Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciências da Computação

Uma abordagem para geração de entradas para PEG usando gramáticas LL(1)

Thiago do Vale Cabral

JUIZ DE FORA
Outubro, 2024

Uma abordagem para geração de entradas para PEG usando gramáticas LL(1)

Thiago do Vale Cabral

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciências da Computação

Bacharelado em Ciências da Computação

Orientador: Leonardo Vieira dos Santos Reis

Coorientador: Elton Máximo Cardoso

JUIZ DE FORA

Outubro, 2024

Uma abordagem para geração de entradas para PEG usando gramáticas LL(1)

Thiago do Vale Cabral

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO.

Aprovada por:

Leonardo Vieira dos Santos Reis
Doutor em Ciências da Computação

Elton Máximo Cardoso
Mestre em Ciências da Computação

Rodrigo Geraldo Ribeiro
Doutor em Ciências da Computação

Gleiph Ghiotto Lima de Menezes
Doutorado em Computação

JUIZ DE FORA

04 DE *Outubro*, 2024

À minha mãe por me ensinar que amar e cuidar através das minhas atitudes.
Ao meu pai por me ensinar a sonhar alto e ser persistente naquilo que acredito.
À minha irmã por me ensinar a ser exemplo através da bondade e caridade.
Aos meus amigos por me ensinar que a vida é mais leve quando dividimos os desafios e conquistas.

Resumo

Os analisadores LL(k) são uma classe específica de analisadores para Gramáticas Livres de Contexto (GLCs). Ao contrário das Gramáticas de Expressões de *Parsing* (PEGs), determinar se uma gramática LL(k) arbitrária terminará ao processar uma entrada é um problema decidível. Considerando que gramáticas LL(1) com uma pequena restrição descrevem a mesma semântica quando interpretadas como GLCs e quando interpretadas como PEGs, esta monografia propõe-se a gerar um subconjunto de PEGs bem formadas por meio das gramáticas LL(1). Para tal, é proposta uma implementação de um algoritmo, como uma biblioteca na linguagem de programação *Racket*, que gere gramáticas LL(1) e suas entradas válidas de forma aleatória por meio do processo de derivação de GLCs. Usando testes baseados em propriedades mostramos evidências que as gramáticas geradas se classificam como PEGs bem formadas.

Palavras-chave: PEG, GLC, LL(1), derivação, gramática.

Abstract

LL(k) parsers are a specific class of parsers for Context-Free Grammars (CFGs). Unlike Parsing Expression Grammars (PEGs), determining whether an arbitrary LL(k) grammar will terminate an input is a decidable problem. Considering that LL(1) grammars with a small restriction describe the same semantics when interpreted as CFGs as when interpreted as PEGs, this monograph proposes to generate a subset of well-formed PEGs by means of LL(1) grammars. To this end, it proposes an implementation of an algorithm, such as a library in the Racket programming language, which generates LL(1) grammars and their valid entries randomly through the process of deriving CFGs. It is concluded that, by performing property-based tests and analyzing the correctness of this algorithm, the randomly generated grammars are classified as well-formed PEGs.

Keywords: PEG, CFG, LL(1), derivative, grammar.

Agradecimentos

Aos meus pais, Leo Wagner e Lilian, por moverem mundos por mim e por minha irmã durante todos esses anos. Não há agradecimento o suficiente que se equipare a todo sacrifício, onde meus sonhos e de minha irmã foram colocados acima dos seus próprios sonhos. Gostaria de agradecer pelo amor incondicional, pelo cuidado, pela presença e pelos valores que sempre carregarei comigo.

À minha irmã Tayenne, por ser minha melhor amiga. Agradeço por dividir toda jornada desde pequenos, pelos conselhos, risadas, ensinamentos e valores. Muito obrigado por estar ao meu lado na decisão de me tornar um profissional na área de Ciências da Computação e pelo apoio que me ajudou a chegar até a linha de chegada.

Aos meus amigos por serem comunidade em minha vida. Agradeço me ensinarem que os desafios ficam mais leves quando temos alguém para dividir, e as conquistas se tornam mais gostosas ao ter alguém para brindá-las.

À Yuí, Meg e Mia, pelo amor incondicional, pelo conforto e por serem as melhores pets do mundo.

Ao professor orientador Leonardo Vieira dos Santos Reis, e ao coorientador Elton Máximo Cardoso por todo o processo de orientação do trabalho. Gostaria de agradecer por todo o zelo, paciência, carinho e paixão para com o projeto e pela sua importância à academia.

A todos os professores do Departamento de Ciência da Computação e aos funcionários do curso, que durante esses anos fundamentaram todo meu conhecimento científico na área e contribuíram para meu crescente amor pela Computação.

"Daria tudo o que sei pela metade do que ignoro."

René Descartes

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

DCC	Departamento de Ciência da Computação
ER	Expressão Regular
FNG	Forma Normal de Greibach
GLC	Gramática Livre de Contexto
PEG	Gramáticas de Expressões de Parsing
UFJF	Universidade Federal de Juiz de Fora

1 Introdução

Os modelos formais são fundamentais no desenvolvimento de linguagens de programação ao proporcionar uma base matemática sólida para a especificação e reconhecimento de linguagens, assim como suas inter-relações, classificações, estruturas, propriedades e características. Klein et al. (2012) afirmam que a comunidade desenvolvedora de linguagens de programação utiliza dos modelos formais como principais ferramentas de comunicação, seja para expressar novas construções ou até mesmo para validar sistemas de tipagem.

Entretanto, apesar dos modelos formais terem sido construídos manualmente por muitas décadas, o projeto de novas linguagens de programação tem se mostrado cada vez mais complexo e sujeito a falhas impercebíveis (KLEIN et al., 2012). Este é o principal motor motivador para o desenvolvimento de ferramentas mais robustas capazes de analisar e validar a corretude de novos projetos de linguagens de programação.

Uma das formas mais rápidas e simplificadas de se validar modelos formais é através dos testes de propriedade (CLAESSEN; HUGHES, 2000; FEITOSA; RIBEIRO; BOIS, 2020; HRITCU et al., 2013), sendo bibliotecas responsáveis pela geração de um grande volume de casos de teste de forma randomizada. Testes de propriedade são populares em paradigmas funcionais, como o arcabouço do *QuickCheck* para a linguagem de programação *Haskell* ou na linguagem de domínio específico *PLT Redex*.

Neste trabalho objetiva-se gerar entradas válidas para Gramáticas de Expressões de *Parsing* (do inglês *Parsing Expression Grammars* – PEGs) (FORD, 2004). Desta forma, em conjunto com o trabalho do Cardoso et al. (2022), obtém-se um arcabouço para testes baseados em propriedades de modelos e ferramentas baseados em PEGs. No entanto, dada uma PEG, gerar entradas válidas pode ser reduzida ao problema de determinar se a linguagem reconhecida por uma PEG é vazia, um problema indecidível (FORD, 2004). Portanto, alternativamente à geração de entradas a partir de PEGs, este presente trabalho propõe-se uma abordagem alternativa para geração das entradas: a trabalhar com gramáticas LL(1), que são Gramáticas Livres de Contexto (GLCs) preditivas com um símbolo de *lookahead*, ao invés de PEGs. De acordo com Mascarenhas, Medeiros e

Ierusalimschy (2014), quando aplicada uma pequena restrição, as gramáticas LL(1) descrevem a mesma linguagem ao ser interpretadas como GLCs ou PEGs. A possibilidade de transformar GLCs LL(k)-forte em PEGs equivalentes permite-nos deslocar do problema indecidível de geração de entradas para PEG para uma classe de gramáticas cuja mesma verificação é decidível. Portanto, visto a relação entre LL(1) e PEGs proposta por Mascarenhas, Medeiros e Ierusalimschy (2014), este trabalho foca-se na geração de palavras válidas para gramáticas LL(1) e que, consequentemente, também serão válidas para suas PEG equivalentes.

1.1 Objetivo

Neste contexto, o objetivo deste trabalho é construir um algoritmo gerador de gramáticas LL(1). Além disso, combina-se o resultado deste gerador com um algoritmo de geração de entradas válidas descrito pelo processo de derivação de gramáticas, formalizado por Henriksen, Bilardi e Pingali (2019). As gerações de gramáticas e entradas válidas devem ser integradas com o *Rackcheck*, uma biblioteca de testes de propriedade do *Racket* (FLATT, 2010; POPA, 2020).

1.2 Estrutura

Como estrutura deste presente documento, no Capítulo 2 introduzem-se os conceitos fundamentais sobre GLCs, PEGs e Gramáticas LL(1). Também são abordados os conceitos de derivação de GLCs e os trabalhos anteriores que motivaram o estudo da geração de entradas válidas. No Capítulo 3 abordam-se as estratégias para geração de entradas válidas de uma GLC e geração de gramáticas LL(1) na Forma Normal de Greibach. O Capítulo 4 discute acerca dos testes baseados em propriedade, seus resultados e restrições adotadas. Por fim, o Capítulo 5 conclui este trabalho e discute os possíveis trabalhos futuros.

2 Referencial Teórico

Neste capítulo são descritas as formalizações, termos e técnicas empregadas para o entendimento e contextualização do trabalho. Para tal, nas seções 2.1, 2.2 e 2.3 formaliza-se a definição geral de Gramáticas Livres de Contexto (GLCs), Gramáticas de Expressão de *Parsing* (PEGs) e Gramáticas LL(1), respectivamente. Além disso, na Seção 2.4 é apresentado o conceito base de derivação de gramáticas, crucial para elaboração do algoritmo de geração de entradas válidas proposto no capítulo 3. Na seção 2.5 discutem-se os trabalhos relacionados que motivaram esta presente pesquisa e, finalmente, na seção 2.6 é apresentada a linguagem de programação usada para o desenvolvimento da geração de palavras.

2.1 Gramáticas Livres de Contexto

Uma gramática livre de contexto é um modelo matemático usado para descrever linguagens formais. Essas gramáticas são compostas por um conjunto finito de regras de produção que, a partir de um símbolo inicial, permitem gerar todas as cadeias de caracteres pertencentes à linguagem definida pela gramática (HOPCROFT; MOTWANI; ULLMAN, 2006). Podemos definir uma GLC pela 4-tupla $G = (V, \Sigma, R, S)$, em que:

1. V é um conjunto finito, no qual cada elemento $v \in V$ é denominado de um símbolo não-terminal. Cada não-terminal define uma sub-linguagem da linguagem definida por G;

2. Σ é o conjunto finito de terminais, disjuntos de V, que compõem a formação de palavras. O conjunto de terminais também é chamado de alfabeto da linguagem definida por G;

3. R é um conjunto finito de regras de produção da forma $A \rightarrow \beta$, tal qual:

    - A é um símbolo não-terminal;

- β é uma *string* de símbolos pertencentes à união de V e Σ ($\beta \in (V \cup \Sigma)^*$);

4. S é o conjunto de símbolos $s \in V$, que definem a variável de início da formação de gramáticas na gramática G.

Os símbolos terminais são os elementos básicos das cadeias da linguagem. Por outro lado, os símbolos não-terminais funcionam como variáveis que podem ser substituídas por combinações de símbolos terminais e não-terminais, conforme definido pelas regras de produção. A distinção entre terminais e não-terminais é crucial para a estrutura de uma GLC (HOPCROFT; MOTWANI; ULLMAN, 2006).

As regras de produção são o coração de uma GLC. Cada regra define como um símbolo não-terminal pode ser substituído (ou reescrito) por uma sequência de símbolos terminais e não-terminais. Estas regras determinam as estruturas sintáticas válidas na linguagem descrita pela gramática (HOPCROFT; MOTWANI; ULLMAN, 2006).

O símbolo inicial é o ponto de partida para a geração de strings na linguagem. Todas as derivações em uma GLC começam com o símbolo inicial. A escolha do símbolo inicial é fundamental, ao definir a abrangência e os limites da linguagem que a gramática pode representar (HOPCROFT; MOTWANI; ULLMAN, 2006).

As gramáticas livres de contexto são particularmente notáveis por sua capacidade de representar muitas estruturas importantes em linguagens de programação e linguagens naturais, como a hierarquia de expressões matemáticas e a estrutura de sentenças. Elas são poderosas o suficiente para descrever a sintaxe de quase todas as linguagens de programação, mas não tão poderosas a ponto de descrever qualquer linguagem formal (HOPCROFT; MOTWANI; ULLMAN, 2006).

2.2 Gramáticas PEG

Parsing Expression Grammars, introduzidas por Ford (2004), são um tipo de gramática formal utilizada para definir a sintaxe de linguagens de programação e outros sistemas formais. Diferentemente das gramáticas livres de contexto, as PEGs são baseadas em expressões de análise que incluem operadores de sequência, escolha, zero ou mais, um ou mais, opcional, e *lookahead*. Estas expressões determinam como uma *string* de entrada é

analisada e aceita pela gramática.

Podemos definir formalmente uma PEG como uma 4-tupla $G = (V, \Sigma, P, S)$, em que:

1. V é um conjunto finito, no qual cada elemento $v \in V$ é denominado de um símbolo não-terminal.

2. Σ é o conjunto finito de terminais, disjuntos de V, que compõem a formação de palavras.

3. P é um conjunto finito de regras de produção da forma $A \leftarrow e$, tal qual:

   - A é um símbolo não-terminal;
   - e é uma expressão de *parsing*. Diferentemente das expressões regulares, a expressão hierárquica possui uma ordem de prioridade, conforme a tabela 2.17;

4. S é o conjunto de símbolos $s \in V$, que definem a variável de início da formação de gramáticas na gramática G.

Tabela 2.1: Prioridade das operações das expressões de *parsing*

Operador	Prioridade
(e)	5
$\&e, !e$	4
$e^*, e^+, e?$	3
$e1e2?$	2
$e1/e2?$	2

Uma PEG é composta por um conjunto de regras de produção, cada uma associada a um nome (um símbolo não-terminal) e definida por uma expressão de análise. Essas expressões são construídas usando símbolos terminais, símbolos não-terminais, e operadores que definem padrões de reconhecimento. Diferente das gramáticas livres de contexto, em que a ordem das regras não importa, em PEGs a ordem das regras e das escolhas nas expressões é significativa, estabelecendo uma prioridade no processo de análise (FORD, 2004).

Um aspecto chave das PEGs é que elas são determinísticas. Ao contrário das gramáticas livres de contexto, que podem gerar ambiguidades na análise sintática, as

PEGs garantem uma única interpretação de uma *string*, eliminando a ambiguidade. Isso é alcançado através do mecanismo de escolha ordenada, onde a primeira alternativa que permite uma análise bem-sucedida é a escolhida. Esse determinismo torna as PEGs particularmente atraentes para a análise de linguagens de programação, nas quais a ambiguidade é indesejável (FORD, 2004).

As PEGs são utilizadas em diversas aplicações, desde a análise de linguagens de programação até o processamento de linguagem natural. A clareza e a precisão das PEGs, combinadas com seu determinismo, tornam-nas uma ferramenta valiosa para os projetos de compiladores e interpretadores. Além disso, a abordagem de PEGs influencia o desenvolvimento de novas teorias e ferramentas em análise sintática, contribuindo para avanços significativos na área de teoria da computação e linguagens formais (FORD, 2004).

2.3 Gramáticas LL(1)

As gramáticas LL(1) representam uma classe específica de gramáticas livres de contexto, particularmente adequadas para análise preditiva *top-down*. A nomenclatura "LL(1)" indica que a análise é feita da esquerda para a direita (*Left-to-right*), construindo uma derivação mais à esquerda (*Leftmost derivation*) da *string* de entrada, com um olhar adiante de um símbolo (também chamado de *lookahead*). Este tipo de gramática é caracterizado por permitir que um analisador sintático decida qual regra de produção aplicar, olhando apenas um símbolo à frente na entrada (AHO et al., 2007).

A construção de um analisador para gramáticas LL(1) é facilitada pela previsibilidade das regras de produção. Tais analisadores utilizam uma tabela de análise, na qual cada entrada associa um símbolo não-terminal e um símbolo de *lookahead* a uma regra de produção específica. Isso elimina a necessidade de *backtracking*, tornando a análise mais eficiente em termos de tempo de execução (AHO et al., 2007).

Para uma gramática ser LL(1), ela deve satisfazer duas condições principais: não deve haver ambiguidades, o que significa que duas regras de produção diferentes não devem começar com o mesmo símbolo terminal; e deve ser livre de recursão à esquerda, já que a recursão à esquerda levaria a um *loop* infinito na análise *top-down*. Essas restrições

garantem que a análise possa ser realizada de forma eficiente e determinística (AHO et al., 2007).

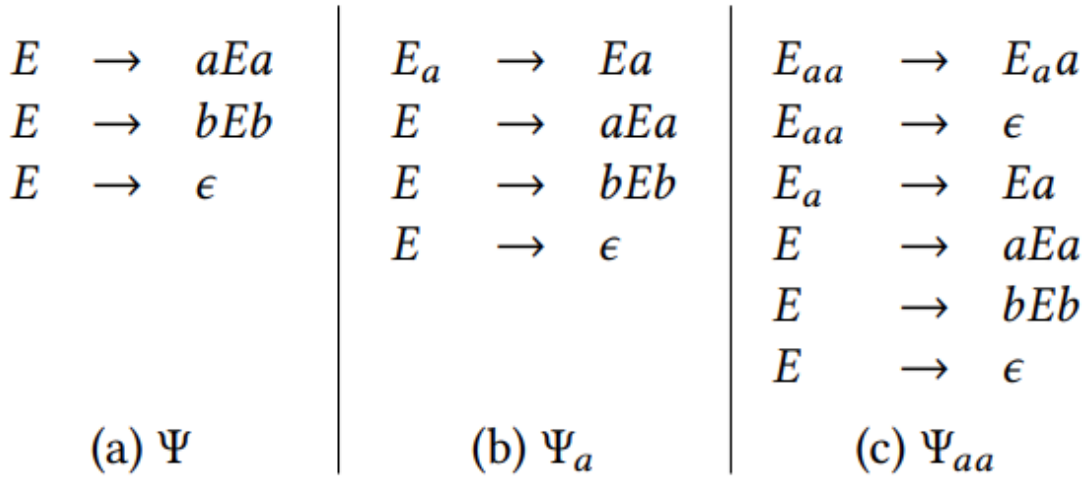
Uma das principais vantagens das gramáticas LL(1) é a facilidade de implementação de analisadores sintáticos, além de sua eficiência e determinismo. No entanto, sua principal desvantagem é a limitação na classe de linguagens que podem ser analisadas. Muitas linguagens de programação modernas não podem ser descritas por gramáticas LL(1) devido às suas restrições, o que leva ao uso de técnicas mais sofisticadas de análise sintática para essas linguagens.

Apesar de suas limitações, as gramáticas LL(1) são amplamente usadas em contexto acadêmico e em situações nas quais a simplicidade e a previsibilidade são mais importantes do que a capacidade de lidar com linguagens complexas. Elas servem como uma excelente introdução aos conceitos de análise sintática e teoria de linguagens formais, fornecendo uma base sólida para o estudo de técnicas de análise mais avançadas (AHO et al., 2007).

2.4 Derivação de Gramáticas

Informalmente, Henriksen, Bilardi e Pingali (2019) apresentam o processo de derivação conforme a figura 2.1:

Figura 2.1: Exemplo de uma derivação da gramática Ψ com uma *string* de entrada $w = aa$



Dada a formalização de GLCs descrita na Seção 2.1, Henriksen, Bilardi e Pingali

(2019) estendem o exemplo da figura 2.1 para a definição de primeira derivada de uma GLC, descrita como:

> A primeira derivada de uma gramática em relação a um terminal t é a construção de uma nova gramática Γ_t que gera a linguagem derivada L_t. Esta gramática é obtida adicionando novos não-terminais e produções baseadas em um conjunto de regras de inferência. Para cada produção na gramática original que pode ser anulável até o ponto do terminal t, cria-se uma nova produção na gramática derivada, substituindo t por ε e mantendo o restante das produções originais.

Henriksen, Bilardi e Pingali (2019) também definem a derivada de ordem superior:

> A derivada de ordem superior estende a noção de derivação para a sequência de terminais $w = t_1 t_2 ... t_n$, aplicando a regra de inferência da primeira derivada iterativamente. Isto significa que, para cada terminal na sequência, a gramática derivada anterior é usada para derivar a próxima gramática na sequência.

Nas seguintes subseções será discutida a formalização de derivação e Smart Constructors. Na Subseção 2.4.1 será abordado o conceito de derivações de Brzozowski para Expressões Regulares e, em seguida, na Subseção 2.4.3, esta formalização será expandida para o domínio de Gramáticas Livres de Contexto.

2.4.1 Derivações de Brzozowski

A derivação de expressões regulares é um conceito fundamental introduzido por Janusz A. Brzozowski em 1964 e oferecem uma abordagem sistemática para o processamento de expressões regulares, sendo uma base teórica robusta para a construção de algoritmos de *parsing*. A derivação de uma expressão regular com respeito a um símbolo é essencialmente uma nova expressão regular que descreve o conjunto de cadeias restantes após a remoção do símbolo inicial.

Conforme apresentado por Cardoso et al. (2021), a derivação L_a de uma linguagem L dado um símbolo a pode ser formalmente descrita como $L_a = \{\, w \mid aw \in L \}$ é a

derivada de uma Expressão Regular e em relação a um símbolo a, denotada por $\partial_a(e)$, é definida por recursão na estrutura de e como se segue:

$$\partial_a(\emptyset) = \emptyset \tag{2.1}$$

$$\partial_a(\varepsilon) = \emptyset \tag{2.2}$$

$$\partial_a(b) = \begin{cases} \varepsilon & \text{se } b = a, \\ \emptyset & \text{caso contrário.} \end{cases} \tag{2.3}$$

$$\partial_a(e\,e') = \partial_a(e)e' + \nu(e)\partial_a(e') \tag{2.4}$$

$$\partial_a(e + e') = \partial_a(e) + \partial_a(e') \tag{2.5}$$

$$\partial_a(e^*) = \partial_a(e)e^* \tag{2.6}$$

A regra do conjunto vazio (Regra 2.1) descreve que a derivada de uma expressão regular que representa o conjunto vazio $\emptyset$ é sempre $\emptyset$. Isso se deve ao fato de que não há palavras para derivar de um conjunto vazio e, portanto, qualquer símbolo derivado resultará em um conjunto vazio.

Conforme a regra da cadeia vazia (Regra 2.2), a derivada da cadeia vazia ε é também o conjunto vazio $\emptyset$., uma vez que a cadeia vazia não contém nenhum símbolo para ser consumido. Assim, qualquer derivada aplicada à cadeia vazia resulta em $\emptyset$.

Segundo a regra da cadeia terminal (Regra 2.3), a derivada de uma cadeia terminal b em relação a um símbolo a é ε caso $b = a$. Caso contrário, resulta em $\emptyset$, pois não há nenhuma correspondência.

Descreve-se a Regra da Concatenação como a derivada de uma expressão regular concatenada ee' descrita pela Regra 2.4. Ela é dada pela união entre a concatenação da derivada da expressão e com a segunda expressão e' e; a concatenação da expressão $\nu(e)$ e a derivada da expressão e'. Por sua vez, a função $\nu(e)$ verifica se a expressão e pode gerar uma cadeia vazia (ε). $\nu(e)$ pode ser expandida como se segue:

- $\nu(\varepsilon) = \varepsilon$: A expressão regular que representa a cadeia vazia sempre aceita a cadeia vazia;

- $\nu(\emptyset) = \emptyset$: O conjunto vazio nunca aceita nenhuma cadeia, incluindo a cadeia vazia;
- $\nu(a) = \emptyset$: Um símbolo individual a não pode ser a cadeia vazia;
- $\nu(e + e') = \nu(e) + \nu(e')$: A união de duas expressões regulares aceita a cadeia vazia se pelo menos uma das expressões aceitar a cadeia vazia;
- $\nu(ee') = \nu(e) \cdot \nu(e')$: A concatenação de duas expressões regulares aceita a cadeia vazia ambas as expressões aceitam a cadeia vazia;

A derivada da união de duas expressões $e + e'$ descrita pela Regra 2.5 é conhecida como a Regra da União, sendo ela dada pela união entre a derivada de e e a derivada de e'.

Por último, a derivada do Fecho de Kleene e^* descrita pela Regra 2.6 é dada pela derivada de e concatenada ao próprio fecho e^*.

2.4.2 Smart Constructors

Smart constructors são utilizados para identificar ERs equivalentes a módulo identidade e elementos anuláveis, ε e $\emptyset$, respetivamente. Uma equivalência de ER é denotada por $e \approx e'$, e seus axiomas de equivalência são descritos como (CARDOSO et al., 2021):

$$e + \emptyset \approx e \tag{2.7}$$

$$\emptyset + e \approx e \tag{2.8}$$

$$e\,\varepsilon \approx e \tag{2.9}$$

$$\varepsilon\,e \approx e \tag{2.10}$$

$$e\,\emptyset \approx \emptyset \tag{2.11}$$

$$\emptyset\,e \approx \emptyset \tag{2.12}$$

$$\emptyset^* \approx \emptyset \tag{2.13}$$

$$\varepsilon^* \approx \varepsilon \tag{2.14}$$

Conforme denotado pelas regras 2.7 e 2.8, o conjunto união de uma expressão e

e $\emptyset$, é reduzido para e visto que o resultado da união depende estritamente de e. Este comportamento é semelhante à regra da concatenação prevista em 2.9 e 2.10 entre as expressões e e ε, visto a cadeia vazia não interfere no resultado da expressão.

Entretanto, como previsto pelas regras 2.11 e 2.12, o conjunto da concatenação de uma expressão e e $\emptyset$, é reduzido para $\emptyset$ tendo em vista que na regra da concatenação ambas as expressões são determinantes para a ER resultante.

Finalmente, define-se pelas regras 2.13 e 2.14, os smart constructors do fecho de Kleene. Neste caso, a expressão podem ser reduzidas para $\emptyset$ e ε, respectivamente, visto que a repetição da expressão não alterará o resultado da ER.

2.4.3 Formalização para Gramáticas Livres de Contexto

De imediato, as derivações de Brzozowski podem ser estendidas para o contexto de GLCs. Entretanto, deve se considerar que a implementação das derivadas recursivamente esbarra com a existente natureza recursiva das GLCs, o que leva ao problema da não-terminação (MIGHT; DARAIS; SPIEWAK, 2011). Dado um símbolo não-terminal L e um terminal x, pode-se observar o seguinte exemplo de linguagem recursiva:

$$L = Lx + x \tag{2.15}$$

Seguindo a regra da união (2.5), aplica-se $\partial(L)$:

$$\partial(L) = \partial(Lx) + \partial(x) \tag{2.16}$$

Aplicando a regra da concatenação (2.4), pode-se expandir $\partial(L)$:

$$\partial(L) = \partial(L)x + \nu(L)\partial(x) + \partial(x) \tag{2.17}$$

Matematicamente, essa derivação é sensata. Computacionalmente ela não é. A

derivação representada em 2.17 irá se repetir indefinidamente até que o resultado de $\partial(L)$ seja encontrado. Como alternativa, Might, Darais e Spiewak (2011) apresentam três alternativas: a avaliação preguiçosa, a memoização, e o pontos fixos.

Avaliação Preguiçosa

Esta estratégia visa evitar que uma derivação faça uma descida infinita caso um não-terminal se autorreferencie.

> Especificamente, é necessário tornar os campos de concatenação, união e fecho de Kleene como chamadas *by-need*. Com campos *by-need*, a computação de quaisquer derivadas (potencialmente autorreferenciais) nesses campos é suspensa até que os valores nesses campos sejam necessários (MIGHT; DARAIS; SPIEWAK, 2011).

Memoização

A memoização é usada para evitar que o cálculo da derivada se torne ineficiente e entre em loops infinitos ao encontrar linguagens recursivas. A memoização armazena os resultados de cálculos anteriores, associando uma linguagem derivada a um determinado símbolo. Ao memorizar o derivado, ele estabelece uma associação quando reencontra uma linguagem que já viu (MIGHT; DARAIS; SPIEWAK, 2011).

Pontos Fixos

Somente a avaliação preguiçosa e a memoização não podem ajudar a contornar auto dependências como fizeram com a derivada (MIGHT; DARAIS; SPIEWAK, 2011). O ponto fixo é o valor estável que a derivada atinge após aplicar as operações repetidamente. Usar um ponto fixo garante que a derivada eventualmente converge e não entra em um ciclo infinito de recursão. Considere a seguinte nulidade da linguagem L recursiva à esquerda:

$$\partial(L) = \partial(L)x + \varepsilon \tag{2.18}$$

O ponto fixo aqui é a menor linguagem L que satisfaça a equação. Para tal avaliamos primeiro a expressão terminal da união. Neste caso se pode concluir que a solução de L contém a cadeia vazio ε e, a partir dela, todas as cadeias de x.

2.5 Trabalhos Relacionados

Nesta etapa do desenvolvimento do projeto foi realizada a pesquisa de diferentes trabalhos responsáveis pela contribuição com este estudo. Inicialmente, deve-se destacar que esta monografia é um desdobramento do estudo de Cardoso et al. (2022), tendo em vista que a geração de gramáticas LL(1) é uma hipótese alternativa à estratégia adotada em seu trabalho, visto em mais detalhes na Subseção 2.5.1.

Além disso, discute-se na Subseção 2.5.2 o estudo de Mascarenhas, Medeiros e Ierusalimschy (2014), evidenciando as condições para se produzir gramáticas que geram gramáticas que produzem a mesma linguagem em LL(1) e PEG.

2.5.1 Geração de PEGs

Como modelo alvo e precursor desta monografia, Cardoso et al. (2022) apresenta a motivação por trás de geração de PEGs. Para os autores, é necessário criar gramáticas que sejam não apenas sintaticamente corretas, mas também logicamente consistentes. Uma PEG bem formada é fundamental para análises de linguagem e *parsing* eficientes, evitando ambiguidades e problemas de interpretação comuns em gramáticas menos estruturadas.

Cardoso et al. (2022) abordam gerar expressões de *parsing* baseando-se em definições indutivas, um método matemático que permite a construção de expressões de forma progressiva e controlada. Ao utilizar definições indutivas, o algoritmo pode garantir que cada expressão de *parsing* gerada seja válida no contexto de uma PEG, assegurando a correção sintática e a consistência lógica. A construção de definições indutivas torna-se interessante no contexto de *PLT Redex* (FLATT, 2010), tendo em vista que a representação de sua sintaxe em Redex possibilita o uso manipulações como redução e fecho compatível e sua validação com a construção de julgamentos PLT resultantes.

Após a geração das expressões de *parsing*, os autores realizam as construções

de regras e a composição completa das gramáticas PEGs. Esta etapa é crucial, por ser onde as expressões individuais são combinadas para formar uma estrutura gramatical coesa. Entretanto, os autores demonstram a dificuldade de validação, mencionando a necessidade de um método para gerar entradas válidas e inválidas para testar a eficácia das gramáticas PEGs geradas. Isso é crucial para validar a robustez das gramáticas em diferentes cenários e garantir que elas possam lidar com uma variedade de casos de uso. A falta de um método sistemático para essa geração de entradas representa uma limitação na avaliação abrangente da eficácia das gramáticas.

O trabalho dos autores não aborda problemas de não-terminação. Em gramáticas de *parsing*, especialmente em contextos complexos, pode haver casos onde o processo de *parsing* entra em um *loop* infinito ou não consegue chegar a uma conclusão. A falta de estratégias para lidar com essas situações de não-terminação pode limitar a aplicabilidade das gramáticas PEGs em certos contextos ou linguagens.

Dado o contexto deste trabalho, torna-se interessante gerar PEGs a partir de gramáticas LL(1), a fim de se aproveitar a previsibilidade e a simplicidade das LL(1) enquanto se alcança a flexibilidade e a expressividade das PEGs. Isso pode ser particularmente útil em cenários onde a precisão e a desambiguação são críticas, e se torna necessário remover por completo os problemas de não-terminação. Além disso, a redução para o problema da gramática LL(1) permite a facilidade da descrição da geração de entradas válidas para a gramática, sendo o grande limitador apontado pelos autores.

2.5.2 Equivalência entre Gramáticas LL(1) e PEGs

Em seu trabalho, Mascarenhas, Medeiros e Ierusalimschy (2014) exploram a relação entre PEGs e gramáticas livres de contexto. Os autores apresentam uma semântica baseada em reconhecimento de GLCs para demonstrar a principal diferença entre PEGs e GLCs é o operador de escolha, que no caso das GLCs é não-determinístico. Mascarenhas, Medeiros e Ierusalimschy (2014) também formalizam a relação de PEGs com outras classes de gramáticas *top-down*, como LL(1), LL(k)-forte, e linguagens LL-regulares.

O determinismo das PEGs é central para a sua relação com as gramáticas LL(1), um subconjunto específico das GLCs. As gramáticas LL(1) são conhecidas por sua ca-

pacidade de analisar a entrada da esquerda para a direita, tomando decisões baseadas em um símbolo de previsão por vez. Os autores apontam que esta previsibilidade e a inequivocidade das PEGs formam a base para a correlação entre elas e discutem transformações que podem converter as gramáticas LL(1) em PEGs equivalentes, preservando sua estrutura e linguagem.

No caso de gramáticas LL(1) que não incluem expressões ε (*strings* vazias), os autores afirmam que existe uma correspondência direta com as PEGs. Se um analisador LL(1) pode escolher uma alternativa com um único símbolo de previsão, então um analisador PEG falhará em todas as alternativas que não sejam a correta.

Em contrapartida, para gramáticas LL(1) que incluírem produções vazias ε, Mascarenhas, Medeiros e Ierusalimschy (2014) ainda mostram que existe uma correspondência entre essas gramáticas e as PEGs, desde que a ordenação das expressões de escolha respeite certas condições. Isto significa introduzir um marcador de fim de entrada e restringir árvores de prova para considerar apenas aquelas que consomem a entrada e deixam apenas o marcador para garantir que a natureza determinística se mantenha.

2.6 Racket

Racket é uma linguagem de programação que se destaca por sua flexibilidade e capacidade de criação de novas sintaxes e linguagens de domínio específico, permitindo o desenvolvimento de ferramentas personalizadas. Derivada da família de linguagens LISP, ela é amplamente utilizada em contextos acadêmicos, tanto no ensino de conceitos de programação quanto na pesquisa em linguagens de programação e sistemas formais (FINDLER et al., 2002).

Em termos de utilidade, Racket é usada tanto para pequenos projetos quanto para sistemas complexos. Ela é especialmente valorizada em áreas como ensino de linguagens de programação e prototipagem de linguagens de domínio específico (FELLEISEN et al., 2015).

3 Geradores de entradas válidas

Neste capítulo é apresentada a metodologia adotada neste trabalho baseada na estratégia utilizada por Cardoso et al. (2022) para geração de PEGs. Entretanto, enquanto Cardoso et al. (2022) restringe o domínio de geração para PEGs bem formadas, este trabalho aborda uma restrição distinta tratando estas gramáticas como as da classe LL(1). A redução deste problema para o de gramáticas LL(1) torna-se razoável tendo em vista que Mascarenhas, Medeiros e Ierusalimschy (2014) demonstram que PEGs e gramáticas LL(1) reconhecem a mesma linguagem.

Adicionalmente, este trabalho propõe a criação de um algoritmo de geração de entradas válidas para gramáticas a serem geradas. É importante destacar aqui que, enquanto a geração de gramáticas está definida para a classe das LL(1), a geração de entradas válidas poderá ser usada para gramáticas livres de contexto em geral, o que será discutido com maior profundidade na Seção 3.1.

Para validação, uso e integração, este trabalho utiliza do *framework* de testes baseados em propriedade, conforme descrito no Capítulo 4. Assim, será possível verificar uma ampla variedade de propriedades das gramáticas e entradas geradas com o mínimo de teste codificado.

O código para geração de palavras, gramáticas LL(1), e seus respectivos testes foram desenvolvidos e disponibilizados no seguinte repositório: ⟨https://github.com/lives-group/cfg-entry-generator⟩.

3.1 Implementação da geração de entradas válidas

Para gerar entradas válidas, propõe-se a abordagem de gerar palavras através da escolha aleatória de símbolos seguida da derivação da produção, conforme demonstrado na Seção 2.4.

3.1.1 Definição das structs

Inicialmente, são definidas as structs que configuram uma GLC em Racket, como se pode ver abaixo:

```
1 (struct Production (nt rhs) #:transparent)
2 (struct T  (String) #:transparent)
3 (struct Alt (l r) #:transparent)
4 (struct Seq (l r) #:transparent)
5 (struct NT  (String) #:transparent)
```

Para este projeto uma GLC é definida em Racket como uma lista de produções, sendo a produção inicial convencionalmente a mesma produção sujeita à derivação. Define-se uma *Production* como um par de símbolo não-terminal e sua expressão à direita rhs (*right hand side*). Este par é memoizado em uma estrutura de dicionário para futuras substituições de derivação.

Uma expressão contida em rhs pode ser definida como um terminal T que corresponde a um símbolo final para geração da palavra. Ela também pode ser definida como um não-terminal NT, que pode ser substituído pela sua expressão rhs correspondente memoizada no dicionário.

A regra da união 2.5 e a regra da concatenação 2.4 são representadas por Alt e Seq, respectivamente. Ambos são representados por uma expressão à esquerda l e uma expressão à direita r. Múltiplas uniões e concatenações podem ser representadas pelo aninhamento das estruturas de Seq e Alt.

Finalmente, não há uma representação direta do fecho de Kleene em GLC. A repetição da expressão é dada pela recursão direta ou indireta aplicada pela substituição de um não-terminal por sua expressão rhs correspondente.

Dadas as definições em Racket acima, pode-se observar que dado o seguinte exemplo de gramática:

$$S \rightarrow A \,|\, B \tag{3.1}$$

$$A \rightarrow ab \tag{3.2}$$

$$B \rightarrow cd \tag{3.3}$$

Esta gramática pode ser representada em Racket como a seguinte lista de regras de produção:

```
(list
    (Production (NT S) (Alt (NT A) (NT B))
    (Production (NT A) (Seq (T a) (T b))
    (Production (NT B) (Seq (T c) (T d))
)
```

3.1.2 Smart constructors

Para a construção das structs presentes na Subseção 3.1.1, tornou-se necessário também elaborar seus construtores:

```
; Construtor para regra da união.
(define (alt l r)
  (cond
    ((and (rhs-empty? l) (rhs-empty? r)) ε)
    ((and (rhs-invalid? l) (rhs-invalid? r)) ∅)
    ((rhs-invalid? l) r)
    ((rhs-invalid? r) l)
    (else (Alt l r))))

; Construtor para regra da concatenação.
(define (seq-lazy l-thunk r-thunk)
  (define l (l-thunk))
  (cond
    [(rhs-invalid? l) ∅]
    [(rhs-empty? l) (r-thunk)]
    [else (define r (r-thunk))
          (cond
            [(rhs-invalid? r) ∅]
            [(rhs-empty? r) l]
            [else (Seq l r)])]
  ))
```

Para o construtor da regra da união, é necessário avaliar ambas as expressões l e r. Caso ambas sejam cadeias vazias, a união pode ser substituída por ε. Analogamente, a simplificação para um conjunto vazio pode ser feita caso ambas as expressões configurem em $\emptyset$. Caso apenas uma de suas expressões resulte em $\emptyset$, a união pode ser simplificada pela outra expressão válida.

Em contrapartida, a regra da concatenação envolve um construtor com avaliação preguiçosa, tendo em vista que caso a expressão l seja um conjunto vazio, todo o conjunto

concatenado pode ser simplificado por $\emptyset$, sem que a avaliação de r. Posteriormente, caso apenas uma de suas expressões resulte em ε, a concatenação pode ser simplificada pela outra expressão não vazia.

3.1.3 Implementação da nulidade

Para a verificação da nulidade de uma expressão, define-se delta seguindo as definições de nulidade na Subseção 2.4.1. Entretanto, aqui torna-se necessário fazer a avaliação preguiçosa da união e concatenação para evitar recursões infinitas. Para estes casos, prioriza-se avaliar terminais antes de avaliar não-terminais a fim de se encontrar simplificações e resoluções de ponto fixo.

3.1.4 Implementação da derivada

O algoritmo 1 descreve em alto nível como uma produção *rhs* é derivada em relação a um símbolo c, tendo uma gramática G como referência.

Algoritmo 1: Derivação de gramáticas livres de contexto

Input: Produção *rhs*, Gramática G, Símbolo c
Output: Nova produção

```
1  switch rhs do
2      case Vazio
3          Erro;
4      endsw
5      case Símbolo terminal
6          rhs_c ← derivateTerminal(rhs);
7          return rhs_c
8      endsw
9      case Símbolo não-terminal
10         rhs_c ← derivateNonTerminal(rhs);
11         return rhs_c
12     endsw
13     case Sequência
14         first ← getFirstFromSeq(rhs);
15         second ← getSecondFromSeq(rhs);
16         rhs_c ← derivateSeq(first, second);
17         return rhs_c
18     endsw
19     case isAlt(rhs)
20         first ← getFirstFromAlt(rhs);
21         second ← getSecondFromAlt(rhs);
22         rhs_c ← derivateAlt(first, second);
23         return rhs_c
24     endsw
25     else
26         Erro;
27     end if
28 endsw
```

A função deste algoritmo é, dado um símbolo terminal c, transformar uma regra de produção S de uma gramática em uma nova produção S_c, caso o símbolo terminal seja válido. Uma vez que a produção S_c gerada é vazia, é possível afirmar que se chegou em uma palavra gerada válida. Portanto, é razoável aplicar o Algoritmo 1 em *loop* até que encontrar uma produção vazia juntamente com uma lista de símbolos resultantes, descrito pelo algoritmo 2.

Algoritmo 2: Gerador de palavras por derivação de gramáticas

Input: Gramática G
Output: Palavra válida

```
1 rhs ← getFirstProduction();
2 word ← List.create();
3 while isNotEmpty(rhs) do
4 |   c ← chooseSymbol();
5 |   rhs ← derivateGrammar(rhs, G, c);
6 |   word.append(c);
7 end while
8 return word
```

Em linhas gerais, a abordagem do Algoritmo 2 é satisfatória, mas não o suficiente. Em casos de gramáticas recursivas à esquerda, a escolha aleatória do caminho de derivação pode levar a derivações infinitas porque o critério de escolha é aleatório.

Alternativamente, o Algoritmo 2 foi modificado para gerar todas as possibilidades de derivações em pares com palavras geradas. Esta estratégia troca a escolha de símbolos por escolhas de palavras bem definidas:

Algoritmo 3: Melhoria do gerador de palavras por derivação de gramáticas

Input: Gramática G
Input: Número de iterações maxI
Output: Palavra válida

```
1 i ← 0;
2 firstPair ← Pair.create(getEmptyWord(), getFirstProduction());
3 pairs ← List.create(firstPair);
4 while i < maxI do
5 |   pairs ← derivateAllPairs(pairs);
6 |   i ← i + 1;
7 end while
8 word ← chooseWord(pairs);
9 return word
```

Como exemplo de implementação deste algoritmo, observa-se a derivação da gramática de exemplo apresentada na Subseção 3.1.1:

$$S \rightarrow A \mid B$$

$$A \rightarrow ab$$

$$B \rightarrow cd$$

Ela pode ser derivada em relação ao símbolo a ou ao símbolo c. Ambas as escolhas serão percorridas:

$$\partial(S, a) = \partial(A, a) + \partial(B, a)$$
$$\partial(S, c) = \partial(A, c) + \partial(B, c)$$

Aplicando a substituição das produções memorizadas referentes a A e B, tem-se que:

$$\partial(S, a) = \partial(ab, a) + \partial(cd, a)$$
$$\partial(S, c) = \partial(ab, c) + \partial(cd, c)$$

Aplicando a regra da concatenação:

$$\partial(ab, a) = \partial(a, a)b + \nu(a)\partial(b, a)$$

$$\partial(ab, a) = \varepsilon\, b + \emptyset\, \partial(b, a)$$

$$\partial(ab, a) = \varepsilon\, b$$

$$\partial(ab, a) = b$$

$$\partial(ab, c) = \partial(a, c)b + \nu(a)\partial(b, c)$$

$$\partial(ab, c) = \emptyset\, b + \emptyset\, \partial(b, c)$$

$$\partial(ab, c) = \emptyset\, b$$

$$\partial(ab, c) = \emptyset$$

$$\partial(cd, a) = \partial(c, a)d + \nu(c)\partial(d, a)$$

$$\partial(cd, a) = \emptyset\, d + \emptyset\, \partial(d, a)$$

$$\partial(cd, a) = \emptyset\, d$$

$$\partial(cd, a) = \emptyset$$

$$\partial(cd, c) = \partial(c, c)d + \nu(c)\partial(b, c)$$

$$\partial(cd, c) = \varepsilon\, d + \emptyset\, \partial(b, c)$$

$$\partial(cd, c) = \varepsilon\, d$$

$$\partial(cd, c) = d$$

O que implica que:

$$\partial(S, a) = \partial(ab, a) + \partial(cd, a)$$
$$\partial(S, a) = b + \emptyset = b$$

$$\partial(S, c) = \partial(ab, c) + \partial(cd, c)$$
$$\partial(S, c) = \emptyset + d = d$$

Após um primeiro passo, forma-se a seguinte lista de tuplas:

$$(S_a \rightarrow b),\ a$$
$$(S_c \rightarrow d),\ c$$

Para cada par apresentado acima, tem-se à esquerda a nova regra de produção resultante do passo de derivação e, à direita, o agregado de símbolos que irá formar a palavra. Como as novas regras de produção ainda não são anuláveis, um novo passo de derivação é realizado recursivamente:

$$(S_{ab} \rightarrow \varepsilon),\ ab \tag{3.4}$$
$$(S_{cd} \rightarrow \varepsilon),\ cd \tag{3.5}$$

Ao fim deste passo, tem-se as tuplas elegíveis para escolha, visto que ambas são anuláveis. Isto implica que o gerador poderá escolher ente as palavras *ab* ou *cd*.

3.2 Implementação da geração de gramáticas LL(1)

No que se trata à geração de gramáticas LL(1), prioriza-se garantir a constância da geração desta gramática. Para tal, é preciso garantir, conforme demonstrado na Seção 2.3, que esta gramática não é ambígua, não possui recursão à esquerda e possui um conjunto de

First e *Follow* disjuntos.

Para garantir que estes requisitos sejam atendidos, foi escolhida como estratégia a geração de gramáticas na Forma Normal de Greibach (FNG). Pode-se afirmar que uma gramática livre de contexto está na forma normal de Greibach se toda produção for da forma $A \rightarrow b\alpha$, em que $A \in V, b \in \Sigma$ e $\alpha \in (V \cup \Sigma)^*$.

Isto significa que uma produção deve seguir um formato $A- > a\alpha$, onde:

1. A é um não-terminal;

2. a é um obrigatoriamente um terminal;

3. α pode ser um terminal, não-terminal ou vazio.

A Forma Normal de Greibach permite que algoritmos *top-down* processem a entrada de maneira linear e determinística, além de remover naturalmente casos de recursão à esquerda, satisfazendo a condição para estabelecer sua equivalência com PEGs, conforme discutido na Seção 2.5.2. Além disso, sua implementação torna-se simples tendo em vista que a obrigatoriedade de se iniciar uma produção com um terminal.

Para geração de gramáticas na FNG, a estratégia utilizada foi de escolher um terminal distinto para cada primeiro símbolo da expressão em um conjunto união, removendo assim a ambiguidade da gramática. Após a escolha do primeiro símbolo, de cada expressão de uma união, faz-se a escolha dos símbolos a serem concatenados, sendo eles quaisquer terminais ou não-terminais presentes em V e Σ. Finalmente, para a expressão mais à direita do conjunto união é sempre atribuída um único símbolo terminal para garantir o ponto fixo da gramática.

4 Testes baseados em propriedade

Os testes baseados em propriedade são uma abordagem de teste de software que se concentra em verificar as propriedades gerais de um programa (CLAESSEN; HUGHES, 2000). Essa técnica difere dos testes unitários tradicionais, que geralmente testam pontos específicos com codificação explícita. Testes baseados em propriedade são particularmente úteis para identificar casos de borda e comportamentos inesperados que podem não ser cobertos por testes unitários convencionais.

Para este problema em específico, torna-se extremamente complicado e ineficiente utilizar de testes unitários convencionais, visto que o objetivo principal deste trabalho é gerar saídas aleatórias. Tendo em vista que muitos erros em *software* são causados por falhas generalizáveis, os testes baseados em propriedade se destacam por identificar estas falhas genéricas via geração aleatória de casos de testes (FLATT, 2010; POPA, 2020).

4.1 Rackcheck

Rackcheck é uma biblioteca de testes de propriedade para a linguagem de programação Racket, inspirada por ferramentas semelhantes de outras linguagens, como o QuickCheck de Haskell. Testes de propriedade são uma forma avançada de teste automatizado que se concentra em verificar se o código atende a propriedades gerais, ao invés de comportamentos específicos para entradas pré-definidas. O Rackcheck facilita a definição dessas propriedades e gera automaticamente uma variedade de entradas para testar se a implementação respeita todas as propriedades declaradas. Se uma entrada gerada aleatoriamente não passar no teste, o Rackcheck fornece essa entrada como um contra-exemplo.

4.2 Propriedades

Para este projeto, tornou-se necessário garantir que 4 propriedades fossem satisfeitas tendo em vista gramáticas produzidas pelo gerador descrito na Seção 3.2:

1. O gerador de gramáticas deve produzir apenas gramáticas LL(1). Neste teste, verifica-se se o conjunto First+ de cada não-terminal não se sobrepõem.

2. O gerador de gramáticas deve produzir apenas gramáticas na Forma Normal de Greibach. Assim garante-se que todas as gramáticas geradas são livres de recursão à esquerda e sempre possuirão um ponto fixo de geração;

3. Para cada gramática produzida pelo gerador, todas as palavras geradas devem ser aceitas pelo processo de derivação descrito pela Subseção 2.4.1;

4. Para cada gramática produzida pelo gerador, todas as palavras geradas devem ser aceitas pelo parser de PEGs descrito por Cardoso et al. (2022).

É importante garantir primeiramente que a gramática seja LL(1) para que se garanta a propriedade de que gramáticas LL(1) descrevem a mesma linguagem ao ser interpretadas como GLCs ou PEGs (MASCARENHAS; MEDEIROS; IERUSALIMSCHY, 2014). Tendo em vista esta equivalência, é possível garantir que se a palavra for válida ela deverá ser aceita por um parser de PEG. Por fim, a palavra ser aceita por derivação é uma garantia de que o processo de derivadas é correto e consistente para gramáticas LL(1) na FNG.

4.3 Dados de Teste

Para as propriedades apresentadas, foi proposta a geração de gramáticas utilizando os símbolos não-terminais A, B e C, e seus símbolos terminais a, b e c. Para cada propriedade, foram realizados 1000 testes baseados com um delimitador de até 6 passos de derivação para geração de palavras. Observou-se que, como cada item da lista de derivadas gerava múltiplos itens em um passo sucessor, esta lista cresceu exponencialmente. Este crescimento exponencial foi perceptivo, pois a partir do sétimo passo, os testes tiveram seus respectivos tempos de duração extremamente longos.

A totalidade de testes executou com resultado positivo. Aplicando a cobertura de testes, foi obtida uma cobertura de 99.01%, sendo 402 expressões testadas e 4 não testadas.

Para fins de cobertura foram avaliadas apenas as funções diretamente relacionadas com a geração de palavras, e não foram avaliados funções utilitárias para depuração.

4.4 Resultados

Ao executar os testes baseados em propriedades descritos na Seção 4.2, foi possível validar que as gramáticas geradas eram LL(1) e estavam na FNG. Além disso, pôde-se verificar que as palavras geradas são aceitas no parsing de PEGs, conforme esperado devido ao estudo de Mascarenhas, Medeiros e Ierusalimschy (2014).

4.4.1 Derivação de palavras geradas

Para o contexto de gramáticas na FNG, não foram identificados problemas de derivação. Entretanto, quando retiradas as restrições para gramáticas LL(1) e na FNG, foi possível observar que o teste de palavras aceitas por derivação falham para gramáticas com recursão à esquerda. Isto ocorre devido à recursão infinita e ambiguidade do passo de derivação.

Embora esta limitação não seja um empecilho para este estudo, o problema da recursão à esquerda pode ser um limitador quando queremos verificar se uma palavra é aceita mediante derivadas. Alternativamente, uma possível solução para este problema seria fazer a avaliação preguiçosa da recursão juntamente com a estratégia do ponto fixo descritos na Seção 2.4.3.

4.4.2 Complexidade temporal do gerador de palavras

Inicialmente era preferível uma abordagem ingênua da geração de palavras. Entretanto, a aleatoriedade da escolha da alternativa possibilitou a escolha de derivações infinitas, o que tornava o algoritmo imprevisível, pois o mesmo podia gerar um conjunto vazio em situações nas quais era possível formar palavras.

A alternativa de criar uma lista com todas as possíveis derivações resolve o problema da consistência, mas introduz um problema de complexidade temporal. Devido à natureza da recursão das gramáticas, produções com união de expressões implicam em

um crescimento exponencial na complexidade de tempo da execução do algoritmo.

Para a estratégia de lista de possibilidades de derivações, um possível futuro caminha é introduzir dois limitadores para o gerador de palavras: um limitador de quantidade de palavras em conjunto com um limitador de quantidade de interações. Caso um dos limitadores seja satisfeito, o algoritmo de geração pode ser encerrado.

Outra alternativa é aprimorar a abordagem ingênua original em uma geração $bottom - up$ buscando substituir símbolos não-terminais por suas expressões terminais de ponto fixo. Entretanto, esta alternativa limita o conjunto de palavras geradas para um conjunto de pontos fixos, podendo não ser tão interessante caso se deseja testar gerações de palavras maiores e mais complexas.

5 Conclusão

Neste trabalho, foi apresentado um algoritmo para geração de gramáticas LL(1) e suas entradas válidas, utilizando a linguagem Racket e a biblioteca de testes Rackcheck para testes baseados em propriedades. O estudo demonstrou a eficácia do algoritmo na geração de gramáticas que atendem aos critérios de uma gramática LL(1) e sua equivalência com Parsing Expression Grammars (PEGs).

Os testes mostraram que as gramáticas geradas eram válidas, e as entradas derivadas foram aceitas tanto pelo algoritmo de derivação quanto pelo parser de PEGs, confirmando a correspondência entre gramáticas LL(1) na Forma Normal de Greibach e PEGs. No entanto, foi identificado um desafio no tratamento de gramáticas com recursão à esquerda, que geraram problemas de derivação devido à recursão infinita.

Como perspectivas futuras, sugere-se implementar técnicas mais robustas, como avaliação preguiçosa e memoização, para lidar com essas limitações e otimizar o processo de geração de palavras. Além disso, a aplicação do algoritmo para outros tipos de gramáticas pode expandir ainda mais o escopo deste estudo.

Bibliografia

AHO, A. V. et al. Compiladores: Princípios, técnicas e ferramentas. In: . [S.l.: s.n.], 2007. v. 2ª edição.

CARDOSO, E. M. et al. The design of a verified derivative-based parsing tool for regular expressions. *CLEI Electronic Journal*, Centro Latinoamericano de Estudios en Informática, v. 24, n. 3, p. 2:1–2:14, 2021. Disponível em: ⟨https://clei.org/cleiej/index.php/cleiej/article/view/521⟩.

CARDOSO, E. M. et al. A type-directed algorithm to generate random well-formed parsing expression grammars. In: *SBLP '22: Proceedings of the XXVI Brazilian Symposium on Programming Languages*. [S.l.]: Association for Computing Machinery, 2022.

CLAESSEN, K.; HUGHES, J. Quickcheck: A lightweight tool for random testing of haskell programs. In: *ACM SIGPLAN Notices*. [S.l.]: Association for Computing Machinery, 2000. v. 35, p. 268–279. ISSN 0362-1340.

FEITOSA, S. da S.; RIBEIRO, R. G.; BOIS, A. R. D. A type-directed algorithm to generate random well-typed java 8 programs. In: *Science of Computer Programming*. [S.l.: s.n.], 2020. v. 196.

FELLEISEN, M. et al. A programmable programming language. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, 2015. p. 1–12. Disponível em: ⟨https://dl.acm.org/doi/10.1145/2784731.2784732⟩.

FINDLER, R. B. et al. The racket language. *ACM SIGPLAN Notices*, Association for Computing Machinery, v. 37, n. 9, p. 1–10, 2002. Disponível em: ⟨https://dl.acm.org/doi/10.1145/571157.571162⟩.

FLATT, M. Reference: Racket. In: *Technical Report PLT-TR-2010-1*. [S.l.]: PLT Design Inc, 2010. v. 3.

FORD, B. Parsing expression grammars: A recognition-based syntactic foundation. In: *31st Symposium on Principles of Programming Languages*. [S.l.]: Association for Computing Machinery, 2004. v. 196, p. 111–122.

HENRIKSEN, I.; BILARDI, G.; PINGALI, K. Derivative grammars: a symbolic approach to parsing with derivatives. In: *Proceedings of the ACM on Programming Languages*. [S.l.]: Association for Computing Machinery, 2019. v. 3, p. 1–18.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. Introduction to automata theory, languages, and computation. In: . [S.l.]: Addison-Wesley, 2006. v. 3rd ed.

HRITCU, C. et al. Testing noninterference, quickly. In: *18th ACM SIGPLAN International Conference on Functional Programming*. [S.l.: s.n.], 2013. v. 196.

KLEIN, C. et al. Run your research: On the effectiveness of lightweight mechanization. In: . [S.l.]: Association for Computing Machinery, 2012.

MASCARENHAS, F.; MEDEIROS, S.; IERUSALIMSCHY, R. On the relation between context-free grammars and parsing expression grammars. In: *Science of Computer Programming*. [S.l.: s.n.], 2014. v. 89, p. 235–250.

MIGHT, M.; DARAIS, D.; SPIEWAK, D. Parsing with derivatives: A functional pearl. *Proceedings of the 2011 ACM International Conference on Functional Programming (ICFP)*, ACM, p. 189–195, 2011. Disponível em: ⟨https://matt.might.net/papers/might2011derivatives.pdf⟩.

POPA, B. Rackcheck: property testing. In: . [s.n.], 2020. Disponível em: ⟨https://docs.racket-lang.org/rackcheck/index.html.Acesso21deOutubrode2023⟩.