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

Uma Máquina de Parsing Para PEGs com Atributos

Maria Eduarda de Medeiros Simonassi

JUIZ DE FORA
JANEIRO, 2026

Uma Máquina de Parsing Para PEGs com Atributos

Maria Eduarda de Medeiros Simonassi

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Leonardo Vieira do Santos Reis

Coorientador: Elton Máximo Cardoso

JUIZ DE FORA

JANEIRO, 2026

Uma Máquina de Parsing Para PEGs com Atributos

Maria Eduarda de Medeiros Simonassi

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 do Santos Reis
Doutor em Ciência da Computação/UFMG

Elton Máximo Cardoso
Doutor em Ciência da Computação/UFOP

Rodrigo Geraldo Ribeiro
Doutor em Ciência da Computação/UFMG

Gleiph Ghiotto Lima de Menezes
Doutor em Ciência da Computação/UFF

JUIZ DE FORA

16 DE JANEIRO, 2026

Resumo

A análise sintática constitui um elemento fundamental na construção de linguagens e sistemas computacionais, sendo tradicionalmente baseada em modelos como as Gramáticas Livres de Contexto (CFGs) e, mais recentemente, nas *Parsing Expression Grammars* (PEGs). Embora as PEGs ofereçam um modelo determinístico e livre de ambiguidades, apresentam limitações na descrição de construções que dependem de informações contextuais, como aquelas envolvidas no processamento de formatos de arquivos complexos, a exemplo do PNG. As *Parsing Expression Grammars with Syntactic Attributes* (PEGwSA) estendem esse formalismo ao incorporar atributos herdados e sintetizados ao processo de reconhecimento, possibilitando a modelagem de tais dependências contextuais. Uma abordagem para a implementação de PEGs consiste no uso de Máquinas de *Parsing*, como a proposta por Ierusalimschy (2009). Neste trabalho, é proposta e formalizada uma Máquina de *Parsing* para PEGwSA, que estende o modelo original ao introduzir a noção de memória, bem como um conjunto de instruções específicas para a manipulação de atributos, operações aritméticas e processamento de listas. A proposta foi mecanizada por meio da ferramenta PLT Redex, permitindo a verificação da semântica operacional pela execução do modelo.

Palavras-chave: Parsing Expression Grammars (PEG). Atributos Sintáticos. Máquina de Parsing. PLT Redex. Semântica Formal

Abstract

Syntactic analysis is a fundamental component in the construction of programming languages and computational systems, traditionally grounded in models such as Context-Free Grammars (CFGs) and, more recently, Parsing Expression Grammars (PEGs). Although PEGs provide a deterministic and unambiguous formalism, they present limitations in describing constructions that depend on contextual information, such as those involved in the processing of complex file formats, for example, PNG. Parsing Expression Grammars with Syntactic Attributes (PEGwSA) extend this formalism by incorporating inherited and synthesized attributes into the recognition process, enabling the modeling of such contextual dependencies. One approach to implementing PEGs is through Parsing Machines, such as the one proposed by Ierusalimschy (2009). This work proposes and formalizes a Parsing Machine for PEGwSA, extending the original model by introducing the notion of memory, as well as a set of specific instructions for attribute manipulation, arithmetic operations, and list processing. The proposal was mechanized using the PLT Redex tool, allowing the verification of the operational semantics through the execution of the model.

Keywords: Parsing Expression Grammars (PEG). Syntactic Attributes. Parsing Machine. PLT Redex. Formal Semantics

Agradecimentos

À minha família: ao meu pai e à minha mãe, que, sob muito sol, me conduziram até aqui a sombra, por me encorajarem, apoiarem e ampararem em todas as circunstâncias da vida, e por me ensinarem o valor do conhecimento. À minha avó, que jamais me deixou faltar amor e carinho e nunca duvidou de que eu poderia ser aquilo que desejasse. Às minhas irmãs, Mariana e Manuela, por me fazerem irmã mais velha e, assim, contribuírem para que eu me tornasse muito do que sou e do que faço. Ao meu noivo, Lucas, pela presença encorajadora durante toda a minha jornada acadêmica, por acreditar em tudo o que realizei, pelo apoio incondicional e pela parceria na vida.

Aos professores Leonardo Vieira dos Santos Reis e Elton Máximo Cardoso, pela paciência ao longo do desenvolvimento deste trabalho, pela confiança depositada em mim e pelos valiosos conselhos e palavras de incentivo.

Às minhas amigas Carolina Neves e Julia Heloiza Vargas, por participarem da minha construção como pessoa e por estarem sempre ao meu lado, acreditando e me incentivando. Aos meus amigos Luciana Prachedes, Paula Rinco, Patrick Carvalho, Marcos Porto e Vinicius Souza, por tornarem a trajetória acadêmica mais leve e por me mostrarem que a amizade é parte fundamental da vida.

Por fim, mas não menos importante, às mulheres que vieram antes de mim e que tornaram isso tudo possível, por terem lutado pelo nosso direito ao conhecimento e pelo direito de existência.

"Somos mais do que aquilo que nos fizeram."

Toni Morrison, Amada

Conteúdo

Lista de Figuras

Lista de Abreviações

DCC	Departamento de Ciência da Computução
UFJF	Universidade Federal de Juiz de Fora
PEG	Parsing Expression Grammars
PEGwSA	Parsing Expression Grammars with Syntactic Attributes

1 Introdução

A análise sintática é um componente central na construção de linguagens e sistemas computacionais, envolvendo o reconhecimento e a decomposição estrutural de sequências de símbolos. Tradicionalmente, essa área fundamentou-se em sistemas generativos, como as Gramáticas Livres de Contexto (AHO et al., 2008). Contudo, o surgimento do formalismo das *Parsing Expression Grammars* (PEGs) introduziu um modelo baseado no reconhecimento de cadeias, oferecendo uma base formal rigorosa para o *parsing* descendente (*top-down*) e eliminando ambiguidades comuns em gramáticas generativas (FORD, 2004).

Apesar de sua precisão, as PEGs tradicionais apresentam limitações na descrição de construções sintáticas que dependem de informações contextuais, como aquelas encontradas em linguagens extensíveis ou em formatos de arquivos complexos, a exemplo do PNG, nos quais o tamanho dos dados deve ser determinado previamente (REIS; IORIO; BIGONHA, 2014; ZHANG; MORRISETT; TAN, 2023). Para suprir essa lacuna, surgiram as *Parsing Expression Grammars with Syntactic Attributes* (PEGwSA), que estendem o formalismo original ao incorporar atributos herdados e sintetizados, bem como operadores para a manipulação dessas informações durante o processo de análise (REIS et al., 2014; FERREIRA, 2024).

No que se refere à implementação de PEGs, algoritmos como o *Packrat* garantem tempo de execução linear, à custa de um consumo elevado de memória, o que pode torná-los impraticáveis em cenários que envolvem grandes volumes de dados. Como alternativa a essa abordagem, Ierusalimschy (2009) propôs a Máquina de *Parsing*, um modelo de execução baseado em instruções atômicas e no uso de uma pilha, concebido originalmente para PEGs e caracterizado por um uso mais eficiente dos recursos de memória.

O objetivo deste trabalho é propor e formalizar uma Máquina de *Parsing* para PEGwSA. A proposta estende o modelo original da Máquina de *Parsing* para PEGs, incorporando a noção de memória e um conjunto de instruções específicas para o tratamento da lógica associada aos atributos. A máquina proposta é introduzida por meio de

sua formalização e implementação na ferramenta PLT *Redex*, fornecendo uma base teórica para a construção de analisadores sintáticos mais expressivos e eficientes.

Este trabalho está organizado da seguinte forma. No Capítulo 2, são apresentados os conceitos fundamentais relacionados à semântica operacional e à ferramenta PLT *Redex*. O Capítulo 3 apresenta os conceitos fundamentais relacionados à análise sintática, às *Parsing Expression Grammars* (PEGs), às extensões introduzidas pelas PEGwSA e a descrição da Máquina de *Parsing* original para PEGs. No Capítulo 4, é apresentada a proposta de extensão da Máquina de *Parsing* para suportar atributos sintáticos, incluindo sua formalização operacional. O Capítulo 5 detalha a implementação da máquina proposta na ferramenta PLT *Redex*. Por fim, o Capítulo 6 apresenta as conclusões do trabalho e possíveis direções para pesquisas futuras.

2 Semântica Formal

A semântica corresponde ao significado de símbolos e sentenças. Em Ciência da Computação, a semântica formal é entendida como a especificação rigorosa do significado ou do comportamento de programas. O emprego de métodos formais na definição da semântica de linguagens de programação permite evidenciar ambiguidades presentes em especificações informais, além de fornecer uma base sólida para a implementação e para a prova de correção de programas. Ressalta-se que a semântica é atribuída apenas a programas sintaticamente válidos e bem tipados, uma vez que programas inválidos não devem possuir significado. Distinguem-se, de modo geral, três abordagens principais de semântica formal: operacional, denotacional e axiomática.

A semântica denotacional constitui uma abordagem na qual o significado de programas é expresso mediante objetos matemáticos que descrevem abstratamente o comportamento de cada construção da linguagem. A semântica axiomática, por sua vez, configura-se como uma abordagem em que o significado de programas é especificado indiretamente, por meio de axiomas e regras de inferência que relacionam programas a propriedades lógicas sobre seus estados (NIELSON; NIELSON, 2007). Este trabalho usa exclusivamente a semântica operacional, cuja descrição é apresentada na seção seguinte.

2.1 Semântica Operacional

A abordagem operacional concentra-se em definir como as computações são realizadas em uma determinada máquina abstrata e, a partir dessa caracterização, estabelecer o seu significado. Nesse contexto, a semântica operacional facilita a implementação de um interpretador para a linguagem, uma vez que descreve explicitamente os passos de execução. Tradicionalmente, essa abordagem é subdividida em semântica *big-step* e semâmtica *small-step*, que diferem principalmente no nível de detalhamento e na forma de especificar as transições de execução.

A semântica operacional *big-step*, também denominada semântica natural, relaci-

ona diretamente programas ou expressões aos seus resultados finais, abstraindo os passos intermediários de computação. Já a semântica *small-step* descreve o comportamento de programas como uma sequência de passos de execução elementares, em vez de relacionar diretamente cada programa ao seu resultado final.

A linguagem de expressões aritméticas $\mathbb{A}$, originalmente definida em Pierce (2002) e apresentada na Figura 2.1 será utilizada para exemplificar a abordagem operacional.

$$\begin{aligned} e \ ::= & \ \mathbf{zero} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{suc}\ e \\ & \mid \ \mathbf{pred}\ e \mid \mathbf{iszero}\ e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \end{aligned}$$

Figura 2.1: Sintaxe Abstrata de $\mathbb{A}$

A primeira parte (e) declara um conjunto de elementos sintáticos, indicando que a letra e os representa. Cada regra subsequente fornece uma forma sintática alternativa para tais elementos. Em cada posição onde o símbolo e aparece, pode-se substituí-lo por qualquer construção válida. A linguagem $\mathbb{A}$ contém um conjunto de formas sintáticas: as constantes booleanas *true* e *false*, uma expressão condicional, a constante numérica *zero*, os operadores aritméticos *suc* (sucessor) e *pred* (predecessor), e uma operação de teste *iszero*, que retorna *true* quando aplicada a *zero* e *false* quando aplicada a qualquer outro número.

A semântica operacional define como as computações são realizadas a partir de um conjunto de regras que descrevem o funcionamento de uma determinada máquina abstrata. Essas regras são especificadas em um estilo de dedução natural, conforme o modelo apresentado a seguir.

$$\frac{\text{premissas}}{\text{conclusão}}\text{nome}$$

Cada regra é apresentada no formato de uma inferência, composta por um conjunto de premissas, dispostas acima da linha, e uma conclusão, posicionada abaixo dela. A regra expressa que, sempre que todas as premissas forem satisfeitas, a conclusão correspondente pode ser validamente derivada. O nome associado à regra identifica o tipo de inferência realizada e facilita sua referência ao longo da definição semântica.

A semântica *big-step* descreve como os resultados finais são obtidos, relacionando

as construções da linguagem com o valor final ou efeito que produzem. A Figura 2.2 apresenta uma semântica *big-step* para $\mathbb{A}$. O julgamento $e \Downarrow v$ significa que a expressão e avalia para o valor final v.

$$\frac{}{v \Downarrow v} \qquad \frac{e \Downarrow v}{\text{suc } e \Downarrow \text{suc } v} \qquad \frac{e \Downarrow \text{suc } v}{\text{pred } e \Downarrow v}$$

$$\frac{e \Downarrow \text{zero}}{\text{pred } e \Downarrow \text{zero}} \qquad \frac{e \Downarrow \text{zero}}{\text{iszero } e \Downarrow \text{true}} \qquad \frac{e \Downarrow \text{suc } v}{\text{iszero } e \Downarrow \text{false}}$$

$$\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \qquad \frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}$$

Figura 2.2: Semântica *big-step* de $\mathbb{A}$

A Figura 2.3 apresenta uma derivação da semântica *big-step* que demonstra que a expressão `if true then (pred (suc (pred zero))) else zero` é avaliada para `zero`.

$$\frac{\dfrac{}{\text{true} \Downarrow \text{true}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\text{zero} \Downarrow \text{zero}}}{\text{pred zero} \Downarrow \text{zero}}}{\text{suc (pred zero)} \Downarrow \text{suc zero}}}{\text{pred (suc (pred zero))} \Downarrow \text{zero}}}{\text{if true then (pred (suc (pred zero))) else zero} \Downarrow \text{zero}}$$

Figura 2.3: Semântica *big-step* da expressão `if true then pred suc pred zero else zero zero`

A semântica *small-step* descreve como os passos individuais de uma computação são realizados, detalhando de que forma cada construção atinge seu valor ou efeito final. A Figura 2.4 apresenta a semântica *small-step* para a linguagem $\mathbb{A}$.

O julgamento $e \rightarrow e'$ indica que a expressão e é reduzida a e' em um único passo, enquanto $\rightarrow^*$ representa o fecho reflexivo e transitivo da relação $\rightarrow$.

Embora as regras apresentadas na Figura 2.4 sejam escritas na formatação de um sistema de dedução natural, elas geralmente não são utilizadas dessa forma. Em vez disso, parte-se de um termo inicial, que é reescrito passo a passo por meio da aplicação das regras. Uma outra maneira de interpretar esse processo é como uma sequência de transições de estado: inicia-se em um estado representado por um termo inicial e, a cada passo, alcança-se um novo estado no qual o termo foi modificado de acordo com a regra

$$\frac{}{\text{pred (suc } v) \to v}\ \text{E-Pred-Suc} \qquad \frac{}{\text{iszero zero} \to \text{true}}\ \text{E-Iszero-Zero}$$

$$\frac{e \to e'}{\text{iszero } e \to \text{iszero } e'}\ \text{E-Iszero} \qquad \frac{}{\text{iszero (suc } v) \to \text{false}}\ \text{E-Iszero-Suc}$$

$$\frac{}{\text{pred zero} \to \text{zero}}\ \text{E-Pred-Zero} \qquad \frac{e_1 \to e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \to \text{if } e_1' \text{ then } e_2 \text{ else } e_3}\ \text{E-If}$$

$$\frac{e \to e'}{\text{suc } e \to \text{suc } e'}\ \text{E-Suc} \qquad \frac{}{\text{if true then } e_2 \text{ else } e_3 \to e_2}\ \text{E-If-True}$$

$$\frac{e \to e'}{\text{pred } e \to \text{pred } e'}\ \text{E-Pred} \qquad \frac{}{\text{if false then } e_2 \text{ else } e_3 \to e_3}\ \text{E-If-False}$$

Figura 2.4: Semântica *small-step* de $\mathbb{A}$

aplicada. A Figura 2.5 ilustra esse processo ao mostrar como a expressão `if true then (pred (suc (pred zero))) else zero` é reduzida a `zero`.

$$\begin{aligned}
&\to^* \text{ if true then pred (suc (pred zero)) else zero} \\
&\to^* \text{ if true then pred (suc zero) else zero} && \langle \text{pred zero} \to \text{zero} \rangle \\
&\to^* \text{ if true then zero else zero} && \langle \text{pred suc } v \to v \rangle \\
&\to^* \text{ zero}
\end{aligned}$$

Figura 2.5: Semâmtica *small-step* da expressão `if true then (pred (suc (pred zero))) else zero`

É possível definir as semânticas *big-step* e *small-step* para uma mesma linguagem e, posteriormente, demonstrar que ambas são equivalentes no sentido de que todo programa que termina com um valor em uma delas necessariamente termina com o mesmo valor na outra. No entanto, essas abordagens apresentam diferenças conceituais e metodológicas relevantes, cuja discussão detalhada pode ser encontrada em (NIELSON; NIELSON, 2007; PIERCE, 2002).

2.2 PLT Redex

Desenvolvido como uma linguagem de domínio específico executável dentro do ecossistema *Racket*, o PLT *Redex* destina-se à mecanização de modelos semânticos. A ferramenta permite que engenheiros semânticos construam especificações formais que compreendem gramáticas, regras de redução e meta-funções características de semânticas operacionais.

O ambiente oferece diversos recursos para operacionalizar definições semânticas, tais como: mecanismos de depuração passo a passo aplicáveis a semânticas *small-step*, visualizadores de grafos de redução, infraestrutura para criação de testes unitários e capacidades de execução automatizada de testes, dentre outras funcionalidades. A fim de introduzir o PLT *Redex*, a linguagem $\mathbb{A}$ foi especificada na Figura 2.6.

```
1 (define-language A
2   [e ::= true
3          false
4          zero
5          (suc e)
6          (pred e)
7          (iszero e)
8          (if e then e else e)])
```

Figura 2.6: Especificação em PLT *Redex* da linguagem $\mathbb{A}$

A modelagem de linguagens no PLT *Redex* é realizada mediante a função `define-language`, responsável por estabelecer uma gramática livre de contexto que caracteriza a sintaxe da linguagem em questão. A invocação dessa função requer a especificação de dois elementos fundamentais: a identificação da linguagem e a definição dos não-terminais que a constituem. A linguagem, ilustrada na Figura 2.6, é identificada por A e um único não-terminal `e`, cujas formas válidas incluem: valores booleanos (`true` e `false`), números naturais (representados por `zero`, `(suc e)` e `(pred e)`), construções condicionais `(if e then e else e)` e operações relacionais `(iszero e)`.

```
1 (reduction relation A
2     (-->  (if true then e_1 else e_2) e_1 "if-true")
3     (-->  (if false then e_1 else e_2) e_2 "if-false")
4     (-->  (iszero zero) true "=0")
5     (-->  (iszero (suc e)) false "/=0")
6     (-->  (pred zero) zero "pred0")
7     (-->  (pred (suc e)) e "pred-suc"))
```

Figura 2.7: Relação de redução da linguagem $\mathbb{A}$

Com o intuito de definir a semântica da linguagem A, empregamos a função `reduction-relation`, apresentada na Figura 2.7, para descrever o conjunto de regras de reescrita que especificam a evolução dos termos da linguagem. Um termo corresponde a uma expressão sintaticamente válida de A, construída de acordo com sua gramática e representando um estado possível de computação. Em PLT Redex, tais termos são

descritos por padrões da linguagem e manipulados diretamente pelas regras semânticas, sem a necessidade de uma implementação explícita de um avaliador.

Nesse contexto, uma regra de reescrita possui a forma `(-> <termo> <termo reescrito> <nome da regra>)`, indicando que, sempre que um termo correspondente ao padrão `<termo>` for identificado, ele pode ser substituído pelo `<termo reescrito>`, conforme a regra nomeada. Essa substituição modela um único passo de execução da linguagem, permitindo descrever o comportamento computacional de forma declarativa e incremental. O parâmetro facultativo `<nome da regra>` é uma cadeia de caracteres que nomeia a regra de reescrita.

As regras `"if-true"` e `"if-else"` definem, respectivamente, que uma expressão condicional `(if e then` e_1 `else` e_2`)` deve ser reduzida para e_1 se e é `true` e para e_2 se e é `false`. As regras `"=0"` e `"/=0"` determinam, respectivamente, que a operação `(iszero e)` reduz para *true* se `t` é `zero` e para *false* se `t` é `(suc` e_1`)`, ou seja, se `t` é o sucessor de um número natural. Por fim, as regras `"pred0"` e `"pred-suc"` definem que o predecessor de um número natural `(pred e)` reduz para `zero` se `t` é `zero` e para e_1 se `t` é `(suc` e_1`)`.

A Figura 2.8 apresenta uma captura de tela do visualizador de redução de PLT *Redex* do termo `if true then (if false then (iszero (suc zero) else (pred zero)) else (suc (suc zero)))`.

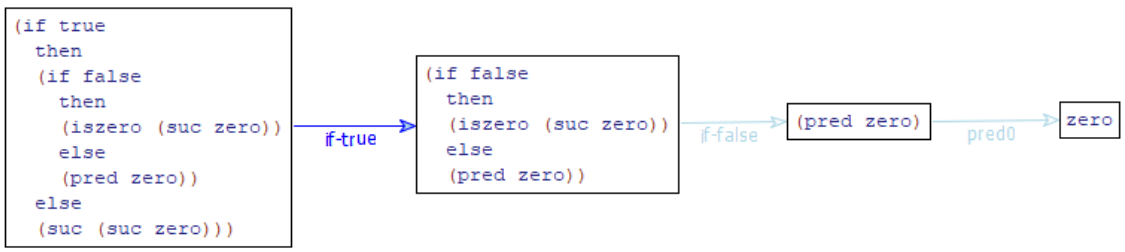

Figura 2.8: Captura de tela do visualizador de redução de *Redex*

3 Análise Sintática

A análise sintática, responsável pelo reconhecimento e pela decomposição estrutural de sequências de símbolos, constitui um componente central na construção de linguagens e sistemas computacionais. Por meio dela, cadeias de símbolos passam a ser interpretadas como estruturas organizadas, viabilizando etapas posteriores do processamento, como a análise semântica e a execução de programas. Historicamente, a teoria e a prática da sintaxe têm sido predominantemente alicerçadas em sistemas generativos, nos quais uma linguagem é formalmente definida por um conjunto de regras aplicadas recursivamente para produzir cadeias válidas, como ocorre nas Gramáticas Livre de Contexto (AHO et al., 2008).

Entretanto, a evolução das linguagens de programação e das técnicas de *parsing* evidenciou limitações práticas desses modelos, especialmente no que diz respeito à ambiguidade e ao controle do processo de reconhecimento sintático. Nesse contexto, surgem abordagens alternativas que deslocam o foco da geração para o reconhecimento de cadeias, oferecendo mecanismos mais diretos e determinísticos para a análise sintática.

3.1 Parsing Expression Grammars

Em contraste com a tradição generativa predominante na teoria formal de linguagens, o formalismo das *Parsing Expression Grammars* (PEGs) propõe um modelo baseado no reconhecimento de cadeias, estabelecendo regras e predicados que decidem se uma *string* pertence ou não a linguagem (FORD, 2004). As PEGs revitalizaram o interesse pelas abordagens de *parsing* descendente (*top-down parsing*), ao oferecer uma fundamentação formal rigorosa para o problema de reconhecimento sintático.

O formalismo das PEGs estabelece regras que determinam se uma sequência de símbolos de entrada pertence à linguagem especificada. Dessa forma, as PEGs atuam como reconhecedoras que validam prefixos da entrada sem necessariamente consumir toda a *string* de entrada, característica fundamental para sua aplicação em análise sintática

descendente. As PEGs são particularmente adequadas para a descrição de linguagens orientadas a máquinas, as quais, por concepção, devem ser precisas e não ambíguas (FORD, 2004).

Formalmente, uma *Parsing Expression Grammar* G é definida como uma quádrupla $G = (V_N, V_T, R, e_S)$ em que:

- V_N é o conjunto finito de símbolos não-terminais;
- V_T é o conjunto finito de símbolos terminais;
- R é a função que mapeia cada não-terminal em V_N para uma *parsing expression*, denotada como $A \leftarrow e$;
- e_S é a expressão de *parsing* inicial

Dessa forma, a sintaxe abstrata das PEGs pode ser definida como na Figura 3.1. Uma expressão de *parsing* e pode ser um terminal (a), uma cadeia vazia (ε), uma sequência ($e \bullet e$), uma escolha ordenada (e/e), uma repetição (e^*), uma negação ($!e$) ou uma chamada de não-terminal (A). A aplicação de uma expressão de *parsing* a uma entrada pode resultar estritamente em sucesso ou falha.

$$e \quad ::= \quad a \mid \varepsilon \mid e \bullet e \mid e/e \mid e* \mid !e \mid A$$

Figura 3.1: Sintaxe abstrata das PEGs.

Já a semântica operacional das PEGs é definida através de uma relação entre expressões de *parsing* e *strings* de entrada. Utilizando a convenção de notação empregada em Daher et al. (2025), $(e, s) \Rightarrow_G (s_p, s_r)$ denota que a expressão e consome o prefixo s_p da entrada s, deixando o sufixo s_r, enquanto $(e, s) \Rightarrow_G \perp$ indica falha no reconhecimento. Uma expressão é considerada bem-sucedida quando não produz $\perp$ como resultado. A Figura 3.2 define a semântica operacional das PEGs.

A regra *Eps* estabelece que a expressão vazia, ε, sempre é bem-sucedida independentemente da entrada s, não consumindo nenhum caractere e deixando a *string* inalterada. A regra $ChrS$ especifica que um terminal a consome com sucesso o primeiro

caractere da entrada quando este corresponde ao símbolo a. Por outro lado, as regras $ChrF$ e $CharNil$ estabelecem que o reconhecimento falha quando o primeiro caractere da entrada não corresponde ao terminal esperado ou quando a entrada está vazia, respectivamente.

A regra Var realiza a análise sintática a partir da expressão associada à variável presente na gramática G. No caso de uma expressão sequencial $e_1 \bullet e_2$, o resultado é construído pela concatenação dos prefixos reconhecidos por e_1 e e_2, enquanto o restante da entrada corresponde ao que permanece após a aplicação de e_2. As regras Cat_{F1} e Cat_{F2} estabelecem que, se qualquer uma das expressões e_1 ou e_2 falhar durante a análise, então toda a expressão sequencial deve ser considerada mal sucedida.

Para operadores de escolha, aplica-se a condição de que a expressão alternativa e_2 somente é avaliada quando e_1 falha, garantindo assim a semântica determinística do operador de escolha em PEGs. Já a análise de uma expressão $e^\star$ consiste na execução repetida de e sobre a cadeia de entrada. Esse processo continua enquanto e obtiver sucesso; quando e eventualmente falha, a expressão $e^\star$ é considerada bem–sucedida, sem consumir nenhum símbolo adicional da entrada.

Por fim, as regras da expressão de predicado de negação $!e$ determinam que, caso e seja bem–sucedida sobre a entrada s, a expressão $!e$ deve falhar. Em contrapartida, quando e falha sobre s, $!e$ é considerada bem–sucedida, também sem consumir qualquer parte da entrada.

Exemplo 3.1.1. Considere a PEG $G_1 = \langle \{a, b\}, \{P\}, R, P \rangle$ e que R possui a regra:

$$P \leftarrow a \bullet P \bullet b / \varepsilon$$

Com essa definição é possível construir a árvore de reconhecimento da *string* ab. O processo começa com a expressão de *parsing* inicial P e a entrada ab. Uma vez que P é um não-terminal, utiliza-se a regra Var para substituí-lo por sua expressão de *parsing* associada, resultando no par $(a \bullet P \bullet b/\varepsilon, ab)$, que representa a expressão de *parsing* corrente e a entrada a ser reconhecida. Atingindo, assim a seguinte configuração:

$$\frac{}{(\varepsilon, s) \Rightarrow_G (\varepsilon, s)}\ Eps \qquad \frac{}{(a, as_r) \Rightarrow_G (a, s_r)}\ ChrS \qquad \frac{a \neq b}{(a, bs_r) \Rightarrow_G \bot}\ ChrF$$

$$\frac{A \leftarrow e \in R \qquad (e, s) \Rightarrow_G r}{(A, s) \Rightarrow_G r}\ Var$$

$$\frac{(e_1, s_{p_1} s_{p_2} s_r) \Rightarrow_G (s_{p_1}, s_{p_2} s_r) \qquad (e_2, s_{p_2} s_r) \Rightarrow_G (s_{p_2}, s_r)}{(e_1 \bullet e_2, s_{p_1} s_{p_2} s_r) \Rightarrow_G (s_{p_1} s_{p_2}, s_r)}\ Cat_{S1}$$

$$\frac{(e_1, s) \Rightarrow_G \bot}{(e_1 \bullet e_2, s) \Rightarrow_G \bot}\ Cat_{F1} \qquad \frac{(e_1, s_{p_1} s_r) \Rightarrow_G (s_{p_1}, s_r) \qquad (e_2, s_r) \Rightarrow_G \bot}{(e_1 \bullet e_2, s_{p_1} s_r) \Rightarrow_G \bot}\ Cat_{F2}$$

$$\frac{(e_1, s_p s_r) \Rightarrow_G (s_p, s_r)}{(e_1 / e_2, s_p s_r) \Rightarrow_G (s_p, s_r)}\ Alt_{S1} \qquad \frac{(e_1, s) \Rightarrow_G \bot \qquad (e_2, s) \Rightarrow_G r}{(e_1 / e_2, s) \Rightarrow_G r}\ Alt_{S2}$$

$$\frac{(e, s_{p_1} s_{p_2} s_r) \Rightarrow_G (s_{p_1}, s_{p_2} s_r) \qquad (e^*, s_{p_2} s_r) \Rightarrow_G (s_{p_2}, s_r)}{(e^*, s_{p_1} s_{p_2} s_r) \Rightarrow_G (s_{p_1} s_{p_2}, s_r)}\ Star_{rec}$$

$$\frac{(e, s) \Rightarrow_G \bot}{(e^*, s) \Rightarrow_G (\varepsilon, s)}\ Star_{end} \qquad \frac{(e, s_p s_r) \Rightarrow_G (s_p, s_r)}{(!e, s_p s_r) \Rightarrow_G \bot}\ Not_F \qquad \frac{(e, s) \Rightarrow_G \bot}{(!e, s) \Rightarrow_G (\varepsilon, s)}\ Not_S$$

$$\frac{}{(a, \varepsilon) \Rightarrow_G \bot}\ ChrNil$$

Figura 3.2: Semântica operacional de expressões de *parsing*.

$$\frac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad (a \bullet P \bullet b/\varepsilon, ab) \Rightarrow_G r_1}{(P, ab) \Rightarrow_G r_1}\ \text{Var}$$

Como a premissa ainda não teve seu resultado r_1 avaliado, a hipótese inicial permanece com resultado indefinido. A expressão de *parsing* $a \bullet P \bullet b/\varepsilon$ constitui uma escolha priorizada e dessa forma, tenta-se primeiramente combinar a entrada com a primeira alternativa, o que resulta na configuração:

$$\frac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{(a \bullet P \bullet b, ab) \Rightarrow_G r_1}{(a \bullet P \bullet b/\varepsilon, ab) \Rightarrow_G r_1}\ \text{Alts}_1}{(P, ab) \Rightarrow_G r_1}\ \text{Var}$$

Dado que $a \bullet P \bullet b$ constitui uma sequência de expressões de *parsing*, o reconhecimento da entrada é realizado de forma segmentada. Como a precedência nas PEGs é à esquerda, a expressão de *parsing* $a \bullet P$ é avaliada primeiro, e a entrada remanescente é então combinada com a expressão de *parsing* b. De modo análogo, a sequência $a \bullet P$ é decomposta nas expressões a e P, obtendo-se sucesso no terminal a, que consome o prefixo "a" da entrada.

$$\frac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{(a, ab) \Rightarrow_G (a, b)}\ \text{Char}_S \qquad (P, b) \Rightarrow_G r_4}{(a \bullet P, ab) \Rightarrow_G r_2}\ \text{Cats}_1 \qquad (b, r_2) \Rightarrow_G r_3}{(a \bullet P \bullet b, ab) \Rightarrow_G r_1}\ \text{Cats}_1}{(a \bullet P \bullet b/\varepsilon, ab) \Rightarrow_G r_1}\ \text{Alts}_1}{(P, ab) \Rightarrow_G r_1}\ \text{Var}$$

Nesse ponto, os resultado r_2, r_3 e r_4 são associados às regras Cat_{S1}, indicando que os valores a serem determinado são independente entre si e em relação a r_1, ainda que r_2 dependa logicamente de r_4 e r_1 dependa de r_2 e r_3. A expressão de *parsing* P é novamente substituida pela sua regra determinada em R, para melhor visualização da árvore de reconhecimento, as primeiras premissas e hipóteses foram ocultadas:

$$\frac{\dfrac{}{(a, ab) \Rightarrow_G (a, b)}\ \text{Char}_S \qquad \dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{(a \bullet P \bullet b, b) \Rightarrow_G r_5}{(a \bullet P \bullet b/\varepsilon, b) \Rightarrow_G r_5}\ \text{AltS}_1}{(P, b) \Rightarrow_G r_4}\ \text{Var}_1}{\vdots}\ \text{Cats}_1$$

A sequência $a \bullet P \bullet b$ é novamente decomposta de forma sucessiva, dessa vez, resultando em falha na avaliação do terminal "a"diante da entrada "b". Essa falha propaga-se

pelas demais sequências, ocasionando o retorno ao ponto correspondente da escolha ordenada:

$$\dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \neq b}{(a,b) \Rightarrow_G \bot}\,\text{CharF}}{(a \bullet P, b) \Rightarrow_G \bot}\,\text{CatF}_1}{(a \bullet P \bullet b, b) \Rightarrow_G \bot}\,\text{CatF}_1}{(a \bullet P \bullet b/\varepsilon, b) \Rightarrow_G \bot}\,\text{AltS}_1}{\vdots}\,\text{Var}_1$$

Contudo, a regra $AltS_1$ pressupõe que a primeira premissa não resulte em falha. Nesse caso, aplica-se a lógica da escolha ordenada: ao obter falha na primeira expressão de *parsing*, a PEG busca sucesso na segunda expressão. Ao adaptar a árvore para substituir a regra $AltS_1$ pela regra $AltS_2$, obtém-se:

$$\dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \neq b}{(a,b) \Rightarrow_G \bot}\,\text{CharF}}{(a \bullet P, b) \Rightarrow_G \bot}\,\text{CatF}_1}{(a \bullet P \bullet b, b) \Rightarrow_G \bot}\,\text{CatF}_1 \qquad (\varepsilon, b) \Rightarrow_G r_5}{(a \bullet P \bullet b/\varepsilon, b) \Rightarrow_G r_5}\,\text{AltS}_2}{\vdots}\,\text{Var}_1$$

O resultado r_5 é obtido mediante a aplicação da regra *Eps*:

$$\dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \neq b}{(a,b) \Rightarrow_G \bot}\,\text{CharF}}{(a \bullet P, b) \Rightarrow_G \bot}\,\text{CatF}_1}{(a \bullet P \bullet b, b) \Rightarrow_G \bot}\,\text{CatF}_1 \qquad \dfrac{}{(\varepsilon, b) \Rightarrow_G (\varepsilon, b)}\,\text{Eps}}{(a \bullet P \bullet b/\varepsilon, b) \Rightarrow_G (\varepsilon, b)}\,\text{AltS}_2}{\vdots}\,\text{Var}_1$$

A árvore de reconhecimento atinge a seguinte configuração:

$$\dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \neq b}{(a,b) \Rightarrow_G \bot}\,\text{CharF}}{(a \bullet P, b) \Rightarrow_G \bot}\,\text{CatF}_1}{(a \bullet P \bullet b, b) \Rightarrow_G \bot}\,\text{CatF}_1 \qquad \dfrac{}{(\varepsilon, b) \Rightarrow_G (\varepsilon, b)}\,\text{Eps}}{(a \bullet P \bullet b/\varepsilon, b) \Rightarrow_G (\varepsilon, b)}\,\text{AltS}_2}{(P, b) \Rightarrow_G r_4}\,\text{Var} \qquad \dfrac{}{(a,ab) \Rightarrow_G (a,b)}\,\text{Char}_S}{(a \bullet P, ab) \Rightarrow_G r_2}\,\text{Cats}_1 \qquad (b, r_2) \Rightarrow_G r_3}{(a \bullet P \bullet b, ab) \Rightarrow_G r_1}\,\text{Cats}_1}{(a \bullet P \bullet b/\varepsilon, ab) \Rightarrow_G r_1}\,\text{Alts}_1}{(P, ab) \Rightarrow_G r_1}\,\text{Var}$$

Os resultados r_2 e r_4 são atualizados:

$$\dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{(a,ab) \Rightarrow_G (a,b)}{(a \bullet P, ab) \Rightarrow_G (a,b)}\,\text{Char}_S \qquad \dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{a \neq b}{(a,b) \Rightarrow_G \bot}\,\text{CharF}}{(a \bullet P, b) \Rightarrow_G \bot}\,\text{CatF}_1}{(a \bullet P \bullet b, b) \Rightarrow_G \bot}\,\text{CatF}_1 \qquad \dfrac{}{(\varepsilon, b) \Rightarrow_G (\varepsilon, b)}\,\text{Eps}}{(a \bullet P \bullet b/\varepsilon, b) \Rightarrow_G (\varepsilon, b)}\,\text{Alt}_{S2}}{(P, b) \Rightarrow_G (\varepsilon, b)}\,\text{Var}}{(a \bullet P, ab) \Rightarrow_G (a,b)}\,\text{Cats}_1 \qquad (b,b) \Rightarrow_G r_3}{(a \bullet P \bullet b, ab) \Rightarrow_G r_1}\,\text{Cats}_1}{(a \bullet P \bullet b/\varepsilon, ab) \Rightarrow_G r_1}\,\text{Alts}_1}{(P, ab) \Rightarrow_G r_1}\,\text{Var}$$

Por fim, aplicando a regra $Char_S$ à entrada "b"com o terminal b, obtém-se a árvore de reconhecimento final que demonstra a geração da cadeia ab pela PEG G_1:

$$\dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{(a,ab) \Rightarrow_G (a,b)}{(a \bullet P, ab) \Rightarrow_G (a,b)}\,\text{Char}_S \qquad \dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{a \neq b}{(a,b) \Rightarrow_G \bot}\,\text{CharF}}{(a \bullet P, b) \Rightarrow_G \bot}\,\text{CatF}_1}{(a \bullet P \bullet b, b) \Rightarrow_G \bot}\,\text{CatF}_1 \qquad \dfrac{}{(\varepsilon, b) \Rightarrow_G (\varepsilon, b)}\,\text{Eps}}{(a \bullet P \bullet b/\varepsilon, b) \Rightarrow_G (\varepsilon, b)}\,\text{Alt}_{S2}}{(P, b) \Rightarrow_G (\varepsilon, b)}\,\text{Cats}_1}{(a \bullet P, ab) \Rightarrow_G (a,b)}\,\text{Cats}_1 \qquad \dfrac{}{(b,b) \Rightarrow_G (b,\varepsilon)}\,\text{Char}_S}{(a \bullet P \bullet b, ab) \Rightarrow_G (ab,\varepsilon)}\,\text{Alts}_1}{(a \bullet P \bullet b/\varepsilon, ab) \Rightarrow_G (ab,\varepsilon)}\,\text{Var}}{(P, ab) \Rightarrow_G (ab,\varepsilon)}$$

À primeira vista, a PEG G_1 parece reconhecer exclusivamente *strings* da linguagem $\{a^n b^n \mid n \geq 0\}$. Contudo, ao analisar a árvore de reconhecimento para a entrada bb, verifica-se que a *string* também é aceita:

$$\dfrac{P \leftarrow a \bullet P \bullet b/\varepsilon \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{a \neq b}{(a,bb) \Rightarrow_G \bot}\,CharF}{\vdots}}{(a \bullet P \bullet b, bb) \Rightarrow_G \bot}\,Cat_{F1} \qquad \dfrac{}{(\varepsilon, bb) \Rightarrow_G (\varepsilon, bb)}\,\text{Eps}}{(a \bullet P \bullet b/\varepsilon, bb) \Rightarrow_G (\varepsilon, bb)}\,\text{Alt}_{S2}}{(P, bb) \Rightarrow_G (\varepsilon, bb)}\,\text{Var}$$

Esse comportamento decorre de uma característica fundamental que distingue as PEGs das Gramáticas Livres de Contexto: nas PEGs, o *parsing* pode alcançar sucesso sem consumir qualquer entrada. Dessa forma, na PEG G_1, em virtude da presença da alternativa ε na escolha ordenada da regra de P, a entrada bb é considerada válida.

Exemplo 3.1.2. Considere a PEG $G_2 = \langle \{c\}, \{S\}, R, S \rangle$ e que R possui a regra:

$$S \leftarrow c^* \bullet c$$

A partir dessa definição, queremos reconhecer a entrada cc. Utilizando a árvore de reconhecimento, o primeiro passo é substituir o não-terminal S pela sua expressão de *parsing* correspondente utilizando a regra *Var*:

$$\frac{S \leftarrow c^* \bullet c \in R \qquad (c^* \bullet c, cc) \Rightarrow_G r_1}{(S, cc) \Rightarrow_G r_1}\ \text{Var}$$

Assim, o passo seguinte consiste na decomposição da sequência $c^* \bullet c$. De acordo com a regra Cat_{S1}, a segunda premissa é definida a partir da entrada remanescente resultante do processamento da primeira expressão de *parsing*. Desse modo, na notação $(\_, r_2)$ empregada, o conteúdo consumido não é relevante, sendo considerado apenas o sufixo restante r_2:

$$\frac{S \leftarrow c^* \bullet c \in R \qquad \dfrac{(c^*, cc) \Rightarrow_G (\_, r_2) \qquad (c, r_2) \Rightarrow_G r_3}{(c^* \bullet c, cc) \Rightarrow_G r_1}\ \text{Cat}_{S1}}{(S, cc) \Rightarrow_G r_1}\ \text{Var}$$

A expressão de *parsing* da repetição é derivada da seguinte forma:

$$\frac{S \leftarrow c^* \bullet c \in R \qquad \dfrac{\dfrac{(c, cc) \Rightarrow_G (\_, r_4) \qquad (c^*, r_4) \Rightarrow_G r_5}{(c^*, cc) \Rightarrow_G (\_, r_2)}\ \text{Star}_{rec} \qquad (c, r_2) \Rightarrow_G r_3}{(c^* \bullet c, cc) \Rightarrow_G r_1}\ \text{Cat}_{S1}}{(S, cc) \Rightarrow_G r_1}\ \text{Var}$$

Aplicando a regra $Char_S$ e substituindo o par resultante na segunda premissa, obtém-se:

$$\frac{S \leftarrow c^* \bullet c \in R \qquad \dfrac{\dfrac{\dfrac{}{(c, cc) \Rightarrow_G (c, c)}\ Char_S \qquad (c^*, c) \Rightarrow_G r_5}{(c^*, cc) \Rightarrow_G (\_, r_2)}\ \text{Star}_{rec} \qquad (c, r_2) \Rightarrow_G r_3}{(c^* \bullet c, cc) \Rightarrow_G r_1}\ \text{Cat}_{S1}}{(S, cc) \Rightarrow_G r_1}\ \text{Var}$$

Utilizando novamente a regra da repetição:

$$\frac{S \leftarrow c^* \bullet c \in R \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{(c, cc) \Rightarrow_G (c, c)}\ Char_S \qquad \dfrac{(c, c) \Rightarrow_G (\_, r_6) \qquad (c^*, r_6) \Rightarrow_G r_7}{(c^*, c) \Rightarrow_G r_5}\ \text{Star}_{rec}}{(c^*, cc) \Rightarrow_G (\_, r_2)}\ \text{Star}_{rec} \qquad (c, r_2) \Rightarrow_G r_3}{(c^* \bullet c, cc) \Rightarrow_G r_1}\ \text{Cat}_{S1}}{(S, cc) \Rightarrow_G r_1}\ \text{Var}$$

Aplicando sucessivamente as regras $Char_S$ e $Star_{end}$, identifica-se a falha da repetição gulosa:

$$\dfrac{S \leftarrow c^* \bullet c \in R \qquad \dfrac{\dfrac{}{(c, cc) \Rightarrow_G (c, c)}\, Char_S \quad \dfrac{\dfrac{}{(c, c) \Rightarrow_G (c, \varepsilon)}\, Char_S \quad \dfrac{(c, \varepsilon) \Rightarrow \bot}{(c^*, \varepsilon) \Rightarrow_G (\varepsilon, \varepsilon)}\, Star_{end}}{(c^*, c) \Rightarrow_G (c, \varepsilon)}\, \text{Star}_S}{(c^*, cc) \Rightarrow_G (cc, \varepsilon)}\, \text{Star}_S \qquad (c, \varepsilon) \Rightarrow_G r_3}{(c^* \bullet c, cc) \Rightarrow_G r_1}\, \text{Cat}_{S1}}{(S, cc) \Rightarrow_G r_1}\, \text{Var}$$

A entrada foi consumida pela repetição gulosa e, ao tentar reconhecer ε com o terminal c, ocorre falha, resultando na árvore final:

$$\dfrac{S \leftarrow c^* \bullet c \in R \qquad \dfrac{\dfrac{}{(c, cc) \Rightarrow_G (c, c)}\, Char_S \quad \dfrac{\dfrac{}{(c, c) \Rightarrow_G (c, \varepsilon)}\, Char_S \quad \dfrac{(c, \varepsilon) \Rightarrow \bot}{(c^*, \varepsilon) \Rightarrow_G (\varepsilon, \varepsilon)}\, Star_{end}}{(c^*, c) \Rightarrow_G (c, \varepsilon)}\, \text{Star}_S}{(c^*, cc) \Rightarrow_G (cc, \varepsilon)}\, \text{Star}_S \qquad \dfrac{}{(c, \varepsilon) \Rightarrow_G \bot}\, \text{CharNil}}{(c^* \bullet c, cc) \Rightarrow_G \bot}\, \text{Cat}_{F2}}{(S, cc) \Rightarrow_G \bot}\, \text{Var}$$

Portanto, a entrada cc não é aceita pela PEG G_2, pois a repetição opera de forma gulosa, consumindo todos os caracteres possíveis, o que resulta na falha da regra S.

3.2 Uma Máquina de Parsing Para PEGs

A implementação das regras definidas pelas PEGs tradicionalmente fundamenta-se em *parsers* descendentes ou no algoritmo de memorização denominado *Packrat*. Embora o *Packrat* ofereça complexidade de tempo linear, ele impõem uma complexidade de espaço também linear, mas associada a uma constante consideravelmente grande. Essa característica torna o *Packrat* pouco adequado ao processamento de grandes volumes de dados, situação comum em ferramentas de *pattern matching*. Além disso, implementações existentes necessitam da entrada completa para funcionar, demandando maior espaço de armazenamento na memória. Essas limitações motivaram o desenvolvimento de abordagens alternativas, como a Máquina de *Parsing* proposta por Ierusalimschy (2009). Na Máquina de *Parsing* cada padrão da PEG é compilado em um programa que é executado dinamicamente. A Máquina apresenta um modelo de execução mais apropriado para contextos que demandam eficiência na utilização de recursos de memória, se mostrando mais apropriado para linguagens dinâmicas como o Lua, visto que os programas são construídos e compostos dinamicamente em tempo de execução, alinhando-se à natureza do formalismo PEG.

O estado de uma Máquina de *Parsing*, que a define formalmente, é composto por três componentes principais:

- **Contador de Programa (pc):** índice que referencia a próxima instrução a ser executada;
- **Posição Atual (i):** registro que mantém a posição corrente na cadeia de entrada;
- **Pilha (e):** estrutura de dados utilizada para armazenar endereços de retorno e entradas de *backtracking*. Um endereço de retorno é um novo valor para o contador de programa, enquanto uma entrada de *backtracking* contém tanto um endereço quanto uma posição na cadeia de entrada.

Portanto, o estado da Máquina é uma tripla $N \times N \times Stack$, contendo a próxima instrução a ser executada (pc), a posição atual na cadeia de entrada (i) e uma pilha (e), ou $Fail\langle e\rangle$, um estado de falha com uma pilha e associada. As pilhas são listas de $N \cup N \times N$, em que uma posição da pilha da forma N representa um endereço de retorno, enquanto uma posição da pilha da forma $N \times N$ representa uma entrada de *backtracking*, com um endereço e uma posição na entrada.

A Máquina de *Parsing* executa programas compostos por instruções atômicas que modificam o estado da máquina. As instruções fundamentais executadas pela Máquina de Parsing são:

- **Char x:** tenta casar o caractere x com a posição atual da entrada. Em caso de sucesso, avança uma posição, consumindo o caractere e caso contrário, falha.
- **Any:** avança uma posição na entrada, consumindo o caractere, se o fim da mesma não tiver sido alcançado; caso contrário, falha.
- **Choice l:** adiciona uma entrada de *backtracking* na pilha. O parâmetro l é o deslocamento para a instrução alternativa.
- **Jump l:** realiza um salto relativo para a instrução localizada no deslocamento l.
- **Call l:** adiciona o endereço da próxima instrução na pilha, como endereço de retorno, e salta para a instrução no deslocamento l.

- **Return:** remove um endereço da pilha e salta para esse endereço.
- **Commit l:** compromete-se com uma escolha, descartando a entrada mais recente da pilha e saltando para a instrução no deslocamento l.
- **Fail:** força uma falha. Em caso de falha, a Máquina desempilha entradas até localizar uma entrada de *backtracking*, utilizada para restaurar o estado da Máquina (posição e endereço).

A Figura 3.3 apresenta a semântica operacional da Máquina de *Parsing* e como os estados da máquina são atualizados através das instruções. A notação adotada em Ierusalimschy (2009) define que o programa P e a entrada $\mathcal{S}$ estão implícitos. A relação $\xrightarrow{\text{Instrução}}$ relaciona dois estados quando a instrução endereçada por pc no estado antecedente corresponde ao rótulo e a condição, se presente, é válida. O fecho transitivo dessa relação constitui uma execução da máquina.

$$
\begin{array}{rcll}
\langle pc, i, e\rangle & \xrightarrow{\text{Char } x} & \langle pc+1, i+1, e\rangle & \mathcal{S}[i] = x \\
\langle pc, i, e\rangle & \xrightarrow{\text{Char } x} & \textbf{Fail}\langle e\rangle & \mathcal{S}[i] \neq x \\
\langle pc, i, e\rangle & \xrightarrow{\text{Any}} & \langle pc+1, i+1, e\rangle & i+1 \leq |\mathcal{S}| \\
\langle pc, i, e\rangle & \xrightarrow{\text{Any}} & \textbf{Fail}\langle e\rangle & i+1 > |\mathcal{S}| \\
\langle pc, i, e\rangle & \xrightarrow{\text{Choice } l} & \langle pc+1, i, (pc+l, i) : e\rangle & \\
\langle pc, i, e\rangle & \xrightarrow{\text{Jump } l} & \langle pc+l, i, e\rangle & \\
\langle pc, i, e\rangle & \xrightarrow{\text{Call } l} & \langle pc+l, i, (pc+1) : e\rangle & \\
\langle pc_0, i, pc_1 : e\rangle & \xrightarrow{\text{Return}} & \langle pc_1, i, e\rangle & \\
\langle pc, i, h : e\rangle & \xrightarrow{\text{Commit } l} & \langle pc+l, i, e\rangle & \\
\langle pc, i, e\rangle & \xrightarrow{\text{Fail}} & \textbf{Fail}\langle e\rangle & \\
\textbf{Fail}\langle pc : e\rangle & \xrightarrow{any} & \textbf{Fail}\langle e\rangle & \\
\textbf{Fail}\langle (pc, i_1) : e\rangle & \xrightarrow{any} & \langle pc, i_1, e\rangle &
\end{array}
$$

Figura 3.3: Semântica Operacional da Máquina de *Parsing*

Exemplo 3.2.1. Considere a PEG $G_1 = \langle \{a, b\}, \{P\}, R, P\rangle$ e que R possui a regra:

$$P \leftarrow a \bullet P \bullet b / \varepsilon$$

Com base nessa definição, constrói-se o programa de instruções para a Máquina de *Parsing* que representa a PEG em questão. A Figura 3.4 ilustra esse programa.

```
1 "P": Call "C1"
2      Halt
3 "C1": Choice "C2"
4       Char ‘a’
5       Call "P"
6       Char ‘b’
7       Commit "End"
8 "C2": Return
9 "End": Return
```

Figura 3.4: Instruções da Máquina de *Parsing* para G_1

Para aprimorar a visualização e a legibilidade, utilizam-se *labels*, isto é, nomes atribuídos a determinadas linhas do código para indicar deslocamentos. O programa inicia pela *label* P, que referencia a expressão de *parsing* inicial. A instrução `Call ‘‘C1’’` desloca o *pc* para a *label* "C_1". Com o desvio do programa para o novo *pc*, a instrução `Choice ‘‘C2’’` adiciona à pilha uma entrada de *backtracking*. Essa instrução indica o início de uma escolha ordenada: caso a primeira expressão de *parsing* resulte em erro, a segunda alternativa é testada. Nesse programa, a instrução que marca o início da segunda alternativa encontra-se na linha endereçada pela *label* "C_2".

Após a adição da entrada de *backtracking*, executa-se a instrução `Char ‘a’`, que tenta reconhecer o caractere *"a"* na posição atual da entrada. Essa instrução foi mapeada a partir do terminal a na expressão de *parsing* $a \bullet P \bullet b$. Na sequência, para representar a invocação do não-terminal P, a instrução subsequente é `Call ‘‘P’’`, indicando o desvio do fluxo para a *label* P. Essa sequência prossegue até que a instrução `Char ‘a’` falhe com a entrada, momento em que o processo de *backtracking* ocorre, desempilhando um endereço e uma posição para o retrocesso.

Ao ocorrer falha na instrução `Char ‘a’`, a Máquina retrocede para a posição de entrada associada ao ponto de *backtracking* previamente empilhado pela instrução `Choice ‘‘C2’’`. Desse modo, efetiva-se a falha da primeira alternativa da escolha ordenada. A segunda alternativa é mapeada para a instrução `Return`, uma vez que, por definição, a cadeia vazia (ε) obtém sucesso independentemente da entrada. Nesse contexto, a instrução `Return` é empregada para devolver o controle ao ponto de chamada. Como a regra `P` é

recursiva, toda a computação retorna sequencialmente para a sua chamada inicial, o `Call ‘‘C1’’` da linha 1. Após esse retorno, o fluxo segue para a próxima instrução, `Halt`, na linha 2 e o programa é finalizado.

Ao finalizar o programa em P, caso a invocação tenha ocorrido na linha 5, isto é, recursivamente durante a execução de outra chamada, o programa prossegue com a instrução `Char ‘b’`, que tenta reconhecer o caractere *“b”* na posição atual da entrada. Obtendo-se sucesso, a instrução `Commit ‘‘End’’` é alcançada, consolidando a escolha mediante o descarte da entrada mais recente da pilha e o desvio para a instrução na *label End*. Assim como na primeira escolha, executa-se a instrução `Return`, devolvendo o controle ao ponto de chamada. E novamente, como a regra `P` é recursiva, toda a computação retorna sequencialmente para a sua chamada inicial, seguindo para a próxima instrução e finalizando o programa.

Exemplo 3.2.2. Considere a PEG $G_2 = \langle \{c\}, \{S\}, R, S \rangle$ e que R possui a regra:

$$S \leftarrow c^* \bullet c$$

Com base nessa definição, constrói-se o programa de instruções para a Máquina de *Parsing* que representa a PEG em questão. A Figura 3.5 ilustra esse programa.

```
1 "S":    Choice "End"
2         Char ‘c’
3         Commit "Cont"
4 "Cont": Jump "S"
5 "End":  Char ‘c’
6         Halt
```

Figura 3.5: Instruções da Máquina de *Parsing* para G_2

Nesse exemplo, a primeira instrução é identificada pela label `‘‘S’’`, a qual inicia o programa com a instrução `Choice ‘‘End’’`. Essa instrução marca o início de um escopo de escolha ordenada, uma vez que empilha um endereço de *backtracking*. Em seguida, o fluxo de execução prossegue para a instrução `Char ‘c’`, responsável por tentar reconhecer o caractere *“c”* na posição atual da entrada. Em caso de sucesso, a execução alcança a instrução `Commit ‘‘Cont’’`, que consolida a escolha realizada ao descartar o ponto de

backtracking mais recente da pilha e desviar o controle para a instrução associada à label `Cont`. Caso a instrução `Char 'c'` falhe, considera-se que a primeira alternativa da escolha ordenada não foi satisfeita, acionando-se o mecanismo de *backtracking*.

Na label `Cont`, ocorre um desvio incondicional para a `''S''`, caracterizando a implementação da repetição gulosa. Quando a escolha ordenada falha em sua primeira alternativa, o ponto de *backtracking* previamente registrado pela instrução `Choice ''End''` é recuperado. Com o fluxo de execução redirecionado para a *label* `''End''`, a instrução `Char 'c'` tenta novamente reconhecer o caractere *"c"* na posição corrente da entrada e, em caso de sucesso, a instrução `Halt` é responsável por encerrar a execução do programa.

3.3 Parsing Expression Grammars with Syntactic Attributes

As *Parsing Expression Grammars with Syntactic Attributes* (PEGwSA) estendem o formalismo tradicional das PEGs ao incorporar atributos e operadores para sua manipulação, permitindo que informações adicionais sejam acopladas ao processo de análise sintática. Nessa abordagem, atributos podem ser associados a não-terminais, possibilitando a transmissão e o armazenamento de informações relevantes durante o *parsing*. Tal mecanismo é especialmente importante para lidar com características de determinadas linguagens que não podem ser expressas apenas por PEGs ou por Gramáticas Livres de Contexto, em razão da limitação desses formalismos em descrever construções sintáticas presentes, por exemplo, em linguagens extensíveis e em certos formatos de arquivo de imagem (REIS; IORIO; BIGONHA, 2014; ZHANG; MORRISETT; TAN, 2023).

Em uma PEGwSA, os atributos estão associados aos não-terminais e são classificados em dois tipos. Os atributos herdados correspondem a valores cujo cálculo depende de informações provenientes de símbolos ancestrais; eles funcionam como parâmetros que configuram o comportamento do não-terminal a partir do ambiente sintático vigente. Complementarmente, os atributos sintetizados são valores produzidos a partir dos atributos de símbolos descendentes, desempenhando o papel de resultados computados pelos não-terminais ao término da análise.

Além disso, a PEGwSA estende a definição padrão das PEGs, introduzindo três novas operações: o *update*, o *bind* e a *constraint*. O *update* é responsável por atualizar o atributo, o *bind* captura a entrada consumida em caso de sucesso e a *constraint* é responsável por testar uma condição.

Utilizando a notação definida em Ferreira (2024), a sintaxe abstrata das PEGwSA pode ser definida como na Figura 3.6. As formalizações servirão de base para o desenvolvimento e a extensão do modelo adotado neste trabalho.

A sintaxe foi dividida em expressões de atributos (e) e expressões de *parsing* (p). As notações $\overline{\overline{x}}$ e $\overline{x}$ são utilizadas para denotar sequências de zero ou mais termos, sendo que $\overline{x}$ requer que a sequência representada esteja limitada por parênteses. Já a notação $x_!$ impõe que o termo encapsulado, x, possua ocorrência única.

$$
\begin{array}{lll}
\tau & ::= & \mathit{Bool} \mid \mathit{Integer} \mid \mathit{String} \mid \langle \tau \rangle \mid [\tau] \mid \overline{\tau} \to \overline{\tau} \\
e & ::= & \textbf{true} \mid \textbf{false} \mid i \mid s \mid \overline{\overline{\langle e / e \rangle}} \mid e : e \mid \textbf{nil} \mid e + e \\
 & & \mid e - e \mid e \times e \mid e \div e \mid e == e \mid e > e \mid e \wedge e \mid e \vee e \\
 & & \mid \neg e \mid \vartheta \mid \textbf{get}\ e\ e \mid \textbf{put}\ e\ e\ e \mid \textbf{head}\ e \mid \textbf{tail}\ e \\
p & ::= & p \bullet p \mid p / p \mid !p \mid p^* \mid N\ \overline{e}\ \overline{\vartheta} \mid \vartheta = p \mid \vartheta \leftarrow e \\
 & & \mid ?e \mid a \mid \varepsilon \\
G & ::= & \overline{N_! / \overline{\vartheta :: \tau}\ \overline{e} \to p}
\end{array}
$$

Figura 3.6: Sintaxe abstrata das PEGwSA.

A primeira regra, τ , retrata os tipos de expressões de atributos. Um tipo (τ) pode retratar booleanos (*Bool*), inteiros (*Integer*), cadeia de caracteres (*String*), mapas ($< \tau >$), listas ($[\tau]$) e, por fim, não-terminais $\overline{\tau} \to \overline{\tau}$.

Expressões de atributos, e, podem ser literais; construtores; operações aritméticas, relacionais e lógicas; atributos (ϑ) e manipuladores de listas (**head** e e **tail** e) e mapas (**get** ee e **put** eee). Um literal pode ser um booleano (**true** ou **false**), um inteiro (i) ou uma cadeia de caracteres (s). A formalização apresentada em Ferreira (2024) define três construtores: um para tratar mapas ($\langle \tau \rangle$), outro para construir listas ($e : e$) e um último para expressar listas vazias (*nil*).

Foram incluídas quatro operações aritméticas: adição ($e + e$), subtração ($e - e$), multiplicação ($e \times e$) e divisão ($e \div e$), duas operações relacionais: igualdade ($e == e$) e

maior que ($e > e$), e três operações lógicas: conjunção ($e \wedge e$), disjunção ($e \vee e$) e negação lógica ($\neg$).

As expressões de *parsing* p seguem a mesma definição das PEGs tradicionais, sendo introduzidas adicionalmente três novas construções para a manipulação dos atributos: o *bind* ($\vartheta = p$), o *update* ($\vartheta \leftarrow e$) e a *constraint* ($?e$).

A última regra, G, define que uma PEGwSA é uma sequência finita de produções. Uma produção, por sua vez, é uma associação entre um não-terminal, N, e uma tripla formada por uma sequência de atributos herdados justapostos pelos seus respectivos tipos ($\overline{\vartheta :: \tau}$), uma sequência de expressões de atributos que serão posteriormente amarradas aos seus atributos sintetizados ($\overline{e}$) e uma expressão de *parsing* (p), que nada mais é que o corpo da regra de produção.

Com a adição da lógica de atributos, é necessário definir dois novos conceitos: valor e ambiente. A Figura 3.7 apresenta a sintaxe de valor e ambiente. Um valor é um elemento de um subconjunto de expressões bem-tipadas que engloba todos os possíveis resultados finais das avaliações de expressões de atributos. Um valor pode ser um booleano (**true** ou **false**), um inteiro (i), uma cadeia de caracteres (s), um mapa ($\overline{\overline{< s_! / v >}}$), uma lista ($v : v$) e, mais especificamente, uma lista vazia (**nil**). É importante ressaltar que um mapa é um valor se, e somente se, ele mapeia de cadeia de caracteres (s) para valores de um tipo arbitrário (v). De maneira semelhante, uma lista pode ser considerada um valor se, e somente se, ambas cabeça (**v**: v) e cauda (v :**v**) também são valores. A notação $x_!$ na definição de valor de mapa $\overline{\overline{< s_! / v >}}$ define que cada chave de um valor de mapa é única.

Um ambiente é uma estrutura de dados que abriga a associação entre atributos (ϑ) e valores (v). São definidas duas operações sobre ambientes: consulta ($\Delta[\![\vartheta]\!]$) e extensão ($\Delta[\![\overline{\overline{\vartheta / v}}]\!]$). A notação ($\Delta[\![\vartheta]\!]$) denota o valor associado ao atributo ϑ no ambiente Δ. A notação ($\Delta[\vartheta_1/v_1\vartheta_2/v_2\vartheta_3/v_3...\vartheta_n/v_n]$ tal que $n \geq 1$) denota a amarração de cada atributo ($\vartheta_1, \vartheta_2, \vartheta_3, ..., \vartheta_n$) ao seu respectivo valor ($v_1, v_2, v_3, ..., v_n$) no ambiente Δ.

$$v ::= \textbf{true} \mid \textbf{false} \mid i \mid s \mid \langle \overline{\overline{s_! / v}} \rangle \mid v : v \mid \textbf{nil}$$

$$\Delta ::= \overline{\vartheta_! / v}$$

Figura 3.7: Sintaxe abstrata de valor.

A semântica operacional das PEGwSA será descrita em uma semântica *big-step*. As regras que constituem a semântica *big-step* foram divididas em duas partições: uma que trata expressões de atributos e outra que, empregando a primeira, trata expressões de *parsing*.

O julgamento da partição que trata expressões de atributos tem a forma $\Delta \vdash e \Rightarrow v$, que deve ser interpretada como: a expressão de atributos e quando avaliada no ambiente Δ produz o valor v. A Figura 3.8 apresenta a semântica *big-step* de literais, construtores e referências a atributos. As expressões de atributos tratadas pelas regras *Booleano*, *Inteiro*, *Cadeia de Caracteres*, *Lista Vazia* e *Mapa Vazio* são valores por si só e, por isso, não possuem premissa.

$$\frac{}{\Delta \vdash b \Rightarrow b}\ Booleano \qquad \frac{}{\Delta \vdash i \Rightarrow i}\ Inteiro \qquad \frac{}{\Delta \vdash s \Rightarrow s}\ Cadeia\ de\ caracteres$$

$$\frac{\Delta[\vartheta] = v}{\Delta \vdash \vartheta \Rightarrow v}\ Atributo \qquad \frac{}{\Delta \vdash \mathbf{nil} \Rightarrow \mathbf{nil}}\ Lista\ Vazia$$

$$\frac{\Delta \vdash e_1 \Rightarrow v_1 \qquad \Delta \vdash e_2 \Rightarrow v_2}{\Delta \vdash e_1 : e_2 \Rightarrow v_1 : v_2}\ Lista \qquad \frac{}{\Delta \vdash \langle\,\rangle \Rightarrow \langle\,\rangle}\ Mapa\ Vazio$$

$$\frac{\Delta \vdash \mathbf{put}\langle e_1/e'_1 \ldots e_{n-1}/e'_{n-1}\rangle e_n e'_n \Rightarrow \langle \overline{\overline{s/v}}^{m} \rangle}{\Delta \vdash \langle e_1/e'_1 \ldots e_{n-1}/e'_{n-1} e_n/e'_n\rangle \Rightarrow \langle \overline{\overline{s/v}}^{m} \rangle}\ Mapa$$

Figura 3.8: Semântica *big-step* de literais, construtores e referências a atributos.

A regra *Atributo* define que uma referência a um atributo é avaliada para o valor ao qual ele (o atributo) está amarrado no ambiente de avaliação. A regra *Lista* só é aplicável a listas compostas por um ou mais elementos. Essa regra define que uma lista é avaliada para um valor lista, composto pelos valores para os quais cabeça e cauda da lista original são avaliadas.

Por fim, a regra *Mapa*, por sua vez, só é aplicável a mapas compostos por ao menos um elemento. Essa regra tira proveito da semântica de **put** para tratar construtores de mapas. Basicamente, o valor produzido pela avaliação de um mapa qualquer é construído através de uma sucessão de inserções de pares chave-valor e_n/e'_n. Se uma mesma chave surgir mais de uma vez em uma mesma instância de construtor de mapa, a regra *Mapa*

define que ela (a chave) deve ser amarrada ao valor que estiver mais à direita, ou seja, ao valor "mais recente".

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i = i_1 \oplus i_2}{\Delta \vdash e_1 \oplus e_2 \Rightarrow i} \text{ Operação Binária}$$

$$\frac{\Delta \vdash e_1 \Rightarrow \textbf{true} \qquad \Delta \vdash e_2 \Rightarrow b}{\Delta \vdash e_1 \wedge e_2 \Rightarrow b} \text{ Conjunção}_1 \qquad \frac{\Delta \vdash e_1 \Rightarrow \textbf{false}}{\Delta \vdash e_1 \wedge e_2 \Rightarrow \textbf{false}} \neg \text{ Conjunção}_1$$

$$\frac{\Delta \vdash e_1 \Rightarrow \textbf{true}}{\Delta \vdash e_1 \vee e_2 \Rightarrow \textbf{true}} \text{ Disjunção}_1 \qquad \frac{\Delta \vdash e_1 \Rightarrow \textbf{false} \qquad \Delta \vdash e_2 \Rightarrow b}{\Delta \vdash e_1 \vee e_2 \Rightarrow b} \neg \text{ Disjunção}_1$$

$$\frac{\Delta \vdash e \Rightarrow \textbf{false}}{\Delta \vdash \neg e \Rightarrow \textbf{true}} \text{ Negação Lógica} \qquad \frac{\Delta \vdash e \Rightarrow \textbf{true}}{\Delta \vdash \neg e \Rightarrow \textbf{false}} \neg \text{ Negação Lógica}$$

$$\frac{\Delta \vdash e_1 \Rightarrow v \qquad \Delta \vdash e_2 \Rightarrow v}{\Delta \vdash e_1 == e_2 \Rightarrow \textbf{true}} \textit{ Igualdade}$$

$$\frac{\Delta \vdash e_1 \Rightarrow v_1 \qquad \Delta \vdash e_2 \Rightarrow v_2 \qquad v_1 \neq v_2}{\Delta \vdash e_1 == e_2 \Rightarrow \textbf{false}} \neg \textit{Igualdade}$$

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i_1 > i_2}{\Delta \vdash e_1 > e_2 \Rightarrow \textbf{true}} \textit{ Maior que}$$

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i_1 \leq i_2}{\Delta \vdash e_1 > e_2 \Rightarrow \textbf{false}} \neg \textit{Maior que}$$

Figura 3.9: Semântica *big-step* de operações aritméticas, lógicas e relacionais.

A Figura 3.9 apresenta a semântica das expressões de atributos. A *Operação Binária* representa as quatro operações lógicas: adição, subtração, multiplicação e divisão. A Figura 3.10 conclui a apresentação da partição das expressões de atributos, apresentando as regras que atuam sobre manipuladores de listas e mapas.

A regra *Head* define a semântica da operação de consulta à cabeça de lista (**head**), enquanto a regra *Tail* define a semântica da operação de consulta à cauda de lista (**tail**). Essas regras implicam que manipuladores de listas só podem ser aplicados a listas compostas por um ou mais elementos. Finalmente, são apresentadas as semânticas big-step dos manipuladores de mapa responsáveis por consulta (**get**) e inserção (**put**) de elementos em mapas.

$$\frac{\Delta \vdash e \Rightarrow v_1 : v_2}{\Delta \vdash \textbf{head}\ e \Rightarrow v_1}\ \mathit{Head} \qquad \frac{\Delta \vdash e \Rightarrow v_1 : v_2}{\Delta \vdash \textbf{tail}\ e \Rightarrow v_2}\ \mathit{Tail}$$

$$\frac{\Delta \vdash e_1 \Rightarrow \langle \overline{\overline{s_! / v}} \rangle \qquad \Delta \vdash e_2 \Rightarrow s' \qquad \langle \overline{\overline{s / v}} \rangle [\![s']\!] = v'}{\Delta \vdash \textbf{get}\ e_1\ e_2 \Rightarrow v'}\ \mathit{Get}$$

$$\frac{\Delta \vdash e_1 \Rightarrow \langle \overline{\overline{s_! / v}} \rangle \qquad \Delta \vdash e_2 \Rightarrow s' \qquad \Delta \vdash e_3 \Rightarrow v'}{\Delta \vdash \textbf{put}\ e_1\ e_2\ e_3 \Rightarrow \langle \overline{\overline{s / v}} \rangle [s' / v']}\ \mathit{Put}$$

Figura 3.10: Semântica *big-step* de manipulações de listas e mapas.

Exemplo 3.3.1. O formato de arquivo de imagem *Portable Network Graphics* (PNG) é uma estrutura de dados organizada da seguinte forma: 8 *bytes* que codificam uma assinatura de formato, a qual corresponde, em ordem fixa, aos valores inteiros 137, 80, 78, 71, 13, 10, 26 e 10; em seguida, 4 *bytes* que representam um número natural n, indicando o tamanho dos dados; depois, 4 *bytes* que codificam o tipo do bloco; na sequência, os dados da imagem, constituídos por n *bytes*; e, por fim, 4 *bytes* correspondentes ao código de verificação CRC.

Nesse contexto, o *parser* precisa interpretar uma estrutura de dados composta por múltiplos blocos, cada um contendo informações específicas e para processar corretamente um arquivo PNG, o *parser* deve identificar e analisar cada bloco na ordem correta, validando sua integridade e extraindo as informações necessárias para reconstruir a imagem sem perdas de qualidade. Dessa forma, pela dependência dos dados, a formalização para um *parser* que interpreta um PNG não pode ser descrita por uma Gramática Livre de Contexto (ZHANG; MORRISETT; TAN, 2023).

A Figura 3.11 apresenta uma formalização simplificada de uma PEGwSA, G_3, que incorpora a antecipação do tamanho dos dados antes dos dados propriamente ditos, conforme a estrutura do PNG:

A regra **Char** caracteriza-se pela ausência de atributos herdados ou sintetizados, tendo sua expressão de *parsing* concluída com sucesso ao reconhecer qualquer caractere. A regra **Digit**, por sua vez, sintetiza o atributo *digit*, correspondente à representação numérica da cadeia que denota um dígito.

A regra **Data**, inicialmente, converte o primeiro caractere da cadeia para sua

$$
\begin{array}{rcl}
\textbf{P}/\ ()\ () & \leftarrow & \textbf{Data}\ ()\ () \\
\textbf{Data}/\ ()\ () & \leftarrow & \mathit{Digit}\ ()\ (\mathit{digit})\ \bullet\ (\ (?\ \mathit{digit} > 0)\ \textbf{Char}\ (\mathit{digit} = \mathit{digit} - 1))* (?\mathit{digit} == 0)\ !. \\
\textbf{Char}/\ ()\ () & \leftarrow & . \\
\textbf{Digit}/\ ()\ (\mathit{digit}) & \leftarrow & \text{`0'}\ (\mathit{digit} = 0) \\
 & / & \text{`1'}\ (\mathit{digit} = 1) \\
 & / & \text{`2'}\ (\mathit{digit} = 2) \\
 & / & \text{`3'}\ (\mathit{digit} = 3) \\
 & / & \text{`4'}\ (\mathit{digit} = 4) \\
 & / & \text{`5'}\ (\mathit{digit} = 5) \\
 & / & \text{`6'}\ (\mathit{digit} = 6) \\
 & / & \text{`7'}\ (\mathit{digit} = 7) \\
 & / & \text{`8'}\ (\mathit{digit} = 8) \\
 & / & \text{`9'}\ (\mathit{digit} = 9)
\end{array}
$$

Figura 3.11: PEGwSA G_3

forma numérica e, em seguida, por meio de uma *constraint*, verifica se o atributo *digit*, sintetizado pelo não-terminal **Digit**, é maior que zero. Em caso afirmativo, a regra **Char** é invocada; caso contrário, ocorre falha. Posteriormente, em caso de sucesso, a regra **Data** atualiza o valor do atributo *digit*, decrementando-o em uma unidade. O conjunto de operações responsável pelo consumo dos caracteres é reiterado em **Data** em decorrência do uso do operador *. Por fim, emprega-se uma nova *constraint* com o objetivo de validar o valor do atributo *digit*. Caso esse valor seja diferente de 0, conclui-se que a regra **Char** não conseguiu consumir a quantidade esperada de caracteres, caracterizando uma falha na correspondência. Por outro lado, quando *digit* é igual a 0, garante-se que exatamente *digit* caracteres foram consumidos com sucesso. A aplicação da negação do consumo de caracteres assegura, adicionalmente, que, após esse consumo, nenhum outro símbolo da cadeia de entrada seja aceito, mesmo que ainda existam símbolos remanescentes, preservando assim a correção do reconhecimento.

Já a regra **P** define a expressão de *parsing* inicial, responsável por iniciar o processo de leitura dos dados.

Para a cadeia '3abc', o processo de *parsing* tem início a partir da expressão inicial **P**, considerando a entrada em sua totalidade. A regra **P** invoca a regra **Data**, cujo primeiro comando consiste na chamada do não-terminal **Digit**, responsável por associar o valor numérico 3 ao atributo *digit* por meio da leitura do caractere '3'. Esse passo sinaliza ao *parser* que são esperados três caracteres subsequentes na entrada.

Em seguida, a regra **Data** passa a controlar a leitura dos três caracteres seguintes por meio da repetição definida pelo operador *. A cada iteração, o valor do atributo *digit*

é comparado com zero; caso seja maior que zero, a derivação prossegue com a invocação do não-terminal **Char**, consumindo um novo caractere da entrada. Após essa operação, o valor de *digit* é decrementado.

A regra **Data** é iniciada com $n = 3$ e, à medida que os caracteres da entrada são processados pela regra **Char**, o valor de *digit* é sucessivamente reduzido, até que o *parser* tenha consumido exatamente três caracteres após o dígito inicial, tendo essa corretude validada pela *constraint* final.

4 Máquina de *Parsing* para PEGwSA

A adoção da Máquina de *Parsing* como estratégia para a implementação de analisadores sintáticos apresenta vantagens significativas, como a simplicidade na definição e manutenção do analisador, além da eficiência no processamento das entradas. Neste contexto, propõe-se a especificação de uma Máquina de *Parsing* direcionada à PEGwSA, articulada a partir de uma fundamentação formal precisa e passível de reprodução.

4.1 Semântica da Máquina

A Máquina de *Parsing* para PEGwSA constitui uma extensão da Máquina de *Parsing* para PEGs proposta por Ierusalimschy (2009). Formalmente, sua definição fundamenta-se na caracterização de seu estado, o qual amplia a definição original a fim de possibilitar a manipulação de atributos por meio da incorporação de duas novas componentes: a memória e um registrador que aponta para o topo da pilha.

A memória é representada como uma lista de valores, os quais podem assumir os tipos número natural, booleano ou lista. A arquitetura proposta incorpora esse componente de memória para viabilizar o acesso, a atualização e a recuperação dos valores dos atributos, uma vez que o controle e a manipulação desses valores exclusivamente por meio da pilha se mostram inviáveis, dada a natureza sequencial e restritiva desse tipo de estrutura de dados.

O registrador *sp* guarda um número que aponta para o topo da pilha associada à chamada de função corrente, desempenhando papel análogo ao de um *stack pointer* em arquiteturas de baixo nível, indicando a área de memória atual. Esse mecanismo permite delimitar o contexto ativo de execução, facilitando o acesso aos dados locais e o controle do fluxo durante chamadas e retornos de não-terminais. Tal organização é amplamente adotada em linguagens de montagem e arquiteturas clássicas de processadores, nas quais o uso explícito de registradores para gerenciamento da pilha é fundamental para a implementação eficiente de chamadas de procedimento e escopo de variáveis (PATTERSON;

HENNESSY, 2017; AHO et al., 2008).

Assim, o estado de uma Máquina de *Parsing* é representado por uma quíntupla $\langle pc, i, e, sp, M \rangle$, na qual pc denota o contador de programa, i indica o índice da posição corrente na cadeia de entrada, e representa a pilha, sp corresponde ao índice da região de memória ativa e M designa a memória.

A Máquina de *Parsing* para PEGwSA formalizada neste trabalho incorpora um conjunto de operações e mecanismos voltados à manipulação de atributos. A sintaxe abstrata de uma PEGwSA tradicional é apresentada formalmente na Figura 3.6. Por sua vez, a sintaxe abstrata da PEGwSA simplificada, adotada como base para o desenvolvimento deste trabalho, encontra-se descrita na Figura 4.1, na qual se destacam, em vermelho, as operações não contempladas pela PEGwSA atualmente formalizada, a saber: a estrutura de mapas e suas respectivas operações, *get* e *put*, bem como a operação de *bind*.

$$
\begin{array}{lll}
\tau & ::= & \mathit{Bool} \mid \mathit{Integer} \mid \mathit{String} \mid \langle \tau \rangle \mid [\tau] \mid \overline{\tau} \to \overline{\tau} \\
e & ::= & \mathbf{true} \mid \mathbf{false} \mid i \mid s \mid \overline{\overline{\langle e / e \rangle}} \mid e : e \mid \mathbf{nil} \mid e + e \\
 & & \mid e - e \mid e \times e \mid e \div e \mid e == e \mid e > e \mid e \wedge e \mid e \vee e \mid e + +e \\
 & & \mid \neg e \mid \vartheta \mid \mathbf{get}\ e\ e \mid \mathbf{put}\ e\ e\ e \mid \mathbf{head}\ e \mid \mathbf{tail}\ e \\
p & ::= & p \bullet p \mid p / p \mid !p \mid p^* \mid N\ \overline{e}\ \overline{\vartheta} \mid \vartheta = p \mid \vartheta \leftarrow e \\
 & & \mid ?e \mid a \mid \varepsilon \\
G & ::= & \overline{N_! / \overline{\vartheta :: \tau}\ \overline{e} \to p}
\end{array}
$$

Figura 4.1: Sintaxe abstrata das PEGwSA.

Além das instruções herdadas da definição da máquina proposta por Ierusalimschy (2009), a Máquina de *Parsing* para PEGwSA introduz novas instruções. A Figura 4.2 apresenta a semântica operacional da Máquina de *Parsing* para PEGwSA referente às instruções *Char*, *Any*, *Jump*, *Commit* e *Fail*. Foi mantida a notação adotada por Ierusalimschy (2009), na qual o programa P e a entrada S são considerados implícitos. Essas instruções preservam a semântica originalmente definida para a Máquina de *Parsing* de PEGs descrita em Ierusalimschy (2009). No entanto, os estados inicial e final passam a incorporar as novas componentes M e sp, as quais permanecem inalteradas durante a execução dessas instruções. Dessa forma, a introdução dessas componentes não implica alterações na semântica original.

Estado Inicial	**Instrução**	**Estado final**	**Condição**
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{\text{Char } x}$	$\langle pc+1, i+1, e, sp, M\rangle$	$S[i] = x$
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{\text{Char } x}$	$\text{Fail}\langle e, sp, M\rangle$	$S[i] \neq x$
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{\text{Any}}$	$\langle pc+1, i+1, e, sp, M\rangle$	$i+1 \leq \lvert S\rvert$
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{\text{Any}}$	$\text{Fail}\langle e, sp, M\rangle$	$i+1 \geq \lvert S\rvert$
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{\text{Jump } l}$	$\langle pc+l, i, e, sp, M\rangle$	
$\langle pc, i, h : e, sp, M\rangle$	$\xrightarrow{\text{Commit } l}$	$\langle pc+l, i, e, sp, M\rangle$	
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{\text{Fail}}$	$\text{Fail}\langle e, sp, M\rangle$	
$\text{Fail}\langle (pc : e), sp, M\rangle$	$\xrightarrow{any}$	$\text{Fail}\langle e, sp, M\rangle$	

Figura 4.2: Semântica Operacional da Máquina de *Parsing* para PEGwSA 1

A instrução *Call*, representada na Figura 4.3, é responsável por redirecionar o fluxo de execução para outro ponto do programa, de maneira análoga a uma chamada de função. Para isso, o contador de programa *pc* é incrementado pelo valor *integer*, indicando o deslocamento relativo para a nova posição. Simultaneamente, o *sp* é atualizado de forma a iniciar um novo escopo de memória, passando a apontar para a posição correspondente ao tamanho corrente da memória, de modo a preservar os dados previamente armazenados. Além disso, à pilha são adicionados, como na máquina tradicional, o valor de *pc* acrescido de uma unidade, indicando o ponto de retorno da execução, e o valor atual de *sp*, que permite restaurar o escopo de memória apropriado quando a chamada é finalizada.

Estado Inicial	**Instrução**	**Estado final**
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{\text{Call } l}$	$\langle pc+l, i, (\langle pc+1, i\rangle : sp : e), M.lenght, M\rangle$
$\langle pc_0, i, (pc_1 : e_1 : ... : e_n : sp' : e), sp, M\rangle$	$\xrightarrow{\text{Return } n}$	$\langle pc_1, i, (e_1 : .. : e_n), sp', M\rangle$

Figura 4.3: Semântica Operacional da Máquina de *Parsing* para PEGwSA 2

A instrução *Return*, ilustrada na Figura 4.3, modela o retorno de uma chamada ao encerrar o escopo corrente de execução e restaurar o contexto previamente ativo. Diferentemente da instrução homônima definida em Ierusalimschy (2009), essa operação é parametrizada por um valor natural, o qual indica a quantidade de elementos da pilha que devem ser retornados à chamada atual, indicando os atributos sintetizados. O valor n corresponde aos n primeiros elementos da pilha, isto é, aos n parâmetros de retorno da função invocada pela instrução *Call l*. A execução da instrução *Return* promove, assim, a restauração do contexto de execução previamente salvo. Para tanto, a pilha é particio-

nada, em função de n, de modo a recuperar o valor do contador de programa armazenado no momento da chamada, que determina o ponto para o qual a execução deve retornar, bem como o valor anterior do *sp*, responsável por delimitar o escopo de memória do chamador. O fluxo de execução é então redirecionado para o ponto imediatamente posterior à chamada por meio da atualização do contador de programa, cujo deslocamento é calculado relativamente à posição corrente, garantindo a continuidade adequada da execução. Ao longo desse processo, a cadeia de entrada e o conteúdo da memória M permanecem inalterados, uma vez que a instrução de retorno não interfere diretamente na leitura da entrada nem na manipulação de atributos consolidados fora do escopo local.

A instrução *Choice* l, definida na Figura 4.4, é responsável por delimitar o escopo de execução de uma escolha ordenada. Essa instrução empilha uma quádrupla composta pelo valor do contador de programa *pc* acrescido de l, que indica o ponto para o qual o fluxo de execução deve retornar em caso de *backtracking*, bem como pelo valor do índice de entrada i, necessário para a restauração da posição de leitura. Adicionalmente, os valores de *sp* e da memória M também são empilhados, possibilitando a recuperação do escopo correto de memória quando ocorre o *backtracking*.

Estado Inicial	**Instrução**	**Estado final**
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{\text{Choice}\, l}$	$\langle pc+1, i, (\langle pc+l, i, sp, M\rangle : e), sp, M\rangle$

Figura 4.4: Semântica Operacional da Máquina de *Parsing* para PEGwSA 3

A Figura 4.5 apresenta a instrução *Load* que transfere para o topo da pilha o valor armazenado na posição $sp + l$. Dessa forma, o acesso é realizado no contexto do escopo atual de execução, exigindo que a posição calculada seja um índice válido no domínio de M; caso contrário, a execução falha. De maneira análoga, a instrução *Store* remove o valor do topo da pilha e o armazena na posição $sp + l$ da memória, assegurando que a escrita ocorra no escopo corrente. As instruções *Pop* e *Push* retiram e empilham, respectivamente, um valor no topo da pilha, não alterando outros parâmetros.

Para manipular atributos do tipo número natural, definem-se as instruções aritméticas *Add*, *Sub*, *Mult* e *Div*, descritas na Figura 4.6. Cada uma remove os dois valores superiores da pilha (n_1 e n_2, respectivamente), aplica a operação correspondente e empilha o resultado, falhando caso os operandos não sejam inteiros. Para *Div*, exige-se adicional-

Estado Inicial	Instrução	Estado final	Condição
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{Load\ l}$	$\langle pc+1, i, (M[sp+l] : e), sp, M\rangle$	$sp + l \in \mathrm{dom}(M)$
$\langle pc, i, (value : e), sp, M\rangle$	$\xrightarrow{Store\ l}$	$\langle pc+1, i, e, sp, [sp+l = value]M\rangle$	
$\langle pc, i, (v : e), sp, M\rangle$	$\xrightarrow{Pop}$	$\langle pc+1, i, e, sp, M\rangle$	
$\langle pc, i, e, sp, M\rangle$	$\xrightarrow{Push\ v}$	$\langle pc+1, i, (v : e), sp, M\rangle$	

Figura 4.5: Semântica Operacional da Máquina de *Parsing* para PEGwSA 4

mente que $n_1 \neq 0$. As instruções *Eq* e *Lt* retiram igualmente os dois valores superiores, empilhando um booleano que indica, respectivamente, se $value_1 = value_2$ ou se $n_1 < n_2$, falhando quando os operandos não são inteiros em *Lt*.

Estado Inicial	Instrução	Estado final	Condição
$\langle pc, i, (n_1 : n_2 : e), sp, M\rangle$	$\xrightarrow{Add}$	$\langle pc+1, i, (n_1 + n_2 : e), sp, M\rangle$	$n \in \mathbb{Z}$
$\langle pc, i, (n_1 : n_2 : e), sp, M\rangle$	$\xrightarrow{Sub}$	$\langle pc+1, i, (n_2 - n_1 : e), sp, M\rangle$	$n \in \mathbb{Z}$
$\langle pc, i, (n_1 : n_2 : e), sp, M\rangle$	$\xrightarrow{Mult}$	$\langle pc+1, i, (n_2 * n_1 : e), sp, M\rangle$	$n \in \mathbb{Z}$
$\langle pc, i, (n_1 : n_2 : e), sp, M\rangle$	$\xrightarrow{Div}$	$\langle pc+1, i, (n_2/n_1 : e), sp, M\rangle$	$n \in \mathbb{Z}$ e $n_1 \neq 0$
$\langle pc, i, (value_1 : value_2 : e), sp, M\rangle$	$\xrightarrow{\mathrm{Eq}}$	$\langle pc+1, i, (value_1 == value_2 : e), sp, M\rangle$	
$\langle pc, i, (n_1 : n_2 : e), sp, M\rangle$	$\xrightarrow{\mathrm{Lt}}$	$\langle pc+1, i, (n_1 > n_2 : e), sp, M\rangle$	n inteiro

Figura 4.6: Semântica Operacional da Máquina de *Parsing* para PEGwSA 5

Para atributos booleanos, definem-se as instruções lógicas *And*, *Or* e *Not*, ilustradas na Figura 4.7. As instruções *And* e *Or* retiram os dois valores superiores da pilha (b_1 e b_2), aplicam a operação lógica correspondente e empilham o resultado booleano, falhando caso os operandos não sejam booleanos. A instrução *Not* retira o valor superior (b), aplica a negação lógica e empilha o resultado, falhando caso não seja booleano. A instrução *Assert* verifica se o valor no topo da pilha é o booleano verdadeiro, prosseguindo com sucesso nessa condição, no caso contrário, ocorre uma falha.

Estado Inicial	Instrução	Estado final	Condição
$\langle pc, i, (b_1 : b_2 : e), sp, M\rangle$	$\xrightarrow{And}$	$\langle pc+1, i, (b_1 \& b_2 : e), sp, M\rangle$	b_1 *e* b_2 booleanos
$\langle pc, i, (b_1 : b_2 : e), sp, M\rangle$	$\xrightarrow{Or}$	$\langle pc+1, i, (b_1 \| b_2 : e), sp, M\rangle$	b_1 *e* b_2 booleanos
$\langle pc, i, (b : e), sp, M\rangle$	$\xrightarrow{Not}$	$\langle pc+1, i, (!b : e), sp, M\rangle$	b é booleano
$\langle pc, i, (\#t : e), sp, M\rangle$	$\xrightarrow{\mathrm{Assert}}$	$\langle pc+1, i, e, sp, M\rangle$	b booleano
$\langle pc, i, (\#f : e), sp, M\rangle$	$\xrightarrow{\mathrm{Assert}}$	$Fail\langle e, sp, M\rangle$	

Figura 4.7: Semântica Operacional da Máquina de *Parsing* para PEGwSA 6

As instruções de manipulação de listas da máquina operam diretamente sobre a pilha e está descritas em 4.8. A instrução *Concat* consome as duas listas do topo da pilha, concatena-as e empilha a lista resultante, falhando caso algum dos operandos não

seja uma lista. As instruções *Head* e *Tail* extraem, respectivamente, a cabeça e a cauda da lista no topo da pilha, empilhando o resultado e falhando quando o valor no topo não é uma lista. Por fim, a instrução *Cons* toma o valor no topo da pilha e o insere na cabeça da lista presente na posição imediatamente inferior, empilhando a nova lista construída e falhando se esse segundo elemento não for uma lista.

Estado Inicial	**Instrução**	**Estado final**	**Condição**
$\langle pc, i, (l_1 : l_2 : e), sp, M\rangle$	$\xrightarrow{Concat}$	$\langle pc+1, i, (l_1 ++ l_2 : e), sp, M\rangle$	$l_1 e\ l_2$ listas
$\langle pc, i, ((x : xs) : e), sp, M\rangle$	$\xrightarrow{Head}$	$\langle pc+1, i, (x : e), sp, M\rangle$	
$\langle pc, i, ((x : xs) : e), sp, M\rangle$	$\xrightarrow{Tail}$	$\langle pc+1, i, (xs : e), sp, M\rangle$	
$\langle pc, i, (value : l : e), sp, M\rangle$	$\xrightarrow{Cons}$	$\langle pc+1, i, ((value : l) : e), sp, M\rangle$	l lista

Figura 4.8: Semântica Operacional da Máquina de *Parsing* para PEGwSA 7

A Figura 4.9 apresenta a semântica da instrução no caso em que o estado inicial corresponde a um *Fail* com uma quádrupla no topo da pilha. Essa quádrupla representa um ponto de *backtracking* a ser restaurado, especificando os valores de *pc* e *i* para o novo estado, bem como restabelecendo o valor de *sp* e o conteúdo da memória, de modo a recuperar o escopo de execução apropriado.

Estado Inicial	**Instrução**	**Estado final**	**Condição**
$Fail\langle((pc, i, spr_1, M_1) : e), sp, M\rangle$	$\xrightarrow{any}$	$\langle pc, i, e, sp_1, M_1\rangle$	

Figura 4.9: Semântica Operacional da Máquina de *Parsing* para PEGwSA 8

4.2 Compilando para PEGwSA

Nesta seção, com fins ilustrativos, apresenta-se o processo de tradução de alguns padrões em programas para a Máquina de *Parsing*.

4.2.1 Definição

Uma definição genérica $A(v_1, ..., v_n)(w_1, ..., w_m) \leftarrow \langle e_a\rangle$ de um não-terminal é traduzida para o programa definido em 4.10.

Na definição de um não-terminal A, o procedimento inicial consiste no armazenamento em memória dos atributos $v_1, ..., v_n$, caracterizados como atributos herdados, isto

```
1   Store <Addr v_1>
2   ...
3   Store <Addr v_n>
4   <e_a>
5   <w_1>
6   ...
7   <w_m>
8   Return m
```

Figura 4.10: Código para definição

é, valores recebidos que serão empregados durante a computação da expressão de *parsing* do não-terminal. A instrução `Store` executa esta etapa, inserindo o valor do atributo na posição específica da memória. Posteriormente ao armazenamento desses valores, as instruções relativas à computação da expressão de *parsing* e_a são seguidas pelas instruções responsáveis pela sintetização e inserção na pilha dos atributos sintetizados $w_1, ..., w_m$. A última instrução do programa referente à definição constitui-se da instrução `Return m`, encarregada de restaurar o programa ao fluxo original subsequente à chamada do não-terminal. O valor natural `m` que acompanha a instrução indica a quantidade de retornos sintetizados pelo não-terminal que se encontram armazenados na pilha.

4.2.2 Constraint

Uma *constraint* ($?e$) é traduzida para o programa definido em 4.11.

```
1   <e>
2   Assert
```

Figura 4.11: Código para *constraint*

Após a computação das instruções referente a expressão de *parsing* e, a instrução `Assert` verifica se o topo da pilha corresponde ao valor booleano `true`. Caso a verificação seja positiva, a execução prossegue para a instrução subsequente; caso contrário, ocorre falha. Este comportamento modela precisamente o caráter de condição imposto pela operação *constraint*.

4.2.3 Update

Um *update* ($v_i = \langle e \rangle$) é traduzida para o programa definido em 4.12.

```
1    <e>
2    Store <Addr v_i>
```

Figura 4.12: Código para *update*

Posteriormente à computação das instruções referentes à expressão de *parsing* e, a instrução `Store` insere na memória o novo valor sintetizado. Este comportamento modela a atualização do atributo.

4.2.4 Chamada de não-terminal

A chamada de um não-terminal $A(v_1, ..., v_n)(w_1, ..., w_m)$ é traduzida para o programa definido em 4.13.

```
1    <v_n>
2    ...
3    <v_1>
4    Call "A"
5    Store <Addr w_m>
6    ...
7    Store <Addr w_1>
```

Figura 4.13: Código para chamada de não-terminal

A chamada de um não-terminal é representada pela instrução `Call`, a qual redireciona o fluxo de execução para um novo ponto do programa, de forma análoga a uma chamada de função. Para tal, o contador de programa é atualizado com o valor da linha indicada pela label `A`, linha do programa que representa a computação do não-terminal A. Simultaneamente, o registrador *sp* é ajustado de modo a iniciar um novo escopo de memória, passando a apontar para a posição correspondente ao tamanho corrente da memória. Esta atualização efetivamente isola o escopo de memória em uso e inicia um novo escopo a partir da próxima posição disponível, determinada pelo próprio tamanho da memória.

Adicionalmente, são empilhados, conforme a definição da máquina tradicional, o valor do contador de programa *pc* acrescido de uma unidade, que indica o ponto de retorno da execução após a conclusão da chamada, bem como o valor corrente do registrador *sp*, o qual permite a restauração do escopo de memória apropriado ao término da chamada. Quando há a noção de passagem de parâmetros para o não-terminal invocado, os n valores

correspondentes aos parâmetros são sintetizados e empilhados, de modo a possibilitar sua utilização durante a execução da chamada.

Após a execução da instrução `Call`, as instruções `Store` asseguram que, posteriormente ao retorno da chamada de não-terminal realizada pelo `Call`, os atributos sintetizados pelo não-terminal sejam atualizados na memória de acordo com os valores empilhados e o valor de `m`.

4.2.5 Escolha ordenada

A escolha ordenada e_1/e_2 é traduzida para o programa definido em 4.14.

```
1       Choice "L1"
2         <e_1>
3       Commit "L2"
4  "L1": <e_2>
5  "L2": ...
```

Figura 4.14: Código para escolha ordenada

O programa inicia salvando o estado corrente da máquina por meio da instrução `Choice`, a qual empilha uma quádrupla composta pelos valores $pc + l$, pelo índice i da entrada, pelo registrador *sp* e pela memória *M*. O armazenamento dessas informações é essencial para possibilitar a restauração completa do estado da Máquina em caso de falha da primeira alternativa da escolha ordenada. Diferentemente da instrução homônima definida em Ierusalimschy (2009), a instrução `Choice` passa a incluir explicitamente a memória e o registrador *sp* na pilha. Essa extensão é necessária para garantir a recuperação do escopo correto de memória, uma vez que a alternativa $\langle e_1 \rangle$ pode realizar modificações no estado da memória antes de falhar; nesse caso, tais modificações devem ser devidamente revertidas durante o processo de *backtracking*.

Após a execução da instrução `Choice`, o programa prossegue com as instruções correspondentes a $\langle e_1 \rangle$, expressão de *parsing* associada à primeira alternativa da escolha ordenada. Caso $\langle e_1 \rangle$ seja bem-sucedida e a execução se complete sem falhas, a instrução `Commit L2` é então acionada, removendo da pilha o estado previamente salvo e desviando o fluxo de execução para o final do padrão, identificado pela label `L2`.

A instrução `Commit` tem por finalidade consolidar a escolha realizada, indicando

que todas as instruções relativas a $\langle e_1 \rangle$ foram executadas com sucesso e que a primeira alternativa foi selecionada. Para tanto, essa instrução descarta a quádrupla empilhada pela instrução `Choice`. Além disso, o `Commit` é parametrizado por um valor natural, que especifica o endereço para o qual o fluxo de execução deve ser redirecionado, permitindo ignorar integralmente o escopo associado à alternativa $\langle e_2 \rangle$.

Caso $\langle e_1 \rangle$ falhe, a avaliação prossegue para a segunda alternativa da escolha ordenada. Para isso, a máquina realiza o processo de *backtracking*, restaurando o estado previamente salvo pela instrução `Choice` e redirecionando o fluxo de execução para a label `L1`, cujo endereço foi determinado a partir do valor $sp + l$ associado à referida instrução. Nesse ponto, a expressão $\langle e_2 \rangle$ é então avaliada. Se $\langle e_2 \rangle$ também resultar em falha, toda a escolha ordenada é considerada mal-sucedida, uma vez que não existem alternativas adicionais registradas na pilha. Por outro lado, caso a segunda alternativa seja bem-sucedida, a execução alcança a instrução identificada pela label `L2`, encerrando-se, assim, o escopo da escolha ordenada.

5 Formalização de PEGwSA em PLT Redex

Para a formalização, foi utilizada a ferramenta PLT *Redex*. A formalização completa encontra-se disponível no repositório https://github.com/lives-group/PEGwSA-parsing-machine. Este capítulo concentra-se nos aspectos mais relevantes da formalização em PLT *Redex*.

5.1 Definição da linguagem

O primeiro passo desse processo consiste na modelagem da linguagem da Máquina. Conforme mencionado anteriormente, o PLT *Redex* disponibiliza a função `define-language` para a realização dessa tarefa. A Figura 5.1 apresenta a definição formal dessa linguagem.

```
1  (define-language ParsingMachineLanguage
2
3      [I ::= Any
4             Fail
5             (Char natural)
6             (Choice integer)  (Commit integer)
7             (Jump integer)
8             (Call integer)  (Return natural)
9             (Load natural)  (Store natural)
10            (Push Value)  Pop
11            Add  Sub  Mult  Div  Eq  Lt
12            And  Or  Not
13            Head  Tail  Cons  Concat
14            Assert
15            Halt])
```

Figura 5.1: Especificação em PLT *Redex* da Linguagem da Máquina de *Parsing* para PEGwSA

Essa definição estabelece que o não-terminal `I` pode assumir a forma de uma instrução pertencente a um conjunto, cujos elementos correspondem às instruções homônimas definidas pela semântica. A utilização dessas instruções torna-se mais evidente no contexto da descrição das reduções.

5.2 Representação do Programa e da Entrada

Na Máquina de *Parsing* descrita por Ierusalimschy (2009), o programa P e a entrada S são tratados como implícitos na descrição semântica. No entanto, para viabilizar a leitura do programa e o consumo da entrada, torna-se necessário que ambos sejam explicitamente incorporados ao estado da máquina.

O não-terminal `Program`, representa o programa de entrada, isto é, o programa que descreve a PEGwSA através das instruções. Esse programa é modelado como uma lista composta por duas sublistas, que representam, respectivamente, a sequência de instruções já processadas e a sequência de instruções ainda não processadas. Por definição, a instrução corrente corresponde à cabeça da segunda sublista.

De forma análoga, o não-terminal `Input` segue a mesma estrutura, sendo representado por uma lista composta por duas sublistas: a primeira corresponde à sequência de números naturais que já foram lidos da entrada, enquanto a segunda representa a porção da entrada que ainda deve ser consumida. Por definição, a posição atual de leitura da entrada é determinada pela cabeça da segunda sublista.

```
1 (define-language ParsingMachineLanguage
2
3     [Program ::= ((I ...) (I ...))]
4
5     [Input ::= ((natural ...) (natural ...))])
```

Figura 5.2: Especificação em PLT *Redex* da Linguagem da Máquina de *Parsing* para PEGwSA

5.3 Valores dos Atributos

O não-terminal `Value`, descrito na Figura 5.3, representa o conjunto de valores aceitos no contexto da Máquina. Esse não-terminal pode assumir valores do tipo número natural, booleano ou lista. Além disso, `Value` é utilizado para definir os valores permitidos nas demais estruturas da linguagem.

O não-terminal `List`, ilustrado na Figura 5.3, por sua vez, define o valores lista, o qual pode ser vazio, representado pelo terminal `nil`, ou uma lista construída pelo terminal *cons* contendo o valor da cabeça e a cauda da lista.

```
(define-language ParsingMachineLanguage

    [Value ::= natural
               boolean
               List]

    [List ::= nill
              (cons Value List)])
```

Figura 5.3: Especificação em PLT *Redex* da Linguagem da Máquina de *Parsing* para PEGwSA

5.4 Definição da pilha

O não-terminal `Stack` define a estrutura de pilha, a qual é modelada como uma lista de elementos do tipo `StackEntry`. Esses elementos representam os tipos de valores admitidos na pilha, podendo corresponder a um número natural ou a uma tupla de números naturais, utilizados no tratamento do *backtracking*, bem como a uma entrada do tipo `Value` ou a uma memória, estruturas destinadas à manipulação de atributos. Além disso, admite-se ainda uma lista do próprio tipo `StackEntry`.

```
(define-language ParsingMachineLanguage

    [Stack ::= (StackEntry ...)]
    [StackEntry ::= natural
                    (natural natural)
                    (StackEntry ...)
                    Value
                    M])
```

Figura 5.4: Especificação em PLT *Redex* da Linguagem da Máquina de *Parsing* para PEGwSA

5.5 Representação da Memória

A memória, ilustrada na Figura 5.5, é modelada pelo não-terminal `M`, o qual representa uma lista de entradas do tipo `Value`. O endereço de memória associado a cada atributo é definido pelo desenvolvedor no momento da construção do programa de instruções, de modo que, durante a execução, a posição correspondente a cada atributo é previamente conhecida quando se faz necessário acessá-lo. O registrador `SPR`, por sua vez, é representado por um número natural.

A organização da memória inspira-se em arquiteturas de mais baixo nível, nas quais o registrador *sp* desempenha um papel análogo ao de um *stack pointer*, sendo responsável por indicar a região de memória atualmente ativa. Esse mecanismo permite delimitar o contexto de execução corrente, facilitando o acesso a dados locais e o controle do fluxo de execução durante chamadas e retornos de não-terminais.

```
1 (define-language ParsingMachineLanguage
2
3      [M ::= (Value ...)]
4      [SPR ::= natural ])
```

Figura 5.5: Especificação em PLT *Redex* da Linguagem da Máquina de *Parsing* para PEGwSA

5.6 Semântica

Com o propósito de definir a semântica da linguagem, utiliza-se a função `reduction-relation` para especificar o conjunto de regras de reescrita, as quais determinam como um termo deve ser transformado nessa linguagem. A cláusula `:domain` é utilizada para especificar o conjunto de configurações válidas sobre as quais a relação de redução está definida. Em outras palavras, ela determina a forma geral dos estados da máquina que podem participar das transições descritas pela semântica operacional.

No caso da Máquina de *Parsing* apresentada, o domínio é definido como uma tupla composta por sete elementos: o resultado da execução (`R`), o programa (`Program`), a entrada (`Input`), o contador de programa (`natural`), o índice da entrada (`natural`), a pilha de execução (`Stack`) e a memória (`M`). Essa definição explicita a estrutura completa de um estado da máquina, garantindo que cada regra de redução opere apenas sobre configurações bem-formadas.

As reduções mais relevantes para o desenvolvimento e a compreensão deste trabalho foram distribuídas em figuras distintas, com o objetivo de facilitar sua visualização; contudo, todas essas regras compõem um único programa semântico.

A Figura 5.6 apresenta a redução denominada `‘‘choice-match’’`. Essa redução estabelece que, quando a instrução corrente. ou seja, a instrução localizada na cabeça da segunda lista, corresponde a um comando `Choice` seguido de um número inteiro, o

programa é transformado da seguinte forma: a instrução consumida é movida para a cauda da primeira lista, o índice *pc* é incrementado em uma unidade, indicando que a próxima instrução a ser lida é a subsequente, e a entrada permanece inalterada.

```
(define PM
  (reduction-relation
   ParsingMachineLanguage
   #:domain (R Program Input natural natural Stack M)

    (--> (suc
         ((I_1 ...) ((Choice integer) I_2 ...))
         Input
         natural_pc
         natural_i
         (StackEntry ...)
         SPR
         M)

        (suc
        ((I_1 ... (Choice integer)) (I_2 ...))
        Input
        ,(+ (term natural_pc) 1)
        natural_i
        ((,(+ (term natural_pc) (term integer)) natural_i
        SPR ,(drop (term M) (term SPR)) ) StackEntry ...)
        SPR
         M)
     "choice-match")))
```

Figura 5.6: Relação de redução da instrução `Choice` da Máquina de *Parsing* para PEGwSA

Além disso, uma quádrupla é empilhada no topo da pilha, na qual o primeiro número natural corresponde à soma do valor corrente do contador de programa *pc* com o deslocamento especificado pelo número inteiro fornecido. O segundo elemento da quádrupla é um número natural que representa o índice i, indicando a posição atual da entrada. Os dois elementos restantes correspondem ao valor corrente do registrador *sp* e à memória.

Na implementação, a memória não é armazenada de forma integral. Em vez disso, ela é particionada por meio da função `drop`, a qual remove da memória M os primeiros SPR elementos, preservando apenas o sufixo restante. Essa redução tem por finalidade registrar o estado de retorno necessário ao mecanismo de *backtracking*, especificando tanto o ponto do programa para o qual a execução deve retornar quanto a posição da entrada a partir da qual o processamento deve ser retomado, bem como os valores do registrador

sp e da partição da memória *M* correspondentes ao escopo ativo.

A redução `call-match`, ilustrada da Figura 5.7, caracteriza o comportamento semântico quando a instrução corrente corresponde a um comando `Call integer`. Essa instrução é responsável por alterar o fluxo do programa, operação modelada pela função auxiliar `move-program`, que recebe como parâmetros uma estrutura do tipo `Program` e um número inteiro `integer`, produzindo um novo `Program` resultante do deslocamento de `integer` posições ao longo das listas que compõem o programa. Esse deslocamento pode ser positivo ou negativo, indicando, respectivamente, avanço ou retrocesso no fluxo de execução.

```
1  (define PM
2    (reduction-relation
3     ParsingMachineLanguage
4     #:domain (R Program Input natural natural Stack M)
5
6       (--> (suc ((I_1 ...) ((Call integer) I_2 ...))
7            Input
8            natural_pc
9            natural_i
10           (StackEntry ...)
11           SPR
12           M)
13
14           (suc
15           (moveProgram((I_1 ...) ((Call integer) I_2 ...)) integer)
16           Input
17           ,(+ (term natural_pc) (term integer))
18           natural_i
19           (,(+ (term natural_pc) 1) SPR StackEntry ...)
20           ,(length (term M))
21           M)
22       "call-match")))
```

Figura 5.7: Relação de redução da instrução `Call` da Máquina de *Parsing* para PEGwSA

No contexto da redução, o valor do contador de programa *pc* é atualizado pela soma com o valor `integer`, refletindo o desvio explícito no fluxo de execução. Simultaneamente, a pilha é estendida com dois valores: o endereço da próxima instrução a ser executada após o retorno da chamada, correspondente ao valor de *pc* acrescido de uma unidade, e o valor corrente do *sp*, que permite a posterior restauração do escopo de memória. Além disso, o *sp* é atualizado para o tamanho atual da memória, passando a delimitar um novo escopo de memória associado à chamada. Ao longo da execução dessa instrução, a cadeia de entrada e o conteúdo da memória *M* permanecem inalterados.

Dessa forma, a instrução `Call` modela a chamada a um novo ponto do programa, promovendo o desvio do fluxo de execução e registrando, tanto o ponto de retorno quanto o contexto de memória anterior, de modo a garantir a correta restauração do estado quando da execução da instrução de retorno.

A redução `load`, ilustrada na Figura 5.8, é aplicada quando a instrução localizada na cabeça da segunda lista do programa corresponde a um comando `Load natural`. Essa redução promove o avanço do fluxo de execução por meio do incremento do valor de *pc* em uma unidade e insere, no topo da pilha, o valor armazenado na memória. Diferentemente de um acesso absoluto, a posição efetiva da memória é calculada como um deslocamento relativo ao valor corrente do *sp*, que delimita o escopo de memória ativo. Assim, o valor é obtido na posição `natural + sp`. Para viabilizar essa operação, utiliza-se a função auxiliar `readMem`, a qual recebe como parâmetros um valor do tipo `natural` e uma memória `M`, retornando a estrutura do tipo `Value` armazenada no índice correspondente.

```
1  (define PM
2    (reduction-relation
3     ParsingMachineLanguage
4     #:domain (R Program Input natural natural Stack M)
5
6       (--> (suc ((I_1 ...) ((Load natural) I_2 ...))
7            Input
8            natural_pc
9            natural_i
10           (StackEntry ...)
11           SPR
12           M)
13
14           (suc ((I_1 ... (Load natural)) ( I_2 ...))
15           Input
16           ,(+ (term natural_pc) 1)
17           natural_i
18           ((readMem ,(+ (term natural) (term SPR)) M)
19           StackEntry ...)
20           SPR
21           M)
22      "load")))
```

Figura 5.8: Relação de redução da instrução `Load` da Máquina de *Parsing* para PEGwSA

A redução `store`, ilustrada na Figura 5.9, altera o fluxo de execução, incrementando o valor de *pc* em uma unidade. Nessa redução, o valor localizado no topo da pilha, representado por $Value_1$, é removido e armazenado na memória. O acesso à memória é realizado de forma relativa ao escopo corrente, utilizando o deslocamento `natural + sp`

para determinar a posição efetiva de escrita. Essa atualização é realizada por meio da função auxiliar `writeMem`, que recebe como parâmetros um valor do tipo `natural`, uma estrutura do tipo `Value` e uma memória `M`, produzindo uma nova memória na qual o valor $Value_1$ é armazenado na posição `natural + sp`.

```
1  (define PM
2    (reduction-relation
3     ParsingMachineLanguage
4     #:domain (R Program Input natural natural Stack M)
5
6      (--> (suc ((I_1 ...) ((Store natural) I_2 ...))
7            Input
8            natural_pc
9            natural_i
10           (Value_1 StackEntry ...)
11           SPR
12           M)
13
14           (suc ((I_1 ... (Store natural)) ( I_2 ...))
15           Input
16           ,(+ (term natural_pc) 1)
17           natural_i
18           (StackEntry ...)
19           SPR
20           (writeMem ,(+ (term natural) (term SPR)) Value_1 M))
21      "store")))
```

Figura 5.9: Relação de redução da instrução `Store` da Máquina de *Parsing* para PEGwSA

A Figura 5.10 ilustra a redução `add`, a qual, além de promover o avanço no fluxo de execução do programa, remove os dois valores posicionados no topo da pilha, realiza a operação de soma entre eles e empilha o resultado obtido. As demais operações definidas sobre pares de atributos numéricos, booleanos e listas apresentam comportamento análogo, diferindo apenas no tipo de operação aplicada, sendo que suas respectivas implementações podem ser consultadas em https://github.com/lives-group/PEGwSA-parsing-machine.

A redução associada à instrução `Return`, apresentada na Figura 5.11, modela o mecanismo de retorno de um procedimento, sendo responsável por restaurar corretamente o contexto de execução previamente armazenado na pilha. Essa redução é aplicada quando a instrução corrente, localizada na cabeça da segunda lista do programa, corresponde a um comando `Return` $natural_n$. O valor $natural_n$ indica a quantidade de entradas da pilha que foram sintetizadas e que devem ser disponibilizadas ao contexto responsável

```
(define PM
  (reduction-relation
   ParsingMachineLanguage
   #:domain (R Program Input natural natural Stack M)

     (--> (suc ((I_1 ...) (Add I_2 ...))
          Input
          natural_pc
          natural_i
          (natural_1 natural_2 StackEntry ...)
          SPR
          M)

          (suc ((I_1 ... Add ) ( I_2 ...))
          Input
          ,(+ (term natural_pc) 1)
          natural_i
          (,(+ (term natural_1) (term natural_2)) StackEntry ...)
          SPR
          M)
     "add")))
```

Figura 5.10: Relação de redução da instrução `Add` da Máquina de *Parsing* para PEGwSA

pela chamada do procedimento.

A redução `return-match` altera explicitamente o contador de programa. O novo valor de *pc*, denotado por `natural_pc1`, é recuperado diretamente da pilha de execução, juntamente com o valor atualizado do registrador de ponteiro de escopo, `SPR_1`. Esses valores correspondem ao contexto previamente salvo no momento da chamada do procedimento. A reorganização da pilha é realizada por meio da função auxiliar `splitStack`, a qual recebe como parâmetros o valor `natural_n` e a pilha corrente `(StackEntry ...)`. Como resultado, essa função particiona a pilha em três segmentos: o primeiro, representado por `(StackEntry_1 ...)`, corresponde às entradas que permanecem ativas após o retorno; o segundo segmento contém explicitamente os valores `natural_pc1` e `SPR_1`, responsáveis por restaurar o fluxo de execução e o escopo; por fim, o terceiro segmento, denotado por `(StackEntry_3 ...)`, representa os valores sintetizados, associadas ao ambiente local do procedimento encerrado. Após a aplicação da redução, o programa é reposicionado por meio da função `moveProgram`, que ajusta o fluxo de execução de acordo com o valor restaurado do contador de programa. A pilha passa a conter apenas as entradas relevantes ao contexto anterior, o valor do *sp* é atualizado para `SPR_1`, e a memória `M` permanece inalterada. Dessa forma, a instrução `Return` garante a correta restauração

```
1  (define PM
2    (reduction-relation
3     ParsingMachineLanguage
4     #:domain (R Program Input natural natural Stack M)
5
6      (--> (suc ((I_1 ...) ((Return natural_n) I_2 ...))
7           Input
8           natural_pc0
9           natural_i
10          (StackEntry ...)
11          SPR
12          M)
13
14         (suc
15         (moveProgram ((I_1 ...) ((Return natural_n) I_2 ...)) ,(- (term
              natural_pc1) (term natural_pc0)))
16         Input
17         natural_pc1
18         natural_i
19         (StackEntry_1 ... StackEntry_3 ...)
20         SPR_1
21         M)
22
23         (where ((StackEntry_1 ...) (natural_pc1 SPR_1 StackEntry_3 ...))
               (splitStack natural_n (StackEntry ...)))
24      "return-match")))
```

Figura 5.11: Relação de redução da instrução `Return` da Máquina de *Parsing* para PEGwSA

do estado de execução, assegurando a continuidade do programa no ponto imediatamente posterior à chamada do procedimento.

A redução responsável pelo mecanismo de *backtracking* é apresentada na Figura 5.12. Essa redução é aplicada a um termo cujo estado indica falha e tem como objetivo restaurar a máquina a um estado consistente, a partir de informações previamente armazenadas. Sua aplicação ocorre quando o topo da pilha contém uma quádrupla composta por três números naturais e uma estrutura de memória. Tal quádrupla é empilhada pela instrução *Choice*, sendo utilizada para registrar o contexto necessário à retomada da execução em caso de insucesso de uma escolha ordenada.

Quando uma falha é detectada — isto é, quando a primeira alternativa de uma escolha ordenada não é bem-sucedida — a máquina, por meio da redução `fail-restore`, retorna ao escopo previamente salvo pela instrução *Choice*. O programa é reposicionado com o auxílio da função auxiliar `moveProgram`, de modo a alcançar o deslocamento correspondente a `natural_newPC - natural_pc`, em que `natural_newPC` representa o valor

do contador de programa armazenado na quádrupla. De maneira análoga, a entrada é ajustada pela função `moveInput`, passando a refletir o deslocamento `natural_newI - natural_i`.

Adicionalmente, os valores de *pc* e *i* são atualizados com os valores recuperados da pilha, efetivando o retorno tanto no fluxo do programa quanto na posição da entrada, revertendo, assim, os efeitos das instruções que conduziram ao estado de falha. O registrador *sp* também é restaurado a partir do valor armazenado na quádrupla. Por fim, a memória é reconstruída por meio da função `mcopy`, a qual gera uma nova memória a partir da cópia seletiva das memórias envolvidas, controlada por um número natural que determina quantos elementos iniciais da primeira memória devem ser preservados antes que a cópia passe a considerar a segunda.

```
1  (define PM
2    (reduction-relation
3     ParsingMachineLanguage
4     #:domain (R Program Input natural natural Stack M)
5
6      (--> (fail
7           Program
8           Input
9           natural_pc
10          natural_i
11          ((natural_newPC natural_newI SPR_new M_sfx)
12          StackEntry ...)
13          SPR
14          M)
15
16         (suc
17         (moveProgram Program ,(-(term natural_newPC) (term natural_pc)))
18         (moveInput Input ,(- (term natural_newI) (term natural_i)))
19         natural_newPC
20         natural_newI
21         (StackEntry ...)
22         SPR_new
23         (mcopy SPR_new M M_sfx))
24      "fail-restore")))
```

Figura 5.12: Relação de redução da instrução `Fail` da Máquina de *Parsing* para PEGwSA

5.7 Limitações da Formalização

Apesar dos resultados obtidos, algumas limitações devem ser reconhecidas. Primeiramente, nem todas as operações previstas na definição formal de PEGwSA apresentada

em FERREIRA (2024) foram integralmente contempladas na implementação desenvolvida. Em particular, onvalor *mapa*, bem como as operações a ele associadas, não foram incluídos. De modo semelhante, a operação de *bind* não foi implementada. Embora essas ausências representem lacunas em relação à definição completa do formalismo, elas não comprometem significativamente os objetivos centrais do trabalho, uma vez que tais operações não são essenciais para a validação do mecanismo principal de execução e controle da Máquina de *Parsing* proposta.

Outra limitação refere-se ao processo de validação experimental. Os testes realizados restringiram-se a testes unitários, o que impossibilitou a cobertura exaustiva de todos os comportamentos e combinações possíveis das instruções e reduções definidas. Consequentemente, não se pode afirmar que todos os cenários de execução foram devidamente explorados, embora os testes efetuados tenham sido suficientes para verificar o funcionamento dos casos representativos considerados.

Por fim, destaca-se a ausência de uma prova formal de equivalência semântica entre a descrição abstrata de uma PEGwSA e o programa da Máquina de *Parsing* que a representa. A relação entre ambos foi estabelecida de maneira intuitiva e operacional, por meio da correspondência entre construções e reduções, mas não foi formalizada por meio de um argumento matemático rigoroso. Assim, embora os resultados obtidos indiquem uma aderência consistente entre os modelos, a equivalência semântica plena permanece como um aspecto em aberto para trabalhos futuros.

6 Conclusão

Este trabalho apresentou a definição semântica e a formalização de uma Máquina de *Parsing* para PEGwSA, utilizando a ferramenta PLT *Redex*. A abordagem adotada possibilitou a descrição precisa do comportamento operacional da máquina, bem como a mecanização de suas regras semânticas, contribuindo para uma compreensão mais rigorosa do modelo proposto.

A principal contribuição deste estudo consiste na extensão do modelo de máquina originalmente apresentado em Ierusalimschy (2009), de modo a torná-lo compatível com as PEGwSA. Essa extensão foi viabilizada pela introdução explícita da noção de memória e pelo desenvolvimento de uma semântica operacional capaz de lidar com instruções de manipulação de atributos, operações lógicas e estruturas de dados, como listas, ampliando significativamente o poder expressivo da máquina. A formalização em PLT *Redex* permitiu a mecanização do modelo semântico, possibilitando a verificação sistemática das regras de redução e a análise do comportamento da máquina em cenários concretos de execução.

Não obstante, algumas limitações devem ser consideradas. Nem todas as operações previstas na definição completa de PEGwSA foram implementadas, destacando-se a ausência do tipo de valor mapa, de suas operações associadas e da operação de *bind*. Além disso, a validação do modelo restringiu-se à realização de testes unitários, o que impossibilitou a cobertura exaustiva de todos os comportamentos possíveis da máquina. Por fim, não foi estabelecida uma prova formal de equivalência semântica entre a descrição abstrata de uma PEGwSA e o programa correspondente na Máquina de *Parsing*, permanecendo essa relação fundamentada em uma correspondência operacional intuitiva.

Como trabalhos futuros, destacam-se a incorporação das operações ausentes da PEGwSA, a ampliação da bateria de testes para incluir cenários mais complexos e abrangentes, bem como o desenvolvimento de uma prova formal de equivalência semântica entre as duas representações. Adicionalmente, investigações voltadas à otimização da máquina podem contribuir para o modelo formal desenvolvido.

Em síntese, as especificações apresentadas constituem uma base teórica para o desenvolvimento de analisadores sintáticos que conciliam o determinismo das PEGs com a expressividade dos atributos sintáticos, preservando a eficiência operacional característica das máquinas de *parsing*.

Bibliografia

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools*. 2. ed. [S.l.]: Pearson, 2008.

DAHER, G. et al. Pest control: A formal model of the pest parser generator. In: SBC. *Proceedings of the XXIII Brazilian Symposium on Programming Languages (SBLP'25)*. Recife, PE, 2025. p. 1–9.

FERREIRA, G. P. *Parsing Expression Grammar with Syntactic Attributes*. Tese (Doutorado) — Federal University of Juiz de Fora, 2024. Available at http://monografias.ice.ufjf.br/tcc-web/tcc?id=871.

FERREIRA, G. P. *Parsing Expression Grammar with Syntactic Attributes: Uma Formalização em PLT Redex*. Dissertação (Mestrado) — Universidade Federal de Juiz de Fora, 2024. Fonte.

FORD, B. Parsing expression grammars: A recognition-based syntactic foundation. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. [S.l.]: ACM, 2004. (POPL '04), p. 111–122. Fonte.

IERUSALIMSCHY, R. A text pattern-matching tool based on parsing expression grammars. *Software: Practice and Experience*, John Wiley and Sons, v. 39, p. 221–258, 2009. ISSN 0038-0644. Fonte.

NIELSON, H. R.; NIELSON, F. *Semantics with Applications: An Appetizer*. London: Springer, 2007. (Undergraduate Topics in Computer Science). ISBN 1846286913. Disponível em: ⟨https://doi.org/10.1007/978-1-84628-692-6⟩.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 5. ed. [S.l.]: Morgan Kaufmann, 2017.

PIERCE, B. C. *Types and Programming Languages*. 1st. ed. Cambridge, Massachusetts: The MIT Press, 2002. ISBN 0262162091, 9780262162098.

REIS, L. V. S. et al. The formalization and implementation of adaptable parsing expression grammars. *Science of Computer Programming*, v. 96, p. 191–210, 2014. Fonte.

REIS, L. V. S.; IORIO, V. O. D.; BIGONHA, R. S. Defining the syntax of extensible languages. In: *Proceedings of the 2014 ACM Symposium on Applied Computing (SAC)*. Gyeongju, Korea: ACM, 2014. p. 1569–1576. ACM Copyright 2014, ISBN 978-1-4503-2469-4/14/03.

ZHANG, J.; MORRISETT, G.; TAN, G. Interval parsing grammars for file format parsing. *Proceedings of the ACM on Programming Languages*, v. 7, n. PLDI, p. 1073–1095, 2023.