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

Uma Formalização em PLT Redex do Algoritmo de Remoção de Recursão à Esquerda em GLCs

Ana Carolina Mendes Lino

JUIZ DE FORA
SETEMBRO, 2024

Uma Formalização em PLT Redex do Algoritmo de Remoção de Recursão à Esquerda em GLCs

Ana Carolina Mendes Lino

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Leonardo Vieira dos Santos Reis

Coorientador: Elton M. Cardoso

JUIZ DE FORA

SETEMBRO, 2024

Uma Formalização em PLT Redex do Algoritmo de Remoção de Recursão à Esquerda em GLCs

Ana Carolina Mendes Lino

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 CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Leonardo Vieira dos Santos Reis
Doutor em Ciência da Computação

Elton M. Cardoso
Mestre em Ciência da Computação

Rodrigo G. Ribeiro
Doutor em Ciência da Computação

Gleiph Ghiotto Lima de Menezes
Doutor em Ciência da Computação

JUIZ DE FORA

13 DE SETEMBRO, 2024

Ao meu namorado e aos amigos.

A minha família, pelo apoio e sustento.

Resumo

O presente trabalho aborda o problema da recursão à esquerda em Gramáticas Livre de Contexto (GLCs), uma questão que compromete o funcionamento de algoritmos de análise sintática descendente. Para contornar esse obstáculo, é possível utilizar o algoritmo de remoção de recursão à esquerda, que transforma gramáticas com recursão à esquerda em equivalentes sem essa característica. Um algoritmo clássico de remoção de recursão à esquerda em GLCs é, normalmente, ensinado em cursos de graduação em disciplinas de linguagens formais e autômatos e teoria dos compiladores. Com o objetivo de prover uma especificação formal e executável deste algoritmo, este trabalho apresenta uma formalização em PLT Redex da remoção de recursão à esquerda em GLCs. Utilizando a biblioteca Rackcheck, conduzimos testes baseados em propriedades para evidenciar que a gramática original e a versão sem recursão são equivalentes.

Palavras-chave: Recursão à esquerda, PLT Redex, Linguagens de programação, Rackcheck.

Abstract

The present work addresses the issue of left recursion in Context-Free Grammars (CFGs), a problem that hinders the functioning of top-down parsing algorithms. To circumvent this obstacle, it is possible to use the left recursion removal algorithm, which transforms grammars with left recursion into equivalent one without this characteristic. A classic left recursion removal algorithm in CFGs is typically taught in undergraduate courses in formal languages, automata, and compiler theory. Aiming to provide a formal and executable specification of this algorithm, this work presents a formalization in PLT Redex for the removal of left recursion in CFGs. Using the Rackcheck library, we conducted property-based tests to demonstrate that the original grammar and the version without recursion are equivalent.

Keywords: Left recursion, PLT Redex, Programming languages, RackCheck.

Agradecimentos

À minha mãe, Bárbara, e ao meu namorado, Thiago, que sempre estiveram ao meu lado durante todo esse processo, oferecendo apoio, incentivo e compreensão.

Aos professores Leonardo, Elton e Rodrigo pela orientação, amizade e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

"A persistência é o caminho do êxito".

Charles Chaplin

Conteúdo

Lista de Figuras

Lista de Abreviações

GLC Gramáticas Livre de Contexto

1 Introdução

Uma linguagem formal é definida como um conjunto de palavras sobre um alfabeto Σ. Existem vários mecanismos para especificar linguagens formais, dentre os quais um dos mais utilizados na Ciência da Computação é o formalismo de Gramáticas Livre de Contexto (GLC). Uma Gramática Livre de Contexto é definida como uma quádrupla (V, Σ, R, P), em que V é um conjunto finito de símbolos denominados de não-terminais, Σ é um alfabeto, $R \subset V \times (V \cup \Sigma)^*$ é um conjunto finito de regras e $P \in V$ é o não-terminal de partida. GLCs descrevem como gerar palavras utilizando regras de reescrita a partir do não-terminal de partida. Esse processo é denominado de derivação. Por exemplo, seja a GLC $G = (\{A, B, C\}, \{a, b\}, R, A)$, em que R contém as regras:

$$A \rightarrow B \mid a$$

$$B \rightarrow AC \mid b$$

$$C \rightarrow AB$$

Podemos gerar a palavra *aab*, aplicando-se sucessivamente as regras de reescrita $A \rightarrow B$, $B \rightarrow AC$, $A \rightarrow a$, $C \rightarrow AB$ e $B \rightarrow b$ a partir do não-terminal de partida, conforme apresentado abaixo:

$$A \Rightarrow B \Rightarrow AC \Rightarrow aC \Rightarrow aAB \Rightarrow aaB \Rightarrow aab$$

Uma gramática é considerada recursiva à esquerda se houver pelo menos uma regra de produção na qual um não-terminal A possa gerar a si mesmo seguido de uma cadeia α, como em $A \rightarrow A\alpha \mid \beta$, em que $\alpha \in (V \cup \Sigma)^*$ e $\beta \in ((V - \{A\}) \cup \Sigma)^*$ (AHO; SETHI; ULLMAN, 1986). Essa gramática apresentada resulta na produção de sentenças da forma $\beta\alpha^*$.

Uma gramática pode ser recursiva à esquerda de maneira indireta. Isso ocorre quando existe pelo menos uma regra de produção no qual um não-terminal A gera um

conjunto de símbolos que eventualmente levam a uma regra de produção que contém o mesmo não-terminal no lado esquerdo. Por exemplo, na gramática G, apresentada anteriormente, o não-terminal B é recursivo à esquerda indiretamente, pois podemos obter uma cadeia na derivação a partir de B em que o primeiro símbolo dessa cadeia será B. A derivação a seguir ilustra isso:

$$B \Rightarrow AC \Rightarrow BC$$

Gramáticas recursivas à esquerda são problemáticas para a construção de analisadores sintáticos descendentes, pois esses constroem uma derivação mais à esquerda para a palavra, i.e., a cada passo da derivação expande-se a variável mais à esquerda da forma sentencial. No entanto, se houver recursividade à esquerda na gramática, tais algoritmos entram em um laço infinito, pois sempre irão escolher o mesmo conjunto de regras de produção para o não-terminal recursivo ocasionando a sua não-terminação.

Para contornar esse problema, pode-se usar o algoritmo de remoção de recursão à esquerda (GREIBACH, 1965), que possibilita transformar uma gramática que possui recursão à esquerda em outra equivalente, sem essa recursão. Dada a sua importância, esse algoritmo é estudado em disciplinas de linguagens formais e autômatos, e teoria de compiladores. Assim, neste trabalho, objetivamos formalizar esse algoritmo de remoção de recursão à esquerda.

1.1 Objetivo

Apresentar uma formalização executável que emprega a semântica de redução para descreve o clássico algoritmo de remoção de recursão à esquerda, utilizando o PLT Redex (KLEIN et al., 2012).

1.2 Contribuições

A formalização do algoritmo clássico de remoção de recursão à esquerda em gramáticas livres de contexto oferece as seguintes contribuições:

1. **Execução e Compreensão:** A formalização é executável e foram feitos testes baseados em propriedades que evidenciam sua corretude. Isso permite estudar e analisar o processo de remoção da recursão à esquerda em um ambiente controlado, facilitando a compreensão dos detalhes do algoritmo e possibilitando comparações com outras abordagens e variações.

2. **Visualização Detalhada:** A função `traces` da biblioteca PLT Redex proporciona uma visualização detalhada do passo a passo da redução. Essa funcionalidade ajuda a entender como a recursão à esquerda é removida, oferecendo uma visão mais intuitiva do processo. A capacidade de visualização pode ser útil para fins educacionais, proporcionando uma melhor compreensão dos conceitos envolvidos.

3. **Base para Desenvolvimento Futuro:** A formalização desenvolvida serve como base para futuras pesquisas e desenvolvimento na área. Pode ser utilizada para explorar novas abordagens, testar melhorias no algoritmo e criar novas ferramentas. Além disso, pode servir como base para a formalização de outros algoritmos.

O repositório contendo o código desenvolvido está disponível em https://github.com/lives-group/Left-Recursion-Elimination-Formalization.

1.3 Estrutura

Na sequência deste documento explicamos o algoritmo de remoção de recursão à esquerda no Capítulo 2. O Capítulo 3 aborda PLT Redex, a linguagem utilizada para realizar a formalização, e a biblioteca de teste baseado em propriedades Rackcheck. No Capítulo 4 é apresentada a formalização do algoritmo de remoção de recursão à esquerda, nesse capítulo também discutem-se os testes de propriedades realizados e a cobertura atingida pelos mesmos. Os trabalhos relacionados e conclusão são apresentados nos Capítulos 5 e 6, respectivamente.

2 Gramáticas e Recursão

Neste capítulo, são abordados conceitos relacionados à teoria de linguagens formais e autômatos. Inicialmente, é apresentado o conceito de gramáticas. Em seguida, discutida a diferença entre recursão à direita e recursão à esquerda em gramáticas, bem como os problemas gerados na implementação de analisadores sintáticos. Por fim, apresentamos o algoritmo clássico para remoção de recursão à esquerda (GREIBACH, 1965).

2.1 Gramática

A gramática generativa, originariamente concebida por Noam Chomsky na década de 50 para estudar as linguagens naturais, como o inglês, é um modelo ou técnica para gerar uma linguagem (CHOMSKY, 1956). Baseia-se em um processo de reescrita, no qual um texto ou padrão é substituído por outro numa forma sentencial, conhecida como regra de reescrita. Por exemplo, uma gramática pode ter as regras $A \rightarrow BC \mid D$, indicando que o símbolo A pode ser substituído pela sequência BC ou por D.

Noam Chomsky (CHOMSKY, 1956) desenvolveu uma classificação em quatro níveis para as gramáticas, iniciando pelo tipo 0, que apresenta o maior grau de liberdade em suas regras, e progredindo até o tipo 3, que impõe maiores restrições. Os tipos 2 e 3 são frequentemente empregados na descrição de linguagens de programação, bem como na implementação de interpretadores e compiladores. Especificamente, a gramática livre de contexto, também conhecida como gramática do tipo 2, desempenha um papel fundamental na análise sintática, sendo responsável por gerar a classe de linguagens denominadas linguagens livres de contexto.

Formalmente, uma gramática livre de contexto é definida como uma quádrupla (SIPSER, 2013):

$$G = (V, T, P, S)$$

em que:

- V é um conjunto finito de símbolos que compõem o vocabulário da gramática, denominados não-terminais, ou seja, são os símbolos utilizados na descrição da linguagem;
- T é um conjunto finito e não-vazio de símbolos terminais da gramática, também conhecido como alfabeto. Esses símbolos são os elementos da linguagem em si, ou seja, são aqueles que podem ser usados para formar as sentenças;
- P representa o conjunto de todas as leis de formação, conhecidas como produções ou regras de substituição, que são utilizadas pela gramática para definir a linguagem;
- S é o símbolo inicial da gramática, S $\in$ V. É a partir desse símbolo que as sentenças da linguagem serão geradas.

2.2 Derivações

O processo de substituir o lado esquerdo pelo lado direito de uma regra é denominado derivação (SIPSER, 2013). Por exemplo, considere a gramática livre de contexto G1 contendo as regras abaixo:

1. $S \rightarrow S + S$
2. $S \rightarrow 1$
3. $S \rightarrow 2$

A cadeia "1 + 1 +2" pode ser obtida a partir da seguinte sequência de derivações:

$$
\begin{aligned}
S &\Rightarrow S + S && \text{(usando a regra 1)} \\
&\Rightarrow S + S + S && \text{(usando a regra 1)} \\
&\Rightarrow 1 + S + S && \text{(usando a regra 2)} \\
&\Rightarrow 1 + 1 + S && \text{(usando a regra 2)} \\
&\Rightarrow 1 + 1 + 2 && \text{(usando a regra 3)}
\end{aligned}
$$

Muitas vezes, uma estratégia é utilizada para determinar o próximo não-terminal que será expandido na derivação:

- Em uma derivação mais à esquerda, é sempre o não-terminal mais à esquerda que é expandido a cada passo da derivação;
- Em uma derivação mais à direita, é sempre o não-terminal mais à direita que é expandido a cada passo da derivação.

2.3 Recursão em gramáticas

As regras de uma gramática podem conter recursão, i.e., referência ao mesmo não-terminal da regra no lado direito da produção. Dois tipos de recursão são relevantes no contexto de linguagens formais e compiladores: recursão mais à direita e recursão mais à esquerda. Nas próximas subseções, cada uma delas é descrita em mais detalhes.

Recursão à direita

Uma gramática é considerada recursiva à direita se houver pelo menos uma regra de produção em que um não-terminal A que possa gerar uma cadeia α seguida por um símbolo não-terminal A novamente, como na forma $A \rightarrow \alpha A|\beta$, em que α pode conter variáveis e símbolos terminais e β pode conter variáveis e símbolos terminais, com exceção do símbolo A (AHO; SETHI; ULLMAN, 1986). Essa gramática apresentada resulta na produção de sentenças da forma α^* β.

Uma gramática também pode ser considerada recursiva à direita de maneira indireta, isso ocorre quando existe pelo menos uma regra de produção em que um não-terminal A possa gerar uma cadeia α seguida por um conjunto de símbolos que eventualmente levam a uma regra de produção que contém o mesmo não-terminal no lado direito. Por exemplo, considere a seguinte gramática:

$$A \rightarrow B|a$$

$$B \rightarrow C|CA$$

$$C \rightarrow a$$

Nessa gramática, o não-terminal B é recursivo à direita indiretamente, pois para derivar em si mesmo, precisa primeiro derivar em A, que por sua vez deriva em B novamente.

Recursão à esquerda

Uma gramática é considerada recursiva à esquerda se houver pelo menos uma regra de produção no qual um não-terminal A possa gerar um não-terminal A seguido de uma cadeia α, como na forma $A \rightarrow A\alpha|\beta$, em que α pode conter variáveis e símbolos terminais e β pode conter variáveis e símbolos terminais, com exceção do símbolo A (AHO; SETHI; ULLMAN, 1986). Essa gramática apresentada resulta na produção de sentenças da forma $\beta\alpha*$.

Uma gramática pode ser recursiva à esquerda de maneira indireta, isso ocorre quando existe pelo menos uma regra de produção no qual um não-terminal A gera um conjunto de símbolos que eventualmente levam a uma regra de produção que contém o mesmo não-terminal no lado esquerdo. Por exemplo, considere a seguinte gramática:

$$A \rightarrow B|a$$

$$B \rightarrow b|AC$$

$$C \rightarrow c|d$$

Nessa gramática, o não-terminal B é recursivo à esquerda indiretamente, pois para derivar em si mesmo, precisa primeiro derivar em A, que por sua vez deriva em B novamente.

A recursão à esquerda, direta ou indireta, pode levar a problemas de implementação e desempenho em alguns algoritmos de análise sintática descendentes (AHO; SETHI; ULLMAN, 1986), pois muitas técnicas e algoritmos utilizados no processamento de gramáticas não suportam esse tipo de recursão.

Os algoritmos descendentes, para análise sintática, realizam a expansão sempre escolhendo o não-terminal mais à esquerda. Deste modo, se houver recursividade à esquerda na gramática, o algoritmo pode entrar em um laço infinito, pois ele não conse-

guirá escolher uma regra de produção para o não-terminal recursivo e ficará preso nesse ponto. Por esse motivo, métodos descendentes de análise sintática não podem manipular gramáticas recursivas à esquerda e uma transformação que elimine a recursão à esquerda é necessária.

2.4 Remoção da recursão à esquerda

A eliminação da recursão à esquerda é uma técnica importante para garantir que algoritmos descendentes funcionem corretamente. Nesse sentido, existem abordagens para realizar a remoção da recursão à esquerda em uma gramática, e este trabalho tem como foco o algoritmo clássico de Greibach (GREIBACH, 1965).

O algoritmo de Greibach para remoção de recursão à esquerda aborda um problema comum em gramáticas, em que a recursão pode levar a resultados imprecisos ou indefinidos. Para resolvê-lo, o algoritmo utiliza duas etapas: primeiro, transforma recursão à esquerda indireta em recursão à esquerda direta e, em seguida, elimina recursão à esquerda direta. A eliminação da recursão à esquerda direta consiste em:

1. para cada não-terminal A recursivo cria um novo não-terminal A';
2. o não-terminal A passa a produzir somente as produções não-recursivas em que o não terminal A' é o último símbolo;
3. o não-terminal A' passa a produzir as produções recursivas de A sem o primeiro símbolo e com o não terminal A' no final;
4. o não-terminal A' deve produzir a cadeia vazia, ϵ.

O Algoritmo 1 detalha o processo de eliminação de recursão à esquerda indireta. O algoritmo começa definindo uma ordenação para os não-terminais e, para cada não-terminal, busca garantir que o primeiro símbolo de seu lado direito seja um terminal, ou um não-terminal, cuja posição na ordenação seja maior que a do lado esquerdo da regra. Esse processo prossegue até todas as regras terem como o primeiro símbolo o não-terminal presente no lado esquerdo da regra, ou um terminal qualquer, ou um não-terminal cuja

posição seja maior a do lado esquerdo da regra. Após esse processo, as recursões à esquerda diretas emergirão e são eliminadas.

Algoritmo 1: Eliminação de recursão à esquerda indireta

```
Entrada: Uma gramática G
Saída: A gramática G' resultante da transformação
1  início
2      ordenar os não-terminais como A_1, A_2, ..., A_n em sequência;
3      para i := 1 até n faça
4          para j := 1 até i − 1 faça
5              para cada produção A_j → A_iα faça
6                  remover A_j → A_iα da gramática ;
7                  para cada produção A_j → β faça
8                      adicionar A_i → βα na gramática ;
9                  fim para
10             fim para
11         fim para
12     fim para
13     eliminar as recursões diretas das A_i-produções ;
14 fim
```

Por exemplo, considere a seguinte gramática com recursão à esquerda:

$$A \to Ba$$

$$B \to Cd$$

$$C \to Aac \mid c$$

O primeiro passo do algoritmo é ordenar os não-terminais. Considere a seguinte ordenação: A, B, C. Na sequência, toda regra $X \to Y\alpha$ cujo não-terminal Y está anterior a X na ordenação deve ser removida. Nesse exemplo, a única regra que se enquadra nessa condição é a regra $C \to Aac$. Portanto, essa regra é substituída por outras regras com A sendo substituída pelo seus lados direitos, resultando na nova gramática:

$$A \to Ba$$

$$B \to Cd$$

$$C \to Baac \mid c$$

A regra resultante $C \to Baac$ também tem o primeiro símbolo, B, um não-

terminal que vem antes de C na ordenação dos não-terminais. Portanto, deve ser eliminada e, obtemos como resultado, a nova gramática:

$$A \rightarrow Ba$$

$$B \rightarrow Cd$$

$$C \rightarrow Cdaac \mid c$$

Após essa etapa, a recursão à esquerda do não-terminal C emergiu e pode ser eliminada. A gramática final após a eliminação da recursão à esquerda é:

$$A \rightarrow Ba$$

$$B \rightarrow Cd$$

$$C \rightarrow cC'$$

$$C' \rightarrow daacC' \mid \epsilon$$

3 Racket

Neste trabalho, utilizamos a linguagem Racket (FLATT; PLT, 2010). Racket é uma linguagem multiparadigma que oferece suporte tanto à programação funcional quanto procedural e pode ser vista como um dialeto moderno de Scheme/Lisp (FLATT; PLT, 2010). As próximas seções dedicam-se a apresentar duas bibliotecas Racket utilizadas neste trabalho: PLT Redex e Rackcheck.

3.1 PLT Redex

PLT Redex é uma biblioteca Racket que permite a rápida prototipação da sintaxe, semântica e predicados indutivos e tem sido utilizada com sucesso na modelagem de diversos resultados envolvendo linguagens de programação (KLEIN et al., 2012).

Regras semânticas expressas em Redex seguem a estratégia conhecida como semântica de redução (FELLEISEN; HIEB, 1992), em que cada regra semântica determina como os termos devem ser reescritos até obter o resultado final da computação. Regras são definidas utilizando o comando `reduction-relation` da seguinte forma:

(**reduction-relation** *linguagem regras-de-reescrita* ...)

regras-de-reescrita = (--> padrão termo *extras* ...)

A linguagem define como o Redex interpreta os padrões. Cada regra de reescrita começa com o símbolo "`--->`" e contém um padrão à esquerda, representando a entrada, e um termo à direita, representando a saída. Opcionalmente, após o padrão e o termo, pode-se incluir o nome da regra de redução, cláusulas *where* e/ou condições secundárias.

Além de recursos para especificação semântica, o Redex fornece ferramentas de visualização para explorar o comportamento da redução. Uma dessas ferramentas é a função `traces`. Esta função aceita uma relação de redução e um termo, e então mostra um gráfico que demonstra como esse termo é reduzido.

O exemplo a seguir demonstra o PLT Redex na definição de uma linguagem chamada *And*. Primeiramente, apresentamos a sintaxe abstrata de *And* e, na sequência, a semântica desta linguagem sobre valores booleanos de forma a interpretar o símbolo $\wedge$ como o conectivo de conjunção: as duas primeiras regras mostram que a constante $\top$ é o elemento neutro para a conjunção e as duas últimas regras mostram que $\bot$ é o elemento nulo para esta operação lógica.

```
#lang racket  ;Define a linguagem
(require redex) ;Importa o Redex

; Define a linguagem And
(define-language And
  [L ::= ⊤ ⊥ p
     (∧ L L)]
  [p ::= variable-not-otherwise-mentioned])

; Regras da reduction
(define r
  (reduction-relation And
     (--> (∧ ⊤ L) L)
     (--> (∧ L ⊤) L)
     (--> (∧ ⊥ L) ⊥)
     (--> (∧ L ⊥) ⊥)
    ))
```

A Figura 3.1 ilustra o resultado da execução da função traces sobre a seguinte expressão booleana:

```
;Traces para exibir os passos e resultado
(traces r (term (∧ ⊤ (∧ (∧ p ⊥) ⊤))))
```

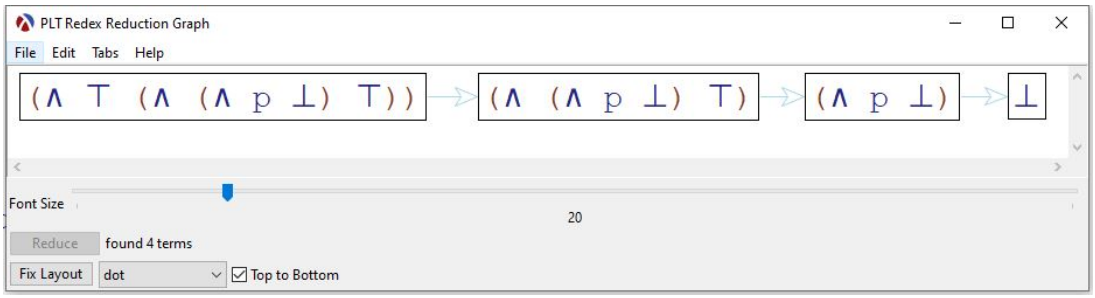

Figura 3.1: Redução do termo $(\wedge \top (\wedge (\wedge \, p \, \bot) \top))$

3.2 Rackcheck

Rackcheck (POPA, 2021) é uma biblioteca de testes baseados em propriedades para Racket (FLATT; PLT, 2010). Em vez de testar apenas alguns casos de entrada e saída, os testes baseados em propriedades geram automaticamente um grande número de casos de teste verificando se o código a ser testado satisfaz uma propriedade para todas as entradas produzidas. Caso alguma entrada não satisfaça a propriedade, este caso de teste é retornado como um contra-exemplo e pode ser utilizado pelo desenvolvedor para depurar a falha presente no código.

O exemplo a seguir demonstra o uso do Rackcheck para testar a seguinte propriedade da função que calcula a distância euclidiana entre dois pontos: para quaisquer pontos p1 e p2, temos que distance p1 p2 = distance p2 p1.

```
1  ; Cacula a distancia entre pontos
2  (define (distance p1 p2)
3    (define dx (- (point-x p1) (point-x p2)))
4    (define dy (- (point-y p1) (point-y p2)))
5    (sqrt (+ (* dx dx) (* dy dy))))
6
7  (module+ test
8    ; Importa o rackcheck
9    (require rackcheck
10             rackunit)
11
12   ; Gera os pontos
13   (define gen:point
14     (gen:let ([x (gen:integer-in -9999 9999)]
15               [y (gen:integer-in -9999 9999)])
16       (point x y)))
17
18  ; Define as propriedades a serem testadas
19   (check-property
20    (property verificaDistancia ([p1 gen:point]
21                [p2 gen:point])
22      (check-equal? (distance p1 p2) (distance p2 p1)))))
```

Propriedades são especificadas utilizando a função property que deve receber os parâmetros a serem utilizados na propriedade e a propriedade em si, um valor booleano. A verificação da propriedade é iniciada pela função check−property que inicia a geração de valores para cada um dos parâmetros da propriedade fornecida como argumento. Por padrão, o Rackcheck produz 100 entradas e verifica se cada uma delas satisfaz a propriedade fornecida.

A biblioteca Rackcheck oferece configurações para produzir um número diferente de casos de teste e utilidades para construção de geradores de entradas e para combinar propriedades usando operadores da lógica. A figura a seguir apresenta o resultado da execução da propriedade da função distance.

Figura 3.2: Resultado dos testes executados

4 Formalização

Neste capítulo descrevemos nossa formalização do algoritmo de Greibach para remoção de recursão à esquerda em GLCs. Primeiro, apresentamos a sintaxe da linguagem para expressar GLCs e, na sequência, mostramos como modelar o processo de eliminação de recursão à esquerda como uma semântica de redução utilizando PLT Redex.

4.1 Definição da Linguagem

Iniciamos nossa formalização definido a sintaxe de uma linguagem para representação de GLCs utilizando a construção define−language do Redex, que representa a sintaxe de linguagens utilizando uma notação similar a conhecida Backus-Naur Form (BNF).

```
1  (define-language G
2    [nt V]
3    [trm number]
4    [rhs (seq ...)]
5    [seq (t ...)]
6    [ord (nt flag)]
7    [flag 0 1]
8    [t nt trm]
9    [prd (nt rhs)]
10   [V variable-not-otherwise-mentioned]
11   [prds (prd ...)]
12   [ords (ord ...)]
13   [grammar (ords prds)])
```

A linguagem G tem como objetivo definir uma gramática válida, conforme indicado na linha 13. Uma *grammar* (linha 13) é composta por duas estruturas principais: uma lista de ordens (*ords*) na linha 12, que determina a ordem das produções na gramática, e uma lista de produções (*prds*) na linha 11, que especifica as regras da gramática. A

ordem *ord* é definida por um não-terminal, *nt*, e uma marcação que indica se a regra de derivação associada a este não-terminal já teve a recursão à esquerda removida. Regras de derivação são representadas pelo não-terminal *prd* composto por seu não-terminal e o respectivo lado direito, representado pela variável *rhs*. Cada lado direito de regras é representado por uma sequência de símbolos que, por sua vez, podem ser um terminal ou uma variável. Na gramática definida (*grammar*), os não-terminais (*nt*) são representados por letras maiúsculas (V) (linha 10), e os terminais (*trm*) por valores numéricos inteiros.

Como um exemplo de como a linguagem anterior é capaz de representar GLCs quaisquer, considere a seguinte gramática S:

$$A \rightarrow A2 \mid B$$
$$B \rightarrow 1 \mid BA$$

A representação da gramática S utilizando a linguagem G é:

```
(
  ((A 0) (B 0))
  (
    (A ((A 2) (B)))
    (B ((1) (B A)))
  )
)
```

A primeira linha define a ordem entre os não-terminais e suas respectivas marcações. As linhas 4 e 5 do exemplo representam as regras para os não-terminais A e B, respectivamente.

4.2 Relação de Redução

Após especificar a sintaxe da linguagem a ser utilizada, o próximo passo é definir a relação de redução. A relação de redução consiste em uma sequência de expressões que representam as possíveis maneiras pelas quais uma expressão da linguagem G pode ser avaliada e substituída. Para o algoritmo clássico de remoção de recursão à esquerda, foram definidas três regras de reescrita, a saber:

1. **Caso base:**

```
1 (--> [((nt 0) (nt_1 0) ...)
2  ((nt ((trm t ...) ... (nt_2  t_1 ...) ...)) prd ...)]
3
4  [((nt 1) (nt_1 0) ...)
5  (concat-prds
6   (check-left-recursion (nt ((trm t ...) ... (nt_2 t_1 ...)
       ...)) (prd ...))
7   (prd ...))] )
```

   Neste caso verifica, na linha 1, se todos os itens *flag* da lista *ords* da gramática definida possuem valor igual a 0. Isso indica que o processo de redução está iniciando e que nenhuma regra de derivação foi avaliada. Portanto, a *flag* referente à primeira regra de derivação recebe o valor 1 (linha 4) e é verificado se há recursão à esquerda nesta produção (linha 6). Em caso positivo, remove-se a recursão da produção atual.

2. **Caso alguma regra possua recursão indireta:**

```
1  (--> [(((name n0 nt_!_1) 1) ... (nt_0 1) ((name n1 nt_!_1)
      1) ... (nt 0) ord_0 ...)
2   (prd ... (nt_0 ((t ...) ...)) prd_0 ... (nt ((trm t_0 ...
      ) ... (nt_2 t_2 ...) ... (nt_0  t_1 ...) seq_1 ...))
      prd_1 ... )]
3
4   [((n0 1) ... (nt_0 1) (n1 1) ... (nt 0) ord_0 ...)
5    (concat-prds (prd ... (nt_0 ((t ...) ...)) prd_0 ...
       (ord-prd nt ((trm t_0 ... ) ... (nt_2 t_2 ...) ... (t
       ... t_1 ...) ... seq_1 ...)))
6    (prd_1 ...))]
7  (where 1 (check-difference ((n0 1) ... (nt_0 1) (n1 1) ...)
      ((nt_2 t_2 ...) ...))))
```

   Este caso avalia se existe pelo menos um item *flag* na lista *ords* da gramática definida com valor igual a 1 e pelo menos um item *flag* com valor igual a 0 (linha 1). Além

disso, verifica-se se a regra de derivação do primeiro item da lista *ords* com *flag* igual a 0 possui um item de *seq* iniciado com um *nt* que possui o valor de *flag* igual a 1 (linha 2). Isso indica que a produção atual tem, como uma das opções de substituição, uma sequência em que o primeiro termo aparece como um não-terminal a ser derivado em uma produção anterior à atual, sugerindo uma possível recursão indireta.

Na linha 5, a possível recursão à esquerda indireta é removida, mas a *flag* referente a essa regra de derivação é mantida com valor igual a 0 (linha 4). Isso ocorre porque, durante o processo de remoção da recursão indireta, podem surgir mais recursões indiretas, exigindo que a produção passe por esse processo novamente.

3. **Caso alguma regra possua recursão direta:**

```
1 (--> [((nt_0 1) (nt_1 1) ... (nt 0) ord ...)
2  (prd_0 ...(nt ((trm t_0 ... ) ... (nt_2 t_1 ...) ...)) prd
      ...)]
3
4 [((nt_0 1) (nt_1 1) ... (nt 1) ord ...)
5  (concat-prds  (prd_0 ...)
6    (concat-prds
7     (check-left-recursion (nt ((trm t_0 ... ) ... (nt_2 t_1
         ...) ...)) (prd_0 ... prd ...))
8      (prd ...) ))]
9  (where 1 (check-difference ((nt_0 1) (nt_1 1) ...) ((nt_2
      t_1 ...) ...))))
```

Este caso avalia se existe pelo menos um item *flag* na lista *ords* da gramática definida com valor igual a 1 e pelo menos um item *flag* com valor igual a 0 (linha 1). Além disso, verifica-se se a regra de derivação do primeiro item da lista de *ords* com *flag* igual a 0 não possui nenhum item de *seq* iniciado com um *nt* que tenha *flag* igual a 1 (linha 2). Isso indica que a produção atual não possui recursão à esquerda indireta. Após isso, a *flag* referente a essa produção recebe o valor 1 (linha 4) e verifica-se se essa regra de derivação possui recursão à esquerda direta. Se sim, a remoção é feita (linha 7).

O processo de redução finaliza quando nenhuma regra de derivação da gramática definida for compatível com o conjunto de regras de reescrita definidas. A Figura 4.1,

abaixo, mostra um exemplo de como a gramática S, definida na seção anterior, fica após aplicar a relação da redução apresentada.

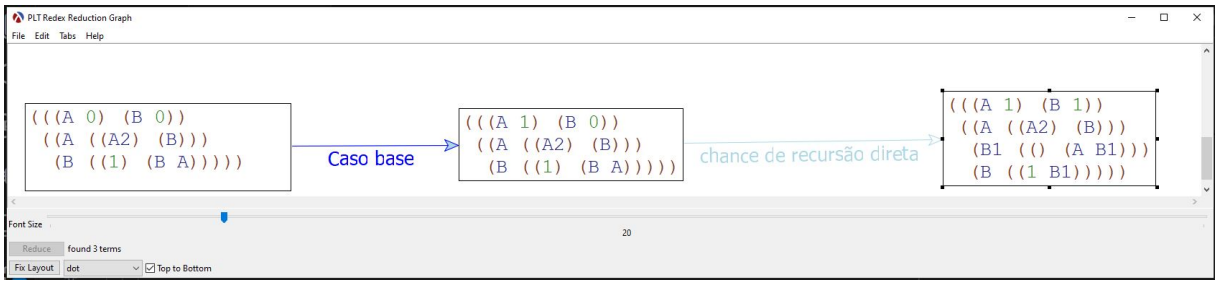

Figura 4.1: Redução do termo da gramática S

4.3 Decisões de implementação

Um dos principais desafios ao criar a relação de redução está na definição dos padrões das regras de reescrita. No caso do algoritmo clássico, era importante conseguir distinguir entre recursão direta e indireta, pois a remoção em cada um desses casos ocorre de maneira distinta. A solução adotada foi ordenar as regras de não-terminais de acordo com a forma de seu lado direito: primeiro listamos regras cujo lado direito inicia com um terminal, seguido de regras com recursão à esquerda direta e por último as demais regras.

Após definir a ordenação do lado direito de regras, foram criadas duas regras de reescrita: uma para identificar recursões diretas e outra para detectar possíveis recursões indiretas. Inicialmente, consideramos adotar uma estratégia de redução não-determinística. A redução não-determinística ocorre quando pelo menos um termo a ser reduzido é compatível com mais de uma regra de reescrita. Ou seja, as produções poderiam ser reduzidas em qualquer ordem, e cada produção poderia ter um padrão que fosse correspondente a mais de uma regra ao mesmo tempo (caso apresentassem recursão à esquerda direta e indireta). No entanto, essa decisão causou problemas que descreveremos a seguir.

O primeiro problema identificado ocorria em gramáticas semelhantes à exempli-

ficada abaixo:

1. $A \rightarrow B \mid 1$
2. $B \rightarrow 2 \mid B3$
3. $A \rightarrow A2 \mid 4$

Neste exemplo, a produção de número 3 se encaixava no padrão de ambas as regras de reescrita. Isso acontecia porque a regra para identificar as possíveis recursões indiretas consistia em verificar se havia alguma produção anterior à atual que tivesse um não-terminal de derivação coincidente com o primeiro elemento de alguma *seq* na produção atual em análise.

A solução para corrigir esse problema foi desenvolver um método que unificava todas as regras de derivação relacionadas a um mesmo não-terminal. Dessa forma, inicialmente, não foi necessário alterar a estratégia anteriormente estabelecida, permitindo uma identificação mais precisa e evitando conflitos entre as regras de reescrita.

Posteriormente, foi encontrado um segundo problema que estava relacionado à possibilidade de aplicar as duas regras de reescrita criadas até então em qualquer ordem em uma das produções da gramática a ser reduzida. Isso poderia levar a um ciclo infinito na relação de redução, como exemplificado a seguir:

1. $A \rightarrow 1 \mid A2$
2. $B \rightarrow 23 \mid A$

Nesse exemplo, um ciclo infinito poderia ocorrer, pois uma das direções seguida seria reduzir a produção de número 2 antes da regra de derivação de número 1. Assim, o *rhs* de B seria: $23 \mid 1 \mid A2$, o que novamente corresponderia à regra de reescrita de possível recursão indireta, resultando em $23 \mid 1 \mid 12 \mid A22$, e assim por diante, em um ciclo sem fim.

Para resolver o problema do ciclo que poderia ocorrer na relação de redução, foi necessário analisar como o algoritmo clássico abordava essa questão. O algoritmo clássico realizava a ordenação dos não-terminais e fazia a remoção da recursão indireta seguindo

tal ordem, e posteriormente realizava remoção da recursão direta.

Além de ser necessário realizar a remoção de todas as recursões à esquerda indiretas antes de remover as recursões diretas, é importante observar a necessidade de realizar a remoção das recursões indiretas conforme a ordem em que aparecem. Caso contrário, pode-se gerar ciclos infinitos, como exemplificado a seguir:

1. $A \rightarrow 1 \mid B2$
2. $B \rightarrow 23 \mid A1$
3. $C \rightarrow 4 \mid A$

Nesse exemplo, um ciclo infinito ocorreria se a produção número 3 passasse pelo processo de remoção de recursão indireta antes das regras de derivação números 1 e 2. O *rhs* de C seria: $4 \mid 1 \mid B2$, o que novamente corresponderia a uma possível recursão indireta, resultando em $4 \mid 1 \mid 232 \mid A12$, e assim por diante, em um ciclo sem fim.

Assim, foi necessário alterar a estratégia utilizada no processo de redução, tornando-a determinística. Cada regra de reescrita só poderia ter seu padrão compatível com uma produção da gramática de cada vez, e as produções precisariam ser analisadas de forma ordenada e priorizando a remoção da recursão indireta. Para isso, foi acrescentada na definição da linguagem uma lista de ordens, *ords*, explicada na seção 4.1, para estabelecer a sequência das produções da gramática.

Com a alteração na definição da linguagem, foi preciso ajustar as regras de reescrita existentes. Inicialmente, foi criada uma regra para identificar o caso base. Esse caso ocorre quando o processo de redução se inicia e nenhuma produção foi analisada. Nesse momento, a primeira produção é marcada como analisada e verifica-se se há recursão à esquerda direta; se sim, é removida. Assim, é possível determinar que a próxima produção a ser reduzida é a seguinte à última marcada como analisada, e quando não houver mais uma próxima produção, o processo de redução é encerrado.

A regra inicialmente criada para identificar recursão à esquerda indireta foi reescrita para validar, além da existência de recursão, se a produção atual é a próxima na sequência. Por outro lado, a regra que identificava a recursão direta foi excluída.

A regra relacionada à identificação de recursão direta foi substituída por um padrão que verificava a ausência de recursão indireta. Essa decisão foi tomada para evitar a necessidade de criar outras regras de reescrita. Se a regra de identificação de recursão direta fosse mantida, ela teria que analisar se havia recursão à esquerda direta e a ausência de recursão indireta para evitar que uma produção fosse compatível com mais de uma regra de reescrita. Além disso, seria necessário criar uma nova regra de substituição para verificar se a produção atual não possui recursão à esquerda. Essa regra seria necessária porque, caso não houvesse, e se na gramática houvesse uma produção com essa característica, a relação de redução poderia ser encerrada antes de analisar toda a gramática, pois a produção não seria compatível com nenhuma regra de reescrita e a redução seria finalizada, deixando possíveis recursões não tratadas.

Diante disso, foi preferível remover a regra que identificava a recursão direta e criar apenas mais uma nova regra para verificar a ausência de recursão indireta. Essa regra pode lidar tanto com o caso de uma produção que possui recursão direta quanto com o caso em que a produção não possui recursão. A análise para determinar se é necessário remover a recursão não é feita no padrão, mas dentro do termo de substituição, utilizando um método que verifica e remove as recursões.

Por fim, todas as regras de redução estão considerando que a gramática esteja livre de símbolos anuláveis antes de ser submetida ao processo de redução. No entanto, os métodos desenvolvidos até o momento não incluem a eliminação desses símbolos. Para uma próxima etapa de desenvolvimento, será implementado um método para remover os símbolos anuláveis. Vale ressaltar que a ausência dessa implementação não compromete a formalização do algoritmo de remoção de recursão à esquerda.

4.4 Testes

Para validar a implementação e identificar possíveis erros de implementação, a remoção de recursão à esquerda foi testada utilizando a biblioteca Rackcheck (POPA, 2021). Para isso, foram desenvolvidos geradores de GLCs com recursão à esquerda e geradores de palavras aceitas por essas gramáticas.

A geração de gramáticas recursivas à esquerda segue o algoritmo 2 a seguir:

Algoritmo 2: Geração de Gramática

Entrada: $rhs - size$ número máximo de alternativas
$seq - size$ tamanho máximo da sequência
T conjunto de terminais
NT conjunto de não-terminais
Saída: Uma gramática G

```
1  início
2      G := ∅;
3      enquanto houver produções a serem geradas faça
4          lhs := escolha símbolo de NT; rhs := ;
5          enquanto tamanho do rhs < rhs − size faça
6              seq := ε;
7              enquanto tamanho do seq < seq − size faça
8                  se Está a produzir o primeiro símbolo então
9                      Decida se haverá recursão direta ou indireta.;
10                     se Optou por recursão direta então
11                         s := lhs;
12                     fim se
13                     senão
14                         s := escolha um dos NT cujo corpo já foi produzido;
15                     fim se
16                 fim se
17                 senão
18                     se Está a produzir o último símbolo desta sequência então
19                         Sorteie s ∈ T;
20                     fim se
21                     senão
22                         Sorteie s ∈ T ∪ NT;
23                     fim se
24                 fim se
25                 seq := Concatena seq s;
26             fim enquanto
27             rhs := Alternativa seq rhs;
28         fim enquanto
29         G := G ∪ Produção lhs rhs;
30     fim enquanto
31     retorne G
32 fim
```

O algoritmo produz uma gramática construindo uma produção para cada não-terminal de um dado conjunto. Ao gerar o primeiro símbolo de cada alternativa, escolhemos entre gerar o próprio não terminal do lado esquerdo da regra corrente (recursão à esquerda direta) ou um dentre os não-terminais previamente gerados. As gramáticas produzidas com esta estratégia sempre apresentam recursão à esquerda direta e indireta.

Por questões de simplicidade de implementação, limitados a geração de GLCs à no máximo 26 não-terminais, correspondendo a cada uma das letras do alfabeto. O símbolo de partida produzido é sempre o não-terminal "S" e os terminais são representados como números inteiros positivos.

A geração de palavras aceitas pela GLC foi realizada com o uso de derivadas sobre linguagens livres de contexto, conforme descrito por Migth *et all* (MIGHT; DARAIS; SPIEWAK, 2011), aliada a definição do conjunto de *first* de uma sentença. Conceitualmente a derivada de uma linguagem L em relação a um símbolo a, denotado por $\partial_a L$, é a linguagem $L' = \{w | aw \in L\}$. Dada uma forma sentencial W qualquer de uma GLC G, uma estrategia simples para obter uma palavra é iterar a derivada sobre uma forma sentencial de G até que se obtenha uma forma sentencial que aceite ε. A sequência de símbolos escolhidos até este ponto forma uma palavra aceita por G.

Para verificar o pertencimento de uma palavra à gramática, foi utilizado o algoritmo CYK, devido à sua capacidade de lidar com GLCs que não sejam nem LL e nem LALR, apesar de sua ineficiência.

Três testes foram realizados para cada gramática gerada. Para os testes, foram geradas 10.000 gramáticas, e para cada gramática foram geradas 100 palavras, com comprimentos variados. Cada teste tem o objetivo de verificar que:

1. Ao aplicar a relação de redução, as produções da gramática não possuem recursão à esquerda direta ou indireta.

2. As sentenças geradas pela gramática com recursão à esquerda são aceitas pela gramática sem recursão à esquerda.

3. As sentenças geradas pela gramática sem recursão à esquerda são aceitas pela gramática com recursão à esquerda.

Todos os testes foram bem-sucedida e foi possível confirmar que a remoção da recursão à esquerda ocorreu conforme o esperado e que os testes não apontaram alteração na linguagem gerada pela gramática. Em adição aos teste foi realizado um verificação de cobertura, produzindo 94.43% de cobertura no algoritmo de eliminação de recursão à esquerda.

5 Trabalhos Relacionados

O primeiro trabalho relacionado é o *Run your research: on the effectiveness of lightweight mechanization* (KLEIN et al., 2012) que apresenta a utilização da mecanização leve com a biblioteca Redex em pesquisas científicas. Os autores validam essa abordagem explorando nove artigos específicos em diversos domínios. Eles destacam os benefícios, como a obtenção de descobertas mais confiáveis e reprodutíveis, e a identificação de problemas relevantes. A mecanização leve é elogiada pela sua facilidade de uso em comparação com abordagens tradicionais, permitindo a automação eficiente da verificação de propriedades. No entanto, existem desafios, como o suporte limitado para construções de ligação em linguagens de objeto e a falta de suporte direto para relações não algorítmicas. Os autores também discutem maneiras de melhorar a eficácia dos geradores de casos de teste aleatórios do Redex.

O trabalho *Decoding Lua: formal semantics for the developer and the semanticist* (SOLDEVILA et al., 2017) apresenta uma semântica formal para um grande subconjunto da linguagem de programação Lua (versão 5.2), com o objetivo de oferecer uma representação precisa da linguagem e facilitar o desenvolvimento, teste e análise de implementações alternativas. O modelo de semântica operacional em passos pequenos é modular e baseado em conceitos da semântica de redução de Felleisen-Hieb. Os autores descrevem as peculiaridades da linguagem Lua e como elas são modeladas, usando a biblioteca Redex para mecanização e teste do modelo. A validação do projeto, comparando-o com a suíte de testes do interpretador de referência da linguagem Lua, foi bem-sucedida. Destacam-se a formalização das principais características da linguagem Lua e a modularidade do modelo, que permite a adição eficiente de novos recursos e extensões. Em suma, o projeto oferece uma semântica formal abrangente e validada para a linguagem Lua, com potencial para aprimorar o desenvolvimento e a compreensão da linguagem.

No trabalho *Property-Based Testing via Proof Reconstruction* (BLANCO; MILLER; MOMIGLIANO, 2019) é introduzida uma abordagem teórico-prova para o teste baseado em propriedades, visando especificações relacionais. Utilizando a estrutura do

certificado de prova fundamental, os autores descrevem várias estratégias de geração de testes, desde aleatória até exaustiva, incluindo redução de contraexemplos. Eles implementaram um protótipo em λProlog, destacando a aplicabilidade em diferentes estratégias de geração de testes e a validação em *benchmarks*. Apesar da eficiência limitada do protótipo e do estágio de desenvolvimento inicial, o trabalho representa uma contribuição significativa para o teste baseado em propriedades e sua aplicação em ambientes reais.

6 Conclusão

Este estudo formalizou o algoritmo clássico de Greibach para remoção de recursão à esquerda em gramáticas livres de contexto, tornando-o mais acessível para os estudantes entenderem os passos envolvidos nesse processo. A implementação prática usando a biblioteca PLT Redex possibilita que seja utilizadas diferentes gramáticas e que seja observado diretamente como o algoritmo afeta a gramática resultante.

Além disso, a ferramenta Traces do PLT Redex proporciona uma visualização dinâmica do processo de redução, facilitando a compreensão dos conceitos abstratos envolvidos. Essa visualização passo a passo ajuda os alunos a acompanhar mais claramente como o algoritmo manipula a gramática e como as transformações ocorrem.

Em última análise, esta pesquisa não só contribui para a compreensão do algoritmo de Greibach para a remoção de recursão à esquerda, mas também pode servir como base para a formalização de outros algoritmos. A metodologia utilizada pode ser aplicada em diferentes contextos, fornecendo uma estrutura para futuras pesquisas e desenvolvimentos nessa área.

Bibliografia

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers, Principles, Techniques, and Tools*. [S.l.]: Addison-Wesley, 1986.

BLANCO, R.; MILLER, D.; MOMIGLIANO, A. Property-based testing via proof reconstruction. In: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. New York, NY, USA: Association for Computing Machinery, 2019. (PPDP '19). ISBN 9781450372497. Disponível em: ⟨https://doi.org/10.1145/3354166.3354170⟩.

CHOMSKY, N. Three models for the description of language. *IRE Transactions on Information Theory*, v. 2, p. 113–124, 1956. Disponível em: ⟨http://www.chomsky.info/articles/195609--.pdf⟩.

FELLEISEN, M.; HIEB, R. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, v. 103, n. 2, p. 235–271, 1992. Disponível em: ⟨https://doi.org/10.1016/0304-3975(92)90014-7⟩.

FLATT, M.; PLT. *Reference: Racket*. [S.l.], 2010. ⟨https://racket-lang.org/tr1/⟩. Disponível em: ⟨https://racket-lang.org/tr1/⟩.

GREIBACH, S. A. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 12, n. 1, p. 42–52, jan 1965. ISSN 0004-5411. Disponível em: ⟨https://doi.org/10.1145/321250.321254⟩.

KLEIN, C.; CLEMENTS, J.; DIMOULAS, C.; EASTLUND, C.; FELLEISEN, M.; FLATT, M.; MCCARTHY, J. A.; RAFKIND, J.; TOBIN-HOCHSTADT, S.; FINDLER, R. B. Run your research: On the effectiveness of lightweight mechanization. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2012. (POPL '12), p. 285–296. ISBN 9781450310833. Disponível em: ⟨https://doi.org/10.1145/2103656.2103691⟩.

MIGHT, M.; DARAIS, D.; SPIEWAK, D. Parsing with derivatives: a functional pearl. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: Association for Computing Machinery, 2011. (ICFP '11), p. 189–195. ISBN 9781450308656. Disponível em: ⟨https://doi.org/10.1145/2034773.2034801⟩.

POPA, B. *Rackcheck: Property Testing*. 2021. ⟨https://docs.racket-lang.org/rackcheck/index.html⟩. Acessado em 12 de maio de 2023. Disponível em: ⟨https://docs.racket-lang.org/rackcheck/index.html⟩.

SIPSER, M. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. ISBN 113318779X.

SOLDEVILA, M.; ZILIANI, B.; SILVESTRE, B.; FRIDLENDER, D.; MASCARENHAS, F. Decoding lua: Formal semantics for the developer and the semanticist. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*. New York, NY, USA: Association for Computing Machinery, 2017. (DLS 2017), p. 75–86. ISBN 9781450355261. Disponível em: ⟨https://doi.org/10.1145/3133841.3133848⟩.

A Apêndice

Este apêndice fornece a documentação sobre a formalização do algoritmo clássico de remoção de recursão à esquerda, desenvolvido em Racket. O objetivo é fornecer informações necessárias para a execução e teste da formalização, bem como acesso ao repositório com o código-fonte.

A.1 Execução e Testes

A.1.1 Geração de Gramáticas Aleatórias

Para configurar a geração de gramáticas aleatórias, edite o arquivo `gerador.rkt`. Os seguintes parâmetros podem ser ajustados:

- `max-trms`: Quantidade máxima de terminais.
- `min-trms`: Quantidade mínima de terminais.
- `max-non-trms`: Quantidade máxima de não-terminais.
- `min-non-trms`: Quantidade mínima de não-terminais.
- `max-rhs`: Quantidade máxima de termos no lado direito das regras.
- `max-seq`: Tamanho máximo de um termo no lado direito das regras.

Esses parâmetros são usados para gerar gramáticas que serão aplicadas nos testes.

A.1.2 Execução dos Testes

Para configurar e executar os testes, abra o arquivo `validador.rkt`. Ajuste os seguintes parâmetros conforme necessário:

- `num-tests`: Número de testes a serem realizados.
- `num-words`: Número de palavras em cada teste.

- `max-wrd-size`: Tamanho máximo de cada palavra.

Também é possível definir o tempo limite para a execução dos testes. O parâmetro `deadline` determina o tempo máximo permitido para a execução dos testes, em milissegundos a partir do momento atual. O valor padrão é de 24 horas. Para ajustar o tempo limite, modifique a seguinte linha do código:

```
(check-property (make-config #num-tests
            #(* (+ (current-inexact-milliseconds) 3600000) 24))
```

Após ajustar os valores conforme necessário, clique em `Run` para iniciar os testes.

A.1.3 Execução do Algoritmo de Remoção de Recursão à Esquerda

Para processar uma gramática específica com o algoritmo, abra o arquivo `classico.rkt` e insira o código abaixo, substituindo a gramática de exemplo pela gramática desejada:

```
; Alterar o valor de input para a gramática desejada
(define input '(
(S ((B 2) (A 4) (2)))
(C ((A) (7 2)))
(B ((S 2) (B 3)))
(A ((C A) (S 2)))
(B ((A) (7 2)))
))

(define orded-prds (ord-rhs (unify-prds input)))
(traces i--> orded-prds)
```

Clique em `Run` para executar o algoritmo.

O código desenvolvido está disponível no repositório em https://github.com/lives-group/Left-Recursion-Elimination-Formalization.