessAST T0 → IsWellformed TS
                → IsWellformed (.star s e T0 TS)
    | star_F :  ∀ {T0 TS : PreAST n b},
                s = T0.start → T0.end = TS.start → TS.start = TS.end
                → s = e
                → IsWellformed T0 → FailureAST T0 → SkipPreAST TS
                → IsWellformed (.star s e T0 TS)
    | notP :    ∀ {sub_T : PreAST n b},
                s = e → s = sub_T.start
                → IsWellformed sub_T
                → IsWellformed (.notP s e sub_T)
```

To verify the definition of a wellformed tree, we prove some simple theorems regarding wellformed trees.

Theorem 3.2.3 (Wellformed trees are meaningful) *A wellformed tree is either a successful AST or a failed AST.*

```
theorem PreAST.valid_and_wellformed_implies_meaningful :
    ∀ {T : PreAST n b}, IsValid T → IsWellformed T → IsMeaningful T
```

Proof: This theorem is easy to prove. We basically induct on the definition of a wellformed AST and apply constructors of successful or failed trees accordingly. In some cases with wellformed subtrees, we apply the same theorems to these subtrees.

Theorem 3.2.4 *A wellformed tree must have a non-negative bound $s \leq e$.*

```
theorem PreAST.valid_and_wellformed_implies_start_le_end :
    ∀ {T : PreAST n b}, IsValid T → IsWellformed T → T.start ≤ T.end
```

Proof: This theorem is also easy to prove. We basically induct on the definition of a wellformed AST and uses the bound conditions from the constructor to form the inequality. In some cases with wellformed subtrees, we apply the same theorems to these subtrees.

With the above definitions and theorems, we can formally defined a wellformed tree in the implementation. An AST is considered wellformed if and only if the base datatype `PreAST` is both valid and wellformed. These properties are encoded in a structure.

```
structure AST (n : Nat) (b : Nat) where
    T : PreAST n b
    valid_T : PreAST.IsValid T
    wf_T : PreAST.IsWellformed T
```

3.3 Parser Generator

In the previous sections, we have defined a notion of wellformed grammars and wellformed ASTs. As discussed in the introduction, parsing is an algorithmic process converting a string of tokens into a wellformed abstract syntax tree that respects both the grammar and the input consistency. This section verifies that relationship and formally define a parser generator for any wellformed PEG grammar.

3.3.1 Parsing as a relation between Abstract Syntax Tree, Grammar and Input

To prove a wellformed AST respects the given grammar and input, two additional properties are introduced to an AST.

- A wellformed AST is said to be *true to a grammar G* if the wellformed tree can reproduce the same grammar G.
- A wellformed AST is said to be *true to an input I* if the token stored in the tree corresponds to the token in input I.

3.3.1.1 True to Grammar

The definition of *true to grammar* for an AST is very straight-forward. We simply map each tree node to the corresponding PEG expression operator. One special case is the **skip tree** because this tree can be applied to any grammar (that will never be visited).

```
inductive PreAST.TrueToGrammar : PreAST n b → GProd n → PEG n → Prop
    where
    | skip : TrueToGrammar (.skip s e G) Pexp G
    | ε : TrueToGrammar (.ε s e) Pexp .ε
    | any : TrueToGrammar (.any s e x) Pexp .any
    | terminal : TrueToGrammar (.terminal s e a x) Pexp (.terminal a)
    | nonTerminal : TrueToGrammar T Pexp (Pexp.f A) → TrueToGrammar
    (.nonTerminal s e A T) Pexp (.nonTerminal A)
    | seq : TrueToGrammar T1 Pexp e1 → TrueToGrammar T2 Pexp e2 →
    TrueToGrammar (.seq s e T1 T2) Pexp (.seq e1 e2)
    | prior : TrueToGrammar T1 Pexp e1 → TrueToGrammar T2 Pexp e2 →
    TrueToGrammar (.prior s e T1 T2) Pexp (.prior e1 e2)
    | star : TrueToGrammar T0 Pexp e0 → TrueToGrammar TS Pexp (.star
    e0) → TrueToGrammar (.star s e T0 TS) Pexp (.star e0)
    | notP : TrueToGrammar T Pexp e0 → TrueToGrammar (.notP s e T) Pexp
    (.notP e0)

def AST.TrueToGrammar : AST n b → GProd n → PEG n → Prop := fun T =>
    PreAST.TrueToGrammar T.T
```

3.3.1.2 True to Input

The definition of *true to input* for an AST is also very straight-forward. We only pay particular attention to the constructors $any(s,e,x)$ and $terminal(s,e,a,x)$ where we require the input token at the starting index s must match the consumed token x.

```
def Input (b : Nat) := Fin b → Char

inductive PreAST.TrueToInput : PreAST n b → (inp : Input b) → Prop
    where
    | skip : TrueToInput (.skip s e G) inp
    | ε : TrueToInput (.ε s e) inp
    | any : inp s = x → TrueToInput (.any s e x) inp
    | terminal : inp s = x → TrueToInput (.terminal s e a x) inp
    | nonTerminal : TrueToInput T inp → TrueToInput (.nonTerminal s e A
    T) inp
    | seq : TrueToInput T1 inp → TrueToInput T2 inp → TrueToInput
    (.seq s e T1 T2) inp
    | prior : TrueToInput T1 inp → TrueToInput T2 inp → TrueToInput
    (.prior s e T1 T2) inp
    | star : TrueToInput T0 inp → TrueToInput TS inp → TrueToInput
    (.star s e T0 TS) inp
    | notP : TrueToInput T inp → TrueToInput (.notP s e T) inp
```

```
def AST.TrueToInput : AST n b → Input b → Prop := fun T =>
    PreAST.TrueToInput T.T
```

3.3.1.3 Uniqueness for Wellformed Trees

With these properties, we can prove some important theorems about the uniqueness of such wellformed AST.

Theorem 3.3.1 (Unique Grammar) *If two grammars G_1 and G_2 are both true to a wellformed tree T, then $G_1 = G_2$.*

```
theorem AST.unique_grammar :
    ∀ {T : AST n b} {G1 G2 : PEG n} {Pexp : GProd n},
    TrueToGrammar T Pexp G1 → TrueToGrammar T Pexp G2 → G1 = G2
```

Proof: This theorem can be simply proved by induction over AST constructors of T. For the cases where subtrees are introduced, we re-apply the theorems to the subtrees.

Theorem 3.3.2 (Unique Input) *If two input I_1 and I_2 are both true to a wellformed tree T with its bound $[s,e)$, then*

$$\forall i \in [s,e), I_1(i) = I_2(i)$$

```
theorem AST.unique_input : ∀ {T : AST n b} {inp1 inp2 : Input b},
    TrueToInput T inp1 → TrueToInput T inp2
    → T.start ≤ i → i < T.end → inp1 i = inp2 i
```

Proof: This theorem can be proved by induction over AST constructors of T and apply the bound conditions in the wellformed property. The base cases are $any(s,e,x)$ and $terminal(s,e,a,x)$ where the input condition is enforced.

One last theorem outlines sufficient conditions for the uniqueness of wellformed trees.

Theorem 3.3.3 (Unique Tree) *If two wellformed AST T_1 and T_2 are both true to grammar G and input I, and if both ASTs have the same starting point $s(T_1) = s(T_2)$, then*

$$T_1 = T_2$$

```
theorem AST.unique_tree :
    ∀ {T1 T2 : AST n b} {inp : Input b} {G : PEG n} {Pexp : GProd n},
    AST.TrueToInput T1 inp → AST.TrueToInput T2 inp →
    AST.TrueToGrammar T1 Pexp G → AST.TrueToGrammar T2 Pexp G →
    T1.start = T2.start → T1 = T2
```

Proof: This theorem is slightly complex to prove. We first show that T_1 and T_2 are never skip trees due to the wellformness. It follows that both trees T_1 and T_2 are built using the same constructor because both trees are true to the same grammar G. We then induct over AST constructors of both T_1 and T_2 and apply the wellformed properties of T_1 and T_2 accordingly. The outline of the proof is shown below:

```
theorem AST.unique_tree :
    ∀ {T1 T2 : AST n b} {inp : Input b} {G : PEG n} {Pexp : GProd n},
    AST.TrueToInput T1 inp → AST.TrueToInput T2 inp →
    AST.TrueToGrammar T1 Pexp G → AST.TrueToGrammar T2 Pexp G →
    T1.start = T2.start → T1 = T2 := by
    intro (.mk T1 valid_T1 wf_T1) (.mk T2 valid_T2 wf_T2);
    intro inp G Pexp hi1 hi2 hg1 hg2 hstart;
    induction T1 generalizing T2 inp G with
    | skip _ _ _ => cases wf_T1;
    | ε s1 e1 => cases T2 with
      | skip _ _ _ => cases wf_T2;
      | ε s2 e2 => ... -- empty expression case
      | _ => cases hg1; cases hg2;
    | any s1 e1 x1 => cases T2 with
      | skip _ _ _ => cases wf_T2;
      | any s2 e2 x2 => ... -- any expression case
      | _ => cases hg1; cases hg2;
    | terminal s1 e1 a1 x1 => cases T2 with
      | skip _ _ _ => cases wf_T2;
      | terminal s2 e2 a2 x2 => ... -- terminal case
      | _ => cases hg1; cases hg2;
    | nonTerminal s1 e1 A1 T1 ih => cases T2 with
      | skip _ _ _ => cases wf_T2;
      | nonTerminal s2 e2 A2 T2 => ... -- non terminal case
      | _ => cases hg1; cases hg2;
    | seq s1 e1 T11 T21 ih1 ih2 => cases T2 with
      | skip _ _ _ => cases wf_T2;
      | seq s2 e2 T12 T22 => ... -- sequence case
      | _ => cases hg1; cases hg2;
    | prior s1 e1 T11 T21 ih1 ih2 => cases T2 with
      | skip _ _ _ => cases wf_T2;
      | prior s2 e2 T12 T22 => ... -- prior case
      | _ => cases hg1; cases hg2;
    | star s1 e1 T01 TS1 ih1 ih2 => cases T2 with
      | skip _ _ _ => cases wf_T2;
      | star s2 e2 T02 TS2 => ... -- greedy repetition case
      | _ => cases hg1; cases hg2;
    | notP s1 e1 T1 ih => cases T2 with
      | skip _ _ _ => cases wf_T2;
      | notP s2 e2 T2 => ... -- not predicate case
      | _ => cases hg1; cases hg2;
```

This is an important theorem to show a parser is complete because given a wellformed grammar and input there is only one possible wellformed AST which respects the grammar and input. Together with the fact that wellformed tree is always meaningful, we know that PEG grammar always either succeed or fail with any input.

3.3.2 Definition of Parser Generator

With the uniqueness of wellformed trees, we can formally define the parser generator function.

To start, we first define the set of arguments required for the parser generator function.

- P_{exp} is the production rules from all non-terminals. PEG expressions for all non-terminal must be wellformed.
- $A \in N$ is the current non-terminal to be parsed.
- $G \sqsubseteq P_{exp}(A)$ the current grammar node to be parsed.
- inp the input string of tokens.
- b the bound of the input, typically it is the size of the input.
- s the starting index of the current grammar node.
- s_A the starting index of parsing current non-terminal. We require $s_A \leq s$ because G is the subterm of $P_{exp}(A)$. In addition, if $s_A = s$, we further require G to be pattern wellformed because no token has been consumed since the start of parsing current non-terminal.

The outputs of this function is complex as it not only returns a wellformed AST but also ensures the returned tree must be coherent with the given grammar and consumed input. Aside from the wellformed AST T to be returned, the following conditions must be satisfied:

- $s = s(T)$, the starting index must match with the returned tree.
- T is true to the grammar G.
- T is true to the input.
- If T is successful and no token is consumed by the tree $s(T) = e(T)$, then G must have *Success Without Consumption* property.
- If T is successful and some tokens are consumed by the tree $s(T) < e(T)$, then G must have *Success With Consumption* property.
- If T is failed, then G must have *Fail* property.

The last three conditions are essential for ensuring the wellformed tree is consistent with the properties of the current grammar node. Also, because the wellformed AST is true to both grammar and input, thanks to the uniqueness theorem for wellformed trees, these conditions ensure that the only one successful or failed parse tree for the input and grammar is the one produced from this parser generator function. This means the parser described above is complete.

In the implementation, we create an output type as a structure which encodes a wellformed AST and all the conditions listed above.

```
structure ParserOutput  (WFPexp : Wellformed_GProd n) (A : Fin n)
                        (G : PEG n) (inp : Input b) (s : Fin b)
```

```
                    (sA : Fin b) where
  is_subterm : G ≤ WFPexp.get A
  sA_le_s : sA ≤ s
  lt_or_pattern_wf : sA < s ∨ PatternWF WFPexp.Pexp WFPexp.σ A G
  T : AST n b
  s_eq_sT : s = T.start
  true_to_grammar : AST.TrueToGrammar T WFPexp.Pexp G
  true_to_input : AST.TrueToInput T inp
  tree_consistent_success_prop0 : PreAST.SuccessAST T.T → s = T.end →
   IsKnown (getProp0 WFPexp.Pexp G)
  tree_consistent_success_propS : PreAST.SuccessAST T.T → s < T.end →
   IsKnown (getPropS WFPexp.Pexp G)
  tree_consistent_failure_propF : PreAST.FailureAST T.T → IsKnown
  (getPropF WFPexp.Pexp G)
```

The type signature of the parser generator function is shown below. Due to the complexity of the function, the body of this function is not shown in this paper.

```
def parse (WFPexp : Wellformed_GProd n) (A : Fin n) (G : PEG n)
    (is_subterm : G ≤ WFPexp.get A) (inp : Input b) (s : Fin b) (sA :
   Fin b) (sA_le_s : sA ≤ s)
    (lt_or_pattern_wf : sA < s ∨ PatternWF WFPexp.Pexp WFPexp.σ A G)
    : ParserOutput WFPexp A G inp s sA
```

3.3.3 Termination of Parser Generator

The parser generator function is guaranteed to be terminated because the following 4-tuple is strictly decreasing over one step.

$$(b - s_A, b - s, A, |G|)$$

At each step, one of the following scenarioes is encountered:

- The current grammar node is moved to its subterm, hence the size of grammar node $|G|$ decreases.
- The current non-terminal is changed to a smaller non-terminal, hence A decreases.
- The current non-terminal is changed to a larger non-terminal. According to pattern wellformness, this case only happens when at least one token has been consumed, meaning s must be strictly greater than s_A. The recursive call of function is made with $s \mapsto s_A$, so s increases.
- If the current grammar node is greedy repetition, since this grammar node is wellformed, its subterm cannot succeed without token consumption. That means s increases.

Chapter 4

Extensible Grammar Specification

This chapter describes the specification and semantics of an extensible grammar. We will look that what Lean 4 has been done and attempt to develop a formalism with respect to the semantics of the extensible syntax based on the core syntax in Lean 4.

4.1 Grammar modification syntax

Since this is an extensible syntax, we must introduce a set of syntax rules to modify the existing grammar. We refer to the syntax extension feature in Lean 4 because it is very straight-forward to declare a new syntax (only two keywords `declare_syntax_cat` and `syntax` are needed). We first define a non-terminal synbol Ω for a group of possible syntax declarations.

4.1.1 Declaring syntax category

In Lean 4, a new syntax categoty can be declared by writing

```
declare_syntax_cat <name>
```

This essentially introduces a new non-terminal to the existing grammar.

We will use the similar construction here:

$$\Omega \longmapsto [\text{"}declare\_syntax\_cat\text{"}]; A$$

where A is the name of non-terminal that has not been declared $A \notin N$.

This syntax appends the new non-terminal symbol to the existing grammar $G = (N, T, P)$ and add a default production rule for the new non-terminal A. Because we must ensure that any grammar which employs the new non-terminal must be failed when no production rule has been defined for the new non-terminal. We define the default production rule as

$$A \longmapsto (![\cdot]*)$$

Here, we use a not-predicate to throw error when the pattern is matched. Because $[\cdot]*$ can match any input (including the empty string of tokens), it will immediately

fail if this pattern is employed. Therefore, this expression is the default case for any non-terminal.

As a result, the declaration of syntax category results into the change in grammar as follows

$$(N,T,P) \xrightarrow{\text{declare\_syntax\_cat } A} (N+[A],T,P')$$

where

$$P'(B) = \begin{cases} (![\cdot]*) & (B = A) \\ P(B) & (B \neq A) \end{cases}$$

This notion of syntax category declaration allows us to prove one obvious but important theorem.

Theorem 4.1.1 *If the current grammar is wellformed, after the declaration of a new syntax category, the new grammar must also be wellformed.*

Proof: Declaring a new syntax category A creates a default production rule for that non-terminal $A \longmapsto (![\cdot]*)$. This PEG expression is clearly structurally wellformed. In addition, because the new production rule does not employ any non-terminal and no non-terminal dependency has been made, the new grammar is still pattern wellformed. The order of non-terminals in this new grammar is almost the same as the previous order. In that case, you may insert A at any place in the order of non-terminals because there is no dependency from A. Therefore, the new grammar must be wellformed.

4.1.2 Declaring syntax production rule

In Lean 4, a new syntax production rule can be expressed by

```
syntax <expr> : <name>
```

We will use the similar construction here:

$$\Omega \longmapsto [\text{"}syntax\text{"}];\delta;[\text{" : "}];A$$

where A is a non-terminal token that has been declared $A \in N$ and δ is a valid PEG expression $\Delta_{N,T}$.

The parser would try to parse $\delta \in \Delta_{N,T}$ from this statement into a PEG expression and append a new production rule $[A \longmapsto \delta]$ to the existing grammar. In Lean 4, a recently declared syntax production rule is always prioritised over other production rules.

That means the declaration of a new production rule $[A \longmapsto \delta]$ results into the change in grammar as follows

$$(N,T,P) \xrightarrow{\text{syntax } \delta : A} (N,T,P')$$

where

$$P'(B) = \begin{cases} \delta / P(B) & (B = A) \\ P(B) & (B \neq A) \end{cases}$$

4.1.3 Recognising non-terminal identifiers with grammar rules

To allow the parser to recognise seen and unseen non-terminal identifiers, we introduce a pair of syntax categories for both existing non-terminals ($\$NT$) and a new non-terminal ($\$NewNT$).

Before that, a format of a non-terminal identifier should be introduced because we should not allow users to modify the set of non-terminals by declaring a custom production rule for syntax categories $\$NT$ and $\$NewNT$. A non-terminal identifier must be a word made up of alphabets and numbers and the first character must be an alphabet. Other formats are also possible if these reserved non-terminal identifiers cannot be called by the users but we will use this format for this thesis.

$$\$FormatNT \longmapsto [a-zA-Z];[a-zA-Z0-9]*$$

It should be noted that the format does not contain any special character like "$". This is essential as it would avoid some essential non-terminals modified by the user.

With that, the production rules of $\$NT$ and $\$NewNT$ are defined below:

$$\begin{aligned} \$NT &\longmapsto [\text{"}A\text{"}]/[\text{"}B\text{"}]/[\text{"}C\text{"}]/\cdots && \text{if} \quad N = \{A,B,C,\cdots\} \\ \$NewNT &\longmapsto (!\$NT);\$FormatNT \end{aligned}$$

To parse the declaration of a syntax category, it first checks the proposed identifier is different from the existing non-terminals, then it checks the identifier against the given format. Upon completion, the identifier is added to the set of non-terminals and the production rule of $\$NT$ is updated.

For exmaple, assuming the non-terminals set contains only one symbol A, if we added the following declaration to the program.

```
declare_syntax_cat B
```

Because the identifier B fails to be parsed by the non-terminal $\$NT \longmapsto [\text{"}A\text{"}]$, a new non-terminal symbol B is added to the non-terminals set and the production rule is updated to $\$NT \longmapsto [\text{"}B\text{"}]/[\text{"}A\text{"}]$. But if we try to declare the same non-terminal again, the identifier is matched with the non-terminal $\$NT$ which results into a parsing failure.

4.1.4 Parsing PEG expression

With that, we introduce the production rules (in PEG format) for parsing PEG expression in the following syntax production rule.

$$
\begin{aligned}
\$PEG &:= \$PEG1; ([;]; \$PEG1)* \\
\$PEG1 &:= \$PEG2; ([/]; \$PEG2)* \\
\$PEG2 &:= \$PEG3; [*] \\
&/ \ \$PEG3 \\
\$PEG3 &:= [!]; \$PEG4 \\
&/ \ \$PEG4 \\
\$PEG4 &:= \$NT \\
&/ \ [”]; CHAR; [”] \\
&/ \ any \\
&/ \ epsilon \\
&/ \ [(]; \$PEG; [)]
\end{aligned}
$$

This set of grammar allows us to parse the user-defined PEG expression from the tokens that respects the intended PEG expression.

4.2 Possible Implementation of Extensible Grammar Parser

4.2.1 Assumptions

We assume the entire program is made up of a series of commands which can be defined in any arbitrary way. In this project, we focus on the case where each line of the program is a command. We also assume that there is a way to separate the entire program into commands that can be parsed individually because parsing the entire program with extensible grammar syntax in one pass is very hard to be done. Although the assumption of "program as multiple distinctive commands" is not practical and applicable in many modern languages such as C language[1] and Python[15], the formalised parser for a more general extensible grammar is beyond the scope of this project and this is something that could be explored in the future.

4.2.2 Parser Structure

The entire parser can be broken down into three sub-parsers, which are

- **Syntax category parser** only parses the command of syntax category declarations.
- **Production rule parser** only parses the command of syntax production rule.
- **General parser** parses the rest of the commands when all other parsers fail to parse.

The parser state stores both a list of non-terminals and a list of production rules for all non-terminals. The first two parsers have relatively fixed grammars as the only changing non-terminal is $\$NT$. The last general parser uses the user-defined grammar defined via other two parsers. Initially, there is only one non-terminal called *command* because we need the top level non-terminal for general parser to begin parsing. The initial production rule for *command* is the default production rule.

$$command \longmapsto (![\cdot]*)$$

Users are able to declare new syntax categories and production rule for the non-terminal *command* and other user-defined non-terminals.

The parser first uses the syntax category parser. If it returns a successful parse, the parser adds the new non-terminal to the parser state and creates the default production rule for that non-terminal.

If the first parser fails, it then attempts to use the production rule parser. If it returns a successful parse, it extracts the PEG expression from the AST, then checks if the proposed change in production rule satisfies wellformness. If the resulting grammar is not wellformed, it reports a syntax error. Otherwise, the production rule is updated accordingly.

Finally, if the above two parsers fail, it uses the general parser to parse the command. Upon successful parse, the parser returns the wellformed AST. In the event of failed parse, it reports a syntax error together with the wellformed AST which is failed.

This 3-pass parser is one of the possible implementation for the extensible grammars. It allows new syntax declarations to be checked during compile time and reports error when parsing termination and determinism are violated. Other implementation (with possibly different syntax extension syntax) is possible and it remains a popular research topic in the field of extensible grammars.

Chapter 5

Conclusions

Throughout this honours project, we developed a largely formalised PEG parser generator in Lean 4 based on Blaudeau and Shankar's work[2] on verified PEG parser generator implemented in PVS. It is shown that a PEG grammar satisfying wellformness criteria is guaranteed to produce an unique wellformed parse tree which respects the given grammar from any input tokens. More importantly, the parser for such wellformed grammar is guaranteed to terminate parsing given any input. Based on the formalised parser generator for a static PEG grammar, we described a special version of an extensible grammar with the assumption of "program as multiple distinctive commands" and outlined one possible use case of this formalised parser generator on parsing this kind of extensible grammars.

Many concepts and theorems on verified PEG parser generator from the previous work[2] cannot be directly translated to Lean 4 theorem prover due to different logical constructions between PVS and Lean 4. Many definitions and propositions which are relatively easy to state in PVS are sometimes very hard to construct using Lean 4's rigorous inductive logic. Nonetheless, the translation of a formalised parser generator implementation to Lean 4 would potentially bring opportunities to verify some aspects of Lean 4 ecosystem. Many features like macros and tactics rely heavily on the currently insecure syntax extension in Lean 4 and the safety of parsing these macros and tactics is not yet guaranteed. The introduction of PEG-based extensible grammar would empower developers to confidently implement practical and verified grammar that always satisfies parsing termination and determinism.

5.1 Future Work

While our formalism in extensible grammars is mostly complete, there are still several area where further work could be done.

In our proposed implementation on parsing extensible grammars, we assume that a program is made up of a series of commands that can be separated and parsed individually (we also further assume the program can be splitted by lines). Based on this assumption, we defined a mechanism of a parser that allows the grammar state in

a parser to be changed by consuming commands with grammar modification syntax. However, this mechanism is no longer feasible when a parser is unable to separate the entire program into number of individual commands. For example, in C language a statement can be spanned over multiple lines and it can even be nested by a statement block[1]. Removing this assumption allows us to verify parsers for more practical programming languages. One possible solution for parsing a more general extensible grammar is to allow parser to change the command separator by introducing a new type of grammar modification syntax.

Currently the extensible grammar we defined in this thesis is based on the parsing expression grammar (PEG). We use this type of grammar because there has been some work on defining wellformed PEG grammars and verifying PEG parser generator. It would be interesting to investigate employing other types of grammar[3] such as context-free grammar (CFG) and the associated Backus-Naur Form (BNF) into the extensible grammar. It is also not clear what constraints should be enforced on CFG such that parsing termination and determinism can be guaranteed. Investigation on other types of grammar would allow us to understand more characteristics of a general extensible grammar.

Lastly, while less related, we would like to investigate the feasibility of tokenisation in the scope of extensible grammar. Currently, individual characters from the input is individually discovered and consumed by the parser. Although the resulting parser generator is relatively easy to prove, it is very inefficient to parse the non-tokenised input. This investigation may help us discover a more efficient mechanism and algorithm for parsing extensible grammars.

Bibliography

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi c specification language. *CEA-LIST, Saclay, France, Tech. Rep. v1*, 2, 2008.

[2] Clement Blaudeau and Natarajan Shankar. A verified packrat parser interpreter for parsing expression grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 3–17, New York, NY, USA, 2020. Association for Computing Machinery.

[3] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[4] Henning Christiansen. *The syntax and semantics of extensible languages*. Roskilde Universitetscenter, Datalogisk Afdeling, 1988.

[5] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[6] Matthew Flatt. Creating languages in racket. *Communications of the ACM*, 55(1):48–56, 2012.

[7] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, jan 2004.

[8] Robert Grimm. Better extensibility through modular syntax. *SIGPLAN Not.*, 41(6):38–51, jun 2006.

[9] Graham Hutton and Erik Meijer. Monadic parser combinators. 09 1999.

[10] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[11] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[12] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021.

[13] Sam Owre, Natarajan Shankar, John M Rushby, and David WJ Stringer-Calvert. Pvs language reference. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1(2):21, 1999.

[14] Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 167–182, Cham, 2020. Springer International Publishing.

[15] Guido VanRossum and Fred L Drake. *The python language reference*. Python Software Foundation Amsterdam, Netherlands, 2010.