Science of Computer Programming 89 (2014) 235–250

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

On the relation between context-free grammars and parsing expression grammars

Fabio Mascarenhas [a,*], Sérgio Medeiros [b,*], Roberto Ierusalimschy [c,*]

[a] Department of Computer Science, UFRJ, Rio de Janeiro, Brazil
[b] School of Science and Technology, UFRN, Natal, Brazil
[c] Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil

HIGHLIGHTS

- We present new formalizations of CFGs and PEGs, based on natural semantics.
- We discuss the correspondence between the languages of CFGs and PEGs.
- We show transformations from classes of CFGs to equivalent PEGs.
- Transformations of LL(1), strong-LL(k) and LL-regular grammars are presented.
- The transformations preserve the structure of the original grammars.

ARTICLE INFO

Article history:
Received 29 July 2013
Received in revised form 13 January 2014
Accepted 20 January 2014
Available online 30 January 2014

Keywords:
Context-free grammars
Parsing expression grammars
LL(1)
LL(k)
Natural semantics

ABSTRACT

Context-Free Grammars (CFGs) and Parsing Expression Grammars (PEGs) have several similarities and a few differences in both their syntax and semantics, but they are usually presented through formalisms that hinder a proper comparison. In this paper we present a new formalism for CFGs that highlights the similarities and differences between them. The new formalism borrows from PEGs the use of *parsing expressions* and the recognition-based semantics. We show how one way of removing non-determinism from this formalism yields a formalism with the semantics of PEGs. We also prove, based on these new formalisms, how LL(1) grammars define the same language whether interpreted as CFGs or as PEGs, and also show how strong-LL(k), right-linear, and LL-regular grammars have simple language-preserving translations from CFGs to PEGs. Once these classes of CFGs can be automatically translated to equivalent PEGs, we can reuse classic top-down grammars in PEG-based tools.

1. Introduction

Context-Free Grammars (CFGs) are the formalism of choice for describing the syntax of programming languages. A CFG describes a language as the set of strings generated from the grammar's initial symbol by a sequence of rewriting steps. CFGs do not, however, specify a method for efficiently recognizing whether an arbitrary string belongs to its language. In other words, a CFG does not specify how to *parse* the language, an essential operation for working with the language (in a compiler, for example). Another problem with CFGs is ambiguity, where a string can have more than one parse tree.

Parsing Expression Grammars (PEGs) [1] are an alternative formalism for describing a language's syntax. Unlike CFGs, PEGs are unambiguous by construction, and their standard semantics is based on recognizing strings instead of generating

* Corresponding authors.
E-mail addresses: fabiom@dcc.ufrj.br (F. Mascarenhas), sergiomedeiros@ect.ufrn.br (S. Medeiros), roberto@inf.puc-rio.br (R. Ierusalimschy).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2014.01.012

them. A PEG can be considered both the specification of a language and the specification of a top-down parser for that language.

The idea of using a formalism for specifying parsers is not new; PEGs are based on two formalisms first proposed in the early seventies, Top-Down Parsing Language (TDPL) [2] and Generalized TDPL (GTDPL) [3]. PEGs have in common with TDPL and GTDPL the notion of *limited backtracking* top-down parsing: the parser, when faced with several alternatives, will try them in a deterministic order (left to right), discarding remaining alternatives after one of them succeeds. Compared with the older formalisms, PEGs introduce a more expressive syntax, based on the syntax of regexes, and add *syntactic predicates* [4], a form of unrestricted lookahead where the parser checks whether the rest of the input matches a parsing expression without consuming the input.

Ford [1] has already proven that PEGs can recognize any deterministic context-free language, but leaves open the question of the relation between context-free grammars and PEGs. In this paper, we argue that the similarities between CFGs and PEGs are deeper than usually thought, and how these similarities have been obscured by the way the two formalisms have been presented. PEGs, instead of a formalism completely unrelated to CFGs, can be seen as a natural outcome of removing the ambiguity of CFGs.

We start with a new semantics for CFGs, using the framework of natural semantics [5,6]. The new semantics borrows the syntax of PEGs and is also based on recognizing strings. We make the source of the ambiguity of CFGs, their non-deterministic alternatives for each non-terminal, explicit in the semantics of our new non-deterministic choice operator. We then remove the non-determinism, and consequently the ambiguity, by adding explicit failure and ordered choice to the semantics, so now we can only use the second alternative in a choice if the first one fails. By that point, we only need to add the *not* syntactic predicate to arrive at an alternative semantics for PEGs (modulo syntactic sugar such as the repetition operator and the *and* syntactic predicate). We prove that our new semantics for both CFGs and PEGs are equivalent to the usual ones.

Our semantics for CFGs and PEGs make it clear that the defining characteristic that sets PEGs apart from CFGs is the ordered choice. For example, the grammar $S \rightarrow (aba \mid a)b$ is both a CFG and a PEG in our notation, but consumes the prefix *ab* out of the subject *abac* when interpreted as a CFG, and fails as a PEG.

We also show in this paper how our new semantics for CFGs gives us a way to translate some subsets of CFGs to PEGs that parse the same language. The idea is that, as the sole distinction between CFG and PEG semantics is in the choice operator, we will have a PEG that is equivalent to the CFG whenever we can make the PEG choose the correct alternative at each choice, either through reordering or with the help of syntactic predicates. We show transformations from CFGs to PEGs for three unambiguous subsets of CFGs: LL(1), strong-LL(k), and LL-regular.

A straightforward correspondence between LL(1) grammars and PEGs was already noted [7], but never formally proven. The correspondence is that an LL(1) grammar describes the same language whether interpreted as a CFG or as a PEG. The intuition is that, if an LL(1) parser is able to choose an alternative with a single symbol of lookahead, then a PEG parser will fail for every alternative that is not the correct one. We prove that this intuition is correct if none of the alternatives in the CFG can generate the empty string, and also prove that a simple ordering of the alternatives suffices to hold the correspondence even if there are alternatives that can generate the empty string. In other words, any LL(1) grammar is already a PEG that parses the same language, modulo a reordering of the alternatives.

There is no such correspondence between strong-LL(k) grammars and PEGs, not even by imposing a specific order among the alternatives for each non-terminal. Nevertheless, we also prove that we can transform a strong-LL(k) grammar to a PEG, just by adding a predicate to each alternative of a non-terminal. We can either add a predicate to the beginning of each alternative, thus encoding the choice made by a strong-LL(k) parser in the grammar, or, more interestingly, add a predicate to the end of each alternative.

Our transformations lead to efficient parsers for LL(1) and strong-LL(k) grammars, even in PEG implementations that do not use memoization to guarantee O(n) performance, because the resulting PEGs only use backtracking to test the lookahead of each alternative, so their use of backtracking is equivalent to a top-down parser checking the next k symbols of lookahead against the lookahead values of each production.

There is no direct correspondence between LL-regular grammars and PEGs, either, given that strong-LL(k) grammars are a proper subset of LL-regular grammars. But we also show that we can transform any LL-regular grammar into a PEG that recognizes the same language: we first prove that right-linear grammars for languages with the prefix property, a property that is easy to achieve, have the same language whether interpreted as CFGs or as PEGs, then use this result to build lookahead expressions for the alternatives of each non-terminal based on which regular partition this alternative falls.

While LL(1) grammars are a proper subset of strong-LL(k) grammars, which are a proper subset of LL-regular grammars, thus making the LL-regular transformation work on grammars belonging to these simpler classes, the simpler classes have more straightforward transformations which merit a separate treatment.

Given that these classes of top-down CFGs can be automatically translated into equivalent PEGs, we can reuse classic top-down grammars in PEG-based tools. So grammars written for tools such as ANTLR [4] could be reused in a parser tool that has PEGs as its backend language. As PEGs are composable, these grammars can then be used as components in larger grammars. The language designer can then start with a simple, LL(1) or strong-LL(k) subset of the language, and then grow it into the full language.

The rest of this paper is organized as follows: Section 2 presents our new semantics for CFGs and PEGs, showing how to arrive at the latter from the former, and proves their correctness. Section 3 shows how an LL(1) grammar describes the

$$\textbf{Empty}\ \frac{}{G[\varepsilon]\ x \overset{\text{CFG}}{\rightsquigarrow} x}\ (\textbf{empty.1}) \qquad \textbf{Terminal}\ \frac{}{G[a]\ ax \overset{\text{CFG}}{\rightsquigarrow} x}\ (\textbf{char.1})$$

$$\textbf{Non-terminal}\ \frac{G[P(A)]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}{G[A]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}\ (\textbf{var.1})$$

$$\textbf{Concatenation}\ \frac{G[p_1]\ xyz \overset{\text{CFG}}{\rightsquigarrow} yz \qquad G[p_2]\ yz \overset{\text{CFG}}{\rightsquigarrow} z}{G[p_1\ p_2]\ xyz \overset{\text{CFG}}{\rightsquigarrow} z}\ (\textbf{con.1})$$

$$\textbf{Choice}\ \frac{G[p_1]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}\ (\textbf{choice.1}) \qquad \frac{G[p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{CFG}}{\rightsquigarrow} y}\ (\textbf{choice.2})$$

Fig. 1. Natural semantics of $\overset{\text{CFG}}{\rightsquigarrow}$.

same language when interpreted as a PEG, and proves this correspondence. Section 4 shows how a simple transformation generates a PEG from any strong-LL(k) grammar, keeping the same general structure and describing the same language as the original grammar, and proves the latter assertion. Section 5 shows the equivalence between some right-linear CFGs and PEGs, and how this can be used to build a simple transformation that generates a PEG from an LL-regular grammar. Finally, Section 6 reviews related work, and Section 7 summarizes the paper's contributions and gives our final remarks.

2. From CFGs to PEGs

This section presents a new definition of CFGs, based on natural semantics, and shows how from it we can establish a correspondence between CFGs and PEGs, which formalization is also given through natural semantics.

The section is structured as follows: Section 2.1 briefly reviews the traditional definition of CFGs and presents our new definition of CFGs, that is called PE-CFGs and uses parsing expressions, borrowed from PEGs, and natural semantics. Section 2.2 shows how we can obtain a PE-CFG from a CFG and proves that both definitions are equivalent. Next, Section 2.3 discusses the relationship between PE-CFGs and PEGs, and defines PEGs by adapting the formalization of PE-CFGs. Finally, Section 2.4 proves that our definition of PEGs is equivalent to Ford's definition.

2.1. From CFGs to PE-CFGs

The traditional definition of a CFG is as a tuple (V, T, P, S) of a finite set V of non-terminals symbols, a finite set T of terminal symbols, a finite relation P between non-terminals and strings of terminals and non-terminals, and an initial non-terminal S. We say that $A \to \beta$ is a *production* of G if and only if $(A, \beta) \in P$.

A grammar G defines a relation $\Rightarrow_G$ where $\alpha A \gamma \Rightarrow_G \alpha\beta\gamma$ if and only if $A \to \beta$ is a production of G. The language of G is the set of all strings of terminal symbols that relate to S by the reflexive–transitive closure of $\Rightarrow_G$. We can interpret the relation $\Rightarrow_G$ as a rewriting step, and then the language of G is the set of all strings of terminals that can be generated from S by a finite number of rewriting steps.

We want to give a new definition for CFGs that is closer to PEGs, so the similarities between the two formalisms will be more visible. Our new definition begins by borrowing the concept of a *parsing expression* from PEGs. The abstract syntax of parsing expressions is given below:

$$p = \varepsilon \mid a \mid A \mid p_1 p_2 \mid p_1 \mid p_2$$

Parsing expressions are defined inductively as the empty expression ε, a terminal symbol a, a non-terminal symbol A, a concatenation $p_1 p_2$ of two parsing expressions p_1 and p_2, or a *choice* $p_1 \mid p_2$ between two parsing expressions p_1 and p_2.

We now define a PE-CFG (short for *CFG using parsing expressions*) G as a tuple (V, T, P, p_S), where V and T are still the sets of non-terminals and terminals, but P is now a function from non-terminals to parsing expressions, and p_S is the initial parsing expression of the grammar. As P is a function, we will use the standard notation for function application, $P(A)$, to refer to the parsing expression associated with a non-terminal A in G.

Instead of the relation $\Rightarrow_G$, we define a new relation, $\overset{\text{CFG}}{\rightsquigarrow}$, among a grammar G, a string of terminal symbols v, and another string of terminal symbols w. We will use the notation $G\,v \overset{\text{CFG}}{\rightsquigarrow} w$ to say that $(G, v, w) \in \overset{\text{CFG}}{\rightsquigarrow}$. The intuition for the $\overset{\text{CFG}}{\rightsquigarrow}$ relation is that the first string is the *input*, and the second string is a suffix of the input that is left after G matches a prefix of this input. We will usually say $G\,xy \overset{\text{CFG}}{\rightsquigarrow} y$ to mean that G matches a prefix x of input string xy.

Fig. 1 shows our semantics for $\overset{\text{CFG}}{\rightsquigarrow}$ using natural semantics, as a set of inference rules. $G\,xy \overset{\text{CFG}}{\rightsquigarrow} y$ if and only if there is a finite proof tree for it, built using these rules. The notation $G[p'_S]$ denotes a new grammar (V, T, P, p'_S) that is equal to G except for the initial parsing expression p_S, which is replaced by p'_S. Each rule follows naturally from the intuition of $\overset{\text{CFG}}{\rightsquigarrow}$: an empty parsing expression does not consume any input (**empty.1**); a terminal consumes itself if it is the first symbol of the input (**char.1**); a non-terminal matches its corresponding production in P (**var.1**); a concatenation first matches p_1 and

then matches p_2 with what is left of the input (**con.1**); and a choice can match either p_1 or p_2 (**choice.1** and **choice.2**, respectively). The rules guarantee that if $G\,v \overset{\text{CFG}}{\rightsquigarrow} w$ then w is a suffix of v.

The language of G, $L(G)$, is now the set of prefixes that G matches, that is, all strings x where $G\,xy \overset{\text{CFG}}{\rightsquigarrow} y$ for some string y. In the traditional definition of CFGs, the language of a grammar is the set of strings the grammar generates; in our new definition, the language is the set of strings the grammar matches. We could have defined the language of G as the set of strings x where $G\,x \overset{\text{CFG}}{\rightsquigarrow} \varepsilon$, that is, the set of strings that G matches completely, but it is a corollary of the following lemma that the two definitions are equivalent:

Lemma 2.1. *Given a PE-CFG G, if $G[p]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$ then we have $\forall y'.G[p]\,xy' \overset{\text{CFG}}{\rightsquigarrow} y'$.*

Proof. By induction on the height of the proof tree for $G[p]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$. □

The previous lemma shows that the suffix in the relation $\overset{\text{CFG}}{\rightsquigarrow}$ is superfluous; we could have defined relation $\overset{\text{CFG}}{\rightsquigarrow}$ as a binary relation between a grammar G and an input w, meaning just *G recognizes w*. We chose to keep the suffix to emphasize the similarities between this semantics and our semantics for PEGs, where the suffix matters.

2.2. Correspondence between CFGs and PE-CFGs

We need a way to systematically transform a traditional CFG G to a corresponding PE-CFG G', and vice-versa. To simplify our proofs, we will assume that grammars do not have useless symbols. The main obstacle for these transformations is the type of P. P is a relation for CFGs, with different productions for the same non-terminal being different entries in this relation. In PE-CFGs, however, P is a function, with all the different productions encoded as choices in the parsing expression for the non-terminal.

The choice operator is commutative, associative, and idempotent; both left and right concatenations distribute over choice, that is, $p_1(p_2 \mid p_3) = p_1p_2 \mid p_1p_3$ and $(p_1 \mid p_2)p_3 = p_1p_3 \mid p_2p_3$.[1] So any parsing expression may be rewritten as a choice $p_1 \mid \cdots \mid p_n$, where the subexpressions $p_1, \ldots, p_n$ are distinct and do not have choice operators. We can then go from a PE-CFG G' to a CFG G using $A \to p_1, \ldots, A \to p_n$ as the productions of each non-terminal A, where $p_1, \ldots, p_n$ are the subexpressions obtained by rewriting the expression $P'(A)$ in the way above.

Going from a CFG G to a PE-CFG G' is easier: the right side of each production of G is a concatenation of non-terminals and terminals, which translates directly to a concatenation of parsing expressions (the concatenation of expressions is associative); we assign an arbitrary order to the productions of each non-terminal A of G, and then combine these productions right-associatively into a choice expression, and this is $P'(A)$. We will call the transformation of CFGs to PE-CFGs $\mathcal{T}$, so $\mathcal{T}(G) = G'$.

As an example, take the CFG G with the following set of productions:

$$P = \{A \to BC,\ B \to a,\ B \to b,\ C \to c,\ C \to d,\ C \to e\}$$

Its corresponding PE-CFG $\mathcal{T}(G) = G'$ has the following definition for the function P':

$$P'(A) = BC \qquad P'(B) = a \mid b \qquad P'(C) = c \mid d \mid e$$

We used the order that we listed the productions of G to order the choices, but commutativity and associativity of the choice operator guarantees that any other order would yield a grammar with the same language as G', so we could have used the following definition for P' instead:

$$P'(A) = BC \qquad P'(B) = a \mid b \qquad P'(C) = e \mid c \mid d$$

The proof that G and $\mathcal{T}(G)$ define the same language for any CFG G is a direct corollary of the following lemma:

Lemma 2.2. *Given a CFG G and its corresponding PE-CFG $\mathcal{T}(G) = G'$, we have $\alpha \overset{*}{\Rightarrow}_G x$ if and only if $G'[\alpha]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$, where x is a string of terminals and α is a string of terminals and non-terminals.*

Proof. ($\Rightarrow$) By induction on the number of steps in the derivation of x. The base case, where $\alpha = x$, is trivial, with an application of the **empty.1** rule or repeated applications of the **con.1** and **char.1** rules.

The induction step has α composed of three parts: a prefix α', a non-terminal A, and a suffix γ, with $\alpha' A\gamma \Rightarrow_G \alpha'\beta\gamma \overset{*}{\Rightarrow}_G x$. By the properties of $\overset{*}{\Rightarrow}_G$, x can be decomposed into x_1, x_2 and x_3 with $\alpha' \overset{*}{\Rightarrow}_G x_1$, $\beta \overset{*}{\Rightarrow}_G x_2$, and $\gamma \overset{*}{\Rightarrow}_G x_3$. By the induction hypothesis we have $G'[\alpha']\,x_1x_2x_3y \overset{\text{CFG}}{\rightsquigarrow} x_2x_3y$, $G'[\beta]\,x_2x_3y \overset{\text{CFG}}{\rightsquigarrow} x_3y$, and $G'[\gamma]\,x_3y \overset{\text{CFG}}{\rightsquigarrow} y$. We combine these

[1] The proof of these properties is straightforward from the semantics of both operators.

$$\textbf{Empty}\quad \frac{}{G[\varepsilon]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\ (\textbf{empty}.1) \qquad \textbf{Non-terminal}\quad \frac{G[P(A)]\ x \overset{\text{PEG}}{\rightsquigarrow} X}{G[A]\ x \overset{\text{PEG}}{\rightsquigarrow} X}\ (\textbf{var}.1)$$

$$\textbf{Terminal}\quad \frac{}{G[a]\ ax \overset{\text{PEG}}{\rightsquigarrow} x}\ (\textbf{char}.1) \qquad \frac{}{G[b]\ ax \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}},\ b \neq a\ (\textbf{char}.2)$$

$$\frac{}{G[a]\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{char}.3)$$

$$\textbf{Concatenation}\quad \frac{G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y \qquad G[p_2]\ y \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\ p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} X}\ (\textbf{con}.1)$$

$$\frac{G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p_1\ p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{con}.2)$$

$$\textbf{Ordered Choice}\quad \frac{G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}\ (\textbf{ord}.1)$$

$$\frac{G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail} \qquad G[p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{G[p_1 \mid p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}\ (\textbf{ord}.2)$$

$$\frac{G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail} \qquad G[p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p_1 \mid p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{ord}.3)$$

Fig. 2. Natural semantics of $\overset{\text{PEG}}{\rightsquigarrow}$.

proof trees in a proof tree for $G'[\alpha' A \gamma]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$ with rules **con.1**, **var.1**, and applications of the **choice** rules to select the alternative corresponding to production $A \to \beta$.

($\Leftarrow$) By induction on the height of the proof tree for $G'[\alpha]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$. The interesting case is **var.1**; we need to use the fact that the use of choice operators in G' follows a known structure, where each production is a right-associative choice of parsing expressions that do not have choice operators and correspond to the right side of productions in G. So the proof tree for $G'[P'(A)]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$ ends with a succession of **choice** rules that select which of the alternatives is taken for that non-terminal. We apply the induction hypothesis to the subtree above the last **choice** rule used, from the consequent to the antecedents. □

A corollary of Lemma 2.2 is that $S \overset{*}{\Rightarrow}_G x$ if and only if $\mathcal{T}(G)\,xy \overset{\text{CFG}}{\rightsquigarrow} y$, so the language of G and the language of $\mathcal{T}(G)$ are the same.

A traditional CFG is ambiguous if and only if there is some string with more than one leftmost (or rightmost) derivation. We can define ambiguity for PE-CFGs via proof trees: a PE-CFG G is ambiguous if and only if there is more than one proof tree for $G\,xy \overset{\text{CFG}}{\rightsquigarrow} y$ for some x and y.

We can show that a CFG G is ambiguous if and only if its corresponding PE-CFG $\mathcal{T}(G)$ is ambiguous. The proof is a corollary of the proposition that there is only one leftmost derivation for $\alpha \overset{*}{\Rightarrow}_G x$ if and only if there is only one proof tree for $G'[\alpha]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$, where x is a string of terminals and α is a string of non-terminals and terminals. This proposition has a straightforward proof by induction (on the number of steps in the derivation and on the height of the proof tree), and the corollary follows by denial of the consequent.

Ambiguity, in our semantics, is directly tied to the choice operator: if we try to prove that there cannot be more than one proof tree for a $G[p]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$, by induction on the height of the tree, our proof fails for case **choice.1**, because even if there is only one proof tree for the $G[p_1]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$, we might have $G[p_2]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$, so we can get another proof tree for $G[p_1 \mid p_2]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$ by using **choice.2**. The proof fails for case **choice.2** in a similar way.

If we can change the semantics of choice so that a single proof tree for its antecedents guarantees a single proof tree for the choice then we will guarantee that all grammars will be unambiguous. Obviously we will not have CFGs anymore; in particular, we will invalidate Lemma 2.2. In fact, our changes will take us from CFGs to a restricted form of PEGs, and we will prove that our changed semantics is equivalent to the semantics of PEGs as defined by Ford [1].

2.3. From PE-CFGs to PEGs

Now we will discuss in detail how we can obtain the semantics of PEGs by changing the semantics of $\overset{\text{CFG}}{\rightsquigarrow}$ presented in Fig. 1.

In order to define the semantics of PEGs, we will make the choice operator *ordered*: in a choice $p_1 \mid p_2$ we try p_2 only if p_1 does not match. But we need a way to have a proof tree for "p_1 does not match", so we will also introduce an explicit failure result, `fail`, to indicate the cases where a match is not possible. We will combine these changes in the semantics of a new relation $\overset{\text{PEG}}{\rightsquigarrow}$. Fig. 2 lists its inference rules, where X means either `fail` or the remainder of the input string in a successful match.

$$\textbf{Not Predicate}\quad \frac{G[p]\,x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[!p]\,x \overset{\text{PEG}}{\rightsquigarrow} x}\ (\textbf{not.1}) \qquad \frac{G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} y}{G[!p]\,xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{not.2})$$

Fig. 3. Natural semantics of the *not*-predicate.

Just introducing `fail` does not change the semantics enough to be incompatible with regular CFGs; if we take the semantics of Fig. 2 and replace rule **ord.2** with **choice.2** then we have a conservative extension of our PE-CFG semantics that introduces `fail`, so all our previous proofs remain valid. Ordered choice, represented by rule **ord.2**, is what changes the semantics so it is not representing CFGs anymore. A simple example that shows this change is the grammar G below:

$$S \to AB \qquad A \to aba \mid a \qquad B \to b$$

We have $G\,abac \overset{\text{CFG}}{\rightsquigarrow} ac$, but $G\,abac \overset{\text{PEG}}{\not\rightsquigarrow} ac$, as the only proof tree under $\overset{\text{PEG}}{\rightsquigarrow}$ for the input string $abac$ is for $G\,abac \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

We will use $L^{\text{PEG}}(G)$ for the language of a PE-CFG G interpreted with $\overset{\text{PEG}}{\rightsquigarrow}$; as with $\overset{\text{CFG}}{\rightsquigarrow}$, this is the set of strings x for which there is a string y with $G\,xy \overset{\text{PEG}}{\rightsquigarrow} y$. Informally, this set is still the set of all the prefixes that G matches, only using $\overset{\text{PEG}}{\rightsquigarrow}$ instead of $\overset{\text{CFG}}{\rightsquigarrow}$. But there is no equivalent of Lemma 2.1 for the $\overset{\text{PEG}}{\rightsquigarrow}$ relation; for example, the grammar above matches ab but fails for $abac$.

Properties of the operators also change under $\overset{\text{PEG}}{\rightsquigarrow}$: the choice operator is not commutative, and concatenation does not distribute over choice on the right anymore (although it still distributes on the left).

In Section 3, we will show a class of PE-CFGs where $L(G) = L^{\text{PEG}}(G)$. For now, an interesting result is the following lemma, which proves that $L^{\text{PEG}}(G)$ is a subset of $L(G)$ for any PE-CFG G:

Lemma 2.3. *Given a PE-CFG G, if $G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} y$ then we have $G[p]\,xy \overset{\text{CFG}}{\rightsquigarrow} y$.*

Proof. By induction on the height of the proof tree for $G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} y$. The only rule that does not have an identical rule in $\overset{\text{CFG}}{\rightsquigarrow}$ is **ord.2**, but it can trivially be replaced by **choice.2**. □

The intuition of $G\,x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ is that G does not match any prefix of x (including the empty string). This is a corollary of the following lemma, which formally says that the result of $G[p]\,x$ is unique under $\overset{\text{PEG}}{\rightsquigarrow}$, for any G, p and x:

Lemma 2.4. *Given a PE-CFG G, if $G[p]\,x \overset{\text{PEG}}{\rightsquigarrow} X$ and $G[p]\,x \overset{\text{PEG}}{\rightsquigarrow} X'$ then we have $X = X'$, and there is only one proof tree for $G[p]\,x \overset{\text{PEG}}{\rightsquigarrow} X$.*

Proof. By induction on the height of the proof tree for $G[p]\,x \overset{\text{PEG}}{\rightsquigarrow} X$. The interesting cases are **ord.1** and **ord.2**; for **ord.1**, the induction hypothesis rules out the possibility of $G[p_1]\,x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, so **ord.2** cannot apply even if we have $G[p_2]\,x \overset{\text{PEG}}{\rightsquigarrow} X$. For **ord.2**, we must have $G[p_1]\,x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ by the induction hypothesis; even if X is `fail` we cannot use rule **ord.1**. □

The PE-CFGs that we will be dealing with in the rest of the paper will have an important property that becomes possible to express by introducing failure: they will be *complete* grammars [1]. A complete PE-CFG is one where for any expression p and any input x either $G[p]\,x \overset{\text{PEG}}{\rightsquigarrow} x'$ or $G[p]\,x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Ford [1] proves that any grammar that does not have direct or indirect left recursion (a property which can be structurally checked) is complete.

A PE-CFG G is left-recursive when there is a non-terminal A of G and an input x where trying to derive a proof tree for $G[A]\,x$ can make $G[A]\,x$ appear again higher up in the tree. Because the semantics of $\overset{\text{PEG}}{\rightsquigarrow}$ is deterministic this means that a left-recursive PE-CFG may not have any proof tree for $G[A]\,x$ under $\overset{\text{PEG}}{\rightsquigarrow}$; in this case, an implementation of PEGs that tries to match x with the expression $G[A]$ will not terminate.

Once we have failure and unambiguity, it is natural to introduce a way to turn a failure into a success. This is the *not* syntactic predicate ($!p$ for any parsing expression p), which is a conservative extension of our semantics described in Fig. 3.

PE-CFGs extended with the not-predicate and interpreted using $\overset{\text{PEG}}{\rightsquigarrow}$ are equivalent, syntactically as well as semantically, to PEGs. Ford [1] also included the repetition operator in the abstract syntax of PEGs, but eliminating repetition is a simple matter of replacing each repetition expression p^* with a new non-terminal A_p with the production $A_p \to pA_p \mid \varepsilon$, which is just a step up from simple syntactic sugar.

2.4. Correspondence with Ford's definition

Ford [1] defines the semantics of PEGs using a relation $\Rightarrow_G$ that is similar to $\overset{\text{PEG}}{\rightsquigarrow}$. Unlike the relation $\Rightarrow_G$ for traditional CFGs, Ford's $\Rightarrow_G$ is not a single step in the match, but the whole match. The notation $(p, x) \Rightarrow_G (n, X)$, for $(p, x, n, X) \in \Rightarrow_G$,

means that either the parsing expression p matches the prefix x' of input x, if X is x', or the match fails, if X is `fail`. The number n is a step counter, used in proofs by induction involving $\Rightarrow_G$.

Ford's definition of relation $\Rightarrow_G$ is similar to our definition of the $\overset{\text{PEG}}{\rightsquigarrow}$ relation, using a similar set of cases. The following lemma states that both definitions are equivalent:

Lemma 2.5. *Given a PE-CFG G and a parsing expression p, $(p, xy) \Rightarrow_G (n, x)$ if and only if $G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} y$ and $(p, xy) \Rightarrow_G (n, \texttt{fail})$ if and only if $G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.*

Proof. The proof of the ($\Rightarrow$) direction is a straightforward induction on the step count n, while the proof ($\Leftarrow$) is a straightforward induction on the height of the proof tree for $G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} X'$. □

Our definition for the *language* of a PEG is different from Ford's, though. Ford defines L^{PEG} as the set of strings for which a PEG recognizes some prefix of the string, while we use the set of strings that the PEG recognizes. In particular, the language of ε is T^* by Ford's definition and ε with ours.

Ordered choice is what makes PEGs essentially different from CFGs, though, so one way to go from a CFG that can be parsed top-down to a PEG that parses the same language, without changing the structure of the grammar, is to make sure that the PEG always chooses the correct alternative at each choice. In Sections 3, 4, and 5 we show how this intuition leads to translations from three classes of CFGs for top-down parsing, LL(1), strong-LL(k), and LL-regular, to equivalent PEGs.

3. LL(1) grammars and PEGs

LL(1) grammars are the subset of CFGs where a top-down parser can decide which production to use for a non-terminal by examining just the next symbol of the input. An LL(1) parser can then parse the whole input by starting with the initial non-terminal of the grammar and then choosing which production to apply, making a choice again whenever it encounters a non-terminal, without needing to backtrack on its choices. A correspondence between them and PEGs has already been noted [7], but not formally proven, so they are a nice starting point for applying our new semantics of CFGs and PEGs to the task of finding translations from subsets of CFGs to corresponding PEGs.

We will divide this task in two parts: first we will consider LL(1) grammars without ε expressions and show that there is a correspondence between these grammars and PEGs. Then we will consider grammars with ε expressions and show that there is a correspondence between these grammars and PEGs if the ordering of the choice expressions respects a simple property.

In Section 2 we presented a method of translating a traditional CFG to a PE-CFG, a CFG using parsing expressions. That method generates PE-CFGs with a property that will be useful in the proofs for this section; because we are going to use this property in our proofs, we will formalize it with the following definition:

BNF structure A PE-CFG $G = (V, T, P, p_s)$ has *BNF structure* if it obeys the following properties:

1. No choice expression of G is part of a concatenation expression;
2. p_s is a single non-terminal;
3. For every choice $p_1 \mid p_2$ of G, if p_1 matches the empty string then p_2 must also match the empty string.

Any traditional CFG G has a corresponding PE-CFG G' that has BNF structure; in particular, it is trivial to ensure that $\mathcal{T}(G)$ always has BNF structure. Properties 1 and 2 of BNF structure are an obvious outcome of the transformation $\mathcal{T}$: the expression associated to each non-terminal is of the form $p_1 \mid \cdots \mid p_n$, where the choices associate to the right and $p_1, \ldots, p_n$ do not have choice expressions, so property 1 applies; the initial expression of G' is the initial non-terminal of G, so property 2 also applies; finally, property 3 can be guaranted by choosing an order for the productions of each non-terminal of G so the productions that can generate the empty string are last.

Any PE-CFG without BNF structure also can be rewritten to have it, by distributivity of concatenation over choice on the left and on the right, associativity and commutativity of choice, and the addition of an extra non-terminal to be the start expression. Throughout the rest of this section we will only consider PE-CFGs that have BNF structure in our definitions and proofs.

A traditional CFG without ε productions is LL(1) if and only if, for each of its non-terminals A_i, the *FIRST* sets for the right sides of the productions of A_i are disjoint. Before we can give a definition for LL(1) PE-CFGs, we need to define what is the *FIRST* set of a parsing expression. We will use the following definition for the *FIRST* set of an expression p with a PE-CFG G:

$$FIRST^G(p) = \left\{ a \in T \;\middle|\; G[p]\,axy \overset{\text{CFG}}{\rightsquigarrow} y \right\}$$

This definition of *FIRST* is equivalent to the definition for traditional CFGs for any parsing expression that does not have choice operators (that is, any parsing expression that has a corresponding string of terminals and non-terminals). We can

use the PE-CFG to CFG equivalence lemma (Lemma 2.2) to conclude that $p \overset{*}{\Rightarrow}_G ax$ from $\mathcal{T}(G)[p]axy \overset{\text{CFG}}{\rightsquigarrow} y$, where G is a traditional CFG. The *FIRST* set of p in the traditional definition is the set $\{a \in T \mid p \overset{*}{\Rightarrow}_G a\beta\}$. As we assumed in Section 2 that G does not have useless symbols, this is the same as the set $\{a \in T \mid p \overset{*}{\Rightarrow}_G ax\}$, and the two definitions of *FIRST* are equivalent.

We can now give a definition for LL(1) PE-CFGs without ε expressions: a PE-CFG G without ε expressions is LL(1) if and only if, for every choice $p_1 \mid p_2$ in the grammar, the *FIRST* sets of p_1 and p_2 are disjoint.

It is straightforward to prove that a traditional CFG G without ε productions is LL(1) if and only if its corresponding PE-CFG $\mathcal{T}(G)$ is also LL(1). The proof uses property 1 of BNF structure, associativity of choice, and the property that $FIRST^{\mathcal{T}(G)}(p_1 \mid p_2) = FIRST^{\mathcal{T}(G)}(p_1) \cup FIRST^{\mathcal{T}(G)}(p_2)$.

Now we have a definition for LL(1) PE-CFGs without ε expressions, we can show that these grammars can be interpreted as PEGs (by the relation $\overset{\text{PEG}}{\rightsquigarrow}$) without changing their language. The assertion that the language of an LL(1) grammar is the same whether interpreted as a CFG or as a PEG is a corollary of the following lemma:

Lemma 3.1. *Given an* LL(1) *PE-CFG G without ε expressions, $G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y$ if and only if $G[p]xy \overset{\text{PEG}}{\rightsquigarrow} y$.*

Proof. ($\Rightarrow$) By induction on the height of the proof tree for $G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y$. The interesting case is **choice.2**. For this case, we have $G[p_2]xy \overset{\text{CFG}}{\rightsquigarrow} y$. As G does not have ε expressions, x cannot be empty. Let a be the first symbol of x. It is obvious that $a \in FIRST^G(p_2)$, so $a \notin FIRST^G(p_1)$ by the LL(1) property. So $G[p_1]xy \overset{\text{CFG}}{\not\rightsquigarrow} w$, and, by denial of the consequent of Lemma 2.3, $G[p_1]xy \overset{\text{PEG}}{\not\rightsquigarrow} w$. LL(1) grammars cannot have left recursion [8], so they are complete and $G[p_1]xy \overset{\text{PEG}}{\not\rightsquigarrow} w$ implies $G[p_1]xy \overset{\text{PEG}}{\rightsquigarrow}$ `fail`. With the induction hypothesis and the application of **ord.2** we have $G[p_1 \mid p_2]xy \overset{\text{PEG}}{\rightsquigarrow} y$.

($\Leftarrow$) Just a special case of Lemma 2.3. □

We will now show that a correspondence between LL(1) grammars and their corresponding PEGs still exists when we allow ε expressions, as long as the LL(1) grammars have BNF structure. Grammars with ε expressions can have ε in the *FIRST* sets of their expressions, so we need a slightly different definition of *FIRST*:

$$FIRST^G(p) = \left\{a \in T \;\middle|\; G[p]axy \overset{\text{CFG}}{\rightsquigarrow} y\right\} \cup nullable(p)$$

$$nullable(p) = \begin{cases} \{\varepsilon\} & \text{if } G[p]x \overset{\text{CFG}}{\rightsquigarrow} x \\ \emptyset & \text{otherwise} \end{cases}$$

The LL(1) property for grammars with ε expressions also uses a *FOLLOW* set, defined below:

$$FOLLOW^G(A) = \left\{a \in T \cup \{\$\} \;\middle|\; G[A]\,yaz \overset{\text{CFG}}{\rightsquigarrow} az \text{ is in a proof tree for } G\; w\$ \overset{\text{CFG}}{\rightsquigarrow} \$\right\}$$

Like with the *FIRST* set, it is straightforward to prove that our definition of *FOLLOW* is equivalent to the definition for traditional CFGs. The restriction involving the proof tree for $G\,w\$ \overset{\text{CFG}}{\rightsquigarrow} \$$ of our definition proceeds directly from the fact that the traditional definition only uses derivations starting from the initial symbol of the grammar, and CFG derivations correspond to PE-CFGs proof trees.

The general statement $G\,xy \overset{\text{CFG}}{\rightsquigarrow} y \Rightarrow G\,xy \overset{\text{PEG}}{\rightsquigarrow} y$ that we proved true for LL(1) PE-CFGs without ε expressions is false for grammars with ε expressions, as the following simple grammar shows:

$$S \to a \mid \varepsilon$$

This grammar is LL(1), and we have $G\,a \overset{\text{CFG}}{\rightsquigarrow} a$ through rule **choice.2**, but $G\,a \overset{\text{PEG}}{\not\rightsquigarrow} a$, although simple inspection shows that the language of G is $\{a, \varepsilon\}$ whether interpreted as a CFG or as a PEG.

We solve the above problem by introducing an end-of-input marker \$ ($\$ \notin T$), and using this marker to constrain proof trees so we only consider trees that consume the input and leave just the marker. Instead of trying to prove that $G\,xy \overset{\text{CFG}}{\rightsquigarrow} y \Rightarrow G\,xy \overset{\text{PEG}}{\rightsquigarrow} y$, we will prove that $G\,x\$ \overset{\text{CFG}}{\rightsquigarrow} \$ \Rightarrow G\,x\$ \overset{\text{PEG}}{\rightsquigarrow} \$$, which will still be enough to prove that G has the same language either interpreted as a PE-CFG or as a PEG.

A PE-CFG G with ε expressions is LL(1) if and only if the following two restrictions hold for every production $A \to p$ of G and every choice $p_1 \mid p_2$ of p:

1. $FIRST^G(p_1) \cap FIRST^G(p_2) = \emptyset$;
2. $FIRST^G(p_1) \cap FOLLOW^G(A) = \emptyset$ if $\varepsilon \in FIRST^G(p_2)$.

This is a direct restatement of the LL(1) restrictions for traditional CFGs [9,3], and it is straightforward to show that a CFG G is LL(1) if and only if its corresponding PE-CFG G' is LL(1).

We can now show that an LL(1) PE-CFG G has the same language whether interpreted as a CFG or as a PEG. The proof is a corollary of the following lemma:

Lemma 3.2. *Given an* LL(1) *PE-CFG G, if there is a proof tree for $Gx\$ \overset{\text{CFG}}{\rightsquigarrow} \$$ then, for every subtree $G[p]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$, we have that $G[p]x'\$ \overset{\text{PEG}}{\rightsquigarrow} x''\$$.*

Proof. By induction on the height of the proof tree for $G[p]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$. The interesting case is **choice.2**. For this case, we have $G[p_2]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$. Because of BNF structure, this is a subtree of $G[A]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$ for some non-terminal A. By the definition of *FOLLOW*, the first symbol a of $x''\$$ is in $FOLLOW^G(A)$. We now have two subcases, one where $x' = x''$ and another where $x' = bwx''$.

In the first subcase, we have $a \notin FIRST^G(p_1)$ by the second LL(1) restriction. So $G[p_1]x''\$ \overset{\text{CFG}}{\not\rightsquigarrow} y$ and, by denial of the consequent of Lemma 2.3 and completeness of LL(1) grammars, $G[p_1]x''\$ \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. With the induction hypothesis and the application of **ord.2** we have $G[p_1 \mid p_2]x''\$ \overset{\text{PEG}}{\rightsquigarrow} x''\$$.

In the second subcase, where $x' = bwx''$, we have $b \in FIRST^G(p_2)$, so $b \notin FIRST^G(p_1)$ by the first LL(1) restriction. The rest of the proof is similar to the first subcase. □

The proof that $Gx\$ \overset{\text{CFG}}{\rightsquigarrow} \$$ if and only if $Gx\$ \overset{\text{PEG}}{\rightsquigarrow} \$$ for any LL(1) PE-CFG G is now trivial, from the above lemma and from Lemma 2.3.

In the next section we will show how any strong-LL(k) grammar can be translated to a PEG that recognizes the same language, while keeping the overall structure of the grammar.

4. Strong-LL(k) grammars and PEGs

Strong-LL(k) grammars are a subset of CFGs where a top-down parser can predict which production to use for a non-terminal just by examining the next k symbols of the input, where k is arbitrary but fixed for each grammar. They are a special case of LL(k) grammars, in which the parser can use both the next k symbols of the input and the history of which productions it already picked during parsing.

Unlike LL(1) grammars, there are strong-LL(k) grammars that have different languages when interpreted as CFGs and as PEGs, no matter how we order their choice expressions. For example, take the PE-CFG G with the following productions:

$$S \to A \mid B \qquad A \to ab \mid C \qquad B \to a \mid Cd \qquad C \to c$$

G is a strong-LL(2) grammar, and its language, when interpreted as a CFG, is $\{a, ab, c, cd\}$. But interpreting G as a PEG yields the language $\{a, ab, c\}$; when matching cd, non-terminal A succeeds (through its second alternative, non-terminal C), and non-terminal B (the second alternative of S) is never tried. Changing S to $S \to B \mid A$ changes the PEG's language to $\{a, c, cd\}$, which is still different from the language of G as a CFG, because what happened to cd now happens to ab.

Nevertheless, any strong-LL(k) language can be parsed by a top-down parser without backtracking while using k symbols of lookahead. So it seems intuitive that we can use syntactic predicates to direct a PEG parser to the right alternative. We cannot interpret G as a PEG and recognize the same language, but we can add predicates to G, to emulate the predictions that a strong-LL(k) parser makes.

An approach for translating a PE-CFG G to a PEG that recognizes the same language is to add an and-predicate (syntactical sugar for a double application of the not-predicate) in front of every alternative of a non-terminal; this and-predicate tests the next k symbols of the input against the possible lookahead values that a strong-LL(k) parser would use for that alternative. For the strong-LL(2) grammar above, the translation results in the following PEG:

$$S \to \&(ab \mid c\$)A \mid \&(a\$ \mid cd)B$$
$$A \to \&(ab)ab \mid \&(c\$)C$$
$$B \to \&(a\$)a \mid \&(cd)Cd$$
$$C \to \&(cd \mid c\$)c$$

It is easy to check that this PEG recognizes the language $\{a, ab, c, cd\}$, the same as G, if we include the marker \$ at the end of the input strings for the PEG. The formal definition of the translation does not add the predicate to the last alternative of a non-terminal (or to the sole alternative, in case of non-terminal C above).

Before formalizing our translation and proving its correctness, we will give definitions of strong-LL(k) properties using our new CFG formalism. First we need an auxiliary function $take_k$, with the definition below:

$$take_k(\varepsilon) = \varepsilon$$
$$take_k(a_1 \dots a_n) = \begin{cases} a_1 \dots a_k & \text{if } n > k \\ a_1 \dots a_n & \text{otherwise} \end{cases}$$

We will say that $take_k(x)$ is the *k-prefix* of x. We also need to define $\bullet_k$, a language concatenation operation that results in k-prefixes (i.e. concatenates each string of the first language with each string of the second language, taking the k-prefix of each result):

$$X \bullet_k Y = \left\{ take_k(x) \mid x \in X \cdot Y \right\}$$

A property of k-prefixes is that the k-prefix of the concatenation of two strings is also the k-prefix of the concatenation of their k-prefixes (proof by case analysis on the definition of $take_k$):

$$take_k(xy) = take_k\big(take_k(x) take_k(y)\big)$$

This leads directly to the following simple lemma, which we only include to reference in later proofs:

Lemma 4.1. *If $take_k(x) \in X$ and $take_k(y) \in Y$ then we have $take_k(xy) \in X \bullet_k Y$.*

Proof. Trivial. □

We can now define the $FIRST_k$ sets, the strong-LL(k) analog of the LL(1) *FIRST* sets. The $FIRST_k$ set of an expression p is the set of the k-prefixes of every string that p matches:

$$FIRST_k^G(p) = \left\{ take_k(x) \mid G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y \right\}$$

The definition of $FOLLOW_k$ sets is also a straightforward extension of the definition of *FOLLOW* sets for LL(1) grammars:

$$FOLLOW_k^G(A) = \left\{ take_k(y) \mid G[A]xy \overset{\text{CFG}}{\rightsquigarrow} y \text{ is in a proof tree for } G\ w\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k \right\}$$

To ensure that all members of $FOLLOW_k$ have length k, we use k end-of-input markers $\$ \notin T$ instead of the single marker we used with LL(1) grammars. The semantics of $\overset{\text{CFG}}{\rightsquigarrow}$ guarantee that $\$^k$ is a suffix of y, so the length of y is at least k and the length of $take_k(y)$ is always k.

We can now state the strong-LL(k) property: a PE-CFG G with BNF structure is strong-LL(k) if and only if every choice expression $p_1 \mid p_2$ of every production $A \to p$ satisfies the following condition:

$$\big(FIRST_k^G(p_1) \bullet_k FOLLOW_k^G(A)\big) \cap \big(FIRST_k^G(p_2) \bullet_k FOLLOW_k^G(A)\big) = \emptyset$$

The strong-LL(k) property is just a formal way of saying that the next k symbols of the input are enough to choose among the choice expressions of a given non-terminal.

We also need an auxiliary function *choice* that takes a set of strings and makes a choice expression with each string as an alternative of this choice:

$$choice(\emptyset) = \varepsilon$$
$$choice\big(\{p_1, \ldots, p_n\}\big) = p_1 \mid \cdots \mid p_n$$

We will use *choice* to transform a lookahead set into a lookahead expression. Our translation inserts lookahead expressions to direct the PEG parser to the correct alternative in a choice, so we only need to changes choice operations. Because we are assuming that our PE-CFGs have BNF structure, these choice operations are at the "top-level" of each production. Intuitively, if $p_1 \mid p_2$ is a choice of non-terminal A, $\varphi_k^G(p_1 \mid p_2, A)$ adds the lookahead expression $\mathcal{L}^G(p_1, A)$ to p_1 and recursively transforms p_2; any expression that is not a choice is not transformed:

$$\varphi_k^G(p_1 \mid p_2, A) = \mathcal{L}^G(p_1, A) p_1 \mid \varphi_k^G(p_2, A)$$
$$\varphi_k^G(\varepsilon, A) = \varepsilon$$
$$\varphi_k^G(a, A) = a$$
$$\varphi_k^G(p_1 p_2, A) = p_1 p_2$$
$$\varphi_k^G(B, A) = B$$
$$\text{where } \mathcal{L}^G(p, A) = choice\big(FIRST_k^G(p) \bullet_k FOLLOW_k^G(A)\big)$$

The definition of our translation now is straightforward. From a strong-LL(k) grammar G with BNF structure we can generate a PEG $\Phi_b(G)$ (the *before* LL(k)-PEG of G) by replacing each production $A \to p$ with $A \to \varphi_k^G(p, A)$.

To prove the correctness of the translation, we will use the same approach that we took in the proof for LL(1) grammars with ε expressions. We will prove that in any derivation of $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ all of the subparts of the derivation have correspondents in $\Phi_b(G)$ via function φ_k^G. One subtlety of the proof is the parameter A of φ_k^G; our definition of $\Phi_b(G)$ makes it

clear that A in $\varphi_k^G(p, A)$ is the non-terminal that "owns" the expression p. For an expression p that appears in a subpart of the derivation of $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ as $G[p]$, A is the first non-terminal that appears as $G[A]$ in a path from this subpart to the conclusion $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$. Formally, we can state the following lemma, a version of Lemma 3.2:

Lemma 4.2. *Given a strong-LL(k) PE-CFG G, if there is a proof tree for $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ then, for every subtree $G[p]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$ of this proof tree, we have $\Phi_b(G)[\varphi_k^G(p, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$, where A is the first non-terminal that appears as $G[A]$ in a path from the conclusion $G[p]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$ of the subtree to the conclusion $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ of the whole tree.*

Proof. By induction on the height of the proof tree for $G[p]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$. The interesting cases are **choice.1** and **choice.2**. For case **choice.1**, we have $G[p_1]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$. Because of BNF structure, this is a subtree of $G[A]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$, and we have $take_k(x''\$^k) \in FOLLOW_k^G(A)$ by the definition of $FOLLOW_k$. If we combine this with Lemma 4.1 we have $take_k(x'\$^k) \in FIRST_k^G(p_1) \bullet_k FOLLOW_k^G(A)$, because the k-prefix of what p_1 matches is in $FIRST_k^G(p_1)$. So $\Phi_b(G)[\mathcal{L}^G(p_1, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x'\$^k$ by the definition of $\mathcal{L}^G$. By the induction hypothesis, $\Phi_b(G)[p_1]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$, and with applications of rules **con.1** and **ord.1** we have $\Phi_b(G)[\varphi_k^G(p_1 \mid p_2, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$.

For case **choice.2**, we can use the LL(k) property and an argument similar to the one used in **choice.1** to conclude that $take_k(x'\$^k) \notin FIRST_k^G(p_1) \bullet_k FOLLOW_k^G(A)$. So, by the definition of $\mathcal{L}^G$, $\Phi_b(G)[\mathcal{L}^G(p_1, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. By the induction hypothesis, we have $\Phi_b(G)[\varphi_k^G(p_2, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$, and by rules **con.2** and **ord.2** we have $\Phi_b(G)[\varphi_k^G(p_1 \mid p_2, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$. □

We also need to prove that for any strong-LL(k) PE-CFG G we have $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ if $\Phi_b(G)\,x\$^k \overset{\text{PEG}}{\rightsquigarrow} \$^k$. Intuitively, if $(\&p_1)p_2$ matches a string x then p_2 also matches x, so if we have a proof tree for $\Phi_b(G)\,x\$^k \overset{\text{PEG}}{\rightsquigarrow} \$^k$ we will be able to erase all the predicates introduced by Φ_b and build a proof tree for $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$.

First, let us define predicate erasure as follows: the erasure of $(!p_1)p_2$ is the erasure of p_2. Any predicate occurring alone is replaced by ε. All other expressions just recursively erase predicates on their subparts. We get the erasure of a grammar by erasing the predicates in the right sides of every production, plus the initial symbol. The purpose of having a special case for the erasure of $(!p_1)p_2$ is to have the erasure of $\Phi_b(G)$ be G. Now we can prove the following lemma, which states that removing the predicates of a PEG G gives us a PE-CFG with a language that is a superset of the language of G.

Lemma 4.3. *Given a PEG G and an expression p, and the PE-CFG G' and expression p' obtained by erasing all predicates of G and p, if $G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} y$ then $G'[p']\,xy \overset{\text{CFG}}{\rightsquigarrow} y$.*

Proof. By induction on the height of the proof tree for $G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} y$. □

The proof that $\Phi_b(G)$ has the same language as G is now a corollary of Lemmas 4.2 and 4.3. This lemma will also be useful in the rest of this section and in the following one, to prove the correctness of our other transformations.

There is another approach for translating a strong-LL(k) PE-CFG G to an equivalent PEG. This approach uses a subtle consequence of the strong-LL(k) property: take the alternatives p_1 to p_n of a non-terminal A. Now let's say that two alternatives p_i and p_j both match prefixes of an input w, say x_i and x_j, with $x_i y_i = x_j y_j = w$; that is, $G[p_i]\,x_i y_i \overset{\text{CFG}}{\rightsquigarrow} y_i$ and $G[p_j]\,x_j y_j \overset{\text{CFG}}{\rightsquigarrow} y_j$. By the definition of $FIRST_k$, we have $take_k(x_i) \in FIRST_k^G(p_i)$ and $take_k(x_j) \in FIRST_k^G(p_j)$. Therefore we cannot have both $take_k(y_i) \in FOLLOW_k^G(A)$ and $take_k(y_j) \in FOLLOW_k^G(A)$, or we would violate the strong-LL(k) property by having $take_k(w)$ in both $FIRST_k^G(p_i) \bullet_k FOLLOW_k^G(A)$ and $FIRST_k^G(p_j) \bullet_k FOLLOW_k^G(A)$ (Lemma 4.1).

The fact that we cannot have the first k symbols of both y_i and y_j in $FOLLOW_k^G(A)$ is the core of this other approach, which is to add a guard *after* each alternative of a non-terminal A to test if the next k symbols of the input are in $FOLLOW_k^G(A)$. PEG's local backtracking then guarantees that the wrong alternative will not be taken even if it matches a prefix of the input.

For the strong-LL(2) grammar we used as an example in the beginning of this section, this approach yields the following translated PEG:

$$S \to A\,\&(\$\$) \mid B\,\&(\$\$)$$
$$A \to ab\,\&(\$\$) \mid C\,\&(\$\$)$$
$$B \to a\,\&(\$\$) \mid Cd\,\&(\$\$)$$
$$C \to c\,\&(\$\$ \mid d\$)$$

It is easy to check that this PEG recognizes the correct language $\{a, ab, c, cd\}$ if we include the marker \$\$ at the end of the input string.

Like with our first translation, our second translation uses a function ϕ_k^G that translates the choice expressions in the production of a non-terminal A, adding an and-predicate built from a choice of every string in $FOLLOW_k^G(A)$ to the first half of the choice and recursively translating the second half. As with φ_k^G, ϕ_k^G is the identity function for other kinds of expressions, as the translation only changes choice expressions and we assume BNF structure:

$$\begin{aligned}
\phi_k^G(p_1 \mid p_2, A) &= p_1 \,\&choice\big(FOLLOW_k^G(A)\big) \mid \phi_k^G(p_2, A)\\
\phi_k^G(\varepsilon, A) &= \varepsilon\\
\phi_k^G(a, A) &= a\\
\phi_k^G(p_1 p_2, A) &= p_1 p_2\\
\phi_k^G(B, A) &= B
\end{aligned}$$

The definition of the second translation is now straightforward. From a strong-LL(k) grammar G with BNF structure we can generate a PEG $\Phi_a(G)$ (the *after* LL(k)-PEG of G) by replacing each production $A \to p$ with $A \to \phi_k^G(p, A)$.

The following lemma is like Lemma 4.2 in that it proves that all subparts of a derivation for $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ have correspondents in $\Phi_a(G)$ via function ϕ_k^G. As with Lemmas 4.2 and 3.2, we need to restrict ourselves to matches that consume all input but the end-of-input marker $\$^k$ so we can use the $FOLLOW_k$ sets of the non-terminals in our proof, and by extension the LL(k) properties of G.

Lemma 4.4. *Given a strong-*LL(k) *PE-CFG G, if there is a proof tree for $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ then, for every subtree $G[p]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$ of this proof tree, we have $\Phi_a(G)[\phi_k^G(p, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$, where A is the first non-terminal that appears as $G[A]$ in a path from the conclusion $G[p]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$ of the subtree to the conclusion $G\,x\$^k \overset{\text{CFG}}{\rightsquigarrow} \$^k$ of the whole tree.*

Proof. By induction on the height of the proof tree for $G[p]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$. The interesting cases are **choice.1** and **choice.2**. For case **choice.1**, we have $G[p_1]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$. Because of BNF structure, this is a subtree of $G[A]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} x''\$^k$, and $take_k(x''\$^k) \in FOLLOW_k^G$ by the definition of $FOLLOW_k$. It is then easy to see that $\Phi_a(G)[\&choice(FOLLOW_k^G(A))]\,x''\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$. By the induction hypothesis, we have $\Phi_a(G)[p_1]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$, and with applications of **con.1** and **ord.1** we have $\Phi_a(G)[\phi_k^G(p_1 \mid p_2, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$.

For case **choice.2**, if there is no w so $\Phi_a(G)[p_1]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} w$ then we can conclude $\Phi_a(G)[p_1]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ by completeness of LL(k) grammars, and we can apply the induction hypothesis on p_2 and rules **con.2** and **ord.2** to get $\Phi_a(G)[\phi_k^G(p_1 \mid p_2, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$. Now suppose we have $\Phi_a(G)[p_1]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} w$. Expression p_1 is predicate-free, so we have $G[p_1]\,x'\$^k \overset{\text{CFG}}{\rightsquigarrow} w$ by Lemma 4.3. We have $take_k(x''\$^k) \in FOLLOW_k^G(A)$ by the definition of $FOLLOW_k$, and we have already seen that means $take_k(w) \notin FOLLOW_k^G(A)$ or the LL(k) property is violated. So we have $\Phi_a(G)[\&choice(FOLLOW_k^G(A))]\,w \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, and we can apply the induction hypothesis on p_2 and rules **con.1** and **ord.2** to conclude $\Phi_a(G)[\phi_k^G(p_1 \mid p_2, A)]\,x'\$^k \overset{\text{PEG}}{\rightsquigarrow} x''\$^k$. □

The proof that $\Phi_a(G)$ has the same language as G is a corollary of Lemmas 4.4 and 4.3.

LL(1) grammars are a special case of LL(k) grammars where $k = 1$, and every LL(1) grammar is also a strong-LL(1) grammar (and vice-versa) [3,8], so the two transformations we presented can also be used for LL(1) grammars, although the lookahead expressions become redundant.

5. Right-linear and LL-regular grammars

A *right-linear CFG* is one where the right side of every production has at most one non-terminal, and this non-terminal can only appear as the last symbol of the production. Right-linear CFGs can only define regular languages, and any regular language R has a right-linear CFG G; it is straightforward to encode any NFA as a right-linear CFG, and vice-versa [10].

For PE-CFGs, we will define a *right-linear PE-CFG* as a CFG where the right side of every production is a *right-linear parsing expression*. We define right-linear parsing expressions as the following predicate on parsing expressions: ε, a and A are right-linear; $p_1 p_2$ is right-linear if and only if p_1 is a terminal and p_2 is right-linear; $p_1 \mid p_2$ is right-linear if and only if p_1 and p_2 are right-linear. It is easy to see that any right-linear CFG has a corresponding right-linear PE-CFG, and vice-versa, as the transformations between CFGs and PE-CFGs we have on Section 2 preserve right-linearity.

Right-linear PE-CFGs, in the general case, do not recognize the same language when interpreted by $\overset{\text{CFG}}{\rightsquigarrow}$ and $\overset{\text{PEG}}{\rightsquigarrow}$. An obvious example is the grammar $S \to a \mid aa$, which is right-linear and has the language $\{a, aa\}$ under $\overset{\text{CFG}}{\rightsquigarrow}$ but $\{a\}$ under $\overset{\text{PEG}}{\rightsquigarrow}$.

But the simplicity of right-linear grammars lets us prove an equivalence between right-linear CFGs and PEGs by adding a single restriction: the grammar's language must have the *prefix property*, that is, there are no distinct strings x and y in the language such that x is a prefix of y.

We could prove this equivalence directly, but it is more interesting to combine a few more general lemmas that individually deal with the relation of the prefix property and PEGs, and of right-linear grammars and the prefix property. The first lemma gives an equivalence between a class of PE-CFGs that have a stronger form of the prefix property and PEGs.

Lemma 5.1. *Given a PE-CFG G and a parsing expression p where, for any choice $q \mid r$ in $G[p]$, $L(G[q \mid r])$ has the prefix property, if $G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y$ then $G[p]xy \overset{\text{PEG}}{\rightsquigarrow} y$.*

Proof. By induction on the height of the proof tree for $G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y$. The interesting case if **choice.2**. We have $p = p_1 \mid p_2$, so $L(G[p]) = L(G[p_1]) \cup L(G[p_2])$, and both $L(G[p_1])$ and $L(G[p_2])$ have the prefix property. This means that either $G[p_1]xy \overset{\text{CFG}}{\rightsquigarrow} y$, so we can use the induction hypothesis and rule **ord.1** to get $G[p]xy \overset{\text{PEG}}{\rightsquigarrow} y$, or there is no suffix z of xy with $G[p_1]xy \overset{\text{CFG}}{\rightsquigarrow} z$, as that would violate the prefix property of $L(G[p_1])$. In this case, we have $G[p_1]xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ by modus tollens of Lemma 2.3, and we can now use the induction hypothesis with p_2, and rule **ord.2**, to get $G[p]xy \overset{\text{PEG}}{\rightsquigarrow} y$. □

The second lemma distributes the prefix property to the components of a right-linear parsing expression, when the language of the expression has the prefix property:

Lemma 5.2. *Given a right-linear PE-CFG G and a right-linear parsing expression p, if $L(G[p])$ has the prefix property then, for any parsing expression q in p, $L(G[q])$ has the prefix property.*

Proof. By structural induction on p. The interesting case is $p = p_1\, p_2$. As p is right-linear, p_1 is a terminal and p_2 is right-linear, so $L(G[p_1])$ trivially has the prefix property. As $L(G[p]) = L(G[p_1]) \cdot L(G[p_2])$, $L(G[p_2])$ must have the prefix property, and we can use the induction hypothesis to conclude that any parsing expression $L(G[q])$ will also have the prefix property for any parsing expression q in p_2. □

The third lemma distributes the prefix property to all of the components of a right-linear grammar, when the language of the starting expression has the prefix property:

Lemma 5.3. *Given a right-linear PE-CFG G and a right-linear parsing expression p, if $L(G[p])$ has the prefix property then, for any parsing expression q in $G[p]$, $L(G[q])$ has the prefix property.*

Proof. By induction on the number of steps necessary to reach q from p: zero steps if $p = q$, one step if q is a part of p, $k+1$ steps if q is a part of $P(A)$ where A is reachable from p in k steps. □

The final lemma uses the first and the third lemma, plus Lemma 2.3, to prove the equivalence between right-linear CFGs with the prefix property and PEGs:

Lemma 5.4. *Given a right-linear PE-CFG G and a right-linear parsing expression p, if $L(G[p])$ has the prefix property then $G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y$ if and only if $G[p]xy \overset{\text{PEG}}{\rightsquigarrow} y$.*

Proof. By Lemma 5.3, the choice expression in $G[p]$ also has the prefix property, so, by Lemma 5.1, $G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y$ implies $G[p]xy \overset{\text{PEG}}{\rightsquigarrow} y$. By Lemma 2.3, $G[p]xy \overset{\text{PEG}}{\rightsquigarrow} y$ implies $G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y$. □

The prefix-property restriction may seem overly restrictive, but we can obtain a right-linear grammar with the prefix property from any right-linear grammar G by applying the following transformation, where we use $\$ \notin T$ as an end-of-input marker, to the right side of G's productions and to its initial parsing expression:

$$
\begin{aligned}
\Pi(\varepsilon) &= \$ \\
\Pi(a) &= a\$ \\
\Pi(A) &= A \\
\Pi(p_1\, p_2) &= p_1 \Pi(p_2) \\
\Pi(p_1 \mid p_2) &= \Pi(p_1) \mid \Pi(p_2)
\end{aligned}
$$

A (strong) LL-regular CFG [11,12] is a generalization of LL(k) CFGs where a predictive top-down parser may decide which alternative to take based on where the rest of the input falls on a set of regular partitions of T^*. Formally, a CFG G is LL-regular if there is a regular partition π of T^* such that for any two leftmost derivations of the following form, if $x \equiv y \pmod{\pi}$ then $\gamma = \delta$:

$$S \overset{*}{\Rightarrow}_G w_1 A \alpha_1 \Rightarrow_G w_1 \gamma \alpha_1 \overset{*}{\Rightarrow}_G w_1 x$$
$$S \overset{*}{\Rightarrow}_G w_2 A \alpha_2 \Rightarrow_G w_1 \delta \alpha_2 \overset{*}{\Rightarrow}_G w_2 y$$

Restating the definition for PE-CFGs is straightforward. First we introduce the $BLOCK^G_\pi$ set that tells in which blocks of partition π the input for p falls:

$$BLOCK^G_\pi(p, A) = \left\{ B_k \in \pi \;\middle|\; G[p]xy \overset{\text{CFG}}{\rightsquigarrow} y \text{ and } G[A]xy \overset{\text{CFG}}{\rightsquigarrow} y \text{ are in a proof tree for } G\, w\$ \overset{\text{CFG}}{\rightsquigarrow} \$ \text{ and } xy \in B_k \right\}$$

A PE-CFG G with BNF structure is LL-regular if and only if there is a partition π of $T^* \cdot \{\$\}$ such that every choice expression $p_1 \mid p_2$ of every production $A \to p$ has $BLOCK^G_\pi(p_1, A) \cap BLOCK^G_\pi(p_2, A) = \emptyset$.

The original definition of LL-regular grammars uses the partition where the rest of the input falls to predict alternatives, so our addition of an end-of-input marker \$ is not changing the class of grammars we are defining, while ensuring that the blocks of the regular partition have the prefix property.

Any strong-LL(k) grammar is also LL-regular [13], so a simple reordering of alternatives is not sufficient for obtaining a PEG that recognizes the same language as an LL-regular grammar G. But we can use the same approach we used in the translation Φ_b to translate an LL-regular grammar G into a PEG $\mathcal{R}(G)$ that recognizes the same language, using an and-predicate to add a lookahead expression to the front of the alternatives of each choice.

We assume that each block B_k of the regular partition π has a corresponding right-linear grammar G_{B_k}, where the intersection of the non-terminal sets of G and of all these grammars is empty; the non-terminal set of $\mathcal{R}(G)$ is the union of these sets.

We form the regular lookahead of an alternative p, $\mathcal{L}^G_r(p, A)$, by making a choice of the grammars for each block in $BLOCK^G_\pi(p, A)$, and wrapping this choice in an and-predicate:

$$\mathcal{L}^G_r(p, A) = \&choice\left(\left\{ S_{B_k} \;\middle|\; B_k \in BLOCK^G_\pi(p, A) \right\}\right)$$

where *choice* is the function we used to build the lookahead expressions for φ_b and ϕ_b.

Function $\rho^G(p, A)$ adds lookahead expressions where necessary, assuming that the original grammar has BNF structure:

$$\begin{aligned}
\rho^G(p_1 \mid p_2, A) &= \mathcal{L}^G_r(p_1, A) p_1 \mid \rho^G(p_2, A) \\
\rho^G(\varepsilon, A) &= \varepsilon \\
\rho^G(a, A) &= a \\
\rho^G(p_1 p_2, A) &= p_1 p_2 \\
\rho^G(B, A) &= B
\end{aligned}$$

We obtain the productions of $\mathcal{R}(G)$ by applying ρ^G to the right-side of each production of G, and then adding the productions for each G_{B_k}. We can prove a lemma similar to Lemma 4.2:

Lemma 5.5. *Given an LL-regular PE-CFG G, if there is a proof tree for $G\, x\$ \overset{\text{CFG}}{\rightsquigarrow} \$$ then, for every subtree $G[p]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$, we have $\mathcal{R}(G)[\rho^G(p, A)]x'\$ \overset{\text{PEG}}{\rightsquigarrow} x''\$$, where A is the first non-terminal that appears as $G[A]$ in a path from the conclusion $G[p]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$ of the subtree to the conclusion $G\, x\$ \overset{\text{CFG}}{\rightsquigarrow} \$$ of the whole tree.*

Proof. By induction on the height of the proof tree for $G[p]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$. The interesting cases are **choice.1** and **choice.2**. For **choice.1**, we have $G[p_1]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$. Because of the BNF structure, this is a subtree of $G[A]x'\$ \overset{\text{CFG}}{\rightsquigarrow} x''\$$, and we have $x'\$$ in some block $B_k \in BLOCK^G_\pi(p_1, A)$. So $x'\$ \in L(G_{B_k})$. It is easy to see that $x'\$ \in L(choice(\{S_{B_k} \mid B_k \in BLOCK^G_\pi(p, A)\}))$; this grammar is right-linear, so by Lemma 5.4 and the semantics of the and-predicate we have $\mathcal{R}(G)[\mathcal{L}^G_r(p_1, A)]x'\$ \overset{\text{PEG}}{\rightsquigarrow} x'\$$. We have $\mathcal{R}(G)[p_1]x'\$ \overset{\text{PEG}}{\rightsquigarrow} x''\$$ by the induction hypothesis, and can use rules **con.1** and **ord.1** to get $\mathcal{R}(G)[\rho^G(p_1 \mid p_2, A)]x'\$ \overset{\text{PEG}}{\rightsquigarrow} x''\$$.

For case **choice.2**, we can use the LL-regular property to conclude that the block B_k with $x'\$ \in B_k$ is not in $BLOCK^G_\pi(p_1, A)$, and then use the definition of $\mathcal{L}^G_r$ and an argument similar to the one used in **choice.1** to get $\mathcal{R}(G)[\mathcal{L}^G_r(p_1, A)]x'\$ \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. We can use the induction hypothesis to get $\mathcal{R}(G)[\rho^G(p_2, A)]x'\$ \overset{\text{PEG}}{\rightsquigarrow} x''\$$, and then use rules **con.2** and **ord.2** to get $\mathcal{R}(G)[\rho^G(p_1 \mid p_2, A)]x'\$ \overset{\text{PEG}}{\rightsquigarrow} x''\$$. □

The proof that $\mathcal{R}(G)$ has the same language as G is a corollary of Lemmas 5.5 and 4.3. We can use Lemma 4.3 even though $\mathcal{R}(G)$ has non-terminals that are not present in G; these extra non-terminals are only referenced inside predicates, so they become useless when the predicates are removed, and can also be removed.

6. Related work

Parsing Expression Grammars have generated much academic interest since their introduction by Ford [1], with over sixty citations of Ford's paper in ACM's Digital Library. But just a few of these works are concerned with the theory of PEGs and their relation to other parsing tools; this section discusses these works and how they relate to our work.

Ford [1] leaves open the problem of how PEGs and CFGs relate, and does not outline a strategy to solve this problem. The solution of this problem for the major classes of top-down CFGs is a contribution of our work, and our recasting of the CFGs in a recognition-based formalism is another contribution that shows where CFGs and PEGs diverge.

This paper is an extension of previous, unpublished work done in [14]. This older work has the first version of the PE-CFG semantics, as well a the transformations between strong-LL(k) grammars and PEGs, and the proof of the correspondence between LL(1) grammars and PEGs. The proofs have been revised and improved.

Redziejowski [15] adapts the *FIRST* and *FOLLOW* relations of CFGs to the study of Parsing Expression Grammars, defining PEG analogs of these two relations. Redziejowski then uses the analogs for a conservative approximation of when an ordered choice is commutative or not, by defining a PEG analog of the LL(1) restriction. He admits that the approximation is too conservative for practical use, specially because of its treatment of syntactic predicates. We do not attempt to give definitions of *FIRST* and *FOLLOW* for PEGs, limiting our redefinitions of these relations just to our PE-CFG formalism, and using them to prove correspondences between LL(1) and strong-LL(k) CFGs and structurally similar PEGs, something that Redziejowski does not explore.

An earlier work by Redziejowski [16] presents several identities regarding the languages defined by PEGs, although the author concludes that the identities are only useful for obtaining approximations of a PEGs language, and he also does not try to relate the languages of PEGs and of CFGs.

In a more recent work [17], Redziejowski, based on our earlier, unpublished work [14], extends our correspondence between LL(1) grammars and PEGs to a larger class of grammars, and calls these grammars LL(1p). He notes that checking whether a grammar is LL(1p) is harder than checking if the grammar is LL(1). The paper also uses earlier versions of the new semantics for CFGs and PEGs that we presented in Section 2.

Schmitz [18] presents both an ambiguity detection algorithm for CFGs in the context of the SDF2 formalism, an extension of CFGs, and an algorithm for detecting whether an ordered choice in a PEG is commutative or not. Schmitz notes that an overly strict ambiguity detector can be as restrictive as introducing ordering, but does not attempt to further study the relation between CFGs and PEGs.

Parr and Quong [19] add semantic and syntactic predicates to LL(k) grammars to get the *pred*-LL(k) parsing strategy. The syntactic predicates of pred-LL(k) are only used in productions that have LL(k) conflicts. In these productions, the parser tries to match the predicates of each alternative in the order they are given in the grammar definition, choosing the first alternative with a predicate that succeeds. Backtracking is strictly local, as with PEGs, so a subsequent failure does not make the parser try other productions. The paper does not give a formal specification of pred-LL(k) grammars, nor how they relate to the class of LL(k) grammars.

Parr and Fisher [20] introduce the LL($*$) parsing strategy, which uses the basic idea of LL-regular grammars, with a predictive top-down parser for these grammars that uses a deterministic finite automata to select which alternative of a non-terminal to take. If the grammar is not LL-regular, the LL($*$) parser can use semantic and syntactic predicates with local backtracking, as in pred-LL(k), and also automatically introduce predicates, via a suitably named "PEG mode".

Generalized LL parsing [21,22] extends the idea of LL(1) recursive descent parsing to the full class of context-free grammars (including ambiguous grammars), by making the parser proceed among the different conflicting alternatives "in parallel". It replaces the call stack of a recursive descent parser with a *graph structured stack* (GSS), a data structure adapted from Generalized LR parsers [23]. As the resulting parsers can parse ambiguous grammars, construction of the parse tree is non-trivial to implement efficiently [24,21], using a *shared packed parse forest* data structure also adapted from generalized bottom-up techniques [25].

Even if we restrict our domain to unambiguous grammars, we cannot use GLL parsing as a basis for a correspondence between CFGs and PEGs; a GLL parser is not a predictive parser, so we cannot try to encode the predictive part of the parser in a syntactic predicate, as we did for LL-regular grammars, nor we can exploit a property of the grammars, as we did for LL(1) and strong-LL(k) grammars.

7. Conclusions

We presented a new formalism for context-free grammars that is based on recognizing (parts of) strings instead of generating them. We adopted a subset of the syntax of parsing expression grammars, and the notion of letting a grammar recognize just part of an input string, to purposefully get a definition for CFGs that is closer to PEGs, yet defines the same class of languages as traditional CFGs. These PE-CFGs define the same class of language as traditional CFGs, and simple transformations let us get a PE-CFG from a CFG and vice-versa.

Our semantics for PE-CFGs has a non-deterministic choice operation. We showed how a deterministic choice operation based on the notions of failure and ordering turns PE-CFGs into quasi-PEGs; the addition of a *not* syntactic predicate then gave us a semantics that is equivalent to Ford's original semantics for PEGs. We then used our new formulations of CFGs and PEGs to study correspondences between four classes of CFGs and PEGs: LL(1), strong-LL(k), right-linear and LL-regular. We proved that LL(1) grammars already define the same language either interpreted as CFGs or as PEGs, as was already suspected, and gave transformations that yield equivalent PEGs for the grammars in the other three classes.

All our transformations preserve the structure of the original grammars; we do not change or remove non-terminals, nor change the alternatives of a non-terminal, just add predicates to the beginning or the end of each alternative. This means that our transformations have a practical application in reusing grammars made for one formalism with tools made for the other, as conserving the structure of the grammar makes it easier to carry semantic actions from one tool to another without modification.

Our transformations assume the presence of some kind of end-of-input marker, and incorporate this marker in the resulting PEG. This does not affect our equivalence results, but it does have implications in composability of PEGs resulting from our transformations. To use a PEG obtained from one of our transformations as part of a larger PEG (for embedding one language in another, for example) requires a suitable "end-of-input marker" to be picked (the boundaries between languages have to be explicit). We believe this problem should be easily solvable in practice.

A possible next step of this work would be the study of relationship between PEGs and classic bottom-up CFGs, such as LR(k) [26] and Simple LR(k) [27]. A key issue regarding this study is the fact that bottom-up grammars can have left-recursive rules, but a PEG with left-recursive rules is not complete [1].

Recently, it was suggested a conservative extension of PEGs' semantics that gives meaning for left-recursive rules, so PEGs with these rules would also be complete [28]. Based on this PEGs' extension we could try to establish a correspondence between bottom-up grammars and PEGs, and see if it is possible to achieve an equivalent PEG, with a similar structure, from a bottom-up CFG.

References

[1] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: Proceedings of the 31st ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, POPL '04, ACM, New York, NY, USA, 2004, pp. 111–122.
[2] A. Birman, J.D. Ullman, Parsing algorithms with backtrack, Inf. Control 23 (1) (1973) 1–34.
[3] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation, and Compiling, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.
[4] T.J. Parr, R.W. Quong, Antlr: a predicated-LL(k) parser generator, Softw. Pract. Exp. 25 (7) (1995) 789–810, http://dx.doi.org/10.1002/spe.4380250705.
[5] G. Kahn, Natural semantics, in: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, STACS '87, Springer-Verlag, London, UK, 1987, pp. 22–39.
[6] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, Foundations of Computing, MIT Press, 1993.
[7] C.F. Clark, Message to comp.compilers reference 05-08-115, http://compilers.iecc.com/comparch/article/05-09-009, 2005.
[8] D. Grune, C.J. Jacobs, Parsing Techniques – A Practical Guide, Ellis Horwood, 1991.
[9] D.E. Knuth, Top-down syntax analysis, Acta Inform. 1 (2) (1971) 79–110, http://dx.doi.org/10.1007/BF00289517.
[10] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, 1st edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979.
[11] A. Nijholt, LL-regular grammars, Int. J. Comput. Math. 8 (1980) 303–318.
[12] S. Jarzabek, T. Krawczyk, LL-regular grammars, Inf. Process. Lett. 4 (2) (1975) 31–37.
[13] A. Nijholt, From LL-regular to LL(1) grammars: Transformations, covers and parsing, RAIRO Theor. Inform. Appl. 16 (1982) 387–406.
[14] S.Q. de Medeiros, Correspondência entre PEGs e classes de gramáticas livres de contexto, PhD thesis, PUC-Rio, 2010.
[15] R.R. Redziejowski, Applying classical concepts to parsing expression grammar, Fundam. Inform. 93 (2009) 325–336.
[16] R.R. Redziejowski, Some aspects of parsing expression grammar, Fundam. Inform. 85 (2008) 441–451.
[17] R.R. Redziejowski, From EBNF to PEG, Fundam. Inform. 128 (2013) 177–191.
[18] S. Schmitz, Modular syntax demands verification, Tech. rep. I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice-Sophia Antipolis, France, Oct. 2006.
[19] T.J. Parr, R.W. Quong, Adding semantic and syntactic predicates to LL(k): pred-LL(k), in: Proceedings of the 5th International Conference on Compiler Construction, CC '94, Springer-Verlag, London, UK, 1994, pp. 263–277.
[20] T. Parr, K. Fisher, LL(*): the foundation of the ANTLR parser generator, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, New York, NY, USA, 2011, pp. 425–436.
[21] E. Scott, A. Johnstone, GLL parse-tree generation, Sci. Comput. Program. 78 (10) (2013) 1828–1844, http://dx.doi.org/10.1016/j.scico.2012.03.005.
[22] A. Johnstone, E. Scott, Modelling GLL parser implementations, in: B. Malloy, S. Staab, M. Brand (Eds.), Software Language Engineering, in: Lect. Notes Comput. Sci., vol. 6563, Springer, Berlin, Heidelberg, 2011, pp. 42–61.
[23] M. Tomita, Graph-structured stack and natural language parsing, in: Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Buffalo, New York, USA, 1988, pp. 249–257, http://www.aclweb.org/anthology/P88-1031.
[24] E. Scott, A. Johnstone, Gll parsing, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 177–189, http://dblp.uni-trier.de/db/journals/entcs/entcs253.html#ScottJ10.
[25] E. Scott, A. Johnstone, R. Economopoulos, BRNGLR: a cubic Tomita-style GLR parsing algorithm, Acta Inform. 44 (6) (2007) 427–461, http://dx.doi.org/10.1007/s00236-007-0054-z.
[26] D.E. Knuth, On the translation of languages from left to right, Inf. Control 8 (6) (1965) 607–639, http://dx.doi.org/10.1016/S0019-9958(65)90426-2.
[27] F.L. DeRemer, Simple LR(k) grammars, Commun. ACM 14 (7) (1971) 453–460, http://dx.doi.org/10.1145/362619.362625.
[28] S. Medeiros, F. Mascarenhas, R. Ierusalimschy, Left recursion in parsing expression grammars, in: Proceedings of the 16th Brazilian Conference on Programming Languages, SBLP'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 27–41.