Guilherme Dantas de Oliveira

Formalization of Key Algorithms from LPeg

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática, do Departamento de Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
April 2025

Guilherme Dantas de Oliveira

Formalization of Key Algorithms from LPeg

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee:

Rio de Janeiro, April 24th, 2025

Guilherme Dantas de Oliveira

Bachelor in Computer Engineering from PUC-Rio.

Bibliographic data

Oliveira, Guilherme Dantas de

Formalization of Key Algorithms from LPeg / Guilherme Dantas de Oliveira; advisor: Roberto Ierusalimschy. – 2025.

62 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2025.

Inclui bibliografia

1. Informática – Teses. 2. Gramáticas de Análise Sintática de Expressão. 3. Boa formação. 4. LPeg. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, sisters and family for their support and engouragement.

Acknowledgments

To my advisor Professor Roberto Ierusalimschy for the incentive and partnership to carry out this work.

To PUC-Rio, for the aids granted, without which this work could not have been accomplished.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Oliveira, Guilherme Dantas de; Ierusalimschy, Roberto (Advisor). **Formalization of Key Algorithms from LPeg**. Rio de Janeiro, 2025. 62p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Parsing Expression Grammars (PEGs) are a class of deterministic formal grammars originally described by Ford. They are widely used to describe and parse machine-oriented languages and have been implemented by several projects. One such project is LPeg, a Lua library that compiles PEGs into optimized code that is run by a specialized virtual machine.

The implementation of LPeg features two key algorithms that have never been published or verified before. First, LPeg has its own implementation of the well-formedness check introduced by Ford, which is crucial for ensuring that parsing terminates. Second, LPeg implements an algorithm that computes the set of first characters that may be accepted by a pattern, which it uses to optimize the virtual-machine code for certain patterns.

This work formalizes these algorithms and proves their correctness using the Coq proof assistant. We also prove their termination using a gas-based approach.

Keywords

Parsing Expression Grammars; Well-formedness; LPeg.

Resumo

Oliveira, Guilherme Dantas de; Ierusalimschy, Roberto. **Formalização de Algoritmos-Chave de LPeg**. Rio de Janeiro, 2025. 62p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Gramáticas de Análise Sintática de Expressão (PEGs, do inglês Parsing Expression Languages) são uma classe de gramáticas formais determinísticas originalmente descritas por Ford e amplamente utilizadas para descrever e analisar linguagens de programação. PEGs foram implementadas por diversos projetos. Um desses projetos é LPeg, uma biblioteca Lua que compila PEGs para código otimizado que é executado por uma máquina virtual especializada.

A implementação de LPeg apresenta dois algoritmos-chave que nunca foram publicados ou verificados formalmente. Primeiramente, LPeg possui sua própria implementação da verificação de boa-formação introduzida por Ford, essencial para garantir que a análise sintática termine. Em segundo lugar, LPeg implementa um algoritmo que computa o conjunto de primeiros caracteres que podem ser aceitos por um padrão, utilizado para gerar código de máquina virtual mais eficiente para certos padrões.

Este trabalho formaliza esses algoritmos e prova que estão corretos usando o provador de teoremas Coq. Além disso, provamos que esses algoritmos terminam utilizando uma abordagem baseada em consumo de gás.

Palavras-chave

Gramáticas de Análise Sintática de Expressão; Boa formação; LPeg.

Table of contents

Mathematics, rightly viewed, possesses not only truth, but supreme beauty—a beauty cold and austere, like that of sculpture, without appeal to any part of our weaker nature, without the gorgeous trappings of painting or music, yet sublimely pure, and capable of a stern perfection such as only the greatest art can show.

Bertrand Russell, *A History of Western Philosophy.*

1
Introduction

Parsing Expression Grammars (PEGs) are a type of formal grammar introduced by Ford (1) to describe and parse machine-oriented languages. In contrast to Context-Free Grammars (CFGs), established by Chomsky (2), PEGs are deterministic, which makes them particularly suitable for parsing programming languages and structured data formats.

PEGs have been implemented by several projects and in a variety of programming languages. One such implementation is LPeg (3), a library for the Lua programming language. Following the SNOBOL tradition, patterns in LPeg are first-class citizens and can be programmatically constructed in a bottom-up fashion. Starting from basic primitives such as character classes, patterns can be combined through the use of operators: `*` for sequences, `+` for ordered choices, `^` (caret) for repetitions, and so on.

LPeg features several interesting algorithms in its implementation, but two algorithms stand out for their complexity and lack of formal documentation: The well-formedness check and the first-set computation.

The first algorithm, the well-formedness check, ensures that a pattern is complete, meaning it yields a match result for any input string. The concept of well-formedness was introduced by Ford as a conservative approximation to completeness, after proving that the problem of detecting completeness is undecidable, and noticing that incompleteness is caused by left-recursive rules and degenerate loops.

The well-formedness algorithm proposed by Ford iteratively constructs a set of well-formed expressions until a fixed-point is reached. A grammar is then deemed well-formed if its expression set matches the set of well-formed expressions derived from the fixed-point iteration. Koprowski et al. (4) formalized a similar algorithm for an extended definition of PEGs and proved its correctness using the Coq proof assistant.

Meanwhile, the well-formedness check implemented in LPeg does not use iteration, fixed-point checks, or data structures to represent sets of expressions. These details make the algorithm simpler to implement, specially in programming languages with modest standard libraries, such as C. Moreover, this alternative algorithm has neither been published nor formally verified yet, which makes it an interesting research topic.

The second algorithm implemented in LPeg that we highlight in this work is the first-set algorithm. It takes a pattern and returns the first-set and the emptiness value of the pattern. The definition of first-set is well-established in the area of CFGs, but their application in PEGs has not yet been documented in the literature. Basically speaking, the first-set of a pattern is the set of first characters that can be accepted by the pattern. More precisely, a pattern fails any string that starts with a character that is not in its first-set.

The first-set algorithm also returns an emptiness value, which, when false, indicates that the pattern fails to match the empty string. (This Boolean value corresponds to the inclusion of ε in the first-set of some formalizations for CFGs.) This other return value is necessary because the first-set cannot be used to determine whether a pattern fails the empty string, because it has no first character. Together, both return values help LPeg optimize certain patterns, such as ordered choices.

Both aforementioned algorithms implemented in LPeg are complex and lack formal documentation, which can make them difficult to maintain and reason about. This work aims to bridge this gap. We present and analyze these algorithms, proving their correctness using the Coq proof assistant[1]. Moreover, we also prove their termination through a gas-based approach. Also, during the process of proving key properties about these algorithms, we have noticed a few issues that could be reviewed in future versions of LPeg.

The remainder of this work is structured as follows: Chapter 2 presents the syntax and semantics of PEGs. In Chapter 3 we formalize the well-formedness algorithm and prove its correctness and termination. Chapter 4 does the same for the first-set algorithm. In Chapter 5, we analyze how our work differs from prior research, highlighting key differences and improvements. Finally, Chapter 6 summarizes our findings and outlines directions for future research.

[1]The code is publicly available on GitHub at https://github.com/guidanoli/peg-coq

2
Syntax and Semantics of PEGs

When Ford (1) introduced PEGs, he also defined a syntax with several useful constructions for practical language description purposes: literal strings (e.g. “abc”), character classes (e.g. $[a\text{–}z]$), any character (.), non-terminals, optionals $p?$, zero-or-more repetitions $p^\star$, one-or-more repetitions p^+, not-predicates $!p$, and-predicates $\&p$, sequences $p_1\ p_2$, and ordered choices $p_1\ /\ p_2$.

However, for the purposes of formal analysis, Ford realized it would be more convenient to define an abstract syntax for PEGs that represents its essential structure. This abstract syntax does not include several aforementioned constructions, which are treated as *syntactic sugar*, and includes two new constructions: empty (ε) and single characters (e.g. ‘a’).

With these new constructions, Ford *desugars* the PEG syntax as follows. First, the any-character pattern (.) is reduced into a character class with all characters in the alphabet. Then, literal strings are reduced into sequences of characters, and character classes are reduced into choices of characters. And-predicate patterns $\&p$ are reduced into $!(!p)$, optionals $p?$ into $p\ /\ \varepsilon$, and one-or-more repetitions p^+ into $p\ p^\star$.

This desugared syntax allowed Ford to reduce the size of proofs, as there were fewer cases to be treated during case analysis. However, for the purposes of formalizing the first-set algorithm, we realized it would be beneficial for us to have our own desugaring of the PEG syntax. Compared to Ford’s desugared syntax, *our* desugared syntax includes and-predicates $\&p$ and replaces characters (e.g. ‘a’) with character sets (e.g. $[a\text{–}z]$).

Characters and character sets have equivalent expressiveness, as character sets can be reduced into choices of characters. However, in our case, we prefer character sets as they allow us to simplify the definition of the first-set algorithm. Moreover, we decided to keep and-predicates $\&p$ and repetitions $p^\star$ to stay more loyal to the implementation of LPeg, which doesn’t translate $\&p$ into $!(!p)$ and $p^\star$ into $A \leftarrow pA/\varepsilon$, handling them differently in the first-set algorithm. LPeg keeps $\&p$ and $p^\star$ patterns because they lead to smaller and more efficient virtual machine code, when compared to their desugared equivalents. We also denote non-terminals by R_i, where i is the index of the rule in the grammar. Our desugared syntax is depicted in Figure 2.1.

Having defined the syntax of PEGs, we now define their semantics. In the context of a grammar, a pattern is parsed against an input string, and the match may be either a success or a failure. In the case of a success, the pattern

$$\text{Pattern} \quad p \quad := \quad \varepsilon \mid [cs] \mid R_i \mid p^\star \mid !p \mid \&p \mid p_1\ p_2 \mid p_1\ /\ p_2$$

Figure 2.1: Our desugared syntax of PEGs.

leaves the unconsumed suffix of the input string as a result.

To be more precise, we define a match predicate. The predicate takes a grammar g, a pattern p, and an input string s, and returns a match result res. It is denoted as $(g, p, s) \xrightarrow{m} res$. A successful match result is represented by s', the unconsumed suffix of the input string, while a failed match result is represented as $\perp$.

The predicate is inductively defined in Figure 2.2. The empty pattern ε matches any string, without consuming anything. A character set pattern $[cs]$ matches only strings that start with a character in the set $\{cs\}$, and consumes this character. The nonterminal pattern R_i matches the i^{th} rule of the grammar. The repetition pattern $p^\star$ matches p as many times as possible in sequence. The not-predicate pattern $!p$ matches iff p does not match. (It never consumes any input, as either p or $!p$ always fail.) The and-predicate pattern $\&p$ matches iff p matches, but without consuming any input. A sequence pattern $p_1\ p_2$ matches p_1 followed by p_2. Lastly, the pattern $p_1\ /\ p_2$ constitutes an *ordered choice*: It first tries to match p_1, and, if that fails, tries to match p_2.

In case the reader is not familiar with the semantics of PEGs, it can be helpful to go through some examples. Let us start with the pattern $[ab]$, which matches the letters “a” or “b”. Unsurprisingly, this pattern matches and consumes the whole string “a”, leaving the empty string as a result. It also matches the string “baby”, consuming just the first letter “b” and leaving the suffix “aby” unconsumed. Meanwhile, for strings that do not start with either the character “a” or “b”, the pattern fails. For example, the pattern fails for the empty string and for the string “kaaba”.

If we wish the pattern to match zero or more letters “a” or “b”, we can use the repetition operator on the previous pattern, resulting in the pattern $[ab]^\star$. This pattern matches the string “baby”, consuming the prefix “bab” and leaving the letter “y” unconsumed. It also matches the empty string.

Some repetitions, however, are incomplete, meaning they may yield no match result. Take, for example, the pattern $\varepsilon^\star$, which matches ε as many times as possible. The repetition body ε always matches and consumes no input, so the match never makes any progress and gets stuck in an infinite loop.

Some grammar rules can also be incomplete if they reference themselves while consuming no input in-between. These are so-called left-recursive rules. The simplest example of such a rule is $R_i \leftarrow R_i$. We will get into greater detail about them later in this chapter.

$$\frac{}{(g, \varepsilon, s) \xrightarrow{m} s}\ (\textsc{m-eps}) \qquad \frac{}{(g, [cs], nil) \xrightarrow{m} \bot}\ (\textsc{m-set-nil})$$

$$\frac{a \in cs}{(g, [cs], a :: s) \xrightarrow{m} s}\ (\textsc{m-set-cons-in})$$

$$\frac{a \notin cs}{(g, [cs], a :: s) \xrightarrow{m} \bot}\ (\textsc{m-set-cons-not-in})$$

$$\frac{g[i] = \mathit{Some}\ p \qquad (g, p, s) \xrightarrow{m} res}{(g, R_i, s) \xrightarrow{m} res}\ (\textsc{m-nonterminal})$$

$$\frac{(g, p, s) \xrightarrow{m} \bot}{(g, p^\star, s) \xrightarrow{m} s}\ (\textsc{m-rep-fail})$$

$$\frac{(g, p, s) \xrightarrow{m} s' \qquad (g, p^\star, s') \xrightarrow{m} res}{(g, p^\star, s) \xrightarrow{m} res}\ (\textsc{m-rep-succ})$$

$$\frac{(g, p, s) \xrightarrow{m} s'}{(g, !p, s) \xrightarrow{m} \bot}\ (\textsc{m-not-succ}) \qquad \frac{(g, p, s) \xrightarrow{m} \bot}{(g, !p, s) \xrightarrow{m} s}\ (\textsc{m-not-fail})$$

$$\frac{(g, p, s) \xrightarrow{m} s'}{(g, \&p, s) \xrightarrow{m} s}\ (\textsc{m-and-succ}) \qquad \frac{(g, p, s) \xrightarrow{m} \bot}{(g, \&p, s) \xrightarrow{m} \bot}\ (\textsc{m-and-fail})$$

$$\frac{(g, p_1, s) \xrightarrow{m} \bot}{(g, p_1\ p_2, s) \xrightarrow{m} \bot}\ (\textsc{m-seq-fail})$$

$$\frac{(g, p_1, s) \xrightarrow{m} s' \qquad (g, p_2, s') \xrightarrow{m} res}{(g, p_1\ p_2, s) \xrightarrow{m} res}\ (\textsc{m-seq-succ})$$

$$\frac{(g, p_1, s) \xrightarrow{m} s'}{(g, p_1\ /\ p_2, s) \xrightarrow{m} s'}\ (\textsc{m-choice-succ})$$

$$\frac{(g, p_1, s) \xrightarrow{m} \bot \qquad (g, p_2, s) \xrightarrow{m} res}{(g, p_1\ /\ p_2, s) \xrightarrow{m} res}\ (\textsc{m-choice-fail})$$

Figure 2.2: The match predicate.

Besides making repetitions of patterns and referencing rules, we can also chain patterns together with the sequence operator. For example, the pattern $[ab]^\star$ $[y]$ matches the string “baby” but not the string “babies”.

If we wish a pattern to be optional, similar to how the $p?$ operator from the complete PEG syntax does, we can make a choice between this pattern and the empty pattern ε. For example, the pattern $[ab]^\star$ $([y]\ /\ \varepsilon)$ matches both strings “baby” and “babies”, but leaves the suffix “ies” unconsumed.

We can also choose to fail when a given pattern matches, through the not-predicate operator $!p$. Let us say, for example, that the consumed input prefix must not contain the substring “aa’. This behavior is implemented in the pattern $(!([a]\ [a])\ [ab])^\star$. This pattern matches the strings “baba” and “baaba”, but leaves the suffix “aaba” unconsumed.

We can also check whether a pattern matches but not consume any input. That is possible with the and-predicate operator $\&p$. For example, the pattern $\&[ab]$ checks whether a string starts with either the character “a” or “b”, but does not consume it. This pattern matches the string “baby”, while not consuming any characters, and fails to match the string “kaaba”.

One aspect of PEGs that can vary depending on the implementation is how rules are identified and referenced, that is, the set of nonterminals. The original syntax proposed by Ford uses strings (1). Meanwhile, LPeg allows rules to be identified by arbitrary Lua values (5). Meanwhile, our formalization uses natural indices to identify the rules, as it allows us to represent grammars as lists of rules in Coq.

Regarding the match predicate, we can already state two important, yet simple, properties. First, the predicate is deterministic: Given a grammar, a pattern, and an input string, there is at most one possible match result. We state this property below as a lemma, which, like every other lemma in this text, was proven in Coq by us.

Lemma 2.1. *If* $(g, p, s) \xrightarrow{m} res_1$ *and* $(g, p, s) \xrightarrow{m} res_2$*, then* $res_1 = res_2$*.*

The second property states that the result string is a suffix of the input string. We use the symbol “$\preceq$” to denote the suffix relation. This lemma might seem obvious, but we assure the reader that it will be necessary later to prove lemmas by induction on the length of the input string.

Lemma 2.2. *If* $(g, p, s) \xrightarrow{m} s'$*, then* $s' \preceq s$*.*

An important characteristic of this predicate, which has profound consequences in our formalization efforts, is that it not represent a total function: Not all combinations of grammars, patterns, and input strings relate to a match result. This may happen for several reasons, which we will go through in the following paragraphs.

The first reason is that the grammar may be empty, meaning it does not have any rule. This is a problem, because we use the first rule of the grammar as the initial pattern. If the grammar is empty, then there is no first rule.

The second reason is that the pattern may reference a non-existent rule. This problem is not exclusive to our formalization, which references rules by indices. Ford and LPeg also suffer from this problem, because, in order to textually describe a recursive grammar, you need to use some form of reference, which may be invalid.

The third reason for a pattern to not yield a match result is because of so-called *left-recursive* rules. To illustrate what we mean by such, consider the following rule.

$$R_i \leftarrow R_i \tag{2.1}$$

If we were to parse the pattern R_i, we would resolve it to the body of the i^{th} rule, which is also the pattern R_i. In doing so, neither the pattern nor the input string would change, which clearly leads to an infinite loop. This is the simplest example of a left-recursive rule.

The previous example involves only one rule. Left-recursive rules, however, can also involve multiple rules. The following grammar, for example, has three rules, one referencing another, all of which are left-recursive.

$$\begin{cases} R_i \leftarrow R_j \\ R_j \leftarrow R_k \\ R_k \leftarrow R_i \end{cases}$$

The reader might notice that, in both examples, there are rules referencing one another. However, this doesn't necessarily imply in left recursion. The following rule, for example, references itself, but is not left-recursive.

$$R_i \leftarrow [cs]\ R_i\ /\ \varepsilon \tag{2.2}$$

What makes the rule from Equation (2.1) left-recursive, but not the one from Equation (2.2), is a subtle, yet important difference. In Equation (2.1), the nonterminal pattern R_i is always visited with the same input string as the one provided to the rule. Meanwhile, in Equation (2.2), the nonterminal pattern R_i is always visited with an input string shorter than the one provided to the rule, since the character set pattern $[cs]$ always consumes a character when it matches. Given that input strings are finite, we can prove that this rule yields a result for any input string. The proof is carried out by induction on the length of the input string, using Lemma 2.2.

The fourth and last reason for the lack of a match result are so-called degenerate loops, which are repetition patterns $p^\star$ whose body p may match without consuming any input. If p matches, but does not consume any input, then it will do so infinitely many times. One example of a degenerate loop is the pattern $([cs]^\star)^\star$.

In any of these cases, a match result is not guaranteed to exist. In practice, we would like to ensure that a grammar yields a match result for any input string. This property is called *completeness*.

$$g \text{ is complete} \iff \forall s, \exists res, (g, R_0, s) \xrightarrow{m} res \tag{2.3}$$

Ford (1) showed that the problem of knowing whether a grammar is complete is undecidable. He then presented a conservative approximation to completeness known as *well-formedness*, which is decidable.

We formalize this well-formedness check as a computable function *wf* that takes a grammar and returns a Boolean value indicating whether the grammar is well-formed or not. We also prove its correctness by showing that, for any given grammar g, if wf $g = true$, then g is complete.

How LPeg implements this function *wf* is the focus of the next chapter, as well as its proof of correctness. All definitions and proofs are publicly available on this project's GitHub repository[1].

[1]https://github.com/guidanoli/peg-coq

3
Well-formedness Algorithm

In this chapter, we present the well-formedness algorithm implemented in LPeg and prove its correctness. We begin by introducing the signature of the function *verifygrammar*, which implements the well-formedness algorithm in Coq: it takes a grammar g and a natural number *gas* as parameters and returns an optional Boolean value. The parameter *gas* is decremented on every recursive call, in order to convince Coq that the function terminates for every input. If this parameter ever reaches zero, the function returns $None$, signaling it ran out of gas. Otherwise, it returns *Some* b, where the Boolean value b indicates whether the grammar is well-formed or not.

We proved in Coq that, for any input grammar g, there exists a lower bound for the parameter *gas* for which *verifygrammar* returns *Some b*. We do not need to assume anything about the grammar, because the function performs all the necessary checks. This works as a proof of termination for the algorithm. In practice, this results allows LPeg to implement this function in C without the gas parameter, as it is not necessary.

Lemma 3.1. $\forall gas \geq mingas_{vg}(g)$, $\exists b$ *verifygrammar* g *gas* $=$ *Some* b.

We define this lower bound in Figure 3.1. The function takes into account $|g|$, the number of rules in the grammar, and $||g||$, the size of the grammar. We define the size of a pattern p, also denoted as $||p||$, as the number of nodes in its abstract syntax tree, and the size of a grammar g as the summation of the sizes of its rules, that is, $\sum_{r \in g} ||r||$. One neat property about grammar rules is that they are always smaller than the grammar themselves, as displayed in Lemma 3.2. This property follows from the definition of grammar sizes, and from the fact that pattern sizes are non-negative.

Lemma 3.2. $\forall r \in g,\ ||r|| \leq ||g||$.

At this point, it can be hard to explain the rationale behind this equation, as we haven't discussed each step in-depth yet. We will therefore leave the explanation for this lower bound to Section 3.2, when we introduce the left recursion check.

$$mingas_{vg}(g) = (|g| + 2) \cdot ||g||$$

Figure 3.1: The well-formedness function gas lower bound.

```
wf $g$ = **match** verifygramar $g$ $mingas_{vg}(g)$ **with**
         | $Some\ b \Rightarrow b$
         | $None \Rightarrow true$
         **end**
```

Figure 3.2: The well-formedness function.

With this lower bound, we can define the function *wf*, which takes a grammar g and returns a Boolean value indicating whether g is well-formed. It basically calls *verifygrammar* with the gas lower bound. If *verifygrammar* returns *Some b*, then *wf* returns *b*. If, otherwise, *verifygrammar* returns *None*, then *wf* returns *true*. This last case is irrelevant, because we know it cannot happen. Nevertheless, we return *true* to demonstrate that the function and gas estimation are correct. Figure 3.2 displays the function *wf*.

Let us now get into the function *verifygrammar*, which is divided into four steps. Each step is implemented by a different function. The first two functions, named *coherent* and *lcoherent*, do not receive a gas parameter, as they are defined recursively on the structure of patterns and lists of patterns, respectively. This structural recursion is enough to convince Coq that these functions terminate. Meanwhile, the last two functions, named *lverifyrule* and *lcheckloops*, receive a gas parameter, as their recursion goes beyond the structure of patterns, and involves visiting arbitrary grammar rules. Therefore, in order to prove that *verifygrammar* terminates, we need to provide lower bounds for the gas parameter of both *lverifyrule* and *lcheckloops*.

The implementation of these functions will be discussed in the following sections. For now, let us briefly go over them. The first two steps are relatively simple, while the third and fourth steps are more complex, as they involve symbolically parsing each rule.

The first step of the algorithm is trivial. It merely ensures the existence of the first rule of the grammar, given that it is used as the starting point for parsing the grammar.

The second step is similar to the first one, as it makes sure that every rule in the grammar only references rules that exist in the grammar. This ensures that we can safely dereference any nonterminal patterns later on. This step is an extended version of the previous one for lists of patterns.

The third step ensures that the grammar contains no left-recursive rules. It does so by symbolically executing the parsing routine for each rule in the grammar and checking whether it can reach the same rule twice without

```
coherent $g$ $p$ =
**match** $p$ **with**
| $\varepsilon \Rightarrow true$
| $[cs] \Rightarrow true$
| $R_i \Rightarrow$ **match** $g[i]$ **with**
          | $Some\ p \Rightarrow true$
          | $None \Rightarrow false$
          **end**
| $p^\star \Rightarrow$ coherent $g$ $p$
| $!p \Rightarrow$ coherent $g$ $p$
| $\&p \Rightarrow$ coherent $g$ $p$
| $p_1\ p_2 \Rightarrow$ coherent $g$ $p_1 \wedge$ coherent $g$ $p_2$
| $p_1\ /\ p_2 \Rightarrow$ coherent $g$ $p_1 \wedge$ coherent $g$ $p_2$
**end**
```

Figure 3.3: The coherence function.

consuming any input.

The fourth and final step makes sure that the grammar contains no degenerate loops, which are repetitions of patterns that may match while consuming no input.

If a grammar passes all these checks, then it is considered well-formed. In the following sections, we go into each of these steps in greater detail. We also define equivalent inductive predicates for each step and for the *verifygrammar* function, to aid us in the proofs. We also prove these predicates follow the corresponding fixed-point definitions.

3.1 References to nonexistent rules

The verification process starts by checking whether every nonterminal pattern references an existing rule in the grammar. This process is quite simple, but we present it here in the name of completeness.

We say a pattern is *coherent* in respect to a grammar if all of its nonterminals reference existing rules in the grammar. Figure 3.3 defines a fixed-point function that performs this verification. To aid us in later induction proofs, we also define an equivalent predicate in Figure 3.4. Lemma 3.3 states that the predicate is deterministic on the result, and Lemma 3.4 states that the predicate follows the function. It is easy to see that both lemmas together imply that the predicate is equivalent to the function.

$$\frac{}{(g, \varepsilon) \xrightarrow{c} true} \text{(C-EPS)} \qquad \frac{}{(g, [cs]) \xrightarrow{c} true} \text{(C-SET)}$$

$$\frac{g[i] = Some\ (p)}{(g, R_i) \xrightarrow{c} true} \text{(C-NONTERMINAL-SOME)}$$

$$\frac{g[i] = None}{(g, R_i) \xrightarrow{c} false} \text{(C-NONTERMINAL-NONE)} \qquad \frac{(g, p) \xrightarrow{c} res}{(g, p^\star) \xrightarrow{c} res} \text{(C-REP)}$$

$$\frac{(g, p) \xrightarrow{c} res}{(g, !p) \xrightarrow{c} res} \text{(C-NOT)} \qquad \frac{(g, p) \xrightarrow{c} res}{(g, \&p) \xrightarrow{c} res} \text{(C-AND)}$$

$$\frac{(g, p_1) \xrightarrow{c} res_1 \qquad (g, p_2) \xrightarrow{c} res_2}{(g, p_1\ p_2) \xrightarrow{c} res_1 \wedge res_2} \text{(C-SEQ)}$$

$$\frac{(g, p_1) \xrightarrow{c} res_1 \qquad (g, p_2) \xrightarrow{c} res_2}{(g, p_1\ /\ p_2) \xrightarrow{c} res_1 \wedge res_2} \text{(C-CHOICE)}$$

Figure 3.4: The coherence predicate.

```
lcoherent $g$ $rs$ = **match** $rs$ **with**
                     | $nil \Rightarrow true$
                     | $r :: rs' \Rightarrow$ coherent $g$ $r \wedge$ lcoherent $g$ $rs'$
                     **end**
```

Figure 3.5: The coherence function for lists of patterns.

Lemma 3.3. *If* $(g, p) \xrightarrow{c} res_1$ *and* $(g, p) \xrightarrow{c} res_2$*, then* $res_1 = res_2$*.*

Lemma 3.4. *If coherent* g $p = res$*, then* $(g, p) \xrightarrow{c} res$*.*

Figure 3.5 trivially generalizes the coherence check for a list of patterns. This function is defined over an arbitrary list of rules, but is meant to be called for the whole grammar. We also define, in Figure 3.6, an inductive predicate equivalent to this function to be later used in proofs by induction. We also show that this predicate is deterministic and follows the original fixed-point definition. See Lemmas 3.5 and 3.6.

Lemma 3.5. *If* $(g, rs) \xrightarrow{lc} res_1$*, and* $(g, rs) \xrightarrow{lc} res_2$*, then* $res_1 = res_2$*.*

Lemma 3.6. *If lcoherent* g $rs = res$*, then* $(g, rs) \xrightarrow{lc} res$*.*

$$\frac{}{(g, nil) \xrightarrow{lc} true} \text{ (LC-NIL)} \qquad \frac{(g, r) \xrightarrow{c} b_1 \qquad (g, rs) \xrightarrow{lc} b_2}{(g, r :: rs) \xrightarrow{lc} b_1 \wedge b_2} \text{ (LC-CONS)}$$

Figure 3.6: The coherence predicate for lists of patterns.

Finally, we prove that if a list of patterns passes the list-based check, then any pattern in the list passes the individual check.

Lemma 3.7. *If* $(g, rs) \xrightarrow{lc} true$, *then,* $\forall r \in rs, (g, r) \xrightarrow{c} true$.

3.2 Left-recursive rules

In general, we consider a rule to be left-recursive if it can wind up in itself without consuming any input in-between. This brings us to the heart of the algorithm that detects left-recursive rules. On a high level, it symbolically parses each rule, until it either consumes some input, visits some rule twice without consuming any input, or simply finishes.

The algorithm categorizes patterns into three groups. If a pattern can be parsed until its end without consuming any input, it is said to be *nullable*. If, otherwise, it always consumes some input, it is categorized as *non-nullable*. Alternatively, if it can lead to some rule twice, without consuming any input, it is categorized as left-recursive.

In order to check whether a pattern is guaranteed to consume some input, the algorithm uses a conservative approximation proposed by Ford (1), which makes two assumptions. The first one is that $!p$ may match, and the second one is that, in the case of p_1/p_2, it may visit p_2, without checking whether p_1 always matches. For illustrative purposes, we present a simple counterexample for each assumption.

A counterexample for the first assumption is the pattern $!\varepsilon$, which never matches. Meanwhile, for the second assumption, a simple counterexample is the pattern ε/p_2, because ε always matches, and, therefore, p_2 is never visited. By the simplicity of the counterexamples, the reader might think that these cases can be easily spotted. However, Ford (1) proved that the general case of this problem is undecidable: If there were such an algorithm, then you could determine whether the language of two arbitrary patterns p_1 and p_2 have a non-empty intersection, which is a knownly undecidable problem, by running this hypothetical algorithm for the pattern $!p_1\&p_2$.

One of the conditions for our algorithm to yield a result is when it visits a pattern that is guaranteed to consume some input. As a result, it exclusively visits patterns that may be reached without consuming any input. Therefore, if the algorithm revisits a rule, this means a path exists in which the parsing routine may reach the same rule and with the same input string, which would indicate that such rule is left-recursive. We will now discuss possible ways to detect when a rule has been visited twice.

One possible way to detect left-recursive rules is through a set of visited rules, which is checked and updated every time a nonterminal pattern is visited. For a grammar with n rules, this set could be implemented as an array of n Boolean values, each representing a rule. This method achieves a computation and spatial completity of $O(n)$.

Another approach, which is simpler and takes less memory space, uses a counter of visited rules, which starts at zero and gets incremented every time a rule is visited. If this value ever surpasses the number of grammar rules, then we know, by the pigeonhole principle, that some rule has been visited more than once. In the case of grammars with left-recursive rules, we may visit more rules than necessary, however, we are not particularly worried about the performance of the algorithm in the case of errors.

LPeg adopts this last approach. Our formalization follows LPeg, though with a small twist: instead of counting visited rules from zero until the limit, we count to-be-visited rules from the limit down to zero. This simplifies our formalization by moving the limit calculation out of the algorithm body, and letting the limit be passed down as a parameter instead.

At this point, it is important to draw a distinction between exhausting the counter of to-be-visited rules and correctly identifying a left-recursive rule. When the algorithm starts, the counter is initialized with the provided limit. It is then decremented every time a rule is visited. If the counter ever reaches zero, then attempting to visit any rule will return an error. The algorithm does not determine whether this error indicates left recursion, because it would require the algorithm to check whether the limit is greater than the number of grammar rules. Instead, we leave it to the caller to provide a high enough limit, in which case the algorithm indeed correctly labels rules as left-recursive by returning an error.

Because of this shift in responsibilities, we adapt the nomenclature for the counter parameter and associated error, based on an analogy with call stacks. If, every time a rule is visited, it were pushed onto a stack k, then we could think of the counter parameter d as the stack depth limit; and surpassing it would be similar to a stack overflow error.

For the sole purpose of helping us prove certain properties about the algorithm, we will also include this stack k, a list of rule indices, as an output, though it doesn't affect the algorithm. As we will soon see, it is either appended, passed along, or ignored. It works as a trace of the inner workings of the function, a high-level concept we only use for proving lemmas about this algorithm. LPeg also implements this output, but it is only used when formatting error messages about left-recursive rules.

We now describe the algorithm for detecting left-recursive rules, starting with its inputs and outputs. It receives a pattern, a grammar, and a stack depth limit, and returns a label and a stack. We represent labels by optional Boolean values $Some\ true$ (nullable), $Some\ false$ (non-nullable) and $None$ (stack overflow error); and stacks by either nil (an empty stack) or $i :: k$ (a rule of index i concatenated with a stack k). For now, we will work with this signature, but beware that the actual function, displayed in Figure 3.7, receives an extra parameter which we will introduce later in this section.

The function is defined recursively. In most cases, it calls itself for each sub-pattern. In the case of nonterminal patterns, however, it calls itself for the referenced rule. Furthermore, the function propagates any stack overflow errors. This means that, if some recursive call returns $None$, signaling a stack overflow, and a stack k, then the function also returns $None$ and k.

For the empty pattern ε, the function returns a label $Some\ true$ and nil, because it is nullable and doesn't visit any nonterminal. We categorize it as nullable because it may match while consuming no input. In particular, it always matches while consuming no input.

As for character set patterns $[cs]$, the function returns $Some\ false$ and nil, because it is non-nullable, meaning it always consumes some input when it matches. It also doesn't visit any nonterminal.

For a nonterminal pattern R_i, the function first checks the stack depth limit d. If $d = 0$, it returns $None$ and nil, signaling a stack overflow and that it didn't visit any nonterminal. Otherwise, if $d \geq 1$, then the function calls itself for the i^{th} rule of the grammar, while passing a stack depth limit of $d - 1$. If this recursive call returns a label res and a stack k, then the function returns res and $i :: k$. This way, the stack accumulates the indices of the grammar rules in the same order in which they are visited.

For a repetition pattern $p^\star$, the function evaluates p, which returns res and k, to check for any stack overflow errors. If $res \neq None$, then it returns $Some\ true$ and k, as it can match while consuming no input, in case p fails. We assume that p can fail because, in the final verification step, we ensure that p is non-nullable, and we know that non-nullable patterns fail to match the empty string.

Predicate patterns $!p$ and $\&p$ are evaluated in the same way as repetition

```
verifyrule $g\ p\ d\ nb\ 0 = None$
verifyrule $g\ p\ d\ nb\ (1 + gas) =$
**match** $p$ **with**
| $\varepsilon \Rightarrow Some\ (Some\ true, nil)$
| $[cs] \Rightarrow Some\ (Some\ nb, nil)$
| $p'^{\star} \Rightarrow$ verifyrule $g\ p'\ d\ true\ gas$
| $!p' \Rightarrow$ verifyrule $g\ p'\ d\ true\ gas$
| $\&p' \Rightarrow$ verifyrule $g\ p'\ d\ true\ gas$
| $R_i \Rightarrow$ **match** $g[i]$ **with**
        | $None \Rightarrow None$
        | $Some\ p' \Rightarrow$ **match** $d$ **with**
                    | $0 \Rightarrow Some\ (None, nil)$
                    | $1 + d' \Rightarrow$ **match** verifyrule $g\ p'\ d'\ nb\ gas$ **with**
                              | $Some\ (res, k) \Rightarrow Some\ (res, i :: k)$
                              | $None \Rightarrow None$
                              **end**
                    **end**
        **end**
| $p_1\ p_2 \Rightarrow$ **match** verifyrule $g\ p_1\ d\ false\ gas$ **with**
          | $Some\ (Some\ true, k) \Rightarrow$ verifyrule $g\ p_2\ d\ nb\ gas$
          | $Some\ (Some\ false, k) \Rightarrow Some\ (Some\ nb, k)$
          | $res \Rightarrow res$
          **end**
| $p_1\ /\ p_2 \Rightarrow$ **match** verifyrule $g\ p_1\ d\ nb\ gas$ **with**
            | $Some\ (Some\ nb', k) \Rightarrow$ verifyrule $g\ p_2\ d\ nb'\ gas$
            | $res \Rightarrow res$
            **end**
**end**
```

Figure 3.7: The left recursion detection function.

```
$eval\ g\ [\![p_1\ /\ p_2]\!]\ d\ nb =$ match $eval\ g\ p_1\ d\ nb$ with
                              | $(Some\ nb', k) \Rightarrow eval\ g\ p_2\ d\ nb'$
                              | $(None, k) \Rightarrow (None, k)$
                              end
```

Figure 3.8: Pseudocode of the evaluation of choice patterns.

patterns, but for different reasons. Repetition patterns are nullable because they can always match without consuming any input. Meanwhile, predicate patterns are nullable by approximation, under the assumption that p may match, in the case of $\&p$, or fail to match, in the case of $!p$.

For a sequence pattern $p_1\ p_2$, the function first evaluates p_1. If p_1 is non-nullable, then so is the sequence $p_1\ p_2$, and the function returns the same label and stack as p_1. Note that p_2 is not even evaluated in this case, because it would be visited with a shorter input string during parsing. This is the only case in which the nullable property comes into play in this algorithm. If, otherwise, p_1 is nullable, then it evaluates p_2 and returns the same label and stack as p_2.

Finally, for a choice pattern $p_1\ /\ p_2$, it first evaluates p_1. If it returns *Some* b_1, indicating that p_1 did not overflow the stack, then it evaluates p_2. If it also returns *Some* b_2, then the function returns *Some* $(b_1 \lor b_2)$ and the same stack as p_2.

The algorithm we've just described is quite similar to the one implemented in LPeg. There is, however, one small difference related to the use of tail calls as an optimization technique. In C, tail calls are implemented with `goto` statements. To apply this optimization technique, LPeg adds an extra parameter to the function to work as an accumulator for the nullable property. Without this accumulator parameter, the evaluation of choice patterns $p_1\ /\ p_2$ would rely solely on recursion. It would evaluate p_1 and p_2, then perform a Boolean OR operation on the results.

With the addition of a Boolean parameter nb, we can turn the evaluation of p_2 into a tail call. Instead of making the Boolean OR operation explicitly, we let the accumulator do it under-the-hood. This works because, in the base cases of the recursion, in which the function would return either *Some true* or *Some false*, we return instead *Some* $(true \lor nb)$ and *Some* $(false \lor nb)$, which get simplified to *Some true* and *Some nb*, respectively. Figure 3.8 shows a Coq-like pseudocode of how choice patterns are evaluated with the Boolean parameter nb.

We would also like to highlight how this nullable accumulator allows the evaluation of repetitions and predicates to be rewritten as tail calls. Previously, we would have to check if p evaluated to $None$, before returning $Some\ true$. Now, we can simply pass $true$ as the nullable accumulator, which guarantees that, if p does not evaluate to $None$, it evaluates to $Some\ true$.

There are some ways in which this function could be implemented in Coq as a fixed-point. The classical way is to add a gas parameter, which gets decremented in every recursive call. We make the function return an optional value, such that, if the gas parameter ever reaches zero, it returns $None$. Other ways are providing a well-formedness proof, or a measure function. We choose the first strategy, because it is the simplest to implement.

Finally, Figure 3.7 presents the algorithm defined as a fixed-point function. It returns an optional value, because we adopted the gas strategy, but also because it cannot evaluate nonterminal patterns that reference nonexistent rules. In this case, the function also returns $None$. In all other cases, the function returns $Some\ (res, k)$, with res being a label, and k, a stack.

At this point, the reader should be warned that we will not attempt to prove the correctness of this function in isolation. In fact, we will not even try to formally define left-recursive rules. This might frustrate the reader, but we assure you that such proof will not be necessary. Instead, we will later prove the correctness of the whole algorithm once we introduce all steps of the verification process. In this section, we will simply prove that the label returned by the function is monotonic and eventually constant with respect to the gas counter and stack depth limit.

A function f is said to be monotonically increasing if, for any x and y, such that $x \leq y$, it is always true that $f(x) \leq f(y)$. In the case of the *verifyrule* function, this will be true for the gas counter and stack depth limit parameters, and the order between optionals is $None < Some\ res$, for any res. This means that, if the function ever returns $Some\ res$, increasing the gas counter or the stack depth limit will not alter the return value.

A function f is eventually constant if, for some N and for any x and y, such that $x, y \geq N$, it is true that $f(x) = f(y)$. In the case of *verifyrule* function, this will be true for the gas counter and stack depth limit parameters and for the label return value. This means that both the gas counter and stack depth limit have lower bounds for which the returned label stabilizes.

About this fixed-point definition, we will initially prove some basic lemmas. Starting with Lemma 3.8, we state that, if the function returns $Some\ (res, k)$, then increasing the value of the gas parameter will not change the result. This is what we mean by the function being monotonic and eventually constant with respect to the gas counter.

Lemma 3.8. *If $verifyrule\ g\ p\ d\ nb\ gas = Some\ (res, k)$, then $\forall gas' \geq gas$, $verifyrule\ g\ p\ d\ nb\ gas' = Some\ (res, k)$.*

Lemma 3.9 states that, for any coherent pattern and grammar, there exists a lower bound for the gas parameter, for which the function returns $Some\ (res, k)$. The lower bound takes into account the size of the pattern $||p||$, the size of the grammar $||g||$, and the stack depth limit d.

Lemma 3.9. *If $(g, p) \xrightarrow{c} true$, and $(g, g) \xrightarrow{lc} true$, then, $\forall gas \geq ||p|| + d \cdot ||g||$, $\exists res\ \exists k\ verifyrule\ g\ p\ d\ nb\ gas = Some\ (res, k)$.*

Proof. For most patterns, the proof follows from induction on the pattern p. Meanwhile, for non-terminal patterns, the proof follows from induction on the stack depth limit d. We show below how the lower bound for a rule r and stack depth limit d is derived from a non-terminal R_i that references r and stack depth limit $d+1$. We use Lemma 3.2 to show that $||g|| \geq ||r||$.

$$\begin{aligned} gas &\geq ||R_i|| + (d+1) \cdot ||g|| \\ &\geq 1 + (d+1) \cdot ||g|| \\ &\geq 1 + ||g|| + d \cdot ||g|| \\ &\geq 1 + ||r|| + d \cdot ||g|| \end{aligned}$$

□

Now, we would like to prove that the label returned by the *verifyrule* function is monotonic and eventually constant with respect to the stack depth limit. We discard the returned stack in this context because, in the case of left-recursive rules, the stack returned by the function will, in fact, diverge. However, we are not interested in the output stack, in this case. What really matters to the following steps of the verification process is the label. In particular, we would like to make sure that no rule in the grammar is marked with the label $None$, meaning "stack overflow".

In order to prove such lemma, we realized an inductive, gasless predicate would be better suited than the fixed-point definition, as it would be easier to perform proofs by induction, and without having to deal with a gas parameter. Figure 3.9 defines such predicate, denoted as $(g, p, d, nb) \xrightarrow{vr} (res, k)$. It takes a grammar g, a pattern p, a stack depth limit d, and a nullable accumulator nb, and outputs a result res, and a stack trace k.

In order to reach our final goal of proving that the label returned by the *verifyrule* function is monotonic and eventually constant with respect to the stack depth limit, we need to first prove some intermediary lemmas. First, we

$$\frac{}{(g, \varepsilon, d, nb) \xrightarrow{vr} (\mathit{Some\ true}, \mathit{nil})} \ (\textsc{v-eps})$$

$$\frac{}{(g, [cs], d, nb) \xrightarrow{vr} (\mathit{Some\ nb}, \mathit{nil})} \ (\textsc{v-set})$$

$$\frac{}{(g, R_i, 0, nb) \xrightarrow{vr} (\mathit{None}, \mathit{nil})} \ (\textsc{v-nonterminal-zero})$$

$$\frac{g[i] = \mathit{Some}\ p \qquad (g, p, d, nb) \xrightarrow{vr} (res, k)}{(g, R_i, d+1, nb) \xrightarrow{vr} (res, i :: k)} \ (\textsc{v-nonterminal-succ})$$

$$\frac{(g, p, d, \mathit{true}) \xrightarrow{vr} (res, k)}{(g, p^\star, d, nb) \xrightarrow{vr} (res, k)} \ (\textsc{v-rep}) \qquad \frac{(g, p, d, \mathit{true}) \xrightarrow{vr} (res, k)}{(g, !p, d, nb) \xrightarrow{vr} (res, k)} \ (\textsc{v-not})$$

$$\frac{(g, p, d, \mathit{true}) \xrightarrow{vr} (res, k)}{(g, \&p, d, nb) \xrightarrow{vr} (res, k)} \ (\textsc{v-and}) \qquad \frac{(g, p_1, d, \mathit{false}) \xrightarrow{vr} (\mathit{None}, k)}{(g, p_1\ p_2, d, nb) \xrightarrow{vr} (\mathit{None}, k)} \ (\textsc{v-seq}1)$$

$$\frac{(g, p_1, d, \mathit{false}) \xrightarrow{vr} (\mathit{Some\ false}, k)}{(g, p_1\ p_2, d, nb) \xrightarrow{vr} (\mathit{Some\ nb}, k)} \ (\textsc{v-seq}2)$$

$$\frac{(g, p_1, d, \mathit{false}) \xrightarrow{vr} (\mathit{Some\ true}, k_1) \qquad (g, p_2, d, nb) \xrightarrow{vr} (res, k_2)}{(g, p_1\ p_2, d, nb) \xrightarrow{vr} (res, k_2)} \ (\textsc{v-seq}3)$$

$$\frac{(g, p_1, d, nb) \xrightarrow{vr} (\mathit{None}, k)}{(g, p_1\ /\ p_2, d, nb) \xrightarrow{vr} (\mathit{None}, k)} \ (\textsc{v-choice}1)$$

$$\frac{(g, p_1, d, nb) \xrightarrow{vr} (\mathit{Some}\ nb', k_1) \qquad (g, p_2, d, nb') \xrightarrow{vr} (res, k_2)}{(g, p_1\ /\ p_2, d, nb) \xrightarrow{vr} (res, k_2)} \ (\textsc{v-choice}2)$$

Figure 3.9: The left recursion detection predicate.

need to relate the predicate and the fixed-point definition together, so that we can apply the proofs about the former to the latter.

We begin with Lemma 3.10, which states that, for identical input, the predicate yields the same output. We can therefore state that the predicate is deterministic.

Lemma 3.10. *If* $(g, p, d, nb) \xrightarrow{vr} (res_1, k_1)$*, and* $(g, p, d, nb) \xrightarrow{vr} (res_2, k_2)$*, then* $res_1 = res_2$ *and* $k_1 = k_2$*.*

Lemma 3.11 shows that every result returned by the fixed-point definition can be inductively constructed using the predicate definition.

Lemma 3.11. *If* $\mathit{verifyrule}\ g\ p\ d\ nb\ gas = \mathit{Some}\ (res, k)$*, then* $(g, p, d, nb) \xrightarrow{vr} (res, k)$*.*

Lemma 3.12 shows that, if a pattern evaluates to either nullable or non-nullable, then increasing the stack depth limit doesn't affect the result. This is expected, because, in both cases, it doesn't surpass the limit, and increasing it preserves this property by transitivity.

Lemma 3.12. *If* $(g, p, d, nb) \xrightarrow{vr} (Some\ nb', k)$*,*
then $\forall d' \geq d, (g, p, d', nb) \xrightarrow{vr} (Some\ nb', k)$*.*

Lemma 3.13 shows that, on stack overflow, the output stack k has length d. That is expected, because a stack overflow happens when the stack is full before trying to visit a rule.

Lemma 3.13. *If* $(g, p, d, nb) \xrightarrow{vr} (None, k)$*, then* $|k| = d$*.*

Lemma 3.14 states that the output stack only contains references to existing rules in the grammar. Since we are identifying rules by their indices in a list, we prove this by showing that these indices are less than the number of rules in the grammar, denoted as $|g|$. This lemma may seem trivial, but it is necessary for us to later prove, using the pigeonhole principle, that a stack with more rules than the grammar will have at least one repeated rule.

Lemma 3.14. *If* $(g, p, d, nb) \xrightarrow{vr} (res, k)$*, then* $\forall i \in k, i < |g|$*.*

Lemma 3.15 states that, for any evaluation that results in a stack overflow, we can pick any rule i from the output stack k, and evaluate it with a certain stack depth limit, so that it also results in a stack overflow, and returns a suffix of the original stack k, starting from i.

Lemma 3.15. *If* $(g, p, d, nb) \xrightarrow{vr} (None, k_1 +\!\!+\, i :: k_2)$*,*
then $(g, R_i, 1 + |k_2|, nb') \xrightarrow{vr} (None, i :: k_2)$*.*

Under the same assumptions, Lemma 3.16 shows that, if we evaluate a rule i from the stack with an increased stack depth limit and it still results in a stack overflow and returns a stack $i :: k_3$, then we can increase the stack depth limit of the original evaluation by the same amount, it will also result in a stack overflow, and return a stack that ends with $i :: k_3$.

Lemma 3.16. *If* $(g, p, d, nb) \xrightarrow{vr} (None, k_1 +\!\!+\, i :: k_2)$*,*
and $(g, R_i, 1 + |k_3|, nb') \xrightarrow{vr} (None, i :: k_3)$*, and* $|k_2| \leq |k_3|$*,*
then $(g, p, 1 + |k_1| + |k_3|, nb) \xrightarrow{vr} (None, k_1 +\!\!+\, i :: k_3)$*.*

Lemma 3.17 shows that, if an evaluation results in a stack overflow, and a rule i occurs more than once in the output stack, then we can increase the stack depth limit by a certain amount, and both conditions will still hold true.

```
lverifyrule $g$ $rs$ $gas$ :=
match $rs$ with
| $nil \Rightarrow Some\ true$
| $r :: rs' \Rightarrow$ let $d$ := $|g| + 1$ in
          match verifyrule $g$ $r$ $d$ $false$ $gas$ with
          | $Some\ (Some\ b, k) \Rightarrow$ lverifyrule $g$ $rs'$ $gas$
          | $Some\ (None, k) \Rightarrow Some\ false$
          | $None \Rightarrow None$
          end
end
```

Figure 3.10: The left recursion detection function for lists of patterns.

Lemma 3.17. *If* $(g, p, d, nb) \xrightarrow{vr} (None, k_1 +\!\!+\, i :: k_2 +\!\!+\, i :: k_3)$, *then* $\exists d'$, *such that* $(g, p, d', nb) \xrightarrow{vr} (None, k_1 +\!\!+\, i :: k_2 +\!\!+\, i :: k_2 +\!\!+\, i :: k_3)$.

Finally, we present the main lemma that we wanted to prove. Lemma 3.18 shows that, if an evaluation with a stack depth limit greater than the number of grammar rules yields a result, then any evaluation with an even greater stack depth limit yields the same result. The stacks can be different, but they are irrelevant for our purpose of identifying left-recursive rules.

Lemma 3.18. *If* $(g, p, d, nb) \xrightarrow{vr} (res, k)$, *and* $d > |g|$, *then, for any* $d' \geq d$, $\exists k'$, *such that* $(g, p, d', nb) \xrightarrow{vr} (res, k')$.

We now explain the proof of this lemma. For the cases in which the evaluation does not result in a stack overflow, the proof follows from Lemma 3.12. Now, in the case of a stack overflow, we know from Lemma 3.13 that the length of the stack k is equal to the stack depth limit d, which, in this case, we assume to be greater than n, the number of grammar rules. Therefore, $|k| > n$. We know from Lemma 3.14 that the stack only contains valid grammar rule indices. That is, $\forall i \in k, i < n$. We use these two observations and the pigeonhole principle to conclude that the stack must have at least one repeated rule. From Lemma 3.17, we show that we can increase the stack depth limit arbitrarily, and it will still result in a stack overflow.

Having defined the algorithm that checks if a pattern is free of left recursion, we now use this definition to create a function that performs this check for a list of patterns. Figure 3.10 defines this function, which receives a grammar, a list of patterns, and a gas counter, and returns an optional Boolean value indicating whether all patterns in the grammar are free of left recursion.

$$\frac{}{(g, nil) \xrightarrow{lvr} true} \quad \text{(LVR-NIL)}$$

$$\frac{(g, r, d, false) \xrightarrow{vr} (Some\ nb, k) \qquad (g, rs) \xrightarrow{lvr} b}{(g, r :: rs) \xrightarrow{lvr} b} \quad \text{(LVR-CONS-SOME)}$$

$$\frac{|g| < d \qquad (g, r, d, false) \xrightarrow{vr} (None, k)}{(g, r :: rs) \xrightarrow{lvr} false} \quad \text{(LVR-CONS-NONE)}$$

Figure 3.11: The left recursion detection predicate for lists of patterns.

This new function provides values for two of the parameters of the underlying function: the stack depth limit d, initialized with $|g| + 1$, the lower bound from Lemma 3.18, and the nullable accumulator *nb*, initialized with $false$. We could have omitted the gas counter, by providing the lower bound from Lemma 3.9, but we decided to postpone this omission to the top-most definition of well-formedness in our formalization.

We provide a lower bound for the gas parameter, for which the function returns some result. Note that we're assuming that both the grammar g and the list of rules rs are coherent, because they could be different. In practice, however, they will be the same. In this case, where $rs = g$, the equation for the lower bound can be simplified to $(|g| + 2) \cdot ||g||$. That is the origin of the lower bound of the *verifygrammar* function as displayed in Figure 3.1.

Lemma 3.19. *If* $(g, g) \xrightarrow{lc} true$ *and* $(g, rs) \xrightarrow{lc} true$*,*
then, $\forall gas \geq ||rs|| + (|g| + 1) \cdot ||g||$*,* $\exists res$ *lverifyrule* $g\ rs\ gas = Some\ res$*.*

Proof. The proof follows by induction on the list of rules rs, and from the gas lower bound for the function *verifyrule* from Lemma 3.9, substituting the stack depth limit d with $|g| + 1$. □

We will use this function for verifying that the grammar contains no left-recursive rules, since it's implemented as a list of rules. Since we will also be using it in our proofs, we will need an analogous inductive definition. Figure 3.11 defines this predicate, which also receives a grammar and a list of patterns, and yields a Boolean value indicating whether all patterns in the list are free of left recursion.

This predicate differs from the function in one aspect. While the function provides an exact value for the stack depth limit, the predicate allows any stack depth limit to identify a pattern as either nullable or non-nullable. That is because, according to Lemma 3.12, the returned label stays constant with increasing stack depth limits in such cases. In the general case, however, a stack depth limit greater than the number of rules in the grammar is necessary.

We prove some lemmas about this predicate. Lemma 3.20 states that this predicate is deterministic, and Lemma 3.21 states that it follows the fixed-point definition.

Lemma 3.20. *If* $(g, rs) \xrightarrow{lvr} b_1$ *and* $(g, rs) \xrightarrow{lvr} b_2$*, then* $b_1 = b_2$.

Lemma 3.21. *If* $lverifyrule\ g\ rs\ gas = Some\ b$*, then* $(g, rs) \xrightarrow{lvr} b$.

Lemma 3.22 states that, if a list of patterns passes the check, then every pattern in the list passes the individual check, being either nullable or non-nullable.

Lemma 3.22. *If* $(g, rs) \xrightarrow{lvr} true$*,*
then, $\forall r \in rs, \exists d\ \exists b\ \exists k\ (g, r, d, nb) \xrightarrow{vr} (Some\ b, k)$.

Before we end this section, there is one final lemma we would like to present, which uses all the predicates of the verification algorithm we have defined up until now. Lemma 3.23 shows that, if a grammar is free of incoherent and left-recursive rules, then any coherent pattern is either nullable or non-nullable.

Lemma 3.23. *If* $(g, p) \xrightarrow{c} true$*, and* $(g, g) \xrightarrow{lc} true$*,*
and $(g, g) \xrightarrow{lvr} true$*, then* $\exists d\ \exists b\ \exists k$*, such that* $(g, p, d, nb) \xrightarrow{vr} (Some\ b, k)$.

3.3 A simpler algorithm for detecting nullable patterns

Once we have made sure a grammar is free of left-recursive rules, the next step is to check for degenerate loops, which involves checking if certain patterns are nullable. To this end, we could use the algorithm we have just described in Section 3.2, but LPeg implements a simpler version, which takes advantage of the fact that the grammar contains no left-recursive rules.

The difference between the algorithm from Section 3.2 and this simpler version can be seen, for example, in the case of choice patterns $p_1\ /\ p_2$. More specifically, if p_1 is nullable, the function from Section 3.2 would still need to evaluate p_2, as it could potentially lead to left recursion. Meanwhile, if we assume that p_2 cannot lead to left recursion, then if p_1 is nullable, we can state that $p_1\ /\ p_2$ is nullable, without having to visit p_2. We can also avoid visiting sub-patterns in the cases of repetition patterns and predicate patterns, because they are all nullable.

In Figure 3.12, we define this simpler version as a fixed-point, which

```
nullable $g\ p\ d\ 0 = None$
nullable $g\ p\ d\ (1 + gas) =$
match $p$ with
| $\varepsilon \Rightarrow Some\ Some\ true$
| $[cs] \Rightarrow Some\ Some\ false$
| $p^\star \Rightarrow Some\ Some\ true$
| $!p \Rightarrow Some\ Some\ true$
| $\&p \Rightarrow Some\ Some\ true$
| $R_i \Rightarrow$ match $g[i]$ with
        | $Some\ p \Rightarrow$ match $d$ with
                    | $0 \Rightarrow Some\ None$
                    | $1 + d' \Rightarrow$ nullable $g\ p\ d'\ gas$
                    end
        | $None \Rightarrow None$
        end
| $p_1\ p_2 \Rightarrow$ match nullable $g\ p_1\ d\ gas$ with
          | $Some\ Some\ true \Rightarrow$ nullable $g\ p_2\ d\ gas$
          | $res \Rightarrow res$
          end
| $p_1\ /\ p_2 \Rightarrow$ match nullable $g\ p_1\ d\ gas$ with
            | $Some\ Some\ false \Rightarrow$ nullable $g\ p_2\ d\ gas$
            | $res \Rightarrow res$
            end
end
```

Figure 3.12: The nullable function.

takes a grammar, a pattern, a stack depth limit, and a gas counter, and returns an optional label. Possible return values are, therefore, *None* (out-of-gas), *Some None* (stack overflow), *Some Some true* (nullable), and *Some Some false* (non-nullable).

The gas parameter is still necessary to convince Coq that the function terminates, and because it could be called with an incoherent pattern or with a grammar that contains an incoherent rule. The stack depth limit is also necessary because the function could be called with a grammar that contains a left-recursive rule. For these two reasons, this function may still return *None* (out-of-gas) or *Some None* (stack overflow).

In reality, however, this function should only be called after the grammar and pattern are guaranteed to be coherent and free of left-recursive rules. For this reason, implementations of this algorithm such as the one in LPeg are able to safely drop both parameters, and return just a Boolean value indicating whether the pattern is nullable or not.

Also note that, unlike the function from Figure 3.7, this function does not receive a nullable accumulator as parameter and does not return a stack. The nullable accumulator is not necessary, thanks to the assumption that the grammar contains no left-recursive rules, and the output stack was only used for proofs, which we will be able to reuse from Section 3.2.

Just as we did for the function from Section 3.2, we prove that this simpler version is also monotonic with respect to the gas counter.

Lemma 3.24. *If $nullable\ g\ p\ d\ gas = Some\ res$,*
then, $\forall gas' \geq gas$, $nullable\ g\ p\ d\ gas' = Some\ res$.

Besides that, we also prove the function is eventually constant with respect to the gas counter by giving a gas lower bound for which the function returns some result for any coherent pattern and grammar.

Lemma 3.25. *If $(g, p) \xrightarrow{c} true$, and $(g, g) \xrightarrow{lc} true$,*
then, $\forall gas \geq ||p|| + d \cdot ||g||$, $\exists res\ nullable\ g\ p\ d\ gas = Some\ res$.

Furthermore, we would also like to prove that this function is eventually constant and monotonic with respect to the stack depth limit. However, in order to do that, we found it better to first define an equivalent inductive predicate. Figure 3.13 defines the predicate $(g, p, d) \xrightarrow{n} res$ which takes a grammar g, a pattern p, a stack depth limit d, and returns a label res.

About this predicate, we proved some basic lemmas. First, we proved that it is deterministic.

Lemma 3.26. *If $(g, p, d) \xrightarrow{n} res_1$, and $(g, p, d) \xrightarrow{n} res_2$, then $res_1 = res_2$.*

We also proved that it follows the fixed-point definition.

Lemma 3.27. *If $nullable\ g\ p\ d\ gas = Some\ res$, then $(g, p, d) \xrightarrow{n} res$.*

We also tied this predicate to the one from Section 3.2, showing how similar they are, when the nullable accumulator is $false$, and the pattern is either nullable or non-nullable.

Lemma 3.28. *If $(g, p, d, false) \xrightarrow{vr} (Some\ b, k)$, then $(g, p, d) \xrightarrow{n} Some\ b$.*

We also proved that if a pattern was identified as either nullable or non-nullable, then increasing the stack depth does not impact the result.

Lemma 3.29. *If $(g, p, d) \xrightarrow{n} Some\ b$, then $\forall d' \geq d, (g, p, d') \xrightarrow{n} Some\ b$.*

$$\frac{}{(g, \varepsilon, d) \xrightarrow{n} Some\ true}\ (\textsc{n-eps}) \qquad \frac{}{(g, [cs], d) \xrightarrow{n} Some\ false}\ (\textsc{n-set})$$

$$\frac{}{(g, p^\star, d) \xrightarrow{n} Some\ true}\ (\textsc{n-rep}) \qquad \frac{}{(g, !p, d) \xrightarrow{n} Some\ true}\ (\textsc{n-not})$$

$$\frac{}{(g, \&p, d) \xrightarrow{n} Some\ true}\ (\textsc{n-and})$$

$$\frac{g[i] = Some\ p}{(g, R_i, 0) \xrightarrow{n} None}\ (\textsc{n-nonterminal-zero})$$

$$\frac{g[i] = Some\ p \qquad (g, p, d) \xrightarrow{n} res}{(g, R_i, 1 + d) \xrightarrow{n} res}\ (\textsc{n-nonterminal-succ})$$

$$\frac{(g, p_1, d) \xrightarrow{n} None}{(g, p_1\ p_2, d) \xrightarrow{n} None}\ (\textsc{n-seq-none})$$

$$\frac{(g, p_1, d) \xrightarrow{n} Some\ false}{(g, p_1\ p_2, d) \xrightarrow{n} Some\ false}\ (\textsc{n-seq-some-false})$$

$$\frac{(g, p_1, d) \xrightarrow{n} Some\ true \qquad (g, p_2, d) \xrightarrow{n} res}{(g, p_1\ p_2, d) \xrightarrow{n} res}\ (\textsc{n-seq-some-true})$$

$$\frac{(g, p_1, d) \xrightarrow{n} None}{(g, p_1\ /\ p_2, d) \xrightarrow{n} None}\ (\textsc{n-choice-none})$$

$$\frac{(g, p_1, d) \xrightarrow{n} Some\ false \qquad (g, p_2, d) \xrightarrow{n} res}{(g, p_1\ /\ p_2, d) \xrightarrow{n} res}\ (\textsc{n-choice-some-false})$$

$$\frac{(g, p_1, d) \xrightarrow{n} Some\ true}{(g, p_1\ /\ p_2, d) \xrightarrow{n} Some\ true}\ (\textsc{n-choice-some-true})$$

Figure 3.13: The nullable predicate.

Maybe the most important lemma about the nullable predicate relates to the match predicate. It states that a non-nullable pattern never matches without consuming some part of the input string.

Lemma 3.30. *If* $(g, p, d) \xrightarrow{n} Some\ false$*, then* $\nexists s$ *such that* $(g, p, s) \xrightarrow{m} s$*.*

This lemma is used to prove that, when matching a non-nullable pattern, the output string is a proper suffix of the input string. We use the symbol "$\prec$" to denote this relation.

Lemma 3.31. *If* $(g, p, d) \xrightarrow{n} Some\ false$*, and* $(g, p, s) \xrightarrow{m} s'$*, then* $s' \prec s$*.*

Finally, we show that the predicate is eventually constant with respect to the stack depth limit, past a lower bound given by the number of rules in the grammar, denoted as $|g|$.

Lemma 3.32. *If* $(g, p) \xrightarrow{c} true$*, and* $(g, g) \xrightarrow{lc} true$*, and* $(g, g) \xrightarrow{lvr} true$*, and* $(g, p, d) \xrightarrow{n} res$*, where* $d > |g|$*, then,* $\forall d' \geq d$*,* $(g, p, d') \xrightarrow{n} res$*.*

3.4 Degenerate loops

After making sure that all rules are coherent, and that the grammar is free of left-recursive rules, the next step is to look for degenerate loops, which are repetition patterns $p^\star$ where p is nullable. To detect nullable patterns, we use the algorithm from Section 3.3.

Figure 3.14 defines this step of the verification process as a fixed-point, which takes a grammar, a pattern, a stack depth limit, and a gas counter, and returns an optional label. Possible return values are $None$ (out-of-gas), $Some\ None$ (stack overflow), $Some\ Some\ true$ (degenerate), and $Some\ Some\ false$ (non-degenerate).

This fixed-point does not visit rules referenced by nonterminal patterns. Instead, each rule is checked separately. In this case, the stack depth limit parameter is simply passed down on to the function that checks whether a pattern is nullable or not. However, as we've discussed in Section 3.3, actual implementations can safely drop this parameter, given that the grammar has been checked for left-recursive rules already.

Just as with the other gas-based functions, we would like to prove that this function is monotonic with respect to the gas counter. Lemma 3.33 states that, if this function returns some label, then increasing the gas counter won't change the returned label.

Lemma 3.33. *If* $checkloops\ g\ p\ d\ gas = Some\ res$*, then,* $\forall gas' \geq gas$*,* $checkloops\ g\ p\ d\ gas' = Some\ res$*.*

```
checkloops $g$ $p$ $d$ $0 = None$
checkloops $g$ $p$ $d$ $(1 + gas) =$
**match** $p$ **with**
| $\varepsilon \Rightarrow Some\ Some\ false$
| $[cs] \Rightarrow Some\ Some\ false$
| $R_i \Rightarrow Some\ Some\ false$
| $!p \Rightarrow$ checkloops $g$ $p$ $d$ $gas$
| $\&p \Rightarrow$ checkloops $g$ $p$ $d$ $gas$
| $p^\star \Rightarrow$ **match** nullable $g$ $p$ $d$ $gas$ **with**
        | $Some\ Some\ false \Rightarrow$ checkloops $g$ $p$ $d$ $gas$
        | $res \Rightarrow res$
        **end**
| $p_1\ p_2 \Rightarrow$ **match** checkloops $g$ $p_1$ $d$ $gas$ **with**
          | $Some\ Some\ false \Rightarrow$ checkloops $g$ $p_2$ $d$ $gas$
          | $res \Rightarrow res$
          **end**
| $p_1\ /\ p_2 \Rightarrow$ **match** checkloops $g$ $p_1$ $d$ $gas$ **with**
            | $Some\ Some\ false \Rightarrow$ checkloops $g$ $p_2$ $d$ $gas$
            | $res \Rightarrow res$
            **end**
**end**
```

Figure 3.14: The degenerate loop detection function.

Moreover, Lemma 3.34 states that, for any coherent pattern and grammar, there exists a lower bound for the gas parameter, for which the function returns some result.

Lemma 3.34. *If* $(g, p) \xrightarrow{c} true$*, and* $(g, g) \xrightarrow{lc} true$*, and* $gas \geq ||p|| + d \cdot ||g||$*, then* $\exists res\ checkloops\ g\ p\ d\ gas = Some\ res$*.*

We would also like to prove that this function is monotonic and eventually constant with respect to the stack depth limit. However, in order to do that, it is better to work with an inductively-defined predicate. Figure 3.15 shows the predicate we have defined. It takes a grammar, a pattern, and a stack depth limit, and returns an optional Boolean value.

As usual, we first prove some basic lemmas about the predicate. Lemma 3.35 states that the predicate is deterministic, meaning that, for the same input, it yields the same output.

Lemma 3.35. *If* $(g, p, d) \xrightarrow{cl} res_1$*, and* $(g, p, d) \xrightarrow{cl} res_2$*, then* $res_1 = res_2$*.*

$$\frac{}{(g, \varepsilon, d) \xrightarrow{cl} Some\ false}\ (\textsc{cl-eps}) \qquad \frac{}{(g, [cs], d) \xrightarrow{cl} Some\ false}\ (\textsc{cl-set})$$

$$\frac{}{(g, R_i, d) \xrightarrow{cl} Some\ false}\ (\textsc{cl-nonterminal}) \qquad \frac{(g, p, d) \xrightarrow{cl} res}{(g, !p, d) \xrightarrow{cl} res}\ (\textsc{cl-not})$$

$$\frac{(g, p, d) \xrightarrow{cl} res}{(g, \&p, d) \xrightarrow{cl} res}\ (\textsc{cl-and}) \qquad \frac{(g, p, d) \xrightarrow{n} None}{(g, p^\star, d) \xrightarrow{cl} None}\ (\textsc{cl-rep-lr})$$

$$\frac{(g, p, d) \xrightarrow{n} Some\ true}{(g, p^\star, d) \xrightarrow{cl} Some\ true}\ (\textsc{cl-rep-nullable})$$

$$\frac{(g, p, d) \xrightarrow{n} Some\ false \qquad (g, p, d) \xrightarrow{cl} res}{(g, p^\star, d) \xrightarrow{cl} res}\ (\textsc{cl-rep-non-nullable})$$

$$\frac{(g, p_1, d) \xrightarrow{cl} None}{(g, p_1\ p_2, d) \xrightarrow{cl} None}\ (\textsc{cl-seq-none-1})$$

$$\frac{(g, p_2, d) \xrightarrow{cl} None}{(g, p_1\ p_2, d) \xrightarrow{cl} None}\ (\textsc{cl-seq-none-2})$$

$$\frac{(g, p_1, d) \xrightarrow{cl} Some\ b_1 \qquad (g, p_2, d) \xrightarrow{cl} Some\ b_2}{(g, p_1\ p_2, d) \xrightarrow{cl} Some\ (b_1 \vee b_2)}\ (\textsc{cl-seq-some})$$

$$\frac{(g, p_1, d) \xrightarrow{cl} None}{(g, p_1\ /\ p_2, d) \xrightarrow{cl} None}\ (\textsc{cl-choice-none-1})$$

$$\frac{(g, p_2, d) \xrightarrow{cl} None}{(g, p_1\ /\ p_2, d) \xrightarrow{cl} None}\ (\textsc{cl-choice-none-2})$$

$$\frac{(g, p_1, d) \xrightarrow{cl} Some\ b_1 \qquad (g, p_2, d) \xrightarrow{cl} Some\ b_2}{(g, p_1\ /\ p_2, d) \xrightarrow{cl} Some\ (b_1 \vee b_2)}\ (\textsc{cl-choice-some})$$

Figure 3.15: The degenerate loop detection predicate.

```
lcheckloops $g\ rs\ gas$ = match $rs$ with
                           | $nil \Rightarrow Some\ false$
                           | $r :: rs' \Rightarrow$ let $d$ := $|g| + 1$ in
                                      match checkloops $g\ r\ d\ gas$ with
                                      | $Some\ Some\ false \Rightarrow$ lcheckloops $g\ rs'\ gas$
                                      | $Some\ Some\ true \Rightarrow Some\ true$
                                      | $res \Rightarrow None$
                                      end
                           end
```

Figure 3.16: The degenerate loop detection function for lists of patterns.

Lemma 3.36 states that every result returned by the function can be constructed using the predicate. We therefore say the predicate follows the function.

Lemma 3.36. *If checkloops $g\ p\ d\ gas = Some\ res$, then $(g, p, d) \xrightarrow{cl} res$.*

Lemma 3.37 states that, if the predicate yields some result, then increasing the stack depth limit will not alter the result.

Lemma 3.37. *If $(g, p, d) \xrightarrow{cl} Some\ res$, then, $\forall d' \geq d$, $(g, p, d') \xrightarrow{cl} Some\ res$.*

Finally, Lemma 3.38 states that, for any coherent pattern and grammar without left-recursive rules, the label returned by the predicate is constant when the stack depth limit is greater than the number of rules in the grammar.

Lemma 3.38. *If $(g, p) \xrightarrow{c} true$, and $(g, g) \xrightarrow{lc} true$, and $(g, g) \xrightarrow{lvr} true$, and $(g, p, d) \xrightarrow{cl} res$, where $d > |g|$, then, $\forall d' \geq d, (g, p, d') \xrightarrow{cl} res$.*

Having defined the algorithm that checks if a pattern contains any degenerate loops, we now define a function that performs this check for a list of patterns. Naturally, we will be using this function to check all the rules of a grammar. Figure 3.16 displays this function, which takes a grammar, a list of patterns, and a gas counter, and returns an optional Boolean value, indicating whether it has found any degenerate loop. We pass $|g| + 1$ as the stack depth limit to the underlying function.

We prove that there is a lower bound for the gas counter for which this function returns some result, assuming the grammar contains no incoherent or left-recursive rules, and that the list of patterns only contains coherent patterns. In reality, we will be calling this function while passing g as the rs parameter, so, in our case, it would suffice to state that g contains no incoherent or left-recursive rules.

$$\frac{}{(g, nil) \xrightarrow{lcl} false} \quad (\textsc{lcl-nil})$$

$$\frac{(g, r, d) \xrightarrow{cl} Some\ b_1 \qquad (g, rs) \xrightarrow{lcl} b_2}{(g, r :: rs) \xrightarrow{lcl} b_1 \lor b_2} \quad (\textsc{lcl-cons})$$

Figure 3.17: The degenerate loop detection predicate for lists of patterns.

Lemma 3.39. *If $(g, g) \xrightarrow{lc} true$, and $(g, rs) \xrightarrow{lc} true$, and $(g, g) \xrightarrow{lvr} true$, then, $\forall gas \geq ||rs|| + (|g| + 1) \cdot ||g||$, $\exists b$ lcheckloops g rs gas = Some b.*

In order to abstract away the gas counter, and to help us in later induction proofs, we also define an equivalent inductive predicate for this list-based degenerate loop checker. Figure 3.17 displays this predicate, which takes a grammar and a list of patterns, and returns a Boolean value, indicating whether none of the patterns in the list contain a degenerate loop.

As usual, we prove some basic lemmas about this predicate. Lemma 3.40 states that it is deterministic, and Lemma 3.41 states that it follows the fixed-point definition.

Lemma 3.40. *If $(g, rs) \xrightarrow{lcl} b_1$, and $(g, rs) \xrightarrow{lcl} b_2$, then $b_1 = b_2$.*

Lemma 3.41. *If lcheckloops g rs gas = Some b, then $(g, rs) \xrightarrow{lcl} b$.*

We also prove that if a list of patterns passes this list-based check, then each pattern in this list also passes the individual check.

Lemma 3.42. *If $(g, rs) \xrightarrow{lcl} false$, then $\forall r \in rs, \exists d\ (g, r, d) \xrightarrow{cl} Some\ false$.*

3.5 Correctness

Having introduced each step of the well-formedness algorithm, we can now present the definition of the *verifygrammar* function, which implements the algorithm step-by-step. It starts by checking whether the grammar defines a first rule, and whether every rule in the grammar is coherent. It then makes sure the grammar contains no left-recurive rules, and no degenerate loops, in this order. Figure 3.18 displays the function.

Now, let us prove that the well-formedness check is correct. To do so, we first define an inductive predicate equivalent to the *verifygrammar* function to help us in proofs by induction. Figure 3.19 presents this predicate, which takes a grammar and returns a Boolean value indicating whether the grammar passes

```
verifygrammar $g$ $gas$ =
match coherent $g$ $R_0$ with
| $true \Rightarrow$ match lcoherent $g$ $g$ with
          | $true \Rightarrow$ match lverifyrule $g$ $g$ $gas$ with
                    | $Some\ true \Rightarrow$ match lcheckloops $g$ $g$ $gas$ with
                                      | $Some\ b \Rightarrow Some\ \neg b$
                                      | $None \Rightarrow None$
                                      end
                    | $res \Rightarrow res$
                    end
          | $false \Rightarrow Some\ false$
          end
| $false \Rightarrow Some\ false$
end
```

Figure 3.18: The well-formedness function with gas.

$$\frac{(g, R_0) \xrightarrow{c} false}{g \xrightarrow{vg} false}\ (\textsc{vg}1) \qquad \frac{(g, R_0) \xrightarrow{c} true \qquad (g, g) \xrightarrow{lc} false}{g \xrightarrow{vg} false}\ (\textsc{vg}2)$$

$$\frac{(g, R_0) \xrightarrow{c} true \qquad (g, g) \xrightarrow{lc} true \qquad (g, g) \xrightarrow{lvr} false}{g \xrightarrow{vg} false}\ (\textsc{vg}3)$$

$$\frac{(g, R_0) \xrightarrow{c} true \qquad (g, g) \xrightarrow{lc} true \qquad (g, g) \xrightarrow{lvr} true \qquad (g, g) \xrightarrow{lcl} true}{g \xrightarrow{vg} false}\ (\textsc{vg}4)$$

$$\frac{(g, R_0) \xrightarrow{c} true \qquad (g, g) \xrightarrow{lc} true \qquad (g, g) \xrightarrow{lvr} true \qquad (g, g) \xrightarrow{lcl} false}{g \xrightarrow{vg} true}\ (\textsc{vg}5)$$

Figure 3.19: The well-formedness predicate.

all the checks. It is based on the predicates presented in the previous sections. Lemma 3.43 states that the predicate is deterministic, and Lemma 3.44 states that it follows the fixed-point definition.

Lemma 3.43. *If* $g \xrightarrow{vg} b_1$*, and* $g \xrightarrow{vg} b_2$*, then* $b_1 = b_2$*.*

Lemma 3.44. *If* $verifygrammar\ g\ gas = Some\ b$*, then* $g \xrightarrow{vg} b$*.*

In order to prove correctness, we need to generalize the pattern from R_0 to any pattern p, and to break down the function wf into its separate steps. We need to make this generalization because the match predicate is defined recursively on the current pattern. The generalized theorem we need to prove is the following: Given a grammar g and a pattern p, if g only contains coherent rules that do not lead to left recursion and that do not have any degenerate loops, and if p is coherent and does not have degenerate loops, then $\forall s, \exists res\ (g, p, s) \xrightarrow{m} res$.

We begin the proof by doing a strong induction on n, the length of the input string s, which gives us the inductive hypothesis “IHn”. This hypothesis states that for any input string shorter than s and any pattern, we can assume that the match yields some result. From Lemma 3.23 and the assumption that the grammar contains no left-recursive rules, we can infer that the pattern also does not lead to left recursion. This gives us the inductive predicate $(g, p, d, nb) \xrightarrow{vr} (Some\ \ b, k)$, which tells us that p is either nullable or non-nullable. We do an induction on this predicate and handle each case separately.

Let us start with the basic patterns. The case of the empty pattern ε is trivial, as it matches any input string without consuming anything. The case of the character set pattern $[cs]$ is also simple. If s is the empty string, the pattern fails to match. Otherwise, the string may or may not begin with the character $a \in cs$. If it does, then it matches while consuming a. Otherwise, it fails to match.

The sequence pattern $p_1\ p_2$ has two cases: one in which p_1 is non-nullable and p_2 is not visited, and another in which p_1 is nullable and p_2 is visited. In both cases, we have an inductive hypothesis stating that p_1 has a match result for input string s. If this match result is a failure, then the whole sequence $p_1\ p_2$ also fails. If the match result is a success, then p_1 leaves a suffix string s' unconsumed. Let us handle this case for both scenarios separately.

If p_1 is non-nullable, then we can use Lemma 3.31 to state that s' is a proper suffix of s, and therefore shorter than s. This allows us to use IHn, and state that p_2 has a match result for the input string s'. This implies that the sequence has this same match result.

Otherwise, if p_1 is nullable, then we can use Lemma 2.2 to state that s' is a suffix of s. This means that s' is either equal to s, or a proper suffix of s. If s is equal to s', then we can use the inductive hypothesis to state that p_2 yields a match result for $s' = s$. Otherwise, then we can use IHn in the same way as the previous case, because s' would be shorter than s.

Now let us consider the case of the choice pattern p_1 / p_2. Similar to the case of the sequence pattern, we have induction hypotheses for p_1 and p_2 yielding a match result for the input string s. This case is simpler because s is the same input string for both choice options, so these induction hypotheses are enough to prove this case.

The case of the repetition pattern $p^\star$ is the most interesting one, as we get to use the fact that p must be non-nullable in the proof. From the induction step, we have the inductive hypothesis that p yields a match result for the input string s. If p fails to match, then $p^\star$ matches without consuming anything. If, otherwise, p matches, then it leaves a string s' unconsumed. Since p is non-nullable, s' must be a proper suffix of s. Therefore, we can use IHn to state that $p^\star$ yields a match result res for s', because it is shorter than s. In this case, $p^\star$ also yields res for s.

The case of the predicates !p and &p are pretty straightforward. We first use the inductive hypothesis that p yields a match result for the input string s. If p matches, then &p matches without consuming anything, and !p fails to match. Otherwise, if p fails to match, then so does &p, and !p matches without consuming anything.

Finally, we prove the case of the non-terminal pattern R_i, which is surprisingly simple. From the induction step, we are given p, the i^{th} rule of the grammar. From the initial hypotheses, we know that p is coherent and free of degenerate loops, because it is a grammar rule. We can then use the inductive hypothesis from the verifyrule predicate to state that p yields a match result for the input string s.

Theorem 3.45. *Given a grammar g and a pattern p, if g only contains coherent rules that do not lead to left recursion and that free of degenerate loops, and if p is coherent and free of degenerate loops, then $\forall s, \exists res\ (g, p, s) \xrightarrow{m} res$.*

Having proved Theorem 3.45, we can finally prove the original theorem, which states that, for any grammar g that satisfies wf $g = true$, and for any input string, the non-terminal pattern R_0 yields a match result.

Lemma 3.1 states that the function *verifygrammar* returns *Some b* when given a gas counter greater or equal to $mingas_{vg}(g)$. In the implementation of the function *wf*, we pass $mingas_{vg}(g)$ as the gas counter for the function *verify-*

grammar. Therefore, if *wf* returns $true$, then it must be because *verifygrammar* returned $Some\ true$.

If the function *verifygrammar* returns $Some\ true$, then, according to Lemma 3.44, we can construct the predicate $g \xrightarrow{vg} true$. We can see that this only happens when the input grammar has passed all the checks. From Figure 3.19, we can see that this implies in several other predicates, many of which are necessary to use Theorem 3.45. There are only two missing predicates: $(g, R_0) \xrightarrow{c} true$, which states that the initial pattern is coherent, and $(g, R_0, d) \xrightarrow{cl} false$, which states that it does not contain any degenerate loops.

We can derive both predicates from the fact that R_0 is a rule, which we have checked already. We use Lemma 3.7 to prove that R_0 is coherent, and Lemma 3.42 to prove that it contains no degenerate loops. With this, we are able to prove the original theorem.

Theorem 3.46. *For any grammar g, if wf $g = true$, then g is complete.*

4 First-set Algorithm

In Chapter 3, we presented the well-formedness check implemented in LPeg, which ensures that the input PEG is complete. However, this is not the only role of this algorithm: It also ensures that other algorithms implemented in LPeg terminate, as they traverse patterns in similar ways. This chapter presents one such algorithm, which we label as the first-set algorithm.

The first-set algorithm is responsible for computing the set of first characters that can possibly be accepted by a pattern, and a Boolean value that indicates whether the pattern may accept the empty string. We call this Boolean value the *emptiness* value of the pattern. The key properties of the algorithm, which we later prove, are the following: Any pattern fails any string that starts with a character that is not in the first-set of the pattern; and the pattern fails the empty string if its emptiness value is $false$.

Both are conservative approximations, which means that a full first-set and an emptiness value of *true* are the safest options, yet the least useful ones. That is because LPeg uses this algorithm and its properties to optimize certain patterns in the code generation phase. So, ideally, we would like the first-set to be as small as possible, and the emptiness value of $false$ as common as possible, so that LPeg is able to optimize code more often.

One of the patterns that LPeg tries to optimize using this algorithm is the ordered choice. As a base reference, we display below the code that LPeg generates for an ordered choice p_1 / p_2 without any optimizations.

$$\begin{aligned}
& Choice(L_2) \\
& p_1 \\
& Commit(L_{end}) \\
L_2 :\ & p_2 \\
L_{end} :\ &
\end{aligned}$$

This unoptimized code starts with a choice instruction, which creates a checkpoint for the initial match state, so that, if p_1 fails, it is able to restore this state before running p_2. If, instead, p_1 succeeds, it runs a commit instruction, which deletes this checkpoint and jumps to L_{end}.

However, if p_1 and p_2 have disjoint first-sets, and p_1 has an emptiness value of $false$, then LPeg generates the following optimized code.

$$\begin{aligned} & TestSet(first(p_1), L_2) \\ & p_1 \\ & Jump(L_{end}) \\ L_2 :\ & p_2 \\ L_{end} :\ & \end{aligned}$$

This optimized code begins with an instruction (test set) that checks whether the input string starts with a character in the first-set of p_1, and jumps to L_2 if not. In the case where the input string is empty, we know that p_1 would fail, because p_1 has an emptiness value of $false$. If, otherwise, the input string starts with a character that is not in the first-set of p_1, then we know that p_1 would also fail, by the key property of first-sets. So, in either case, we know that p_1 would fail, making the choice pattern equivalent to p_2. That is why, in the case of failure, the test-set instruction jumps to the code of p_2, without ever running the code of p_1.

Meanwhile, the test-set instruction succeeds if the input string starts with a character that *is* in the first-set of p_1, and, therefore, is non-empty. In this case, because the first-sets of p_1 and p_2 are disjoint, we know that this character is *not* in the first-set of p_2. And, from the key property of first-sets, this means that p_2 would fail, making the choice pattern equivalent to p_1. So, in this case, LPeg simply executes the code of p_1.

This optimization significantly improves the performance LPeg when matching some choice patterns. This improvement is possible by replacing the costly creation/restoration/deletion of checkpoints with inexpensive jumps and inspections on the input string.

In this chapter, we discuss the first-set algorithm and prove its key properties.

4.1 Building intuition

Before we formally define the algorithm, we would like to first develop some intuition for the function signature. We start with an informal definition: the function receives a pattern and returns the set of first characters that can be accepted by the pattern. Let us try to informally define the first-set of some simple patterns using this initial signature.

Starting with ε and $p^\star$, we know both patterns accept every input string, so it makes sense for this function to return the full character set, which we represent as Σ. As for the character set pattern $[cs]$, we know that it only accepts strings that start with a character in the set $\{cs\}$.

$$\begin{aligned}\mathcal{F}(\varepsilon) &= \Sigma \\ \mathcal{F}(p^\star) &= \Sigma \\ \mathcal{F}([cs]) &= \{cs\}\end{aligned}$$

However, when we try to define the first-set of sequence patterns, we run into some issues. To better comprehend them, it can be helpful to mentally compute the first-set of some simple sequences, using the pattern types for which we have defined the function so far. This may give us some intuition as to how the first-set of sequences may be derived from its sub-patterns. It is worth highlighting that these definitions are informal and derived purely from intuition. The actual formal definitions are presented later in this chapter.

$$\begin{aligned}\mathcal{F}(\varepsilon\ \varepsilon) &= \Sigma \\ \mathcal{F}([cs_1]\ \varepsilon) &= \{cs_1\} \\ \mathcal{F}(\varepsilon\ [cs_2]) &= \{cs_2\} \\ \mathcal{F}([cs_1]\ [cs_2]) &= \{cs_1\} \\ \mathcal{F}([cs_1]\ [cs_2]^\star) &= \{cs_1\} \\ \mathcal{F}([cs_1]^\star\ \varepsilon) &= \Sigma \\ \mathcal{F}([cs_1]^\star\ [cs_2]) &= \{cs_1\} \cup \{cs_2\} \\ \mathcal{F}([cs_1]^\star\ [cs_2]^\star) &= \Sigma\end{aligned}$$

From the examples above, we can observe that when p_1 is non-nullable, the first-set of p_2 doesn't seem to be relevant to the first-set of the sequence. That is because p_1, being non-nullable, is guaranteed to consume a character. As a result, p_2 is given a proper suffix of the original string, which doesn't contain its first character. Based on this observation, we can define the function for sequences $p_1\ p_2$ where p_1 is non-nullable as the first-set of p_1.

$$\mathcal{F}(p_1\ p_2) = \begin{cases} \mathcal{F}(p_1) & \text{if } p_1 \text{ is non-nullable} \\ \ldots? & \text{if } p_1 \text{ is nullable} \end{cases}$$

We still need to define the function for the case where p_1 is nullable. From the patterns we have defined so far, only the empty pattern ε and the repetition pattern $[cs_1]^\star$ are nullable. Let us see their behavior in sequence with the character set pattern $[cs_2]$.

$$\begin{aligned}\mathcal{F}(\varepsilon\ [cs_2]) &= \{cs_2\}\\ \mathcal{F}([cs_1]^\star\ [cs_2]) &= \{cs_1\} \cup \{cs_2\}\end{aligned}$$

Note that, when alone, their first-sets are equal. However, when in sequence with the same pattern, the resulting first-sets differ. This discrepancy indicates that we cannot define $\mathcal{F}(p_1\ p_2)$ as a pure function of $\mathcal{F}(p_1)$ and $\mathcal{F}(p_2)$. Instead, we would have to break the definition down for each case of p_1.

$$\begin{aligned}\mathcal{F}(\varepsilon\ p_2) &= \mathcal{F}(p_2)\\ \mathcal{F}(p^\star\ p_2) &= \mathcal{F}(p) \cup \mathcal{F}(p_2)\\ &\ldots\end{aligned}$$

With this signature, we would have to define the function for every nullable pattern twice: one when alone, and another when followed by another pattern. However, we would like to define the function recursively only once for each pattern type.

To solve this issue, LPeg adds an accumulator parameter for the first-set of the following pattern in the sequence. When the pattern is not followed by a pattern, LPeg uses the full character set as the accumulator. We call this accumulator the *follow-set*. Let us adapt our definitions for this new signature: For each basic pattern p_1, we can derive the definition of $\mathcal{F}(p_1, follow)$ from the previous definition of $\mathcal{F}(p_1\ p_2)$, and by replacing $\mathcal{F}(p_2)$ with the new parameter $follow$.

$$\begin{aligned}\mathcal{F}(\varepsilon, follow) &= follow\\ \mathcal{F}(p^\star, follow) &= \mathcal{F}(p, follow) \cup follow\\ \mathcal{F}([cs], follow) &= \{cs\}\end{aligned}$$

The first-set of the character set pattern $[cs]$ does not use the follow parameter, because it doesn't depend on the first-set of the following pattern. This is because the character set pattern is non-nullable. We later prove this property for any non-nullable pattern.

The case of the sequence pattern is interesting, as it demonstrates the follow-set working as an accumulator: In the case where p_1 is nullable, we use the first-set of p_2 as the follow-set of p_1. As for the case in which p_1 is non-nullable, we use the first-set of p_1 with any follow-set. We choose the full character set as the follow-set to demonstrate this independence from the accumulator, but any character set could be used.

$$\mathcal{F}(p_1\ p_2, follow) = \begin{cases} \mathcal{F}(p_1, \mathcal{F}(p_2, follow)) & \text{if } p_1 \text{ is nullable} \\ \mathcal{F}(p_1, \Sigma) & \text{if } p_1 \text{ is non-nullable} \end{cases}$$

Let us now define the first-set of the other pattern types. Starting with the choice pattern p_1/p_2, we intuitively define it as the union of the first-sets of p_1 and p_2, passing down the follow-set parameter to each sub-call.

$$\mathcal{F}(p_1/p_2, follow) = \mathcal{F}(p_1, follow) \cup \mathcal{F}(p_2, follow)$$

The case of the not-predicate pattern $!p$ highlights the conservative nature of first-sets. From the first-set of p, we can only infer the set of first characters that make p fail, and, therefore, make $!p$ succeed. This, however, gives us no information about first characters that make p succeed, and, therefore, make $!p$ fail. Therefore, in general, we cannot use the first-set of p to compute the first-set of $!p$.

There is, however, information that we can extract from the follow-set parameter. When $!p$ is followed by a pattern p_2, the follow-set indicates which first characters make p_2 fail. Given that $!p$ doesn't consume any input, these first characters also make the sequence $!p\ p_2$ fail, so they should be part of the first-set of $!p$. Therefore, we could, in general, use the follow-set as the first-set of any predicate. However, for the specific case of $![cs]$, LPeg instead computes the first-set of patterns as $\Sigma \setminus \{cs\}$.

$$\mathcal{F}(!p, follow) = \begin{cases} \Sigma \setminus \{cs\} & \text{if } p = [cs] \\ follow & \text{otherwise} \end{cases}$$

One topic for future research is to investigate whether it would be possible to replace Σ with the provided follow-set parameter in LPeg.

As for the and-predicate pattern $\&p$, we use both the first-set of p and the follow-set. That is because in order to match the sequence $\&p\ p_2$, the input string must match both p and p_2. Conversely, if the string starts with a character that is not in the first-set of either pattern, the sequence fails. So, in this case, we define the first-set as the intersection of both sets.

$$\mathcal{F}(\&p, follow) = \mathcal{F}(p, \Sigma) \cap follow$$

One subtle difference between this definition and the actual implementation of the algorithm in LPeg is the follow-set parameter used to calculate the first-set of p. While LPeg simply passes along the follow-set parameter, we provide the full character set Σ. This change was necessary for us to prove the key property of first-sets in Coq, which we will show later in this chapter. Future research may check whether passing along the follow-set parameter is incorrect, equivalent, or even better than passing Σ as the follow-set.

Finally, in order to define the case of the non-terminal pattern, we need to add the grammar as a parameter, so that we can look up the referenced grammar rule. In all cases, this grammar parameter is simply passed along to each recursive call.

$$\mathcal{F}(g, R_i, follow) = \mathcal{F}(g, p, follow) \qquad \text{if } g[i] = Some\ p$$

This case brings up the topic of termination, as it does not define the recursion on the structure of the pattern, like the other cases do. Instead, termination in this case relies on the assumption that the input PEG is well-formed, and, therefore, free of left-recursive rules.

On a deeper level, termination is derived from the way in which the well-formedness and first-set algorithms traverse patterns similarly. The most interesting case is that of the sequence pattern: when p_1 is non-nullable, both algorithms do not visit p_2. In the case of the well-formedness check, visiting p_2 is not necessary because the whole sequence is non-nullable, and any rules visited in p_2 would be matched against a proper suffix of the input string s (avoiding infinite loops).

Meanwhile, in the case of the first-set algorithm, visiting p_2 can be avoided for two reasons: If p_1 is non-nullable, then, as we later prove, its emptiness value is $false$, which means that it fails the empty string. If p_1 fails the empty string, then so does the sequence, which allows the emptiness value of the sequence to also be $false$, regardless of the emptiness value of p_2. The second reason is that, if p_1 is non-nullable, then its first-set is independent of the follow-set parameter, which, in the general case, would be the first-set of p_2. Therefore, when p_1 is non-nullable, we can provide any follow-set parameter, such as Σ, in order to avoid making a recursive call to p_2.

4.2 Matching the empty string

Besides the first-set of a pattern, the algorithm implemented in LPeg also returns a Boolean value, which indicates whether the pattern may match the empty string, a property we call *emptiness*. It is another conservative approximation: the value *true* has no meaning, while the value $false$ indicates that the pattern fails to match the empty string. LPeg needs this information because it cannot use the first-set to verify whether a pattern fails the empty string, since the empty string has no first character.

Let us see how LPeg computes the emptiness of patterns. The base cases are quite simple. The empty pattern ε and the repetition pattern $p^\star$ match every input string, which includes the empty string. So, for these patterns, the function returns $true$.

The character class pattern $[cs]$, as with any non-nullable pattern, does not match the empty string. Therefore, the value for this pattern is $false$.

For the not-predicate pattern $!p$, LPeg is rather conservative, always returning $true$. In fact, this seems to be the only case making the emptiness value a conservative approximation. If instead this function were to call itself recursively for p and negate its emptiness value, we would effectively compute whether the pattern matches the empty string. However, the cases of $!p$ where p matches the empty string are not common nor useful in practice.

The and-predicate pattern $\&p$ and the non-terminal pattern R_i simply forward the Boolean value from the underlying pattern, because they fail if and only if the underlying pattern fails.

The sequence pattern $p_1\,p_2$ matches the empty string if both p_1 and p_2 do. Intuitively, this would mean that the emptiness value of the sequence would be the Boolean AND of the emptiness values of p_1 and p_2, but that is not exactly what is implemented in the algorithm. As we have discussed at the end of the previous section, when p_1 is non-nullable, p_2 is not visited, and, therefore, the emptiness value of p_2 is not calculated. However, we don't need this value, since the emptiness value of p_1 is $false$ in this case, which allows for the short-circuit evaluation of the Boolean AND expression to $false$. Meanwhile, when p_1 is nullable, the emptiness value of both p_1 and p_2 are computed, and their Boolean AND is calculated normally.

Finally, the case of the choice pattern p_1/p_2 is similar to that of the sequence pattern, but instead of a Boolean AND operation, it performs a Boolean OR of the emptiness values of p_1 and p_2. That is because the choice matches the empty string if one of the options does.

4.3 Formal definition

Figure 4.1 presents the formal definition of the first-set algorithm. It takes a grammar, a pattern, a follow-set, and some gas, and returns an optional tuple. The recursion is defined on the gas parameter, so that, if it reaches zero, the function returns $None$. Otherwise, the function returns $Some\ (b, first)$, where

```
$\mathcal{F}\ g\ p\ follow\ 0 = None$
$\mathcal{F}\ g\ p\ follow\ (1 + gas) =$
match $p$ with
| $\varepsilon \Rightarrow Some\ (true, follow)$
| $[cs] \Rightarrow Some\ (false, \{cs\})$
| $p^\star \Rightarrow$ match $\mathcal{F}\ g\ p\ follow\ gas$ with
        | $Some\ (b, first) \Rightarrow Some\ (true, first \cup follow)$
        | $None \Rightarrow None$
        end
| $!p \Rightarrow$ match $p$ with
        | $[cs] \Rightarrow Some\ (true, \Sigma \setminus \{cs\})$
        | $otherwise \Rightarrow Some\ (true, follow)$
        end
| $\&p \Rightarrow$ match $\mathcal{F}\ g\ p\ \Sigma\ gas$ with
         | $Some\ (b, first) \Rightarrow Some\ (b, first \cap follow)$
         | $None \Rightarrow None$
         end
| $R_i \Rightarrow$ match $g[i]$ with
        | $Some\ p \Rightarrow \mathcal{F}\ g\ p\ follow\ gas$
        | $None \Rightarrow None$
        end
| $p_1\ p_2 \Rightarrow$ match nullable $g\ p_1\ gas$ with
           | $Some\ false \Rightarrow \mathcal{F}\ g\ p_1\ \Sigma\ gas$
           | $Some\ true \Rightarrow$ match $\mathcal{F}\ g\ p_2\ follow\ gas$ with
                              | $Some\ (b_2, first_2) \Rightarrow b_2 \otimes (\mathcal{F}\ g\ p_1\ first_2\ gas)$
                              | $None \Rightarrow None$
                              end
           | $None \Rightarrow None$
           end
| $p_1\ /\ p_2 \Rightarrow (\mathcal{F}\ g\ p_1\ follow\ gas) \oplus (\mathcal{F}\ g\ p_2\ follow\ gas)$
end
```

Figure 4.1: The first-set function.

```
$b \otimes res =$
match $res$ with
| $Some\ (b', first') \Rightarrow Some\ (b \wedge b', first')$
| $None \Rightarrow None$
end
```

```
$res_1 \oplus res_2 =$
match $res_1, res_2$ with
| $Some\ (b_1, first_1), Some\ (b_2, first_2) \Rightarrow Some\ (b_1 \vee b_2, first_1 \cup first_2)$
| $otherwise \Rightarrow None$
end
```

Figure 4.2: The auxiliary $\otimes$ and $\oplus$ functions.

b is the emptiness value, and $first$ is the first-set. If $b = false$, then the pattern fails to match the empty string; and if a string starts with a character $x \notin first$, then it is guaranteed to fail to match that string. The follow-set parameter is an accumulator that should be initialized with the full character set Σ. In order to improve the legibility of the function for sequence and choice patterns, we also define in Figure 4.2 the auxiliary functions $\otimes$ and $\oplus$, respectively.

Having formally defined the first-set algorithm, we now prove its key properties. We begin by proving that if the function returns some result for some gas amount, it will return the same result if you provide a higher gas amount. In some sense, this means the function is stable when you increase the gas amount.

Lemma 4.1. *If $\mathcal{F}\ g\ p\ follow\ gas = Some\ res$,*
then, $\forall gas' \geq gas, \mathcal{F}\ g\ p\ follow\ gas' = Some\ res$.

One natural consequence of this lemma is that, for the same grammar, pattern, and follow-set, the function cannot return contradicting results.

Lemma 4.2. *If $\mathcal{F}\ g\ p\ follow\ gas = Some\ res$,*
and $\mathcal{F}\ g\ p\ follow\ gas' = Some\ res'$,
then $res = res'$.

The previous two lemmas show how consistent the return of the function is, but both assume the existence of a gas amount for which the function returns some result. However, we know this is not always the case. In fact, for ill-formed grammars, it may return $None$ for any gas amount. So, it is important to prove that, for well-formed PEGs, there exists a lower bound for the gas amount, for which the function returns some result. This lower bound effectively shows that the algorithm terminates even without the gas parameter, as it is implemented in LPeg.

Lemma 4.3. *If* $wf\ g = true$,
then $\forall gas \geq ||p|| + (1 + |g|) \cdot ||g||$,
$\exists res, \mathcal{F}\ g\ p\ follow\ gas = Some\ res$.

Having proved termination, let us now focus on the key properties of the first-set algorithm, starting with the emptiness value. We prove that, for well-formed PEGs, if $b = false$, then the pattern fails the empty string (denoted as nil). Note that, in this case, the follow-set parameter is irrelevant.

Lemma 4.4. *If* $wf\ g = true$,
and $\mathcal{F}\ g\ p\ follow\ gas = Some\ (false, first)$,
then $(g, p, nil) \xrightarrow{m} \bot$.

Now, we prove lemmas about the relation between the follow-set parameter and the first-set return value. These lemmas are necessary to prove a more important lemma later in this section. We start by proving that if the follow-set parameter is incremented by an extra set (through a set union operation), then the first-set return value is incremented by a subset of this extra set. Note that the emptiness value stays the same with this follow-set increment.

Lemma 4.5. *If* $\mathcal{F}\ g\ p\ follow\ gas = Some\ (b, first)$,
then $\forall extra, \exists extra' \subseteq extra$,
such that $\mathcal{F}\ g\ p\ (follow \cup extra)\ gas = Some\ (b, (first \cup extra'))$.

A particular case is when this extra set is the first-set itself, as if it were fed back into the function through the follow-set parameter. In this case, the first-set output by the function is the same, since $first \cup extra' \equiv first$ when $extra' \subseteq first$. This particular lemma is the one we actually use to prove the more important lemma.

Lemma 4.6. *If* $\mathcal{F}\ g\ p\ follow\ gas = Some\ (b, first)$,
then $\mathcal{F}\ g\ p\ (follow \cup first)\ gas = Some\ (b, first)$.

The following lemma is the cornerstone of the key property of first-sets: If p matches some string s, leaving a suffix s' unconsumed, then s must be either empty or start with a character that *is* in the first-set of p. We also assume that the input PEG is well-formed, and that s' is either empty or starts with a character in the follow-set. This last assumption is necessary to prove the lemma in the case of the sequence pattern.

Lemma 4.7. *If $wf\ g = true$,*
and $\mathcal{F}\ g\ p\ follow\ gas = Some\ (b, first)$,
and $(g, p, s) \xrightarrow{m} s'$,
and s' either is empty or starts with $x \in follow$,
then s either is empty or starts with $y \in first$.

In the case of the and-predicate pattern $\&p$, we noticed that it would be easier to prove this lemma if we passed Σ as the follow-set of p. That is because $\&p$ matches when p matches, but p leaves an unconsumed suffix s' that is discarded and whose starting character (if non-empty) we know nothing about. Ultimately, we cannot say that s' is either empty or starts with a character in an arbitrary follow-set. Instead, we use Σ as the follow-set of p, as this turns this hypothesis into a tautology.

Finally, we prove the key property of first-sets: For a well-formed PEG, if the emptiness value is $false$, then the pattern fails for any string that does not start with a character in its first-set. Note that we use the full character set Σ as the follow-set.

Lemma 4.8. *If $wf\ g = true$,*
and $\mathcal{F}\ g\ p\ \Sigma\ gas = Some\ (false, first)$,
and s either is empty or starts with $x \notin first$,
then $(g, p, s) \xrightarrow{m} \bot$.

Besides this main property, we also prove that, for non-nullable patterns, the follow-set parameter does not influence the result.

Lemma 4.9. *If $nullable\ g\ p\ gas_n\ = Some\ false$,*
and $\mathcal{F}\ g\ p\ follow_1\ gas_1 = Some\ res_1$,
and $\mathcal{F}\ g\ p\ follow_2\ gas_2 = Some\ res_2$,
then $res_1 = res_2$.

This lemma explains why, in the cases of character set patterns and sequence patterns with non-nullable first patterns, the follow-set parameter can be completely ignored. We can also observe that, in the case of repetitions $p^\star$, LPeg passes along the follow-set parameter to p, but any follow-set could be provided, given that p is non-nullable from the well-formedness property.

Another fact about non-nullable patterns is that their emptiness value is always $false$. From the key property of emptiness values, this indicates that non-nullable pattern fail to match the empty string, which we know is true.

Lemma 4.10. *If $nullable\ g\ p\ gas_n\ = Some\ false$,*
and $\mathcal{F}\ g\ p\ follow\ gas = Some\ (b, first)$,
then $b = false$.

4.4 Application in LPeg

Having proved the key properties of the first-set algorithm, we would like to formalize its application in LPeg. As discussed at the beginning of this chapter, LPeg uses this algorithm when generating code for choice patterns, making use of test-set instructions. Despite this optimization occurring at the virtual machine code level, we would like to formalize it at the syntactic level.

The test-set instruction basically checks whether the input string starts with a character in a given set $\{cs\}$, jumping to a given label if it does not. We can check the first character of the input string through the character set pattern $[cs]$, and emulate the logic of "if p_{cond} matches, then try p_1, otherwise try p_2" through the following pattern construction.

$$\&p_{cond}\ p_1\ /\ !p_{cond}\ p_2$$

In the optimized code of the choice pattern, the test-instruction checks if the input string starts with a character in the first-set of p_1, and jumps to the code of p_2 if it does not. This instruction is followed by the code of p_1, which is executed if the check succeeds. We can represent this optimized code as the following pattern. Let $\{first_1\}$ denote the first-set of p_1.

$$\&[first_1]\ p_1\ /\ ![first_1]\ p_2$$

We now prove the correctness of this optimization. Assuming the grammar g and patterns p_1 and p_2 are well-formed, and that the first-sets of p_1 and p_2 are disjoint, and that the emptiness value of p_1 is $false$, we first prove that if the original choice pattern matches a string s, the optimized choice also matches s, yielding the same unconsumed suffix s'. We also need to assume that s' either is empty or starts with a character in the follow-set of p_2.

Lemma 4.11. *If g, p_1 and p_2 are well-formed,*
and s' either is empty or starts with $x \in follow$,
and $\mathcal{F}\ g\ p_1\ \Sigma\ gas_1 = Some\ (false, first_1)$,
and $\mathcal{F}\ g\ p_2\ follow\ gas_2 = Some\ (b, first_2)$,
and $first_1 \cap first_2 = \varnothing$,
and $(g, p_1\ /\ p_2, s) \xrightarrow{m} s'$,
then $(g, \&[first_1]\ p_1\ /\ ![first_1]\ p_2, s) \xrightarrow{m} s'$.

As for the case in which the choice fails, we also show the optimized choice fails as well. The proof follows from two facts: The choice fails either because of p_1 or p_2; And, for any input string, either $\&[first_1]$ matches and $![first_1]$ fails, or the other way around.

Lemma 4.12. *If* $(g, p_1\ /\ p_2, s) \xrightarrow{m} \perp$,
then $(g, \&[first_1]\ p_1\ /\ ![first_1]\ p_2, s) \xrightarrow{m} \perp$.

5
Related Work

Ford (1) introduced PEGs and provided initial theoretical results about them. He proved that the problem of knowing whether a PEG is complete is undecidable, and presented *well-formedness* as a conservative approximation to completeness. In this approximation, he defined the conservative notion of *nullable* expressions, which can accept an input string without consuming any characters. Ford's well-formedness algorithm is the main source of inspiration for LPeg's well-formedness algorithm. However, we did not prove whether they are equivalent. This would require formalizing Ford's algorithm in Coq.

Medeiros et al. (6) presented a conservative extension to the semantics of PEGs based on bounded left recursion and proved its correctness. They have also proved that every PEG in this extension is complete, assuming that every non-terminal is valid. We have chosen not to go in this direction, as our main goal was to formalize LPeg, which categorizes left-recursive rules as ill-formed.

Ribeiro et al. (7) formalized the syntax and semantics of PEGs using the Agda proof assistant. They also formalized the well-formedness verification process in a way similar to a typing procedure. Expressions are typed according to the set of nonterminals that can be reached without consuming any character (called *head-set*) and whether the expression is nullable, following Ford's definition.

The most interesting restrictions in this typing are those on nonterminal expressions and repetitions. For nonterminals, the typing prohibits the nonterminal itself from being contained in its head-set. For repetitions $p^\star$, it prohibits p from being nullable. Although correct, this method does not provide a direct algorithm for this verification, instead relying on a typing algorithm.

Koprowski et al. (4) developed TRX, a parser interpreter formalized using the Coq proof assistant. Their work extended PEGs to support semantic values and actions, and focused on extracting parsers from them with proofs of termination and correctness.

They formalized a well-formedness check for these extended PEGs that is largely based on Ford's original work: The algorithm iteratively computes a set of well-formed expressions until a fixed-point is reached, and then checks if this set coincides with the expression set of the grammar. Although proven correct, this algorithm seems harder to implement using low-level programming languages, when compared to the algorithm implemented in LPeg, which is written in C in order to better interact with the Lua C API.

Blaudeau et al. (8) specified a verified packrat parser interpreter for PEGs in PVS, emphasizing the formal verification of the parsing process. In order for their algorithm to correctly detect left recursion, they assume there exists a correct order for visiting non-terminals. However, they do not provide an algorithm for computing such an order.

In contrast, our work presents a direct and practical algorithm for verifying the well-formedness of PEGs. By providing concrete implementation details, we offer a more straightforward approach to well-formedness verification compared to the formal proof-based methods of the aforementioned works.

Another contribution of our work is the formalization of the first-set algorithm implemented in LPeg. Although the concept of first-sets is well-established in the area of context-free grammars (2), to the extent of our knowledge their application in PEGs has not been documented yet. We proved the key properties of the first-set algorithm, and the soundness of its application in LPeg, as an optimization technique during the code generation phase.

6 Conclusion

In this work, we formalized two key algorithms implemented in LPeg: the well-formedness check and the first-set computation. Both algorithms were defined as functions in Coq using a fixed-point construction, with the recursion being defined on a gas parameter. We proved that both algorithms terminate by providing a lower bound for the gas parameter. While the well-formedness check guarantees this property for any input PEG, the first-set computation assumes the input PEG has successfully passed the well-formedness check.

Besides proving their termination, we have proved these algorithms are correct, in their own respective ways. For the well-formedness check, we have proved that it correctly detects complete PEGs, which, in turn, guarantees that parsing terminates. Meanwhile, for the first-set algorithm, we have proven that it computes the set of first characters that make a pattern fail, and that it checks whether the pattern fails for the empty string.

Moreover, we used the properties of the first-set algorithm to prove that an optimization performed by LPeg on certain choice patterns is correct. This optimization is also performed on other types of patterns, but we leave the proof of their correctness as a topic for future research.

Still on the topic of future research, while formalizing these algorithms, we identified some details that deserve future review, as they could lead to future improvements in LPeg. In the particular case of the first-set algorithm, we modified the definition for the and-predicate pattern $\&p$ so that we would be able to prove the key properties of the algorithm. Future research should investigate whether it would be possible to prove these properties for the actual implementation in LPeg. Furthermore, we suspect the case of the pattern $![cs]$ could be reviewed to make the first-set even smaller, and, therefore, more likely to be used in optimizations.

Future work may also seek to measure the computational and space complexity of these algorithm in terms of some notion of grammar size, such as the total number of nodes in the abstract syntax tree of rules. Such measurements may even help us find opportunities for improvements.

7
Bibliography

1 FORD, B. Parsing Expression Grammars: A recognition-based syntactic foundation. In: **Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 2004. (POPL '04), p. 111–122. ISBN 978-1-58113-729-3.

2 CHOMSKY, N. Three models for the description of language. **IRE Transactions on Information Theory**, v. 2, n. 3, p. 113–124, 1956.

3 IERUSALIMSCHY, R. A text pattern-matching tool based on Parsing Expression Grammars. **Softw.: Practice and Experience**, v. 39, n. 3, p. 221–258, mar. 2009. ISSN 00380644, 1097024X.

4 KOPROWSKI, A.; BINSZTOK, H. TRX: A formally verified parser interpreter. **Logical Methods in Computer Science**, Volume 7, Issue 2, jun. 2011. ISSN 1860-5974. Publisher: Episciences.org. Disponível em: https://lmcs.episciences.org/686.

5 IERUSALIMSCHY, R. **LPeg - Parsing Expression Grammars for Lua**. 2024. Disponível em: https://www.inf.puc-rio.br/~roberto/lpeg/.

6 MEDEIROS, S.; MASCARENHAS, F.; IERUSALIMSCHY, R. Left recursion in parsing expression grammars. **Science of Computer Programming**, v. 96, p. 177–190, dez. 2014. ISSN 0167-6423. Disponível em: https://www.sciencedirect.com/science/article/pii/S0167642314000288.

7 RIBEIRO, R. et al. Towards typed semantics for Parsing Expression Grammars. In: **Proceedings of the XXIII Brazilian Symposium on Programming Languages**. New York, NY, USA: Association for Computing Machinery, 2019. (SBLP '19), p. 70–77. ISBN 978-1-4503-7638-9.

8 BLAUDEAU, C.; SHANKAR, N. A verified packrat parser interpreter for Parsing Expression Grammars. In: **Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs**. Association for Computing Machinery, 2020. (CPP 2020), p. 3–17. ISBN 978-1-4503-7097-4. Disponível em: https://dl.acm.org/doi/10.1145/3372885.3373836.