Paulo Roberto Massa Cereda

Macros como mecanismos de abstração em transformações textuais

São Paulo
2018

Paulo Roberto Massa Cereda

Macros como mecanismos de abstração em transformações textuais

Tese apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Doutor em Ciências.

São Paulo
2018

Paulo Roberto Massa Cereda

Macros como mecanismos de abstração em transformações textuais

Tese apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Doutor em Ciências.

Área de Concentração:
Engenharia de Computação

Orientador:
Prof. Dr. João José Neto

São Paulo
2018

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

FICHA CATALOGRÁFICA

Cereda, Paulo Roberto Massa

Macros como mecanismos de abstração em transformações textuais / P. R. M. Cereda – versão corr. – São Paulo, 2018.

220 p.

Tese (Doutorado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Linguagens formais 2. Linguagens formais (Geração) 3. Gramática (Redução) 4. Autômatos finitos 5. Transformação de programas I. Universidade de São Paulo, Escola Politécnica, Departamento de Engenharia de Computação e Sistemas Digitais II. t.

AGRADECIMENTOS

> Muito prazer, ao seu dispor
> se for por amor às causas
> perdidas.
>
> *Dom Quixote*
> ENGENHEIROS DO HAWAII

A Deus, pela presença em minha vida e pelos amigos que me deste para caminhar ao meu lado, fortalecendo os meus passos, suportando minhas fraquezas e ajudando-me a crescer.

Aos meus pais, Ederval e Lúcia, pelo amor, carinho, dedicação e paciência incondicionais no decorrer de tantos anos de esforços, lutas e sacrifícios. Poder contar com meus pais e partilhar as próprias dificuldades e sofrimentos é um grande dom de Deus. É um bálsamo que alivia a dor, é força em minha fraqueza.

Ao meu orientador, amigo e catequista, professor João José Neto, por me ajudar a compreender um pouco mais dos meus limites e potencialidades, e por sempre me ensinar a buscar as coisas do alto.

À minha amiga Renata Luiza Stange Carneiro Gomes, pelo apoio incondicional em todos os momentos, pela inaudita competência em nossas discussões técnicas e pela próspera parceria nas publicações.

À querida amiga Maria Lídia Rebello Pinho Dias Scoton, pesquisadora nata e modelo de excelência acadêmica, pelo valioso encorajamento durante a escrita desta tese até sua derradeira conclusão.

Ao amigo Newton Kiyotaka Miura, pelo companheirismo ao longo de toda a caminhada acadêmica, pelos inestimáveis conselhos e orientações, e pelos agradáveis momentos de descontração.

Aos professores Ítalo Santiago Vega, João Eduardo Kögler Júnior, Amaury Antônio de Castro Júnior, José de Oliveira Guimarães, Ricardo Luis de Azevedo da Rocha e Marcus Vinícius Midena Ramos, pelo estímulo e inspiração necessários durante esta caminhada acadêmica.

À Escola Politécnica da Universidade de São Paulo, ao Departamento de Engenharia de Computação e Sistemas Digitais e ao Laboratório de Linguagens e Técnicas Adaptativas, pela viabilização de um ambiente propício para pesquisa científica.

Finalmente, agradeço a todos que, direta ou indiretamente, contribuíram para a conclusão efetiva desta tese. Existem sentimentos e experiências que marcam nossa vida eternamente. De todos os laços, porém, o mais forte é o da amizade.

NOTA AO LEITOR

Inspirado pelo conselho de Sua Santidade, o Papa emérito Bento XVI, no prefácio do livro *Jesus de Nazaré*, peço ao potencial leitor um adiantamento de simpatia, sem o qual, nos diz o pontífice, não há nenhuma compreensão.

ACKNOWLEDGMENTS

Finished your thesis?

DAVID CARLISLE

Surprisingly, this text is not a carbon copy translation of its Portuguese counterpart, as I want to acknowledge the kind support of my friends from all around the globe.

First and foremost, I want to thank Barbara Beeton for her gentle nagging and motivation. Creative process is not linear at all. Her dedication to pushing me towards writing was invaluable.

Speaking of nagging, or more so on the motivation aspect, I want to thank David Carlisle for his constant reminders on the need of an actual text rather than a couple of recursive macros along the lines of `\def\thesis{\thesis}`. However, since my thesis is about macros, I still argue that such set of definitions would be significant regardless. Moreover, David taught me the importance of never trusting an Englishman regarding English grammar. Ay up me duck.

Also on the constructive and supportive side, I want to thank Enrico Gregorio for his utmost attention and dedication to helping me write better code and value typographical aspects. Of course, as a consequence, I became paranoid with spotting kerning issues and other nuisances. It is a good gift, in retrospect. Grazie mille, maestro.

I kindly thank my friends from the LaTeX3 project for their understanding and patience in my absence from the discussions and code writing, in particular Bruno Le Floch, Frank Mittelbach, Joseph Wright and Will Robertson. Of course, now I have no excuse to not go back to the Lua projects (specially `l3build`), and this text might as well act as a confession note.

At last, but not least, I want to express my gratitude to all my friends that directly or indirectly supported the writing of my thesis, namely Alan Munn, Andrew Stacey, Arthur Reutenauer, Brent Longborough, Carla Maggi, Christian Hupfer, Clea F. Rees, Clemens Niederberger, Claudio Beccari, Christopher Hughes, Heiko Oberdiek, Harish Kumar Holla, Francesco Endrici, Gonzalo Medina, Henri Menke, İlhan Polat, Johannes Böttcher, Karl Berry, Martin Scharrer, Nicola Talbot, Patrick Gundlach, Robin Laakso, Sean Allred, Susanne Raab, Stefan Kottwitz, Stephan Lehmke, Claudio Fiandrino, Tomáš Hejda, Herbert Voß, Martin Schröder, Jürnjakob Dugge, Paul Gessler, Petr Olšák, Torbjørn Taskjelle and Ulrike Fischer. Thank you, friends.

Now, I need to address a very sensitive concern. It has been part of the worldwide TeX folklore that I am stuck in a never-ending thesis writing endeavour. However, I believe this document sort of debunks and invalidates such a fable, as it is my actual thesis. So I would like to offer a compromise solution: for comedic purposes only, as to perpetuate the fable, this thesis does not contain my thesis. It is silly and deliciously paradoxical. It is exactly this sort of witty humour that is missing in the world today, so let us break such barrier. After all, “a smile is the shortest distance between two people” (Victor Borge).

RESUMO

Abstração é um processo que consiste em encontrar similaridades em artefatos e omitir detalhes desnecessários em um particular momento. Em geral, tal processo resulta em simplificação, substituindo situações do mundo real complexas e excessivamente detalhadas por modelos compreensíveis que admitem resolução. Na computação, existem estilos de programação que fornecem ao programador uma visão particular sobre a organização e execução de um programa. Cada estilo viabiliza formas de representação e tratamento de abstrações aderentes ao conjunto de conceitos, valores, percepções e práticas compartilhadas por uma comunidade. Em particular, o fenômeno de reescrita de termos viabiliza transformações entre espaços de abstração. Como instância de tal fenômeno, macros constituem um padrão sintático que especifica uma transformação simbólica ou algorítmica sobre uma sequência de símbolos associada. Na ocorrência de uma instância de tal padrão sintático, este é substituído pela aplicação de sua transformação correspondente. Dada a importância da disponibilização de estruturas de representação mais convenientes às necessidades dos usuários, o objetivo principal desta tese é tratar da utilização de sistemas de reescrita como mecanismos de abstração em transformações textuais. Para tal, técnicas de projeto e aspectos de implementação de tais sistemas são apresentados, com enfoque em macros.

Palavras-chave: abstração, mecanismos de abstração, sistemas de reescrita, macros

ABSTRACT

Abstraction is a process of finding similarities in artifacts and omitting unnecessary details at a particular moment. In general, such a process results in simplification, replacing complex and overly detailed real-world situations with understandable models that admit resolution. In computing, there are programming styles that give the programmer a particular insight into the organization and execution of a program. Each style enables forms of representation and treatment of abstractions adhering to the set of concepts, values, perceptions and practices shared by a community. In particular, the term rewriting phenomenon enables transformations along spaces of abstraction. As an instance of such a phenomenon, macros constitute a syntactic pattern that specifies a symbolic or algorithmic transformation over an associated symbol sequence. In the occurrence of an instance, the matched syntactic pattern is replaced by the application of its corresponding transformation. Given the importance of making representation structures more convenient to users' needs, this thesis aims at addressing the use of rewriting systems as abstraction mechanics in textual transformations. To this end, design techniques and implementation aspects of such systems are presented, focusing on macros.

Keywords: abstraction, abstraction mechanisms, rewriting systems, macros

LISTA DE FIGURAS

1.1 Exemplos de abstrações. 26
1.2 Conjunto de conceitos, valores, percepções e práticas compartilhadas por uma comunidade constituem uma visão particular da realidade. 27
1.3 Representações do conceito de média aritmética utilizando diferentes mecanismos de abstração. 28
1.4 Adaptatividade para expressão do fenômeno de repetição, utilizando ações adaptativas para acomodar novas sequências na instância do programa em execução. 31
1.5 Instâncias do fenômeno de reescrita de termos. 34
1.6 Exemplo de utilização de uma macro em um código-fonte escrito na linguagem C. 37

2.1 Notação de Wirth, em sua forma original (WIRTH, 1977), expressa utilizando a própria notação que a define. 42
2.2 Exemplo de chamada da submáquina a_j. 44
2.3 Notação gráfica utilizada para determinar a situação de execução das funções adaptativas associadas a uma transição. 48

3.1 Síntese do conceito de macro, de acordo com a terminologia proposta nas Definições 41, 42 e 43. 52
3.2 Exemplo de macros na linguagem TEX. 53
3.3 Exemplo de ocorrência da macro sintática `when` existente na linguagem Clojure. O comando `macroexpand` realiza a expansão da macro. 54
3.4 Árvores de sintaxe concreta da ocorrência da macro `when` e de sua transformação de substituição; símbolos terminais são representados nas folhas, enquanto símbolos não-terminais são representados nos nós interiores. 55
3.5 Definição da macro sintática `when` através do mecanismo de extensão sintática `defmacro` disponibilizado na linguagem Clojure (HIGGINBOTHAM, 2015). 55
3.6 Padrão sintático da macro `when` existente na linguagem Clojure e sua estrutura de transformação correspondente. 56
3.7 Exemplo de uma macro com transformação algorítmica na linguagem TEX. 56
3.8 Exemplo de aplicação da regra de autorreferência do compilador GCC. Consequentemente, o pré-processador adota a estratégia de expansão de um passo e interrompe expansões subsequentes. 60
3.9 Exemplo de estratégia de expansão de múltiplos passos adotada pelo expansor de macros da linguagem TEX. 61
3.10 Diagrama das estratégias de expansão, com $a \rightarrow^* c$, e c está na forma normal. Termos tracejados indicam reescritas intermediárias. 62
3.11 Estrutura do método de desenvolvimento em níveis de abstração. 62
3.12 Alfabetos de entrada e saída e composição funcional das camadas c_1, c_2 e c_3, tal que $c_3 \circ c_2 \circ c_1 : \Sigma_1^* \mapsto^* \Sigma_4^*$. 63

3.13 Níveis de abstração de um expansor de macros com padrão sintático regular, sem parâmetros e com transformações puramente simbólicas. 64
3.14 Exemplo de expansão de macros em um texto, de acordo com a especificação proposta do expansor da Figura 3.13. 65
3.15 Fluxograma das etapas de expansão de macros simples em um texto. . . . 66
3.16 Autômato finito determinístico M_1 que descreve o analisador léxico para o expansor de macros da Figura 3.13. 68
3.17 Motor de eventos ME_1 como modelo de implementação para o analisador léxico proposto (nível 2 da Figura 3.13). A sequência de símbolos utiliza o texto da Figura 3.14. 70
3.18 Funções auxiliares para a definição de *tokens* e a manipulação de cadeias de símbolos. 71
3.19 Implementação do analisador léxico proposto na Subseção 3.4.1. 71
3.20 Execução do analisador léxico da Figura 3.19 com o texto da Figura 3.14, de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Espaços em branco (exceto os que foram utilizados como separadores de *tokens* na impressão) estão deliberadamente marcados como visíveis. 72
3.21 Motor de eventos ME_2 como modelo de implementação da função de reclassificação χ (Equação 3.8, nível 3 da Figura 3.13). 74
3.22 Autômato finito determinístico M_2 que descreve o processamento de *tokens* (nível 4) para o expansor de macros da Figura 3.13. 75
3.23 Motor de eventos ME_3 como modelo de implementação do processamento de *tokens* (nível 4) para o expansor de macros da Figura 3.13. 76
3.24 Composição hierárquica dos três motores de eventos ME_1, ME_2 e ME_3 (Figuras 3.17, 3.21 e 3.23, respectivamente), resultando no projeto completo do expansor de macros da Figura 3.13. 77
3.25 Implementação do expansor de macros simples, sintetizando as fases de análise léxica (nível 2 da Figura 3.13) e processamento de *tokens* (níveis 3 e 4 da Figura 3.13). 78
3.26 Execução do expansor de macros simples da Figura 3.25 com o texto e tabela de macros da Figura 3.14, imprimindo a cadeia de símbolos resultante no terminal de comando. 79

4.1 Sintaxe e operação das primitivas de gerenciamento da tabela de macros, de acordo com a Definição 62. 82
4.2 Sintaxe e operação das primitivas de incremento e decremento de valores inteiros, de acordo com a Definição 63. 83
4.3 Sintaxe e operação das primitivas de controle de fluxo, de acordo com a Definição 64. 83
4.4 Sintaxe e operação das primitivas de comparação entre valores numéricos, de acordo com a Definição 65. 84
4.5 Sintaxe e operação das primitivas de cálculos aritméticos, de acordo com a Definição 66. 84
4.6 Sintaxe e operação das primitivas de operações lógicas, de acordo com a Definição 67. 85
4.7 Sintaxe e operação das primtivas de repetição, de acordo com a Definição 68. 86
4.8 Sintaxe e operação das primitivas de gerenciamento de símbolos, conforme a Definição 69. 86

4.9 Sintaxe e operação da primitiva de reprodução literal, de acordo com a Definição 70. 87
4.10 Sintaxe e operação da primitiva de estratégia de expansão, de acordo com a Definição 71. 87
4.11 Sintaxe e operação da primitiva de projeção, de acordo com a Definição 72. 88
4.12 Sintaxe e operação da primitiva que implementa a ação adaptativa de inserção (Definição 73). 89
4.13 Sintaxe e operação da primitiva que implementa a ação adaptativa de remoção (Definição 73). 90
4.14 Sintaxe e operação da primitiva que implementa a ação adaptativa de consulta (Definição 73). 91
4.15 Autômato finito determinístico que descreve o reconhecimento de sentenças válidas da brincadeira *duck, duck, goose* do Exemplo 14. 92
4.16 Definição da macro `DDG` que obtém uma sentença válida da brincadeira *duck, duck, goose* através de repetição iterativa. 93
4.17 Definição da macro `DDG` que obtém uma sentença válida da brincadeira *duck, duck, goose* através de repetição recursiva. 94
4.18 Definição da macro `DDG` que obtém uma sentença válida da brincadeira *duck, duck, goose* através de repetição por adaptatividade. 94
4.19 Exemplo de expansão da instância de macro `DDG[(1)]`, de acordo com a definição apresentada na Figura 4.18. 95
4.20 Definição da macro `FIB` que obtém o n-ésimo termo da sequência de Fibonacci, de acordo com a Equação 4.1. 97
4.21 Definições das macros `AD`, `SB`, `ML` e `DV` para expressão das operações aritméticas de soma, subtração truncada, multiplicação e divisão de números inteiros através de indução. 97
4.22 Autômato adaptativo M que reconhece cadeias pertencentes à linguagem $L = \{w \in \{a,b,c\}^* \mid w = a^n b^n c^n, n \in \mathbb{N}, n \geq 1\}$. A função adaptativa $\mathcal{A}$ é apresentada no Algoritmo 4.5. 99
4.23 Definição da macro `ABC` para geração de sentenças dependentes de contexto, na forma $a^n b^n c^n$ através de adaptatividade. 99
4.24 Trecho de expansão da instância da macro `ABC[(2)]`, destacando a inserção dos símbolos durante as duas iterações previstas. Por razões de legibilidade, as primitivas adaptativas de remoção foram omitidas. 100
4.25 Exemplo de organização de um livro em capítulos, seções e subseções, dispostos em uma estrutura de árvore. 101
4.26 Sintaxe e operação da primitiva de exibição de uma caixa de diálogo de entrada de dados, utilizando recursos do sistema operacional. 102
4.27 Definições das macros `CHAPTER`, `SECTION` e `SUBSECTION` para a especificação de capítulos, seções e subseções de um livro. 102
4.28 Editoração da subárvore e_3 (excerto da Figura 4.25), composta por um capítulo e duas seções. 103

5.1 Políticas de visibilidade das ações adaptativas em um expansor de macros de propósito geral, com as regras de reescrita (a,b), (b,c), (c,d) e (h,i). . 107

A.1 Exemplo de expansão de macros em um texto, de acordo com a especificação proposta do expansor de macros com parâmetros. 126

A.2 Fluxograma das etapas de expansão de macros com parâmetros em um texto. 127
A.3 Motor de eventos ME_4 como modelo de implementação da função de reclassificação χ_2 (Equação A.1). 129
A.4 Autômato de pilha estruturado M_3 que descreve a identificação de parâmetros para o expansor de macros da Seção A.1. 131
A.5 Motor de eventos M_5 como modelo de implementação da identificação de parâmetros. 132
A.6 Funções auxiliares `slice` e `extract` para extração de parâmetros em uma lista, de acordo com a estrutura de delimitação definida na especificação do expansor de macros da Seção A.1. 133
A.7 Versão modificada da função `take` (Figura 3.19), estendida para incluir a identificação e extração de potenciais parâmetros de macros. 134
A.8 Execução da função modificada `take` da Figura A.7 com o texto da Figura A.1, de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Espaços em branco (exceto os que foram utilizados como separadores de *tokens* na impressão) estão deliberadamente marcados como visíveis. 135
A.9 Motor de eventos ME_2 reclassificando a sequência de *tokens* para o expansor de macros Seção A.1. 136
A.10 Motor de eventos ME_3 processando a sequência de *tokens* reclassificados para o expansor de macros da Seção A.1. 137
A.11 Composição hierárquica dos cinco motores de eventos ME_1, ME_4, ME_5, ME_2 e ME_3 (Figuras 3.17, A.3, A.5, 3.21 e 3.23, respectivamente), resultando no projeto completo do expansor de macros com parâmetros da Seção A.1. 138
A.12 Implementação do expansor de macros com parâmetros, sintetizando as fases de análise léxica, identificação de parâmetros e processamento de *tokens*. 140
A.13 Execução do expansor de macros com parâmetros da Figura A.12 com o texto e tabela de macros da Figura A.1, imprimindo a cadeia de símbolos resultante no terminal de comando. 140
A.14 Exemplo de expansão de macros em um texto, de acordo com a especificação do expansor de macros com delimitadores contextuais. 141
A.15 Fluxograma das etapas de expansão de macros com delimitadores contextuais em um texto. 143
A.16 Autômato adaptativo M_3 que descreve o analisador léxico para o expansor de macros com delimitadores contextuais da Seção A.2. As funções adaptativas $\mathcal{B}$ e C são apresentadas nos Algoritmos A.1 e A.2. 144
A.17 Autômato adaptativo M_4 da Figura A.16 acrescido das ações semânticas a_4, a_6, a_7, a_8, a_9 e a_{10}. 146
A.18 Motor de eventos ME_6 como modelo de implementação do analisador léxico para o expansor de macros da Seção A.2. A sequência de símbolos utiliza o texto da Figura A.14. 148
A.19 Funções auxiliares de normalização de espaços em branco em nomes de macros e de verificação de símbolos potencialmente descartáveis (espaços em branco, tabulações e quebras de linha). 149

A.20 Implementação do analisador léxico proposto na Subseção A.2.1. O trecho de código implementa os estados 1, 2, 3 e 4 do autômato adaptativo M_4 da Figura A.16. 150
A.21 Execução da função `take` da Figura A.20 com o texto da Figura A.14, de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Espaços em branco (exceto os que foram utilizados como separadores de *tokens* na impressão) estão deliberadamente marcados como visíveis. 151
A.22 Motor de eventos ME_3 processando a sequência de *tokens* para o expansor de macros da Seção A.2. 152
A.23 Composição hierárquica dos dois motores de eventos ME_6 e ME_3 (Figuras A.18 e 3.23, respectivamente), resultando no projeto completo do expansor de macros da Seção A.2. 153
A.24 Implementação do expansor de macros da Seção A.2, sintetizando as fases de análise léxica e processamento de *tokens*. 154
A.25 Execução do expansor de macros da Figura A.24 com o texto e tabela de macros da Figura A.14, imprimindo a cadeia de símbolos resultante no terminal de comando. 154
A.26 Exemplo de dependência cíclica na expansão de macros com padrão sintático regular (tipo 3 na hierarquia de Chomsky) em um texto. A reescrita de termos jamais terminará. 155
A.27 Implementação da validação da tabela de macros da Subseção A.3.1. 157
A.28 Execução da função de validação `validate` da Figura A.27 com as tabelas de macros das Figuras 3.14 e A.26. 158
A.29 Exemplo de expansão de macros em um texto utilizando uma estratégia de resolução de parâmetros como macros de escopo local com padrão sintático regular. 159
A.30 Tabela de macros estendida com a inclusão de escopo. A política de resolução de nomes prioriza definições locais. 160
A.31 Fluxograma das etapas da resolução de parâmetros como macros de escopo local na fase de expansão. 160
A.32 Implementação do expansor de macros da Seção A.2 utilizando a estratégia de resolução de parâmetros como macros de escopo local. 162
A.33 Execução do expansor de macros da Figura A.32 com o texto e tabela de macros da Figura A.29, imprimindo a cadeia de símbolos resultante no terminal de comando. 163
A.34 Exemplo de expansão de macros com transformações algorítmicas em um texto. 164
A.35 Fluxograma das etapas de expansão de macros com suporte a primitivas. . 164
A.36 Implementação do expansor de macros da Seção A.2, estendido para incluir suporte a transformações algorítmicas. 166
A.37 Execução do expansor de macros da Figura A.36 com o texto e tabela de macros da Figura A.34, imprimindo a cadeia de símbolos resultante no terminal de comando. 167
A.38 Implementação das primitivas de gerenciamento da tabela de macros (Definição 62). 168
A.39 Implementação das primitivas de incremento e decremento de valores inteiros (Definição 63). 169
A.40 Implementação das primitivas de controle de fluxo (Definição 64). 169

A.41 Implementação das primitivas de comparação entre valores numéricos (Definição 65). 170
A.42 Implementação das primitivas de operações aritméticas (Definição 66). . . 171
A.43 Implementação das primitivas de de operações lógicas (Definição 67). . . . 172
A.44 Implementação das primitivas de repetição (Definição 68). 172
A.45 Implementação das primitivas de gerenciamento de símbolos (Definição 69).173
A.46 Implementação da primitiva de reprodução literal (Definição 70). 173
A.47 Implementação da primitiva de estratégia de expansão (Definição 71). . . . 174
A.48 Implementação da primitiva de projeção (Definição 72). 175
A.49 Implementação das primitivas de adaptatividade (Definição 73). 176

B.1 Notação de Wirth estendida para inclusão de metadados descritivos, expressa utilizando a notação original. 180
B.2 Autômato finito determinístico W que descreve o analisador léxico para a notação de Wirth estendida (Figura B.1). 181
B.3 Implementação do analisador léxico proposto para a notação de Wirth estendida (Figura B.1). 182
B.4 Execução do analisador léxico da Figura B.3, de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. 183
B.5 Função auxiliar `transition` para representação semântica de uma transição estendida do autômato sendo construído. 184
B.6 Autômato de pilha estruturado G para geração do analisador sintático a partir de uma gramática escrita em notação de Wirth estendida. 185
B.7 Ação semântica de criação de nova submáquina. 186
B.8 Ação semântica de novo escopo. 186
B.9 Ação semântica de fechamento de escopo. 187
B.10 Ação semântica de nova transição. 187
B.11 Ação semântica de abertura de colchetes. 187
B.12 Ação semântica de abertura de chaves. 187
B.13 Ação semântica de adição de opção. 188

C.1 Possível organização de um motor de eventos de propósito geral. 190
C.2 Exemplo de uma especificação de um motor de eventos no formato YAML, destacando as três seções principais. 191
C.3 Organização da ferramenta `eventengine` em modo interativo. 192
C.4 Organização da ferramenta `eventengine` em modo de biblioteca. 193
C.5 Exemplo de utilização da ferramenta `eventengine` em modo de biblioteca. Por razões didáticas, partes supérfluas do código-fonte foram omitidas. . . 193
C.6 Especificação YAML de um servidor de eco como um motor de eventos. . . 194
C.7 Exemplo de sessão interativa do motor de eventos de um servidor de eco (Figura C.6). 195
C.8 Especificação YAML de um analisador de palíndromos como um motor de eventos. 195
C.9 Exemplo de sessão interativa do motor de eventos de um analisador de palíndromos (Figura C.8). 196
C.10 Especificação YAML de um autômato finito determinístico H que reconhece cadeias pertencentes à linguagem $L = \{w \in \{a,b\}^* \mid w = a(ba)^*\}$ como um motor de eventos. 197

C.11 Autômato finito determinístico H que reconhece cadeias pertencentes à linguagem $L = \{w \in \{a,b\}^* \mid w = a(ba)^*\}$. 197
C.12 Exemplo de sessão interativa do motor de eventos de um autômato finito (Figura C.10). 198

D.1 Sintaxe e operação da primitiva `get input` de exibição de uma caixa de diálogo de entrada de dados, utilizando recursos do sistema operacional. . 200
D.2 Sintaxe e operação da primitiva `ask question` de exibição de uma caixa de diálogo de confirmação, utilizando recursos do sistema operacional. . . 201
D.3 Sintaxe e operação da primitiva `show message` de exibição de uma caixa de diálogo de mensagem, utilizando recursos do sistema operacional. . . . 202
D.4 Interface gráfica do usuário disponibilizada pelo expansor de macros E. . . 202
D.5 Definição da macro `fatorial` que calcula recursivamente o fatorial de um número, de acordo com o Algoritmo 1.1-a. 203
D.6 Expansão de algumas instâncias da macro `fatorial` (Figura D.5) na interface gráfica do usuário. 204
D.7 Definição da macro `imprime fatorial` que imprime recursivamente a fórmula do fatorial de um número, inspirada no Algoritmo 1.1-a. 204
D.8 Expansão de algumas instâncias da macro `imprime fatorial` (Figura D.7) na interface gráfica do usuário. 205

E.1 Representação textual das produções de uma gramática livre de contexto como formato de entrada da ferramenta de geração de sentenças. 207
E.2 Interface gráfica do usuário disponibilizada pelo gerador de sentenças. . . 208
E.3 Caixa de diálogo de definição de uma restrição na ferramenta de geração de sentenças. 209
E.4 Representação textual das produções da gramática livre de contexto G, tal que $L(G) = \{w \in \{a,b\}^* \mid w = a^n b^n, n \geq 0\}$. 210
E.5 Geração de uma sentença a partir da representação textual das produções da gramática livre de contexto G (Figura E.4), sem restrições, na interface gráfica do usuário. 211
E.6 Representação textual das produções da gramática regular H. 211
E.7 Geração de uma sentença a partir da representação textual das produções da gramática regular H (Figura E.6), com restrições, na interface gráfica do usuário. 212
E.8 Representação textual das produções da gramática livre de contexto I. . . . 213
E.9 Geração de uma sentença a partir da representação textual das produções da gramática livre de contexto I (Figura E.8) na interface gráfica do usuário. 213

LISTA DE TABELAS

3.1 Correspondências entre as terminologias histórica e unificada. 53
3.2 Captura das regras sintáticas `BOOL-EXPR` e `{ STMT }` do padrão sintático referente à ocorrência da macro `when` (Figura 3.3). 54
3.3 Classes gramaticais do analisador léxico do expansor de macros da Figura 3.13. 67
3.4 Exemplos de cadeias pertencentes às classes gramaticais da Tabela 3.3. . . 67
3.5 Ações semânticas disponibilizadas no processamento de *tokens* do expansor de macros da Figura 3.13. 72

A.1 Ações semânticas disponibilizadas na identificação de parâmetros do expansor de macros da Seção A.1. 130

B.1 Classes gramaticais do analisador léxico proposto para a notação de Wirth estendida (Figura B.1). 180
B.2 Descrição dos símbolos pertencentes às transições do autômato finito determinístico W da Figura B.2, no qual $\sigma \in \Sigma$, S, L e D denotam o símbolo corrente e os conjuntos de símbolos, letras e dígitos, respectivamente. . . . 181
B.3 Ações semânticas associadas às transições do autômato de pilha estruturado G (Figura B.6), representadas como funções na linguagem Lua. 186

D.1 Correspondências entre as primitivas descritas na Seção 4.1 e as disponibilizadas no expansor de macros E proposto neste apêndice. 200

LISTA DE ALGORITMOS

1.1 Cálculo do fatorial de um número 30
4.1 Sentença de *duck, duck, goose* através de repetição iterativa 92
4.2 Sentença de *duck, duck, goose* através de repetição recursiva 93
4.3 Obtenção do n-ésimo termo da sequência de Fibonacci 96
4.4 Operações aritméticas em números inteiros através de indução 98
4.5 Função adaptativa $\mathcal{A}(p_1, p_2)$ 99
A.1 Função adaptativa $\mathcal{B}(a, b, c, d)$ 144
A.2 Função adaptativa $C(a, b, c)$ 145

SUMÁRIO

1 Introdução **25**
1.1 Justificativa 33
1.2 Objetivos 38
1.3 Organização 39

2 Conceitos **41**
2.1 Notação de Wirth 41
2.2 Autômato de pilha estruturado 42
2.3 Autômato adaptativo 44

3 Macros **51**
3.1 Terminologia 51
3.2 Estratégias de expansão 60
3.3 Estratificação em camadas 62
3.4 Aspectos de implementação 65
3.4.1 Análise léxica 66
3.4.2 Processamento de *tokens* 71

4 Operações do expansor de macros **81**
4.1 Conjunto de primitivas 81
4.2 Exemplos de uso 89

5 Conclusões **105**
5.1 Discussões 105
5.2 Contribuições 108
5.3 Trabalhos futuros 109
5.4 Considerações finais 110

Referências bibliográficas **113**

A Exemplos de implementações de expansores de macros **125**
A.1 Expansão de macros com parâmetros 125
A.1.1 Identificação de parâmetros 128
A.1.2 Processamento de *tokens* 134
A.2 Expansão de macros com delimitadores contextuais 139
A.2.1 Análise léxica 142
A.2.2 Processamento de *tokens* 149
A.3 Funcionalidades complementares 155
A.3.1 Validação da tabela de macros 155
A.3.2 Resolução de parâmetros como macros de escopo local 158
A.3.3 Primitivas e transformações algorítmicas 163
A.4 Implementação do conjunto de primitivas 167

B Uma proposta de extensão para a notação de Wirth **179**
B.1 Aspectos da metalinguagem 179
B.2 Análise léxica 180
B.3 Geração de autômatos 184

C Uma ferramenta para geração de motores de eventos **189**
C.1 Aspectos de implementação 189
C.2 Modos de operação 191
C.3 Exemplos de uso 193

D Um expansor de macros implementado na linguagem Java **199**
D.1 Conjunto de primitivas 199
D.2 Interface gráfica do usuário 201
D.3 Exemplos de uso 203

E Um gerador de sentenças livres de contexto **207**
E.1 Representação textual das produções 207
E.2 Interface gráfica do usuário 208
E.3 Exemplos de uso 210

Glossário **215**

CAPÍTULO 1

INTRODUÇÃO

> The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.
>
> DONALD KNUTH

Abstração é um processo que consiste em encontrar similaridades em artefatos e omitir detalhes que não são importantes em um particular momento (KRAMER, 2007). De modo mais abrangente, dois aspectos destacam-se:

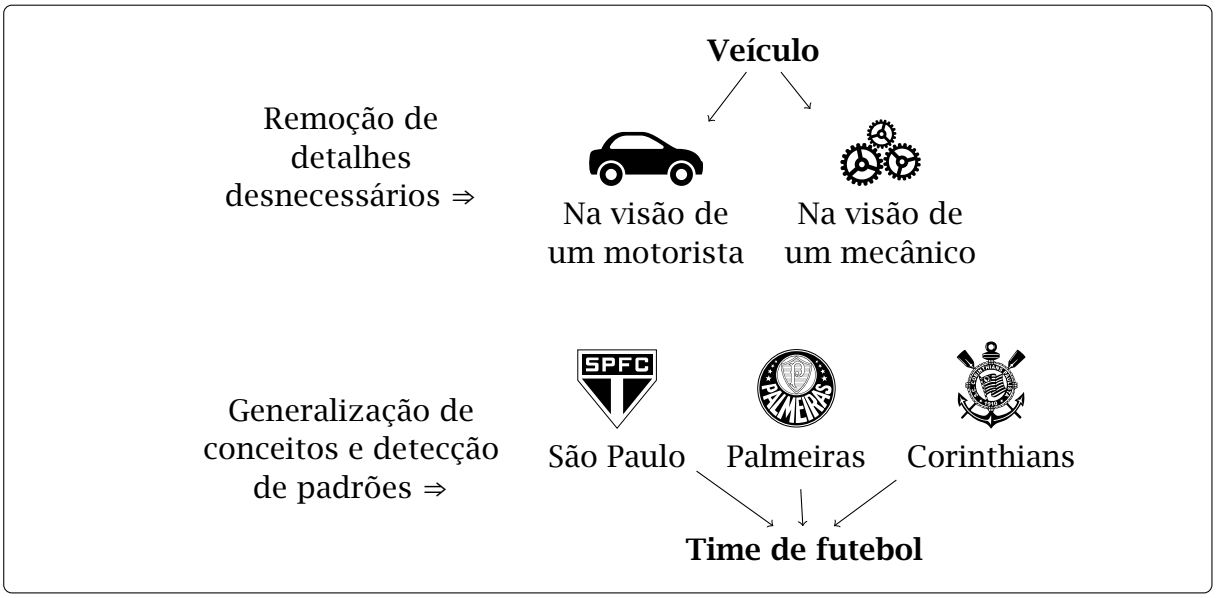
i) Remoção de detalhes desnecessários, isto é, detalhes que não contribuem efetivamente para a compreensão do artefato ou que aumentam, desnecessariamente, a complexidade no nível de observação corrente (ROBERTS, 2009).

ii) Generalização de conceitos e detecção de padrões, eventualmente a partir de instâncias específicas, tratando da representação do conhecimento acerca do artefato, de acordo com o nível de observação corrente (AHO; ULLMAN, 1995).

Exemplo 1 (abstração)**.** Considere os exemplos de abstrações apresentados na Figura 1.1. Na primeira parte (remoção de detalhes desnecessários), o artefato é um veículo; este pode ser representado como um automóvel convencional na observação de um motorista ou como um conjunto de componentes mecânicos, elétricos e hidráulicos na observação de um mecânico. Na segunda parte (generalização de conceitos e detecção de padrões), artefatos específicos – São Paulo, Palmeiras e Corinthians – são generalizados para representar, em uma observação mais abrangente, o conceito de um time de futebol. □

Em geral, abstração resulta em simplificação, substituindo situações do mundo real complexas e excessivamente detalhadas por modelos compreensíveis que admitem resolução (AHO; ULLMAN, 1995). Adicionalmente, pode-se reduzir a complexidade de um determinado problema através de seu particionamento em subproblemas menores, com abstrações em diferentes níveis de observação (KORF, 1987; KNOBLOCK, 1989, 1993; ANZAI; SIMON, 1979). A resolução de um problema em

Figura 1.1: Exemplos de abstrações.



Fonte: autor.

um espaço de abstração simplificado oferece subsídios para o tratamento de problemas similares em níveis de abstrações superiores (KNOBLOCK, 1990). O processo é então repetido até que o problema original seja resolvido em seu espaço original de abstração (PÓLYA, 1945). Tal estratégia de resolução potencialmente resulta em reduções significativas no espaço de busca (MINSKY, 1963; NEWELL; SIMON, 1962; KNOBLOCK, 1991; STANGE; CEREDA; JOSÉ NETO, 2017).

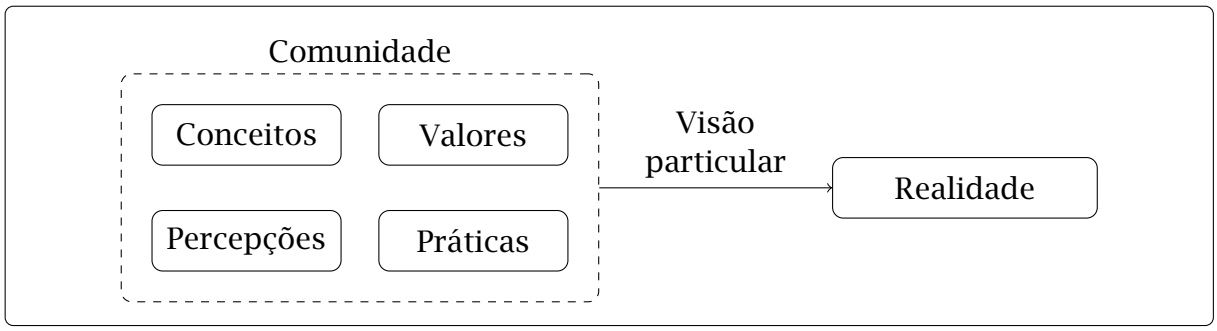
Definição 1 (abstração)**.** *Abstração* é um processo que extrai e preserva alguma propriedade de um artefato. Essa propriedade serve como base no qual o intelecto forma uma imagem ou conceito cognitivo de tal artefato (MARYNIARCZYK, 2000). Em linhas gerais, abstração é a supressão ou omissão proposital de detalhes de um artefato de forma a evidenciar mais claramente outros aspectos, detalhes ou estruturas (BUDD, 2002; KRAMER, 2007; KELLER, 2001). □

De acordo com Roberts (2009), o pensamento abstrato não é simplesmente um descritor da inteligência geral; é uma competência separada de outras habilidades cognitivas. Tal habilidade de uso lógico dos símbolos relacionados a conceitos abstratos foi inicialmente identificada e estudada por Piaget (1969) e mais recentemente descrita por Kramer (2007) e Roberts (2009) como elemento-chave na computação, ao viabilizar a capacidade de produção de modelos a partir de situações do mundo real. Escolher a abstração correta, entretanto, depende do que é essencial no momento, de acordo com níveis de detalhamento e granularidade (HAILPERIN; KAISER; KNIGHT, 1999; KORF, 1987; KNOBLOCK, 1989); cada nível de abstração tem sua própria definição e especificação (KELLER, 2001; BALABAN; BARZILAY; ELHADAD, 2002; ORLAREY et al., 1994).

Abstração, de acordo com Parnas (1971, 1972), está relacionada com o princípio de *ocultação da informação* (do inglês *information hiding*), no qual detalhes potencialmente mutáveis devem ser ocultados. A abstração é, portanto, um processo para identificar qual informação deve ser ocultada, de tal modo que as interfaces sejam definidas para revelar o mínimo possível sobre os detalhes internos (PARNAS; CLEMENTS; WEISS, 1985; PARNAS, 1978). Booch et al. (2007) discorrem sobre abstração e encapsulamento como conceitos complementares: o primeiro trata do comportamento observável de um artefato, enquanto o segundo (alcançado por meio de ocultação da informação) oculta os detalhes de um artefato que não contribuem para a observação de suas características essenciais.

Abstrações permitem formas diversas de observação sobre um mesmo artefato e sua representação; portanto, é possível obter modelos diferentes em sua especificação, mas que representam um mesmo artefato ou fenômeno (AHO; ULLMAN, 1995; KOHLBECKER, 1986). Tais modelos estão relacionados ao conjunto de conceitos, valores, percepções e práticas compartilhadas por uma comunidade, constituindo uma visão particular da realidade que é a base de organização da própria comunidade (KUHN, 1970; HARMAN, 1970; BARKER, 1992; CAPRA, 1996) (Figura 1.2).

Figura 1.2: Conjunto de conceitos, valores, percepções e práticas compartilhadas por uma comunidade constituem uma visão particular da realidade.



Fonte: autor.

Definição 2 (mecanismo de abstração)**.** Dá-se o nome de *mecanismo de abstração* à forma de representação e tratamento de abstrações (CAMPBELL, 1987). □

Exemplo 2 (mecanismo de abstração)**.** Na computação, existem estilos de programação que fornecem ao programador uma visão particular sobre a organização e execução de um programa (SEBESTA, 2013). Cada estilo oferece mecanismos de abstração tais como funções declarativas matemáticas e lógicas, procedimentos imperativos lineares e multiplexados, tipos de dados abstratos, polimorfismo de tipos, entre outros (SEBESTA, 2013; TURBAK; GIFFORD, 2008; HILFINGER, 1981). A Figura 1.3 ilustra duas representações do conceito de média aritmética utilizando diferentes mecanismos de abstração – a formulação matemática clássica e uma implementação utilizando a linguagem Python. □

Figura 1.3: Representações do conceito de média aritmética utilizando diferentes mecanismos de abstração.

Representação matemática ⇒

$$\bar{x} = \frac{x_1 + \ldots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Representação em Python ⇒

```
def mean(*x):
    return float(sum(x)) / len(x)
```

Fonte: autor.

Historicamente, mecanismos de abstração são identificados no projeto de linguagens de programação (HILFINGER, 1981). Nos primórdios da computação, programadores observaram que a programação em linguagem de máquina poderia tornar-se facilitada se uma forma abstrata de instrução fosse usada; assim, mnemônicos que descreviam instruções e posições de memória tornaram-se opções viáveis em comparação à manipulação direta de sequências de bits (KOHLBECKER, 1986).

Adicionalmente, programadores observaram que certas sequências de instruções apresentavam padrões de repetição; assim, foram propostas formas de abstração para expressar essa característica, de tal forma que a codificação de um algoritmo procurasse evitar a duplicação desnecessária de sequências bem definidas de instruções (KOHLBECKER, 1986; KOHLBECKER; WAND, 1987). Construtos iterativos foram, então, introduzidos nas linguagens de programação, como mecanismos de abstração que representassem, de forma algorítmica, a ocorrência da repetição de sequências de instruções (CAMPBELL, 1987).

Como consequência da busca por abstrações para expressar sequências de instruções que se repetiam, a introdução de procedimentos, funções e subrotinas permitiu que um programa fosse constituído de unidades independentes que realizassem tarefas específicas durante a execução. Cada unidade tornou-se, de certa forma, um programa autônomo a ser chamado, quando necessário, pelo programa principal (KOHLBECKER; WAND, 1987). Através de reuso de código, uma mesma unidade (seja ela um procedimento, função ou subrotina) pode simplificar tarefas de outros programas (NEIGHBORS, 1980).

Definição 3 (transparência referencial)**.** Uma unidade é dita *referencialmente transparente* se esta pode ser substituída por seu valor correspondente sem causar alterações no comportamento do programa (SØNDERGAARD; SESTOFT, 1990). Neste caso, a execução da unidade depende tão somente de seus parâmetros (isto é, não sofre interferência externa). Adicionalmente, uma unidade que é dita referencialmente transparente é determinística (QUINE; CHURCHLAND; FØLLESDAL, 2013), no sentido

de que esta sempre retorna os mesmos resultados para os mesmos parâmetros. □

Definição 4 (opacidade referencial)**.** Uma unidade é dita *referencialmente opaca* se esta não é referencialmente transparente (Definição 3) (FØLLESDAL, 2009; SØNDERGAARD; SESTOFT, 1990). Nesse caso, é impossível garantir que a substituição da unidade por seu resultado correspondente não causará alterações no comportamento do programa. □

Adicionalmente, chamadas recursivas de procedimentos ou funções contribuíram para a expressão de repetição de sequências de instruções com um forte apelo matemático (KOHLBECKER et al., 1986).

Definição 5 (unidade recursiva)**.** Uma unidade é dita *recursiva* quando esta, direta ou indiretamente, chama-se a si mesma, resolvendo partes da tarefa principal e combinando resultados intermediários até o término de todas as chamadas, retornando, enfim, o resultado final (CAMPBELL, 1987). □

Definição 6 (equivalência entre as formas iterativa e recursiva de um algoritmo)**.** As formas iterativa e recursiva de um algoritmo são *equivalentes*, de acordo com as afirmações a seguir:

i) Qualquer algoritmo que utilize sequências, instruções condicionais e recursão pode ser reduzido a um outro algoritmo que utilize sequências, instruções condicionais e, pelo menos, uma forma de iteração sem limite (do inglês *unbounded iteration*, que significa um laço de iteração sem limite definido *a priori*) (KESSLER; ANDERSON, 1986; AHO; ULLMAN, 1995; HOPCROFT; ULLMAN; MOTWANI, 2003; LEWIS; PAPADIMITRIOU, 2000). Complementarmente, de acordo com o teorema de Böhm-Jacopini, qualquer algoritmo pode ser expresso utilizando uma composição de sequências, instruções condicionais e formas de iteração (BÖHM; JACOPINI, 1966).

ii) Qualquer algoritmo que utilize sequências, instruções condicionais e, pelo menos, uma forma de iteração sem limite pode ser reduzido a um outro algoritmo que utilize sequências, instruções condicionais e recursão (KESSLER; ANDERSON, 1986; AHO; ULLMAN, 1995).

iii) Qualquer tarefa computável pode ser expressa através de sequências, instruções condicionais e, pelo menos, uma forma de iteração sem limite (KESSLER; ANDERSON, 1986; AHO; ULLMAN, 1995).

iv) Qualquer tarefa computável pode ser expressa através de sequências, instruções condicionais e recursão (KESSLER; ANDERSON, 1986; AHO; ULLMAN, 1995).

É importante destacar que utilizar recursão ou iteração não acrescenta poder computacional, apenas apresenta conveniências para a escrita de algoritmos que utilizam repetição (KESSLER; ANDERSON, 1986; AHO; ULLMAN, 1995). □

Exemplo 3 (equivalência entre as formas iterativa e recursiva de um algoritmo)**.** Considere as soluções iterativa e recursiva para cálculo do fatorial de um número, $n! = \prod_{i=1}^{n} i$, apresentadas no Algoritmo 1.1. Observe que a solução iterativa utiliza um laço para controle do fluxo de execução (linha 3-a), enquanto a correspondente recursiva invoca a si mesma (linha 5-b) até que a condição de parada seja atingida (linha 2-b). □

Algoritmo 1.1 Cálculo do fatorial de um número

```
1: function FACT1(n)
2:     r ← 1
3:     for i ← 1, n do
4:         r ← r · i
5:     end for
6:     return r
7: end function
```

(a) solução iterativa

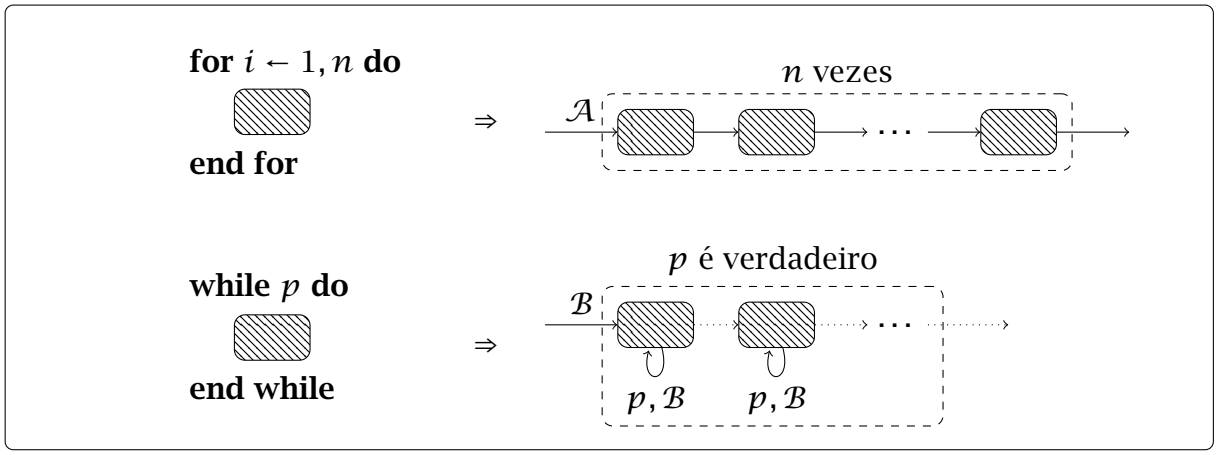
```
1: function FACT2(n)
2:     if n < 1 then
3:         return 1
4:     else
5:         return n · FACT2(n − 1)
6:     end if
7: end function
```

(b) solução recursiva

Uma forma alternativa para a expressão de repetição dá-se através de *adaptatividade*, utilizando ações adaptativas para acomodar novas sequências na instância do programa em execução (JOSÉ NETO, 1993), conforme ilustra a Figura 1.4. A automodificação, nesse caso, é utilizada para personalizar o caso particular de repetição (CEREDA; JOSÉ NETO, 2015b). De acordo com José Neto e Rocha (2000), o autômato adaptativo apresenta poder computacional equivalente ao da máquina de Turing, qualificando-o como mecanismo de abstração viável para expressar repetição de sequências de instruções, tal qual soluções iterativas e recursivas (CEREDA; JOSÉ NETO, 2015b). A adaptatividade tem como característica maior promover a extensão de formalismos já consolidados, aumentando seu poder de expressão (PISTORI, 2003).

Definição 7 (adaptatividade)**.** Dá-se o nome de *adaptatividade* à manifestação do fenômeno no qual um dispositivo modifica seu próprio comportamento espontaneamente, em resposta ao seu histórico de operação e aos dados de entrada (JOSÉ NETO, 1993, 1994, 2001). Diferentes dados de entrada submetidos a um mesmo dispositivo adaptativo podem resultar configurações de comportamento e de topologia totalmente distintos (JOSÉ NETO, 2007; CEREDA; JOSÉ NETO, 2015b; STANGE; CEREDA; JOSÉ NETO, 2017). □

Figura 1.4: Adaptatividade para expressão do fenômeno de repetição, utilizando ações adaptativas para acomodar novas sequências na instância do programa em execução.



Fonte: autor.

Alternativamente, abstrações podem ser representadas e tratadas através de *sistemas de reescrita* (BOICHUT et al., 2008; BERT; ECHAHED, 1995). Um sistema de reescrita (também chamado *sistema de redução* em sua forma geral) realiza transformações entre termos de acordo com um conjunto de *regras de substituição* (também chamadas *regras de reescrita*) (BAADER; NIPKOW, 1998). Aplicações de sistemas de reescrita incluem especificações de tipos abstratos de dados (propriedades de consistência) (RUSSELL, 1908; CHURCH, 1940), teoria de computabilidade (FRIEDMAN; SHEARD, 1995), decidibilidade de problemas de palavra (KNUTH; BENDIX, 1983), prova de teoremas (BERTOT; CASTÉRAN, 2004; HUTTON, 1994), implementações de linguagens de programação funcionais (KIKUCHI; KATAYAMA, 1991) e dedução automática (HSIANG et al., 1992; GALDINO, 2008).

Exemplo 4 (sistema de reescrita)**.** Considere o problema da lata de café (GRIES, 1981), no qual uma lata contém grãos de café de duas variedades, ◊ e ⧫, dispostos em alguma ordem. Representando o conteúdo da lata como uma sequência de grãos, por exemplo, ◊◊⧫⧫◊◊⧫⧫, as regras do jogo são apresentadas na Equação 1.1.

$$\begin{aligned} ⧫◊ &\rightarrow ⧫ \\ ◊⧫ &\rightarrow ⧫ \\ ⧫⧫ &\rightarrow ◊ \end{aligned} \tag{1.1}$$

O conjunto de regras apresentado na Equação 1.1 é um exemplo de sistema de reescrita. Cada regra descreve um movimento do jogo: as duas primeiras regras descartam um grão da variedade ◊ quando este constitui um par com um grão da variedade ⧫, enquanto a terceira regra substitui dois grãos adjacentes da variedade ⧫ por um grão da variedade ◊. A Equação 1.2 apresenta uma possível sequência

de movimentos, tal que os grãos de café sublinhados representam o movimento corrente.

$$\begin{array}{c} \Diamond\Diamond\blacklozenge\underline{\blacklozenge\Diamond}\Diamond\blacklozenge\blacklozenge \\ \Diamond\Diamond\underline{\blacklozenge\blacklozenge}\Diamond\blacklozenge\blacklozenge \\ \Diamond\Diamond\Diamond\underline{\Diamond\blacklozenge}\blacklozenge \\ \Diamond\Diamond\underline{\Diamond\blacklozenge}\blacklozenge \\ \underline{\Diamond\blacklozenge}\blacklozenge \\ \underline{\blacklozenge\blacklozenge} \\ \Diamond \end{array} \tag{1.2}$$

O objetivo do jogo é terminar com a menor quantidade possível de grãos. Observe que um número par (diferente de zero) de grãos de café da variedade $\blacklozenge$ resultará, após substituições sucessivas, em um grão de café da variedade $\Diamond$ (GRIES, 1981; DERSHOWITZ; JOUANNAUD, 1990). □

Definição 8 (sistema abstrato de redução)**.** Um *sistema abstrato de redução* R é definido como $R = (A, I)$, no qual A é um conjunto de elementos e I é uma sequência de relações binárias $\rightarrow_\alpha$ sobre A, também chamadas de relações de redução ou reescrita (KLOP, 1992). Um sistema abstrato de redução com apenas uma relação de redução é chamado *sistema de substituição* (STAPLES, 1975) ou *sistema de transformação* (JANTZEN, 1988). Se $a, b \in A$ e $(a, b) \in \rightarrow_\alpha$, tal relação de redução pode ser escrita como $a \rightarrow_\alpha b$ e b é dita uma α-redução (de um passo) de a. Analogamente, $a \rightarrow_\alpha^* b$, sendo $\rightarrow_\alpha^*$ o fecho transitivo e reflexivo de $\rightarrow_\alpha$, se existe uma sequência finita, potencialmente vazia, de passos de redução $a \equiv a_0 \rightarrow_\alpha a_1 \rightarrow_\alpha \ldots \rightarrow_\alpha a_n \equiv b$, no qual $\equiv$ denota a identidade de elementos de A (HUET; OPPEN, 1980). Um termo que não pode ser reescrito é dito estar na *forma normal*. □

A terminação de reescrita de termos (isto é, quando não há mais regras de substituição que possam ser aplicadas na sequência de termos) é, em geral, um problema indecidível (HERMANN; KIRCHNER; KIRCHNER, 1991). Entretanto, existem estudos no desenvolvimento de condições para garantia de terminação através de ordenações de redução (DERSHOWITZ, 1982, 1987; JOUANNAUD; LESCANNE; REINIG, 1982; LESCANNE, 1984; HUET, 1980). Um sistema de reescrita é dito *noetheriano* (isto é, possui garantias de terminar a reescrita de termos) se cada termo possui uma forma normal (HERMANN; KIRCHNER; KIRCHNER, 1991).

Tais mecanismos de abstração permitem que o usuário trabalhe com estruturas mais convenientes e, assim, represente modelos do mundo real na busca de soluções computacionais mais aderentes às suas necessidades. Em particular, sistemas de reescrita proveem subsídios necessários para transformações textuais (BERT; ECHAHED, 1995), tema principal desta tese.

1.1 JUSTIFICATIVA

Sistemas de reescrita, através de um formalismo simples e elegante, com poder computacional equivalente ao da máquina de Turing (DAUCHET, 1992; MARCHIORI, 1994; DERSHOWITZ; JOUANNAUD, 1990; BAADER; NIPKOW, 1998), constituem um conceito formal poderoso e um mecanismo eficiente para tratar problemas de *raciocínio equacional* (do inglês *equational reasoning*) (HERMANN; KIRCHNER; KIRCHNER, 1991; GALDINO, 2008; PLAISTED, 1993). Tais problemas incluem prova automática de teoremas, programação lógica, síntese de programas e computação simbólica (HERMANN; KIRCHNER; KIRCHNER, 1991; GALDINO, 2008; BENNINGHOFEN; KEMMERICH; RICHTER, 1987). Adicionalmente, sistemas de reescrita proveem o conceito de *abstração textual* (KOHLBECKER, 1986).

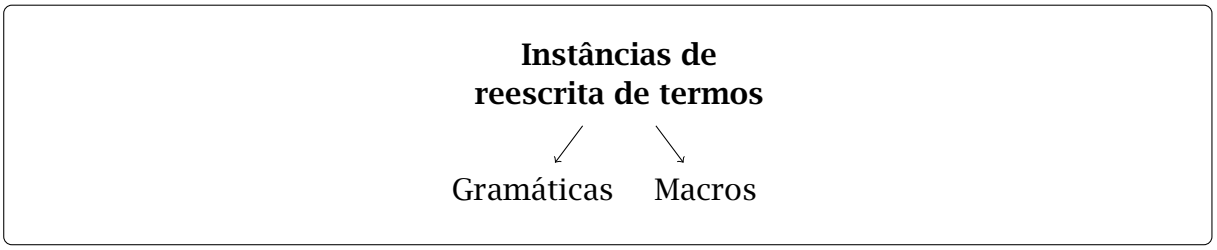
Definição 9 (abstração textual)**.** Dá-se o nome de *abstração textual* ao processo de abstração que identifica que determinados fragmentos de texto podem ser retirados de seus contextos. Esses fragmentos são comparados de tal forma que o resultado seja uma descrição de suas estruturas comuns. No texto original, tais fragmentos são, então, substituídos por instâncias de outros termos que os representem. A versão original, não abstraída, pode ser restaurada através da aplicação de transformações que realizam a reescrita de termos existentes para suas formas normais (KOHLBECKER, 1986). □

Na área de linguagens de programação, a abstração textual oferece conveniências tais como representação de código como dado (processo conhecido como *reificação*, que transforma um conceito abstrato em realidade concreta; no caso específico, a transformação produz um modelo de dados explícito (OLIVÉ, 2007)), sobrecarga ou polimorfismo da semântica da linguagem hospedeira através de virtualização, otimizações específicas de domínio (fundamentadas em conhecimento sobre o programa), representação do programa e seus contratos em tempo de compilação (verificação estática), e construção algorítmica de trechos de programa (KOHLBECKER, 1986; BURMAKO, 2012). Adicionalmente, é possível implementar construtos de programação orientada a aspectos (SKALSKI; MOSKAL; OLSZTA, 2004), padrões de projeto (SKALSKI, 2005), classes, *mixins* e *traits* (FLATT; FINDLER; FELLEISEN, 2006) e módulos (FLATT, 2010).

A partir do sistema abstrato de redução, considerado a forma mais abrangente de sistema de reescrita, derivam formalismos com propriedades particulares (KLOP, 1992). Em particular, gramáticas e macros constituem instâncias do fenômeno de reescrita de termos (Figura 1.5).

Definição 10 (símbolo)**.** Um *símbolo* (também chamado *átomo*) é uma representação gráfica única e indivisível em um determinado nível de granularidade de uma

Figura 1.5: Instâncias do fenômeno de reescrita de termos.



Fonte: autor.

entidade abstrata (RAMOS; JOSÉ NETO; VEGA, 2009; SIPSER, 2012). É importante destacar que o conceito de indivisibilidade é restrito ao sistema particular que se está estudando. □

Definição 11 (alfabeto)**.** Um *alfabeto* é definido como um conjunto finito e não-vazio de símbolos (HOPCROFT; ULLMAN; MOTWANI, 2003; RAMOS; JOSÉ NETO; VEGA, 2009). □

Definição 12 (cadeia)**.** Também chamada *texto*, uma *cadeia* é definida como uma sequência finita de símbolos de um alfabeto (SIPSER, 2012; HOPCROFT; ULLMAN; MOTWANI, 2003). □

Definição 13 (comprimento de uma cadeia)**.** O *comprimento de uma cadeia* α é um número natural n, $n \in \mathbb{N}$, que designa a quantidade de símbolos que a compõem (RAMOS; JOSÉ NETO; VEGA, 2009; LEWIS; PAPADIMITRIOU, 2000). Em geral, é denotado por $|\alpha| = n$. □

Definição 14 (cadeia vazia)**.** Uma cadeia α é dita *vazia* se esta possui zero ocorrências de símbolos, $|\alpha| = 0$ (HOPCROFT; ULLMAN; MOTWANI, 2003). □

Definição 15 (cadeia unitária)**.** Uma cadeia α é dita *unitária* se esta é formada por um único símbolo, $|\alpha| = 1$ (RAMOS; JOSÉ NETO; VEGA, 2009). □

Definição 16 (linguagem)**.** Uma *linguagem* é definida como um conjunto arbitrário, finito ou infinito, de cadeias específicas sobre um alfabeto Σ. Se $L \subseteq \Sigma^*$, então L é uma linguagem sobre Σ (HOPCROFT; ULLMAN; MOTWANI, 2003). □

Definição 17 (substituição)**.** Define-se *substituição* como uma função s que mapeia os elementos de um alfabeto Σ_1 em linguagens sobre um alfabeto Σ_2, tal que $s\colon \Sigma_1 \mapsto 2^{\Sigma_2^*}$ (RAMOS; JOSÉ NETO; VEGA, 2009). □

Definição 18 (homomorfismo)**.** Uma substituição h é dita um *homomorfismo* se $h\colon \Sigma_1 \mapsto \Sigma_2^*$, isto é, h é uma função que mapeia cada símbolo de um alfabeto Σ_1 em uma cadeia única contida em uma linguagem Σ_2^* (RAMOS; JOSÉ NETO; VEGA, 2009). □

Uma *gramática* constitui um sistema formal no qual é possível sintetizar, de forma exaustiva, através de regras de substituição, o conjunto das cadeias que compõem uma determinada linguagem (RAMOS; JOSÉ NETO; VEGA, 2009).

Definição 19 (gramática)**.** Uma *gramática* G é definida como $G = (V, \Sigma, P, S)$, tal que V é o vocabulário da gramática (correspondendo a um conjunto finito e não-vazio de símbolos), Σ é o alfabeto finito e não-vazio dos símbolos terminais, P é o conjunto finito e não-vazio das produções (também chamadas regras de substituição), e $S \in N$ é a raiz ou símbolo inicial da gramática. Define-se $N = V - \Sigma$ como sendo o conjunto dos símbolos não-terminais da gramática (RAMOS; JOSÉ NETO; VEGA, 2009). □

Definição 20 (produção gramatical)**.** Uma produção gramatical $p \in P$ obedece à forma geral $p = \alpha \rightarrow \beta$, com $\alpha \in V^*NV^*$, $\beta \in V^*$, e $\rightarrow$ é uma relação sobre os conjuntos V^*NV^* e V^*. Observe que $P = \{(\alpha, \beta) \mid (\alpha, \beta) \in V^*NV^* \times V^*\}$ (RAMOS; JOSÉ NETO; VEGA, 2009). □

Definição 21 (forma sentencial)**.** Dá-se o nome de *forma sentencial* a qualquer cadeia obtida pela aplicação recorrente das seguintes regras de substituição, a partir da raiz S da gramática (RAMOS; JOSÉ NETO; VEGA, 2009):

i) S é, por definição, uma forma sentencial.

ii) Seja $\alpha\rho\beta$ uma forma sentencial com α e β representando cadeias quaisquer de símbolos da gramática, e seja $\rho \rightarrow \gamma$ uma produção da gramática. A aplicação desta produção à forma sentencial, substituindo a particular ocorrência de ρ por γ, produz uma nova forma sentencial $\alpha\gamma\beta$. □

Definição 22 (derivação)**.** Dá-se o nome de *derivação* à aplicação de zero ou mais regras de substituição a um forma sentencial. Se $\alpha\rho\beta \Rightarrow_G \alpha\gamma\beta$, dá-se o nome de *derivação direta*, em que o índice G designa o fato de que a produção aplicada pertence ao conjunto de produções que define a gramática G (este índice pode ser omitido caso a gramática possa ser facilmente identificada). Uma sequência de zero ou mais derivações diretas, $\alpha \Rightarrow \beta \Rightarrow \ldots \Rightarrow \mu$ é chamada simplesmente *derivação*, e pode ser abreviada como $\alpha \Rightarrow^* \mu$. Derivações em que ocorre a aplicação de pelo menos uma produção à forma sentencial são denominadas *derivações não-triviais* e denotadas por $\alpha \Rightarrow^+ \mu$ (RAMOS; JOSÉ NETO; VEGA, 2009). □

Definição 23 (sentença)**.** Uma forma sentencial w é dita uma *sentença* se esta contiver apenas símbolos terminais e for obtida pela aplicação de uma derivação não-trivial à raiz S de uma gramática. Sua derivação denota-se por $S \Rightarrow^+ w$ (RAMOS; JOSÉ NETO; VEGA, 2009). □

Definição 24 (linguagem de uma gramática)**.** A *linguagem de uma gramática* G, denotada como $L(G)$, é o conjunto de todas as sentenças sobre o alfabeto Σ, tal que $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^+ w\}$ (HOPCROFT; ULLMAN; MOTWANI, 2003). □

Gramáticas não serão consideradas no escopo principal desta tese. Entretanto, o Apêndice E apresenta uma ferramenta para geração de sentenças a partir de gramáticas livres de contexto, com a possibilidade de inclusão de restrições positivas e negativas acerca das produções de tais gramáticas.

De acordo com a Figura 1.5, a segunda instância do fenômeno de reescrita de termos compreende macros. *Macro* é, em linhas gerais, uma abreviatura que representa um artefato em um determinado nível de abstração. Tal abstração determina qual a granularidade de exposição das propriedades do artefato, omitindo detalhes excessivos ou irrelevantes, ou evidenciando características consideradas importantes naquele específico nível de abstração (AHO; ULLMAN, 1995; KOHLBECKER, 1986). Na computação, uma macro constitui um padrão sintático que especifica uma transformação simbólica ou algorítmica sobre uma sequência de símbolos associada. Na descoberta de uma instância de tal padrão, este é substituído por sua representação (isto é, aplicação da transformação correspondente) na posição de ocorrência no texto. Observe que macros se assemelham ao conceito de símbolos não-terminais de uma gramática; entretanto, podem incluir capturas, reproduzir padrões sintáticos arbitrários e viabilizar interpretações ou cálculos (BURMAKO, 2012; LEAVENWORTH, 1966). Adicionalmente, macros podem ser vistas como procedimentos abertos (é o caso de linguagens de programação que utilizam tal conceito) (KOHLBECKER, 1986). O conceito de macro, no sentido utilizado neste texto, é definido formalmente no Capítulo 3.

É importante destacar que o conceito de macro é abrangente, com discussões em diversas áreas do conhecimento, incluindo linguística (HOENIGSWALD, 1965; BYBEE, 2010) e semiótica (ZEMANEK, 1966; TANAKA-ISHII, 2010). Nesta tese, por questão de simplicidade, o termo *macro* refere-se a sua vertente textual.

Definição 25 (macro textual)**.** Uma *macro textual* é definida como uma abreviatura aplicada em textos (também chamados cadeias ou sequências de símbolos), abstraindo fragmentos e os representando por construtos equivalentes (KOHLBECKER, 1986). □

Exemplo 5 (macro)**.** A Figura 1.6 apresenta um exemplo de utilização de uma macro chamada `AGE` em um código-fonte escrito na linguagem C. Observe que a macro `AGE` possui a cadeia `25` como sequência de substituição. Durante a fase de pré-processamento do código-fonte, toda e qualquer ocorrência de `AGE` é substituída por `25`; logo após o pré-processamento, o novo código-fonte é então submetido ao processo de compilação (KERNIGHAN; RITCHIE, 1988). □

Figura 1.6: Exemplo de utilização de uma macro em um código-fonte escrito na linguagem C.

Antes do pré-processamento:

```
#include <stdio.h>

#define AGE 25

int main(void) {
  printf("My age is %d.",
     AGE);
  return 0;
}
```

Após o pré-processamento:

```
#include <stdio.h>

int main(void) {
  printf("My age is %d.", 25);
  return 0;
}
```

Fonte: autor.

De acordo com Kohlbecker (1986), a comunidade de linguagens de programação tem preferido o termo *extensão sintática* em relação ao termo *macro*, ainda que estes representem conceitualmente a mesma compreensão. Entrentanto, é importante observar que nem sempre as macros acabam se manifestando em extensões de sintaxe. As transformações ocorrem por intermédio da manipulação de nós em árvores de sintaxe concreta (BRABRAND; SCHWARTZBACH, 2002).

Definição 26 (linguagem de programação extensível)**.** Uma linguagem de programação é dita *extensível* se esta apresenta recursos que permitam a definição de novas características na linguagem (ZINGARO, 2007; ZENGER, 2004; BROOKER; MORRIS, 1962). Tais características incluem novas notações ou operações, estruturas de controle novas ou modificadas, ou elementos provenientes de diferentes paradigmas de programação (SCHUMAN; JORRAND, 1970; STANDISH, 1975; CASTRO JÚNIOR, 2009; MCILROY, 1960; CEREDA; JOSÉ NETO, 2015a). □

A motivação para o desenvolvimento de linguagens de programação extensíveis surgiu da exigência e necessidade de se obter representações mais expressivas dos elementos componentes de uma linguagem, de modo a torná-la mais aderente às necessidades de seus usuários (SCHUMAN; JORRAND, 1970; BUCKLEY et al., 2005). A extensão de linguagens de programação permite que os novos construtos inseridos sejam transformados em construtos da linguagem base em fases subsequentes, oferecendo conveniências não previstas na linguagem base ao programador, incluindo representações mais claras ou concisas (LANDIN, 1964; FELLEISEN, 1991). Tais transformações são usualmente viabilizadas através de expansões de macros (HOARE, 2010; DHIMAN et al., 2013; HIGGINBOTHAM, 2015).

Definição 27 (transformação)**.** Uma *transformação* ρ sobre uma sequência s é definida como a aplicação de um sistema de reescrita de termos R sobre s, tal que $\rho(R, s) = s'$, no qual todos os elementos da sequência resultante s' estão na forma normal. □

Transformações textuais constituem um importante recurso para a busca de estruturas de representação mais convenientes em textos (KOHLBECKER, 1986; WOODSEND; LAPATA, 2014). Sistemas de reescrita de termos viabilizam tais transformações através de regras de substituição que determinam a correspondência entre elementos textuais (GMEINER; GRAMLICH, 2008). Busca-se, portanto, no escopo desta tese, a exposição de técnicas de projeto e aspectos de implementação de sistemas de reescrita de termos utilizando macros como mecanismos de abstração em transformações textuais.

1.2 OBJETIVOS

Dada a importância da disponibilização de estruturas de representação mais convenientes e aderentes às necessidades dos usuários, o objetivo principal desta tese é tratar da utilização de sistemas de reescrita de termos como mecanismos de abstração em transformações textuais. Para tal, técnicas de projeto e aspectos de implementação de tais sistemas são apresentados, com enfoque em macros. Objetivos adicionais incluem:

i) Identificar e resgatar, de forma sistemática, o conceito de macros na literatura e unificar a terminologia existente, com o propósito de elucidar e favorecer sua compreensão, bem como dirimir eventuais dúvidas advindas de inconsistências introduzidas ao longo do tempo.

ii) Apresentar um método de desenvolvimento de um expansor de macros de propósito geral através de uma estratificação em camadas representando níveis de abstração.

iii) Propor a utilização de motores de eventos como modelos de implementação para os níveis de abstração de um expansor de macros de propósito geral.

iv) Disponibilizar um conjunto de construtos elementares (primitivas) para definição de macros, controle de fluxo de expansão e operações básicas sobre alguns tipos de dados (tais como operações lógicas e aritméticas).

v) Definir um conjunto de primitivas adaptativas de inserção, remoção e consulta de padrões sintáticos em um texto, viabilizado de acordo com as ações adaptativas elementares propostas originalmente no trabalho de José Neto (1993).

Adicionalmente, resultados secundários significativos direta ou indiretamente relacionados aos conceitos principais contemplados nesta tese são apresentados nos Apêndices A a E.

1.3 ORGANIZAÇÃO

A organização desta tese é detalhada a seguir. O Capítulo 2 apresenta uma revisão dos conceitos necessários para o embasamento teórico do tema desenvolvido, contemplando definições da notação de Wirth, autômato de pilha estruturado e autômato adaptativo. O Capítulo 3 apresenta uma unificação da terminologia acerca de macros, fundamentada em um resgate histórico, e trata de técnicas de projeto e aspectos de implementação de um sistema de reescrita de termos utilizando macros como mecanismo de abstração em transformações textuais. O Capítulo 4 propõe um conjunto de construtos elementares para definição de macros, controle de fluxo de expansão, operações básicas sobre alguns tipos de dados e ações adaptativas de inserção, remoção e consulta, incluindo exemplos de uso. Finalmente, as conclusões são apresentadas no Capítulo 5, incluindo discussões, contribuições e recomendações para trabalhos futuros.

CAPÍTULO 2

CONCEITOS

> You want weapons? We're in a library. Books are the best weapon in the world. This room's the greatest arsenal we could have. Arm yourself!
>
> DOCTOR WHO

Este capítulo apresenta uma compilação dos conceitos necessários para o embasamento teórico do tema desenvolvido nesta tese. As seções seguintes contemplam definições da notação de Wirth, autômato de pilha estruturado e autômato adaptativo, de acordo com a literatura existente.

2.1 NOTAÇÃO DE WIRTH

Wirth (1977) apresentou uma metalinguagem para a descrição de linguagens de programação, com o objetivo de oferecer uma notação simplificada em relação às iniciativas existentes. Tal metalinguagem popularizou-se como *notação de Wirth*.

Definição 28 (notação de Wirth)**.** A *notação de Wirth* é uma metalinguagem para para descrição sintática de aproximações livres de contexto de linguagens de programação, contemplando as seguintes propriedades:

i) Existe uma distinção clara entre símbolos especiais (chamados *metassímbolos*), símbolos terminais e símbolos não-terminais. Os metassímbolos existentes são `=`, `.`, `(`, `)`, `[`, `]`, `{`, `}`, `|` e `"`. Um símbolo não-terminal é denotado por um identificador convencional, isto é, uma letra seguida de zero ou mais letras e dígitos (como uma definição de variável em uma linguagem de programação), enquanto o símbolo terminal é expresso por uma cadeia de símbolos entre aspas duplas.

ii) Não há restrição quanto à utilização de metassímbolos como símbolos da linguagem sendo descrita. Consequentemente, o metassímbolo `|`, por exemplo, difere do símbolo terminal `"|"`.

iii) A notação permite evitar o uso intenso de recursão para expressar repetições simples, disponibilizando para isso um construto para iteração explícita.

iv) Não há a necessidade da utilização de um símbolo explícito que represente a cadeia vazia (como ⟨empty⟩ na notação BNF ou a letra grega ϵ), em razão da existência de construtos que tratam dessa situação particular.

v) A notação é fundamentada no conjunto de símbolos ASCII (acrônimo para *American Standard Code for Information Interchange*, ou *Código Padrão Americano para o Intercâmbio de Informação* no vernáculo) (AMERICAN STANDARDS ASSOCIATION, 1963; MACKENZIE, 1980).

A metalinguagem proposta por Wirth é apresentada na Figura 2.1. Observe que as palavras `identifier`, `literal` e `character` denotam símbolos não-terminal, terminal e pertencente ao conjunto ASCII, respectivamente.

Figura 2.1: Notação de Wirth, em sua forma original (WIRTH, 1977), expressa utilizando a própria notação que a define.

```
syntax = { production } .
production = identifier "=" expression "." .
expression = term { "|" term } .
term = factor { factor } .
factor = identifier | literal | "(" expression ")" |
         "[" expression "]" | "{" expression "}" .
literal = """" character { character } """" .
```

Fonte: Wirth (1977).

De acordo com Wirth (1977), a repetição é denotada por chaves, de modo que `{ a }` represente ϵ `| a | aa | aaa |` ... (fecho de Kleene). Elementos opcionais são expressos através de colchetes, de modo que `[ a ]` represente `a` $\mid \epsilon$. Parênteses são utilizados para representar agrupamentos, de modo que `( a | b ) c` representa `ac | bc`. Os símbolos terminais são expressos entre aspas; caso as aspas apareçam como símbolos literais, estas são duplicadas. □

Algumas representações alternativas expressam aspas duplas literais como `"\""` ao invés de `""""`. Observe que o trabalho original de Wirth mantém a opção pela duplicação das aspas (WIRTH, 1977).

2.2 AUTÔMATO DE PILHA ESTRUTURADO

O autômato de pilha estruturado (JOSÉ NETO; MAGALHÃES, 1981; JOSÉ NETO, 1993) é um tipo de autômato de pilha formado por um conjunto de autômatos, também chamados *submáquinas*, a cada um dos quais cabe a tarefa de efetuar o reconhecimento de uma das diferentes classes de subcadeias que compõem uma cadeia de entrada em análise (RAMOS; JOSÉ NETO; VEGA, 2009). Diferentemente do autômato de

pilha tradicional, a pilha tem a finalidade exclusiva de armazenar estados de retorno a cada chamada de uma submáquina. As chamadas e retornos consistem em transferir o controle entre uma submáquina e outra; essa transição consiste em utilizar o símbolo de entrada apenas para a tomada de decisão do autômato em relação a qual transição executar, sendo o tal símbolo consumido na transição subsequente (JOSÉ NETO, 1993, 1994).

Definição 29 (autômato de pilha estruturado)**.** Um *autômato de pilha estruturado* M é definido como $M = (Q, A, \Sigma, \Gamma, P, Z_0, q_0, F)$, em que Q é um conjunto finito não-vazio de estados, A é um conjunto de submáquinas (introduzidas formalmente na Definição 30), Σ é o alfabeto do autômato, correspondendo ao conjunto finito não-vazio dos símbolos de entrada, Γ é o conjunto finito não-vazio dos símbolos de pilha, a serem armazenados na memória auxiliar do autômato, P é a relação de transição de estados, $q_0 \in Q$ é o estado inicial (da primeira submáquina), Z_0 é o símbolo marcador de pilha vazia, e $F \subseteq Q$ é o conjunto dos estados de aceitação do autômato (da primeira submáquina) (JOSÉ NETO; MAGALHÃES, 1981; JOSÉ NETO, 1993). □

Definição 30 (submáquina do autômato de pilha estruturado)**.** Uma *submáquina* $a_i \in A$ é definida como um autômato finito tradicional, da forma $a_i = (Q_i, \Sigma_i, P_i, q_{i,0}, F_i)$, no qual $Q_i \subseteq Q$ é o conjunto de estados de a_i, $\Sigma_i \subseteq \Sigma$ é o conjunto de símbolos de entrada de a_i, $q_{i,0} \in Q_i$ é o estado de entrada da submáquina a_i, $P_i \subseteq P$ é a relação de transição de estados de a_i, e $F_i \subseteq Q_i$ é o conjunto de estados de retorno da submáquina. □

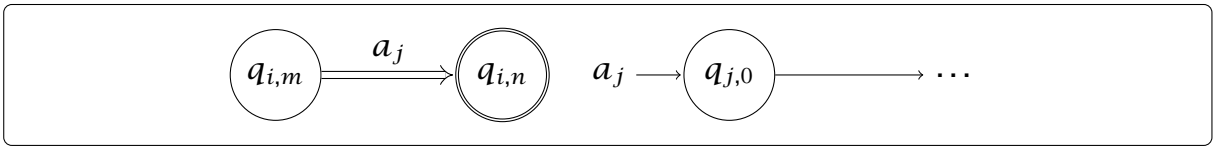
Definição 31 (relação de transição do autômato de pilha estruturado)**.** A *relação de transição* P é definida como $P \subseteq \Gamma \times Q \times \Sigma \times \Gamma \times Q$, na forma $(\gamma g, q, s\alpha) \rightarrow (\gamma g', q', \alpha)$, na qual q, q' são os estados corrente e de destino, respectivamente, s é o símbolo consumido, α é o restante da cadeia de entrada, g é o topo da pilha, g' é o novo topo da pilha, e γ é o restante da pilha. Uma configuração é um elemento de $Q \times \Sigma^* \times \Gamma^*$, e uma relação entre configurações sucessivas $\vdash$ é definida como:

- *Consumo de símbolo:* $(q, \sigma w, uv) \vdash (q', w, xv)$, com $q, q' \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $\sigma \in \Sigma \cup \{\epsilon\}$, $w \in \Sigma^*$, se σ foi consumido pelo autômato, $x = u$, e $(\gamma, q, \sigma\alpha) \rightarrow (\gamma, q', \alpha) \in P$.
- *Chamada de submáquina:* $(q, w, uv) \vdash (q', w, xv)$, com $q, q' \in Q$, $u \in \Gamma$, $v, x \in \Gamma^*$, $w \in \Sigma^*$, $x = pu$, com chamada da submáquina R, estado inicial q', retorno em p, e $(\gamma, q, \alpha) \rightarrow (\gamma p, q', \alpha) \in P$.
- *Retorno de submáquina:* $(q, w, uv) \vdash (q', w, v)$, com $q, q' \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $w \in \Sigma^*$, $u = q'$, com retorno de submáquina para q', e $(\gamma g, q, \alpha) \rightarrow (\gamma, g, \alpha) \in P$. □

Definição 32 (linguagem reconhecida por um autômato de pilha estruturado)**.** A *linguagem reconhecida por um autômato de pilha estruturado M* é dada por $L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (f, \epsilon, Z_0), f \in F\}$. □

Notação 1 (representação gráfica de chamada de submáquina)**.** Uma chamada de submáquina pode ser representada graficamente através de uma transição com linhas duplas, conforme ilustra a Figura 2.2. Observe que, a partir do estado $q_{i,m}$ da submáquina a_i, a execução é transferida para a submáquina a_j e o endereço referente ao estado de retorno $q_{i,n}$ é inserido no topo da pilha. No exemplo, o estado corrente passa a ser $q_{i,0}$, que é o estado inicial da submáquina a_j. □

Figura 2.2: Exemplo de chamada da submáquina a_j.



Fonte: autor.

É importante notar que, por uma questão de organização do modelo, admite-se que $a_i, a_j \in A$, $a_i = (Q_i, \Sigma_i, P_i, q_{i,0}, F_i)$, $a_j = (Q_j, \Sigma_j, P_j, q_{j,0}, F_j)$, $Q_i \cap Q_j = \emptyset$ e $P_i \cap P_j = \emptyset$, isto é, os conjuntos de estados e mapeamentos das submáquinas são disjuntos entre si.

Do ponto de vista de implementação, é possível automatizar a geração e execução de autômatos de pilha estruturados a partir de gramáticas escritas na notação de Wirth através das ferramentas de linha de comando `wsn2spa` e `spa2run` (CEREDA; JOSÉ NETO, 2017a). Alternativamente, a biblioteca AA4J pode ser utilizada para implementar autômatos de pilha estruturados utilizando diretamente a linguagem Java (CEREDA; JOSÉ NETO, 2016).

2.3 AUTÔMATO ADAPTATIVO

O autômato adaptativo (JOSÉ NETO, 1993) é um extensão do formalismo do autômato de pilha estruturado (Seção 2.2) que permite o reconhecimento de linguagens recursivamente enumeráveis. O termo *adaptativo*, neste contexto, pode ser definido como a capacidade de um dispositivo em alterar seu comportamento de forma espontânea. Um autômato adaptativo, portanto, tem como característica a possibilidade de alteração de sua própria topologia durante o processo de reconhecimento de uma dada cadeia (JOSÉ NETO, 1994). A alteração potencial do autômato ocorre através da aplicação de ações adaptativas, que lidam com situações esperadas, embora possivelmente não consideradas, detectadas na cadeia submetida para reconhecimento pelo autômato (JOSÉ NETO, 2001). Ao executar uma transição que contém

uma ação adaptativa associada, o autômato pode sofrer mudanças, obtendo-se nesse caso uma nova configuração.

Definição 33 (trajetória em um espaço de autômatos)**.** Para a aceitação de uma determinada cadeia, o autômato percorrerá uma *trajetória em um espaço de autômatos*. Haverá, portanto, um autômato E_0 que iniciará o reconhecimento de uma cadeia w, autômatos intermediários E_i que serão criados ao longo do reconhecimento, e um autômato final E_n que corresponde ao final do reconhecimento de w. Considerando $w = \alpha_0\alpha_1 \ldots \alpha_n$, o autômato M descreverá uma trajetória de autômatos $\langle E_0, \alpha_0\rangle \rightarrow \langle E_1, \alpha_1\rangle \rightarrow \ldots \rightarrow \langle E_n, \alpha_n\rangle$, no qual E_i representa um autômato correspondente à aceitação da subcadeia a_i. A disposição dos elementos $\langle E_i, \alpha_i\rangle$ em sequência denota a execução de transições adaptativas. □

Definição 34 (autômato adaptativo)**.** Um *autômato adaptativo* M é definido como $M = (Q, A, \Sigma, \Gamma, P, Z_0, q_0, F, E, \Phi)$, no qual Q é o conjunto finito não-vazio de estados, A é um conjunto de submáquinas, definidas a seguir, Σ é o alfabeto do autômato, correspondendo ao conjunto finito não-vazio dos símbolos de entrada, Γ é o conjunto finito não-vazio de símbolos de pilha, armazenados na memória auxiliar do autômato, P é a relação de transição de estados, $q_0 \in Q$ é o estado inicial (da primeira submáquina), Z_0 é o símbolo marcador de pilha vazia, $F \subseteq Q$ é o conjunto dos estados de aceitação do autômato (da primeira submáquina), E representa o modelo de autômato utilizado (inicialmente, o reconhecimento inicia-se no autômato E_0 da trajetória a ser descrita em um espaço de autômatos apresentado na Definição 33), e Φ é o conjunto das funções adaptativas, aplicáveis às transições (JOSÉ NETO, 1993, 1994). □

Definição 35 (submáquina do autômato adaptativo)**.** Uma *submáquina* $a_i \in A$ do autômato adaptativo M é definida como $a_i = (Q_i, \Sigma_i, P_i, q_{i,0}, F_i, \Phi_i)$, na qual $Q_i \subseteq Q$ é o conjunto de estados de a_i, $\Sigma_i \subseteq \Sigma$ é o conjunto de símbolos de entrada de a_i, $q_{i,0}$ é o estado de entrada da submáquina a_i, $P_i \subseteq P$ é a relação de transição de estados de a_i, $F_i \subseteq Q_i$ é o conjunto de estados de retorno da submáquina a_i, e $\Phi_i \subseteq \Phi$ é o conjunto de funções adaptativas aplicáveis. □

Definição 36 (relação de transição do autômato adaptativo)**.** A *relação de transição* P é definida como $P \subseteq \Gamma \times Q \times \Sigma \times (\Phi \cup \{\epsilon\}) \times \Gamma \times Q \times \Sigma \times (\Phi \times \{\epsilon\})$, na forma $(\gamma g, q, s\alpha), \mathcal{B} \rightarrow (\gamma g', q', s'\alpha), \mathcal{A}$, na qual q, q' são os estados corrente e de destino, respectivamente, s é o símbolo consumido, α é o restante da cadeia de entrada, g é o topo da pilha, g' é o novo topo da pilha, γ é o restante da pilha, $\mathcal{B}$ é uma função adaptativa anterior, e $\mathcal{A}$ é uma função adaptativa posterior. Uma configuração de um autômato adaptativo é um elemento de $E \times Q \times \Sigma^* \times \Gamma^*$, e uma relação entre configurações sucessivas $\vdash$ é definida como:

- *Consumo de símbolo:* $(E_i, q, \sigma w, uv) \vdash (E_i, q', \sigma' w, xv)$, com $q, q' \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $\sigma, \sigma' \in \Sigma \cup \{\epsilon\}$, $w \in \Sigma^*$, se σ foi consumido pelo autômato, $x = u$, e $(\gamma, q, \sigma\alpha) \rightarrow (\gamma, q', \sigma'\alpha) \in P$.

- *Chamada de submáquina:* $(E_i, q, w, uv) \vdash (E_i, q', w, xv)$, com $q, q' \in Q$, $u \in \Gamma$, $v, x \in \Gamma^*$, $w \in \Sigma^*$, $x = pu$, com chamada da submáquina R, estado inicial q', retorno em p, e $(\gamma, q, \alpha) \rightarrow (\gamma p, q', \alpha) \in P$.

- *Retorno de submáquina:* $(E_i, q, w, uv) \vdash (E_i, q', w, v)$, com $q, q' \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $w \in \Sigma^*$, $u = q'$, com retorno de submáquina para q', e $(\gamma g, q, \alpha) \rightarrow (\gamma, g, \alpha) \in P$.

- *Transição adaptativa:* $(E_i, q, \sigma w, uv) \vdash^{\mathcal{B},\mathcal{A}} (E_{i+2}, q', \sigma' w, u'v)$, indicando $(E_i, q, \sigma w, uv) \rightarrow^{\mathcal{B}} (E_{i+1}, q, \sigma w, uv) \vdash (E_{i+1}, q', \sigma' w, u'v) \rightarrow^{\mathcal{A}} (E_{i+2}, p, \sigma' w, u'v)$, com $q, q' \in Q$, $u, u' \in \Gamma$, $v \in \Gamma^*$, $w \in \Sigma^*$, se $(\gamma u, q, \sigma\alpha), \mathcal{B} \rightarrow (\gamma u, q', \sigma'\alpha), \mathcal{A} \in P$.

É importante destacar que, por questões puramente práticas, só se usam transições adaptativas em transições internas às submáquinas (com ou sem consumo de símbolo), excluindo-se portanto as chamadas e retornos de submáquinas. □

Definição 37 (variável e gerador)**.** Uma *variável* é um objeto definido no escopo da função adaptativa, denotado na forma ?⟨identificador⟩, utilizado para armazenar valores resultantes de ações adaptativas elementares de inspeção (no momento em que uma variável é preenchida, seu conteúdo fica protegido e não recebe novas alterações até o final da execução da função adaptativa). Um *gerador* é um tipo especial de variável, denotado na forma ⟨identificador⟩*, preenchida ao início da execução da função adaptativa com valores unívocos que referenciam estados recém-incorporados ao modelo do autômato. □

Definição 38 (chamada de função adaptativa)**.** Uma *chamada de função adaptativa* $\mathcal{F}_i \in \Phi$ assume a forma $\mathcal{F}_i(\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,m})$, na qual cada $\tau_{i,j}$ representa um argumento passado à função. A declaração de uma função adaptativa $\mathcal{F}_i$ com m parâmetros consiste de um cabeçalho da forma $\mathcal{F}_i(\Theta_{i,1}, \Theta_{i,2}, \ldots, \Theta_{i,m})$ e um corpo contendo nomes (uma lista de identificadores que representam objetos no escopo da função, incluindo variáveis e geradores) e ações (lista de ações adaptativas elementares precedida por uma ação adaptativa inicial e seguida por outra final). A inicialização de uma função adaptativa preenche os geradores e os parâmetros passados e, a seguir, as ações são efetivamente aplicadas (JOSÉ NETO, 1993, 1994). □

Definição 39 (ação adaptativa elementar)**.** Uma *ação adaptativa elementar* denota uma operação a ser realizada no conjunto de regras (relação de transição de estados

P). O formato de uma ação adaptativa elementar é apresentado na Equação 2.1.

$$(? \mid - \mid +)\langle \text{padrão} \rangle \tag{2.1}$$

São definidos três tipos de ações adaptativas elementares (JOSÉ NETO, 1993) que realizam testes no conjunto de regras ou modificam regras existentes (relação de transição de estados *P*), a saber:

- *ação adaptativa elementar de inspeção:* a ação não modifica o conjunto de regras, mas permite a inspeção deste para a verificação da presença de regras que obedeçam um certo padrão. É denotada por

  $$?(q, \sigma, \gamma), \mathcal{B}(p_{\mathcal{B},1} \ldots p_{\mathcal{B},n}) \rightarrow (q', \sigma', \gamma'), \mathcal{A}(p_{\mathcal{A},1} \ldots p_{\mathcal{A},k})$$

  na qual q, q' são os estados corrente e de destino, respectivamente, σ é o símbolo corrente da cadeia de entrada, σ' é o novo símbolo, γ é o topo corrente da pilha, γ' é o novo topo da pilha, $\mathcal{B}$ é a função adaptativa a ser executada antes da aplicação da transição, $p_{\mathcal{B},1} \ldots p_{\mathcal{B},n}$ é a lista de argumentos passados à função $\mathcal{B}$, $\mathcal{A}$ é a função adaptativa a ser executada após a aplicação da transição, e $p_{\mathcal{A},1} \ldots p_{\mathcal{A},k}$ é a lista de argumentos passados à função $\mathcal{A}$.

- *ação adaptativa elementar de remoção:* a ação remove regras que correspondem a um determinado padrão do conjunto corrente de regras. É denotada por

  $$-(q, \sigma, \gamma), \mathcal{B}(p_{\mathcal{B},1} \ldots p_{\mathcal{B},n}) \rightarrow (q', \sigma', \gamma'), \mathcal{A}(p_{\mathcal{A},1} \ldots p_{\mathcal{A},k})$$

  na qual q, q' são os estados corrente e de destino, respectivamente, σ é o símbolo corrente da cadeia de entrada, σ' é o novo símbolo, γ é o topo corrente da pilha, γ' é o novo topo da pilha, $\mathcal{B}$ é a função adaptativa a ser executada antes da aplicação da transição, $p_{\mathcal{B},1} \ldots p_{\mathcal{B},n}$ é a lista de argumentos passados à função $\mathcal{B}$, $\mathcal{A}$ é a função adaptativa a ser executada após a aplicação da transição, e $p_{\mathcal{A},1} \ldots p_{\mathcal{A},k}$ é a lista de argumentos passados à função $\mathcal{A}$.

- *ação adaptativa elementar de inclusão:* a ação insere uma regra que corresponde a um determinado padrão no conjunto corrente de regras. É denotada por

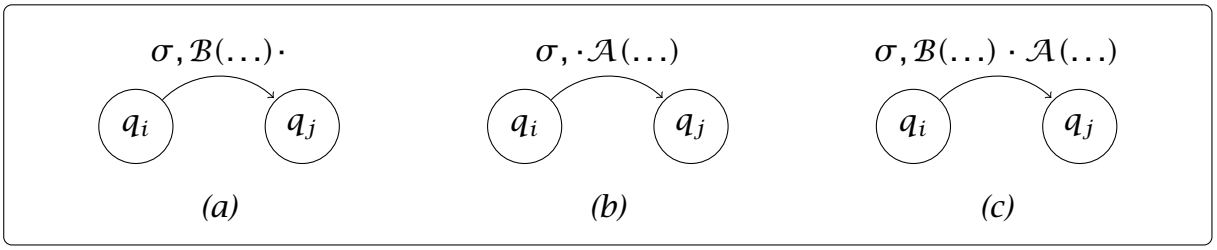
  $$+(q, \sigma, \gamma), \mathcal{B}(p_{\mathcal{B},1} \ldots p_{\mathcal{B},n}) \rightarrow (q', \sigma', \gamma'), \mathcal{A}(p_{\mathcal{A},1} \ldots p_{\mathcal{A},k})$$

  na qual q, q' são os estados corrente e de destino, respectivamente, σ é o símbolo corrente da cadeia de entrada, σ' é o novo símbolo, γ é o topo corrente da pilha, γ' é o novo topo da pilha, $\mathcal{B}$ é a função adaptativa a ser executada antes da aplicação da transição, $p_{\mathcal{B},1} \ldots p_{\mathcal{B},n}$ é a lista de argumentos passados à função $\mathcal{B}$, $\mathcal{A}$ é a função adaptativa a ser executada após a aplicação da transição, e $p_{\mathcal{A},1} \ldots p_{\mathcal{A},k}$ é a lista de argumentos passados à função $\mathcal{A}$.

É importante observar que a notação pode ser simplificada, omitindo elementos quando estes não forem relevantes ao contexto (por exemplo, a manipulação de pilha pode ser omitida caso a ação corrente não a utilize). □

Notação 2 (representação gráfica da situação de execução das funções adaptativas associadas a uma transição)**.** É possível representar graficamente a situação de execução das funções adaptativas associadas a uma transição, através da notação apresentada na Figura 2.3. Em *(a)*, a função adaptativa $\mathcal{B}$ é executada antes da transição. Em *(b)* a função adaptativa $\mathcal{A}$ é executada após a transição. Em *(c)*, as duas funções são executadas, sendo $\mathcal{B}$ antes da transição, e $\mathcal{A}$ após. Observe que as reticências indicam eventuais argumentos passados às funções adaptativas, e $\sigma \in \Sigma \cup \{\epsilon\}$. □

Figura 2.3: Notação gráfica utilizada para determinar a situação de execução das funções adaptativas associadas a uma transição.



Fonte: autor.

Definição 40 (linguagem reconhecida por um autômato adaptativo)**.** A *linguagem reconhecida por um autômato adaptativo M* é dada por $L(M) = \{w \in \Sigma^* \mid (E_0, q_0, w, Z_0) \vdash^* (E_n, f, \epsilon, Z_0), f \in F\}$. □

O autômato adaptativo viabiliza, através de uma estrutura estratificada, uma forma declarativa, unificada e hierárquica de representação para a definição de linguagens (JOSÉ NETO, 1994). Convém observar na literatura que, ao longo do tempo, os trabalhos desenvolvidos em torno deste assunto procuraram meios para tornar menos complexas as implementações das aplicações adaptativas, e uma das mais drásticas dessas simplificações eliminou o caráter declarativo das ações adaptativas elementares em favor de um estilo sequencial similar ao utilizado em linguagens de programação. Consequentemente, isso criou um formalismo que, apesar de ser muito mais prático e rápido, não é equivalente ao original a não ser nos casos particulares em que o paralelismo perdido não seja relevante (como acontece de fato em diversas aplicações úteis, porém muito menos exigentes do que o caso geral originalmente contemplado).

Do ponto de vista de implementação, é possível automatizar a geração e execução de autômatos adaptativos a partir de especificações no formato XML através da

ferramenta de linha de comando XML2AA (CEREDA; JOSÉ NETO, 2017c). Alternativamente, a biblioteca AA4J oferece subsídios para a implementação de autômatos adaptativos utilizando diretamente a linguagem Java (CEREDA; JOSÉ NETO, 2016). Para viabilizar a obtenção de implementações eficientes, Cereda e José Neto (2017b) disponibilizam um conjunto de métricas de instrumentação para dispositivos adaptativos dirigidos por regras e apresentam discussões acerca do fenômeno de automodificação.

CAPÍTULO 3

MACROS

> People think that Computer Science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.
>
> DONALD KNUTH

Macros compreendem uma instância do fenômeno de reescrita de termos. Uma macro (também chamada abreviatura) constitui um padrão sintático associado a uma transformação (AHO; ULLMAN, 1995; KOHLBECKER, 1986). Este capítulo apresenta uma unificação da terminologia acerca de macros, fundamentada em um resgate histórico, bem como suas propriedades, e introduz técnicas de projeto e aspectos de implementação de um sistema de reescrita de termos utilizando macros.

3.1 TERMINOLOGIA

Historicamente, propriedades de um sistema de reescrita de termos utilizando macros foram introduzidas de forma esparsa na literatura (KOHLBECKER, 1986; KOHLBECKER; WAND, 1987; MCILROY, 1960; WEISE; CREW, 1993; STRACHEY, 1965; MOOERS; DEUTSCH, 1965; MOOERS, 1966). Esta seção propõe uma unificação da terminologia resultante, com o propósito de elucidar e favorecer a compreensão de conceitos, bem como dirimir eventuais dúvidas causadas por inconsistências eventualmente introduzidas.

Definição 41 (macro)**.** Uma macro constitui um padrão sintático associado a uma transformação. Tal padrão pode ser classificado de acordo com as propriedades e características estruturais mais significativas deste, conforme as classes distintas de linguagens definidas na hierarquia de Chomsky (CHOMSKY, 1956, 1957). A transformação associada, representada por uma relação $T\colon \Sigma_1^* \mapsto \Sigma_2^*$, por sua vez, pode ser caracterizada como *simbólica* ou *algorítmica*. □

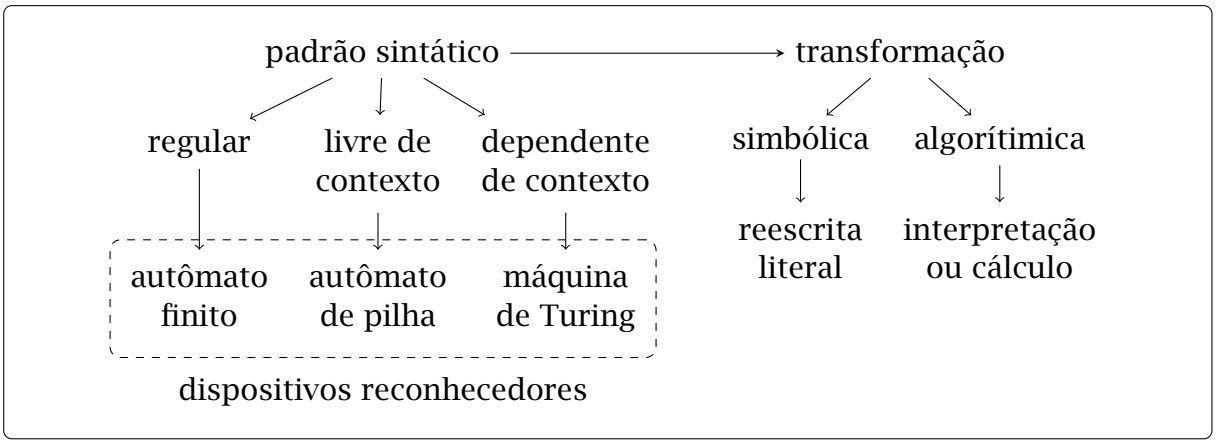
Definição 42 (transformação simbólica)**.** Uma transformação é dita *simbólica* se a sequência de substituição é reproduzida *ipsis litteris*, sem interpretação ou cálculo das unidades sintáticas. Tal transformação pode ainda admitir substituições literais

de estruturas sintáticas capturadas (reescrita simbólica de termos) (KOHLBECKER, 1986; BRABRAND; SCHWARTZBACH, 2002). □

Definição 43 (transformação algorítmica)**.** Uma transformação é dita *algorítmica* se esta não é simbólica, isto é, ocorre interpretação ou cálculo das unidades sintáticas da sequência de substituição (VIERA; SWIERSTRA, 2011, 2014; KERNIGHAN; RITCHIE, 1988; STALLMAN; WEINBERG, 2005; LEAVENWORTH, 1966). □

Uma síntese do conceito de macro é ilustrada na Figura 3.1, de acordo com a terminologia proposta (Definições 41, 42 e 43). Adicionalmente, são incluídas referências aos reconhecedores sintáticos mais simples que identifiquem sentenças pertencentes às classes de linguagens correspondentes.

Figura 3.1: Síntese do conceito de macro, de acordo com a terminologia proposta nas Definições 41, 42 e 43.



Fonte: autor.

A Tabela 3.1 introduz uma terminologia unificada e apresenta correspondências entre esta e a terminologia histórica, para fins de comparação. Os padrões sintáticos regular, livre de contexto e dependente de contexto mencionados correspondem aos tipos 3, 2 e 1 na hierarquia de Chomsky, respectivamente.

Exemplo 6 (macro com transformação simbólica na linguagem TEX)**.** A Figura 3.2 apresenta duas macros na linguagem TEX, `\goodmorning` e `\greeting`; a primeira possui um padrão sintático regular, enquanto a segunda contempla um padrão sintático livre de contexto com parâmetro. Ao encontrar uma instância de `\goodmorning`, o processador expande a macro, substituindo-a pela sequência `Good morning` na cadeia de saída. A macro `\greeting`, por sua vez, admite um parâmetro (indicado pelo seu índice, no exemplo, `#1`), delimitado entre chaves, conforme determina a especificação da linguagem TEX. Ao encontrar uma instância de `\greeting` no texto, o processador expande a macro e realiza a substituição de `#1` no corpo da macro pelo valor correspondente (no exemplo, `Paulo`), gerando a sequência `my name is Paulo`

Tabela 3.1: Correspondências entre as terminologias histórica e unificada.

Terminologia histórica	Terminologia unificada
macro	padrão sintático (PS)
parâmetro	captura de estrutura sintática
macro léxica[1]	PS potencialmente regular
macro sintática[2]	PS potencialmente livre de contexto
macro semântica[3]	PS potencialmente dependente de contexto
macro paramétrica	padrão sintático com parâmetros
estrutura de delimitação[4]	*não se aplica*
modo de aviso[5]	
marcador de aviso[5]	
modo livre[6]	

[1] Kohlbecker (1986), Brabrand e Schwartzbach (2002)
[2] Brabrand e Schwartzbach (2002)
[3] Viera e Swierstra (2011, 2014)
[4] Kohlbecker (1986), Mooers e Deutsch (1965)
[5] Strachey (1965), Mooers e Deutsch (1965), Mooers (1966), Hoare (2010)
[6] Brown (1967), Waite (1970), Kohlbecker (1986)

Fonte: autor.

na cadeia de saída. Os demais símbolos (no exemplo, sinais de pontuação) são reproduzidos *ipsis litteris*. □

Figura 3.2: Exemplo de macros na linguagem TEX.

Código-fonte TEX:

```
\def\goodmorning{Good morning}
\def\greeting#1{my name is #1}

\goodmorning, \greeting{Paulo}!
\bye
```

Cadeia de saída:

```
Good morning, my
name is Paulo!
```

Fonte: autor.

Exemplo 7 (macro com transformação simbólica na linguagem Clojure)**.** A Figura 3.3 apresenta um exemplo de ocorrência da macro `when` existente na linguagem Clojure (DHIMAN et al., 2013), contemplando um padrão livre de contexto, composta por uma combinação dos comandos `if` e `do`; o comando `macroexpand` realiza a expansão da macro passada como parâmetro com o objetivo de inspeção da transformação de substituição (HICKEY, 2008).

De acordo com a Figura 3.3, a macro sintática `when` é expandida quando o padrão sintático `"(" "when" BOOL-EXPR { STMT } ")"` é detectado na sequência de

Figura 3.3: Exemplo de ocorrência da macro sintática `when` existente na linguagem Clojure. O comando `macroexpand` realiza a expansão da macro.

Terminal interativo Clojure:	Resultado do comando:
`user=> (macroexpand '(when (pos? a) (println "positive")))`	`(if (pos? a) (do (println "positive")))`

Fonte: autor.

tokens. Observe que `BOOL-EXPR` representa uma regra sintática que define uma expressão lógica; da mesma forma, `{ STMT }` indica que a regra sintática de uma declaração de comando pode ocorrer zero ou mais vezes antes do término da macro sintática (as chaves representam o fecho de Kleene, de acordo com a notação de Wirth). Quando o padrão sintático completo é encontrado, ocorre a transformação deste na sequência de substituição. Observe que os símbolos não-terminais `BOOL-EXPR` e `{ STMT }` são capturados e substituídos posteriormente no corpo da macro `when` quando esta é transformada em um comando condicional `if` seguido do comando de avaliação `do`. A Tabela 3.2 apresenta a captura das regras sintáticas da ocorrência da macro `when` (Figura 3.3) e suas expressões correspondentes.

Tabela 3.2: Captura das regras sintáticas `BOOL-EXPR` e `{ STMT }` do padrão sintático referente à ocorrência da macro `when` (Figura 3.3).

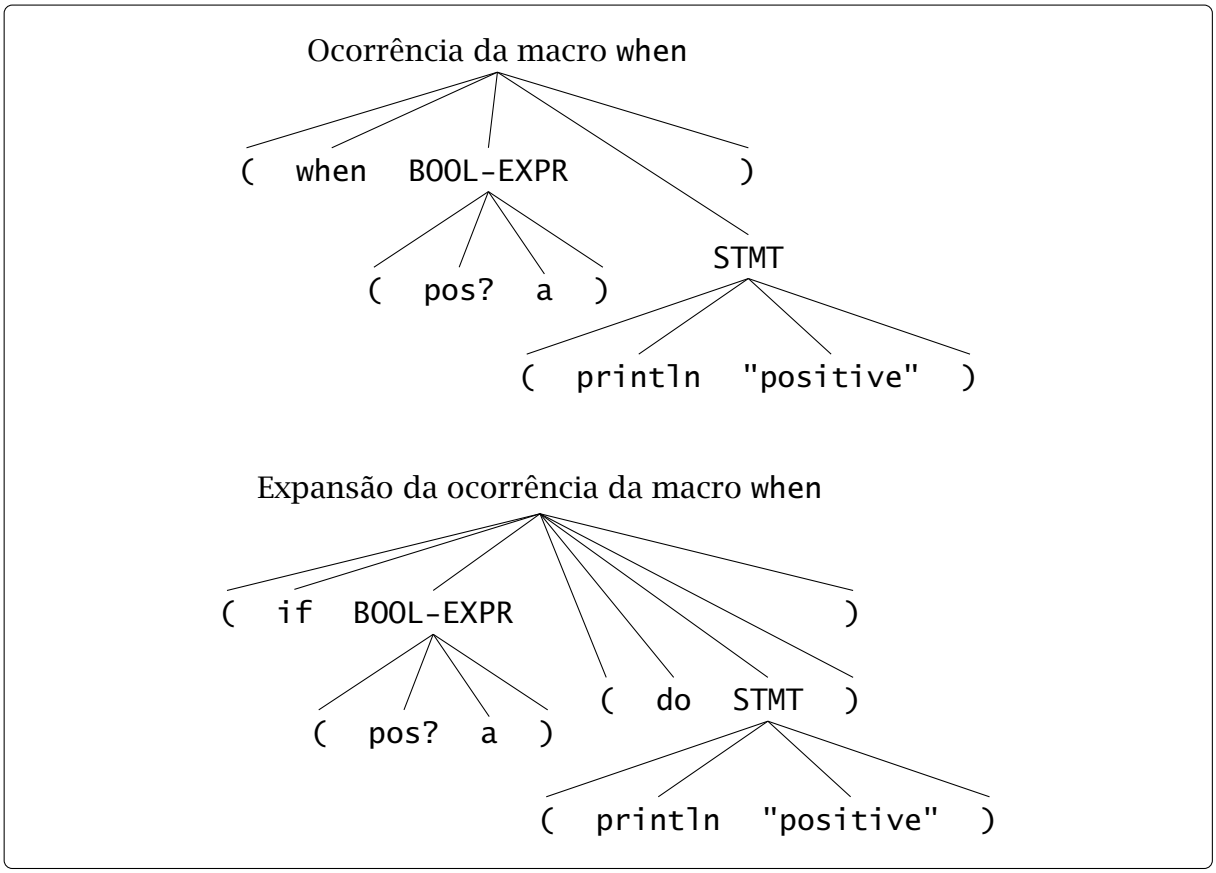
Capturas	Expressões
`BOOL-EXPR`	`(pos? a)`
`{ STMT }`	1 ⇒ `(println "positive")`

Fonte: autor.

A Figura 3.4 apresenta as árvores de sintaxe concreta da ocorrência da macro `when` e de sua transformação de substituição, de acordo com o exemplo da Figura 3.3; observe que as subárvores referentes às capturas são mantidas desde a ocorrência da macro sintática até sua transformação de substituição.

As Figuras 3.5 e 3.6 apresentam a definição da macro `when` através do mecanismo de extensão sintática `defmacro`, seu padrão sintático e a estrutura de transformação correspondente, respectivamente. O comando `defmacro`, conforme ilustra a Figura 3.5, define a macro `when` contendo dois parâmetros (a saber, `test` e `body`, especificados entre colchetes) e os comandos `if` e `do` para teste condicional e avaliação de expressões. De acordo com a Figura 3.6, os não-terminais capturados `BOOL-EXPR` e `STMT-LIST` (representando uma expressão lógica e uma lista de declarações, respectivamente) são substituídos *ipsis litteris* na estrutura de transformação por seus valores correspondentes (no caso, representados por subárvores). □

Figura 3.4: Árvores de sintaxe concreta da ocorrência da macro `when` e de sua transformação de substituição; símbolos terminais são representados nas folhas, enquanto símbolos não-terminais são representados nos nós interiores.



Fonte: autor.

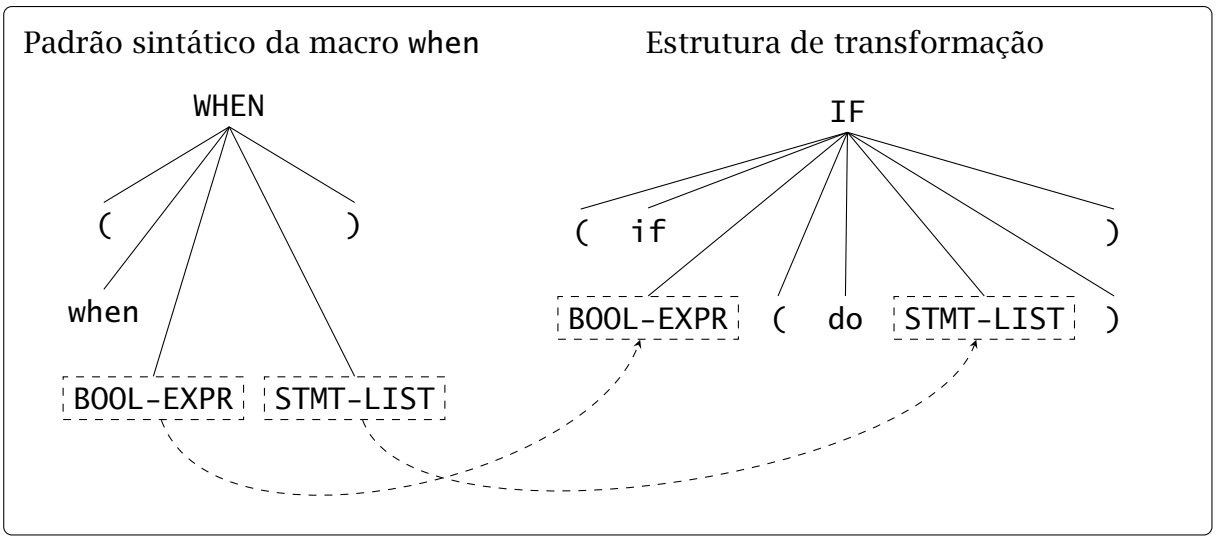
Figura 3.5: Definição da macro sintática `when` através do mecanismo de extensão sintática `defmacro` disponibilizado na linguagem Clojure (HIGGINBOTHAM, 2015).

```
(defmacro when
 "Evaluates test. If logical true, evaluates body in an
  implicit do."
 {:added "1.0"}
 [test & body]
 (list 'if test (cons 'do body)))
```

Fonte: autor.

Exemplo 8 (macro com transformação algorítmica na linguagem TEX)**.** A Figura 3.7 apresenta uma macro com transformação algorítmica na linguagem TEX, `\greet`. Observe que o corpo da macro possui o construto `\ifnum` que realiza uma operação lógica entre números inteiros (no exemplo, é verificado se o parâmetro informado é menor que 12) e desvia o fluxo de substituição para o bloco correspondente ao resul-

Figura 3.6: Padrão sintático da macro `when` existente na linguagem Clojure e sua estrutura de transformação correspondente.



Fonte: autor.

tado da operação (no exemplo, `morning` se a condição é verdadeira, ou `afternoon` caso contrário). A substituição, neste caso, deixa de ser puramente simbólica. Na ocorrência de `\greet{9}`, a condição lógica é satisfeita e `morning` é gerado na cadeia de saída. □

Figura 3.7: Exemplo de uma macro com transformação algorítmica na linguagem TEX.

Código-fonte TEX:

```
\def\greet#1{Good \ifnum#1<12
   morning\else afternoon\fi}

\greet{9}!
\bye
```

Cadeia de saída:

```
Good morning!
```

Fonte: autor.

Definição 44 (primitiva)**.** Dá-se o nome de *primitiva* (ou *comando primitivo*) ao construto elementar embutido em uma linguagem de macro (KOHLBECKER, 1986). Em geral, o conjunto de primitivas oferece recursos para manipulação de macros e transformações algorítmicas (BRABRAND; SCHWARTZBACH, 2002; BURMAKO, 2012; LEAVENWORTH, 1966). □

Exemplo 9 (primitiva)**.** Os construtos `\def` (Exemplo 6, Figura 3.2) e `\ifnum` (Exemplo 8, Figura 3.7) da linguagem TEX são primitivas para definição de novas macros

e desvio condicional de acordo com o resultado lógico de uma comparação entre números inteiros (no exemplo, é verificado se o parâmetro informado é menor que 12), respectivamente (KNUTH, 1986). □

Definição 45 (linguagem de macro de propósito geral)**.** Uma linguagem de macro projetada para integração com um sistema hospedeiro qualquer é classificada como de *propósito geral*. O processo de expansão, neste caso, pode ocorrer em um estágio de pré-processamento de alguma linguagem de programação ou simplesmente de forma autônoma, na geração de textos a partir de uma especificação (por exemplo, a linguagem GPM – *General Purpose Macrogenerator* (STRACHEY, 1965)). □

Definição 46 (linguagem de macro de propósito específico)**.** Uma linguagem de macro vinculada a um sistema hospedeiro distinto é classificada como de *propósito específico*. No caso de uma linguagem de programação, uma expansão de propósito específico estaria vinculada à linguagem hospedeira (por exemplo, suporte a macros nas linguagens LISP (STEELE, 1990) e Clojure (DHIMAN et al., 2013)) (KOHLBECKER, 1986). □

Definição 47 (ponto léxico)**.** Um *ponto léxico* (ou *posição léxica de ocorrência*) é denotado como um par ordenado (x, y), tal que $x, y \in \mathbb{N}$ representam, respectivamente, os índices de início e término do padrão sintático no texto sendo analisado. □

Definição 48 (escopo)**.** Dá-se o nome de *escopo* a uma região de um programa no qual associações (do inglês *bindings*) entre nomes e variáveis são válidas. □

Definição 49 (regras de escopo de um passo)**.** Uma linguagem de macro possui *regras de escopo de um passo* para definições de macros quando uma macro torna-se visível (isto é, adquire significado) na sequência de símbolos a partir de seu ponto léxico de definição em diante (por exemplo, como ocorre nas linguagens TEX (KNUTH, 1986), Dylan (SHALIT, 1996) e no expansor de macros da linguagem Scheme (ADAMS et al., 1998)); tal estratégia implica na importância de ordem na definição de macros (KOHLBECKER, 1986). □

Definição 50 (regras de escopo de dois passos)**.** Uma linguagem de macro pode empregar *regras de escopo de dois passos* para resolução das definições de macros (por exemplo, como ocorre nas linguagens MS2 (WEISE; CREW, 1993) e ⟨bigwig⟩ (BRABRAND; MØLLER; SCHWARTZBACH, 2002)); no primeiro passo, o expansor coleta todas as definições em um conjunto e, no segundo passo, todas as ocorrências de chamadas de macros são efetivamente expandidas (BURMAKO, 2012). □

Uma linguagem pode permitir definições de macros em escopos locais (por exemplo, a linguagem TEX (KNUTH, 1986)) ou simplesmente tratar toda e qualquer definição em nível global (por exemplo, as linguagens Dylan (SHALIT, 1996; BACHRACH;

PLAYFORD, 1999) e ⟨bigwig⟩ (BRABRAND; MØLLER; SCHWARTZBACH, 2002)) (BURMAKO, 2012). Macros definidas em escopos locais são desconhecidas em regiões externas a tais definições (KOHLBECKER, 1986).

Definição 51 (macro com recursão direta)**.** Uma macro é dita possuir *recursão direta* quando sua definição contém uma chamada a si mesma. Construtos semânticos que determinem uma condição de parada (por exemplo, através de operações aritméticas e estruturas condicionais) podem ser utilizados, mas nem sempre há garantias de término da recursão (KOHLBECKER, 1986; BRABRAND; SCHWARTZBACH, 2002). □

Definição 52 (macro com recursão indireta)**.** Uma macro possui *recursão indireta* quando uma nova instância da macro é criada como resultado de sua própria expansão e esta é processada em seguida; não há garantias de término tampouco (BRABRAND; SCHWARTZBACH, 2002). □

Algumas iniciativas admitem a utilização de múltiplas definições de uma mesma macro em conjunto com desdobramento de constantes (utilizando as noções de base de indução e passo indutivo) (STROUSTRUP, 2013).

Definição 53 (expansão ansiosa)**.** A estratégia de *expansão ansiosa* (*eager*) consiste em expandir todas as ocorrências de macros internas em tempo de definição da macro mais externa; tal característica permite que erros na sequência de transformação sejam detectados em tempo de definição, ainda que a macro jamais seja chamada (BRABRAND; SCHWARTZBACH, 2002; BRABRAND; MØLLER; SCHWARTZBACH, 2002). □

Definição 54 (expansão preguiçosa)**.** A estratégia de *expansão preguiçosa* (*lazy*) consiste em expandir ocorrências de macros internas somente no momento de chamada da macro mais externa; neste caso, não é possível detectar erros na sequência de transformação em tempo de definição, mas condiciona-se o processamento computacional para uso efetivo somente no momento de expansão da macro mais externa (BRABRAND; SCHWARTZBACH, 2002). □

Definição 55 (expansão mais interna)**.** A estratégia de *expansão mais interna* (*inner expansion*) consiste em expandir toda e qualquer ocorrência de macro existente em cada parâmetro da macro mais externa até que a sequência resultante não apresente mais ocorrências de macros, mesmo que o parâmetro em questão não seja utilizado na sequência de transformação (BRABRAND; SCHWARTZBACH, 2002); tal estratégia também é conhecida como *chamada por valor* (*call by value*) (BRABRAND; SCHWARTZBACH, 2002; KOHLBECKER, 1986). □

Definição 56 (expansão mais externa)**.** A estratégia de *expansão mais externa* (*outer expansion*) consiste em expandir ocorrências de macros internas tão somente no

momento de uso do parâmetro da macro mais externa na sequência de transformação (BRABRAND; SCHWARTZBACH, 2002); tal estratégia também é conhecida como *chamada por nome* (*call by name*) (BRABRAND; SCHWARTZBACH, 2002; KOHLBECKER, 1986). □

Definição 57 (pré-varredura de parâmetro)**.** A estratégia de *pré-varredura de parâmetro* (*argument prescan*) é semelhante à expansão mais interna, com a adição de um processamento da sequência de transformação em busca de ocorrências de novas macros que, eventualmente, foram produzidas como resultado da expansão de outras macros (BRABRAND; MØLLER; SCHWARTZBACH, 2002). □

Definição 58 (macro higiênica)**.** Dá-se o nome de *macro higiênica* à macro cuja transformação trata da renomeação de ligações (também chamadas de *bindings*) em modelos de código (KOHLBECKER et al., 1986). Algumas linguagens de macros utilizam redução alfa (uma forma básica de equivalência definida para termos do Cálculo lambda) para obter macros higiênicas (KOHLBECKER et al., 1986; BARENDREGT, 1985); outras requerem renomeação explícita e controle manual sobre a disciplina de escopo (KOHLBECKER et al., 1986). □

Definição 59 (transparência)**.** De acordo com Brabrand e Schwartzbach (2002), linguagens de macro são, em geral, projetadas de tal forma que fases subsequentes de análise da sequência de símbolos não tenham ciência da ocorrência de expansões de macros; a essa característica dá-se o nome de *transparência* (TRIANCE; LAYZELL, 1985). □

Como consequência da característica de transparência em uma linguagem de macro, não é possível obter um *rastreamento de erro* (*error trailing*); em uma linguagem de programação, erros léxicos ou sintáticos introduzidos durante o processo de expansão de macros no estágio de pré-processamento são propagados para fases subsequentes, sem a possibilidade de rastreamento do erro de volta às suas respectivas origens nas chamadas de macros (BRABRAND; SCHWARTZBACH, 2002). Algumas iniciativas foram propostas para tratar da ausência de rastreamento de erro; Andersen e Brabrand (2009) apresentam extensões sintáticas obtidas através de transformações utilizando catamorfismos construtivos que permitem a reconstrução dos construtos sintáticos originais (um catamorfismo denota um homomorfismo unívoco de uma álgebra inicial para uma outra álgebra). Entretanto, a representação sintática das extensões torna-se limitada por estabelecer restrições nos tipos de transformações realizadas (ANDERSEN; BRABRAND, 2009; ANDERSEN; BRABRAND; CHRISTIANSEN, 2013).

3.2 ESTRATÉGIAS DE EXPANSÃO

O expansor de macros pode adotar duas estratégias para a expansão de instâncias de macros presentes em um texto. Tais estratégias têm impacto direto sobre a aplicação da reescrita de termos e são descritas a seguir.

Definição 60 (expansão de um passo)**.** Dá-se o nome de *expansão de um passo* à estratégia de aplicar apenas um passo de reescrita na expansão de instâncias de macros. Eventualmente, outras macros podem resultar deste processo de reescrita, sendo reproduzidas literalmente (isto é, sem expansão) na cadeia de saída. Consequentemente, sendo $a \rightarrow b$ uma aplicação de reescrita, não há garantias de que b esteja na forma normal. □

Exemplo 10 (expansão de um passo)**.** O pré-processador do compilador GCC para a linguagem C utiliza a estratégia de expansão de um passo através da avaliação de uma regra de autorreferência (STALLMAN, 2017; STALLMAN; WEINBERG, 2005). Tal regra verifica se a macro a ser expandida possui recursão direta ou indireta (Definições 51 e 52, respectivamente) e, em caso positivo, interrompe expansões subsequentes. A Figura 3.8 apresenta um exemplo de aplicação da regra de autorreferência do compilador GCC. Na ocorrência da macro `foo`, o pré-processador a expande apenas uma vez, ignorando chamadas recursivas a si mesma. □

Figura 3.8: Exemplo de aplicação da regra de autorreferência do compilador GCC. Consequentemente, o pré-processador adota a estratégia de expansão de um passo e interrompe expansões subsequentes.

Antes do pré-processamento:

```
#include <stdio.h>

int main(void) {
  int foo = 5;
  #define foo (4 + foo)
  printf(foo);
  return 0;
}
```

Após o pré-processamento:

```
#include <stdio.h>

int main(void) {
  int foo = 5;
  printf(4 + foo);
  return 0;
}
```

Fonte: autor.

Definição 61 (expansão de múltiplos passos)**.** Dá-se o nome de *expansão de múltiplos passos* à estratégia de aplicar múltiplos passos de reescrita na expansão de instâncias de macros, tal que eventuais macros resultantes do processo de reescrita sejam expandidas sequencialmente até que cadeia de saída não contenha mais ocorrências.

Assim, sendo $a \rightarrow^* b$ a aplicação de múltiplas reescritas, o termo b estará na forma normal. □

Exemplo 11 (expansão de múltiplos passos)**.** O expansor de macros proposto por José Neto (1993, seção 4.4.8, página 92) utiliza a estratégia de expansão de múltiplos passos, tal que macros existentes na sequência de transformação sejam devidamente expandidas em tempo de expansão da macro mais externa. Entretanto, é importante destacar que o expansor em questão não oferece suporte a macros recursivas (JOSÉ NETO, 1993). A Figura 3.9 apresenta um exemplo de código na linguagem TEX, no qual a macro `\who` possui uma instância da macro `\name` em sua definição. O expansor, neste caso, adota a estratégia de expansão de múltiplos passos, retornando a cadeia `I am Paulo` como resultado de expansão da macro `who`, sem macros remanescentes. □

Figura 3.9: Exemplo de estratégia de expansão de múltiplos passos adotada pelo expansor de macros da linguagem TEX.

Código-fonte TEX:	Cadeia de saída:

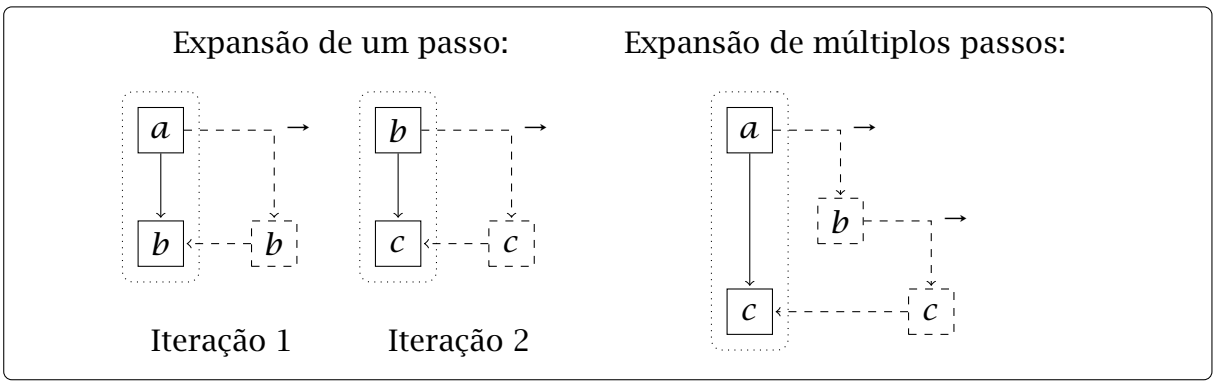
`\def\name{Paulo}`
`\def\who{I am \name}`
 	
`Hello, \who!`
`\bye`
	`Hello, I am Paulo!`

Fonte: autor.

As duas estratégias de expansão diferem quanto à aplicação subsequente da reescrita de termos na sequência de transformação. A expansão de um passo permite o controle, por parte do usuário, das macros resultantes de expansões anteriores e do nível de granularidade obtido a cada reescrita. Ao adotar a expansão de múltiplos passos, o usuário obtém uma cadeia de saída sem ocorrências de macros a partir de seu início até o ponto léxico de ocorrência da macro mais externa. A Figura 3.10 apresenta um diagrama das estratégias de expansão, considerando as regras de reescrita (a, b) e (b, c). Observe que a expansão de um passo requer duas iterações para obtenção do termo c em sua forma normal, enquanto a expansão de múltiplos passos retorna c como resultado obtido através de reescritas intermediárias diretamente a partir do termo a.

É importante destacar que a expansão de um passo sempre termina, ainda que existam termos na sequência de transformação resultante que não estejam em suas formas normais (justificativa pelo qual o compilador GCC adota tal estratégia em casos de dependência cíclica e recursão direta ou indireta). Comparativamente, a

Figura 3.10: Diagrama das estratégias de expansão, com $a \rightarrow^* c$, e c está na forma normal. Termos tracejados indicam reescritas intermediárias.



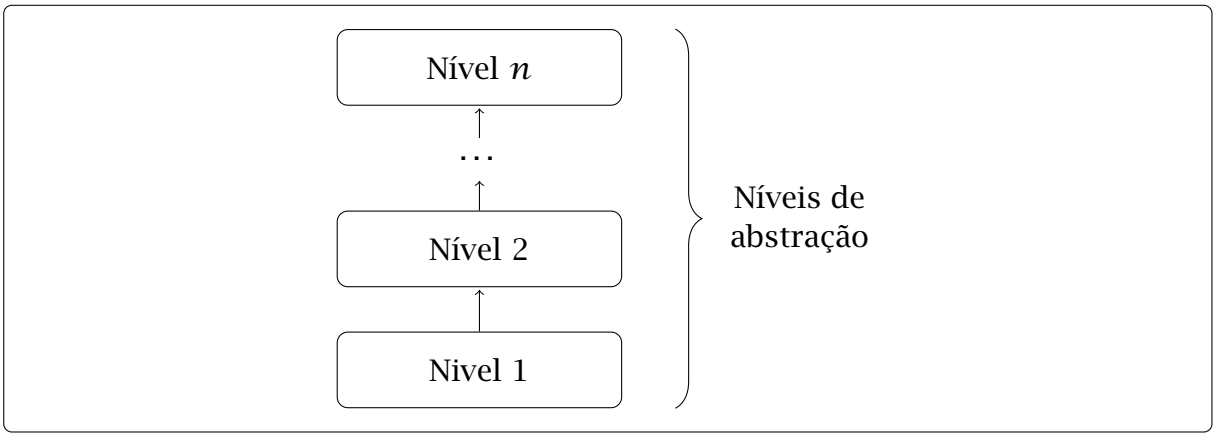
Fonte: autor.

expansão de múltiplos passos garante que todos os termos estejam em suas formas normais, sob o risco da reescrita seguir indefinidamente (por ocasião de macros recursivas sem construtos de controle, por exemplo). O Apêndice A apresenta discussões sobre a validação do conjunto de macros fundamentada em um sistema de reescrita noetheriano (isto é, que possui garantias de terminar a reescrita de termos) inspirado no trabalho de Bove e Arbilla (1991, 1992).

3.3 ESTRATIFICAÇÃO EM CAMADAS

Considere um método de desenvolvimento de um expansor de macros através de uma estratificação em camadas representando níveis de abstração. A Figura 3.11 apresenta uma estrutura de níveis de abstração dispostos hierarquicamente, tal que a camada de nível i estabeleça uma relação de ordem direta com as camadas de níveis $i-1$ e $i+1$ (anterior e seguinte, respectivamente). Por definição, a estrutura proposta inicia-se na camada de primeiro nível.

Figura 3.11: Estrutura do método de desenvolvimento em níveis de abstração.

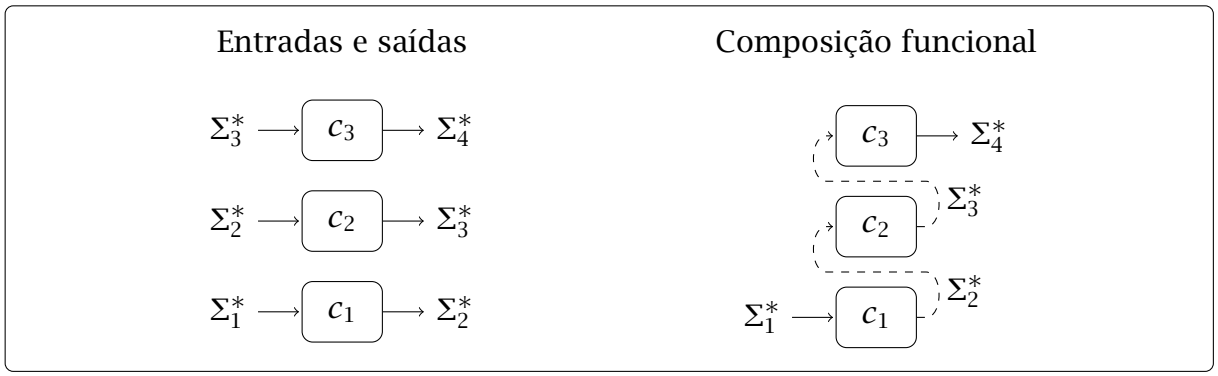


Fonte: autor.

De acordo com a Figura 3.11, o nível de abstração está associado ao tratamento dos dados na camada corrente, com posterior encaminhamento à camada seguinte. Do ponto de vista funcional, o expansor é formado pela composição de camadas ordenadas de acordo com suas abstrações, tal que cada camada resolva um subproblema em um espaço de abstração simplificado (KNOBLOCK, 1990). Cada camada da estrutura proposta, portanto, resolve parcialmente o problema original. Ao término do processamento, espera-se que a última camada (de nível n, mais alto) retorne o resultado final.

As camadas introduzidas na Figura 3.11 podem ser vistas como uma sequência de funções $c_i \colon \Sigma_i^* \mapsto \Sigma_{i+1}^*$, na qual o alfabeto de saída de uma camada constitui o alfabeto de entrada da camada imediatamente seguinte. A Figura 3.12 ilustra a composição funcional das camadas c_1, c_2 e c_3, tal que $c_3 \circ c_2 \circ c_1 \colon \Sigma_1^* \mapsto^* \Sigma_4^*$. É necessário que a hierarquia entre camadas seja obedecida (isto é, a partir da camada i, só será possível encaminhar dados à camada $i+1$, imediatamente seguinte).

Figura 3.12: Alfabetos de entrada e saída e composição funcional das camadas c_1, c_2 e c_3, tal que $c_3 \circ c_2 \circ c_1 \colon \Sigma_1^* \mapsto^* \Sigma_4^*$.

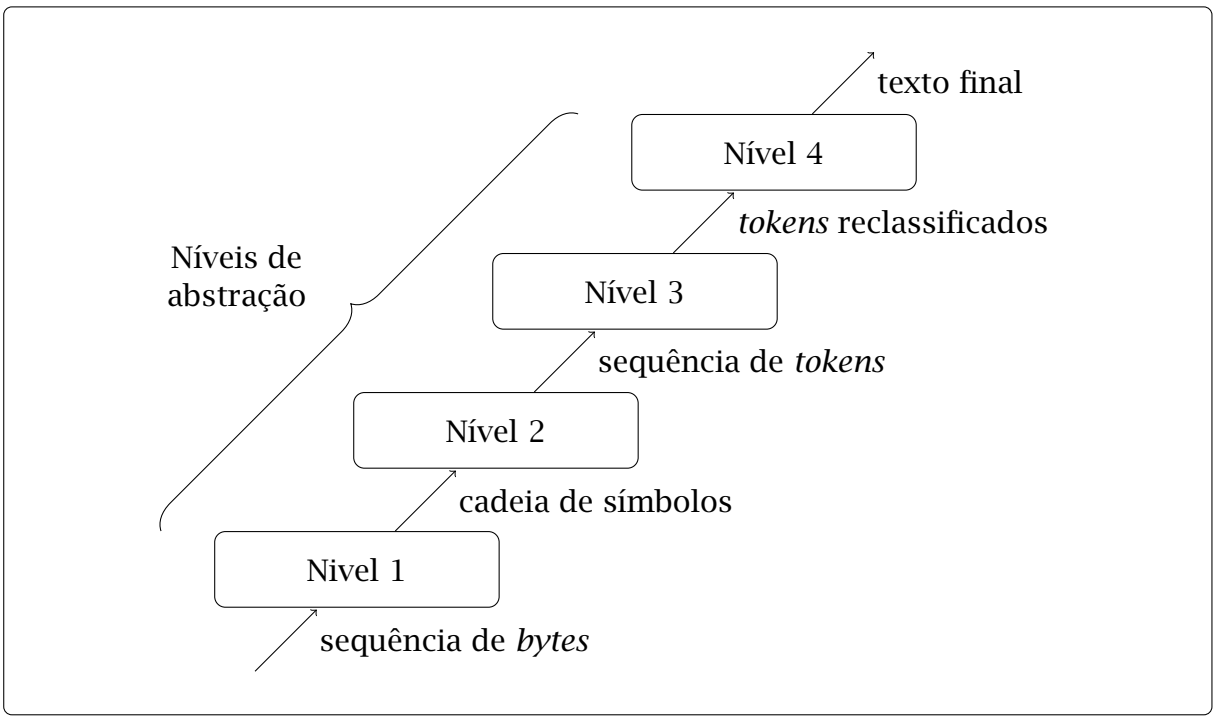


Fonte: autor.

Como exemplo, considere um expansor de macros com padrão sintático regular (tipo 3 na hierarquia de Chomsky), sem parâmetros e com transformações puramente simbólicas. A Figura 3.13 apresenta uma possível organização estratificada em níveis de abstração para o expansor proposto, detalhada a seguir.

De acordo com a Figura 3.13, a camada do nível 1 trata da abstração mais simplificada e converte uma sequência de *bytes* em uma cadeia de símbolos correspondente (por exemplo, utilizando a codificação ASCII). Tal cadeia é então submetida à camada do nível 2, que realiza a análise léxica e retorna uma sequência de *tokens* classificados (por exemplo, palavras, espaços em branco e sinais de pontuação). Na camada do nível 3, *tokens* são eventualmente reclassificados (caso estes estejam presentes em uma tabela contendo definições de macros) e encaminhados ao nível seguinte. Finalmente, os *tokens* são expandidos (caso sejam classificados como instâncias de macros no nível anterior) ou reproduzidos literalmente em uma cadeia de saída,

Figura 3.13: Níveis de abstração de um expansor de macros com padrão sintático regular, sem parâmetros e com transformações puramente simbólicas.



Fonte: autor.

retornada pela camada do nível 4. O expansor de macros então encerra o processamento.

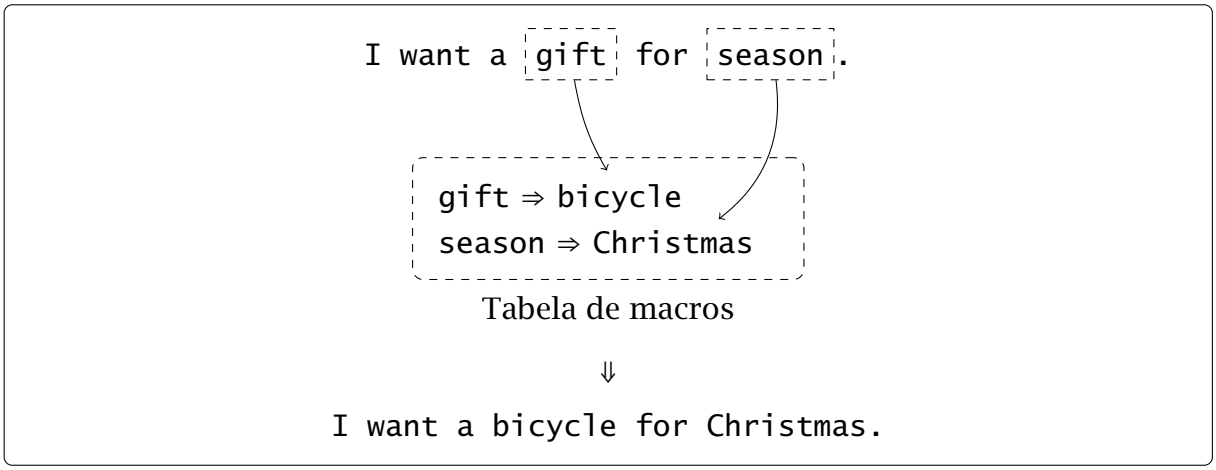
A estratificação em camadas representando níveis de abstração proporciona conveniências para o desenvolvimento de um expansor de macros, propiciando que o problema original seja particionado em subproblemas menores. Cada camada oferece subsídios para resolução de um subproblema em seu próprio espaço de abstração, de forma encapsulada. Tal característica confere independência à camada, tal que esta possa ser reutilizada posteriormente. Adicionalmente, o particionamento do problema original em subproblemas menores (e, consequentemente, espaços de abstração simplificados) reduz a complexidade global e favorece a depuração e eventuais melhorias.

É importante destacar que o método de desenvolvimento de um expansor de macros através de uma estraficação em camadas representando níveis de abstração, apresentada nesta seção, é abrangente o suficiente para ser utilizada em outros domínios, incluindo desenvolvimento de programas (engenharia de software) e projeto de sistemas de grande porte. A composição funcional das camadas agrega os resultados intermediários obtidos até que o problema original seja finalmente resolvido em seu espaço de abstração original.

3.4 ASPECTOS DE IMPLEMENTAÇÃO

De acordo com o método de desenvolvimento fundamentado na estratificação em camadas apresentada na Seção 3.3, é possível mapear tais níveis de abstração em procedimentos e funções computacionais equivalentes. Considere o expansor de macros com padrão sintático regular (tipo 3 na hierarquia de Chomsky), sem parâmetros e com transformações puramente sintáticas ilustrado na Figura 3.13. A Figura 3.14 apresenta um exemplo de expansão de macros em um texto, no qual instâncias de macros são transformadas em sequências de símbolos correspondentes de acordo com as regras de substituição definidas em uma tabela (transformações puramente simbólicas).

Figura 3.14: Exemplo de expansão de macros em um texto, de acordo com a especificação proposta do expansor da Figura 3.13.

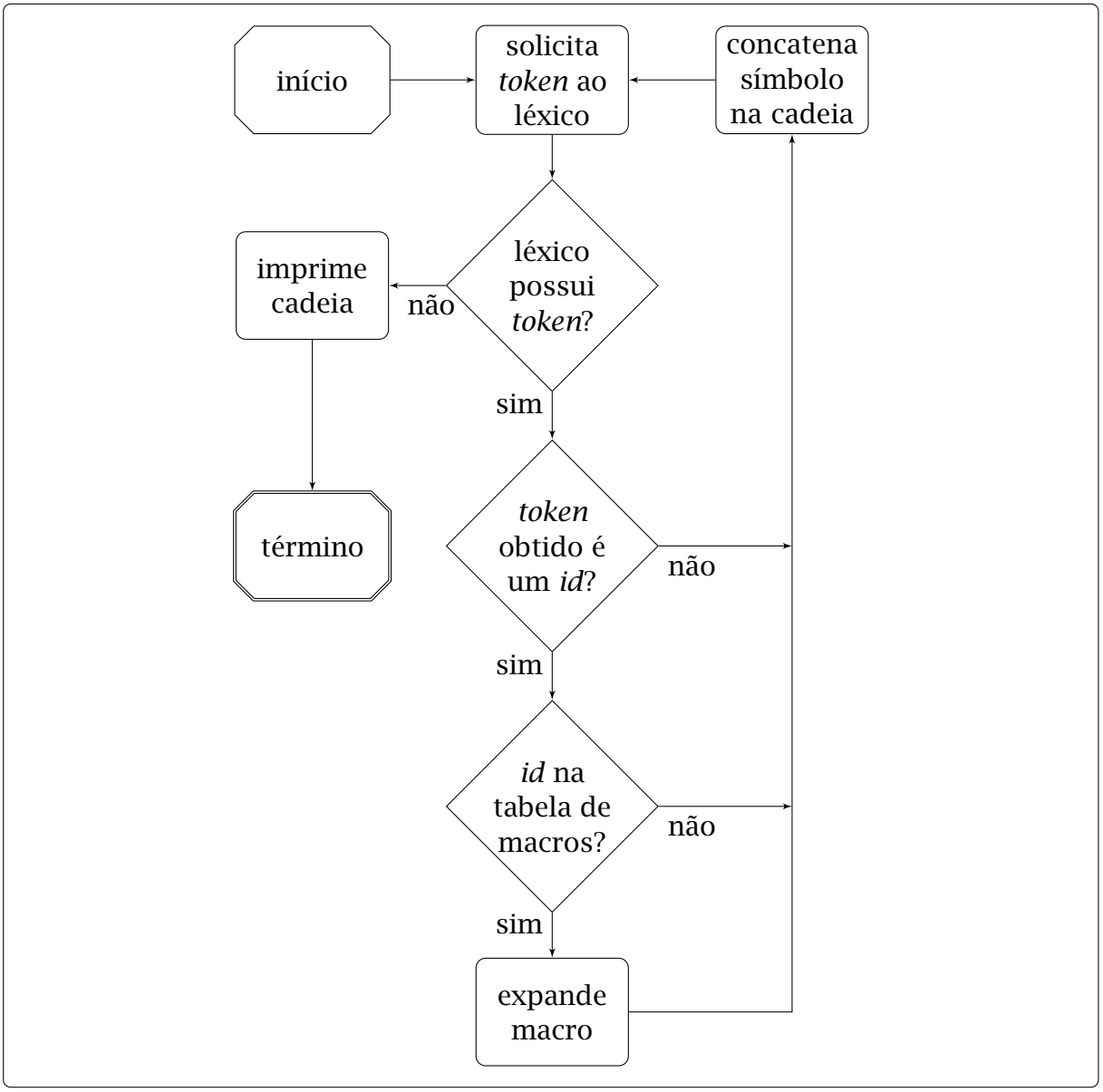


Fonte: autor.

Conforme ilustra a Figura 3.14, as ocorrências de `gift` e `season` são substituídas por `bicycle` e `Christmas`, respectivamente. A estratégia de identificação de fragmentos de texto como instâncias de macros consiste na análise de sequências de *tokens* e uma consulta posterior a uma tabela contendo as definições de macros. A Figura 3.15 apresenta o fluxograma contendo as etapas da expansão de macros simples em um texto. O analisador léxico (nível 2 da Figura 3.13) proposto possui duas classificações gramaticais (*id* e *other*) e será detalhado na Subseção 3.4.1.

De acordo com a Figura 3.15, *tokens* classificados como identificadores são consultados na tabela de macros. Caso o identificador exista na tabela de macros, este é expandido conforme a sequência de transformação correspondente e o resultado é concatenado na cadeia de saída. Eventualmente, a sequência de transformação pode conter referências a outras macros, sendo necessário optar por uma das estratégias apresentadas na Seção 3.2. Nas situações em que o *token* não seja um identificador ou se este não estiver presente na tabela de macros, o seu valor é simplesmente concatenado na cadeia de saída. É importante destacar que, por razões de legibilidade,

Figura 3.15: Fluxograma das etapas de expansão de macros simples em um texto.



Fonte: autor.

o nível 1 da Figura 3.13 não será contemplado, dado que as linguagens de programação de alto nível tratam cadeias de símbolos como tipos primitivos. Convém lembrar que os aspectos de implementação apresentados são apenas ilustrações de uma das inúmeras formas de como tais projetos podem ser viabilizados, e não modelos gerais a serem utilizados sempre.

3.4.1 ANÁLISE LÉXICA

O analisador léxico proposto para representar o nível 2 da Figura 3.13 foi projetado para categorizar uma sequência de símbolos de um texto em uma sequência de *tokens* de acordo com as classes gramaticais apresentadas na Tabela 3.3.

Tabela 3.3: Classes gramaticais do analisador léxico do expansor de macros da Figura 3.13.

Classe	Rótulo	Significado
c_1	*id*	identificador
c_2	*other*	demais símbolos

Fonte: autor.

Conforme ilustra a Tabela 3.3, a classe gramatical determina o tipo do *token* corrente e permite que o expansor realize verificações quanto ao tratamento adequado deste: *tokens* classificados como c_1 estão elegíveis para potenciais instâncias de macros (de acordo com uma consulta posterior a uma tabela contendo as definições de macros), enquanto *tokens* classificados como c_2 são reproduzidos *ipsis litteris* na cadeia de saída (Figura 3.15).

Considere as seguintes regras sintáticas das classes gramaticais introduzidas na Tabela 3.3: um identificador (classe gramatical c_1) é formado por uma letra seguida de zero ou mais ocorrências de uma letra ou dígito; complementarmente, os demais símbolos (classe gramatical c_2) têm comprimento unitário e não são letras (tais como dígitos, espaços em branco e sinais de pontuação). A Tabela 3.4 apresenta exemplos de cadeias pertencentes a tais classes gramaticais.

Tabela 3.4: Exemplos de cadeias pertencentes às classes gramaticais da Tabela 3.3.

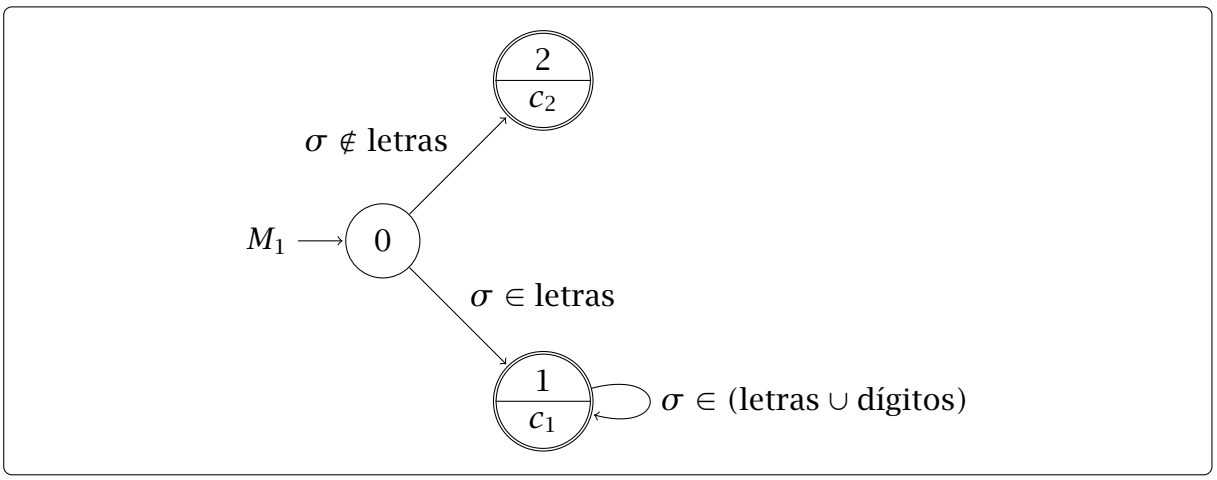
Cadeia	Classe gramatical
`abc`	c_1 (identificador)
`x1`	c_1 (identificador)
`1`	c_2 (demais símbolos)
`,`	c_2 (demais símbolos)
`␣`	c_2 (demais símbolos)

Fonte: autor.

A Figura 3.16 apresenta um autômato finito determinístico M_1 que descreve o analisador léxico proposto, de acordo com as regras sintáticas definidas anteriormente. O símbolo $\sigma \in \Sigma$ denota o símbolo corrente, e estados marcados com c_1 e c_2 indicam as classes gramaticais associadas.

O autômato finito determinístico M_1 da Figura 3.16 representa as regras sintáticas para classificação de *tokens* e reconhece cadeias pertencentes a uma linguagem $L = \{w \mid w \in \Sigma_1^*$, w é um identificador (classe gramatical c_1) ou um símbolo qualquer (classe gramatical c_2) $\}$. Observe que um alfabeto Σ qualquer pode ser particionado de acordo com um número arbitrário de propriedades de interesse (por exemplo, partições distinguindo símbolos imprimíveis e não-imprimíveis). Neste caso particular, Σ_1 = conjunto de símbolos ASCII. Sejam T a tabela de símbolos

Figura 3.16: Autômato finito determinístico M_1 que descreve o analisador léxico para o expansor de macros da Figura 3.13.



Fonte: autor.

ASCII (representada como uma relação binária) e t_i o símbolo indexado pela posição i, $t_i = s \mid (i,s) \in T$, $i \in \mathbb{N}, 0 \leq i \leq 127$, os subconjuntos auxiliares referenciados no autômato finito M_1 são definidos por extensão a seguir, nas Equações 3.1 e 3.2.

$$\text{letras} = \{t_i \mid 65 \leq i \leq 90 \vee 97 \leq i \leq 122\} \tag{3.1}$$

$$\text{dígitos} = \{t_i \mid 48 \leq i \leq 57\} \tag{3.2}$$

Sejam $w \in \Sigma_1^*$ uma cadeia, tal que $w \in L(M_1)$, e $C\colon \Sigma_1^* \mapsto \{c_1, c_2\}$ uma função que mapeia cadeias de Σ_1 em classes gramaticais c_1 e c_2. A cadeia w será classificada como identificador (classe gramatical c_1) se, e somente se, o autômato finito M_1, após o término do reconhecimento de w, referenciar 1 como estado corrente, $C(w) = c_1 \iff (0,w) \vdash^* (1,\epsilon)$. Analogamente, a cadeia w será classificada como um símbolo qualquer (classe gramatical c_2) se, e somente se, após o término do reconhecimento de w, o autômato finito M_1 referenciar 2 como estado corrente, $C(w) = c_2 \iff (0,w) \vdash^* (2,\epsilon)$. O Apêndice B apresenta discussões sobre a geração automática de autômatos com estados anotados através de uma proposta de extensão para a notação de Wirth tradicional.

Como modelo de implementação, propõe-se no escopo desta tese o uso de um *motor de eventos*. Tal modelo pode ser utilizado como ponto de partida na implementação de aplicações fundamentadas na simulação dirigida por estímulos (chamados *eventos*) e respostas correspondentes (chamadas *ações*). Essa técnica é abrangente e mostra-se adequada para tratar ocorrências de interesse ordenadas sequencialmente, como fluxo de símbolos e *tokens* em um expansor de macros. Assim, para o analisador léxico proposto, considere o motor de eventos ME_1 ilustrado na Figura 3.17. Dada uma sequência de símbolos (eventos de entrada), ME_1 dispara ações correspondentes a cada símbolo, transformando-os em uma sequência de *to-*

kens (eventos de saída). No exemplo, o texto da Figura 3.14 é interpretado como uma sequência de 25 símbolos ASCII e submetido ao motor de eventos, disparando ações correspondentes à análise léxica (por exemplo, através do autômato finito determinístico M_1 da Figura 3.16); como resultado, uma sequência de 12 *tokens* é gerada como saída do motor de eventos. Observe que cada *token* da sequência possui uma classe gramatical (definida na Tabela 3.3) e o valor associado. O Apêndice C apresenta uma ferramenta para geração de motores de eventos.

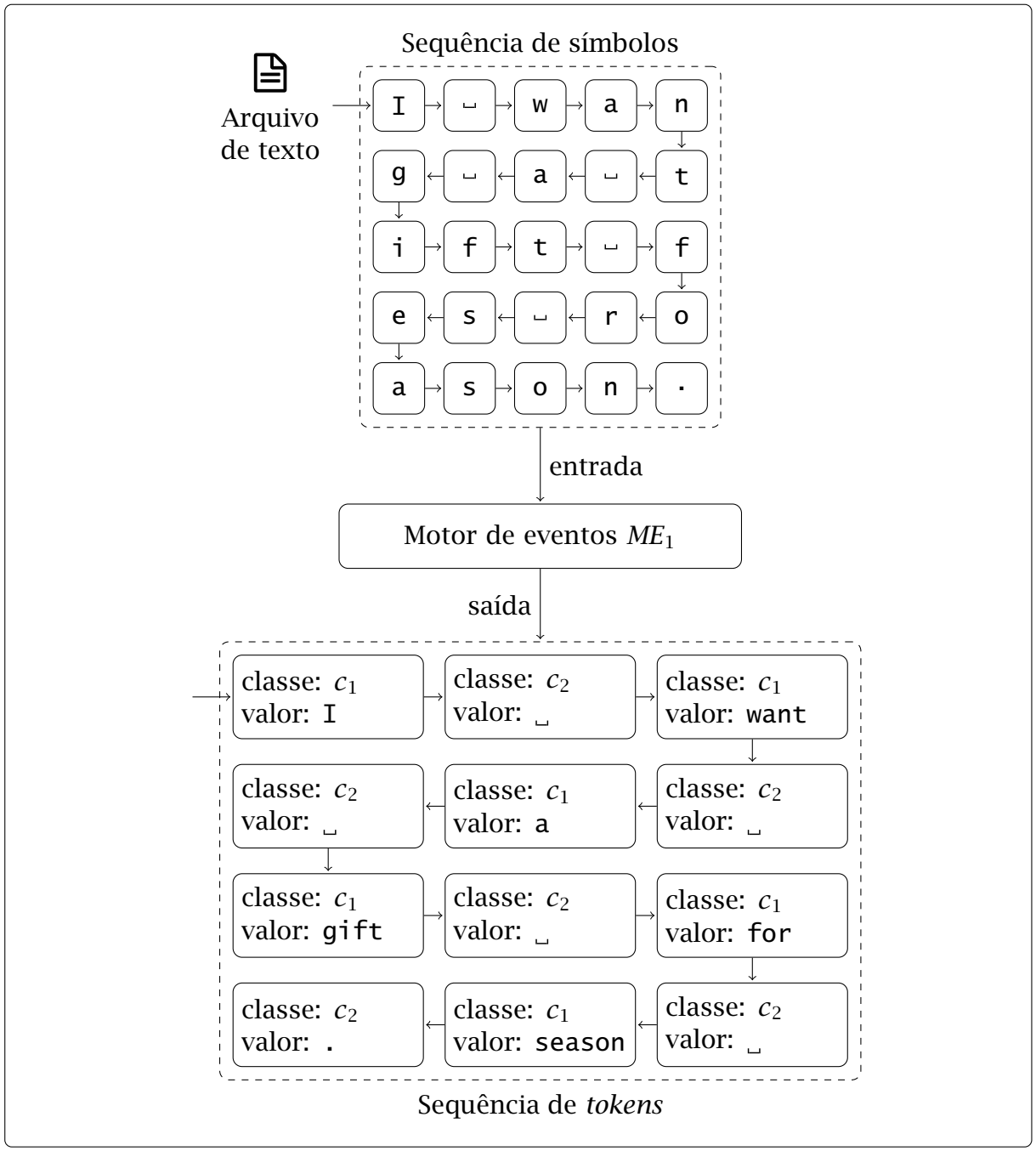
Considere uma implementação do analisador léxico proposto utilizando a linguagem de programação Lua (IERUSALIMSCHY, 2016), detalhada a seguir. A linguagem Lua foi escolhida em virtude de sua simplicidade, clareza e eficiência. A Figura 3.18 apresenta três funções auxiliares para a definição de *tokens* e a manipulação de cadeias de símbolos; tais funções serão referenciadas ao longo do texto.

De acordo com a Figura 3.18, a função `token` retorna uma tabela representando um *token* e contendo o valor e classe gramatical fornecidos como parâmetros (`a` e `b`, respectivamente). É importante destacar que o tipo *tabela* retornado por `token` é a única estrutura de dados definida nativamente na linguagem Lua; entretanto, tal tipo é expressivo o suficiente para representar outras estruturas de dados (IERUSALIMSCHY, 2016; JUNG; BROWN, 2007) e será utilizado ao longo da implementação proposta. A função `head` retorna o primeiro símbolo da cadeia fornecida como parâmetro (a saber, `s`); complementarmente, a função `tail` retorna o restante da cadeia a partir de uma posição inteira positiva, ambos fornecidos como parâmetros (`s` e `i`, respectivamente). Observe que a linguagem Lua adota o valor inteiro positivo 1 como base de indexação padrão (assim como Fortran, Smalltalk, Mathematica e Julia) (IERUSALIMSCHY; FIGUEIREDO; CELES FILHO, 2007). A Figura 3.19 apresenta a função `take` que efetivamente implementa o analisador léxico proposto; tal função recebe uma cadeia de símbolos como parâmetro (a saber, `s`) e retorna uma tupla contendo o *token* obtido e o restante da cadeia para uso posterior. Observe que o fornecimento de *tokens* ocorre sob demanda, isto é, a função `take` retorna somente um *token* por vez; chamadas subsequentes da função sobre o restante da cadeia de símbolos (até que esta torne-se vazia) garantem a obtenção da sequência de *tokens* desejada.

A função `take`, conforme ilustra a Figura 3.19, é implementada utilizando o recurso de *casamento de padrões* (do inglês *pattern matching*) disponibilizado pela linguagem Lua (IERUSALIMSCHY, 2016; JUNG; BROWN, 2007). É importante destacar que os padrões da linguagem Lua (do inglês *Lua patterns*) são mais restritos que as expressões regulares disponibilizadas em linguagens de programação como Java e Perl (IERUSALIMSCHY; FIGUEIREDO; CELES FILHO, 1996); entretanto, tais padrões são expressivos o suficiente para a implementação do analisador léxico proposto.

Exemplo 12 (execução do analisador léxico da Subseção 3.4.1)**.** A Figura 3.20 apre-

Figura 3.17: Motor de eventos ME_1 como modelo de implementação para o analisador léxico proposto (nível 2 da Figura 3.13). A sequência de símbolos utiliza o texto da Figura 3.14.



Fonte: autor.

Figura 3.18: Funções auxiliares para a definição de *tokens* e a manipulação de cadeias de símbolos.

```
local function token(a, b)
  return { value = a, class = b }
end

local function head(s)
  return string.sub(s, 1, 1)
end

local function tail(s, i)
  return string.sub(s, i)
end
```

Fonte: autor.

Figura 3.19: Implementação do analisador léxico proposto na Subseção 3.4.1.

```
local function take(s)
  local a, b, c = string.find(s, "^(%a%w*)")
  if a then
    return token(c, "id"), tail(s, b + 1)
  else
    return token(head(s), "other"), tail(s, 2)
  end
end
```

Fonte: autor.

senta um exemplo da execução do analisador léxico da Figura 3.19 (função `take`) com o texto da Figura 3.14 (representado pela variável `s`), de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Observe que a sequência de *tokens* obtida é a mesma retornada pelo motor de eventos da Figura 3.17, como esperado. □

A categorização de uma sequência de símbolos de um texto em uma sequência de *tokens* permite que cada classe gramatical seja tratada adequadamente pelo expansor em uma fase seguinte. O processamento de tal sequência resultante é discutido em detalhes na Subseção 3.4.2.

3.4.2 PROCESSAMENTO DE *TOKENS*

Um *token t* obtido na fase de análise léxica (Subseção 3.4.1) permite que o expansor verifique qual é o tratamento adequado para tal elemento, de acordo com as

Figura 3.20: Execução do analisador léxico da Figura 3.19 com o texto da Figura 3.14, de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Espaços em branco (exceto os que foram utilizados como separadores de *tokens* na impressão) estão deliberadamente marcados como visíveis.

Código-fonte Lua:

```
local s, a = "I want a gift for season."

while #s ~= 0 do
  a, s = take(s)
  io.write("(" .. a.value .. ", " .. a.class .. ") ")
end
```

Resultado da execução:

```
$ lua ex1.lua

(I, id) (␣, other) (want, id) (␣, other) (a, id) (␣, other)
   (gift, id) (␣, other) (for, id) (␣, other) (season, id)
   (., other)
```

Fonte: autor.

etapas descritas no fluxograma da Figura 3.15. Neste caso, o processamento contempla duas ações semânticas a_1 e a_2, apresentadas na Tabela 3.5, potencialmente aplicáveis ao *token* corrente t.

Tabela 3.5: Ações semânticas disponibilizadas no processamento de *tokens* do expansor de macros da Figura 3.13.

Ação	Significado
a_1	Expande macro e concatena resultado na cadeia r
a_2	Concatena símbolo na cadeia r

Fonte: autor.

Conforme ilustra a Tabela 3.5, associada ao fluxograma da Figura 3.15, a ação semântica a_1 expande uma instância de macro e concatena o resultado da expansão em uma cadeia de símbolos r já existente; tal ação é executada quando o *token* corrente t é classificado como c_1 e seu valor está presente em uma tabela contendo as definições de macros. Analogamente, a ação semântica a_2 concatena o valor de t em uma cadeia r já existente; tal ação é executada quando t é classificado como c_2, sem verificações adicionais, ou c_1, com a condição de estar ausente da tabela de macros.

Sejam $\cdot$ o operador de concatenação, *TM* a tabela de macros (representada como uma relação binária), r uma cadeia de símbolos, y um termo em sua forma normal, e t o *token* corrente, com $class(t)$ e $value(t)$ representando sua classe gramatical e valor associado, respectivamente. A função de reescrita de termos μ e as ações semânticas a_1 e a_2 são definidas a seguir, nas Equações 3.3, 3.4 e 3.5.

$$\mu(x) = y \mid (value(x), \alpha) \in TM \wedge value(x) \rightarrow_{TM}^{*} y \tag{3.3}$$

$$a_1 \equiv r \cdot \mu(t) \tag{3.4}$$

$$a_2 \equiv r \cdot value(t) \tag{3.5}$$

Adicionalmente, é conveniente introduzir duas propriedades, p_1 e p_2, para auxiliar o expansor na escolha da ação semântica adequada, dado o *token* corrente t em análise. Tais propriedades são definidas a seguir, nas Equações 3.6 e 3.7.

$$p_1(x, y) \equiv class(x) = y \tag{3.6}$$

$$p_2(x) \equiv (value(x), \alpha) \in TM \tag{3.7}$$

De acordo com as Equações 3.6 e 3.7, a propriedade p_1 verifica se a classe gramatical do *token* x corresponde a y, enquanto a propriedade p_2 verifica se o valor associado a x está presente na tabela contendo as definições de macros. Seja c_3 uma classe gramatical adicional que denota uma instância de macro. A função χ (nível 3 da Figura 3.13), definida na Equação 3.8, verifica se um *token* previamente categorizado como c_1 está elegível para reclassificação (*tokens* categorizados como c_2 permanecem com sua classe gramatical inalterada).

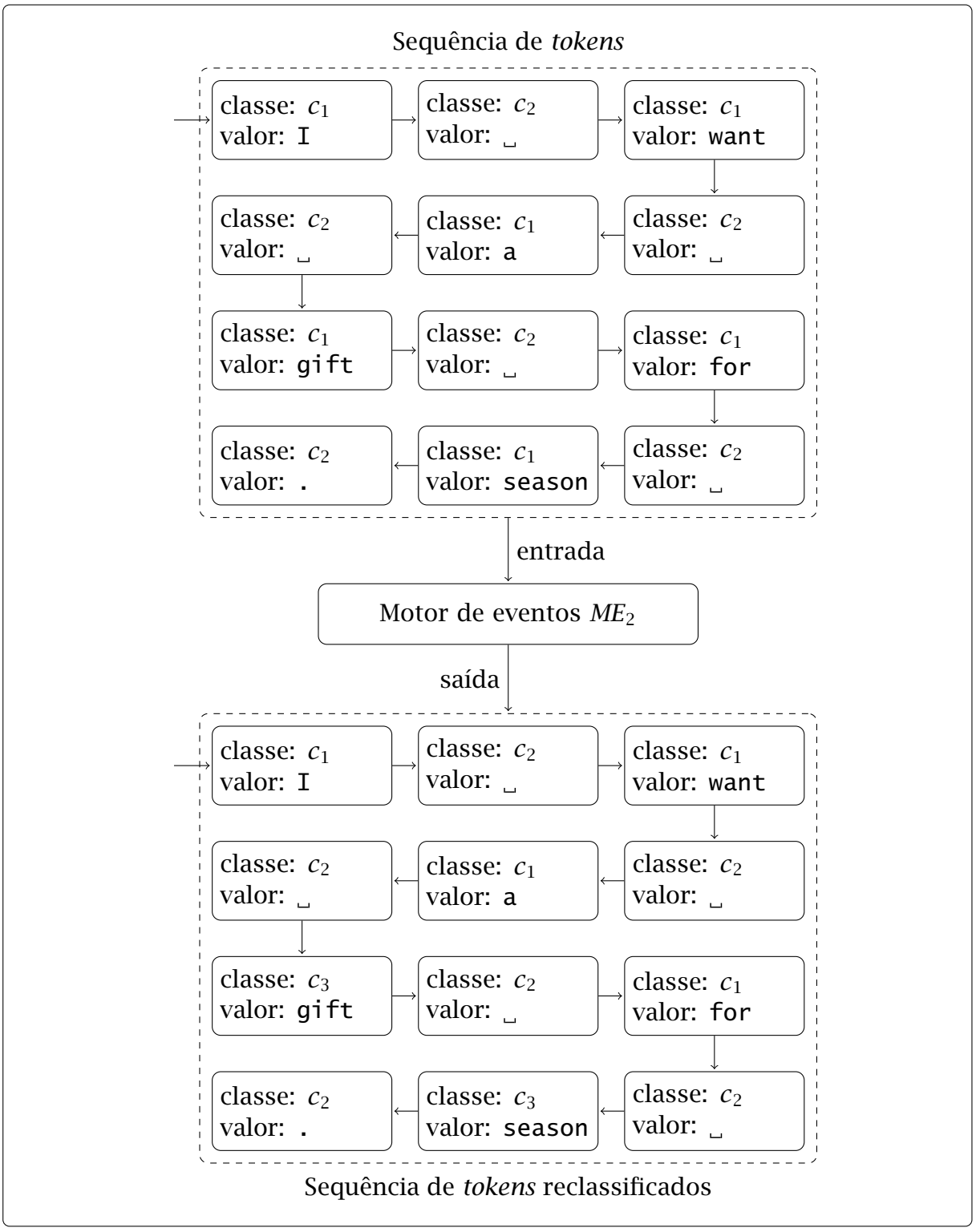
$$\chi(\sigma) = \begin{cases} c_1 & \text{se } p_1(\sigma, c_1) \wedge \neg p_2(\sigma) \\ c_2 & \text{se } \neg p_1(\sigma, c_1) \\ c_3 & \text{se } p_1(\sigma, c_1) \wedge p_2(\sigma) \end{cases} \tag{3.8}$$

Conforme ilustra a Equação 3.8, *tokens* classificados como c_3 representam instâncias de macros e, portanto, devem ser expandidos através da ação semântica a_1 (Equação 3.4). A Figura 3.21 apresenta um motor de eventos ME_2 como modelo de implementação da função de reclassificação χ (nível 3 da Figura 3.13) aplicada em uma sequência de *tokens* obtidos na fase de análise léxica.

Dada uma sequência de *tokens* devidamente reclassificados através da função de reclassificação χ (Equação 3.8), a Figura 3.22 apresenta um autômato finito determinístico M_2 que descreve o processamento de *tokens* proposto (nível 4 da Figura 3.13).

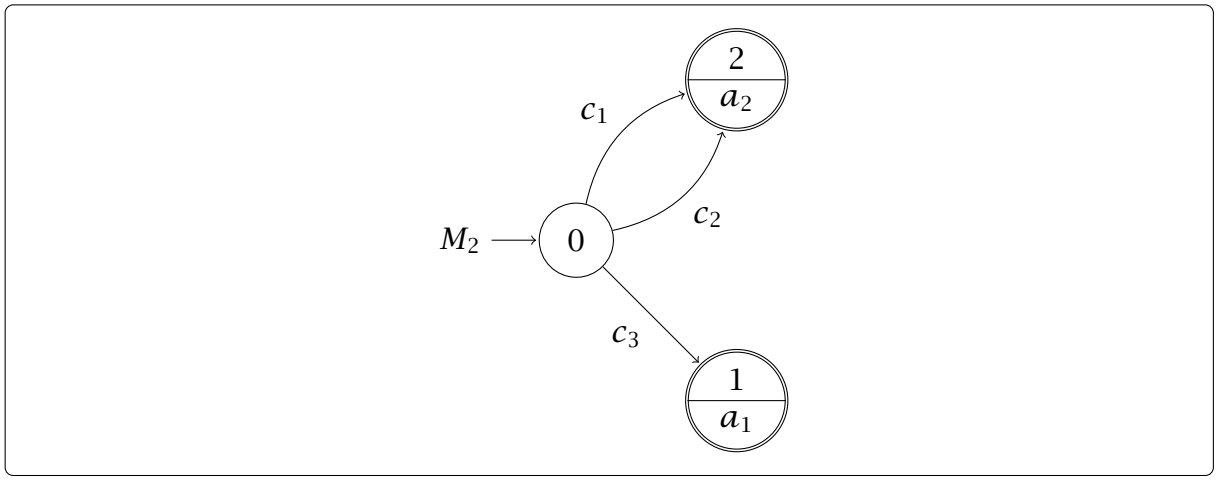
Sejam $w \in \Sigma_2^*$ uma cadeia, tal que $w \in L(M_2)$, e $A\colon Q_2 \mapsto \{a_1, a_2\}$ uma função que mapeia estados de M_2 em ações semânticas a_1 e a_2. A ação semântica a_1 será disparada se, e somente se, o autômato finito M_2, durante o reconhecimento de w, referenciar 1 como estado corrente após o consumo de um símbolo σ_i, $a_1 \iff$

Figura 3.21: Motor de eventos ME_2 como modelo de implementação da função de reclassificação χ (Equação 3.8, nível 3 da Figura 3.13).



Fonte: autor.

Figura 3.22: Autômato finito determinístico M_2 que descreve o processamento de *tokens* (nível 4) para o expansor de macros da Figura 3.13.



Fonte: autor.

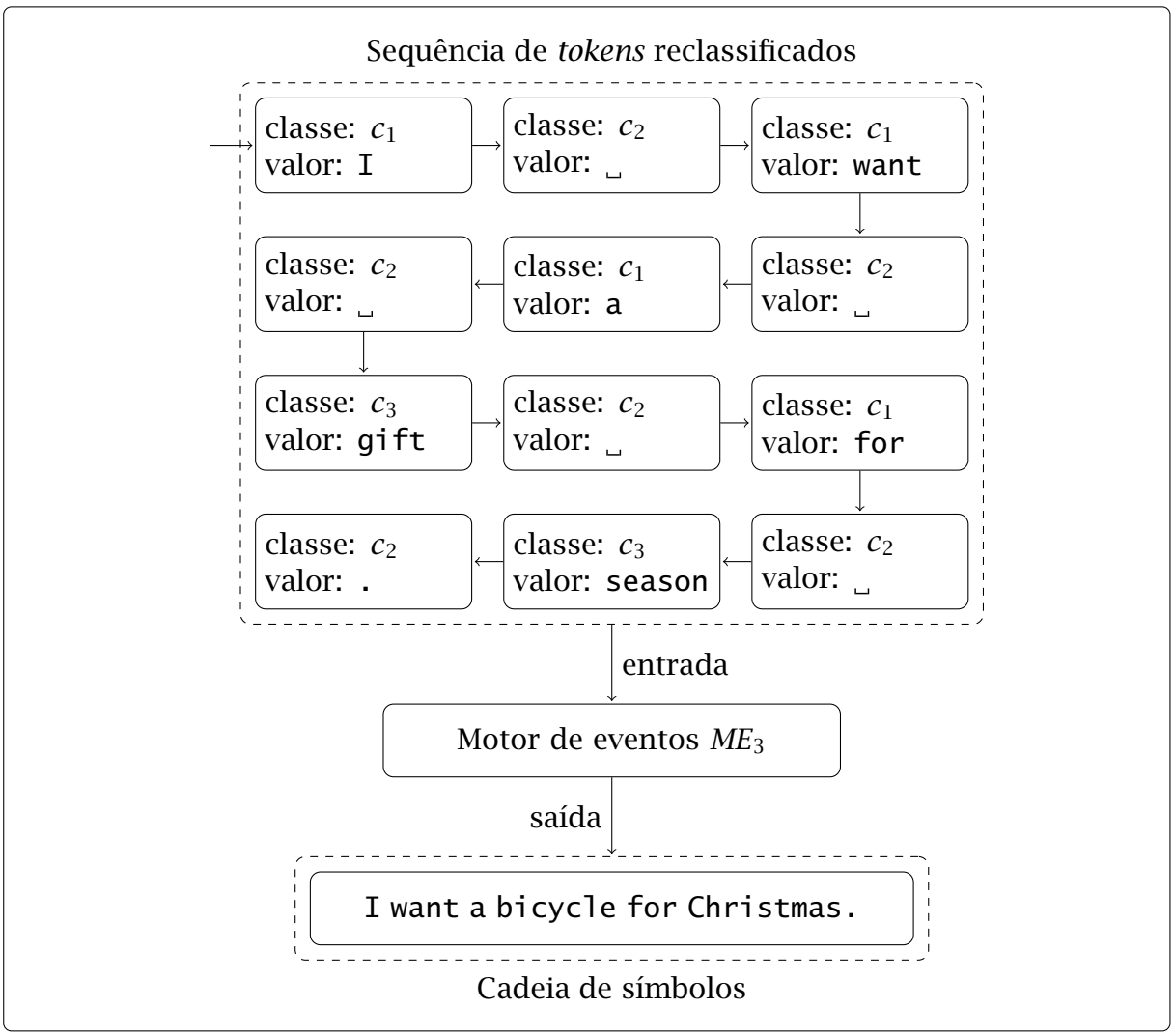
$(0, \sigma_i) \vdash (1, \epsilon)$. Analogamente, a ação semântica a_2 será disparada se, e somente se, durante o reconhecimento de w, o autômato finito M_2 referenciar 2 como estado corrente após o consumo de um símbolo σ_i, $a_2 \iff (0, \sigma_i) \vdash (2, \epsilon)$. A função A, neste caso, é definida por extensão como $A = \{(1, a_1), (2, a_2)\}$.

A função de reescrita de termos μ (Equação 3.3) aplica passos de redução sucessivos até que o termo resultante não possa ser reescrito (forma normal), utilizando a estratégia de expansão de múltiplos passos apresentada na Seção 3.2.

A Figura 3.23 apresenta um motor de eventos ME_3 como modelo de implementação do processamento de *tokens* (nível 4 da Figura 3.13), no qual a sequência de *tokens*, devidamente reclassificada através da aplicação da função χ (Equação 3.8), é reduzida a uma cadeia de símbolos de saída. No exemplo, as ocorrências das instâncias de macros `gift` e `season` no texto da Figura 3.14 foram expandidas para suas formas normais `bicycle` e `Christmas`, respectivamente.

A composição hierárquica dos três motores de eventos ME_1, ME_2 e ME_3 (Figuras 3.17, 3.21 e 3.23, respectivamente) resulta no projeto completo do expansor de macros da Figura 3.13. É importante observar que que o tratamento de eventos ocorre de modo contínuo e sob demanda: quando o motor ME_1 (primeiro nível) dispõe de símbolos suficientes para compor um *token* (incluindo sua classe gramatical), ME_1 o envia como evento de entrada para o motor ME_2. No segundo nível, ME_2 potencialmente reclassifica o *token* obtido e o envia como evento de entrada para o motor ME_3 (terceiro nível). O motor ME_3 executa então a ação semântica de acordo com a classe gramatical do *token* obtido e aguarda novos eventos. Ao término da sequência de símbolos submetida ao primeiro nível, o motor ME_3 emite a cadeia de símbolos resultante. A Figura 3.24 ilustra a composição hierárquica dos três motores de eventos.

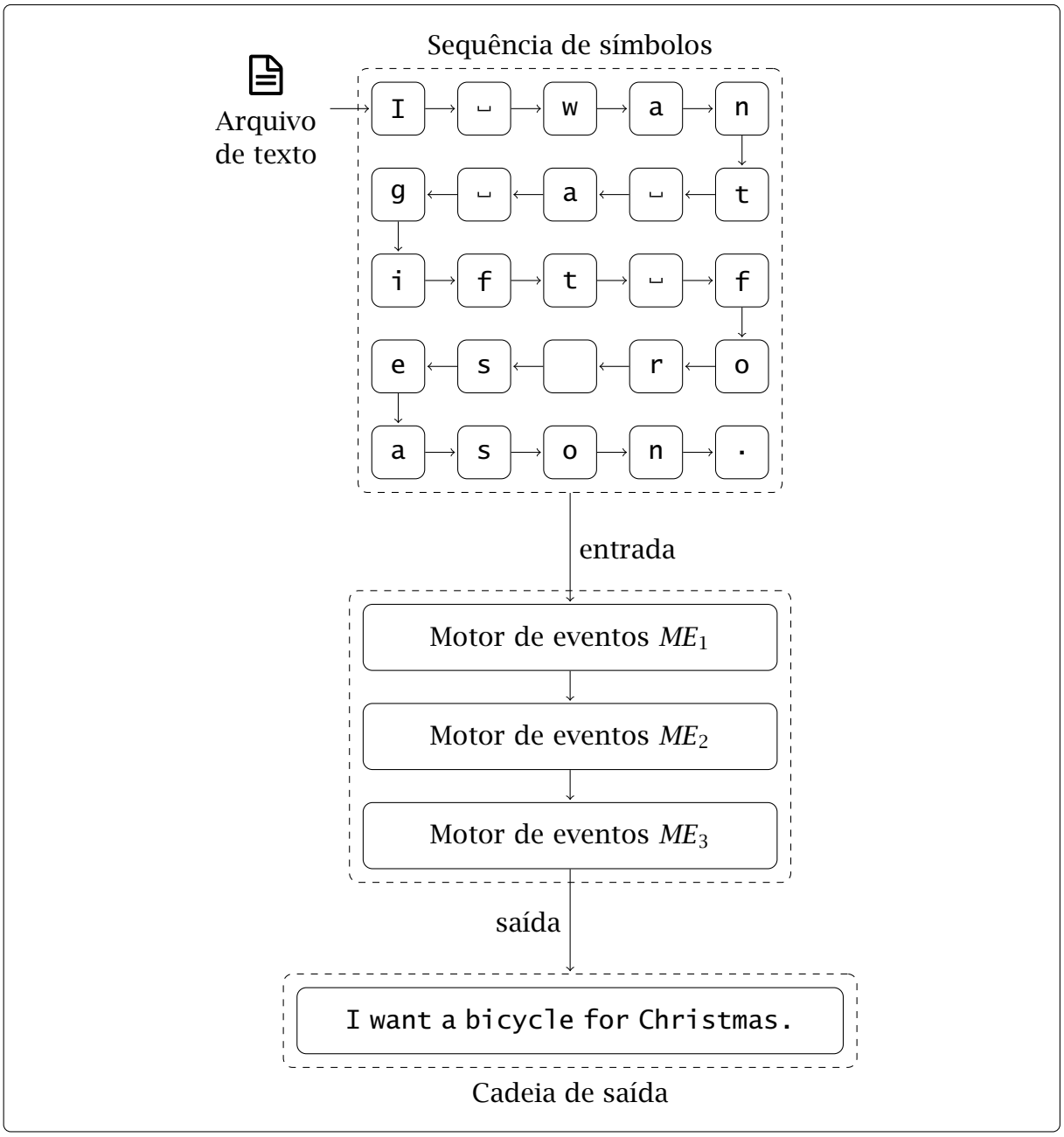
Figura 3.23: Motor de eventos ME_3 como modelo de implementação do processamento de *tokens* (nível 4) para o expansor de macros da Figura 3.13.



Fonte: autor.

Considere uma implementação do expansor de macros da Figura 3.13, apresentada na Figura 3.25 e detalhada a seguir. A função `process` recebe uma cadeia de símbolos ASCII e uma tabela contendo as definições de macros como parâmetros (`s` e `macros`, a saber), sintetizando as fases de análise léxica (nível 2) e processamento de *tokens* (níveis 3 e 4), e retorna uma cadeia de símbolos sem ocorrências de instâncias de macros. A função utiliza o analisador léxico da Figura 3.19 sob demanda, tal que um *token* seja analisado por vez. Caso o *token* corrente (representado pela variável local `a`) seja um identificador (classe gramatical c_1 com rótulo *id*) presente na tabela de macros (representada pelo parâmetro `macros`), a função concatena a cadeia de símbolos de saída (representada pela variável `result`) com o resultado de uma chamada recursiva de `process` com a sequência de símbolos de substituição correspondente e a tabela de macros fornecidas como parâmetros (`macros[word]` e `macros`, respectivamente). Tal estratégia de implementação garante que, no retorno

Figura 3.24: Composição hierárquica dos três motores de eventos ME_1, ME_2 e ME_3 (Figuras 3.17, 3.21 e 3.23, respectivamente), resultando no projeto completo do expansor de macros da Figura 3.13.



Fonte: autor.

das chamadas recursivas da função `process`, a cadeia resultante `result` não conterá instâncias de macros remanescentes. Caso o *token* corrente seja um símbolo qualquer (classe gramatical c_2 com rótulo *other*) ou um identificador ausente na tabela de macros, a função concatena a cadeia de símbolos de saída com o valor do *token* (representado pelo índice nominal `a.value`). O processo é então repetido até que o analisador léxico (representado pela função `take`) não possua mais *tokens* a extrair (isto é, a cadeia de símbolos de entrada `s` está vazia). A cadeia de símbolos resultante da expansão é então retornada.

Figura 3.25: Implementação do expansor de macros simples, sintetizando as fases de análise léxica (nível 2 da Figura 3.13) e processamento de *tokens* (níveis 3 e 4 da Figura 3.13).

```
local function process(s, macros)
  local result = ""
  while #s ~= 0 do
    local a, b = take(s)
    s = b
    local word = a.value
    if a.class == "id" and macros[word] then
      result = result .. process(macros[word], macros)
    else
      result = result .. word
    end
  end
  return result
end
```

Fonte: autor.

Exemplo 13 (execução do expansor de macros simples da Figura 3.25)**.** A Figura 3.26 apresenta um exemplo da execução do expansor de macros simples da Figura 3.25 (função `process`) com o texto e tabela de macros da Figura 3.14 (representados pelas variáveis `s` e `macros`, respectivamente), imprimindo a cadeia de símbolos resultante no terminal de comando. □

Conforme ilustra a Figura 3.26, as ocorrências de `gift` e `season` foram substituídas por `bicycle` and `Christmas`, respectivamente, na cadeia de símbolos de saída. Observe que a tabela de macros (representada pela variável `macros`) contém os padrões sintáticos (lado esquerdo) e as regras de substituição correspondentes (lado direito). A estratégia de expansão utilizada na função `process` permite que macros internas sejam expandidas em sequências de substuição de macros mais externas (expansão de múltiplos passos); entretanto, não há garantias quanto à terminação da reescrita de termos.

Figura 3.26: Execução do expansor de macros simples da Figura 3.25 com o texto e tabela de macros da Figura 3.14, imprimindo a cadeia de símbolos resultante no terminal de comando.

Código-fonte Lua:

```
local macros = {
  gift = "bicycle",
  season = "Christmas"
}

local s = "I want a gift for
   season."
print(process(s, macros))
```

Resultado da execução:

```
$ lua ex2.lua

I want a bicycle for
   Christmas.
```

Fonte: autor.

O Apêndice A apresenta exemplos de expansores de macros com padrões sintáticos livres e dependentes de contextos, com suporte à captura de estruturas sintáticas e transformações algorítmicas, e suas implementações correspondentes.

CAPÍTULO 4

OPERAÇÕES DO EXPANSOR DE MACROS

> To me, programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge.
>
> GRACE HOPPER

O projeto de um expansor de macros discutido no Capítulo 3 viabiliza um arquétipo para transformações simbólicas e algorítmicas (BRABRAND; SCHWARTZBACH, 2002; BRABRAND; MØLLER; SCHWARTZBACH, 2002). Este capítulo propõe um conjunto de primitivas (construtos básicos) para definição de macros, controle de fluxo de expansão e operações básicas sobre alguns tipos de dados, incluindo exemplos de uso. O padrão sintático das primitivas segue a estrutura proposta no expansor de macros da Seção A.1, na página 125 (tipo 2 na hierarquia de Chomsky).

4.1 CONJUNTO DE PRIMITIVAS

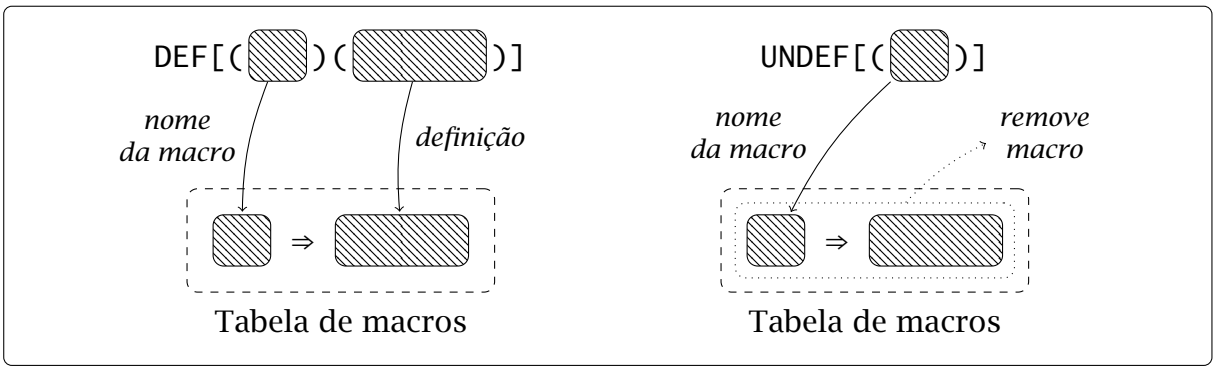
Inspirada nos conjuntos de construtos básicos disponibilizados nas linguagens TEX, Dylan e Rust, esta seção apresenta uma proposta de um conjunto de primitivas, permitindo que usuário possa representar transformações sobre textos de forma conveniente e expressiva. Tais construtos básicos do expansor de macros oferecem subsídios para gerenciamento de macros e controle de fluxo em tempo de expansão, além de operações básicas sobre alguns tipos de dados. É importante destacar que novas macros podem combinar primitivas e construtos derivados definidos previamente, de modo *hierárquico* (SCHUMAN; JORRAND, 1970). Observe ainda que a maioria das primitivas realiza algum tipo de interpretação sobre o texto sendo expandido. Por exemplo, o símbolo `2` (posição 50 na tabela ASCII) pode ser interpretado como o número 2 em uma operação de soma; entretanto, o resultado sempre assume um formato textual. Os blocos hachurados representam capturas e podem conter outras instâncias de macros.

Definição 62 (primitivas de gerenciamento da tabela de macros)**.** As primitivas `DEF` e `UNDEF` gerenciam a tabela de macros, inserindo e removendo definições, respectivamente, inspiradas nos construtos `macro_rules!`, `define` e `\let\foo\relax` das linguagens Rust, Dylan e TEX. Capturas são representadas no corpo da definição de

uma macro por *placeholders* no formato # seguido de um número natural associado. Tal padrão sintático para representação de capturas é fortemente inspirado na linguagem TEX (KNUTH, 1986). □

A Figura 4.1 apresenta a sintaxe e operação das primitivas DEF e UNDEF para gerenciamento da tabela de macros. Observe que tais primitivas inserem novas definições e removem definições existentes na tabela de macros de acordo com o nome (padrão sintático regular, tipo 3 na hierarquia de Chomsky) associado. No caso de inserção, o bloco de definição correspondente é inserido literalmente, sem expansões. Caso a macro sendo definida já exista na tabela de macros, sua entrada correspondente será atualizada para o novo valor.

Figura 4.1: Sintaxe e operação das primitivas de gerenciamento da tabela de macros, de acordo com a Definição 62.



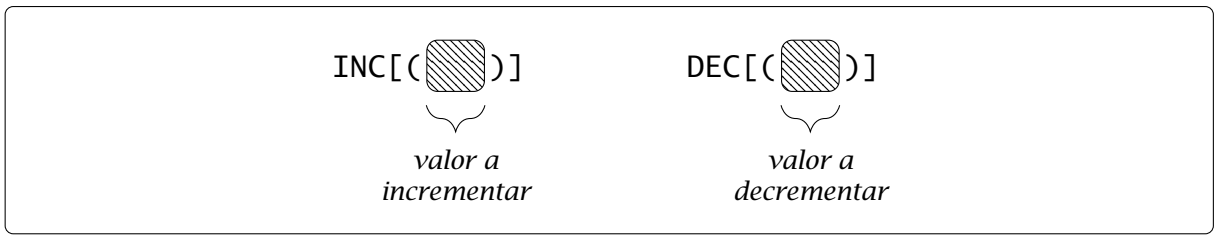
Fonte: autor.

Definição 63 (primitivas de incremento e decremento de valores inteiros)**.** As primitivas INC e DEC implementam as operações de incremento e decremento de valores inteiros, respectivamente. Tais primitivas expandem suas capturas (utilizando a estratégia de pré-varredura de captura, de acordo com a Definição 57) e realizam interpretações sobre os textos resultantes. Neste caso, os textos são convertidos para representações inteiras e adicionados (ou subtraídos) de uma unidade. As primitivas de incremento e decremento são fortemente inspiradas no construto `\advance\cnt by 1` da linguagem TEX. □

A Figura 4.2 apresenta a sintaxe e operação das primitivas INC e DEC para incremento e decremento de valores inteiros. Após a operação aritmética sobre o valor inteiro, o resultado é convertido para uma representação textual correspondente.

Definição 64 (primitivas de controle de fluxo)**.** As primitivas IF e UNLESS avaliam uma condição lógica representada pelo primeiro bloco de captura e retornam o bloco correspondente ao resultado da avaliação, devidamente expandido. Tais primitivas admitem os valores T e F como verdadeiro e falso, respectivamente. As primitivas são inspiradas no construto `if` da linguagem Dylan. □

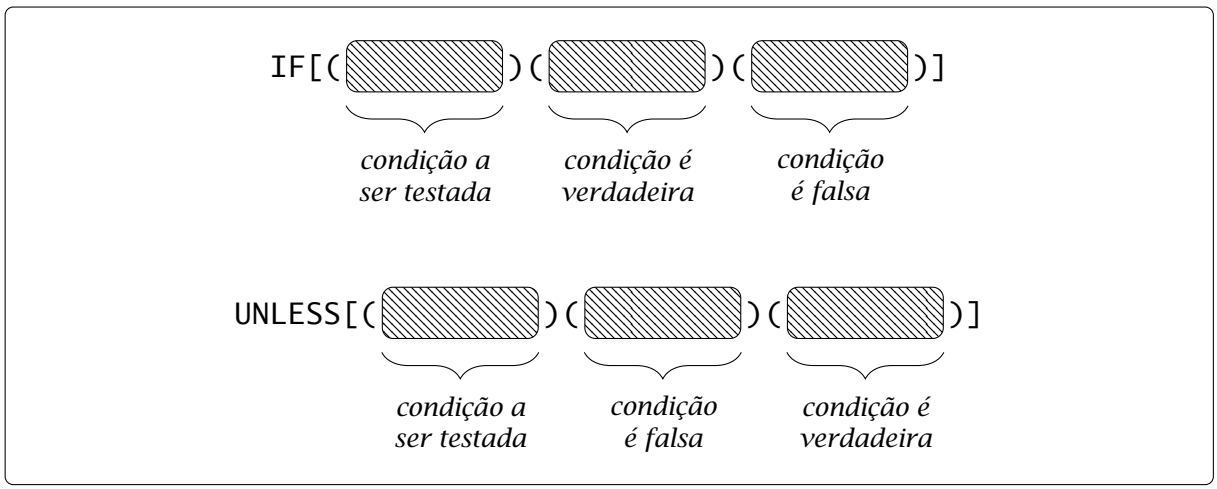
Figura 4.2: Sintaxe e operação das primitivas de incremento e decremento de valores inteiros, de acordo com a Definição 63.



Fonte: autor.

A Figura 4.3 apresenta a sintaxe e operação das primitivas `IF` e `UNLESS` para controle de fluxo. Observe que a primitiva `UNLESS` implementa a primitiva `IF` com os blocos condicionais trocados de posição. É importante destacar que somente o bloco correspondente ao resultado da avaliação da condição lógica é expandido. Internamente, o texto expandido do primeiro bloco é comparado com o símbolo `T`.

Figura 4.3: Sintaxe e operação das primitivas de controle de fluxo, de acordo com a Definição 64.



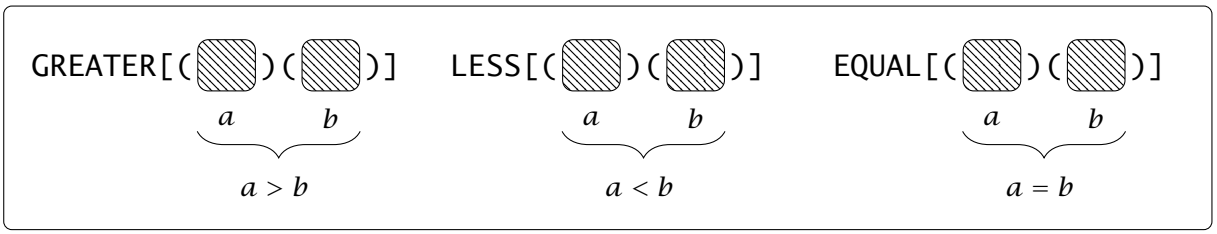
Fonte: autor.

Definição 65 (primitivas de comparação entre valores numéricos)**.** As primitivas `LESS`, `GREATER` e `EQUAL` implementam operações de comparação entre valores numéricos, retornando `T` ou `F` (verdadeiro ou falso, respectivamente), de acordo com o resultado da comparação. As duas capturas são convertidas internamente em representações inteiras. As primitivas são inspiradas nas operações de comparação da linguagem TEX. □

A Figura 4.4 apresenta a sintaxe e a operação das primitivas `GREATER`, `LESS` e `EQUAL` de comparação entre valores numéricos, com a e b denotando as duas capturas devidamente convertidas em representações inteiras. As chaves denotam as operações correspondentes. É importante destacar que, no caso da aplicação de tais

primitivas com valores textuais, optou-se por utilizar seus códigos de ordenação lexicográfica como critério de decisão.

Figura 4.4: Sintaxe e operação das primitivas de comparação entre valores numéricos, de acordo com a Definição 65.

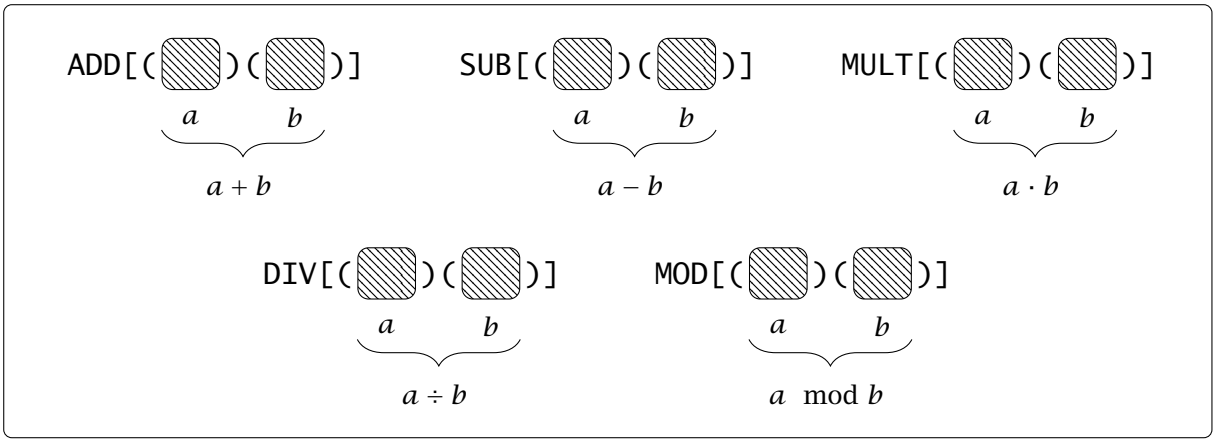


Fonte: autor.

Definição 66 (primitivas de operações aritméticas)**.** As primitivas `ADD`, `SUB`, `MULT`, `DIV` e `MOD` implementam as operações aritméticas de soma, subtração, multiplicação, divisão e módulo, respectivamente, retornando o resultado correspondente. As duas capturas são convertidas internamente em representações inteiras. As primitivas são inspiradas nas operações aritméticas da linguagem Dylan. □

A Figura 4.5 apresenta a sintaxe e operação das primitivas `ADD`, `SUB`, `MULT`, `DIV` e `MOD` de cálculos aritméticos de soma, subtração, multiplicação, divisão e módulo, com a e b denotando as duas capturas devidamente convertidas em representações inteiras. As chaves denotam as operações correspondentes. É importante destacar que, no caso de uma expressão de tipo misto, optou-se pelo tipo do operando mais complexo, com exceção da operação de divisão, cujo resultado será sempre real.

Figura 4.5: Sintaxe e operação das primitivas de cálculos aritméticos, de acordo com a Definição 66.



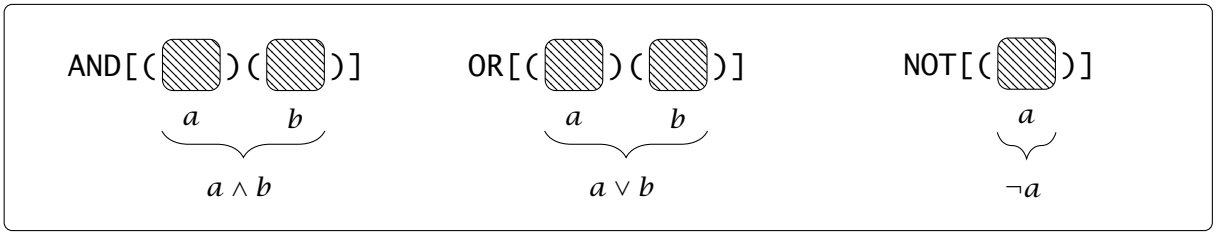
Fonte: autor.

Definição 67 (primitivas de operações lógicas)**.** As primitivas `AND`, `OR` e `NOT` implementam as operações lógicas de conjunção, disjunção e negação, respectivamente,

retornando `T` ou `F` (verdadeiro ou falso, respectivamente), de acordo com o resultado da operação. Internamente, os valores expandidos dos dois blocos são convertidos em representações lógicas. □

A Figura 4.6 apresenta a sintaxe e a operação das primitivas `AND`, `OR` e `NOT` de operações lógicas de conjunção, disjunção e negação, com a e b denotando as duas capturas devidamente convertidas em representações lógicas. As chaves denotam as operações correspondentes. No caso da aplicação de tais primitivas com valores diferentes de `T` ou `F` (verdadeiro ou falso, respectivamente), estas retornarão `F` como resultado.

Figura 4.6: Sintaxe e operação das primitivas de operações lógicas, de acordo com a Definição 67.



Fonte: autor.

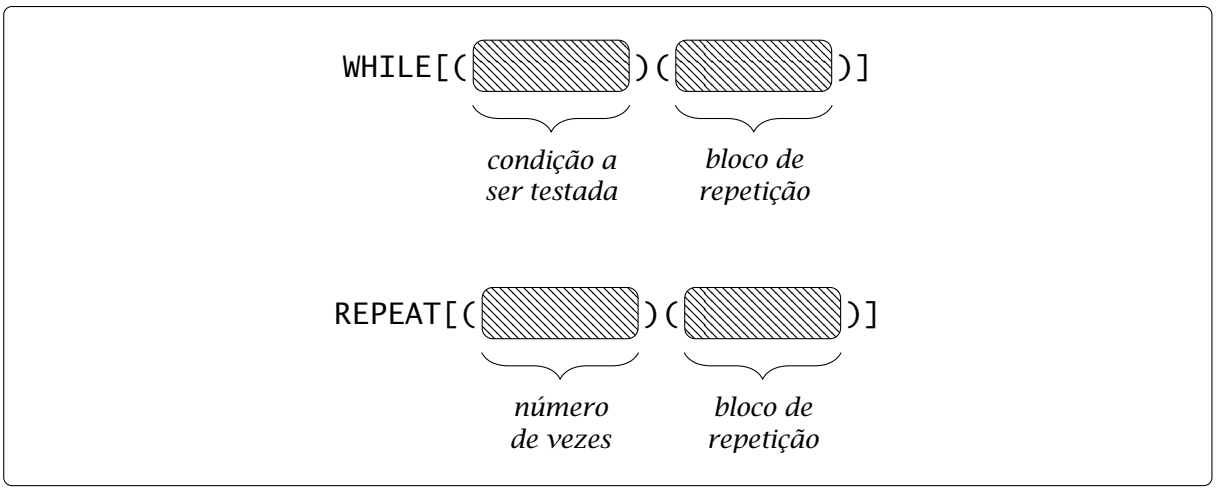
Definição 68 (primitivas de repetição)**.** As primitivas `REPEAT` e `WHILE` implementam estruturas de repetição controlada e com teste de condição *a priori*, respectivamente. O bloco a ser repetido é devidamente expandido a cada iteração utilizando a estratégia de pré-varredura de captura. Na primitiva `REPEAT`, o primeiro bloco é convertido para uma representação inteira, denotando o número de repetições do segundo bloco. No caso da primitiva `WHILE`, o primeiro bloco é convertido para uma representação lógica antes de cada potencial iteração do segundo bloco. □

A Figura 4.7 apresenta a sintaxe e a operação das primitivas `REPEAT` e `WHILE` de repetição. As chaves denotam a condição a ser testada (para `WHILE`) ou o número de vezes (para `REPEAT`), e o bloco de repetição propriamente dito. Ao contrário da repetição controlada (na qual a quantidade de iterações é definida *a priori*), não há garantias de término para a correspondente condicional.

Definição 69 (primitivas de gerenciamento de símbolos)**.** As primitivas `SET` e `GET` gerenciam uma tabela auxiliar de símbolos (definição e obtenção, respectivamente), permitindo que dados sejam armazenados em memória para uso posterior. Tais primitivas são inspiradas nos contadores em TEX. □

A Figura 4.8 apresenta a sintaxe e operação das primitivas `SET` e `GET` para gerenciamento de uma tabela de símbolos. Observe que tais primitivas armazenam e

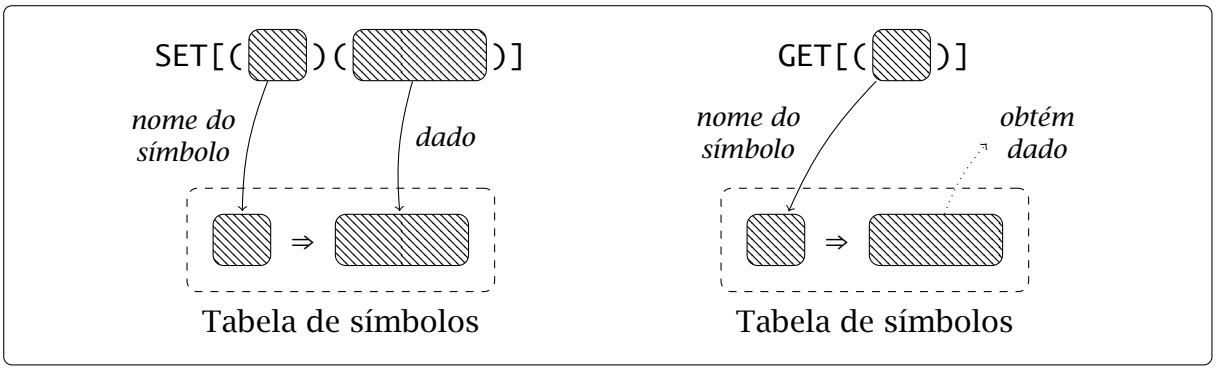
Figura 4.7: Sintaxe e operação das primtivas de repetição, de acordo com a Definição 68.



Fonte: autor.

recuperam representações textuais dos dados em memória para uso durante a expansão. No caso da recuperação de um nome de símbolo inexistente na tabela, a primitiva `GET` retorna ϵ (denotando uma cadeia vazia). O nome do símbolo segue um padrão sintático regular (tipo 3 na hieraquia de Chomsky).

Figura 4.8: Sintaxe e operação das primitivas de gerenciamento de símbolos, conforme a Definição 69.

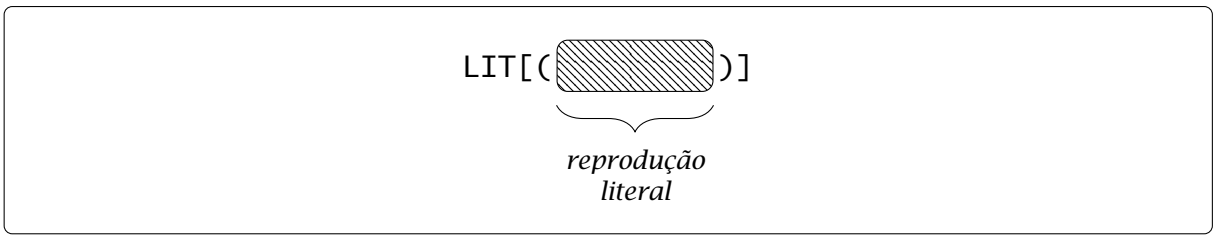


Fonte: autor.

Definição 70 (primitiva de reprodução literal)**.** A primitiva `LIT` permite que sua captura seja reproduzida literalmente na cadeia de saída, sem expansões. Tal primitiva pode ser utilizada para evitar expansões de macros no texto, conforme a conveniência. A inspiração para esta primitiva advém do construto `\noexpand` da linguagem TEX. □

A Figura 4.9 apresenta a sintaxe e a operação da primitiva `LIT` de reprodução literal. A chave denota tal bloco de reprodução. Observe que eventuais macros presentes na captura serão tratadas como símbolos convencionais e reproduzidas *ipsis litteris* na cadeia de saída.

Figura 4.9: Sintaxe e operação da primitiva de reprodução literal, de acordo com a Definição 70.

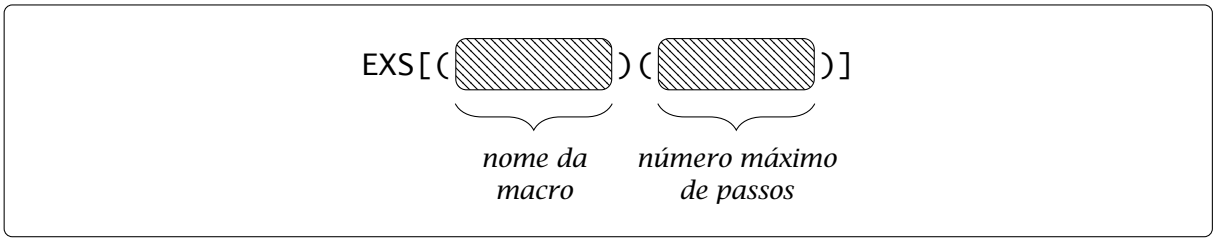


Fonte: autor.

Definição 71 (primitiva de estratégia de expansão)**.** A primitiva EXS permite alterar, em tempo de execução do expansor, a estratégia de expansão para uma macro qualquer, condicionando-a a um número máximo de passos (Definição 60). Tal primitiva atua tão somente no metanível do expansor e não produz transformações na cadeia de saída. □

A Figura 4.10 apresenta a sintaxe e a operação da primitiva EXS de estratégia de expansão. As chaves denotam os blocos contendo nome da macro e o número máximo de passos associado. Internamente, o segundo bloco é convertido para uma representação inteira. Observe que a alteração da estratégia de expansão em tempo de execução do expansor pode resultar na reprodução literal das macros explicitamente anotadas na cadeia de saída. Neste caso específico, o resultado da expansão pode assemelhar-se ao da primitiva LIT, mas tratam-se de operações diferentes.

Figura 4.10: Sintaxe e operação da primitiva de estratégia de expansão, de acordo com a Definição 71.



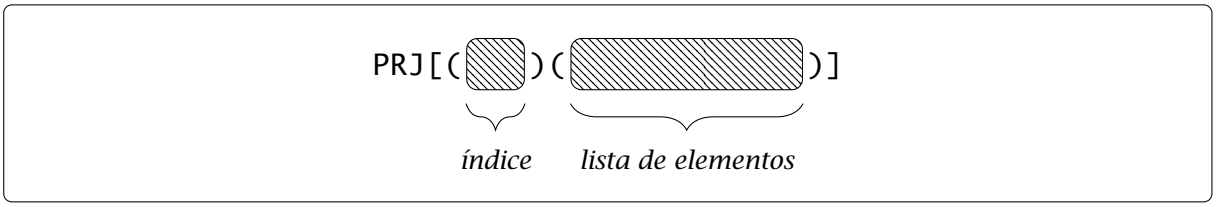
Fonte: autor.

Definição 72 (primitiva de projeção)**.** A primitiva PRJ implementa a operação de projeção em uma lista, retornando o i-ésimo elemento desta. Formalmente, uma *projeção* é definida como $P_k^i(x_1, \ldots, x_k) = x_i$ para $1 \leq i \leq k$ (EPSTEIN; CARNIELLI, 1989; ROBIČ, 2015; ENDERTON, 2010). □

A Figura 72 apresenta a sintaxe e a operação da primitiva PRJ de projeção. A lista de elementos, devidamente expandida, é especificada através do formato serial CSV (do inglês *comma-separated values*, ou [denotação de] *valores separados por vírgula* no vernáculo) para armazenamento de dados tabulados (SHAFRANOVICH, 2005). As

chaves denotam os blocos correspondentes ao índice da lista (internamente convertido para uma representação inteira) e a lista de elementos propriamente dita. No caso de um índice fora do intervalo (incluindo a ocorrência de uma lista vazia), a primitiva PRJ retorna ϵ (denotando uma cadeia vazia).

Figura 4.11: Sintaxe e operação da primitiva de projeção, de acordo com a Definição 72.



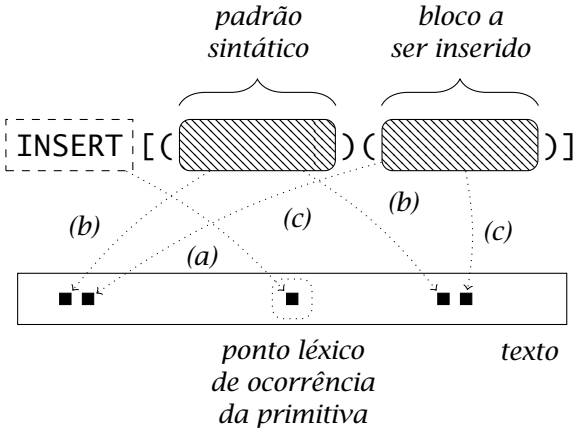
Fonte: autor.

Definição 73 (primitivas de adaptatividade)**.** As primitivas INSERT, REMOVE e QUERY implementam as ações adaptativas elementares de inserção, remoção e consulta, respectivamente (Definição 39). Tais primitivas identificam padrões sintáticos e realizam potenciais modificações de escopo global, extrapolando seus pontos léxicos de ocorrência (isto é, suas posições ao longo do texto). □

As Figuras 4.12, 4.13 e 4.14 apresentam a sintaxe e a operação das primitivas INSERT, REMOVE e QUERY que implementam as ações adaptativas elementares de inserção, remoção e consulta. Os símbolos ■ denotam posições ao longo do texto (pontos léxicos). Estas posições podem indicar padrões sintáticos, ocorrência da própria macro sendo expandida e eventuais inserções, conforme ilustram os exemplos correspondentes. Tais primitivas identificam padrões sintáticos no texto e efetuam as operações correspondentes. As capturas são devidamente expandidas utilizando a estratégia de pré-varredura de captura. A primitiva INSERT insere o bloco capturado no ponto léxico seguinte ao término das ocorrências de tal padrão (Figura 4.12). Complementarmente, a primitiva REMOVE remove do texto todas as ocorrências encontradas do padrão sintático (Figura 4.13). A primitiva QUERY identifica as ocorrências do padrão sintático e as armazena em memória, como ponteiros, na tabela de símbolos, para uso posterior (Figura 4.14). Caso seja necessário evitar a expansão de eventuais macros no padrão sintático capturado (e, eventualmente, no bloco a ser inserido), a primitiva LIT deve ser utilizada para garantir a reprodução literal. Na ausência do padrão sintático no texto, as primitivas de adaptatividade retornam ϵ (denotando a cadeia vazia).

O conjunto de primitivas proposto oferece subsídios para implementação de algoritmos de propósito geral durante a expansão de macros. Em particular, as primitivas INC, DEC e PRJ permitem a representação de funções primitivas recursivas, uma subclasse das funções Turing-computáveis (a primitiva IF pode ser utilizada

Figura 4.12: Sintaxe e operação da primitiva que implementa a ação adaptativa de inserção (Definição 73).



Legenda: *(a)* ponto léxico de ocorrência da primitiva `INSERT` no texto, *(b)* ocorrências do padrão sintático no texto, e *(c)* inserção do bloco nas posições seguintes às ocorrências do padrão sintático.

Exemplos:

```
INSERT[(a)(b)] ac    ⇒  abc
a INSERT[(a)(b)] ac  ⇒  ababc
ca INSERT[(a)(b)]    ⇒  cab
ac INSERT[(a)(b)] c  ⇒  abcc
```

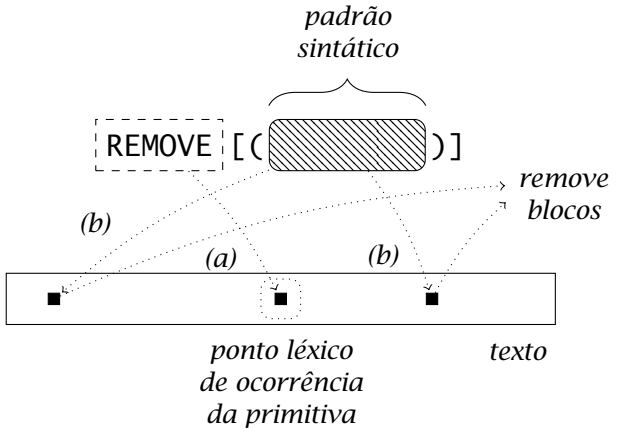
Fonte: autor.

para reproduzir os conceitos de base de indução e passo indutivo) (EPSTEIN; CARNIELLI, 1989; ENDERTON, 2010). Adicionalmente, a tese de Church possibilita considerar que tal conjunto de primitivas é suficiente para expressar qualquer algoritmo (CHURCH, 1936). Entretanto, são necessárias provas de tais propriedades. O Apêndice A, na página 125, discute os aspectos de implementação das primitivas apresentadas utilizando o expansor de macros da Seção A.1 e as transformações algorítmicas viabilizadas através da linguagem Lua, incluindo desafios de projeto que advêm dos efeitos colaterais causados no texto pelas ações adaptativas elementares de inserção e remoção.

4.2 EXEMPLOS DE USO

Esta seção apresenta exemplos de expansão de macros viabilizados através do expansor de macros da Seção A.1, estendido para oferecer suporte a transformações algorítmicas (de acordo com as discussões da Subseção A.3.3, página 163), e do conjunto de primitivas introduzido na Seção 4.1.

Figura 4.13: Sintaxe e operação da primitiva que implementa a ação adaptativa de remoção (Definição 73).



Legenda: *(a)* ponto léxico de ocorrência da primitiva `REMOVE` no texto, e *(b)* ocorrências do padrão sintático no texto.

Exemplos:

```
REMOVE[(a)] ac    ⇒  c
a REMOVE[(a)] ac  ⇒  c
ca REMOVE[(a)]    ⇒  c
c REMOVE[(a)] b   ⇒  cb
```

Fonte: autor.

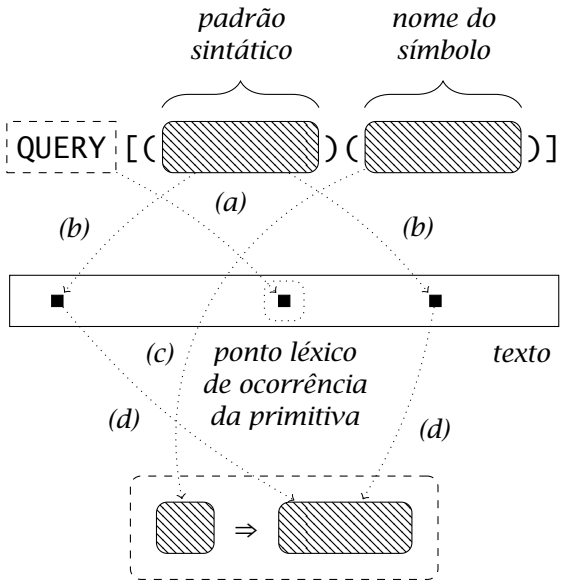
É importante destacar que os exemplos desta seção têm como objetivo exercitar o pensamento abstrato na busca por soluções computacionais a partir de construtos elementares. Como resultado, construtos derivados obtidos de tais exercícios podem ser facilmente estendidos para outros domínios.

O primeiro exemplo contemplado nesta seção (Exemplo 14) trata de repetição iterativa, manifestada em uma brincadeira de roda popular. Observe a utilização das duas primitivas de repetição (a saber, `WHILE` e `REPEAT`), viabilizando soluções equivalentes.

Exemplo 14 (repetição iterativa)**.** Considere uma brincadeira de roda comum em países de língua inglesa chamada *duck, duck, goose* (do original sueco *anka anka grå anka* e conhecida como *jogo do lenço* no Brasil). Tal brincadeira é detalhada a seguir:

1. Um número razoável de crianças senta-se no chão, formando um círculo. Estas sentam-se voltadas para o interior do círculo.

Figura 4.14: Sintaxe e operação da primitiva que implementa a ação adaptativa de consulta (Definição 73).



Legenda: *(a)* ponto léxico de ocorrência da primitive `QUERY` no texto, *(b)* ocorrências do padrão sintático no texto, *(c)* nome do símbolo na tabela, e *(d)* ponteiros para as ocorrências.

Exemplos:

`c QUERY[(a)(x1)] b`	⇒	`cb`
`ca QUERY[(a)(x2)]`	⇒	`ca`
`QUERY[(a)(x3)] ac`	⇒	`ac`
`a QUERY[(a)(x4)] ac`	⇒	`aac`

Símbolo	Ponteiros
`x1`	—
`x2`	$(1,1)$
`x3`	$(2,2)$
`x4`	$(1,1),(2,2)$

Fonte: autor.

2. Uma criança é sorteada para começar a brincadeira. Inicialmente, sua tarefa é simplesmente correr em volta do círculo até decidir qual colega iniciará sua contagem.

3. Ao escolher um colega, sentado no círculo, a criança toca sua cabeça e diz a palavra *duck*. Este é o início da brincadeira.

4. A criança, então, deverá tocar a cabeça do colega seguinte. Ela pode optar por dizer *duck* ou *goose*. Se a criança disser *duck*, repete-se o item 4 (este item).

5. Se a criança disse *goose* ao tocar a cabeça do colega, este deve levantar-se rapidamente do lugar em que está sentado no círculo e perseguir a criança que tocou em sua cabeça. A perseguição ocorre no contorno do círculo.

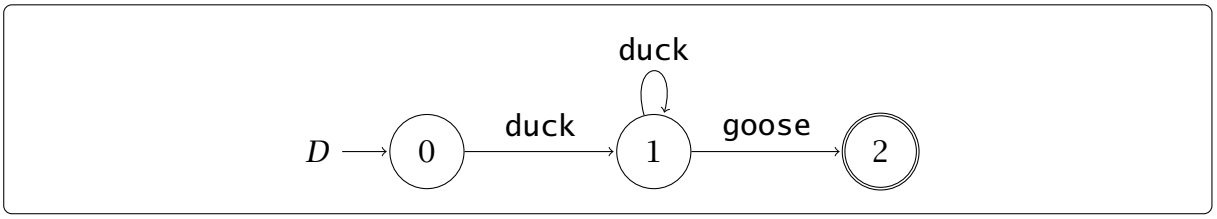
6. O colega sendo perseguido deve sentar-se no local vacante do círculo. Caso tal situação aconteça sem que ele seja apanhado, a brincadeira recomeça no item 2, com o perseguidor sendo escolhido para dar prosseguimento.

7. Caso o colega seja apanhado por seu perseguidor, este é eliminado da brincadeira e deve sentar-se no meio do círculo. O perseguidor, então, recomeça a brincadeira a partir do item 2.

8. A brincadeira encerra-se quando restam poucas crianças no círculo. A criança que conseguir eliminar mais colegas é declarada a vencedora.

A gramática da brincadeira *duck, duck, goose* consiste na ocorrência da palavra `duck` ao menos uma vez, podendo esta repetir-se n vezes, até a ocorrência da palavra `goose`, encerrando a sequência. O autômato finito determinístico da Figura 4.15 descreve o reconhecimento de sentenças válidas da brincadeira apresentada.

Figura 4.15: Autômato finito determinístico que descreve o reconhecimento de sentenças válidas da brincadeira *duck, duck, goose* do Exemplo 14.



Fonte: autor.

As primitivas `WHILE` e `REPEAT` permitem que seja definida uma nova macro chamada `DDG` para obter uma sentença válida da brincadeira *duck, duck, goose* (de acordo com a Figura 4.15) através de repetição iterativa (Algoritmo 4.1). A Figura 4.16 apresenta duas versões usando as primitivas disponíveis.

Algoritmo 4.1 Sentença de *duck, duck, goose* através de repetição iterativa

```
1: procedure DDG1(n)
2:     d ← 1
3:     print duck
4:     while d < n do
5:         d ← d + 1
6:         print duck
7:     end while
8:     print goose
9: end procedure
```

(a) solução com teste *a priori*

```
1: procedure DDG2(n)
2:     print duck
3:     for i ← 1, n do
4:         print duck
5:     end for
6:     print goose
7: end procedure
```

(b) solução com repetição controlada

Conforme ilustra a Figura 4.16 a versão de definição da macro `DDG` utilizando a primitiva `WHILE` requer a definição explícita da condição a ser testada *a priori* para

Figura 4.16: Definição da macro DDG que obtém uma sentença válida da brincadeira *duck, duck, goose* através de repetição iterativa.

Versão com WHILE:

```
DEF[(DDG)(SET[(d)(0)] duck␣
WHILE[(LESS[(GET[(d)])(#1)])
(SET[(d)(INC[(GET[(d)])])]
duck␣)] goose)]
```

Versão com REPEAT:

```
DEF[(DDG)(duck␣
REPEAT[(#1)(duck␣)] goose)]
```

Fonte: autor.

entrada no laço e de uma variável auxiliar para contagem de ocorrências da impressão da palavra duck na cadeia de saída. A versão de definição da macro DDG com a primitiva REPEAT reduz a necessidade de teste e gerenciamento da variável contadora. Entretanto, ambas são equivalentes e produzem a mesma expansão, fornecidos os mesmos valores. □

As macros do Exemplo 4.2 viabilizam a geração de sentenças válidas para a brincadeira *duck, duck, goose*, através de repetição iterativa (primitivas WHILE e REPEAT). No Exemplo 15, utiliza-se recursão para a obtenção de tais sentenças.

Exemplo 15 (repetição recursiva)**.** De acordo com a Definição 6, o algoritmo de obtenção de uma sentença válida da brincadeira *duck, duck, goose* através de repetição iterativa (Exemplo 14, Algoritmo 4.1) pode ser expresso através de uma instrução condicional e recursão (Algoritmo 4.2). A nova versão de definição da macro DDG através de repetição recursiva é apresentada na Figura 4.17.

Algoritmo 4.2 Sentença de *duck, duck, goose* através de repetição recursiva

```
1: procedure DDG3(n)
2:     if n + 1 > 0 then
3:         print duck
4:         DDG3(n − 1)
5:     else
6:         print goose
7:     end if
8: end procedure
```

Conforme ilustra a Figura 4.17, a macro DDG, em sua versão recursiva, utiliza a primitiva condicional IF para testar o número de impressões restantes da palavra duck e, em caso positivo, chama-se a si mesma (recursão direta) com uma impressão a menos. Quando não houver impressões restantes, a macro imprime a palavra goose e retorna. De acordo com a Definição 6, as formas iterativa e recursiva (Figuras 4.16 e 4.17) são equivalentes. □

Figura 4.17: Definição da macro DDG que obtém uma sentença válida da brincadeira *duck, duck, goose* através de repetição recursiva.

```
DEF[(DDG)(IF[(GREATER[(INC[(#1)])(0)])(duck␣
DDG[(DEC[(#1)])])(goose)])]
```

Fonte: autor.

O Exemplo 15 apresentou uma versão recursiva para expressão da repetição na brincadeira *duck, duck, goose*. Observe que definição da macro DDG necessita obrigatoriamente de uma condição de parada (ou uma declaração explícita do número de passos de expansão através da primitiva EXS). O Exemplo 16 utiliza adaptatividade como forma alternativa para obtenção de sentenças válidas da mesma brincadeira.

Exemplo 16 (repetição por adaptatividade)**.** Considere uma forma alternativa para expressão de repetição na brincadeira *duck, duck, goose* (Exemplos 14 e 15) através do fenômeno de adaptatividade. As primitivas INSERT, REMOVE e QUERY identificam a ocorrência de padrões sintáticos no texto (no exemplo, símbolos contextuais fortemente inspirados no trabalho de Iwai (2000)) e efetuam as operações correspondentes, incluindo novas ocorrências da palavra duck e construtos condicionais para avaliação posterior.A nova versão de definição da macro DDG através de repetição por adaptatividade é apresentada na Figura 4.18.

Figura 4.18: Definição da macro DDG que obtém uma sentença válida da brincadeira *duck, duck, goose* através de repetição por adaptatividade.

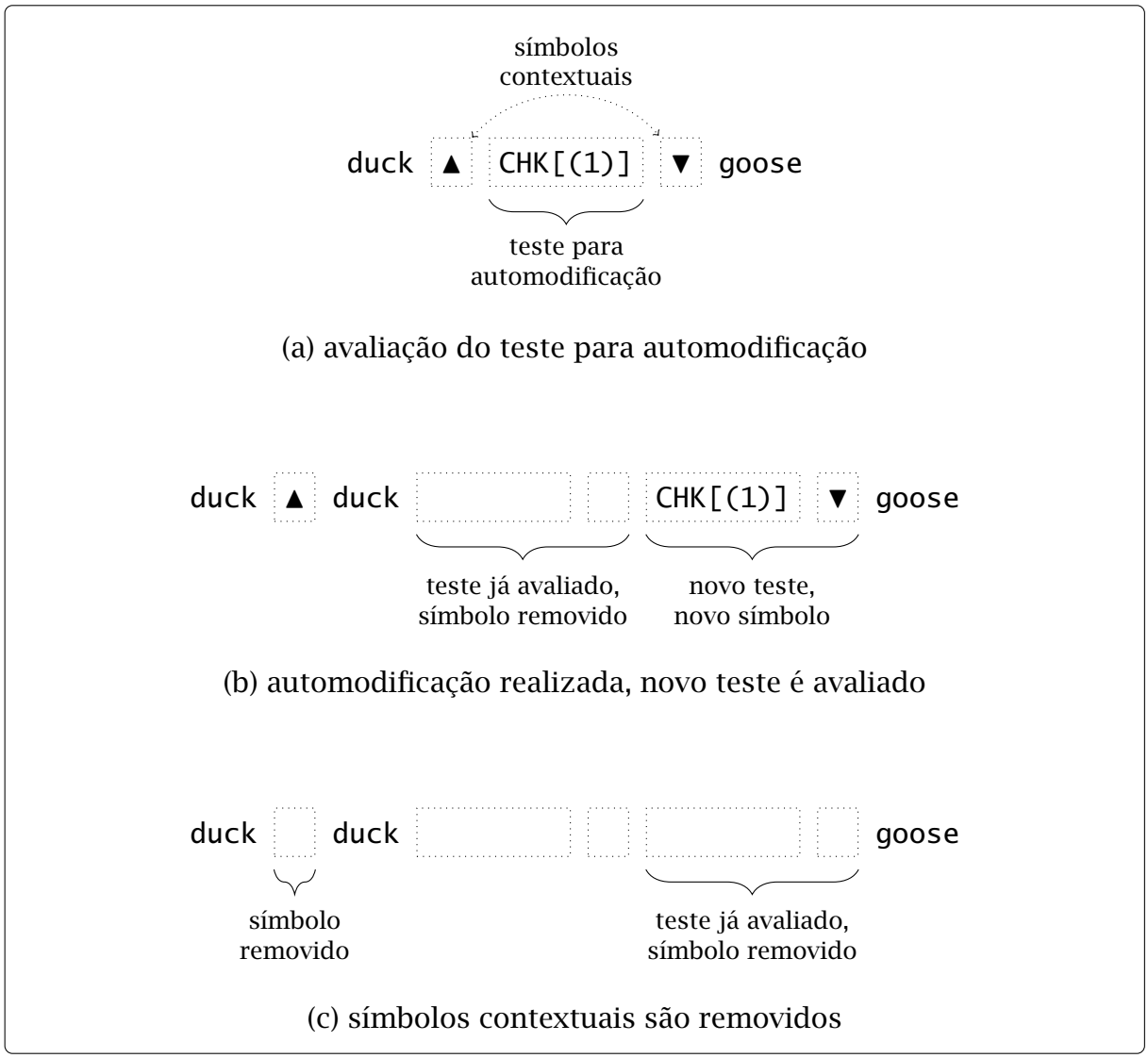
```
DEF[(CHK)(IF[(LESS[(GET[(d)])(#1)])(INSERT[(▲)(duck␣)]
QUERY[(▼)(t)] INSERT[(GET[(t)])(LIT[(CHK[(#1)])] ▼)]
REMOVE[(GET[(t)])] SET[(t)(INC[(GET[(t)])])])
(REMOVE(▲) REMOVE(▼))])]

DEF[(DDG)(SET[(d)(0)] duck␣ ▲ CHK[(#1)]) ▼ goose)]
```

Fonte: autor.

Conforme ilustra a Figura 4.18, os símbolos contextuais ▲ e ▼ são utilizados para sinalizar as posições de inserção de novas ocorrências da palavra duck e construtos condicionais ao longo do texto, respectivamente. Observe que a macro CHK é inserida literalmente em um ponto léxico seguinte à análise corrente. O teste é replicado até que o número de ocorrências de duck após o primeiro símbolo contextual corresponda ao valor da captura informada. Finalmente, o último teste remove os símbolos contextuais existentes. A Figura 4.19 apresenta um exemplo de expansão da instância de macro DDG[(1)], destacando as ações adaptativas elementares de

inserção, remoção e consulta viabilizadas através das primitivas `INSERT`, `REMOVE` e `QUERY` (Definição 73).

Figura 4.19: Exemplo de expansão da instância de macro `DDG[(1)]`, de acordo com a definição apresentada na Figura 4.18.



Fonte: autor.

De acordo com a Figura 4.19-a, a captura é substituída no corpo da macro `DDG` e a expansão propriamente dita inicia-se, da esquerda para a direita. As palavras `duck` e ▲ são reproduzidas literalmente na cadeia de saída, até a ocorrência da macro auxiliar `CHK`. Ao ser expandida, a condição de automodificação existente na definição de `CHK` é satisfeita, e portanto, a palavra `duck` é inserida após o símbolo contextual ▲, já processado. Adicionalmente, um novo teste é inserido após o símbolo contextual ▼, ainda não analisado. Tal símbolo é removido da sua posição corrente e inserido após a nova ocorrência da macro auxiliar `CHK`. A expansão corrente de `CHK` retorna ϵ (denotando uma cadeia vazia) e a cadeia assume a forma apresentada na Figura 4.19-b. O próximo *token* a ser analisado é a macro auxiliar recém-inserida no passo anterior.

O teste condicional é avaliado novamente e o número de ocorrências da palavra `duck` já atingiu o valor informado (no exemplo, apenas uma ocorrência). Como resultado da avaliação, `CHK` remove os símbolos contextuais da cadeia e encerra sua expansão. Finalmente, a palavra `goose` é reproduzida literalmente na cadeia de saída e a expansão de `DDG[(1)]` é concluída, conforme ilustra a Figura 4.19-c. Observe que a adaptatividade é utilizada para expressar o caso particular de repetição. □

O Exemplo 16 utilizou adaptatividade como subsídio para acomodar novos símbolos no texto sendo expandido. Observe que os símbolos contextuais atuaram como sinalizadores para inclusão da palavra `duck` e de um teste condicional, sendo removidos ao término da expansão. O Exemplo 17, a seguir, apresenta o cálculo de um termo da sequência de Fibonacci.

Exemplo 17 (sequência de Fibonacci)**.** Considere uma sequência de números inteiros positivos, na qual cada termo subsequente corresponde à soma dos dois anteriores. Tal sucessão de termos é conhecida como *sequência de Fibonacci* e sua fórmula é definida recursivamente na Equação 4.1. Ao aplicar a proporção entre dois termos consecutivos, obtém-se a razão ϕ (conhecida como *número de ouro* ou *razão áurea*, uma proporção universal de crescimento da natureza (LIVIO, 2003)).

$$F_i = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n \in \{1, 2\} \\ F_{n-1} + F_{n-2} & \text{caso contrário.} \end{cases} \tag{4.1}$$

O Algoritmo 4.3 apresenta a obtenção do n-ésimo termo da sequência de Fibonacci, de acordo com a Equação 4.1, de forma recursiva. A definição da macro `FIB` correspondente é ilustrada na Figura 4.20.

Algoritmo 4.3 Obtenção do n-ésimo termo da sequência de Fibonacci

```
1: function FIB(n)
2:     if n > 2 then
3:         return FIB(n − 1) + FIB(n − 2)
4:     else if n = 0 then
5:         return 0
6:     else
7:         return 1
8:     end if
9: end function
```

Conforme ilustra a Figura 4.20, a macro `FIB` utiliza a primitiva de soma aritmética `ADD` para calcular os dois termos consecutivos anteriores ao índice informado. É importante destacar que o Algoritmo 4.3 apresenta complexidade exponencial e torna-se impraticável à medida que n aumenta (por exemplo, para $n = 20$ serão

Figura 4.20: Definição da macro FIB que obtém o n-ésimo termo da sequência de Fibonacci, de acordo com a Equação 4.1.

```
DEF[(FIB)(IF[(GREATER[(#1)(2)])(ADD[(FIB[(DEC[(#1)])])
(FIB[(DEC[(DEC[(#1)])])])])(IF[(EQUAL[(#1)(0)])(0)(1)])])]
```

Fonte: autor.

efetuadas 13528 chamadas recursivas). Neste caso, em particular, o número de chamadas recursivas cresce aproximadamente na mesma proporção dos próprios números de Fibonacci (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2006). Tal solução apresentada neste exemplo é de caráter puramente didático. □

O Exemplo 17 ilustrou o cálculo de um termo da sequência de Fibonacci, de forma recursiva. O Exemplo 18, a seguir, introduz operações aritméticas fundamentais expressas utilizando recursão, incremento e decremento (sucessor e antecessor, respectivamente), e controle de fluxo.

Exemplo 18 (soma, subtração truncada, multiplicação e divisão de números inteiros através de indução)**.** Considere uma forma alternativa para expressão das operações aritméticas de soma, subtração truncada, multiplicação e divisão de números inteiros através de indução (Algoritmo 4.4), fortemente inspirada na teoria de computabilidade (ENDERTON, 2010) e lógica de aritmética computacional (FLORES, 1963). As definições das macros AD, SB, ML e DV são apresentadas na Figura 4.21.

Figura 4.21: Definições das macros AD, SB, ML e DV para expressão das operações aritméticas de soma, subtração truncada, multiplicação e divisão de números inteiros através de indução.

```
DEF[(AD)(IF[(EQUAL[(#1)(0)])(#2)
(INC[(AD[(DEC[(#1)])(#2)])])])]

DEF[(SB)(IF[(GREATER[(#2)(#1)])(0)(IF[(EQUAL[(#2)
(0)])(#1)(DEC[(SB[(#1)(DEC[(#2)])])])])])]

DEF[(ML)(IF[(EQUAL[(#1)(0)])(0)
(ADD[(ML[(DEC[(#1)])(#2)])(#2)])])]

DEF[(DV)(IF[(OR[(EQUAL[(#2)(0)])(LESS[(#1)
(#2)])])(0)(IF[(EQUAL[(#1)(#2)])(1)
(INC[(DV[(SUB[(#1)(#2)])(#2)])])])])]
```

Fonte: autor.

Algoritmo 4.4 Operações aritméticas em números inteiros através de indução

```
1: function AD(m, n)
2:     if m = 0 then
3:         return n
4:     else
5:         return AD(m − 1, n) + 1
6:     end if
7: end function
```

(a) soma

```
1: function ML(m, n)
2:     if m = 0 then
3:         return 0
4:     else
5:         return n + ML(m − 1, n)
6:     end if
7: end function
```

(b) multiplicação

```
1: function SB(m, n)
2:     if n > m then
3:         return 0
4:     else if n = 0 then
5:         return m
6:     else
7:         return SB(m, n − 1) − 1
8:     end if
9: end function
```

(c) subtração truncada

```
1: function DV(m, n)
2:     if n = 0 ∨ m < n then
3:         return 0
4:     else if m = n then
5:         return 1
6:     else
7:         return 1 + DV(m − n, n)
8:     end if
9: end function
```

(d) divisão

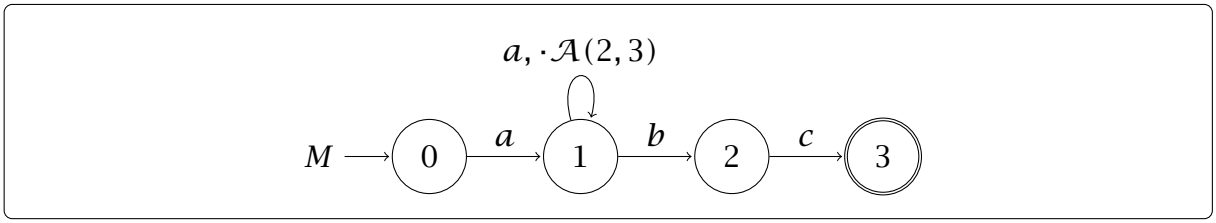
Conforme ilustra a Figura 4.21, as macros `AD`, `SB`, `ML` e `DV` utilizam as primitivas de controle de fluxo para definir as bases de indução e os passos indutivos para as operações aritméticas de soma, subtração truncada, multiplicação e divisão de números inteiros. Observe que a operação de subtração truncada (do inglês *monus*) não admite resultado negativo, tal que $a \dot{-} b = 0$ se $a < b$, ou $a - b$ caso contrário (ROBIČ, 2015). Por razões de eficiência, recomenda-se o uso das primitivas apresentadas na Definição 66 para operações aritméticas. □

As macros apresentadas no Exemplo 18 são elegantes do ponto de vista matemático e podem ser utilizadas como material de apoio em cursos de computação. O Exemplo 19, a seguir, ilustra a geração de sentenças dependentes de contexto através de adaptatividade.

Exemplo 19 (geração de sentenças dependentes de contexto)**.** Considere uma linguagem dependente de contexto $L = \{w \in \{a,b,c\}^* \mid w = a^n b^n c^n, n \in \mathbb{N}, n \geq 1\}$. O autômato adaptativo M que reconhece cadeias pertencentes a tal linguagem é ilustrado na Figura 4.22 (JOSÉ NETO, 1993, 1994).

Uma possível forma de geração de uma sentença válida na linguagem L utilizando adaptatividade decorre da inserção progressiva e controlada das dependências contextuais nos pontos léxicos correspondentes,explicitamente indicados por símbolos

Figura 4.22: Autômato adaptativo M que reconhece cadeias pertencentes à linguagem $L = \{w \in \{a,b,c\}^* \mid w = a^n b^n c^n, n \in \mathbb{N}, n \geq 1\}$. A função adaptativa $\mathcal{A}$ é apresentada no Algoritmo 4.5.



Fonte: José Neto (1993).

Algoritmo 4.5 Função adaptativa $\mathcal{A}(p_1, p_2)$

função adaptativa $\mathcal{A}(p_1, p_2)$
variáveis: $?x, ?y$
geradores: $g_1^\star, g_2^\star$
$?(?x, b) \rightarrow p_1$
$-(?x, b) \rightarrow p_1$
$?(?y, c) \rightarrow p_2$
$-(?y, c) \rightarrow p_2$
$-(1, a) \rightarrow 1, \cdot\mathcal{A}(p_1, p_2)$
$+(?x, b) \rightarrow g_1^\star$
$+(g_1^\star, b) \rightarrow p_1$
$+(?y, c) \rightarrow g_2^\star$
$+(g_2^\star, c) \rightarrow p_2$
$+(1, a) \rightarrow 1, \cdot\mathcal{A}(g_1^\star, g_2^\star)$
fim da função adaptativa

semelhantes aos utilizados no Exemplo 16. A definição da macro ABC é apresentada na Figura 4.23.

Figura 4.23: Definição da macro ABC para geração de sentenças dependentes de contexto, na forma $a^n b^n c^n$ através de adaptatividade.

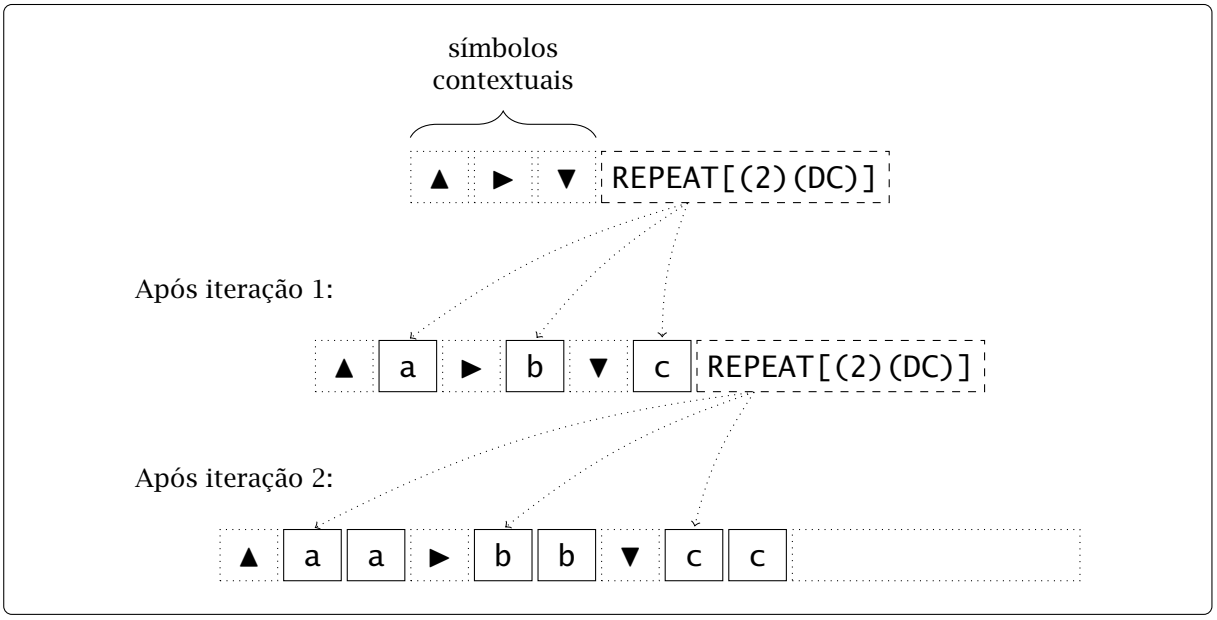
```
DEF[(DC)(INSERT[(▲)(a)] INSERT[(►)(b)] INSERT[(▼)(c)])]

DEF[(ABC)(▲ ► ▼ REPEAT[(#1)(DC)]
REMOVE[(▲)] REMOVE[(►)] REMOVE[(▼)])]
```

Fonte: autor.

Conforme ilustra a Figura 4.23, a macro ABC insere os símbolos contextuais ▲, ► e ▼, e expande a macro auxiliar DC através da primitiva REPEAT de acordo com o valor fornecido como captura. A cada iteração, a, b e c são inseridos à direita de seus símbolos contextuais correspondentes, em decorrência da expansão da macro auxiliar DC e da ação das primitivas de adaptatividade. Ao término da repetição controlada, os símbolos contextuais são removidos. A Figura 4.24 apresenta um trecho

de expansão da instância da macro `ABC[(2)]`, destacando a inserção dos símbolos durante as duas iterações previstas. Por razões de legibilidade, as primitivas adaptativas de remoção foram omitidas.

Figura 4.24: Trecho de expansão da instância da macro `ABC[(2)]`, destacando a inserção dos símbolos durante as duas iterações previstas. Por razões de legibilidade, as primitivas adaptativas de remoção foram omitidas.



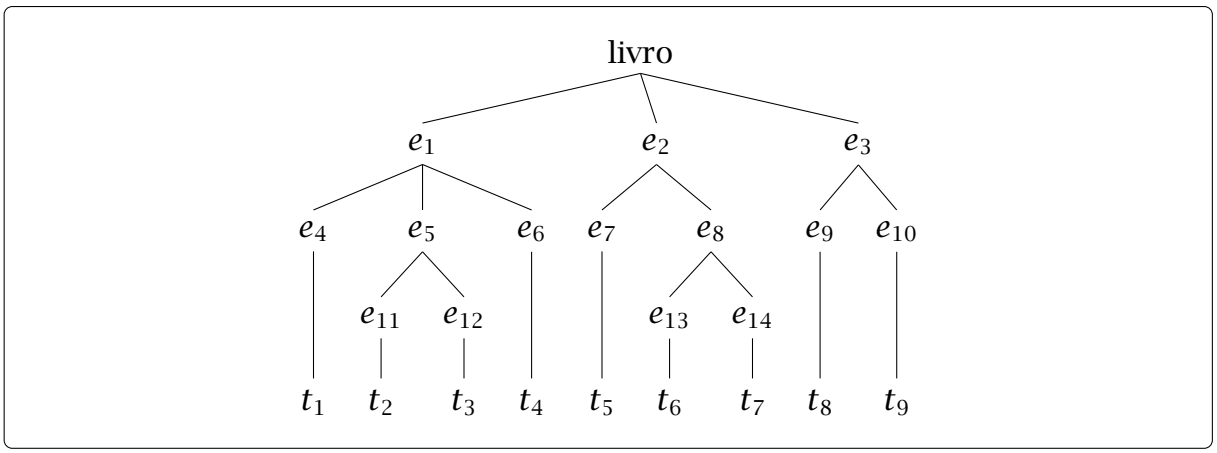
Fonte: autor.

De acordo com a Figura 4.24, na primeira iteração, a macro auxiliar `DC` insere um `a` após ▲, um `b` após ►, e um `c` após ▼, resultando na cadeia `abc` (obviamente, omitindo os símbolos contextuais), válida para $n = 1$. Na segunda iteração, `DC` insere novamente um `a` após ▲ e antes do símbolo `a` já existente (inserido na iteração anterior), e, assim, repete a mesma lógica para os símbolos `b` e `c`, resultando na cadeia `aabbcc`, válida para $n = 2$. Observe que, na i-ésima iteração, a cadeia w é modificada para acomodar novos símbolos que representam a dependência de contexto corrente, tal que $w_i = a^i b^i c^i$ (com exceção dos símbolos contextuais). □

O Exemplo 19 apresentou a geração de sentenças dependentes de contexto através de adaptatividade. O Exemplo 20, a seguir, contempla um projeto de editoração de livro, com a inclusão de uma primitiva para exibição de uma caixa de diálogo de entrada de dados, utilizando recursos do sistema operacional.

Exemplo 20 (editoração de livro)**.** Considere um projeto de editoração de livro, no qual uma sequência de elementos textuais hierarquicamente organizados determina a estrutura para composição do documento. A Figura 4.25 apresenta um exemplo de organização de um livro em capítulos, seções e subseções, dispostos em uma estrutura de árvore, tal que e_i e t_j representam um elemento hierárquico e um bloco de texto, respectivamente.

Figura 4.25: Exemplo de organização de um livro em capítulos, seções e subseções, dispostos em uma estrutura de árvore.



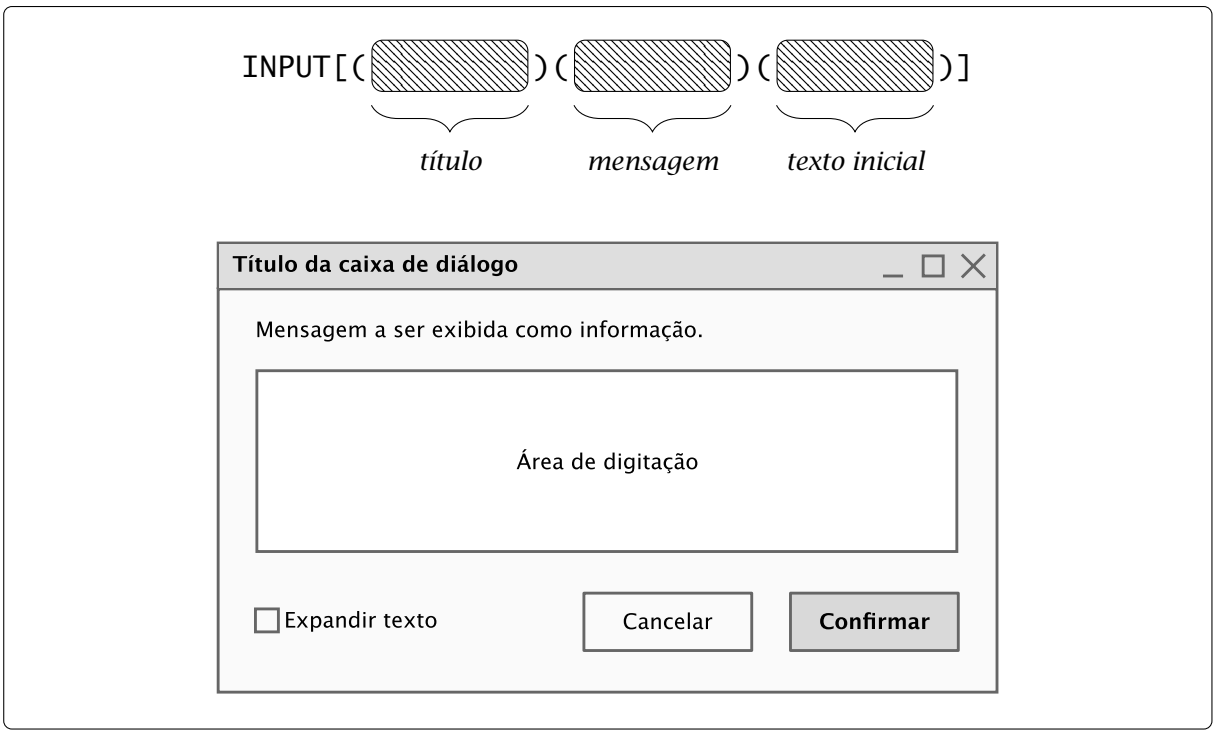
Fonte: autor.

Conforme ilustra a Figura 4.25, o projeto de editoração de um livro pode ser visto como uma gramática, na qual cada elemento textual constitui um símbolo não-terminal. A geração de uma sentença de tal gramática resulta, portanto, na composição efetiva do livro em elaboração (CEREDA, 2013).

Considere a definição das macros `CHAPTER`, `SECTION` e `SUBSECTION` para viabilização do projeto de editoração de um livro, inspiradas no exemplo da Figura 4.25. Cabe ao autor especificar a gramática de seu livro através de tais macros e, em tempo de expansão, os elementos textuais serão convenientemente preenchidos. Como resultado, obtém-se um formato textual intermediário para processamento posterior. Possíveis formatos intermediários incluem CommonMark (MACFARLANE, 2017) e XML (do inglês *extensible markup language*, ou *linguagem de marcação extensível* no vernáculo) (EVJEN et al., 2007). Tais macros utilizam uma primitiva especial `INPUT` para exibição de uma caixa de diálogo de entrada de dados, utilizando recursos do sistema operacional, conforme ilustra a Figura 4.26. Opcionalmente, o texto digitado pode ser expandido através da marcação explícita da caixa de seleção (do inglês *checkbox*) disponibilizada na interface. É possível estender o conjunto de primitivas da Seção 4.1 para incluir a primitiva `INPUT` através de ligações explícitas de bibliotecas de elementos de interface gráfica do usuário com a linguagem Lua (por exemplo, GTK, Qt e Motif). O Apêndice D apresenta uma implementação alternativa de um expansor de macros utilizando a linguagem Java, com a inclusão de primitivas de componentes visuais através das classes utilitárias Swing (GOSLING et al., 2014).

As definições das macros `CHAPTER`, `SECTION` e `SUBSECTION` para a especificação de capítulos, seções e subseções de um livro são apresentadas na Figura 4.27. É importante destacar que, embora suas definições sejam semelhantes, tais macros representam elementos textuais específicos e, portanto, constituem abstrações distintas. Por razões de simplicidade, CommonMark foi escolhido como formato textual

Figura 4.26: Sintaxe e operação da primitiva de exibição de uma caixa de diálogo de entrada de dados, utilizando recursos do sistema operacional.



Fonte: autor.

intermediário resultante da expansão das macros de editoração.

Figura 4.27: Definições das macros CHAPTER, SECTION e SUBSECTION para a especificação de capítulos, seções e subseções de um livro.

```
DEF[(CHAPTER)(INPUT[(New chapter)(Please type the␣
content for chapter #1)(# #2\n\n#3)])]

DEF[(SECTION)(INPUT[(New section)(Please type the␣
content for section #1)(## #2\n\n#3)])]

DEF[(SUBSECTION)(INPUT[(New subsection)(Please type␣
the content for subsection #1)(### #2\n\n#3)])]
```

Fonte: autor.

Como exemplo, considere a editoração da subárvore e_3 (excerto da Figura 4.25), composta por um capítulo e duas seções. A Figura 4.28 apresenta o texto a ser expandido, contendo as ocorrências das macros CHAPTER e SECTION, e sua expansão propriamente dita, incluindo a exibição das caixas de diálogo de entrada de dados correspondentes.

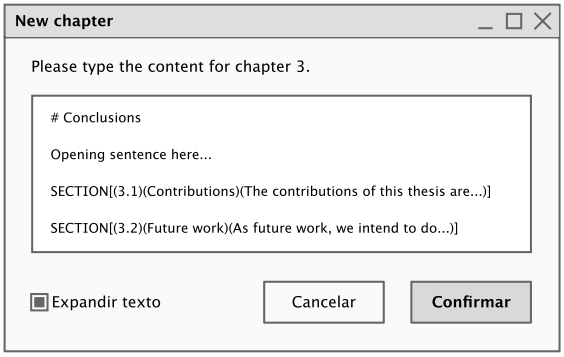
Conforme ilustra a Figura 4.28, por ocasião da expansão da macro CHAPTER, uma caixa de diálogo de entrada de dados correspondente é disponibilizada ao usuário,

Figura 4.28: Editoração da subárvore e_3 (excerto da Figura 4.25), composta por um capítulo e duas seções.

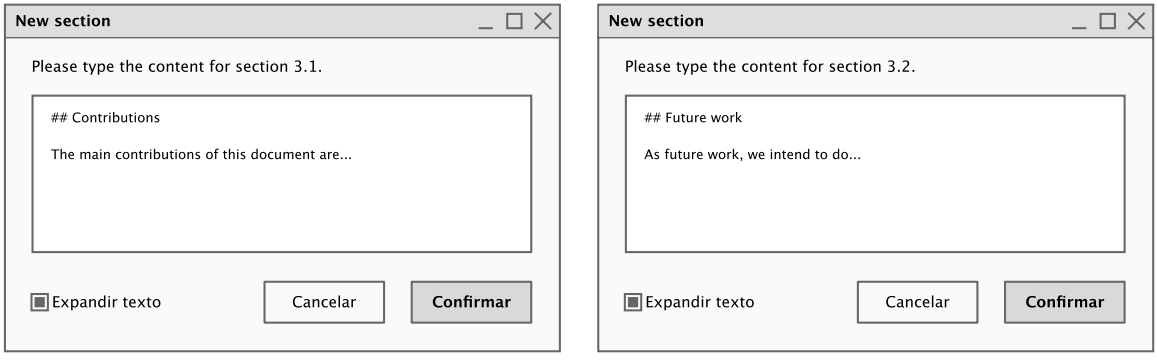
(a) texto a ser expandido

```
CHAPTER[(3)(Conclusions)(Opening sentence here...\n\n
SECTION[(3.1)(Contributions)(The contributions␣
of this thesis are...)] \n\n SECTION[(3.2)(Future␣
work)(As future work, we intend to do...)])]
```

(b) edição do capítulo 3



(c) edição das seções 3.1 e 3.2



Fonte: autor.

com o texto inicial devidamente inserido na área de digitação. Após a confirmação da inserção do texto por parte do usuário, o expansor analisa a sequência resultante e encontra duas ocorrências da macro `SECTION`, expandidas sequencialmente em um momento posterior, de modo idêntico ao tratamento da macro `CHAPTER` (incluindo a exibição das caixas de diálogo de entrada de dados e interação do usuário). Ao término de tais expansões, o texto resultante encontra-se livre de macros e o processamento efetivamente termina. É importante destacar que o adiamento da expansão do texto digitado na caixa de diálogo (através da desabilitação da caixa de seleção) permite que o usuário postergue a editoração do livro para um momento oportuno, em virtude da reprodução literal das macros internas no texto de saída. Alternativamente, é possível combinar diferentes estratégias de expansão para elementos textuais distintos no projeto de editoração. □

Os exemplos apresentados nesta seção são de caráter puramente didático. Entretanto, os conceitos aplicados em tais exemplos podem ser combinados para a definição de abstrações mais complexas e expressivas, conforme o caso de uso e a conveniência.

CAPÍTULO 5

CONCLUSÕES

> We're all stories, in the end.
> Just make it a good one.
>
> DOCTOR WHO

Esta seção apresenta as conclusões desta tese, incluindo discussões sobre o conceito de reescrita de termos, aplicações (abrangendo processamento de linguagem natural) e comentários acerca da semântica operacional das primitivas de adaptatividade. Adicionalmente, são destacadas as contribuições, acentuando os principais resultados obtidos, e apresentadas recomendações para trabalhos futuros.

5.1 DISCUSSÕES

Sistemas de reescrita de termos podem fruir de propriedades desejáveis ou convenientes de acordo com as motivações de utilização. Como exemplo, considere o sistema proposto por Bove e Arbilla (1991, 1992), o qual é provado ser Church-Rosser (ordem das reduções não altera o resultado final), confluente (existem várias formas de se reescrever o sistema, sempre com o mesmo resultado) e noetheriano (não existem sequências infinitas de redução). Restrições aplicadas ao sistema reforçam a efetivação de tais propriedades (por exemplo, através da validação da tabela de macros em tempo de definição) em potencial detrimento de expressividade.

Pragmaticamente, ainda que proporcionem garantias teóricas significativas (como a própria terminação da reescrita de termos), a ausência de certas propriedades não compromete a utilização de um sistema de reescrita de termos de propósito geral. Consequentemente, cabe ao usuário o ônus da definição de um conjunto de relações de reescrita com comportamento previsível ou substancialmente observável (por exemplo, definindo macros recursivas que efetivamente param, como ilustram os exemplos da Subseção 4.2). Este é o caso do expansor de macros apresentado no escopo desta tese.

Do ponto de vista de implementação, é importante destacar o aspecto temporal da expansão de macros. A análise e processamento dos elementos sintáticos de uma sentença ocorrem sequencialmente, admitindo a escrita da esquerda para a direita como convenção de leitura. Assim, o expansor de macros aplica as transformações sob demanda, conforme a ocorrência dos padrões sintáticos ao longo do texto. A circunstância temporal decorrente atua como um vínculo operacional.

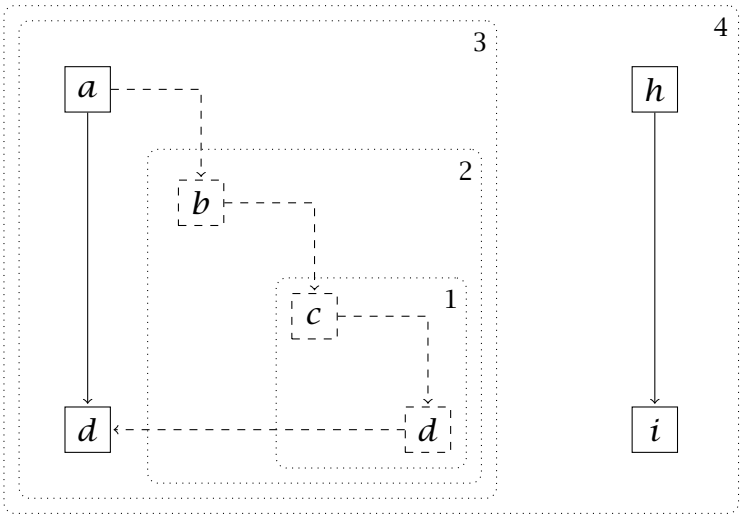
É possível utilizar sistemas de reescrita para construir abstrações de acordo com a ocorrência de certos padrões sintáticos em um texto, reduzindo-os a construtos sintetizados, conforme ilustra o Exemplo 4 (problema da lata de café). Eventualmente, tais construtos armazenam características particulares dos padrões sintáticos originais na forma de metadados descritivos (por exemplo, número, gênero e grau de palavras de um idioma) (ZENG; QIN, 2016). De fato, Cereda, Miura e José Neto (2018) exploram tal iniciativa para análise sintática de sentenças em linguagem natural utilizando adaptatividade. A relação de dependências contextuais entre os elementos sentenciais é modelada através de um sistema de reescrita hierarquicamente organizado.

A captura de metadados descritivos provenientes dos padrões sintáticos identificados em um texto viabiliza o projeto e implementação de aplicações e componentes de etiquetagem e classificação, tais como analisadores léxicos e sintáticos. Adicionalmente, o uso de bibliotecas específicas de domínio incorporadas nas transformações algorítmicas (por exemplo, etiquetagem e agrupamento de *tokens* em unidades linguísticas através da biblioteca NLTK (LOPER; BIRD, 2002) disponibilizada para a linguagem Python) favorece um tratamento adequado e direcionado às abstrações em construção. Entretanto, é recomendável não extrapolar a complexidade além do prescrito no nível de abstração corrente (por exemplo, não há razão de uma análise detalhada de estruturas sintáticas no nível léxico).

A semântica operacional das primitivas de adaptatividade (Definição 73) depende diretamente de uma política de visibilidade definida para as ações adaptativas elementares. A utilização de escopos mais restritos, ainda que condicione a abrangência das ações adaptativas elementares a uma região reduzida, pode mitigar a ocorrência de efeitos colaterais potenciais em tempo de expansão. Por exemplo, a expansão de uma macro recursiva que insere novos símbolos no texto antes de seu ponto léxico possivelmente referenciará posições de retorno inválidas. A Figura 5.1 apresenta um conjunto de políticas de visibilidade das ações adaptativas em um expansor de macros de propósito geral, com as regras de reescrita (a, b), (b, c), (c, d) e (h, i).

Conforme ilustra a Figura 5.1, são propostas quatro políticas de visibilidade. A política 1 trata de visibilidade local, na qual as ações adaptativas estão condicionadas tão somente à expansão corrente. A política 2, por sua vez, determina um escopo hierárquico ascendente limitado, cuja abrangência incorpora o escopo corrente e uma eventual expansão externa (totalizando dois níveis). A política 3 remove a limitação imposta pela política anterior e determina um escopo hierárquico ascendente irrestrito, no qual todos os escopos anteriores à expansão corrente são considerados (de acordo com uma pilha de chamadas de expansão). Finalmente, a política 4 trata de visibilidade global, na qual as ações adaptativas abrangem a região

Figura 5.1: Políticas de visibilidade das ações adaptativas em um expansor de macros de propósito geral, com as regras de reescrita (a, b), (b, c), (c, d) e (h, i).



Legenda: (1) visibilidade local, (2) escopo hierárquico ascendente limitado, (3) escopo hierárquico ascendente irrestrito, e (4) visibilidade global.

Fonte: autor.

completa, extrapolando a expansão corrente e eventuais expansões externas.

A redução da região de abrangência nas políticas 1 a 3 tem impacto significativo na expressividade das primitivas de adaptatividade. Assim, recomenda-se preservar a visibilidade global das ações adaptativas no caso geral e limitar sua abrangência em resoluções particulares, específicas de domínio, conforme a conveniência. O expansor de macros proposto nesta tese adota a política 4 (global) como padrão de visibilidade. A Seção A.4 discute os efeitos colaterais resultantes de ações adaptativas, do ponto de vista de implementação.

As ações adaptativas elementares de inclusão, remoção e consulta (Definição 39) podem ser sintetizadas em uma ação de substituição. Sejam A e B conjuntos de padrões sintáticos, tal que A contém todos os padrões encontrados no texto corrente, p denota uma variável, Ξ é a ação de substituição proposta, e $\Xi(x, y)$ denota a troca de x por y, $x \leftarrow y$. As equivalências são apresentadas na Equação 5.1.

$$\begin{aligned} +(B) &\equiv \Xi(A, A \cup B) \\ -(B) &\equiv \Xi(A, A - B) \\ ?(B, p) &\equiv \Xi(A, A) \text{ e preenche } p \text{ com} \\ &\qquad \text{ocorrências de } B \text{ em } A \end{aligned} \tag{5.1}$$

Do ponto de vista de implementação, a disponibilização de uma primitiva hipotética REPLACE proporciona uma alternativa conceitual às primitivas de adaptatividade existentes. Entretanto, estudos adicionais são necessários para indicar eventuais

vantagens e desvantagens operacionais do uso desta primitiva em relação às outras na identificação e manipulação de padrões sintáticos no texto em análise.

5.2 CONTRIBUIÇÕES

Os resultados mais significativos apresentados no escopo desta tese são enumerados a seguir, destacando a utilização de sistemas de reescrita de termos como mecanismos de abstração em transformações textuais.

i) Unificação da terminologia acerca de macros, fundamentada em um resgate histórico, incluindo suas propriedades mais significativas.

ii) Técnicas de projeto e aspectos de implementação de um expansor de macros de propósito geral utilizando linguagens de programação convencionais.

iii) Apresentação de um método de desenvolvimento de um expansor de macros através de uma estratificação em camadas representando níveis de abstração.

iv) Utilização de motores de eventos como modelos de implementação para os níveis de abstração de um expansor de macros, tal que a composição funcional destes resulte no projeto completo.

v) Proposta e implementação de um conjunto de primitivas para definição de macros, controle de fluxo de expansão e operações básicas sobre alguns tipos de dados (Seção A.4).

vi) Definição e implementação de um conjunto de primitivas de adaptatividade, viabilizado de acordo com as ações adaptativas elementares de inserção, remoção e consulta.

vii) Disponibilização de exemplos de expansão de macros proporcionados através do expansor de macros projetado no decorrer desta tese (Seção 4.2).

Adicionalmente, a pesquisa desenvolvida obteve resultados secundários significativos, direta ou indiretamente relacionados aos conceitos principais contemplados nesta tese. Tais resultados são enumerados a seguir e detalhados nos apêndices ou publicações correspondentes.

i) Disponibilização de uma biblioteca para a implementação de autômatos adaptativos, utilizando a linguagem Java, de forma consistente e aderente à teoria original proposta por José Neto (1993) (CEREDA; JOSÉ NETO, 2016).

ii) Disponibilização de uma ferramenta para geração automática de autômatos adaptativos, utilizando a linguagem Java, a partir de especificações no formato XML (CEREDA; JOSÉ NETO, 2017c).

iii) Definição de um conjunto de métricas de instrumentação para dispositivos adaptativos dirigidos por regras, com a finalidade de oferecer subsídios para implementações eficientes (CEREDA; JOSÉ NETO, 2017b).

iv) Disponibilização de uma ferramenta para geração de motores de eventos, utilizando a linguagem Java, a partir de especificações no formato YAML (Apêndice C).

v) Disponibilização de um expansor de macros de propósito geral, implementado utilizando linguagem Java, com delimitadores dependentes de contexto e primitivas de componentes visuais (Apêndice D).

vi) Proposta de extensão da notação de Wirth tradicional para eventual inclusão de metadados descritivos em símbolos terminais e não-terminais, e geração automática de autômatos com estados anotados (Apêndice B).

vii) Definição de um sistema de reescrita dirigido por regras utilizando tecnologia adaptativa para análise sintática de sentenças em linguagem natural (CEREDA; MIURA; JOSÉ NETO, 2018).

viii) Disponibilização de ferramentas para geração e execução de autômatos de pilha estruturados a partir de gramáticas escritas na notação de Wirth (CEREDA; JOSÉ NETO, 2017a).

ix) Disponibilização de uma ferramenta para geração de sentenças a partir de gramáticas livres de contexto, com a possibilidade de inclusão de restrições positivas e negativas acerca das produções (Apêndice E).

Bibliotecas, ferramentas e publicações apresentadas nesta tese estão disponibilizadas na página do autor na plataforma de hospedagem de código-fonte GitHub e no website do Laboratório de Linguagens e Técnicas Adaptativas:

`https://github.com/cereda`
`https://lta.poli.usp.br`

Em particular, as bibliotecas e ferramentas desenvolvidas estão disponibilizadas sob licença de código aberto (ou *open source* no original, em inglês).

5.3 TRABALHOS FUTUROS

Como desdobramento da pesquisa desenvolvida nesta tese, uma lista não-exaustiva de oportunidades de trabalhos futuros em reescrita de termos, tecnologia adaptativa e temas correlatos é recomendada a seguir.

i) Formalização de um cálculo de expansão de macros com suporte temporal pleno através de lógicas de intervalos adequadas (por exemplo, inspirado no trabalho de Zhou e Hansen (2004) para o desenvolvimento de sistemas de tempo real).

ii) Estudo sobre a obtenção de metadados estruturais em padrões sintáticos para viabilização de adaptatividade multinível (por exemplo, estendendo a formulação algébrica proposta por Silva Filho (2011) para o autômato finito adaptativo de segunda ordem).

iii) Definição de primitivas mais expressivas e convenientes ao usuário para representação do fenômeno da adaptatividade (por exemplo, inspiradas nos decisores e conectores da arquitetura proposta por Silva (2011), incluindo recomendações de projeto).

iv) Estudo sobre o impacto das políticas de visibilidade das ações adaptativas elementares em um expansor de macros de propósito geral apresentadas na Seção 5.1 em relação às potenciais limitações contextuais introduzidas (por exemplo, inspirado nos resultados preliminares reportados por Cereda e José Neto (2015b) acerca dos desafios para codificação de programas com características adaptativas).

v) Estudo sobre a utilização de referências sincronizadas para gerenciamento de dependências de contexto (por exemplo, inspirado nos símbolos contextuais e produções de injeção de dependência do trabalho de Iwai (2000) acerca de um formalismo gramatical adaptativo).

vi) Desenvolvimento de técnicas para geração automática de expansores de macros de propósito geral a partir de especificações gramaticais (por exemplo, inspirado no trabalho de Fisher (1982) para geração de analisadores sintáticos a partir de gramáticas de dois níveis).

As recomendações de trabalhos futuros apresentadas nesta seção podem ser estendidas conforme os avanços teóricos e experimentais nos temas de pesquisa contemplados nesta tese.

5.4 CONSIDERAÇÕES FINAIS

Macros constituem um mecanismo significativo para representação de artefatos em um determinado nível de abstração, sem a necessidade da exposição excessiva de detalhes ou características particulares, viabilizando estruturas mais convenientes e aderentes às necessidades do usuário. Transformações simbólicas e algorítmicas conferem a tal mecanismo expressividade e poder computacional.

As modificações contextuais viabilizadas de forma simplificada através das primitivas de adaptatividade `INSERT`, `REMOVE` e `QUERY` (conforme ilustram os Exemplos 16 e 19) corroboram para a expressividade e abrangência do fenômeno adaptativo em relação ao seu correspondente não-adaptativo convencional (por exemplo, implementado através de variáveis e estruturas condicionais). Consequentemente, a representação da abstração torna-se mais compacta.

É importante destacar que o conceito de macro extrapola sua vertente textual. Moraes (2006) substitui a chamada de submáquina tradicional em um autômato adaptativo por uma expansão desta, dispensando o uso da pilha sintática. Em linhas gerais, uma cópia da submáquina que originalmente seria chamada é incorporada à topologia corrente, com as devidas ligações (por exemplo, transições em vazio dos estados de aceitação para o estado de destino da chamada). Neste caso, o conceito de macro pode ser entendido como uma forma de adaptatividade mais restrita. Alternativamente, é possível definir uma função adaptativa para replicação de submáquinas, tal que esta atue semanticamente como uma expansão.

Em particular, a aplicação de transformações simbólicas e algorítmicas através de um expansor de macros de propósito geral garante ao usuário a adequação do texto a um determinado nível de abstração, representado através de um conjunto de padrões sintáticos e regras de reescrita. Tais adequações determinam o detalhamento e granularidade esperados no texto resultante, em sua forma normal.

Espera-se, portanto, que esta tese contribua significativamente para a difusão do conceito de macro como mecanismo de abstração em transformações textuais, incluindo o fomento de pesquisas teóricas e experimentais nas áreas de sistemas de reescrita e tecnologia adaptativa.

REFERÊNCIAS BIBLIOGRÁFICAS

ADAMS, N. I. et al. Revised report on the algorithmic language Scheme. *SIGPLAN Notices*, v. 33, n. 9, p. 26–76, sep 1998.

AHO, A. V.; ULLMAN, J. D. *Foundations of Computer Science*. [S.l.]: W. H. Freeman and Company, 1995.

AMERICAN STANDARDS ASSOCIATION. *American Standard Code for Information Interchange*. New York, NY, 1963.

ANDERSEN, J.; BRABRAND, C. Syntactic language extension via an algebra of languages and transformations. *Electronic Notes in Theoretical Computer Science*, v. 253, n. 7, p. 19–35, sep 2009.

ANDERSEN, J.; BRABRAND, C.; CHRISTIANSEN, D. R. Banana algebra: Compositional syntactic language extension. *Science of Computer Programming*, v. 78, n. 10, p. 1845–1870, sep 2013.

ANZAI, Y.; SIMON, H. A. The theory of learning by doing. *Psychological Review*, n. 86, p. 124–140, 1979.

BAADER, F.; NIPKOW, T. *Term rewriting and all that*. New York: Cambridge University, 1998.

BACHRACH, J.; PLAYFORD, K. *D-Expressions: LISP power, Dylan style*. [S.l.], 1999.

BALABAN, M.; BARZILAY, E.; ELHADAD, M. Abstraction as a means for end-user computing in creative applications. *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans*, v. 32, n. 6, p. 640–653, nov 2002.

BARENDREGT, H. P. *The Lambda Calculus, Its Syntax and Semantics*. [S.l.]: North Holland, 1985. v. 103. (Studies in Logic and the Foundations of Mathematics, v. 103).

BARKER, J. A. *Paradigms: the business of discovering the future*. New York: Harper Business, 1992.

BEN-KIKI, O.; EVANS, C.; DÖT NET, I. *YAML Ain't Markup Language (YAML) version 1.2 specification*. 2009.

BENNINGHOFEN, B.; KEMMERICH, S.; RICHTER, M. M. *Systems of Reductions*. [S.l.]: Springer-Verlag, 1987. (Lecture Notes in Computer Science, 277).

BERT, D.; ECHAHED, R. Abstraction of conditional term rewriting systems. In: INTERNATIONAL SYMPOSIUM ON LOGIC PROGRAMMING, 1995, **Proceedings...**. [S.l.]: MIT, 1995. p. 162–176.

BERTOT, Y.; CASTÉRAN, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. [S.l.]: Springer-Verlag, 2004.

BÖHM, C.; JACOPINI, G. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, v. 9, n. 5, p. 366–371, may 1966.

BOICHUT, Y. et al. Finer is better: Abstraction refinement for rewriting approximations. In: INTERNATIONAL CONFERENCE ON REWRITING TECHNIQUES AND APPLICATIONS, **Proceedings...**. [S.l.: s.n.], 2008. (Lecture Notes in Computer Science, v. 5117), p. 48–62.

BOOCH, G. et al. *Object-Oriented Analysis and Design with Applications*. [S.l.]: Addison-Wesley, 2007.

BOVE, A.; ARBILLA, L. *A Confluent calculus of macro expansion and evaluation*. Montevideo, Uruguay, 1991.

BOVE, A.; ARBILLA, L. A confluent calculus of macro expansion and evaluation. *SIGPLAN Lisp Pointers*, v. 5, n. 1, p. 278–287, jan 1992.

BRABRAND, C.; MØLLER, A.; SCHWARTZBACH, M. I. The ⟨bigwig⟩ project. *ACM Transactions on Internet Technology*, v. 2, n. 2, p. 79–114, may 2002.

BRABRAND, C.; SCHWARTZBACH, M. I. Growing languages with metamorphic syntax macros. *SIGPLAN Notices*, v. 37, n. 3, p. 31–40, jan 2002.

BROOKER, R. A.; MORRIS, D. A general translation program for phrase structure languages. *Journal of the ACM*, v. 9, n. 1, p. 1–10, jan 1962.

BROWN, P. J. The ML/I macro processor. *Communications of the ACM*, v. 10, n. 10, p. 618–623, oct 1967.

BUCKLEY, J. et al. Towards a taxonomy of software change: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice – Unanticipated Software Evolution*, v. 17, n. 5, p. 309–332, sep 2005.

BUDD, T. *An Introduction to Object-Oriented Programming*. [S.l.]: Addison-Wesley, 2002.

BURMAKO, E. *Metaprogramming with Macros*. [S.l.], 2012.

BYBEE, J. L. Diachronic linguistics. In: GEERAERTS, D.; CUYCKENS, H. (Ed.). *The Oxford Handbook of Cognitive Linguistics*. 1. ed. [S.l.]: Oxford University, 2010, (Oxford Handbooks).

CAMPBELL, M. S. *Chunking As an Abstraction Mechanism*. Tese (PhD) — Carnegie Mellon University, Pittsburgh, 1987.

CAPRA, F. *The web of life: a new scientific understanding of living systems*. New York: Harper Collins, 1996.

CASTRO JÚNIOR, A. A. *Aspectos de projeto e implementação de linguagens para codificação de programas adaptativos*. Tese (Doutorado) — Escola Politécnica, Universidade de São Paulo, São Paulo, 2009.

CEREDA, P. R. M. Projeto de leiaute semi-automático utilizando dispositivo adaptativo. In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 7, 2013, **Anais...**. São Paulo: [s.n.], 2013. p. 23–30.

CEREDA, P. R. M.; JOSÉ NETO, J. Um arcabouço para extensibilidade em linguagens de programação. In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 9, 2015, **Anais...**. São Paulo: [s.n.], 2015. p. 18–28.

CEREDA, P. R. M.; JOSÉ NETO, J. Utilizando linguagens de programação orientadas a objetos para codificar programas adaptativos. In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 9, 2015, **Anais...**. São Paulo: [s.n.], 2015. p. 2–9.

CEREDA, P. R. M.; JOSÉ NETO, J. AA4J: uma biblioteca para implementação de autômatos adaptativos. In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 10, 2016, **Anais...**. São Paulo: [s.n.], 2016. p. 16–26.

CEREDA, P. R. M.; JOSÉ NETO, J. Instrumenting a context-free language recognizer. In: INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS, 19, 2017, **Proceedings...**. Porto: [s.n.], 2017. v. 2, p. 203–210.

CEREDA, P. R. M.; JOSÉ NETO, J. Towards performance-focused implementations of adaptive devices. *Procedia Computer Science*, v. 109, p. 1164–1169, 2017.

CEREDA, P. R. M.; JOSÉ NETO, J. XML2AA: geração automática de autômatos adaptativos a partir de especificações XML. In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 11, 2017, **Anais...**. São Paulo: [s.n.], 2017. p. 72–81.

CEREDA, P. R. M.; MIURA, N. K.; JOSÉ NETO, J. Syntactic analysis of natural language sentences based on rewriting systems and adaptivity. *Procedia Computer Science*, v. 130, p. 1102–1107, 2018.

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

CHOMSKY, N. *Syntactic structures*. The Hague: Mouton, 1957.

CHURCH, A. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, v. 1, p. 40–41, 1936.

CHURCH, A. A formulation of the simple theory of types. *Journal of Symbolic Logic*, v. 5, p. 56–68, 1940.

COOPER, K.; TORCZON, L. *Construindo Compiladores*. 2. ed. Rio de Janeiro: Elsevier, 2014.

DASGUPTA, S.; PAPADIMITRIOU, C. H.; VAZIRANI, U. *Algorithms*. [S.l.]: McGraw-Hill Education, 2006.

DAUCHET, M. Simulation of Turing machines by a regular rewrite rule. *Theoretical Computer Science*, v. 103, n. 2, p. 409–420, September 1992.

DERSHOWITZ, N. Orderings for term-rewriting systems. *Theoretical Computer Science*, n. 17, p. 279–301, 1982.

DERSHOWITZ, N. Termination of rewriting. *Journal of Symbolic Computation*, v. 3, n. 1–2, p. 69–116, 1987.

DERSHOWITZ, N.; JOUANNAUD, J.-P. Rewrite systems. In: LEEUWEN, J. van (Ed.). *Handbook of Theoretical Computer Science*. [S.l.]: North Holland, 1990. p. 243–320.

DHIMAN, K. et al. Clojure: Modular programming with functional, concurrent language on the JVM. In: CONFERENCE OF THE CENTER FOR ADVANCED STUDIES ON COLLABORATIVE RESEARCH, 2013, **Proceedings...**. Riverton: IBM Corporation, 2013. p. 370–371.

ENDERTON, H. B. *Computability Theory: an introduction to Recursion Theory*. [S.l.]: Academic Press, 2010.

EPSTEIN, R. L.; CARNIELLI, W. *Computability: computable functions, logic, and the foundations of Mathematics*. Belmont: Wadsworth & Brooks, Cole Advanced Books & Software, 1989.

EVJEN, B. et al. *Professional XML*. [S.l.]: Wrox, 2007.

FELLEISEN, M. On the expressive power of programming languages. *Science of Computer Programming*, v. 17, n. 1–3, p. 35–75, dec 1991.

FISHER, A. J. *The generation of parsers for two-level grammars*. Tese (PhD) — Department of Computer Science, University College of Wales, Aberystwyth, 1982.

FLATT, M. *Reference: Racket*. [S.l.], 2010. (PLT Technical Report 1).

FLATT, M.; FINDLER, R. B.; FELLEISEN, M. Scheme with classes, mixins, and traits. In: ASIAN SYMPOSIUM ON PROGRAMMING LANGUAGES AND SYSTEMS, 2006, **Proceedings...**. [S.l.: s.n.], 2006. p. 270–289.

FLORES, I. *The logic of computer arithmetic*. [S.l.]: Prentice Hall, 1963. (International series in Electrical Engineering).

FØLLESDAL, D. *Referential Opacity and Modal Logic*. [S.l.]: Routledge, 2009. (Studies in Philosophy).

FRIEDMAN, H.; SHEARD, M. Elementary descent recursion and proof theory. *Annals of Pure and Applied Logic*, v. 71, n. 1, p. 1–45, January 1995.

GALDINO, A. L. *Uma Formalização da Teoria de Reescrita em Linguagem de Ordem Superior*. Tese (Doutorado) — Departamento de Matemática, Instituto de Ciências Exatas, Universidade de Brasília, Brasília, 2008.

GMEINER, K.; GRAMLICH, B. Transformations of conditional rewrite systems revisited. In: CORRADINI, A.; MONTANARI, U. (Ed.). *Recent Trends in Algebraic Development Techniques*. Heidelberg: Springer-Verlag, 2008, (Lecture Notes in Computer Science, v. 2486).

GOSLING, J. et al. *The Java Language Specification, Java SE 8 Edition*. [S.l.]: Addison-Wesley, 2014.

GRIES, D. *The Science of Programming*. New York: Springer-Verlag, 1981. (Monographs in Computer Science).

HAILPERIN, M.; KAISER, B.; KNIGHT, K. *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. Pacific Grove: PWS Publishing, 1999.

HARMAN, W. *An incomplete guide to the future*. New York: W. W. Norton, 1970.

HERMANN, M.; KIRCHNER, C.; KIRCHNER, H. Implementations of term rewriting systems. *The Computer Journal*, v. 34, n. 1, p. 20–33, feb 1991.

HICKEY, R. The Clojure programming language. In: SYMPOSIUM ON DYNAMIC LANGUAGES, 2008, **Proceedings...**. New York: ACM, 2008.

HIGGINBOTHAM, D. *Clojure for the brave and true: learn the ultimate language and become a better programmer*. San Francisco: No Starch Press, 2015.

HILFINGER, P. N. *Abstraction Mechanisms and Language Design*. Tese (PhD) — Carnegie Mellon University, Pittsburgh, 1981.

HOARE, G. *The Rust Programming Language*. [S.l.], 2010.

HOENIGSWALD, H. M. *Language Change and Linguistic Reconstruction*. [S.l.]: University of Chicago, 1965. (Phoenix Books).

HOPCROFT, J. E. *An $n \log n$ algorithm for minimizing states in a finite automaton*. [S.l.], 1970.

HOPCROFT, J. E. An $n \log n$ algorithm for minimizing states in a finite automaton. In: INTERNATIONAL SYMPOSIUM ON THE THEORY OF MACHINES AND COMPUTATIONS, 1971, **Proceedings...**. Haifa, Israel: Academic Press, 1971. p. 189–196.

HOPCROFT, J. E.; ULLMAN, J. D.; MOTWANI, R. *Introdução à teoria de autômatos, linguagens e computação*. [S.l.]: Editora Campus, 2003.

HSIANG, J. et al. The term rewriting approach to automated theorem proving. *The Journal of Logic Programming*, v. 14, n. 1–2, p. 71–99, oct 1992.

HUET, G. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, v. 27, n. 4, p. 797–821, oct 1980.

HUET, G.; OPPEN, D. C. *Equations and Rewrite Rules: A Survey*. Stanford, 1980.

HUTTON, G. An introduction to hol, a theorem-proving environment for higher-order logic. *Journal of Functional Programming*, v. 4, n. 4, p. 557–559, oct 1994.

IERUSALIMSCHY, R. *Programming in Lua*. 4. ed. [S.l.]: LUA.ORG, 2016.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H.; CELES FILHO, W. Lua: an extensible extension language. *Software: Practice and Experience*, v. 26, n. 6, p. 635–652, jun 1996.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H.; CELES FILHO, W. The evolution of Lua. In: ACM SIGPLAN CONFERENCE ON HISTORY OF PROGRAMMING LANGUAGES, 3, 2007, **Proceedings...**. New York: ACM, 2007. (HOPL III), p. 1–26.

IWAI, M. K. *Um formalismo gramatical adaptativo para linguagens de contexto*. Tese (Doutorado) — Escola Politécnica, Universidade de São Paulo, São Paulo, 2000.

JANTZEN, M. Confluent string rewriting and congruences. In: *EATCS (European Association for Theoretical Computer Science) Monographs on Theoretical Computer Science*. Berlin: Springer-Verlag, 1988. v. 14.

JOSÉ NETO, J. *Introdução à compilação*. [S.l.]: LTC, 1987. (Engenharia de Computação).

JOSÉ NETO, J. *Contribuições à metodologia de construção de compiladores*. Tese (Livre docência) — Escola Politécnica, Universidade de São Paulo, São Paulo, 1993.

JOSÉ NETO, J. Adaptive automata for context-sensitive languages. *SIGPLAN Notices*, v. 29, n. 9, p. 115–124, set 1994.

JOSÉ NETO, J. Adaptive rule-driven devices: general formulation and case study. In: INTERNATIONAL CONFERENCE ON IMPLEMENTATION AND APPLICATION OF AUTOMATA, 2001, **Proceedings...**. Pretoria: [s.n.], 2001.

JOSÉ NETO, J. Um levantamento da evolução da adaptatividade e da tecnologia adaptativa. *IEEE Latin America Transactions*, v. 5, p. 496–505, 2007.

JOSÉ NETO, J.; MAGALHÃES, M. E. S. Reconhecedores sintáticos: Uma alternativa didática para uso em cursos de engenharia. In: CONGRESSO NACIONAL DE INFORMÁTICA, 14, 1981, **Anais...**. [S.l.: s.n.], 1981. p. 171–181.

JOSÉ NETO, J.; ROCHA, R. L. d. A. Autômato adaptativo, limites e complexidade em comparação com a máquina de Turing. In: CONGRESS OF LOGIC APPLIED TO TECHNOLOGY, 2, 2000, **Proceedings...**. [S.l.: s.n.], 2000. p. 33–48.

JOUANNAUD, J.-P.; LESCANNE, P.; REINIG, F. Recursive decomposition ordering. In: BJØRNER, D. (Ed.). *Formal Description of Programming Concepts 2*. Garmisch-Partenkirchen, Germany: North Holland, 1982. p. 331–348.

JUNG, K.; BROWN, A. *Beginning Lua Programming*. [S.l.]: Wrox, 2007.

KELLER, R. M. *Computer Science: Abstraction to Implementation*. [S.l.]: Harvey Mudd College, 2001.

KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language*. 2. ed. [S.l.]: Prentice Hall, 1988.

KESSLER, C. M.; ANDERSON, J. R. Learning flow of control: recursive and iterative procedures. *Human-Computer Interaction*, v. 2, n. 2, p. 135–166, jun 1986.

KIKUCHI, Y.; KATAYAMA, T. An application of term rewriting systems for functional programming. In: OHNO, Y.; MATSUDA, T. (Ed.). *Distributed Environments*. Tokyo: Springer-Verlag, 1991. p. 79–106.

KLOP, J. W. *Term Rewriting Systems*. [S.l.], 1992.

KNOBLOCK, C. Learning abstraction hierarchies for problem solving. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 8, 1990, **Proceedings...**. [S.l.: s.n.], 1990. p. 923–928.

KNOBLOCK, C. Search reduction in hierarchical problem solving. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 9, 1991, **Proceedings...**. [S.l.: s.n.], 1991. p. 686–691.

KNOBLOCK, C. *Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning*. [S.l.]: Springer US, 1993.

KNOBLOCK, C. A. Learning hierarchies of abstraction spaces. In: INTERNATIONAL WORKSHOP ON MACHINE LEARNING, 6, 1989, **Proceedings...**. Ithaca: [s.n.], 1989. p. 241–245.

KNUTH, D. E. *The TEXbook*. Massachusetts: Addison-Wesley, 1986.

KNUTH, D. E.; BENDIX, P. B. Simple word problems in universal algebras. In: SIEKMANN, J. H.; WRIGHTSON, G. (Ed.). *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. [S.l.]: Springer-Verlag, 1983. p. 342–376.

KOHLBECKER, E. *Syntactic Extensions in the Programming Language LISP*. Tese (PhD) — Indiana University, Bloomington, 1986.

KOHLBECKER, E. et al. Hygienic macro expansion. In: LISP AND FUNCTIONAL PROGRAMMING, 1986, **Proceedings...**. [S.l.: s.n.], 1986. p. 151–161.

KOHLBECKER, E. E.; WAND, M. Macro-by-example: Deriving syntactic transformations from their specifications. In: ACM SIGACT-SIGPLAN SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 14, 1987, **Proceedings...**. New York: [s.n.], 1987. p. 77–84.

KORF, R. E. Planning as search: A quantitative approach. *Artificial Intelligence*, v. 33, n. 1, p. 65–68, sep 1987.

KRAMER, J. Is abstraction the key to computing? *Communications of the ACM*, v. 50, n. 4, p. 36–42, apr 2007.

KUHN, T. S. *The structure of scientific revolutions*. Chicago: The University of Chicago, 1970.

LANDIN, P. J. The mechanical evaluation of expressions. *The Computer Journal*, v. 6, n. 4, p. 308–320, 1964.

LEAVENWORTH, B. M. Syntax macros and extended translation. *Communications of the ACM*, v. 9, n. 11, p. 790–793, nov 1966.

LESCANNE, P. Uniform termination of term rewriting systems. In: COURCELLE, B. (Ed.). COLLOQUE LES ARBRES EN ALGEBRE ET EN PROGRAMMATION, 9, 1894, **Proceedings...**. Bordeaux: Cambridge University, 1984. p. 182–194.

LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elementos de teoria da computação*. 2. ed. Porto Alegre: Bookman, 2000.

LIVIO, M. *The golden ratio: the story of ϕ, the world's most astonishing mumber*. [S.l.]: Broadway Books, 2003.

LOPER, E.; BIRD, S. NLTK: the natural language toolkit. In: ACL-02 WORKSHOP ON EFFECTIVE TOOLS AND METHODOLOGIES FOR TEACHING NATURAL LANGUAGE PROCESSING AND COMPUTATIONAL LINGUISTICS, 2002, **Proceedings...**. Stroudsburg: Association for Computational Linguistics, 2002. v. 1, p. 63–70.

MACFARLANE, J. *The CommonMark specification, version 0.28*. 2017.

MACKENZIE, C. E. *Coded Character Sets, History and Development*. [S.l.]: Addison-Wesley, 1980.

MARCHIORI, M. Logic programs as term rewriting systems. In: LEVI, G.; RODRÍGUEZ-ARTALEJO, M. (Ed.). INTERNATIONAL CONFERENCE ON ALGEBRAIC AND LOGIC PROGRAMMING, 1994, **Proceedings...**. [S.l.]: Springer-Verlag, 1994. (Lecture Notes in Computer Science, 850).

MARYNIARCZYK, A. *Powszechna encyklopedia filozofii*. [S.l.], 2000.

MCILROY, M. D. Macro instruction extensions of compiler languages. *Communications of the ACM*, v. 3, n. 4, p. 214–220, apr 1960.

MINSKY, M. Steps toward artificial intelligence. In: FEIGENBAUM, E. A. (Ed.). *Computers and Thought*. New York, NY: McGraw-Hill, 1963. p. 406–450.

MOOERS, C. N. TRAC, a procedure-describing language for the reactive typewriter. *Communications of the ACM*, v. 9, n. 3, p. 215–219, mar 1966.

MOOERS, C. N.; DEUTSCH, L. P. TRAC, a text handling language. In: ACM NATIONAL CONFERENCE, 20, 1965, **Proceedings...**. [S.l.: s.n.], 1965. p. 229–246.

MORAES, M. *Alguns aspectos de tratamento de dependências de contexto em linguagem natural empregando tecnologia adaptativa*. Tese (Doutorado) — Escola Politécnica, Universidade de são Paulo, São Paulo, 2006.

NEIGHBORS, J. M. *Software Construction Using Components*. Tese (PhD) — University of California, Irvine, 1980.

NEWELL, A.; SIMON, H. A. The processes of creative thinking. In: *Contemporary Approach to Creative Thinking*. [S.l.]: Atherton Press, 1962. p. 63–119.

OLIVÉ, A. *Conceptual Modeling of Information Systems*. [S.l.]: Springer-Verlag, 2007.

ORLAREY, Y. et al. Lambda calculus and music calculi. In: INTERNATIONAL COMPUTER MUSIC CONFERENCE, 1994, **Proceedings...**. San Francisco: [s.n.], 1994.

PARNAS, D. L. *Information distribution aspects of design methodology*. Pittsburgh, 1971.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, v. 15, n. 12, p. 1053–1058, dec 1972.

PARNAS, D. L. Designing software for ease of extension and contraction. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 3, 1978, **Proceedings...**. Piscataway: IEEE, 1978. p. 264–277.

PARNAS, D. L.; CLEMENTS, P. C.; WEISS, D. M. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11, n. 3, p. 259–266, mar 1985.

PIAGET, J. *The Psychology Of The Child*. 2. ed. [S.l.]: Basic Books, 1969.

PISTORI, H. *Tecnologia adaptativa em engenharia de computação: estado da arte e aplicações*. Tese (Doutorado) — Escola Politécnica, Universidade de São Paulo, São Paulo, 2003.

PLAISTED, D. A. Equational reasoning and term rewriting systems. In: GABBAY, D. M.; HOGGER, C. J.; ROBINSON, J. A. (Ed.). *Handbook of Logic in Artificial Intelligence and Logic Programming*. New York: Oxford University, 1993. v. 1, p. 274–364.

PÓLYA, G. *How to Solve It*. [S.l.]: Princeton University Press, 1945.

QUINE, W. V. O.; CHURCHLAND, P. S.; FØLLESDAL, D. *Word and Object*. [S.l.]: MIT, 2013.

RAMOS, M. V. M.; JOSÉ NETO, J.; VEGA, Í. S. *Linguagens formais: teoria, modelagem e implementação*. Porto Alegre: Bookman, 2009.

ROBERTS, P. Abstract thinking: A predictor of modeling ability? In: ACM/IEEE INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS: EDUCATORS' SYMPOSIUM, 12, 2009, **Proceedings...**. Denver: [s.n.], 2009.

ROBIČ, B. *The foundations of Computability Theory*. [S.l.]: Springer-Verlag, 2015.

RUSSELL, B. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, v. 30, n. 3, p. 222–262, 1908.

SCHUMAN, S. A.; JORRAND, P. Definition mechanisms in extensible programming languages. In: FALL JOINT COMPUTER CONFERENCE, 1970, **Proceedings...**. New York: [s.n.], 1970. p. 9–20.

SEBESTA, R. W. *Concepts of Programming Languages*. 10. ed. [S.l.]: Pearson, 2013.

SHAFRANOVICH, Y. *Common format and MIME type for comma-separated values (CSV) files*. [S.l.], 2005. RFC4180.

SHALIT, A. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. [S.l.]: Addison-Wesley, 1996.

SILVA FILHO, R. I. *Uma nova formulação algébrica para o autômato finito adaptativo de segunda ordem aplicada a um modelo de inferência indutiva*. Tese (Doutorado) — Escola Politécnica, Universidade de São Paulo, São Paulo, 2011.

SILVA, S. R. B. *Software adaptativo, método de projeto, representação gráfica e implementação de linguagem de programação*. Dissertação (Mestrado) — Escola Politécnica, Universidade de São Paulo, 2011.

SIPSER, M. *Introduction to the Theory of Computation*. 3. ed. [S.l.]: Cengage Learning, 2012.

SKALSKI, K. *Syntax-extending and type-reflecting macros in an object-oriented language*. Dissertação (MSc) — University of Wrocław, 2005.

SKALSKI, K.; MOSKAL, M.; OLSZTA, P. Metaprogramming in Nemerle. In: GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, 2004, **Proceedings...**. [S.l.: s.n.], 2004.

SØNDERGAARD, H.; SESTOFT, P. Referential transparency, definiteness and unfoldability. *Acta Informatica*, v. 27, n. 6, p. 505–517, may 1990.

STALLMAN, R. M. *Using the GNU Compiler Collection*. Boston, MA, 2017. GNU Press.

STALLMAN, R. M.; WEINBERG, Z. *The C Preprocessor*. [S.l.], 2005.

STANDISH, T. A. Extensibility in programming language design. *SIGPLAN Notices*, v. 10, n. 7, p. 18–21, jul 1975.

STANGE, R. L.; CEREDA, P. R. M.; JOSÉ NETO, J. Agentes adaptativos reativos: formalização e estudo de caso. In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 11, 2017, **Anais...**. São Paulo: [s.n.], 2017. p. 63–71.

STAPLES, J. Church-Rosser theorem for replacement systems. In: CROSLEY, J. (Ed.). *Algebra and Logic*. [S.l.]: Springer-Verlag, 1975, (Lecture Notes in Mathematics, 450).

STEELE, G. *Common LISP, The Language*. [S.l.]: Digital Press, 1990.

STRACHEY, C. A general purpose macrogenerator. *The Computer Journal*, v. 8, n. 3, p. 225–241, oct 1965.

STROUSTRUP, B. *The C++ Programming Language*. 4. ed. [S.l.]: Addison-Wesley, 2013.

TANAKA-ISHII, K. *Semiotics of Programming*. 1. ed. New York, NY, USA: Cambridge University, 2010.

TANENBAUM, A. S.; BOS, H. *Modern Operating Systems*. 4. ed. [S.l.]: Prentice Hall, 2014.

TRIANCE, J. M.; LAYZELL, P. J. Macro processors for enhancing high-level languages: Some design principles. *The Computer Journal*, v. 28, n. 1, p. 34–43, 1985.

TURBAK, F.; GIFFORD, D. *Design Concepts in Programming Languages*. [S.l.]: MIT, 2008.

TURNER, K. J. *Exploiting the M4 Macro Language*. [S.l.], 1994. Technical Report CSM-126.

VIERA, M.; SWIERSTRA, S. D. *Semantic macros: attribute grammar combinators*. Utrecht, The Netherlands, 2011. Technical Report UU-CS-2011-028.

VIERA, M.; SWIERSTRA, S. D. Attribute grammar macros. *Science of Computer Programming*, v. 96, n. P2, p. 211–229, dec 2014.

WAITE, W. M. The mobile programming system: STAGE2. *Communications of the ACM*, v. 13, n. 7, p. 415–421, jul 1970.

WEISE, D.; CREW, R. Programmable syntax macros. *SIGPLAN Notices*, v. 28, n. 6, p. 156–165, jun 1993.

WIRTH, N. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, v. 10, n. 11, p. 822–823, nov 1977.

WOODSEND, K.; LAPATA, M. Text rewriting improves semantic role labeling. *Journal of Artificial Intelligence Research*, n. 51, p. 133–164, sep 2014.

ZEMANEK, H. Semiotics and programming languages. *Communications of the ACM*, v. 9, n. 3, p. 139–143, mar 1966.

ZENG, L. M.; QIN, J. *Metadata*. 2. ed. [S.l.]: ALA Neal-Schuman, 2016.

ZENGER, M. *Programming Language Abstractions for Extensible Software Components*. Tese (PhD) — École Polytechnique Fédérale de Lausanne, 2004.

ZHOU, C.; HANSEN, M. *Duration Calculus: a formal approach to real-time systems*. 1. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 2004. (Monographs in Theoretical Computer Science).

ZINGARO, D. *Modern Extensible Languages*. [S.l.], 2007.

APÊNDICE A

EXEMPLOS DE IMPLEMENTAÇÕES DE EXPANSORES DE MACROS

> Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a gradual development can be nicely demonstrated.
>
> NIKLAUS WIRTH

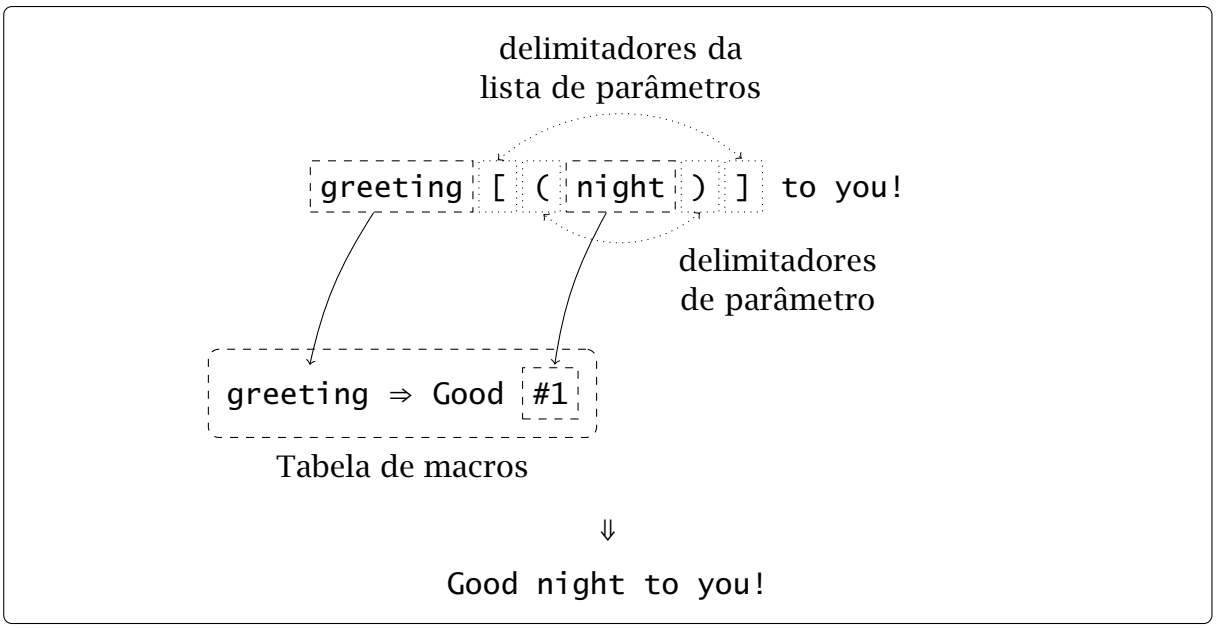
Este apêndice apresenta exemplos de expansores de macros com padrões sintáticos livres e dependentes de contextos (tipos 2 e 1 na hierarquia de Chomsky, respectivamente), com suporte a parâmetros (capturas de estruturas sintáticas) e transformações algorítmicas, e suas implementações correspondentes. Adicionalmente, funcionalidades complementares são discutidas, incluindo desafios de implementação das primitivas de adaptatividade.

A.1 EXPANSÃO DE MACROS COM PARÂMETROS

Considere uma extensão do expansor de macros originalmente apresentado na Seção 3.4, admitindo a presença opcional de parâmetros em instâncias de macros (com padrão sintático livre de contexto). A Figura A.1 apresenta um exemplo de expansão de macros com parâmetros em um texto, no qual uma instância de macro é transformada em uma sequência de símbolos correspondente de acordo com as regras de substituição definidas em uma tabela. Durante a expansão, o valor associado ao parâmetro da instância de macro é devidamente substituído na sequência resultante.

Conforme ilustra a Figura A.1, a ocorrência de `greeting[(night)]` é substituída por `Good night` na cadeia de símbolos resultante. Observe que a sequência associada a `greeting` na tabela de macros possui um *placeholder* (a saber, `#1`) que é substituído pelo valor (devidamente expandido) do parâmetro na instância da macro (a saber, `night`). A sintaxe adotada para *placeholders* é fortemente inspirada na linguagem TEX (KNUTH, 1986), na qual o símbolo # denota o valor de um parâmetro determinado por um número natural positivo associado. Tal número refere-se ao índice do parâmetro na lista de parâmetros da macro; assim, `#1` denota o valor do primeiro parâmetro em uma determinada instância de macro. Nesta seção, para

Figura A.1: Exemplo de expansão de macros em um texto, de acordo com a especificação proposta do expansor de macros com parâmetros.



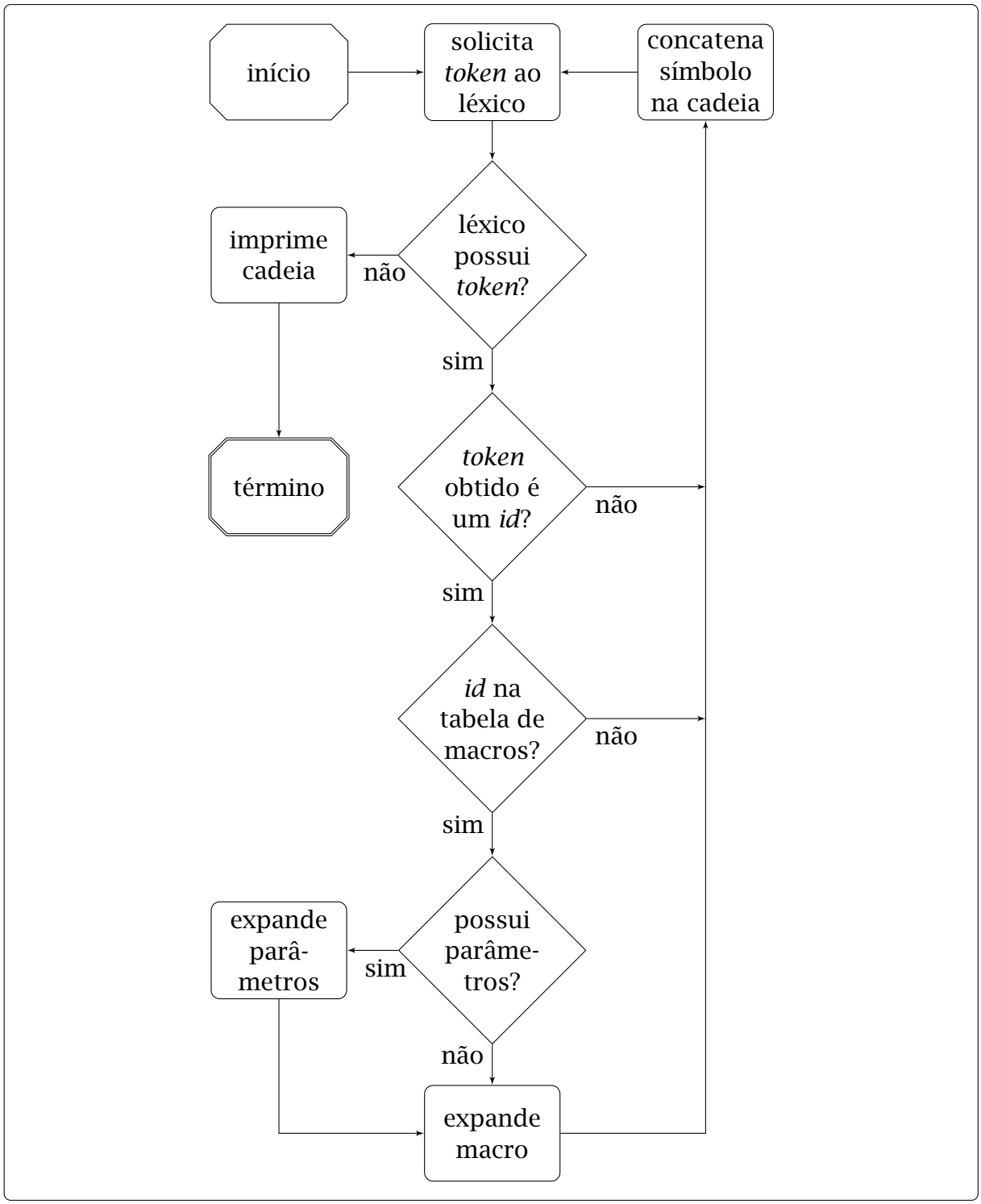
Fonte: autor.

fins didáticos, *placeholders* serão tratados como substituições implícitas; a Subseção A.3.2 apresenta uma estratégia alternativa de resolução de parâmetros como macros de escopo local com padrão sintático regular.

A Figura A.2 apresenta o fluxograma contendo as etapas da expansão de macros com parâmetros em um texto. Observe que as etapas são semelhantes àquelas descritas no fluxograma apresentado na Figura 3.15, com exceção de uma verificação adicional acerca da existência de parâmetros na instância da macro e posterior tratamento destes, caso existam. Tal característica permite a reutilização dos três motores de eventos ME_1, ME_2 e ME_3 (Figura 3.24) apresentados na Seção 3.4 para composição do novo expansor com suporte a macros com parâmetros.

De acordo com a Figura A.2, *tokens* classificados como identificadores são consultados na tabela de macros. Caso o identificador exista na tabela de macros, este é verificado acerca da existência de parâmetros. Em caso positivo, *placeholders* existentes são substituídos pelos valores expandidos dos parâmetros correspondentes na sequência de transformação. Finalmente, o identificador é expandido para a sequência de transformação correspondente e o resultado é concatenado na cadeia de saída. Eventualmente, a sequência de transformação pode conter outras macros. Nas situações em que o *token* não seja um identificador ou se este não estiver presente na tabela de macros, o seu valor é simplesmente concatenado na cadeia de saída.

Figura A.2: Fluxograma das etapas de expansão de macros com parâmetros em um texto.



Fonte: autor.

A.1.1 IDENTIFICAÇÃO DE PARÂMETROS

A especificação do expansor de macros proposto, conforme ilustra o exemplo da Figura A.1, utiliza símbolos especiais que determinam a regra sintática da estrutura de delimitação para demarcação e extração da lista de parâmetros. Tais combinações de símbolos são definidas a seguir:

i) Colchetes representam a estrutura sintática de delimitação da lista de parâmetros de uma instância de macro. A delimitação é denotada pelos símbolos `[` e `]` (abertura e fechamento de colchetes, respectivamente).

ii) Parênteses representam a estrutura sintática de delimitação de um parâmetro em uma lista. A delimitação é denotada pelos símbolos `(` e `)` (abertura e fechamento de parênteses, respectivamente). Uma lista de parâmetros pode conter múltiplos elementos.

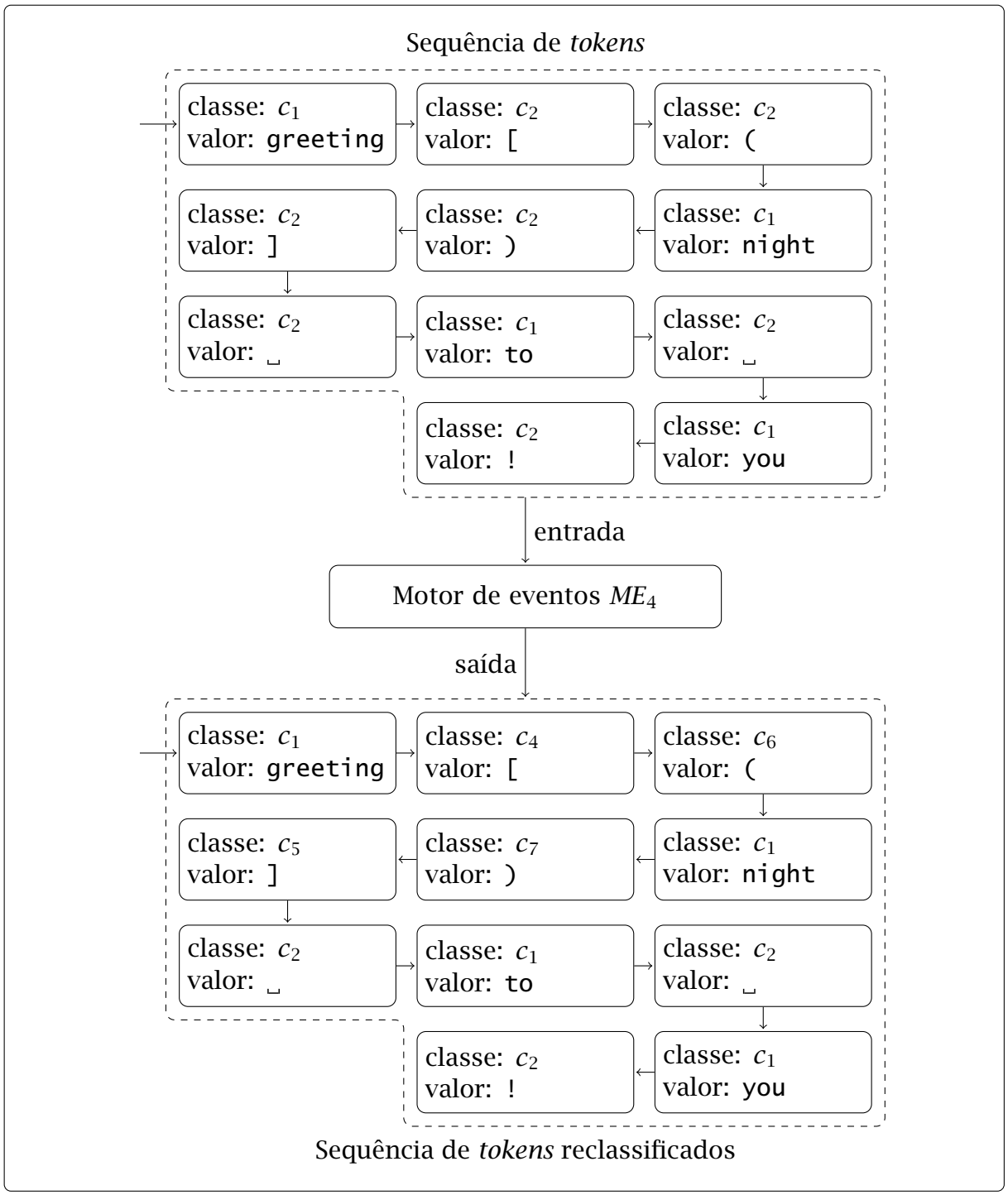
O analisador léxico apresentado na Subseção 3.4.1 não está projetado para distinguir tais delimitadores dos demais símbolos classificados como c_2. Sejam c_4, c_5, c_6 e c_7 classes gramaticais adicionais que denotam abertura e fechamento de colchetes e parênteses, respectivamente. A função χ_2, definida na Equação A.1, verifica se um *token* previamente categorizado como c_2 está elegível para reclassificação (*tokens* categorizados como c_1 permanecem com sua classe gramatical inalterada).

$$\chi_2(\sigma) = \begin{cases} c_1 & \text{se } p_1(\sigma, c_1) \\ c_2 & \text{se } p_1(\sigma, c_2) \wedge \mathit{value}(\sigma) \notin \{\texttt{[}, \texttt{]}, \texttt{(}, \texttt{)}\} \\ c_4 & \text{se } p_1(\sigma, c_2) \wedge \mathit{value}(\sigma) = \texttt{[} \\ c_5 & \text{se } p_1(\sigma, c_2) \wedge \mathit{value}(\sigma) = \texttt{]} \\ c_6 & \text{se } p_1(\sigma, c_2) \wedge \mathit{value}(\sigma) = \texttt{(} \\ c_7 & \text{se } p_1(\sigma, c_2) \wedge \mathit{value}(\sigma) = \texttt{)} \end{cases} \tag{A.1}$$

Conforme ilustra a Equação A.1, *tokens* que representam estruturas de delimitação de parâmetros são reclassificados para suas classes gramaticais correspondentes. A Figura A.3 apresenta um motor de eventos ME_4 como modelo de implementação da função de reclassificação χ_2 aplicada em uma sequência de *tokens* obtidos na fase de análise léxica.

A identificação de parâmetros de uma macro contempla as ações semânticas a_3, a_4, a_5 e a_6, apresentadas na Tabela A.1, potencialmente aplicáveis ao *token* corrente t. Caso a regra sintática se aplique, os parâmetros identificados são devidamente extraídos e associados a t como um conjunto de pares ordenados (k, v), tal que $k \in \mathbb{N}$ e $v \in \Sigma_1^*$ representam o índice do parâmetro na lista e o valor associado, respectivamente.

Figura A.3: Motor de eventos ME_4 como modelo de implementação da função de reclassificação χ_2 (Equação A.1).



Fonte: autor.

Tabela A.1: Ações semânticas disponibilizadas na identificação de parâmetros do expansor de macros da Seção A.1.

Ação	Significado
a_3	Memoriza o *token* corrente
a_4	Inicializa as variáveis para captura
a_5	Início da captura do bloco corrente
a_6	Término da captura do bloco corrente

Fonte: autor.

Sejam $\cdot$ o operador de concatenação, p uma cadeia, $c \in \mathbb{N}$ uma variável denotando o contador de parâmetros, e t e r os *tokens* corrente e memorizado, respectivamente, com $pars(t)$ representando o conjunto de parâmetros associado a t, inicialmente vazio, tal que $pars(t) = \varnothing$. As ações semânticas a_3, a_4, a_5 e a_6 são definidas a seguir, nas Equações A.2, A.3, A.4 e A.5.

$$a_3 \equiv r \leftarrow t \quad \text{e} \quad c \leftarrow 0 \tag{A.2}$$

$$a_4 \equiv p \leftarrow \epsilon \quad \text{e} \quad c \leftarrow c + 1 \tag{A.3}$$

$$a_5 \equiv p \leftarrow p \cdot value(t) \tag{A.4}$$

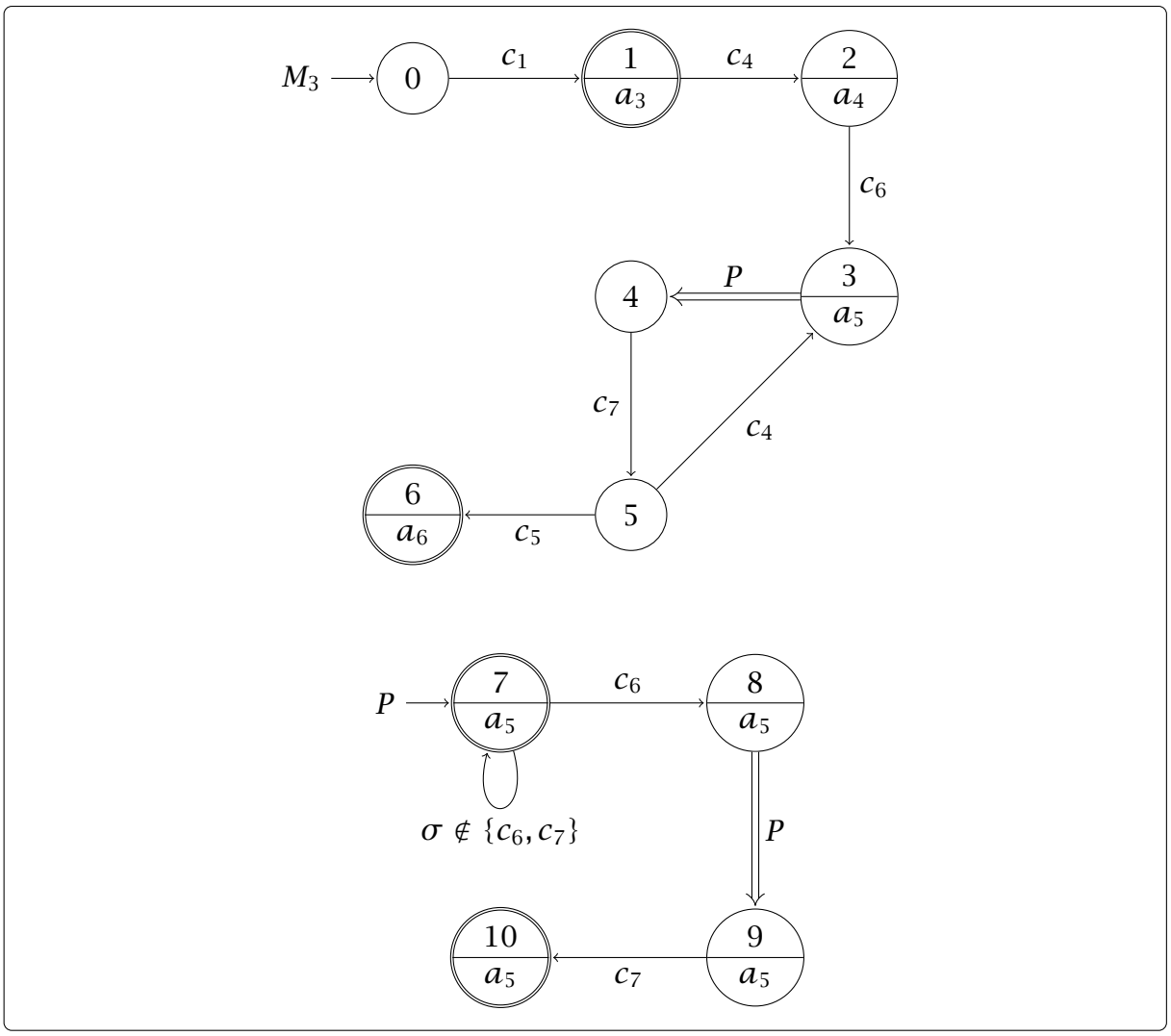
$$a_6 \equiv pars(t) \leftarrow pars(t) \cup \{(c, p)\} \tag{A.5}$$

Dada uma sequência de *tokens* devidamente reclassificados através da função de reclassificação χ_2 (Equação A.1), a Figura A.4 apresenta um autômato de pilha estruturado M_3 que descreve a identificação de parâmetros proposta. O símbolo $\sigma \in \Sigma_3$ representa o *token* corrente, devidamente reclassificado pela aplicação de χ_2, $\Sigma_3 = \{c_1, c_2, c_4, c_5, c_6, c_7\}$, e os estados assinalados com a_3, a_4, a_5 e a_6 indicam as ações semânticas associadas (Tabela A.1).

Sejam $w \in \Sigma_3^*$ uma cadeia, tal que $w \in L(M_3)$, e $A\colon Q_3 \mapsto \{a_3, a_4, a_5, a_6\}$ uma função que mapeia estados de M_3 em ações semânticas a_3, a_4, a_5 e a_6. A ação semântica a_i será disparada se, e somente se, o autômato de pilha estruturado M_3, durante o reconhecimento de w, referenciar j como estado corrente após o consumo de um símbolo $w_k \in w$, $a_i \iff (\gamma, \alpha, w_k) \vdash (\gamma, j, \epsilon)$, com $(j, a_i) \in A$. A função A é definida por extensão como $A = \{(1, a_3), (2, a_4), (3, a_5), (6, a_6), (7, a_5), (8, a_5), (9, a_5), (10, a_5)\}$.

A Figura A.5 apresenta um motor de eventos ME_5 como modelo de implementação da identificação de parâmetros. No exemplo, a sequência de 11 *tokens* devidamente reclassificados através da aplicação da função χ_2 (Equação A.1) é submetida ao motor de eventos, disparando ações correspondentes à análise sintática de identificação de parâmetros (por exemplo, através do autômato de pilha estruturado M_3 da Figura A.4); como resultado, uma sequência de 6 *tokens* é gerada como saída deste motor de eventos. Observe que os parâmetros da macro `greeting` (Figura A.1)

Figura A.4: Autômato de pilha estruturado M_3 que descreve a identificação de parâmetros para o expansor de macros da Seção A.1.



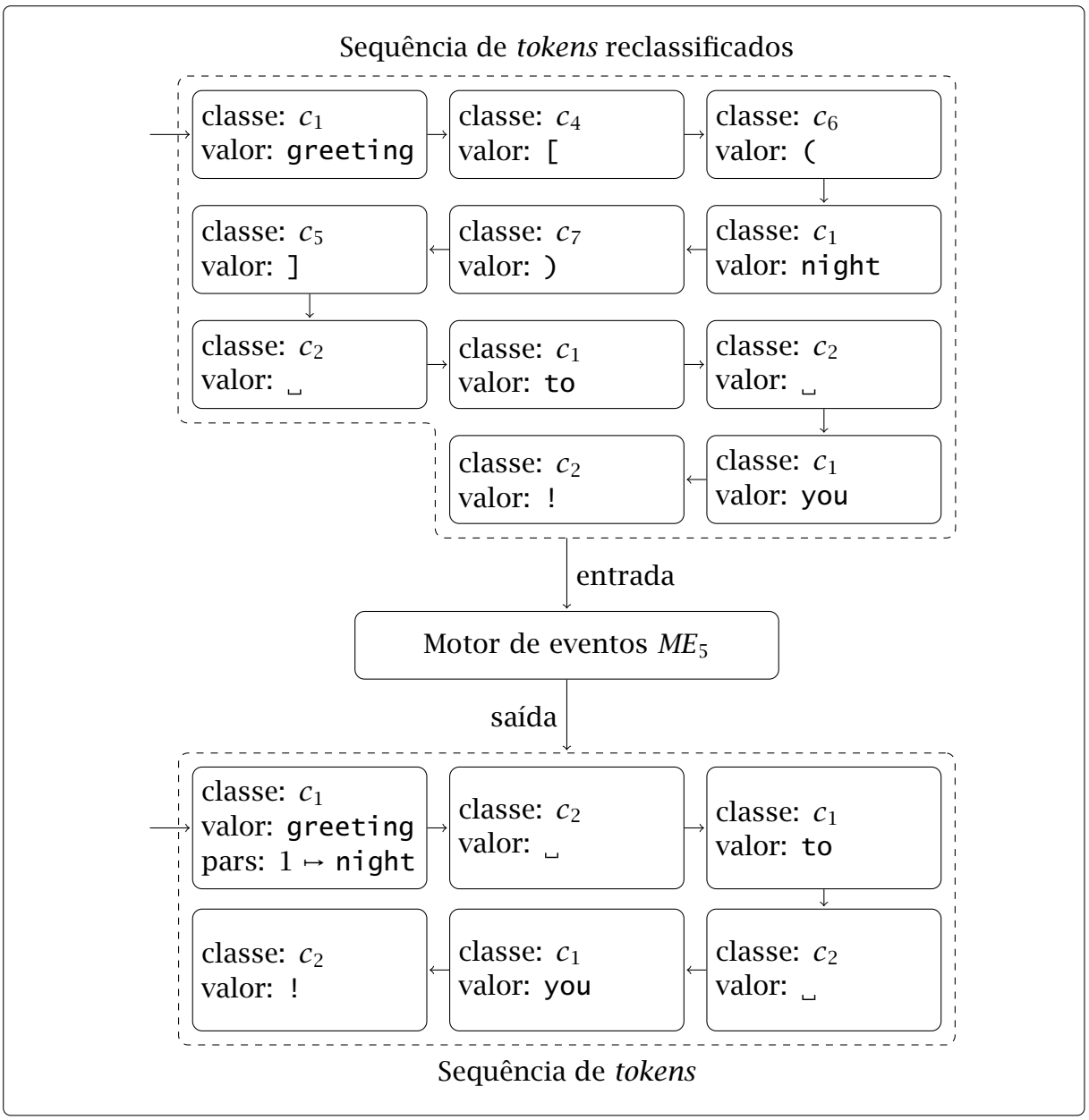
Fonte: autor.

foram devidamente extraídos e associados ao *token* correspondente.

Considere uma implementação da identificação de parâmetros, detalhada a seguir. A Figura A.6 apresenta duas funções auxiliares, `slice` e `extract`, para extração de parâmetros em uma lista, de acordo com a estrutura de delimitação definida na especificação do expansor de macros (a saber, abertura e fechamento de parênteses). Tais funções correspondem semanticamente aos estados 3, 4 e 5 do autômato de pilha estruturado M_3 e à submáquina P (Figura A.4).

De acordo com a Figura A.6, a função `slice` remove o aninhamento sintático da cadeia de símbolos fornecida como parâmetro (a saber, `s`) e a retorna, enquanto a função `extract` identifica e extrai o padrão sintático referente a cada parâmetro da lista e a adiciona em uma tabela (a saber, `parameters`), retornando-a ao término da identificação. A Figura A.7 apresenta uma versão modificada da função `take` (Figura 3.19), estendida para incluir a identificação e captura de potenciais parâmetros

Figura A.5: Motor de eventos M_5 como modelo de implementação da identificação de parâmetros.



Fonte: autor.

Figura A.6: Funções auxiliares `slice` e `extract` para extração de parâmetros em uma lista, de acordo com a estrutura de delimitação definida na especificação do expansor de macros da Seção A.1.

```
local function slice(s)
  return string.sub(s, 2, -2)
end

local function extract(s)
  s = slice(s)
  local parameters = {}
  while #s ~= 0 do
    local a, b, c = string.find(s, "^(%b())")
    if a then
      table.insert(parameters, slice(c))
      s = tail(s, b + 1)
    else
      return {}
    end
  end
  return parameters
end
```

Fonte: autor.

de macros. É importante destacar que a linguagem Lua disponibiliza nativamente o padrão `%b` para identificar aninhamento sintático; tal padrão é seguido de dois símbolos, distintos entre si, que determinam a delimitação a ser identificada e eventualmente obtida (IERUSALIMSCHY, 2016; JUNG; BROWN, 2007). No caso do expansor de macros proposto na Seção A.1, as estruturas de delimitação da lista de parâmetros e dos elementos desta lista são representadas pelos padrões `%b[]` e `%b()`, respectivamente.

A função `take` modificada, conforme ilustra a Figura A.7, implementa o analisador léxico proposto na Subseção 3.4.1 e inclui a identificação e extração de potenciais parâmetros de macro, de acordo com as estruturas de delimitação definidas para o expansor de macros.

Exemplo 21 (execução da função `take` modificada da Figura A.7)**.** A Figura A.8 apresenta um exemplo da execução da função `take` modificada com o texto da Figura A.1 (representado pela variável `s`), de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Observe que a sequência de *tokens* obtida é a mesma retornada pelo motor de eventos da Figura A.5, como esperado. A função auxiliar `pp` produz uma representação textual de tabelas na linguagem Lua. □

Figura A.7: Versão modificada da função `take` (Figura 3.19), estendida para incluir a identificação e extração de potenciais parâmetros de macros.

```
local function take(s)
  local a, b, c, d = string.find(s, "^(%a%w*)(%b[])")
  if a then
    return token({ value = c, parameters = extract(d) }, 'id'),
       tail(s, b + 1)
  else
    a, b, c = string.find(s, "^(%a%w*)")
    if a then
      return token({ value = c }, 'id'), tail(s, b + 1)
    else
      return token({ value = head(s) }, 'other'), tail(s, 2)
    end
  end
end
```

Fonte: autor.

A identificação e captura de potenciais parâmetros permite que *tokens* classificados como c_1, elegíveis como instâncias de macros, sejam tratados adequadamente pelo expansor em uma fase seguinte, substituindo eventuais ocorrências de *placeholders* por seus respectivos valores associados. O processamento da sequência de *tokens* obtida após a identificação de parâmetros é discutido em detalhes na Subseção A.1.2.

A.1.2 PROCESSAMENTO DE *TOKENS*

O processamento de *tokens* para o expansor da linguagem de macros com parâmetros é praticamente idêntico ao descrito na Subseção 3.4.2, com a ressalva de incluir suporte à reescrita de parâmetros nas sequências de substituição. Para tal, propõe-se uma nova versão da função de reescrita de termos μ (Equação 3.3), apresentada a seguir, na Equação A.6, dado que a função existente não contempla reescritas locais (atribuição de parâmetros). É importante destacar que as ações semânticas a_1 e a_2 (Tabela 3.5), a função de reclassificação χ (Equação 3.8), o autômato finito determinístico M_2 (Figura 3.22) e os motores de eventos ME_2 e ME_3 (Figuras 3.21 e 3.23) permanecem inalterados.

$$\begin{aligned} \mu(x) = y \mid & (value(x), \alpha) \in TM \\ & \wedge\ value(x) \rightarrow_{TM} \alpha \rightarrow^*_{pars(x)} \alpha' \rightarrow^*_{TM} y \end{aligned} \tag{A.6}$$

De acordo com a Equação A.6, a função de reescrita de termos μ foi estendida para incluir potenciais substituições de parâmetros por seus valores correspondentes

Figura A.8: Execução da função modificada `take` da Figura A.7 com o texto da Figura A.1, de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Espaços em branco (exceto os que foram utilizados como separadores de *tokens* na impressão) estão deliberadamente marcados como visíveis.

Código-fonte Lua:

```
local function pp(t)
  if type(t) ~= "table" then return tostring(t) end
  local f = t[1] and ipairs or pairs
  local result, comma = "{", ""
  for k, v in f(t) do
    result = result .. comma .. tostring(k) .. " --> " .. pp(v)
    comma = ", "
  end
  return result .. "}"
end

local s, a = "greeting[(night)] to you!"

while #s ~= 0 do
  a, s = take(s)
  io.write("(" .. pp(a.value) .. ", " .. a.class .. ") ")
end
```

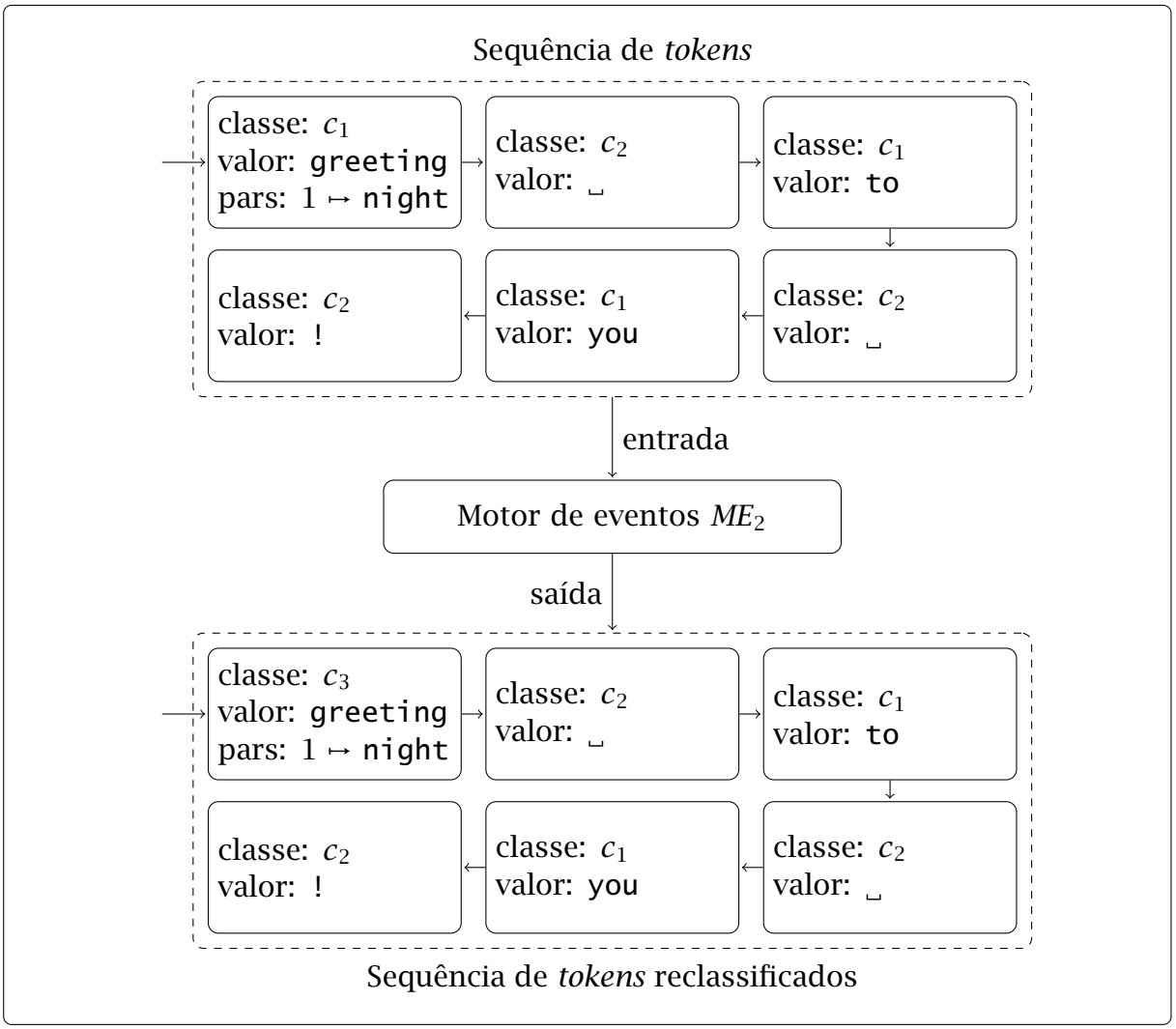
Resultado da execução:

```
$ lua ex3.lua

({value --> greeting, parameters --> {1 --> night}}, id)
   ({value --> ␣}, other) ({value --> to}, id)
   ({value --> ␣}, other) ({value --> you}, id)
   ({value --> !},other)
```

Fonte: autor.

tes. Observe que, no caso de $\alpha = \alpha'$, não ocorreram substituições de parâmetros. As Figuras A.9 e A.10 apresentam os motores de eventos ME_2 e ME_3 reclassificando e processando a sequência de *tokens* obtida na fase anterior, com a potencial identificação e extração de parâmetros em instâncias de macros. Observe que a ocorrência da macro `greeting` no texto da Figura A.1 foi expandida para sua forma normal, considerando o valor do parâmetro informado, a saber, `Good night`.

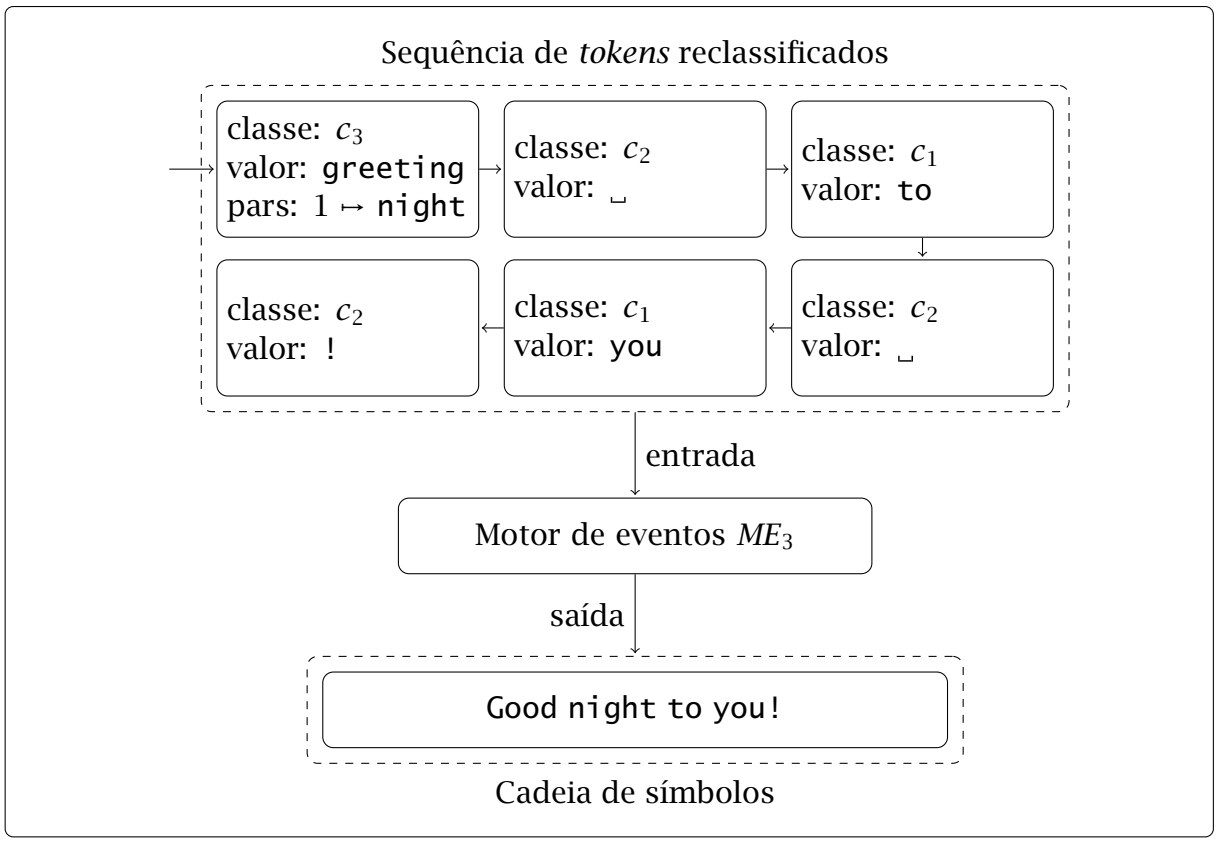
Figura A.9: Motor de eventos ME_2 reclassificando a sequência de *tokens* para o expansor de macros Seção A.1.



Fonte: autor.

A composição hierárquica dos cinco motores de eventos ME_1, ME_4, ME_5, ME_2 e ME_3 (Figuras 3.17, A.3, A.5, 3.21 e 3.23, respectivamente) resulta no projeto completo do expansor de macros com parâmetros da Seção A.1. Observe que o tratamento de eventos ocorre de modo contínuo e sob demanda: quando o motor ME_1 (primeiro nível) dispõe de símbolos suficientes para compor um *token* (incluindo sua classe gramatical), este o envia como evento de entrada para o motor ME_4. No segundo nível, ME_4 potencialmente reclassifica o *token* obtido e o envia como evento

Figura A.10: Motor de eventos ME_3 processando a sequência de *tokens* reclassificados para o expansor de macros da Seção A.1.

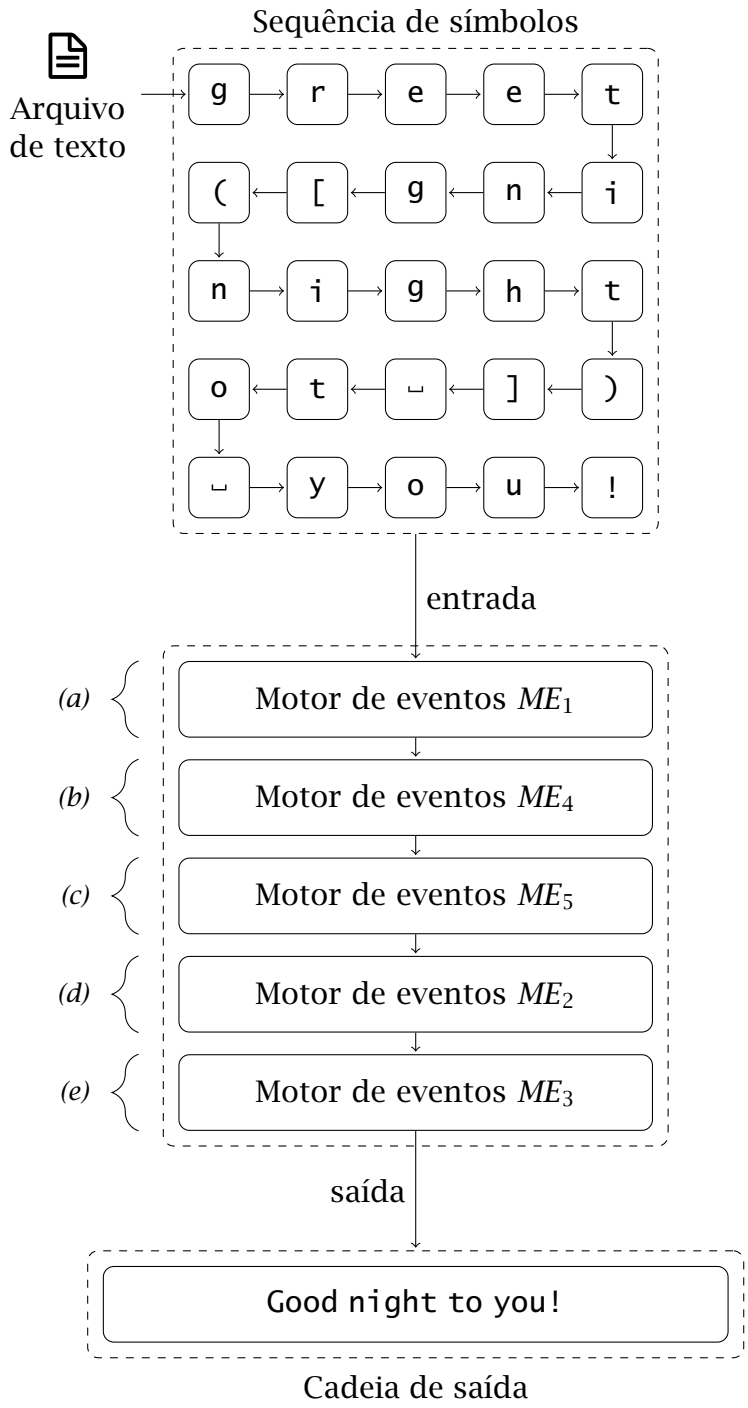


Fonte: autor.

de entrada para o motor ME_5 (terceiro nível). Quando o motor ME_3 dispõe de *tokens* suficientes para identificar potenciais parâmetros (de acordo com um padrão sintático), este compõe um *token* com tais informações e o envia para o motor ME_2 (quarto nível); caso o padrão sintático das estruturas de delimitação não seja identificado, os *tokens* são encaminhados *ipsis litteris* para ME_2. No quarto nível, ME_2 potencialmente reclassifica o *token* obtido e o envia como evento de entrada para o motor ME_3 (quinto nível). O motor ME_3 executa então a ação semântica de acordo com a classe gramatical do *token* obtido e aguarda por novos eventos. Ao término da sequência de símbolos submetida ao primeiro nível, o motor ME_3 emite a cadeia de símbolos resultante. A Figura A.11 ilustra a composição hierárquica dos cinco motores de eventos.

Considere uma implementação do expansor de macros com parâmetros, apresentada na Figura A.12 e detalhada a seguir. A função `process` recebe uma cadeia de símbolos ASCII e uma tabela contendo as definições de macros como parâmetros (`s` e `macros`, a saber), sintetizando as fases de análise léxica, identificação de parâmetros e processamento de *tokens*, e retorna uma cadeia de símbolos sem ocorrências de instâncias de macros. A função utiliza o analisador léxico da Figura A.8 sob demanda, tal que um *token* seja analisado por vez. Caso o *token* corrente (representado

Figura A.11: Composição hierárquica dos cinco motores de eventos ME_1, ME_4, ME_5, ME_2 e ME_3 (Figuras 3.17, A.3, A.5, 3.21 e 3.23, respectivamente), resultando no projeto completo do expansor de macros com parâmetros da Seção A.1.



Legenda: *(a)* sequência de símbolos → sequência de *tokens*, *(b)* sequência de *tokens* → sequência de *tokens* reclassificados, *(c)* sequência de *tokens* reclassificados → sequência de *tokens* (com eventuais parâmetros), *(d)* sequência de *tokens* → sequência de *tokens* reclassificados, e *(e)* sequência de *tokens* → cadeia de símbolos.

Fonte: autor.

pela variável local `a`) seja um identificador (classe gramatical c_1 com rótulo *id*) presente na tabela de macros (representada pelo parâmetro `macros`), a função verifica inicialmente se este possui um conjunto de parâmetros (representado pela variável `parameters`); em caso positivo, `process` realiza substituições de *placeholders* existentes na sequência de substituição por seus respectivos valores devidamente expandidos (através de chamadas recursivas), e concatena a cadeia de símbolos de saída (representada pela variável `result`) com o resultado de uma chamada recursiva de `process` com a sequência de símbolos de substituição correspondente e a tabela de macros fornecidas como parâmetros (`macros[word]` e `macros`, respectivamente). Tal estratégia de implementação garante que, no retorno das chamadas recursivas da função `process`, a cadeia resultante `result` não conterá instâncias de macros remanescentes (expansão de múltiplos passos). Caso o *token* corrente seja um símbolo qualquer (classe gramatical c_2 com rótulo *other*) ou um identificador ausente na tabela de macros, a função concatena a cadeia de símbolos de saída com o valor do *token* (representado pelo índice nominal `a.value`). O processo é então repetido até que o analisador léxico (representado pela função `take`) não possua mais *tokens* a extrair (isto é, a cadeia de símbolos de entrada `s` está vazia). A cadeia de símbolos resultante da expansão é então retornada.

Exemplo 22 (execução do expansor de macros com parâmetros da Figura A.12)**.** A Figura A.13 apresenta um exemplo da execução do expansor de macros com parâmetros da Figura A.12 (função `process`) com o texto e tabela de macros da Figura A.1 (representados pelas variáveis `s` e `macros`, respectivamente), imprimindo a cadeia de símbolos resultante no terminal de comando. □

Conforme ilustra a Figura A.13, a ocorrência de `greeting` (com o parâmetro `night`) foi substituída por `Good night` na cadeia de símbolos de saída. Observe que a tabela de macros (representada pela variável `macros`) contém os padrões sintáticos (lado esquerdo) e as regras de substituição correspondentes (lado direito), incluindo potenciais *placeholders* (representados pelo símbolo # seguido de um índice inteiro positivo).

A.2 EXPANSÃO DE MACROS COM DELIMITADORES CONTEXTUAIS

Considere uma variação da especificação do expansor de macros com parâmetros apresentada na Seção A.1, com delimitadores dependentes de contexto. A Figura A.14 apresenta um exemplo de expansão de macros em um texto, no qual uma instância de macro é transformada em uma sequência de símbolos correspondente de acordo com as regras de substituição definidas em uma tabela. Durante a expansão, o valor associado ao parâmetro da instância de macro é devidamente substituída na sequência resultante.

Figura A.12: Implementação do expansor de macros com parâmetros, sintetizando as fases de análise léxica, identificação de parâmetros e processamento de *tokens*.

```
local function process(s, macros)
  local result = ""
  while #s ~= 0 do
    local a, b = take(s)
    s = b
    local word = a.value.value
    if a.class == 'id' and macros[word] then
      local macro = macros[word]
      local parameters = a.value.parameters or {}
      if #parameters > 0 then
        for i, j in ipairs(parameters) do
          macro = string.gsub(macro, "#" .. tostring(i),
                process(j, macros))
        end
      end
      result = result .. process(macro, macros)
    else
      result = result .. word
    end
  end
  return result
end
```

Fonte: autor.

Figura A.13: Execução do expansor de macros com parâmetros da Figura A.12 com o texto e tabela de macros da Figura A.1, imprimindo a cadeia de símbolos resultante no terminal de comando.

Código-fonte Lua:

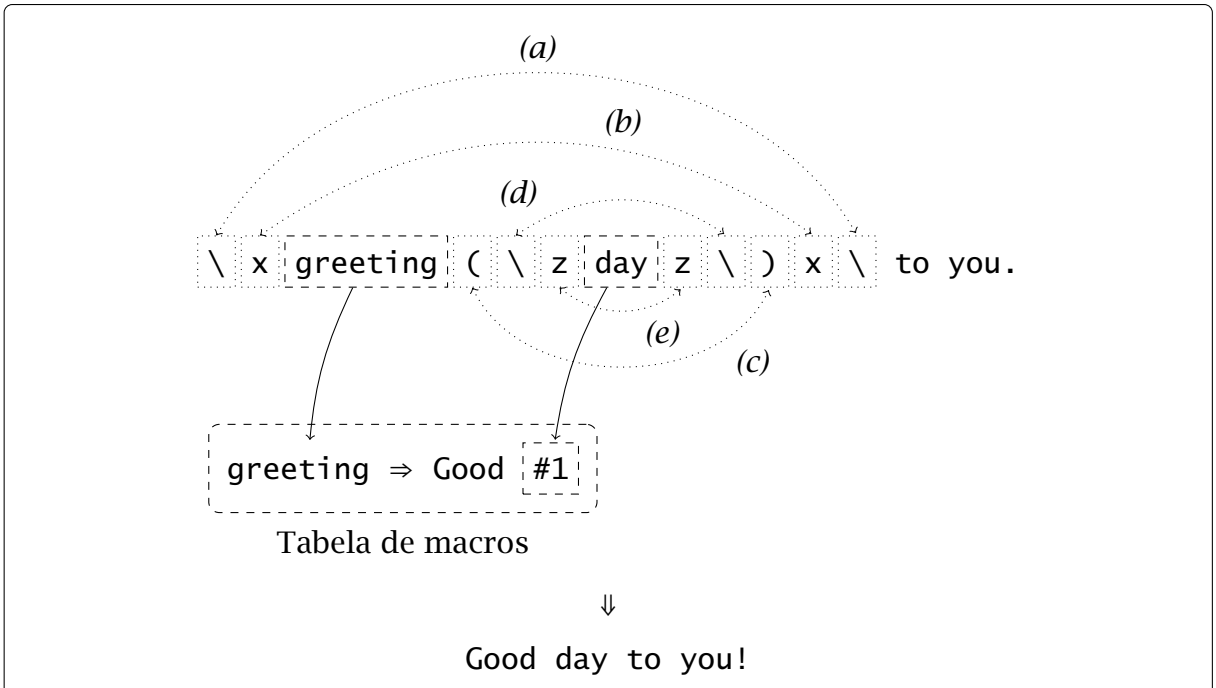
```
local macros = {
  greeting = "Good #1"
}

local s = "greeting[(night)]
   to you!"
print(process(s, macros))
```

Resultado da execução:

```
$ lua ex4.lua

Good night to you!
```

Fonte: autor.

Figura A.14: Exemplo de expansão de macros em um texto, de acordo com a especificação do expansor de macros com delimitadores contextuais.



Legenda: *(a)* delimitadores de macro, *(b)* símbolo contextual de delimitação de macro, *(c)* delimitadores da lista de parâmetros, *(d)* delimitadores de parâmetro, e *(e)* símbolo contextual de delimitação de parâmetro.

Fonte: autor.

Conforme ilustra a Figura A.14, a ocorrência da instância de macro `greeting` (com o parâmetro `day`) é substituída por `Good day` na cadeia de símbolos resultante. Assim como o exemplo do expansor de macros da Seção A.1 (Figura A.1), a sequência associada a `greeting` na tabela de macros possui um *placeholder* (a saber, `#1`) que é substituído pelo valor (devidamente expandido) do parâmetro na instância da macro (a saber, `day`).

A Figura A.15 apresenta o fluxograma contendo as etapas de expansão de macros em um texto. A especificação do expansor de macros proposto (Figura A.14) utiliza marcadores contextuais como delimitadores de abertura e fechamento de macros, tal que uma instância de macro inicializada pelo marcador $\backslash s$ deve ser obrigatoriamente finalizada com o marcador $s\backslash$, com $s \in \text{ASCII} - \{\backslash, (,), \text{␣}\}$, isto é, s pode representar a maioria dos símbolos (imprimíveis) do conjunto ASCII com exceção de barra invertida, parênteses e espaço em branco; consequentemente, o nome da macro não poderá conter s na sequência de símbolos justapostos que o compõe (por exemplo, uma instância da macro `foo` representada por `\ofooo\` seria inválida). Por questões de legibilidade, espaços em branco no início e término dos nomes de macros serão desconsiderados; assim, as instâncias `\xfoox\` e `\x foo x\` referenciam

a mesma macro cujo nome (normalizado) é `foo`. Analogamente, a lista de parâmetros utiliza abertura e fechamento de parênteses como estrutura de delimitação; cada elemento componente da lista é separado por vírgulas e é delimitado pelos marcadores contextuais \$p$ e $p\$, com $p \in \text{ASCII} - \{\backslash, (,), \sqcup\}$. É importante destacar que o parâmetro `\z day z\` não denota uma instância de macro cujo nome é `day`, mas o valor literal associado (incluindo espaços em branco); eventuais instâncias de macros em parâmetros seguem a notação convencional proposta, utilizando delimitadores contextuais (por exemplo, `\x\y foo y\x\` denota uma instância da macro `foo` em um parâmetro). Observe que não há normalização de espaços em branco em valores de parâmetros.

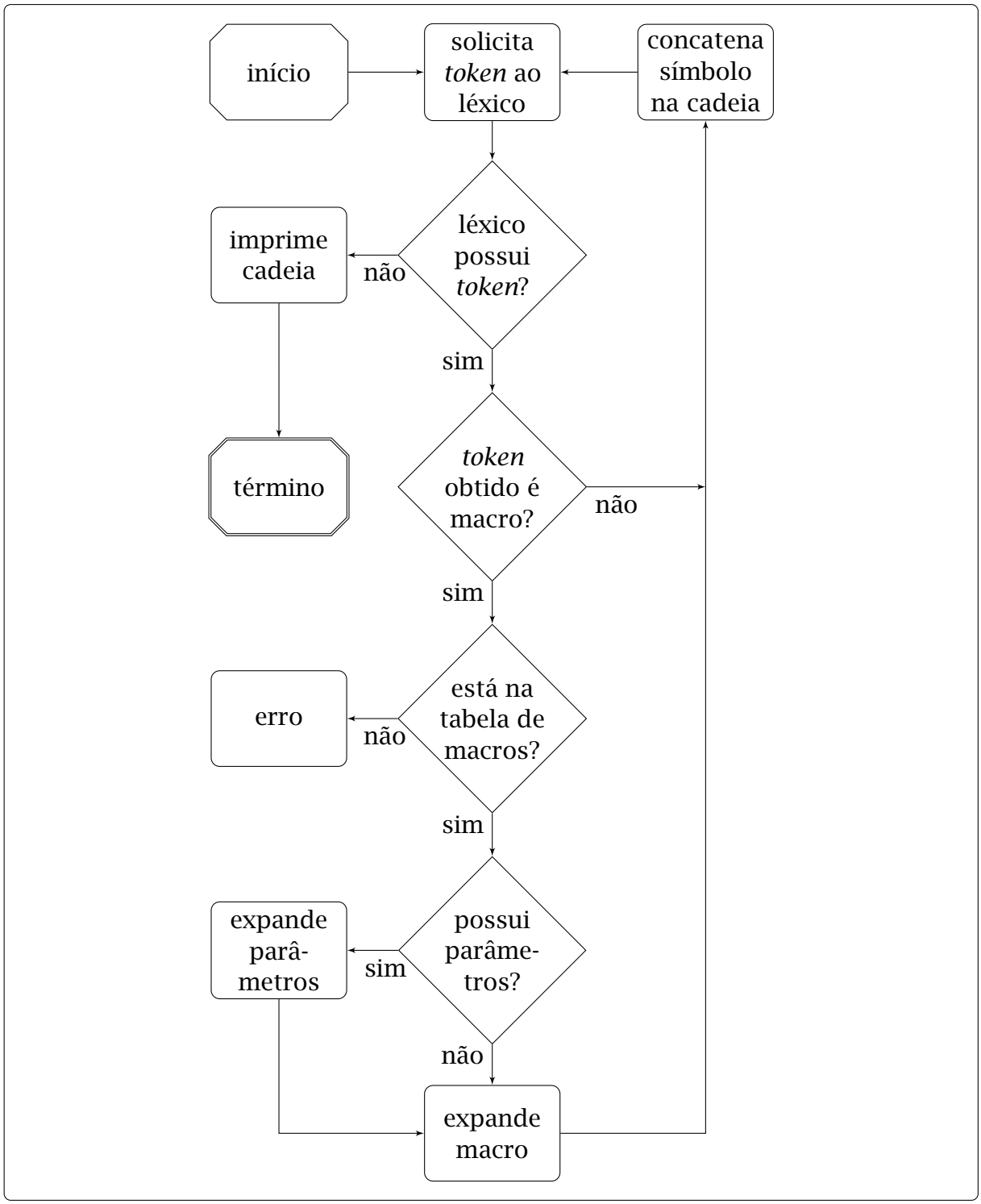
De acordo com a Figura A.15, *tokens* classificados como macros são consultados na tabela de macros. Caso a macro exista na tabela de macros, esta é verificada acerca da existência de parâmetros. Em caso positivo, *placeholders* existentes são substituídos pelos valores expandidos dos parâmetros correspondentes na sequência de transformação. Finalmente, a macro é expandida para a sequência de transformação correspondente e o resultado é concatenado na cadeia de saída. Eventualmente, a sequência de transformação pode conter outras macros. Caso o *token* não seja uma macro (isto é, caso seja um símbolo qualquer), o seu valor é simplesmente concatenado na cadeia de saída. Observe que da ausência de uma macro na tabela de macros resulta um erro de expansão (referência não encontrada), encerrando prematuramente o processamento.

A.2.1 ANÁLISE LÉXICA

O analisador léxico proposto para o expansor de macros com delimitadores contextuais da Seção A.2 foi projetado para categorizar uma sequência de símbolos de um texto em uma sequência de *tokens* de acordo com as classes gramaticais c_2 (símbolo qualquer) e c_3 (instância de macro). A Figura A.16 apresenta um autômato adaptativo M_4 que descreve o analisador proposto, incluindo suporte às dependências de contexto referentes aos marcadores contextuais da instância de macro e dos elementos da lista de parâmetros. O símbolo $\sigma \in \Sigma$ representa o símbolo corrente, e os estados assinalados com c_2 e c_3 indicam as classes gramaticais associadas. As funções adaptativas $\mathcal{B}$ e C são apresentadas nos Algoritmos A.1 e A.2. As linhas tracejadas denotam transições inexistentes na configuração inicial de M_4; estas serão inseridas posteriomente através da aplicação das funções adaptativas $\mathcal{B}$ e C, estabelecendo as dependências contextuais apropriadas.

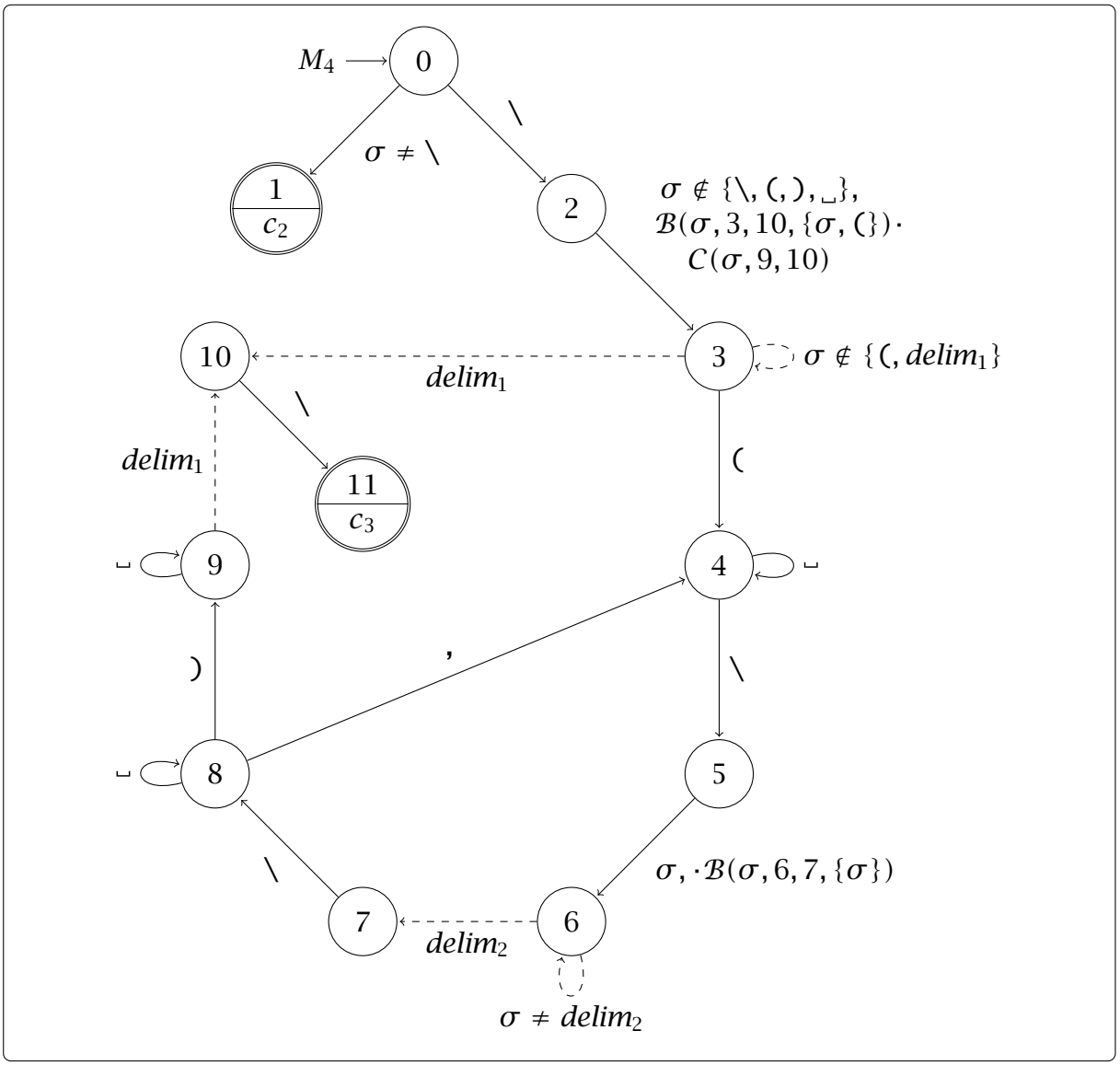
Sejam $w \in \Sigma_1^*$ uma cadeia, tal que $w \in L(M_4)$, e $C\colon \Sigma_1^* \mapsto \{c_2, c_3\}$ uma função que mapeia cadeias de Σ_1 em classes gramaticais c_2 e c_3. A cadeia w será classificada como um símbolo qualquer (classe gramatical c_2) se, e somente se, o autômato adaptativo M_4, após o término do reconhecimento de w, referenciar 1 como estado

Figura A.15: Fluxograma das etapas de expansão de macros com delimitadores contextuais em um texto.



Fonte: autor.

Figura A.16: Autômato adaptativo M_3 que descreve o analisador léxico para o expansor de macros com delimitadores contextuais da Seção A.2. As funções adaptativas $\mathcal{B}$ e C são apresentadas nos Algoritmos A.1 e A.2.



Fonte: autor.

Algoritmo A.1 Função adaptativa $\mathcal{B}(a, b, c, d)$

função adaptativa $\mathcal{B}(a, b, c, d)$
  variáveis: $?x, ?y$
  $?(b, ?x) \rightarrow b$ ▷ atualiza os símbolos válidos como nomes
  $-(b, ?x) \rightarrow b$
  $+(b, \{\alpha \mid \alpha \in \Sigma - d\}) \rightarrow b$
  $?(b, ?y) \rightarrow c$ ▷ atualiza o símbolo de contexto
  $-(b, ?y) \rightarrow c$
  $+(b, a) \rightarrow c$
fim da função adaptativa

Algoritmo A.2 Função adaptativa $C(a, b, c)$

função adaptativa $C(a, b, c)$
  variáveis: $?x$
  $?(b, ?x) \rightarrow c$ ▷ atualiza o símbolo de contexto
  $-(b, ?x) \rightarrow c$
  $+(b, a) \rightarrow c$
fim da função adaptativa

corrente, $C(w) = c_2 \iff (0, w) \vdash^* (1, \epsilon)$. Analogamente, a cadeia w será classificada como uma instância de macro (classe gramatical c_3) se, e somente se, após o término do reconhecimento de w, o autômato adaptativo M_4 referenciar 11 como estado corrente, $C(w) = c_3 \iff (0, w) \vdash^* (11, \epsilon)$. Obserque que, por não estar sendo aqui utilizada, a pilha foi deliberadamente omitida na notação.

Conforme ilustram o exemplo da Figura A.14, o fluxograma da Figura A.15 e o autômato adaptativo M_4 da Figura A.16, potenciais parâmetros são identificados em nível léxico. Considere portanto, as ações semânticas a_7, a_8, a_9 e a_{10} que viabilizam a identificação e extração de tais parâmetros. Sejam $\cdot$ o operador de concatenação, p uma cadeia, $\sigma \in \Sigma$ o símbolo corrente, $delim_1$ e $delim_2$ os símbolos contextuais de delimitação da instância de macro e de um elemento componente da lista de parâmetros, respectivamente, $c \in \mathbb{N}$ uma variável denotando o contador de parâmetros, e t um novo *token*, com $pars(t)$ representando o conjunto de parâmetros associado a t, inicialmente vazio, tal que $pars(t) = \varnothing$. As ações semânticas a_7, a_8, a_9 e a_{10} são definidas a seguir, nas Equações A.7, A.8, A.9 e A.10.

$$a_7 \equiv value(t) \leftarrow \sigma \tag{A.7}$$

$$a_8 \equiv value(t) \leftarrow \epsilon \quad \text{e} \quad c \leftarrow 0 \tag{A.8}$$

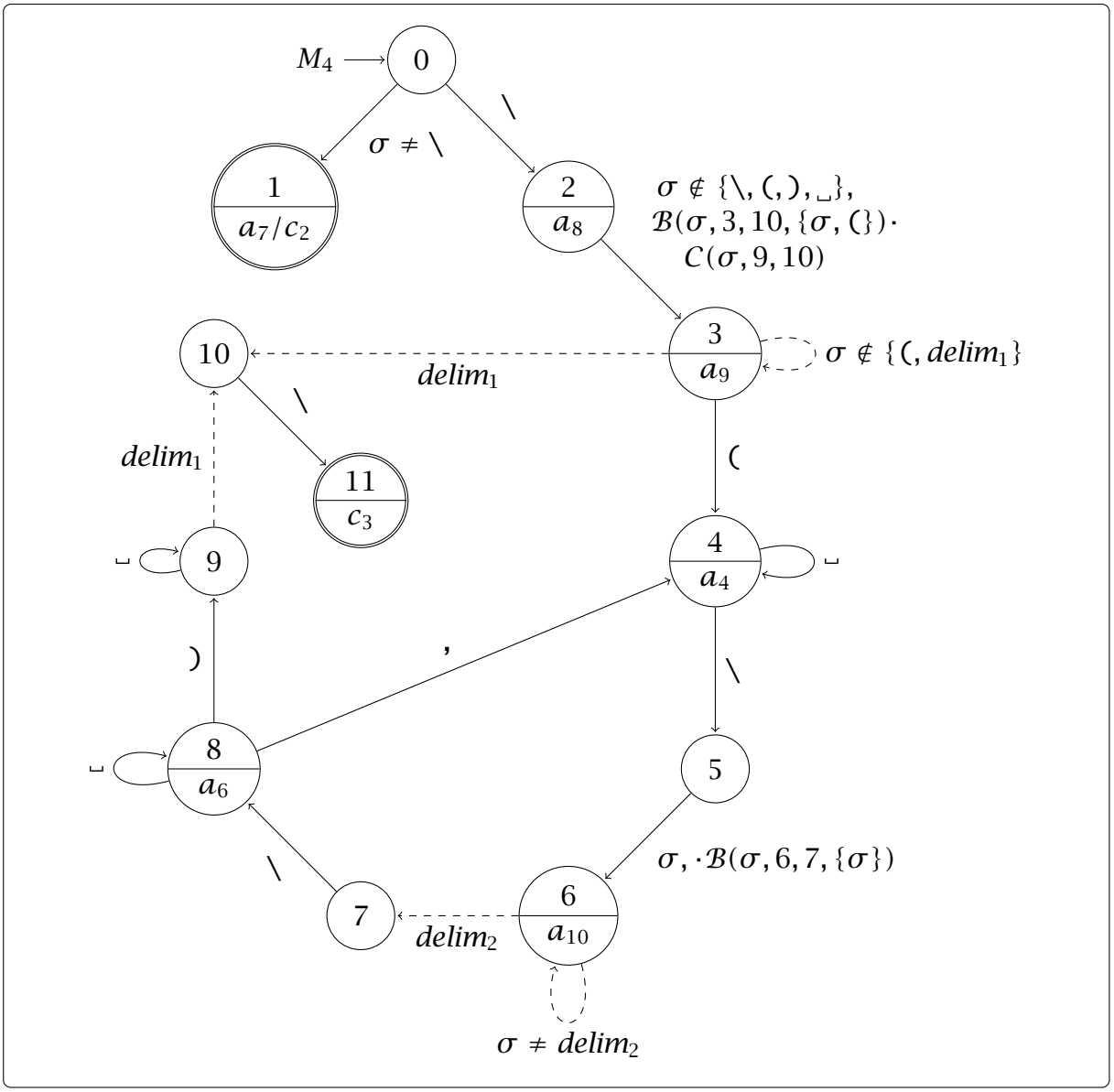
$$a_9 \equiv \sigma \neq delim_1 \implies value(t) \leftarrow value(t) \cdot \sigma \tag{A.9}$$

$$a_{10} \equiv \sigma \neq delim_2 \implies value(t) \leftarrow value(t) \cdot \sigma \tag{A.10}$$

Sejam $w \in \Sigma_4^*$ uma cadeia, tal que $w \in L(M_4)$, e $A\colon Q_4 \mapsto \{a_4, a_6, a_7, a_8, a_9, a_{10}\}$ uma função que mapeia estados de M_4 em ações semânticas a_4, a_6, a_7, a_8, a_9 e a_{10} (as ações semânticas a_4 e a_6 foram definidas na Seção A.1, nas Equações A.3 e A.5). A ação semântica a_i será disparada se, e somente se, o autômato adaptativo M_4, durante o reconhecimento de w, referenciar j como estado corrente após o consumo de um símbolo $w_k \in w$, $a_i \iff (\alpha, w_k) \vdash (j, \epsilon)$, com $(j, a_i) \in A$. A função A é definida por extensão como $A = \{(1, a_7), (2, a_8), (3, a_9), (4, a_4), (6, a_{10}), (8, a_6)\}$. A Figura A.17 apresenta o autômato adaptativo M_4 da Figura A.16 acrescido das ações semânticas a_4, a_6, a_7, a_8, a_9 e a_{10}.

A Figura A.18 apresenta um motor de eventos ME_6 como modelo de implementação do analisador léxico para o expansor de macros com delimitadores contextuais da Seção A.2. No exemplo, a sequência de 29 símbolos é submetida ao motor de

Figura A.17: Autômato adaptativo M_4 da Figura A.16 acrescido das ações semânticas a_4, a_6, a_7, a_8, a_9 e a_{10}.



Fonte: autor.

eventos, disparando ações correspondentes à análise léxica (categorização em classes gramaticais c_2 e c_3) e identificação e extração de potenciais parâmetros (ações semânticas a_4, a_6, a_7, a_8, a_9 e a_{10}); como resultado, uma sequência de 6 *tokens* é gerada como saída do motor de eventos. Observe que o parâmetro da macro `greeting` (Figura A.14) foi devidamente extraído e associado ao *token* correspondente.

Considere uma implementação do analisador léxico proposto, incluindo a identificação e extração de potenciais parâmetros, detalhada a seguir. A Figura A.19 apresenta duas funções auxiliares, `normalize` e `ignored`, para normalização de espaços em branco em nomes de macros e verificação de símbolos potencialmente descartáveis (espaços em branco, tabulações e quebras de linha). Tais funções correspondem semanticamente aos laços nos estados 3, 4, 8 e 9 do autômato adaptativo M_4 (Figura A.16).

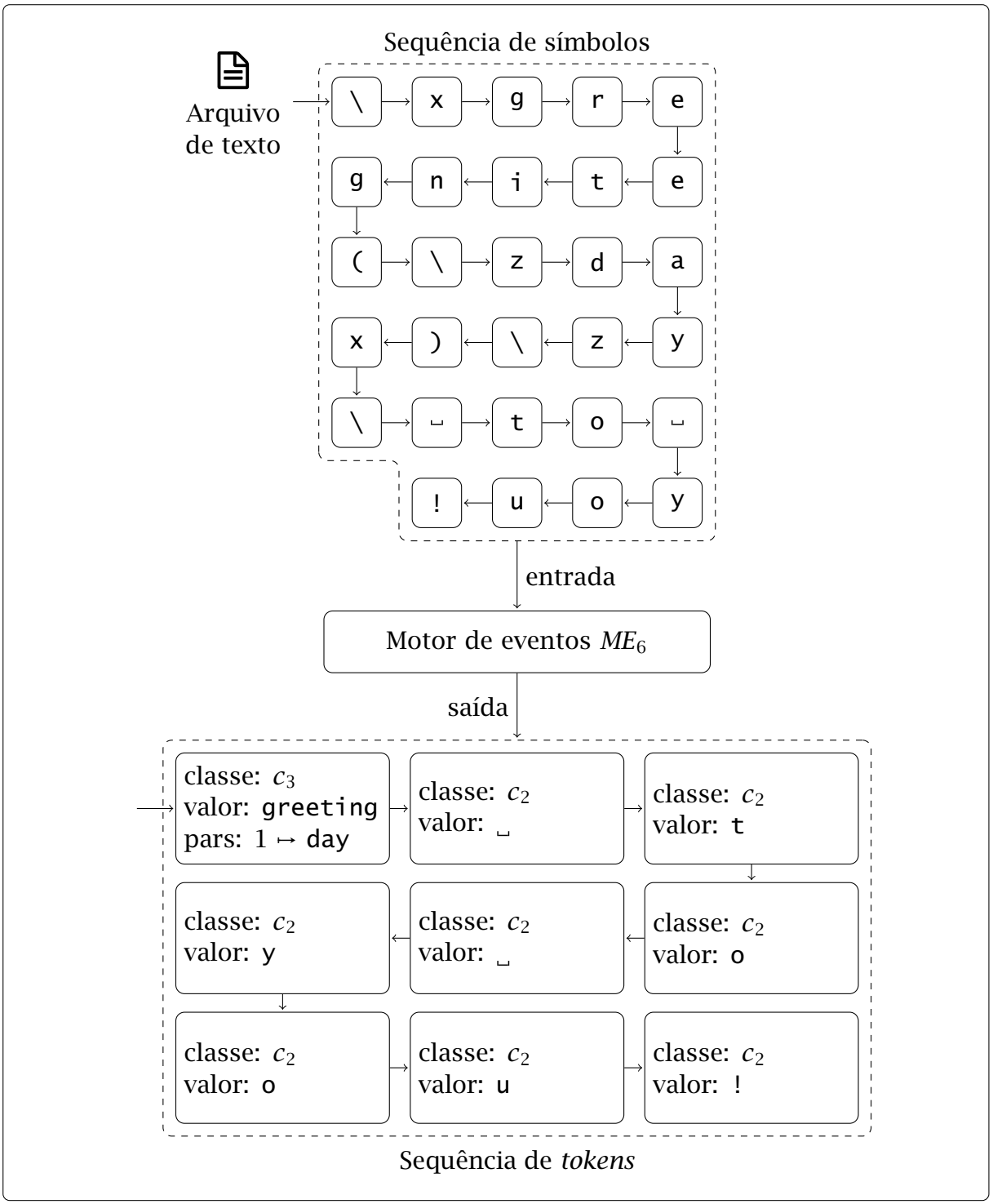
De acordo com a Figura A.19, a função `normalize` remove espaços em branco excedentes da cadeia fornecida como parâmetro (a saber, `s`) e a retorna, enquanto a função `ignored` verifica se o símbolo fornecido como parâmetro (a saber, `s`) é potencialmente descartável, retornando um valor lógico correspondente. A Figura A.20 apresenta a função `take` que efetivamente implementa o analisador léxico proposto; tal função recebe uma cadeia de símbolos como parâmetro (a saber, `s`) e retorna uma tupla contendo o *token* obtido (incluindo potenciais parâmetros de uma instância de macro) e o restante da cadeia para uso posterior.

A função `take`, conforme ilustra a Figura A.20, implementa o autômato adaptativo M_4 da Figura A.16, incluindo as ações semânticas associadas (a saber, a_4, a_6, a_7, a_8, a_9 e a_{10}). É importante destacar que a normalização de espaços (função `normalize`) ocorre somente em nomes de macros; valores associados aos parâmetros permanecem inalterados. Adicionalmente, situações configuradas como erro (por exemplo, delimitadores inválidos, violação de contexto ou instâncias incompletas) encerram prematuramente a execução (através da execução da função `os.exit`).

Exemplo 23 (execução do analisador léxico introduzido na Subseção A.2.1)**.** A Figura A.21 apresenta um exemplo da execução do analisador léxico da Figura A.20 (função `take`) com o texto da Figura A.14 (representado pela variável `s`), de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Observe que a sequência de *tokens* obtida é a mesma retornada pelo motor de eventos da Figura 3.17, como esperado. A função auxiliar `pp`, definida na Figura A.8, produz uma representação textual de tabelas na linguagem Lua. □

Linguagens de macros que apresentam símbolos explícitos de ocorrência de macros, em geral, possibilitam a detecção (e eventual recuperação) de erros na própria análise léxica (por exemplo, como ocorre na linguagem TEX). Tal característica potencialmente reduz a quantidade de verificações adicionais em fases subsequentes

Figura A.18: Motor de eventos ME_6 como modelo de implementação do analisador léxico para o expansor de macros da Seção A.2. A sequência de símbolos utiliza o texto da Figura A.14.



Fonte: autor.

Figura A.19: Funções auxiliares de normalização de espaços em branco em nomes de macros e de verificação de símbolos potencialmente descartáveis (espaços em branco, tabulações e quebras de linha).

```
local function normalize(s)
  return string.gsub(string.gsub(string.gsub(s, "%s+", " "),
             "%s$", ""), "^%s", "")
end

local function ignored(s)
  return string.find(s, "^%s$") ~= nil
end
```

Fonte: autor.

do processo de expansão de macros. O processamento da sequência de *tokens* resultante é discutido em detalhes na Subseção A.2.2.

A.2.2 PROCESSAMENTO DE *TOKENS*

O processamento de *tokens* para o expansor de macros da Seção A.2 é similar ao descrito na Subseção A.1.2, com a ressalva de desconsiderar a reclassificação da sequência (motor de eventos ME_2 da Figura A.9), dado que instâncias de macros são identificadas no primeiro nível (tornando redundante, portanto, a aplicação de uma função de reclassificação de *tokens*). A Figura A.22 apresenta o motor de eventos ME_3 processando a sequência de *tokens* obtida na fase anterior. Observe que a ocorrência da macro `greeting` no texto da Figura A.14 foi expandida para sua forma normal, considerando o valor do parâmetro informado, a saber, `Good day`.

A composição hierárquica dos dois motores de eventos ME_6 e ME_3 (Figuras A.18 e 3.23, respectivamente) resulta no projeto completo do expansor de macros com delimitadores contextuais da Seção A.2. Observe que o tratamento de eventos ocorre de modo contínuo e sob demanda: quando o motor de eventos ME_6 (primeiro nível) dispõe de símbolos suficientes para compor um *token* (incluindo sua classe gramatical e potenciais parâmetros associados), este o envia como evento de entrada para o motor ME_3. No segundo nível, ME_3 executa a ação semântica de acordo com a classe gramatical do *token* obtido e aguarda por novos eventos. Ao término da sequência de símbolos submetida ao primeiro nível, o motor ME_3 emite a cadeia resultante. A Figura A.23 ilustra a composição hierárquica dos dois motores de eventos.

Considere uma implementação do expansor de macros da Seção A.2, apresentada na Figura A.24 e detalhada a seguir. A função `process` recebe uma cadeia de símbolos ASCII e uma tabela contendo as definições de macros como parâmetros (`s` e `macros`, a saber), sintetizando as fases de análise léxica e processamento de *tokens*, e

Figura A.20: Implementação do analisador léxico proposto na Subseção A.2.1. O trecho de código implementa os estados 1, 2, 3 e 4 do autômato adaptativo M_4 da Figura A.16.

```
local function take(s)
  local state, input, symbol = 1, s
  local macro, parameter, delim1, delim2 = "", ""
  local parameters = {}
  while #input > 0 do
    symbol = head(input); input = tail(input, 2)
    if state == 1 then if symbol ~= '\\' then
       return token({ value = symbol }, "other"), input
     else state = 2; end; elseif state == 2 then
     if symbol == "(" then
       print("Invalid delimiter."); os.exit(0)
     else delim1 = symbol; state = 3; end
   elseif state == 3 then
     if symbol == "(" then state = 5
     elseif symbol == delim1 then state = 4
     else macro = macro .. symbol end
   elseif state == 4 then
     if symbol ~= "\\" then
       print("Closing expected."); os.exit(0)
     else state = 1; macro = normalize(macro); break; end
   elseif state == 5 then
     if not ignored(symbol) then
       if symbol ~= "\\" then
         print("Delimiter expected."); os.exit(0)
       else state = 6; end ; end
   elseif state == 6 then
     delim2 = symbol; state = 7
   elseif state == 7 then
     if symbol ~= delim2 then
       parameter = parameter .. symbol
     else state = 8; end
   elseif state == 8 then
     if symbol ~= "\\" then
       print("Closing expected."); os.exit(0)
     else table.insert(parameters, parameter)
       parameter = ""; state = 9; end
   elseif state == 9 then
     if not ignored(symbol) then
       if symbol == "," then state = 5
       else if symbol == ")" then state = 10
         else print("Expected rpar."); os.exit(0)
         end ; end; end
   elseif state == 10 then
     if not ignored(symbol) then
       if symbol ~= delim1 then
         print("Delimiter expected."); os.exit(0)
       else state = 4; end; end; end; end
  if state ~= 1 then print("Invalid macro."); os.exit(0)
  else return token({ value = macro, parameters = parameters }, "macro"),
      input; end; end
```

Fonte: autor.

Figura A.21: Execução da função `take` da Figura A.20 com o texto da Figura A.14, de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. Espaços em branco (exceto os que foram utilizados como separadores de *tokens* na impressão) estão deliberadamente marcados como visíveis.

Código-fonte Lua:

```
local s, a = "\\xgreeting(\\zdayz\\)x\\ to you!"

while #s ~= 0 do
  a, s = take(s)
  io.write("(" .. pp(a.value) .. ", " .. a.class .. ") ")
end
```

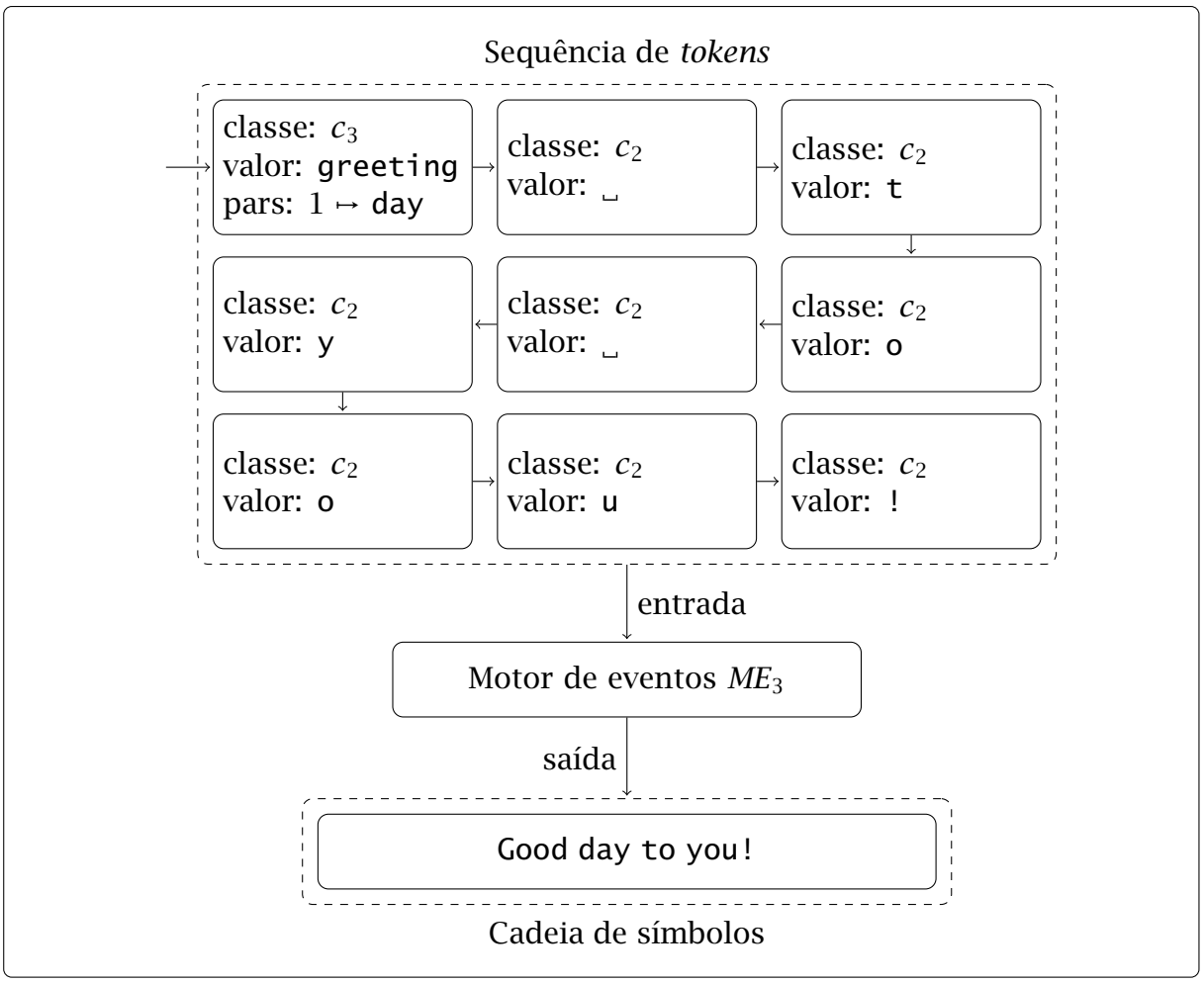
Resultado da execução:

```
$ lua ex5.lua

({parameters --> {1 --> day}, value --> greeting}, macro)
   ({value --> ␣}, other) ({value --> t}, other)
   ({value --> o}, other) ({value --> ␣}, other)
   ({value --> y}, other) ({value --> o}, other)
   ({value --> u}, other) ({value --> !}, other)
```

Fonte: autor.

retorna uma cadeia de símbolos sem ocorrências de instâncias de macros. A função utiliza o analisador léxico da Figura A.21 sob demanda, de modo tal que um *token* seja analisado de cada vez. Caso o *token* corrente (representado pela variável local `a`) seja uma instância de macro (classe gramatical c_3 com rótulo *macro*), a função verifica inicialmente se este possui um conjunto de parâmetros (representado pela variável `parameters`); em caso positivo, `process` realiza substituições de *placeholders* existentes na sequência de substituição por seus respectivos valores devidamente expandidos (através de chamadas recursivas), e concatena a cadeia de símbolos de saída (representada pela variável `result`) com o resultado de uma chamada recursiva de `process` com a sequência de símbolos de substituição correspondente e a tabela de macros fornecidas como parâmetros (`macros[word]` e `macros`, respectivamente). Tal estratégia de implementação garante que, no retorno das chamadas recursivas da função `process`, a cadeia resultante `result` não conterá instâncias de macros remanescentes. Caso o *token* corrente seja um símbolo qualquer (classe gramatical c_2 com rótulo *other*), a função concatena a cadeia de símbolos de saída com o valor do *token* (representado pelo índice nominal `a.value`). O processo é então repetido até que o analisador léxico (representado pela função `take`) não possua

Figura A.22: Motor de eventos ME_3 processando a sequência de *tokens* para o expansor de macros da Seção A.2.



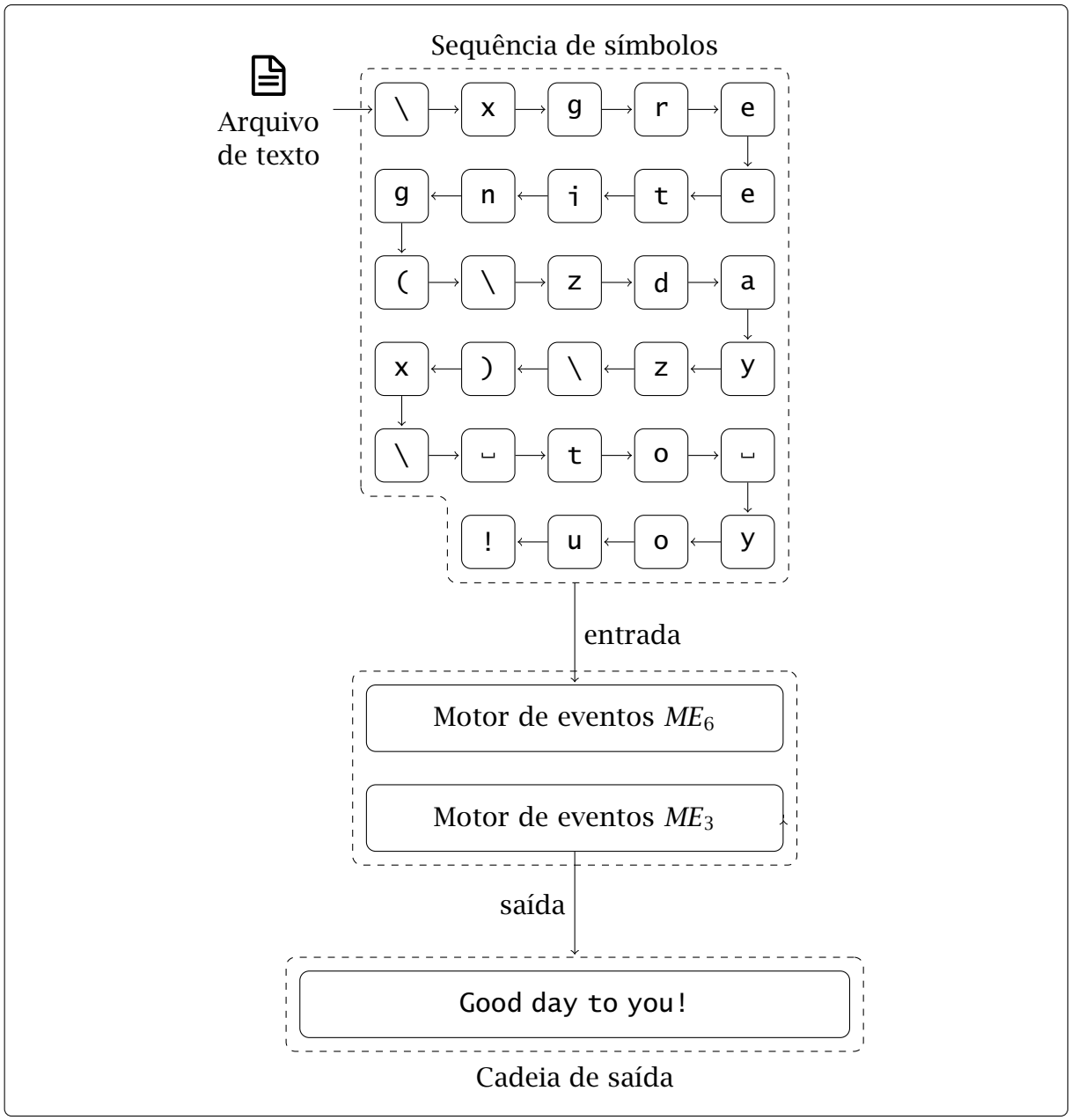
Fonte: autor.

mais *tokens* a extrair (isto é, a cadeia de símbolos de entrada `s` está vazia). A cadeia de símbolos resultante da expansão é então retornada. Observe que, de acordo com o fluxograma da Figura A.15, da ausência de uma macro na tabela de macros resulta um erro de expansão (referência não encontrada), encerrando prematuramente o processamento.

Exemplo 24 (execução do expansor de macros da Figura A.24)**.** A Figura A.25 apresenta um exemplo da execução do expansor de macros da Figura A.24 (função `process`) com o texto e tabela de macros da Figura A.14 (representados pelas variáveis `s` e `macros`, respectivamente), imprimindo a cadeia de símbolos resultante no terminal de comando. □

Conforme ilustra a Figura A.25, a ocorrência de `greeting` (com o parâmetro `day`) foi substituída por `Good day` na cadeia de símbolos de saída. A estratégia de expansão utilizada na função `process` é idêntica à adotada na Seção A.1, na qual assegura-se a expansão de macros internas em sequências de substituição de macros mais

Figura A.23: Composição hierárquica dos dois motores de eventos ME_6 e ME_3 (Figuras A.18 e 3.23, respectivamente), resultando no projeto completo do expansor de macros da Seção A.2.



Fonte: autor.

Figura A.24: Implementação do expansor de macros da Seção A.2, sintetizando as fases de análise léxica e processamento de *tokens*.

```
local function process(s, macros)
  local result = ""
  while #s ~= 0 do
    local a, b = take(s)
    s = b
    local word = a.value.value
    if a.class == 'macro' then
      if macros[word] then
        local macro = macros[word]
        local parameters = a.value.parameters
        if #parameters > 0 then
          for i, j in ipairs(parameters) do
            macro = string.gsub(macro, "#" .. tostring(i),
                process(j, macros))
          end
        end
        result = result .. process(macro, macros)
      else print('Macro not found.'); os.exit(0); end
    else
      result = result .. word
    end
  end
  return result
end
```

Fonte: autor.

Figura A.25: Execução do expansor de macros da Figura A.24 com o texto e tabela de macros da Figura A.14, imprimindo a cadeia de símbolos resultante no terminal de comando.

Código-fonte Lua:

```
local macros = {
  greeting = "Good #1"
}

local s = "\\xgreeting" ..
   (\\zdayz\\)x\\ to you!"
print(process(s, macros))
```

Resultado da execução:

```
$ lua ex6.lua

Good day to you!
```

Fonte: autor.

externas (incluindo expansão de instâncias macros em valores de parâmetros antes de sua efetiva substituição no corpo da macro paramétrica corrente); entretanto, não há garantias quanto à terminação da reescrita de termos.

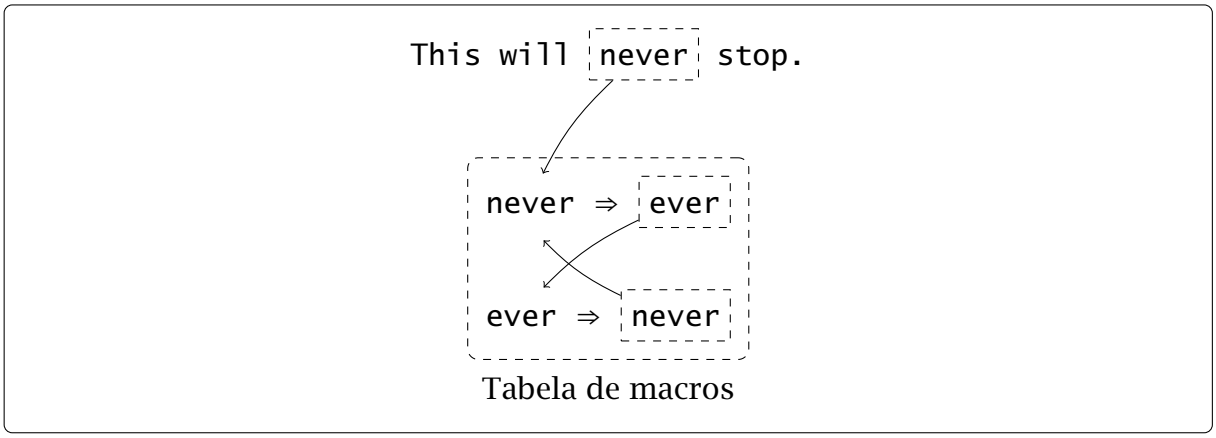
A.3 FUNCIONALIDADES COMPLEMENTARES

Expansores de macros podem incorporar funcionalidades complementares, com o objetivo de torná-los mais expressivos ou aderentes a certas propriedades (KOHLBECKER, 1986; BRABRAND; SCHWARTZBACH, 2002; BURMAKO, 2012). Esta seção contempla três funcionalidades elegíveis para inclusão em expansores de macros, a saber: validação da tabela de macros, tratamento de parâmetros como macros de escopo local e transformações algorítmicas. Tais funcionalidades são detalhadas a seguir.

A.3.1 VALIDAÇÃO DA TABELA DE MACROS

A terminação de reescrita de termos é, em geral, um problema indecidível (HERMANN; KIRCHNER; KIRCHNER, 1991). O exemplo da Figura A.26 apresenta uma dependência cícilica entre duas macros com padrão sintático regular (a saber, `never` e `ever`); tais macros são indiretamente recursivas (Definição 52).

Figura A.26: Exemplo de dependência cíclica na expansão de macros com padrão sintático regular (tipo 3 na hierarquia de Chomsky) em um texto. A reescrita de termos jamais terminará.



Fonte: autor.

Conforme ilustra a Figura A.26, a ocorrência de `never` no texto é substituída inicialmente pela sequência `ever`. Esta, por sua vez, é identificada como uma instância de macro e substituída pela sequência `never`. As transformações se alternarão ininterruptamente entre `never` e `ever` e jamais serão reduzidas às suas formas normais; a reescrita de termos, portanto, não terminará.

É importante destacar que existem estudos no estabelecimento de condições para garantia de terminação através de ordenações de redução (DERSHOWITZ, 1982, 1987; JOUANNAUD; LESCANNE; REINIG, 1982; LESCANNE, 1984; HUET, 1980). Em particular, o trabalho de Bove e Arbilla (1991, 1992) apresenta restrições da lista de definições notacionais que garantem a propriedade noetheriana do sistema de reescrita proposto. A seguir, é apresentada uma técnica de validação da tabela de macros do expansor de macros com padrão sintático regular da Seção 3.4 (página 65), inspirada em tais restrições.

Sejam $\phi\colon \Sigma^* \mapsto \mathbb{N}$ uma função que associa um número natural como identificador unívoco a um nome de macro, de acordo com uma relação de ordem entre nomes de macros estabelecida *a priori* (por exemplo, de acordo com a posição de definição na tabela de macros), *TM* é a tabela de macros (representada como uma relação binária), e $take(w)$ uma função que representa a análise léxica sobre a cadeia de símbolos w, retornando uma sequência de *tokens*. A função de placar *score* é definida na Equação A.11.

$$score(s) = \begin{cases} \phi(s) & \text{se } (s,\alpha) \in TM \\ 0 & \text{caso contrário} \end{cases} \tag{A.11}$$

A função *score*, de acordo com a Equação A.11, retorna o identificador unívoco correspondente à cadeia w (aplicação da função ϕ), na condição de que w exista na tabela de macros, ou zero caso contrário. A função de placar máximo Φ é definida a seguir, na Equação A.12.

$$\begin{aligned} \Phi(w) = \max(\{score(value(t)) \mid (w,\alpha) \in TM \\ \wedge\ t \in take(\alpha)\}) \end{aligned} \tag{A.12}$$

A função de placar máximo Φ (Equação A.12) retorna o maior valor entre os placares dos *tokens* obtidos através da aplicação da função *take* sobre a sequência de transformação correspondente à macro w. A função de validação de macros $validate\colon \Sigma^* \mapsto \{\text{true}, \text{false}\}$ é definida a seguir, na Equação A.13.

$$validate(w) = \begin{cases} \text{true} & \text{se } \phi(w) > \Phi(w) \\ \text{false} & \text{caso contrário} \end{cases} \tag{A.13}$$

De acordo com a Equação A.13, uma macro pode conter instâncias de outras macros definidas previamente (isto é, cujo placar máximo entre as instâncias de macros seja menor do que o placar da macro corrente). Caso contrário, tal macro é considerada inválida. Observe que a relação de ordem entre nomes de macros torna-se determinante. Uma tabela de macros *TM* é, portanto, dita válida se, e somente se, todas as macros que a constitui são válidas, $\forall (a,b) \in TM, validate(a)$.

Considere uma implementação da validação da tabela de macros, apresentada na Figura A.27 e detalhada a seguir. A função `validate` recebe uma tabela contendo

as definições de macros como parâmetro (a saber, `macros`), verifica as macros existentes conforme indica a função *validate* (Equação A.13) e retorna um valor lógico correspondente à validação.

Figura A.27: Implementação da validação da tabela de macros da Subseção A.3.1.

```
local function validate(macros)
  for _, macro in pairs(macros) do
    local s = macro.value
    while #s ~= 0 do
      local a, b = take(s)
      s = b
      if a.class == "id" and macros[a.value] then
        if macro.order <= macros[a.value].order then
          return false
        end
      end
    end
  end
  return true
end
```

Fonte: autor.

De acordo com a Figura A.27, para cada definição presente na tabela de macros (representada pela variável `macros`), a sequência de substituição correspondente (representada pelo índice nominal `macro.value`) é submetida ao analisador léxico (representado pela função `take` da Figura 3.19), que retorna uma sequência de *tokens*. Os placares das eventuais instâncias de macros encontradas na sequência obtida (calculados através da função *score* da Equação A.11) são comparados com o placar da definição de macro corrente; caso um ou mais placares das instâncias de macros sejam iguais ou maiores que o placar da definição de macro corrente, esta é considerada inválida e a verificação é encerrada prematuramente, indicando que a tabela de macros fornecida como parâmetro é inválida. Na situação em que todas as definições de macros sejam válidas (conforme ilustra a Equação A.13) a função retorna um valor lógico indicando que toda a tabela de macros fornecida como parâmetro é válida.

Exemplo 25 (execução da função de validação da Figura A.27)**.** A Figura A.28 apresenta um exemplo da execução da função de validação `validate` da Figura A.27 com as tabelas de macros das Figuras 3.14 e A.26 (representadas pelas variáveis `m1` e `m2`, respectivamente), devidamente acrescidas de identificadores unívocos. Observe que a tabela de macros da Figura 3.14 não apresentou inconsistências, enquanto a de-

pendência cíclica identificada invalidou a tabela de macros da Figura A.26, como esperado. □

Figura A.28: Execução da função de validação `validate` da Figura A.27 com as tabelas de macros das Figuras 3.14 e A.26.

Código-fonte Lua:

```
local m1 = {
  gift = { value = "bicycle", order = 1 },
  season = { value = "Christmas", order = 2 }
}

local m2 = {
  never = { value = "ever", order = 1 },
  ever = { value = "never", order = 2 }
}

print(validate(m1))
print(validate(m2))
```

Resultado da execução:

```
$ lua ex7.lua

true
false
```

Fonte: autor.

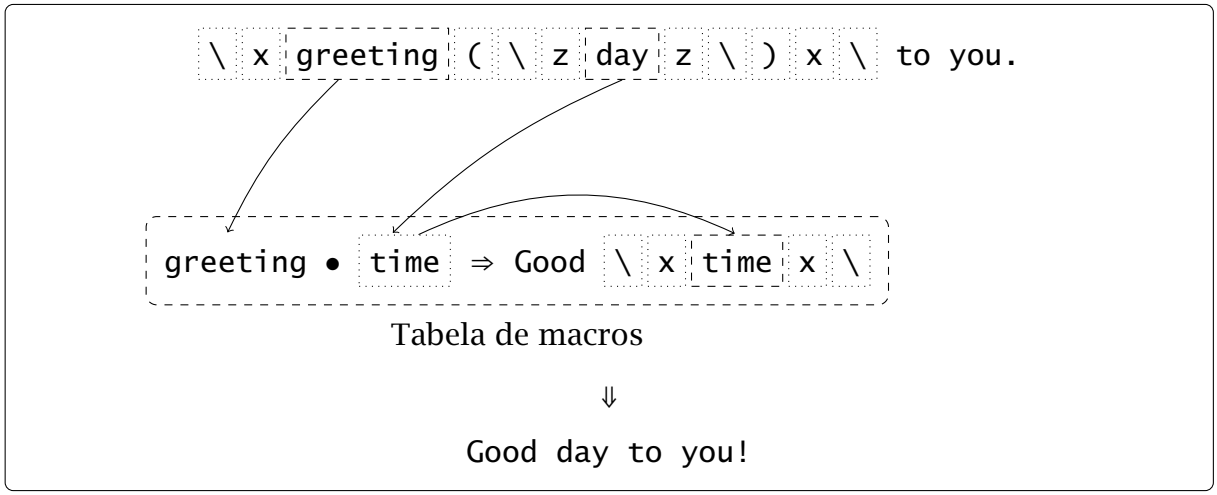
A validação da tabela de macros em tempo de definição garante que os expansores de macros das Seções 3.4, A.1 e A.2 sejam noetherianos (isto é, tenham garantias de término da reescrita de termos) (BOVE; ARBILLA, 1991).

A.3.2 RESOLUÇÃO DE PARÂMETROS COMO MACROS DE ESCOPO LOCAL

As especificações dos expansores de macros das Seções A.1 e A.2 utilizam uma sintaxe fortemente inspirada na linguagem TEX (KNUTH, 1986) para designar *placeholders*, na qual o símbolo # denota o valor de um parâmetro determinado por um número natural associado. A Figura A.29 apresenta uma variação do exemplo de expansão de macros em um texto da Figura A.14, adotando uma estratégia de resolução de parâmetros como macros de escopo local com padrão sintático regular (tipo 3 na hierarquia de Chomsky).

A estratégia de resolução de parâmetros como macros de escopo local com padrão sintático regular substitui *placeholders* por instâncias de macros definidas lo-

Figura A.29: Exemplo de expansão de macros em um texto utilizando uma estratégia de resolução de parâmetros como macros de escopo local com padrão sintático regular.



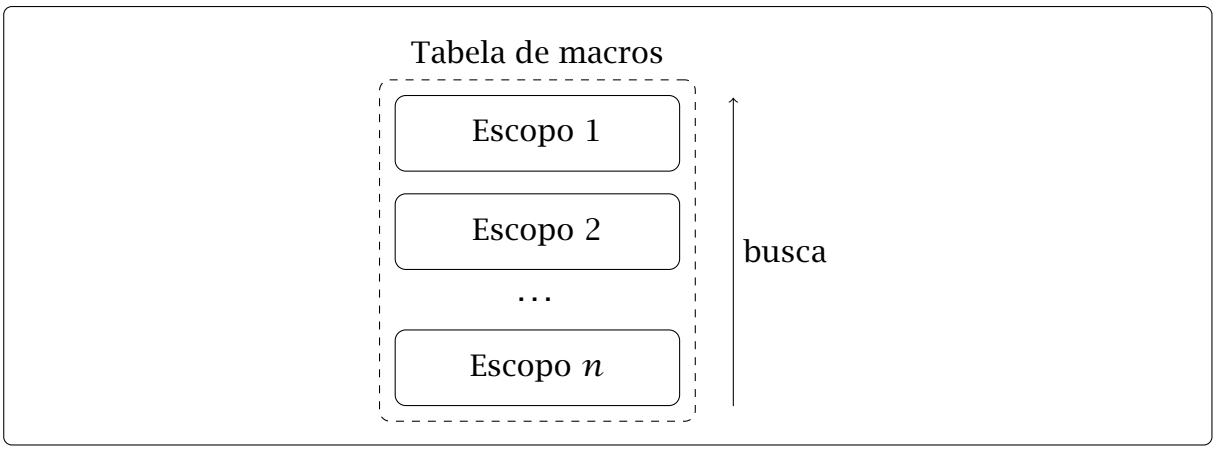
Fonte: autor.

calmente na tabela de macros. Consequentemente, as definições de tais macros correspondem aos valores dos parâmetros (devidamente expandidos) da instância de macro original a ser expandida. Macros locais são removidas do escopo corrente na tabela de macros ao término da expansão da macro original. Conforme ilustra a Figura A.29, a ocorrência da instância de macro `greeting` (com o parâmetro `day`) é substituída por `Good day` na cadeia de símbolos resultante. A sequência associada a `greeting` na tabela de macros possui uma macro local (a saber, `time`) que é definida com o valor (devidamente expandido) do parâmetro na instância da macro (a saber, `day`). Durante a expansão de `greeting`, a macro local `time` é identificada e expandida para o valor correspondente. No término da expansão de `greeting`, a macro local é removida do escopo corrente na tabela de macros.

A tabela de macros utilizada nos expansores de macros das Seções 3.4, A.1 e A.2 não contempla a caracterização de escopo. É necessário, portanto, incluir na tabela de macros regiões ordenadas para eventuais definições de macros locais, conforme o processo de expansão corrente. A Figura A.30 apresenta uma tabela de macros estendida com a inclusão de escopo. É importante destacar que o primeiro nível representa macros definidas globalmente, enquanto o último nível (representado pelo escopo n) representa macros definidas localmente. A busca na tabela de macros incia-se no escopo local e encerra-se no escopo global (optou-se, na situação em que duas macros tenham o mesmo nome, pela priorização da definição local).

De acordo com a Figura A.30, a fase de processamento de *tokens* deve considerar a existência de escopos na tabela de macros, tal que eventuais parâmetros de instâncias de macros sejam tratados adequadamente. A Figura A.31 apresenta o fluxograma contendo as etapas da resolução de parâmetros como macros de escopo

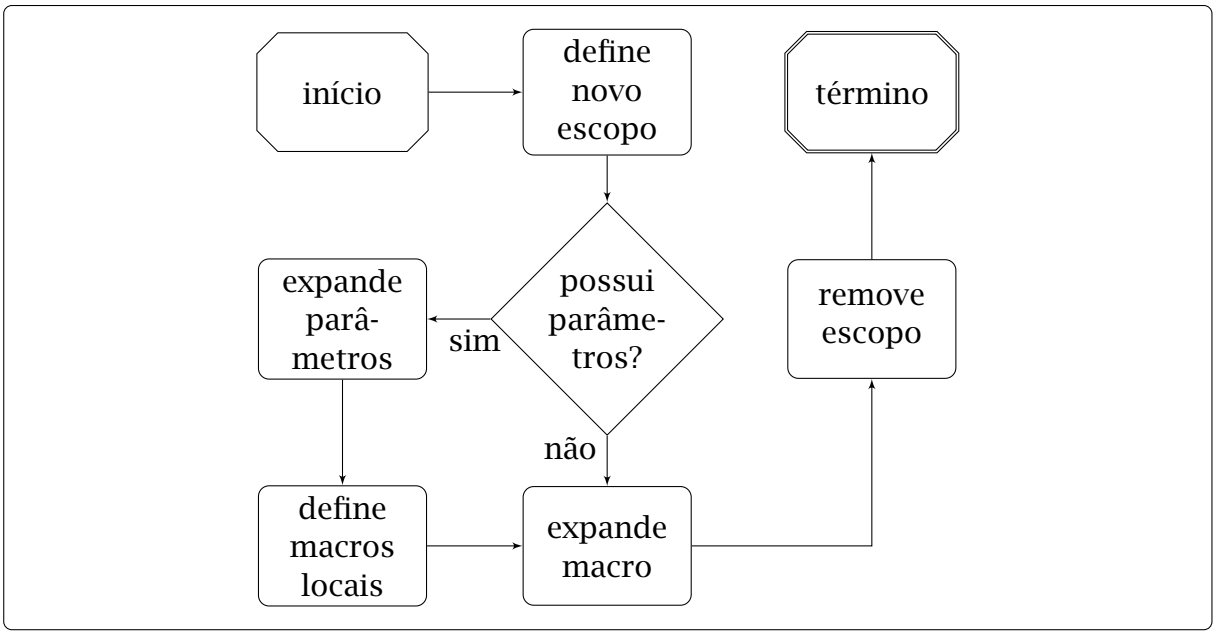
Figura A.30: Tabela de macros estendida com a inclusão de escopo. A política de resolução de nomes prioriza definições locais.



Fonte: autor.

local na fase de expansão.

Figura A.31: Fluxograma das etapas da resolução de parâmetros como macros de escopo local na fase de expansão.



Fonte: autor.

De acordo com o fluxograma da Figura A.31, um novo escopo local é definido para cada instância de macro a ser expandida, e é removido logo após o término da expansão corrente. Eventualmente, macros locais (representando valores de parâmetros da instância de macro) podem ser definidas. Tal política de gerenciamento de escopo permite que definições locais em macros externas (incluindo parâmetros potenciais representados como macros locais) sejam preservadas durante eventuais expansões de macros internas.

Sejam $\tau\colon \Sigma^* \times \mathbb{N} \mapsto \Sigma^*$ uma função que associa um nome de macro e um número natural a um nome de parâmetro, TM a tabela de macros (estendida), e tm_i o conjunto de definições de macros no escopo i, tal que $tm_i = E \mid (i, E) \in TM$. Considere uma nova versão da função de reescrita de termos μ (Equação A.6), definida na Equação A.14, estendida para incluir o gerenciamento de escopo proposto (incluindo a representação de parâmetros como macros locais). Observe que o motor de eventos ME_3 da Figura 3.23 (compartilhado pelos expansores de macros das Seções 3.4, A.1 e A.2) permanece inalterado.

$$\begin{aligned} \mu(x) = y \mid E = \{(\tau(value(x), \beta), \alpha) \mid (\alpha, \beta) \in pars(x)\} \\ TM = TM \cup \{(|TM| + 1, E)\} \\ (value(x), \alpha) \in tm_i \\ \wedge\ value(x) \rightarrow_{tm_{|TM|}} \ldots \rightarrow_{tm_1} \alpha' \rightarrow_{\bigcup_{j=1}^{|TM|} tm_j} y \\ TM = TM - \{(|TM| + 1, E)\} \end{aligned} \tag{A.14}$$

Considere uma implementação do expansor de macros da Seção A.2 utilizando a estratégia de resolução de parâmetros como macros de escopo local com padrão sintático regular, apresentada na Figura A.32 e detalhada a seguir. A função `process`, tal qual a implementação original apresentada na Figura A.24, recebe uma cadeia de símbolos ASCII e uma tabela contendo as definições de macros como parâmetros (`s` e `macros`, a saber), sintetizando as fases de análise léxica e processamento de *tokens*, e retorna uma cadeia de símbolos sem ocorrências de instâncias de macros. Observe que, neste caso, um novo escopo é inserido na tabela de macros e eventuais parâmetros têm seus valores devidamente expandidos e representados como definições de macros locais, conforme ilustrado no fluxograma da Figura A.31. Por questões de simplicidade de implementação, a função `process` utiliza o primeiro e último níveis de escopo como local e global, respectivamente, ao contrário da convenção utilizada na Figura A.30 (ambas são equivalentes). Ao término da expansão, o escopo local é removido da tabela de macros e a cadeia resultante é então retornada.

Exemplo 26 (execução do expansor de macros da Figura A.32)**.** A Figura A.33 apresenta um exemplo da execução do expansor de macros da Figura A.32 (função `process`) com o texto e tabela de macros da Figura A.29 (representados pelas variáveis `s` e `macros`, respectivamente), imprimindo a cadeia de símbolos resultante no terminal de comando. □

Conforme ilustra a Figura A.33, a ocorrência de `greeting` (com o parâmetro `day`) foi substituída por `Good day` na cadeia de símbolos de saída. A estratégia de resolução de parâmetros como macros de escopo local constitui uma alternativa viável à utilização de *placeholders* em sequências de substituição, dado que a representação

Figura A.32: Implementação do expansor de macros da Seção A.2 utilizando a estratégia de resolução de parâmetros como macros de escopo local.

```
local function process(s, macros)
  local result = ""
  while #s ~= 0 do
    local a, b = take(s)
    s = b
    local word = a.value.value
    if a.class == "macro" then
      local macro
      for _, level in ipairs(macros) do
        if level[word] then macro = level[word]; break; end
      end
      if macro then
        table.insert(macros, 1, {})
        local parameters = a.value.parameters
        if #macro.parameters ~= #parameters then
          print("Invalid parameters."); os.exit(0)
        end
        for k, v in ipairs(macro.parameters) do
          macros[1][v] = { value = process(parameters[k], macros),
              parameters = {} }
        end
        result = result .. process(macro.value, macros)
        table.remove(macros, 1)
      else
        print("Undefined macro."); os.exit(0)
      end
    else
      result = result .. word
    end
  end
  return result
end
```

Fonte: autor.

Figura A.33: Execução do expansor de macros da Figura A.32 com o texto e tabela de macros da Figura A.29, imprimindo a cadeia de símbolos resultante no terminal de comando.

Código-fonte Lua:

```
local macros = {
{ greeting = {
   value = "Good \\xtimex\\",
   parameters = { "time" }
  } }
}

local s = "\\xgreeting" ..
   (\\zdayz\\)x\\ to you!"
print(process(s, macros))
```

Resultado da execução:

```
$ lua ex8.lua

Good day to you!
```

Fonte: autor.

de parâmetros através de nomes oferece uma estrutura de trabalho mais conveniente ao usuário.

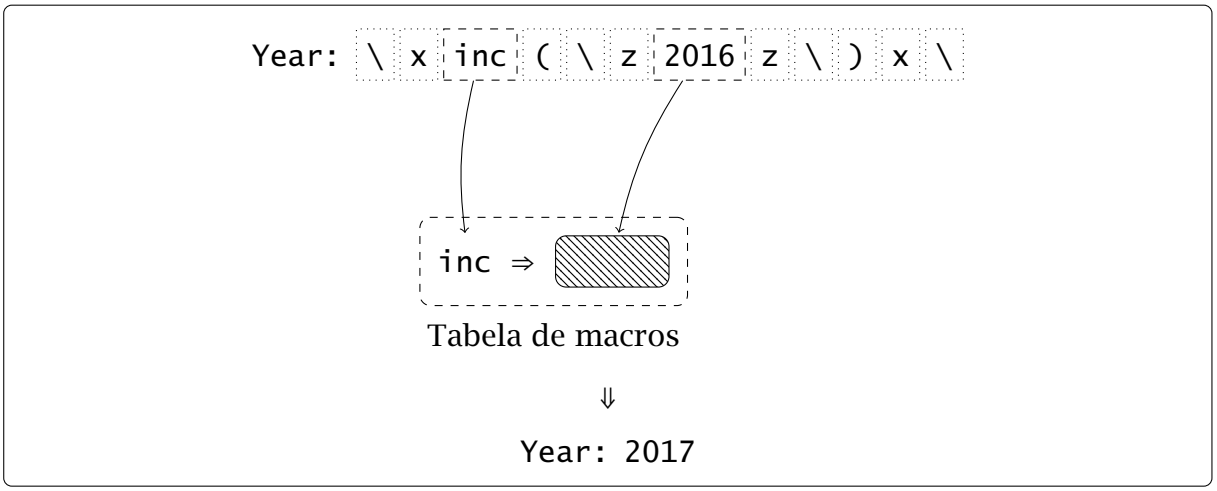
A.3.3 PRIMITIVAS E TRANSFORMAÇÕES ALGORÍTMICAS

Os expansores de macros apresentados nas Seções 3.4, A.1 e A.2 não permitem a definição de macros recursivas (diretas ou indiretas) dada a ausência de construtos que determinem uma condição de parada (por exemplo, através de operações aritméticas e estruturas condicionais). Por conseguinte, somente transformações puramente simbólicas entre sequências são disponibilizadas (Definição 42). A Figura A.34 apresenta um exemplo de expansão de macros com transformações algorítmicas em um texto, utilizando o expansor de macros da Seção A.2 como base.

Conforme ilustra a Figura A.34, a ocorrência da instância de macro `inc` (com o parâmetro `2016`) é substituída por `2017` na cadeia de símbolos resultante. Observe que a macro `inc` disponibilizou uma transformação algorítmica (Definição 43), incrementando o valor do parâmetro obtido (a saber, `2016`) em uma unidade. Em particular, tal macro pode ser considerada um construto elementar (ou primitiva, de acordo com a Definição 44) do expansor de macros proposto (de fato, tal construto compõe o conjunto não-exaustivo de primitivas da Seção 4.1). A Figura A.35 apresenta o fluxograma contendo as etapas de expansão da Seção A.2 com suporte a primitivas.

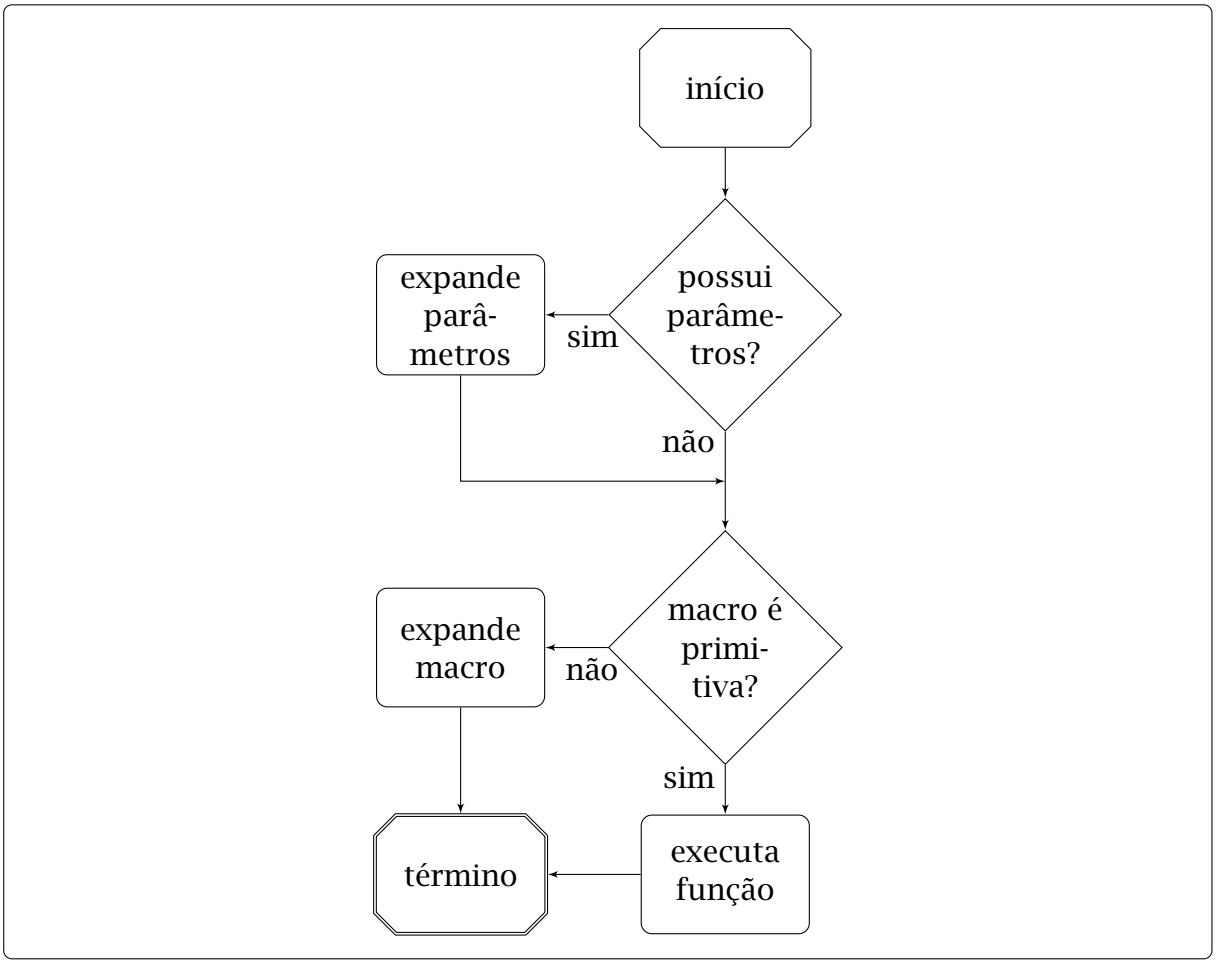
De acordo com o fluxograma da Figura A.35, uma primitiva p pode ser interpretada como uma função computacional aplicada sobre uma sequência de parâmetros (em suas formas normais) $cp = \langle e_1, \ldots, e_n \rangle$, na qual transformações algorítmicas

Figura A.34: Exemplo de expansão de macros com transformações algorítmicas em um texto.



Fonte: autor.

Figura A.35: Fluxograma das etapas de expansão de macros com suporte a primitivas.



Fonte: autor.

são disponibilizadas, retornando um valor v qualquer, tal que $p(cp) = v$. Seja *primitive*: $\Sigma^* \mapsto \{\text{true}, \text{false}\}$ uma função que recebe um nome de macro e verifica se este é uma primitiva do expansor de macros, retornando um valor lógico correspondente. Uma nova versão da função de reescrita de termos μ da Equação A.6 é apresentada a seguir, na Equação A.15, estendida para incluir suporte a primitivas e, por conseguinte, macros com transformações algorítmicas fundamentadas em tais construtos elementares, como corolário. É importante destacar que o motor de eventos ME_3 (Figura 3.23) permanece inalterado.

$$\begin{aligned}
\beta_1(x) &= y \mid (value(x), \alpha) \in TM \\
&\quad \wedge\ value(x) \rightarrow_{TM} \alpha \rightarrow^*_{pars(x)} \alpha' \rightarrow^*_{TM} y \\
\beta_2(x) &= \alpha(pars(x)) \mid (value(x), \alpha) \in TM \\
\mu(x) &= \begin{cases} \beta_2(x) & \text{se } primitive(x) \\ \beta_1(x) & \text{caso contrário} \end{cases}
\end{aligned} \tag{A.15}$$

Considere uma implementação do expansor de macros da Seção A.2, estendido para incluir suporte a transformações algorítmicas, apresentada na Figura A.36 e detalhada a seguir. A função `process`, tal qual a implementação original apresentada na Figura A.24, recebe uma cadeia de símbolos ASCII e uma tabela contendo as definições de macros como parâmetros (`s` e `macros`, a saber), sintetizando as fases de análise léxica e processamento de *tokens*, e retorna uma cadeia de símbolos sem ocorrências de instâncias de macros. Observe que, neste caso, existe uma verificação adicional quanto ao tipo da instância de macro corrente (representadas pela função nativa `type` e pela variável `macro`), de acordo com as etapas descritas no fluxograma da Figura A.35. Caso a instância de macro corrente seja uma função (especificada na linguagem Lua) denotando uma primitiva, o expansor a executa, informando eventuais valores de parâmetros associadas ao *token* correspondente e a tabela de macros. O resultado retornado pela aplicação da função é então concatenado na cadeia de saída. É importante destacar que, na linguagem Lua, funções compreendem um dos oito tipos básicos e são consideradas valores de primeira classe (do inglês *first-class value*, isto é, funções admitem manipulação direta como qualquer outro valor) (IERUSALIMSCHY, 2016; JUNG; BROWN, 2007).

Exemplo 27 (execução do expansor de macros da Figura A.36)**.** A Figura A.37 apresenta um exemplo da execução do expansor de macros da Figura A.36 (função `process`) com o texto e tabela de macros da Figura A.34 (representados pelas variáveis `s` e `macros`, respectivamente), imprimindo a cadeia de símbolos resultante no terminal de comando. Observe que a primitiva `inc` é representada como uma função na linguagem Lua. □

Figura A.36: Implementação do expansor de macros da Seção A.2, estendido para incluir suporte a transformações algorítmicas.

```
local function process(s, macros)
  local result = ""
  while #s ~= 0 do
    local a, b = take(s)
    s = b
    local word = a.value.value
    if a.class == "macro" then
      if macros[word] then
        local macro = macros[word]
        local parameters = a.value.parameters
        if type(macro) == "string" then
          if #parameters > 0 then
            for i, j in ipairs(parameters) do
              macro = string.gsub(macro, "#" .. tostring(i),
                 process(j, macros))
            end
          end
          result = result .. process(macro, macros)
        else
          result = result .. macro(parameters, macros)
        end
      else
        print("Undefined macro."); os.exit(0)
      end
    else
      result = result .. word
    end
  end
  return result
end
```

Fonte: autor.

Figura A.37: Execução do expansor de macros da Figura A.36 com o texto e tabela de macros da Figura A.34, imprimindo a cadeia de símbolos resultante no terminal de comando.

Código-fonte Lua:

```
local macros = {}
macros.inc = function(parameters, m)
  local r = normalize(process(parameters[1], m))
  return tostring(math.floor(r + 1))
end

local s = "Year: \\xinc(\\z2016z\\)x\\"

print(process(s, macros))
```

Resultado da execução:

```
$ lua ex9.lua

Year: 2017
```

Fonte: autor.

Conforme ilustra a Figura A.37, a ocorrência da macro `inc` (com o parâmetro `2016`) foi substituída por `2017` na cadeia de símbolos de saída. A inclusão de construtos elementares em um expansor de macros permite que transformações simbólicas e algorítmicas ocorram de forma concorrente. A partir de um conjunto suficiente de primitivas (por exemplo, inspirado nas recomendações da Seção 4.1), macros mais elaboradas podem ser definidas (TURNER, 1994; LEAVENWORTH, 1966), como evidenciam os exemplos de uso da Seção 4.2. É importante destacar que a inclusão de transformações algorítmicas elimina a propriedade noetheriana do expansor de macros proposto (KOHLBECKER, 1986; BRABRAND; SCHWARTZBACH, 2002).

A.4 IMPLEMENTAÇÃO DO CONJUNTO DE PRIMITIVAS

Esta seção apresenta uma possível implementação para o conjunto de primitivas proposto na Seção 4.1, utilizando a linguagem Lua. É importante destacar que as funções descritas a seguir são independentes do padrão sintático adotado para representação de macros; portanto, é possível utilizar tais implementações nos expansores de macros das Seções 3.4, A.1 e A.2 sem modificações significativas. Os parâmetros de tais funções (variáveis `a` e `b`) denotam uma lista, potencialmente vazia, dos valores não expandidos dos parâmetros e a tabela de macros do expansor,

respectivamente.

A Figura A.38 apresenta a implementação das primitivas de gerenciamento da tabela de macros (Definição 62). Observe que as funções normalizam os nomes das macros (através da função auxiliar `normalize`, introduzida na Figura A.19) e realizam as modificações na tabela global de macros (variável `macros`). No caso da primitiva `DEF`, a definição da nova macro não é expandida antes de sua inserção na tabela; portanto, eventuais inconsistências serão identificadas tão somente em tempo de expansão (Definição 54). As primitivas de gerenciamento da tabela de macros atuam no metanível do expansor e não produzem transformações na cadeia de saída, retornando, portanto, ϵ (representado pela cadeia vazia `""` nas instruções de retorno correspondentes).

Figura A.38: Implementação das primitivas de gerenciamento da tabela de macros (Definição 62).

```
macros["DEF"] = function(a, b)
  local name = normalize(a[1])
  b[name] = a[2]
  return ""
end

macros["UNDEF"] = function(a, b)
  local name = normalize(a[1])
  b[name] = nil
  return ""
end
```

Fonte: autor.

A Figura A.39 apresenta a implementação das primitivas de incremento e decremento de valores inteiros (Definição 63). Observe que as funções expandem e normalizam o valor do parâmetro (posição 1 da lista), armazenando-o em uma variável local (a saber, `r`). Em seguida, tal valor é adicionado (ou subtraído) de uma unidade e retornado após a aplicação de uma conversão textual através da função `tostring`, disponibilizada nativamente na linguagem Lua.

A implementação das primitivas de controle de fluxo (Definição 64) é apresentada na Figura A.40. Observe que as funções expandem e normalizam o valor do primeiro parâmetro (variável `a[1]`, denotando a condição a ser testada), armazenando-o em uma variável local (variável `c`). Em seguida, o teste condicional de igualdade é efetivamente aplicado (`'T'` para `IF` e `'F'` para `UNLESS`) e o bloco correspondente ao resultado da avaliação (a saber, `a[2]` em caso positivo, ou `a[3]` caso contrário) é devidamente expandido e retornado, conforme ilustra a Figura 4.3.

Figura A.39: Implementação das primitivas de incremento e decremento de valores inteiros (Definição 63).

```
macros["INC"] = function(a, b)
  local r = normalize(process(a[1], b))
  return tostring(math.floor(r + 1))
end

macros["DEC"] = function(a, b)
  local r = normalize(process(a[1], b))
  return tostring(math.floor(r - 1))
end
```

Fonte: autor.

Figura A.40: Implementação das primitivas de controle de fluxo (Definição 64).

```
macros["IF"] = function(a, b)
  local c = normalize(process(a[1], b))
  if c == 'T' then return process(a[2], b)
  else return process(a[3], b) end
end

macros["UNLESS"] = function(a, b)
  local c = normalize(process(a[1], b))
  if c == 'F' then return process(a[2], b)
  else return process(a[3], b) end
end
```

Fonte: autor.

A Figura A.41 apresenta a implementação das primitivas de comparação entre valores numéricos (Definição 65). As funções expandem e normalizam os valores dos dois parâmetros (variáveis `a[1]` e `a[2]`), armazenando-os em duas variáveis locais (variáveis `r` e `w`). Em seguida, tais valores são comparados e o resultado lógico correspondente é retornado, em formato textual (`'T'` em caso positivo, ou `'F'` caso contrário).

A implementação das primitivas de operações aritméticas (Definição 66) é apresentada na Figura A.42. As funções expandem e normalizam os valores dos dois parâmetros (variáveis `a[1]` e `a[2]`), armazenando-os em duas variáveis locais (variáveis `r` e `w`). Em seguida, a operação aritmética correspondente é aplicada a tais valores (neste caso, soma, subtração, multiplicação, divisão ou módulo) e o resultado é retornado após a aplicação de uma conversão textual.

A Figura A.43 apresenta a implementação das primitivas de operações lógicas

Figura A.41: Implementação das primitivas de comparação entre valores numéricos (Definição 65).

```
macros["GREATER"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  r, w = math.floor(r), math.floor(w)
  if r > w then return 'T' else return 'F' end
end

macros["LESS"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  r, w = math.floor(r), math.floor(w)
  if r < w then return 'T' else return 'F' end
end

macros["EQUAL"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  if r == w then return 'T' else return 'F' end
end
```

Fonte: autor.

(Definição 67). No caso das primitivas `AND` e `OR`, as funções expandem e normalizam os valores dos dois parâmetros (variáveis `a[1]` e `a[2]`), armazenando-os em duas variáveis locais (`r` e `w`). Em seguida, a operação lógica correspondente é aplicada a tais valores (neste caso, conjunção ou disjunção) e o resultado é retornado, em formato textual (`'T'` em caso positivo, ou `'F'` caso contrário). No caso da primitiva `NOT`, a função expande e normaliza o valor do parâmetro (posição 1 da lista), armazenando-o em uma variável local (a saber, `r`). Em seguida, a operação lógica de negação é aplicada a tal valor e o resultado é retornado, em formato textual.

A implementação das primitivas de repetição (Definição 68) é apresentada na Figura A.44. No caso da primitiva `WHILE`, a função expande e normaliza o valor do primeiro parâmetro (a saber, `a[1]`, denotando a condição a ser testada), armazenando-o em uma variável local (a saber, `r`). Em seguida, o teste condicional de igualdade é efetivamente aplicado e, em caso positivo, o valor do segundo parâmetro (a saber, `a[2]`) é expandido, seguido de um novo teste (devidamente expandido e normalizado). Tal sequência de operações repete-se até que o resultado do teste condicional seja falso (denotado pelo valor `'F'`), retornando a concatenação das expansões executadas. No caso da primitiva `REPEAT`, a função expande e normaliza o valor do primeiro parâmetro (a saber, `a[1]`, denotando o número de repetições), armazenando-o em

Figura A.42: Implementação das primitivas de operações aritméticas (Definição 66).

```
macros["ADD"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  return tostring(math.floor(r + w))
end

macros["SUB"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  return tostring(math.floor(r - w))
end

macros["MULT"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  return tostring (math.floor(r * w))
end

macros["DIV"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  return tostring(math.floor(r / w))
end

macros["MOD"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  return tostring(math.floor(r % w))
end
```

Fonte: autor.

uma variável local (a saber, `r`). Em seguida, o valor do segundo parâmetro (a saber, `a[2]`) é expandido. Tal expansão é repetida até que um contador interno (iniciado em uma unidade) seja igual ao número de repetições especificado em `r`. Finalmente, a concatenação das expansões executadas é retornada.

A Figura A.45 apresenta a implementação das primitivas de gerenciamento de símbolos (Definição 69). No caso da primitiva `SET`, a função expande e normaliza o valor do primeiro parâmetro (a saber, `a[1]`, denotando o nome do símbolo), armazenando-o em uma variável local (a saber, `r`). Em seguida, o valor associado a tal símbolo (a saber, `a[2]`, denotando o segundo parâmetro) é devidamente expandido e armazenado em uma tabela global de símbolos (variável `symbols`) na posição indicada por seu índice nominal (denotado por `symbols[r]`). Tal primitiva atua no

Figura A.43: Implementação das primitivas de de operações lógicas (Definição 67).

```
macros["AND"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  return (r == 'T') and (w == 'T')
end

macros["OR"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  return ((r == 'T') or (w == 'T')) and 'T' or 'F'
end

macros["NOT"] = function(a, b)
  local r = normalize(process(a[1], b))
  return not (r == 'T')
end
```

Fonte: autor.

Figura A.44: Implementação das primitivas de repetição (Definição 68).

```
macros["WHILE"] = function(a, b)
  local r, w = normalize(process(a[1], b)), ''
  while r == 'T' do
    w = w .. process(a[2], b)
    r = normalize(process(a[1], b))
  end
  return w
end

macros["REPEAT"] = function(a, b)
  local r, w = normalize(process(a[1], b)), ''
  for _ = 1, math.floor(r) do
    w = w .. process(a[2], b)
  end
  return w
end
```

Fonte: autor.

metanível do expansor e não produz transformações na cadeia de saída, retornando, portanto, ϵ (representado pela cadeia vazia "" na instrução de retorno correspondente). No caso da primitiva `GET`, a função expande e normaliza o valor do parâmetro (a saber, `a[1]`, denotando o nome do símbolo), armazenando-o em uma variável local (variável `r`). Em seguida, o valor associado a tal índice nominal na tabela de símbolos (denotado por `symbols[r]`) é retornado. Caso o nome do símbolo inexista na tabela, uma cadeia vazia é retornada.

Figura A.45: Implementação das primitivas de gerenciamento de símbolos (Definição 69).

```
macros["SET"] = function(a, b)
  local r = normalize(process(a[1], b))
  symbols[r] = process(a[2], b)
  return ""
end

macros["GET"] = function(a, b)
  local r = normalize(process(a[1], b))
  return symbols[r] or ""
end
```

Fonte: autor.

A implementação da primitiva de reprodução literal (Definição 70) é apresentada na Figura A.46. Observe que a função simplesmente retorna o valor literal do parâmetro (a saber, `a[1]`), sem normalização ou expansão. É importante destacar que a variável `_` representa um símbolo descartável na linguagem Lua (do inglês *throwaway variable*), evitando, assim, a alocação desnecessária de memória (JUNG; BROWN, 2007; IERUSALIMSCHY, 2016). Dado que a tabela de macros não é referenciada no corpo da função, esta pode ser substituída por uma variável de descarte.

Figura A.46: Implementação da primitiva de reprodução literal (Definição 70).

```
macros["LIT"] = function(a, _)
  return a[1]
end
```

Fonte: autor.

A Figura A.47 apresenta a implementação da primitiva de estratégia de expansão (Definição 71). A função normaliza os valores dos dois parâmetros (variáveis `a[1]` e `a[2]`), armazenando-os em duas variáveis locais (variáveis `r` e `w`). Em seguida, a

representação inteira de `w` é armazenada em uma tabela global de estratégias (a saber, `strategy`) na posição indicada pelo nome da macro (variável `r`). Tal primitiva atua no metanível do expansor e não produz transformações na cadeia de saída, retornando, portanto, ϵ (representado pela cadeia vazia "" na instrução de retorno correspondente). A tabela de estratégias deve ser contemplada no expansor de macros, tal que este monitore o número efetivo de chamadas das macros presentes em tal tabela (aplicações sucessivas da estratégia de expansão de um passo, conforme a Definição 60), e ignore as macros ausentes (aplicando, neste caso, a estratégia de expansão de múltiplos passos, conforme a Definição 61). Em linhas gerais, a função de expansão `process` pode ser estendida para oferecer suporte a tais estratégias através da inclusão de um construto condicional adicional para verificação do *token* corrente potencialmente eligível para expansão de acordo com as restrições determinadas na tabela global de estratégias.

Figura A.47: Implementação da primitiva de estratégia de expansão (Definição 71).

```
macros["EXS"] = function(a, _)
  local r, w = normalize(a[1]),
               normalize(a[2])
  strategy[r] = math.floor(w)
  return ""
end
```

Fonte: autor.

A implementação da primitiva de projeção (Definição 72) é apresentada na Figura A.48. A função normaliza e expande os valores dos dois parâmetros (variáveis `a[1]` e `a[2]`), armazenando-os em duas variáveis locais (`r` e `w`). Em seguida, o valor de `w` é convertido em uma lista de elementos no formato serial CSV (através da função auxiliar `split` utilizando o símbolo `,` como separador). Finalmente, o valor inteiro indexado pela posição `r`, devidamente normalizado, é retornado. No caso de um índice fora do intervalo (incluindo a ocorrência de uma lista vazia), ϵ é retornado (representado pela cadeia vazia "" na operação lógica de disjunção correspondente).

A Figura A.49 apresenta uma possível implementação das primitivas de adaptatividade (Definição 73). As funções que implementam as primitivas `INSERT` e `QUERY` potencialmente expandem e normalizam os valores dos dois parâmetros (variáveis `a[1]` e `a[2]`), armazenando-os em duas variáveis locais (variáveis `r` e `w`). No caso da primitiva `INSERT`, o valor de `w` é inserido na posição seguinte aos pontos léxicos de ocorrência do padrão sintático denotado por `r` no texto sendo expandido (denotado pela variável global `text`). No caso da primitiva `QUERY`, os pontos léxicos de ocorrência do padrão sintático denotado por `r` no texto sendo expandido são armazenados

Figura A.48: Implementação da primitiva de projeção (Definição 72).

```
local function split(t, s)
  local f = {}
  local p = string.format("([^%s]+)", s)
  string.gsub(t, p, function(c) f[#f + 1] = c end)
  return f
end

macros["PRJ"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
               normalize(process(a[2], b))
  return normalize(split(w, ',')[math.floor(r)]) or ""
end
```

Fonte: autor.

em uma lista auxiliar (a saber, `g`). Em seguida, tal lista é armazenada em uma tabela global de símbolos (a saber, `symbols`) na posição indicada pelo nome do símbolo (a saber, `symbols[w]`), conforme ilustra a Figura 4.14. A função que implementa a primitiva `REMOVE` potencialmente expande e normaliza o valor do parâmetro (a saber, `a[1]`), armazenando-o em uma variável temporária (a saber, `r`). Em seguida, as ocorrências do padrão sintático denotado por `r` são removidas do texto sendo expandido. As primitivas de adaptatividade atuam no metanível do expansor e não produzem transformações na cadeia de saída, retornando, portanto, ϵ (representado pela cadeia vazia "" nas instruções de retorno correspondentes).

De acordo com a Figura A.49, as funções que implementam as primitivas `INSERT` e `REMOVE` podem, alternativamente, interpretar listas de ponteiros referentes às posições léxicas de ocorrência de padrões sintáticos. Tais listas especiais resultam de consultas provenientes da primitiva `QUERY`, armazenadas em uma tabela global de símbolos. Observe, entretanto, a indisponibilidade de uso direto de listas de ponteiros; estas estão apenas acessíveis para consulta através da primitiva `GET` como suporte às primitivas de adaptatividade.

É importante destacar que a implementação proposta para as primitivas de adaptatividade (Figura A.49) adota uma política de visibilidade global, discutida na Seção 5.1. Para tal, observe que a variável `text` é definida no escopo global, suscitando potenciais modificações no projeto do expansor de macros (em particular, na função de expansão `process`). É possível utilizar certos padrões de projeto (do inglês *design patterns*) para auxiliar no desenvolvimento de tal expansor.

Efeitos colaterais resultantes de ações adaptativas incluem alterações do ponto léxico de análise corrente (e eventuais posições de retorno em expansões aninhadas),

Figura A.49: Implementação das primitivas de adaptatividade (Definição 73).

```
macros["INSERT"] = function(a, b)
  local w = normalize(process(a[2], b))
  if type(a[1]) == "table" then
    local t, i = '', 1
    for _, v in ipairs(a[1]) do
    t = t .. string.sub(text, i, v[2]) .. w ; i = v[2] + 1
  end
  text = t .. string.sub(text, i)
  else
    local r = normalize(process(a[1], b)),
    text = string.gsub(text, string.format("(%s)", r),
         string.format("%%1%s", w))
  end
  return ""
end

macros["REMOVE"] = function(a, b)
  if type(a[1]) == "table" then
    local t, i = '', 1
    for _, v in ipairs(a[1]) do
      t = t .. string.sub(text, i, v[1] - 1) ; i = v[2] + 1
    end
    text = t .. string.sub(text, i)
  else
    local r = normalize(process(a[1], b))
    text = string.gsub(text, r, "")
  end
  return ""
end

macros["QUERY"] = function(a, b)
  local r, w = normalize(process(a[1], b)),
           normalize(process(a[2], b))
  local c, f, g, d, e = r, 0, {}, string.find(c, b)
  while d do
    table.insert(g, { d + f, e + f }) ; f = f + e
    c = string.sub(c, e + 1) ; d, e = string.find(c, b)
  end
  table.insert(symbols[w], g)
  return ""
end
```

Fonte: autor.

invalidação de ponteiros de padrões sintáticos na tabela global de símbolos e ocorrências de macros remanescentes introduzidas *a posteriori*. Tais efeitos constituem desafios para implementação de construtos adaptativos em expansores de macros. Uma possível resolução envolve a utilização de referências sincronizadas entre si, de modo que eventuais modificações causadas por ações adaptativas no texto corrente sejam consistentemente propagadas no espaço de referências. Entretanto, observe que tal iniciativa introduz uma complexidade significativa na implementação. Consequentemente, estudos adicionais são necessários para avaliar a exequibilidade do uso de referências sincronizadas.

APÊNDICE B

UMA PROPOSTA DE EXTENSÃO PARA A NOTAÇÃO DE WIRTH

> It is impossible to be a mathematician without being a poet in soul.
>
> SOFIA KOVALEVSKAYA

O método de desenvolvimento de um expansor de macros através de uma estratificação em camadas representando níveis de abstração (Seção 3.3, página 62) proporciona o particionamento do problema original em subproblemas menores. As primeiras camadas, em geral, realizam a extração e classificação de elementos sintáticos do texto. Este apêndice apresenta uma proposta de extensão da notação de Wirth tradicional (Seção 2.1) para inclusão de metadados descritivos em símbolos terminais e não-terminais, incluindo discussões sobre a geração automática de autômatos com estados anotados para análise léxica.

B.1 ASPECTOS DA METALINGUAGEM

Um estudo preliminar acerca da inclusão de metadados descritivos em especificações de dispositivos reconhecedores foi realizado por Cereda e José Neto (2017a), com o propósito de incorporar pontos de instrumentação para análise de desempenho. Inspirado em tal trabalho, propõe-se uma extensão sintática à metalinguagem definida por Wirth (Seção 2.1) para incluir anotações em terminais e não-terminais. A Figura B.1 apresenta tal proposta.

De acordo com a Figura B.1, a notação estendida introduz um novo metassímbolo opcional de etiquetagem: `^`. Tal metassímbolo é previsto imediatamente após a ocorrência de terminais e não-terminais, e precede um identificador de etiquetagem a ser vinculado a tais símbolos. Como exemplo, a expressão `a^b` denota a associação do identificador `b` com o símbolo não-terminal `a`.

O identificador de etiquetagem pode atuar como ponteiro para uma tabela contendo metadados descritivos. No exemplo, `b` permite referenciar ações semânticas ou definir algumas dependências para tratamento posterior (dado que a linguagem de Wirth descreve gramáticas livres de contexto (WIRTH, 1977)).

Figura B.1: Notação de Wirth estendida para inclusão de metadados descritivos, expressa utilizando a notação original.

```
syntax = { production } .
production = identifier "=" expression "." .
expression = term { "|" term } .
term = factor { factor } .
factor = identifier [ "^" letter { letter | digit } ] |
         literal [ "^" letter { letter | digit } ] |
         "(" expression ")" | "[" expression "]" |
         "{" expression "}" .
literal = """" character { character } """" .
```

Fonte: autor, adaptado de Wirth (1977).

B.2 ANÁLISE LÉXICA

O analisador léxico proposto para a notação de Wirth estendida (Figura B.1) foi projetado para categorizar uma sequência de símbolos em uma sequência de *tokens* de acordo com as classes gramaticais apresentadas na Tabela B.1.

Tabela B.1: Classes gramaticais do analisador léxico proposto para a notação de Wirth estendida (Figura B.1).

Classe	Rótulo	Significado
c_1	$\sigma \in \Sigma$	metassímbolo
c_2	*nonterm*	não-terminal
c_3	*terminal*	símbolo terminal
c_4	*tagged*	c_2 ou c_3 etiquetado

Fonte: autor.

A Figura B.2 apresenta um autômato finito determinístico W que descreve o analisador léxico proposto. A Tabela B.2 caracteriza os símbolos pertencentes às transições do autômato, no qual $\sigma \in \Sigma$, S, L e D denotam o símbolo corrente e os conjuntos de símbolos, letras e dígitos, respectivamente. Estados anotados com c_1 a c_4 indicam as classes gramaticais associadas.

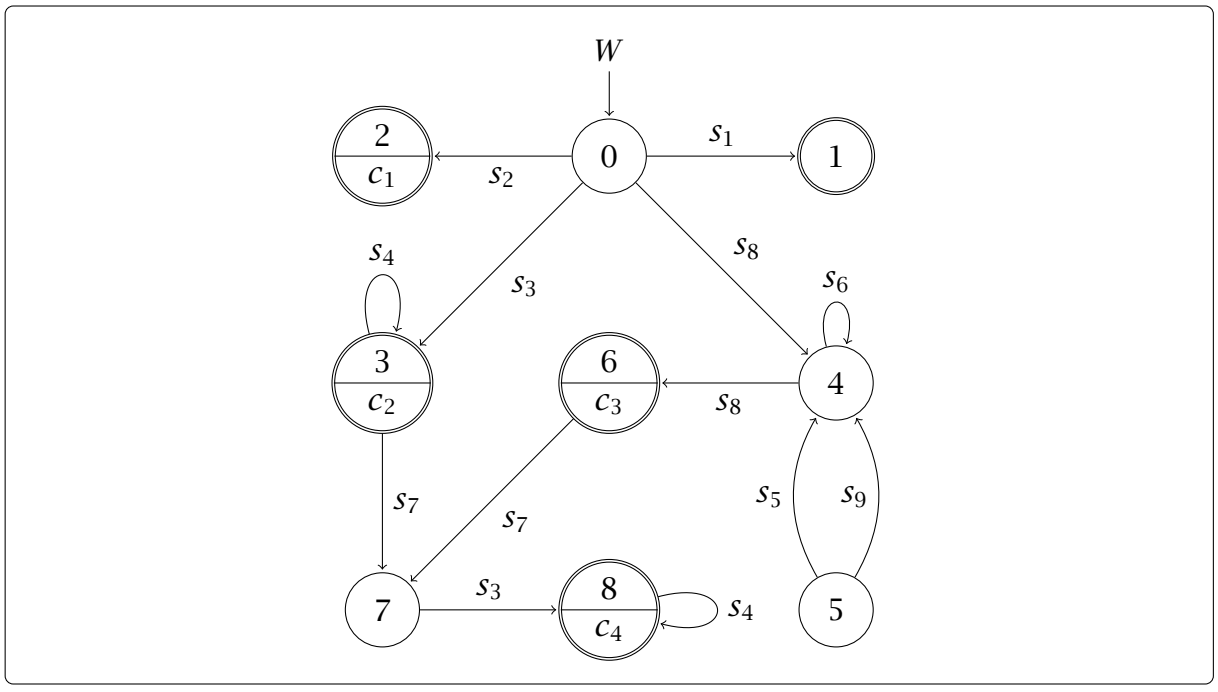
Sejam $w \in \Sigma^*$ uma cadeia, tal que $w \in L(W)$, e $C\colon \Sigma^* \mapsto \{c_1, c_2, c_3, c_4\}$ uma função que mapeia cadeias de Σ em classes gramaticais c_1 a c_4. A cadeia w será classificada como metassímbolo (classe gramatical c_1) se, e somente se, o autômato finito W, após o término do reconhecimento de w, referenciar 2 como estado corrente, $C(w) = c_2 \iff (0, w) \vdash^* (2, \epsilon)$. Analogamente, a cadeia w será classificada como um não-terminal (classe gramatical c_2), terminal (classe gramatical c_3) ou símbolo etiquetado (classe gramatical c_4) se, e somente se, após o término do reconhecimento de w, o autômato finito W referenciar 3, 6 ou 8 como estado corrente,

Tabela B.2: Descrição dos símbolos pertencentes às transições do autômato finito determinístico W da Figura B.2, no qual $\sigma \in \Sigma$, S, L e D denotam o símbolo corrente e os conjuntos de símbolos, letras e dígitos, respectivamente.

Rótulo	Descrição
s_1	$\sigma \in \{\text{␣}, \texttt{\textbackslash t}, \texttt{\textbackslash n}\}$
s_2	$\sigma \in S - \{\texttt{"}\}$
s_3	$\sigma \in L$
s_4	$\sigma \in L \cup D$
s_5	$\sigma \in L \cup D \cup S$
s_6	$\sigma \in L \cup D \cup S - \{\texttt{\textbackslash}, \texttt{"}\}$
s_7	$\sigma = \texttt{\textasciicircum}$
s_8	$\sigma = \texttt{"}$
s_9	$\sigma = \texttt{\textbackslash}$

Fonte: autor.

Figura B.2: Autômato finito determinístico W que descreve o analisador léxico para a notação de Wirth estendida (Figura B.1).



Fonte: autor.

respectivamente, $C(w) = c_2 \iff (0, w) \vdash^* (3, \epsilon)$, $C(w) = c_3 \iff (0, w) \vdash^* (6, \epsilon)$ e $C(w) = c_4 \iff (0, w) \vdash^* (8, \epsilon)$. Operacionalmente, um *token* etiquetado preserva sua classe gramatical anterior, com a inclusão do identificador capturado; a classe gramatical c_4, neste caso, é utilizada tão somente para fins de organização do modelo conceitual.

Considere uma implementação do analisador léxico proposto utilizando a linguagem Lua, detalhada na Figura B.3. Tal implementação utiliza as funções auxiliares `head` e `tail` definidas na Figura 3.18 (página 71). É importante destacar que a função `token` foi redefinida para incluir uma tabela de metadados descritivos (referenciada através do índice nominal `data`).

Figura B.3: Implementação do analisador léxico proposto para a notação de Wirth estendida (Figura B.1).

```
local function token(a, b, c)
  return { value = a, class = b, data = c }
end

local function take(s)
  local a, b, c, d, e
  a, b, c = string.find(s, '^%s*"""%s*')
  if a then
    _, a, c = string.find(s, '^%s*"""%^(%a%w*)%s*')
    return token('"', 'terminal', c),
           tail(s, (a or b) + 1)
  else
    a, b, c = string.find(s, '^%s*(%a%w*)%s*')
    if a then
      _, a, e, d = string.find(s, '^%s*(%a%w*)%^(%a%w*)%s*')
      return token((e or c), 'nonterm', d),
             tail(s, (a or b) + 1)
    else
      a, b, c = string.find(s, '^%s*(%b"")%s*')
      if a then
        _, a, e, d = string.find(s, '^%s*(%b"")%^(%a%w*)%s*')
        return token((e or c), 'terminal', d),
               tail(s, (a or b) + 1)
      else
        return token(head(s), head(s)), tail(s, 2)
      end
    end
  end
end
```

Fonte: autor.

A função `take`, conforme ilustra a Figura B.3, efetivamente implementa o analisador léxico proposto; tal função recebe uma cadeia de símbolos como parâmetro (a saber, `s`) e retorna uma tupla contendo o *token* obtido e o restante da cadeia para uso posterior. Observe que o fornecimento de *tokens* ocorre sob demanda, isto é, a função `take` retorna um *token* por vez; chamadas subsequentes da função sobre o restante da cadeia de símbolos (até que esta torne-se vazia) garanter a obtenção da sentença de *tokens* desejada.

Exemplo 28 (execução do analisador léxico da Figura B.3)**.** A Figura B.4 apresenta um exemplo da execução do analisador léxico da Figura B.3 (função `take`) com um determinado texto (representado pela variável `s`) de forma contínua, imprimindo a sequência de *tokens* no terminal de comando. □

Figura B.4: Execução do analisador léxico da Figura B.3, de forma contínua, imprimindo a sequência de *tokens* no terminal de comando.

Código-fonte Lua:

```
local s, a = 'A = id^var "hello" [ "world"^place ].'

while #s ~= 0 do
  a, s = take(s)
  io.write("(" .. a.value .. ", " .. a.class ..
    (a.data and ", " .. a.data or "") .. ") ")
end
```

Resultado da execução:

```
$ lua wi1.lua

(A, nonterm) (=, =) (id, nonterm, var) ("hello", terminal)
([, []) ("world", terminal, place) (], ]) (., .)
```

Fonte: autor.

É importante destacar que a análise léxica apresentada nesta seção também é válida para gramáticas escritas na notação de Wirth tradicional. O autômato finito determínistico W (Figura B.2) e a implementação de um analisador correspondente (Figura B.3) são suficientemente expressivos para uso com a metalinguagem original e sua extensão sintática (Figura B.1).

B.3 GERAÇÃO DE AUTÔMATOS

José Neto (1987) apresenta um autômato de pilha estruturado acrescido de ações semânticas para geração automática de um analisador sintático a partir de uma gramática escrita em notação de Wirth. Tal dispositivo resultante reconhece sentenças válidas da gramática especificada. Dado que a proposta de extensão à notação de Wirth não contempla, do ponto de vista operacional, classes gramaticais adicionais, a técnica de geração de um autômato de pilha estruturado correspondente a partir da sequência de *tokens* obtida na fase de análise léxica (Seção B.2) pode ser utilizada sem modificações significativas. É necessário, entretanto, que as ações semânticas considerem a potencial ocorrência de metadados descritivos em *tokens* durante a construção do autômato correspondente. Uma versão adaptativa de tal técnica é contemplada no trabalho de Cereda e José Neto (2015a).

Preliminarmente, considere a função auxiliar `transition` para representação semântica de uma transição estendida do autômato em construção, apresentada na Figura B.5. Tal função retorna uma tabela representando uma transição estendida e contendo os estados de origem e destino (a saber, índices nominais `from` e `to`), símbolo ou chamada de submáquina (a saber, índice nominal `with`), e eventuais metadados descritivos (a saber, índice nominal `data`). Observe que a ausência do índice nominal `with` denota uma transição em vazio.

Figura B.5: Função auxiliar `transition` para representação semântica de uma transição estendida do autômato sendo construído.

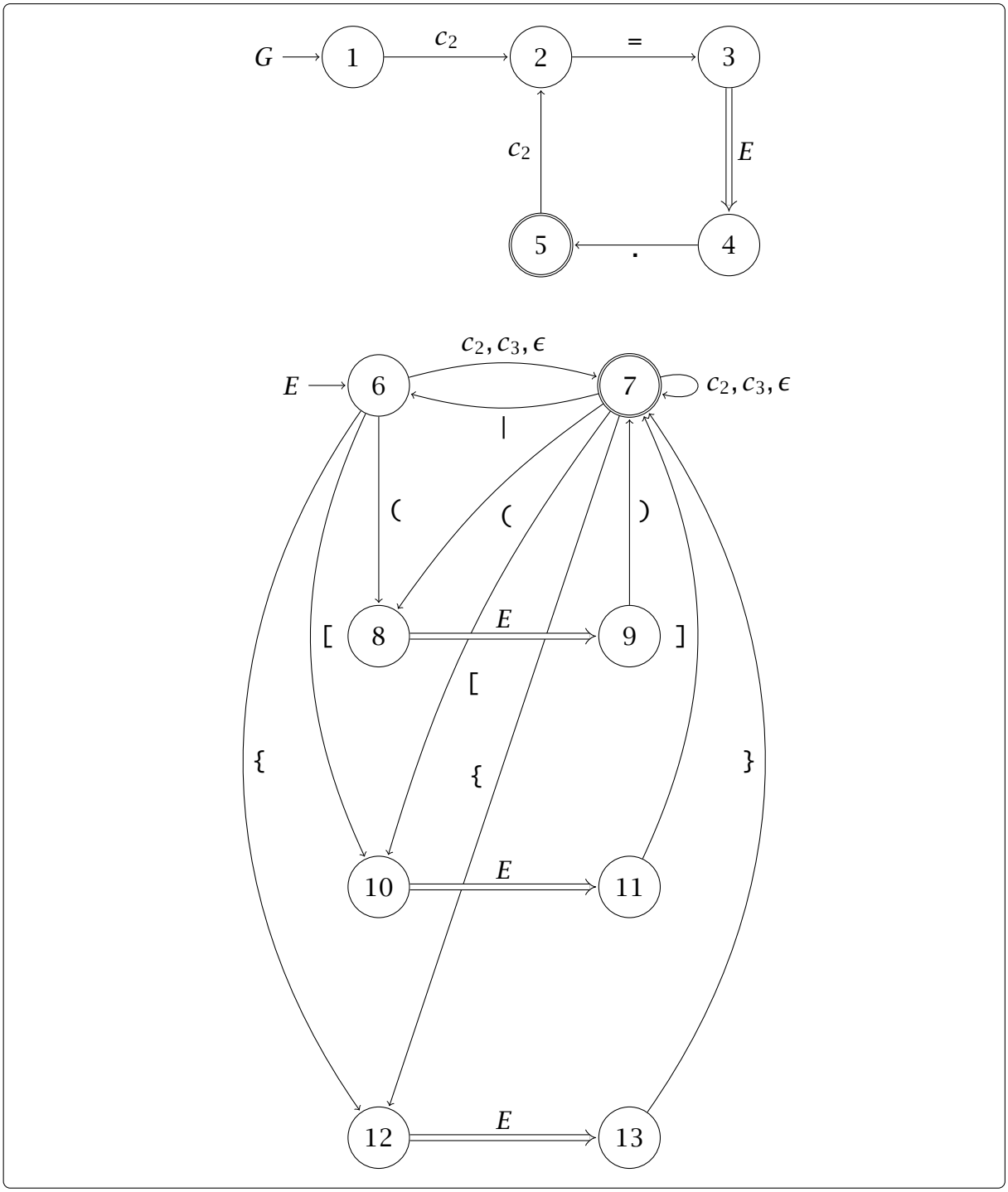
```
local function transition(a, b, c, d)
  return { from = a, to = b, with = c, data = d }
end
```

Fonte: autor.

A Figura B.6 apresenta o autômato de pilha estruturado G para geração de um analisador sintático a partir de uma gramática escrita em notação de Wirth estendida. As ações semânticas associadas às transições de G, representadas como funções na linguagem Lua, são descritas na Tabela B.3. Tais ações compartilham quatro variáveis globais, a saber: `current` e `counter` denotam contadores de estado, `stack` representa uma estrutura de dados em pilha, e `transitions` denota uma tabela de transições do autômato resultante, inicialmente vazia.

Ao término do reconhecimento de uma sequência de *tokens* pelo autômato de pilha estruturado G (Figura B.6), a tabela de transições `transitions` retratará a relação de transição do novo autômato, gerado automaticamente através das ações semânticas descritas na Tabela B.3, incluindo potenciais metadados descritivos as-

Figura B.6: Autômato de pilha estruturado G para geração do analisador sintático a partir de uma gramática escrita em notação de Wirth estendida.



Fonte: autor.

Tabela B.3: Ações semânticas associadas às transições do autômato de pilha estruturado G (Figura B.6), representadas como funções na linguagem Lua.

Transição	Significado	Figura
$(1, c_2) \rightarrow 2$	nova submáquina	B.7
$(2, =) \rightarrow 3$	novo escopo	B.8
$(4, .) \rightarrow 5$	fechamento de escopo	B.9
$(5, c_2) \rightarrow 2$	nova submáquina	B.7
$(6, c_2) \rightarrow 7$	nova transição	B.10
$(6, c_3) \rightarrow 7$	nova transição	B.10
$(6, \epsilon) \rightarrow 7$	nova transição	B.10
$(7, c_2) \rightarrow 7$	nova transição	B.10
$(7, c_3) \rightarrow 7$	nova transição	B.10
$(7, \epsilon) \rightarrow 7$	nova transição	B.10
$(6, () \rightarrow 8$	novo escopo	B.8
$(6, [) \rightarrow 10$	abertura de colchetes	B.11
$(6, \{) \rightarrow 12$	abertura de chaves	B.12
$(7, () \rightarrow 8$	novo escopo	B.8
$(7, [) \rightarrow 10$	abertura de colchetes	B.11
$(7, \{) \rightarrow 12$	abertura de chaves	B.12
$(9,)) \rightarrow 7$	fechamento de escopo	B.9
$(11,]) \rightarrow 7$	fechamento de escopo	B.9
$(13, \}) \rightarrow 7$	fechamento de escopo	B.9
$(7, \vert) \rightarrow 6$	adição de opção	B.13

Fonte: autor.

Figura B.7: Ação semântica de criação de nova submáquina.

```
local function actionB7()
  stack = {}
  current = 0
  counter = 1
end
```

Fonte: autor.

Figura B.8: Ação semântica de novo escopo.

```
local function actionB8()
  table.insert(stack, 1, {current, counter})
  counter = counter + 1
end
```

Fonte: autor.

Figura B.9: Ação semântica de fechamento de escopo.

```
local function actionB9()
  local t = transition(current, stack[1][2])
  table.insert(transitions, t)
  current = stack[1][2]
  table.remove(stack, 1)
end
```

Fonte: autor.

Figura B.10: Ação semântica de nova transição.

```
local function actionB10()
  local t = transition(current, counter, token.value, token.data)
  table.insert(transitions, t)
  current = counter
  counter = counter + 1
end
```

Fonte: autor.

Figura B.11: Ação semântica de abertura de colchetes.

```
local function actionB11()
  local t = transition(current, counter)
  table.insert(transitions, t)
  table.insert(stack, 1, { current, counter })
  counter = counter + 1
end
```

Fonte: autor.

Figura B.12: Ação semântica de abertura de chaves.

```
local function actionB12()
  local t = transition(current, counter)
  table.insert(transitions, t)
  table.insert(stack, 1, { counter, counter })
  counter = counter + 1
end
```

Fonte: autor.

Figura B.13: Ação semântica de adição de opção.

```
local function actionB13()
  local t = transition(current, stack[1][2])
  table.insert(transitions, t)
  current = stack[1][1]
end
```

Fonte: autor.

sociados. Cabe ao usuário definir o tratamento adequado aplicado a tais metadados. Como exemplo, é possível representar metadados descritivos como símbolos de um alfabeto de saída em máquinas de Mealy ou Moore para obtenção de analisadores léxicos em projetos de compiladores.

É importante observar que o autômato resultante é não-determinístico. Sua forma determinística pode ser obtida através da aplicação de algoritmos clássicos de conversão (COOPER; TORCZON, 2014). Adicionamente, é possível otimizar o modelo determinístico resultante através da minimização de estados (HOPCROFT, 1970, 1971). Entretanto, tais algoritmos devem considerar a ocorrência de metadados descritivos, preservando estas informações.

APÊNDICE C

UMA FERRAMENTA PARA GERAÇÃO DE MOTORES DE EVENTOS

> Simplicity is prerequisite for reliability.
>
> EDSGER DIJKSTRA

Os expansores de macros contemplados no escopo desta tese (em particular, no Capítulo 3 e no Apêndice A) utilizam motores de eventos como modelos de implementação. A escolha de tais modelos, amplamente utilizados em sistemas operacionais, evidencia a importância de buscar em outras áreas soluções alternativas, não triviais, para problemas. Este apêndice apresenta a ferramenta `eventengine` para a geração de motores de eventos, utilizando a linguagem Java, a partir de especificações no formato YAML (oferecendo legibilidade e simplicidade na codificação de dados). É possível utilizar tal ferramenta em modo interativo (com submissão de eventos e consultas ao motor de eventos em execução) e de biblioteca (possibilitando a inserção de um motor de eventos no contexto de uma aplicação).

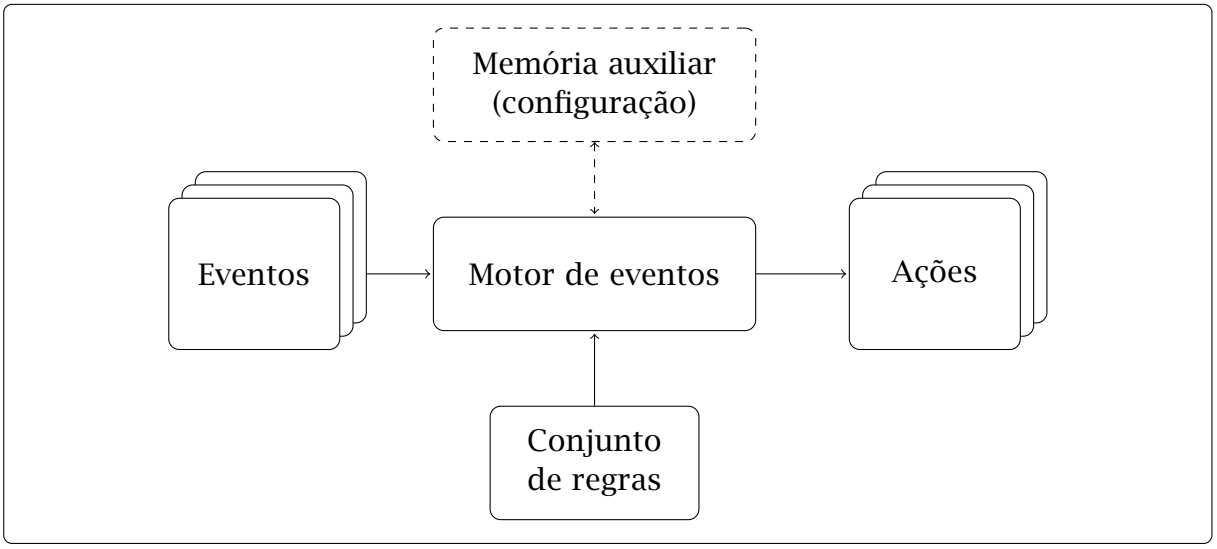
C.1 ASPECTOS DE IMPLEMENTAÇÃO

Um *motor de eventos* é um modelo de implementação para aplicações fundamentadas na simulação dirigida por estímulos e respostas correspondentes denominados *eventos* e *ações*. Tal técnica de simulação é abrangente e suficientemente expressiva para tratar ocorrências de interesse ordenadas sequencialmente (não obrigatoriamente uma ordenação temporal) (TANENBAUM; BOS, 2014). A Figura C.1 apresenta uma possível organização de um motor de eventos de propósito geral.

A ferramenta `eventengine` viabiliza a geração de motores de eventos organizados de acordo com a Figura C.1 a partir de especificações no formato serial YAML (acrônimo recursivo para *YAML ain't markup language* ou *YAML não é linguagem de marcação* no vernáculo) para armazenamento de dados (BEN-KIKI; EVANS; DÖT NET, 2009). Uma especificação de um motor de evento que serve como entrada para a ferramenta é dividida em três seções, apresentadas na Figura C.2 e descritas a seguir:

- A primeira seção contempla o *cabeçalho* da especificação, no qual é fornecido um identificador unívoco que nomeia o motor de eventos sendo definido. Para tal, utiliza-se o atributo nominal `identifier:` seguido de uma cadeia não-vazia de símbolos.

Figura C.1: Possível organização de um motor de eventos de propósito geral.



Fonte: autor.

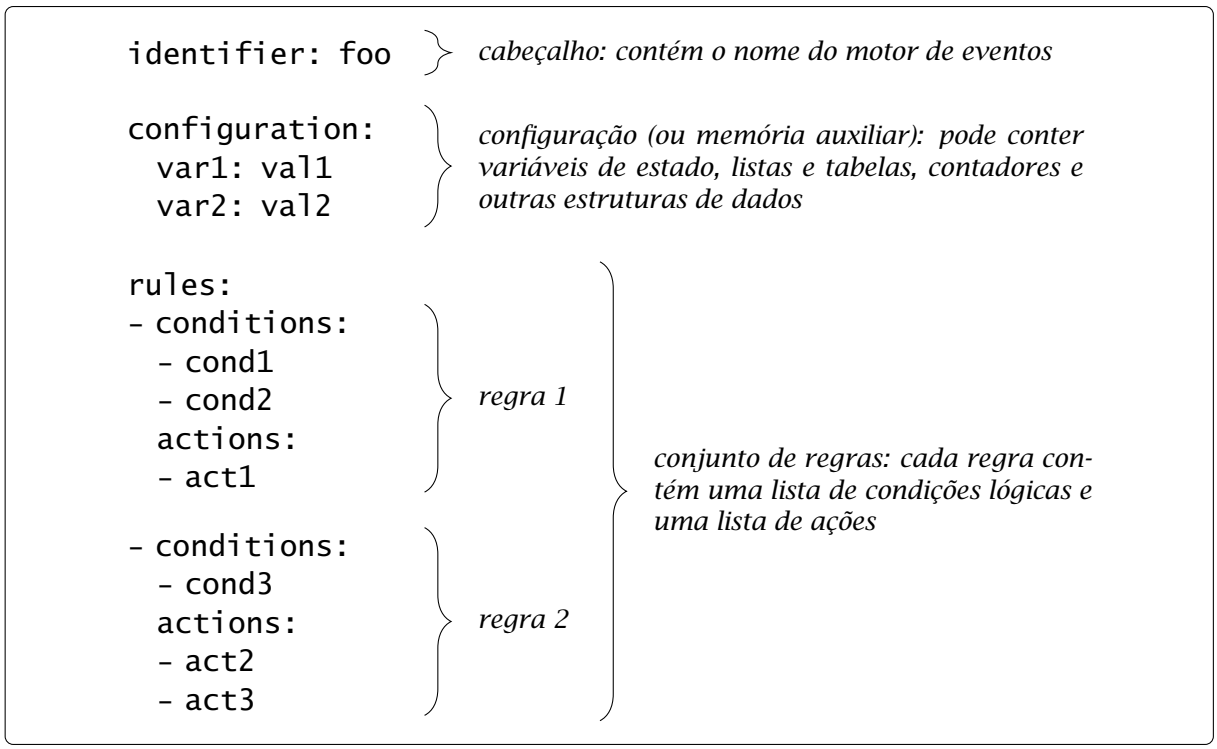
- A segunda seção contempla a *configuração* (ou memória auxiliar) do motor de eventos, conforme ilustra a Figura C.1. Uma configuração pode conter variáveis de estado, listas e tabelas, contadores e outras estruturas de dados, de acordo com o projeto de implementação. Tal seção é especificada através do atributo nominal `configuration:` seguido de um mapa de atributos (denotado por um conjunto de pares ordenados (x, y) tal que x, y representam nome e valor, respectivamente), potencialmente vazio (neste caso, denotado por `{}`).

- A terceira seção contempla o *conjunto de regras* do motor de eventos sendo definido, conforme ilustra a Figura C.1. Tal conjunto é denotado pelo atributo nominal `rules:` seguido de uma lista de regras, potencialmente vazia (neste caso, denotada por `[]`). Uma regra é definida como um mapa contendo uma lista de condições lógicas (denotada pelo atributo nominal `conditions:`) e uma lista de ações (denotada pelo atributo nominal `actions:`), ambas potencialmente vazias. Tais condições e ações são especificadas através de uma linguagem de expressão (do inglês *expression languagem*) e interpretadas em tempo de execução.

Um evento é denotado por um mapa de atributos, potencialmente vazio. Durante a execução do motor de eventos, dada a ocorrência de um evento, na situação em que mais de uma regra esteja elegível para aplicação, a ferramenta define a regra escolhida através de um sorteio. É possível alterar o critério de resolução de regras através da implementação de uma interface correspondente.

A linguagem de expressão disponibilizada na ferramenta `eventengine` permite a utilização das funcionalidades existentes na linguagem Java, incluindo suporte a classes e objetos, e execução de métodos. Adicionalmente, é possível definir eventos

Figura C.2: Exemplo de uma especificação de um motor de eventos no formato YAML, destacando as três seções principais.



Fonte: autor.

de saída, ordenados sequencialmente, para uma eventual composição hierárquica de motores de eventos, conforme ilustrado no Capítulo 3.

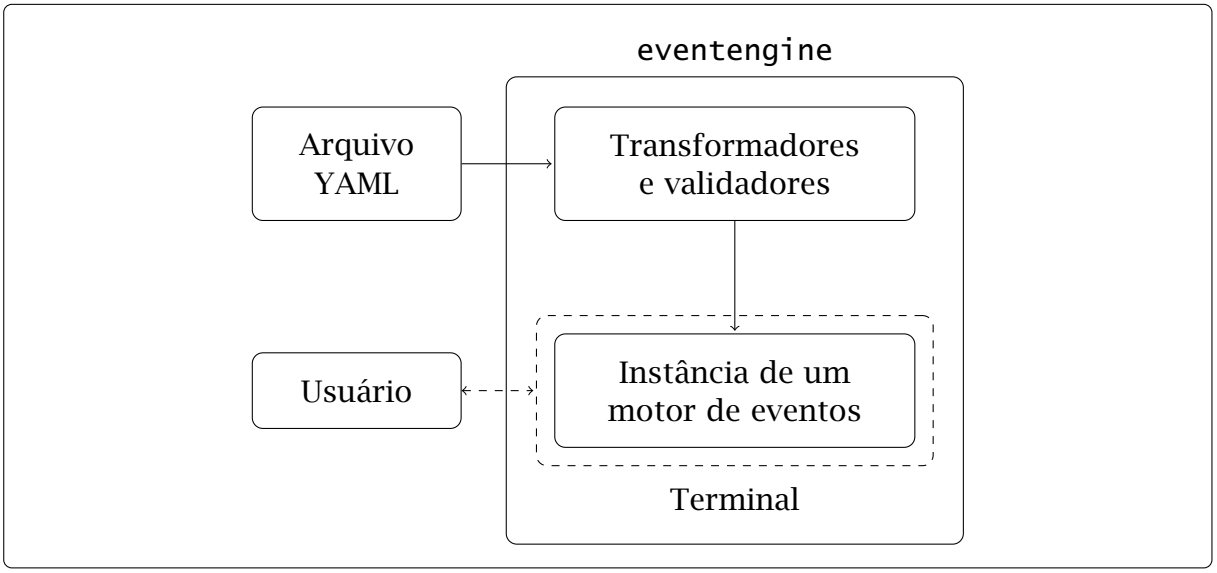
C.2 MODOS DE OPERAÇÃO

A ferramenta `eventengine` possui dois modos de operação. O primeiro é chamado *modo interativo*, no qual um arquivo YAML contendo a especificação do motor de eventos é fornecido à ferramenta e um terminal de consulta e submissão de eventos é disponibilizado ao usuário. A Figura C.3 ilustra a organização do modo interativo.

De acordo com a Figura C.3, a ferramenta `eventengine` realiza o carregamento do arquivo YAML e aplica transformações e validações na estrutura de tal arquivo. Como resultado, uma instância do motor de eventos correspondente é gerada e um terminal de consulta e submissão de eventos é disponibilizado ao usuário. A Seção C.3 apresenta exemplos de motores de eventos. É importante destacar que a máquina virtual Java é necessária para utilização da ferramenta, incluindo acesso através de um terminal de linha de comando do sistema operacional.

O terminal disponibilizado no modo interativo permite que o usuário submeta eventos na forma de mapas de atributos. Como resultado, a ferramenta informa se o motor de eventos processou tais ocorrências corretamente. O carregamento de um arquivo YAML contendo a especificação do motor de eventos ocorre através da ex-

Figura C.3: Organização da ferramenta `eventengine` em modo interativo.



Fonte: autor.

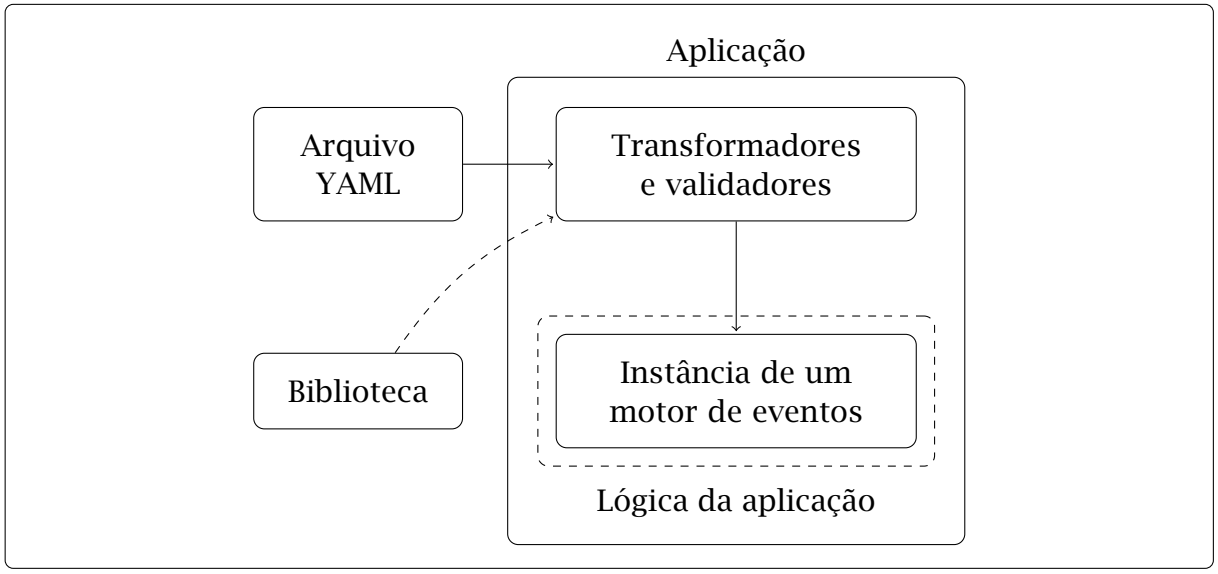
pressão `load` seguida do nome do arquivo. Adicionalmente, é possível inspecionar a configuração corrente, o conjunto de regras e a lista de eventos de saída através das expressões `configuration`, `rules` e `events`, respectivamente. Atributos nominais específicos da configuração podem ser acessados através de `configuration.name` ou `configuration['name']`, nos quais `name` representa o nome do atributo sendo acessado. Analogamente, regras ou eventos de saída específicos podem ser acessados através de seus índices numéricos correspondentes. A expressão `:quit` é utilizada para encerrar o terminal e o modo interativo da ferramenta. Mais informações, incluindo um manual completo da ferramenta `eventengine`, encontram-se disponíveis no endereço eletrônico `https://github.com/cereda/eventengine`.

O segundo modo de execução disponível é chamado *modo de biblioteca*, no qual a ferramenta `eventengine` é inserida no contexto de uma aplicação e atua como biblioteca propriamente dita, disponibilizando suas classes utilitárias para geração de uma instância de um motor de eventos a partir um arquivo YAML contendo uma especificação. A Figura C.4 ilustra a organização do modo de biblioteca.

De acordo com a Figura C.4, a ferramenta `eventengine` atua como biblioteca no contexto de uma aplicação e simplifica o processo de especificação de um motor de eventos diretamente em código Java, tornando-o transparente ao usuário, o qual deve apenas fornecer a especificação YAML e tratar do modelo gerado em alto nível. A Figura C.5 apresenta um exemplo da utilização da ferramenta em modo de biblioteca.

A utilização da ferramenta em modo de biblioteca consiste, em linhas gerais, na transformação do arquivo YAML em uma representação intermediária, validação de tal representação e construção efetiva da instância do motor de eventos correspon-

Figura C.4: Organização da ferramenta `eventengine` em modo de biblioteca.



Fonte: autor.

Figura C.5: Exemplo de utilização da ferramenta `eventengine` em modo de biblioteca. Por razões didáticas, partes supérfluas do código-fonte foram omitidas.

```
public class Main {

   public static void main(String[] args) {
      File file = new File("engine.yaml");

      EngineReader reader = new YAMLReader();
      Engine engine = reader.toEngine(file);
      ...
   }
}
```

Fonte: autor.

dente. Recomenda-se a utilização de ambientes integrados de desenvolvimento de aplicações Java (por exemplo, NetBeans e Eclipse) para facilitar o uso da ferramenta neste modo de operação.

C.3 EXEMPLOS DE USO

Esta seção apresenta exemplos de motores de eventos gerados a partir de especificações no formato YAML (Seção C.1). Observe que a ferramenta `eventengine` executa tais motores de eventos em modo interativo (Seção C.2). O Exemplo 29, a seguir, ilustra um servidor de eco como um motor de eventos.

Exemplo 29 (servidor de eco)**.** Considere a especificação YAML de um servidor de eco que imprime no terminal de comando uma representação textual do evento recebido, apresentada na Figura C.6.

Figura C.6: Especificação YAML de um servidor de eco como um motor de eventos.

```
identifier: echo-server

configuration: {}

rules:
- conditions: []
  actions:
  - System.out.println('ECHO => ' + event)
```

Fonte: autor.

Observe que, de acordo com a Figura C.6, tal especificação não requer memória auxiliar (denotada pelo mapa vazio `{}`). Adicionalmente, a única regra é aplicada a todos os eventos (dado que esta não possui condições associadas), imprimindo a palavra `ECHO =>` seguida de uma representação textual do evento corrente. A Figura C.7 apresenta um exemplo de sessão interativa.

Conforme ilustra a Figura C.7, o motor de eventos `echo-server` implementa um servidor de eco e imprime uma representação textual do evento corrente no terminal de comando. Observe que o tratamento de eventos ocorre sob demanda. □

O Exemplo 29 apresentou um servidor de eco implementado como um motor de eventos, imprimindo representações textuais dos eventos recebidos no terminal de comando. O Exemplo 30, a seguir, ilustra um analisador de palíndromos.

Exemplo 30 (analisador de palíndromos)**.** Considere a especificação YAML de um analisador de palíndromos que imprime no terminal o resultado da análise do atributo `text` do evento recebido, apresentada na Figura C.8.

Observe que, de acordo com a Figura C.8, tal especificação não requer memória auxiliar (denotada pelo mapa vazio `{}`). A primeira regra verifica se o valor do atributo `text` do evento corrente (denotado nominalmente por `event.text`) é igual ao seu reverso. Em caso positivo, este é classificado como um palíndromo. A segunda regra representa o complemento da primeira (isto é, verifica se o valor de `text` não é um palíndromo). O resultado de tal análise é impresso no terminal de comando. A Figura C.9 apresenta um exemplo de sessão interativa.

Conforme ilustra a Figura C.9, o motor de eventos `palindrome-analyzer` implementa um analisador de palíndromos, imprimindo o resultado da análise do atributo

Figura C.7: Exemplo de sessão interativa do motor de eventos de um servidor de eco (Figura C.6).

```
[paulo@cambridge ~] $ java -jar eg.jar
Laboratório de Técnicas e Linguagens Adaptativas
Escola Politécnica, Universidade de São Paulo
Versão 1.1

[1] > load('eco.yaml')
(PAIR, (READ_ENGINE, eco.yaml))
Novo motor de eventos carregado com sucesso.
[2] > { 'symbol' : 'b', 'id' : 12 }
(HASHMAP, {symbol=b, id=12})
Submetendo evento ao motor de eventos...
ECHO => {symbol=b, id=12}
O motor de eventos conseguiu processar o evento informado.
[3] > { 'text' : 'hello' }
(HASHMAP, {text=hello})
Submetendo evento ao motor de eventos...
ECHO => {text=hello}
O motor de eventos conseguiu processar o evento informado.
[4] > :quit
Isso é tudo, pessoal!
```

Fonte: autor.

Figura C.8: Especificação YAML de um analisador de palíndromos como um motor de eventos.

```
identifier: palindrome-analyzer

configuration: {}

rules:
- conditions:
  - event.text == new StringBuilder(event.text).
                  reverse().toString()
  actions:
  - System.out.println('This text is a palindrome')

- conditions:
  - event.text != new StringBuilder(event.text).
                  reverse().toString()
  actions:
  - System.out.println('This text is not a palindrome')
```

Fonte: autor.

Figura C.9: Exemplo de sessão interativa do motor de eventos de um analisador de palíndromos (Figura C.8).

```
[paulo@cambridge ~] $ java -jar eg.jar
Laboratório de Técnicas e Linguagens Adaptativas
Escola Politécnica, Universidade de São Paulo
Versão 1.1

[1] > load('palindromo.yaml')
(PAIR, (READ_ENGINE, palindromo.yaml))
Novo motor de eventos carregado com sucesso.
[2] > { 'text' : 'arara' }
(HASHMAP, {text=arara})
Submetendo evento ao motor de eventos...
This text is a palindrome
O motor de eventos conseguiu processar o evento informado.
[3] > { 'text' : 'parrot' }
(HASHMAP, {text=parrot})
Submetendo evento ao motor de eventos...
This text is not a palindrome
O motor de eventos conseguiu processar o evento informado.
[4] > :quit
Isso é tudo, pessoal!
```

Fonte: autor.

`text` do evento recebido no terminal de comando. Observe que tal motor de eventos não dispõe de tratamento definido para eventos sem o atributo nominal `text` (Figura C.8). Neste caso, os eventos são ignorados. □

O Exemplo 29 apresentou um analisador de palíndromos implementado como um motor de eventos, imprimindo o resultado da análise do atributo `text` dos eventos recebidos no terminal de comando. O Exemplo 31 ilustra um autômato finito.

Exemplo 31 (autômato finito)**.** Considere a especificação YAML de um autômato finito determinístico H que reconhece cadeias pertencentes à linguagem $L = \{w \in \{a,b\}^* \mid w = a(ba)^*\}$, apresentada na Figura C.10. O autômato H correspondente é ilustrado na Figura C.11.

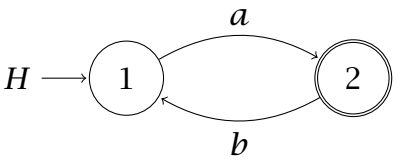
Observe que, de acordo com a Figura C.10, tal especificação utiliza a memória auxiliar para armazenar o estado corrente do autômato (denotado pelo atributo nominal `state`). Adicionalmente, as regras definem a relação de transição do autômato H, considerando o estado corrente armazenado na memória auxiliar e o valor do atributo `symbol` do evento para uma eventual mudança de estado. A Figura C.12 apresenta um exemplo de sessão interativa.

Figura C.10: Especificação YAML de um autômato finito determinístico H que reconhece cadeias pertencentes à linguagem $L = \{w \in \{a,b\}^* \mid w = a(ba)^*\}$ como um motor de eventos.

```
identifier: finite-automaton

configuration:
  state: 1

rules:
- conditions:
  - configuration['state'] == 1
  - event['symbol'] == 'a'
  actions:
  - configuration['state'] = 2

- conditions:
  - configuration['state'] == 2
  - event['symbol'] == 'b'
  actions:
  - configuration['state'] = 1
```

Fonte: autor.

Figura C.11: Autômato finito determinístico H que reconhece cadeias pertencentes à linguagem $L = \{w \in \{a,b\}^* \mid w = a(ba)^*\}$.



Fonte: autor.

Conforme ilustra a Figura C.12, o motor de eventos `finite-automaton` implementa o autômato finito determinístico H da Figura C.11. Observe que o valor do atributo `state` da memória auxiliar é atualizado de acordo com a aplicação das regras válidas sobre os eventos submetidos. □

Os exemplos apresentados nesta seção podem ser estendidos conforme a necessidade da aplicação ou conveniência do usuário. Recomenda-se o uso da ferramenta `eventengine` em modo interativo como material didático de apoio para disciplinas de computação, e em modo de biblioteca para inserção direta no contexto de aplicações Java, disponibilizando classes utilitárias para geração de instâncias de motores de eventos a partir de especificações em formato YAML.

Figura C.12: Exemplo de sessão interativa do motor de eventos de um autômato finito (Figura C.10).

```
[paulo@cambridge ~] $ java -jar eg.jar
Laboratório de Técnicas e Linguagens Adaptativas
Escola Politécnica, Universidade de São Paulo
Versão 1.1

[1] > load('automato.yaml')
(PAIR, (READ_ENGINE, sample1.yaml))
Novo motor de eventos carregado com sucesso.
[2] > { 'symbol' : 'a' }
(HASHMAP, {symbol=a})
Submetendo evento ao motor de eventos...
O motor de eventos conseguiu processar o evento informado.
[3] > { 'symbol' : 'b' }
(HASHMAP, {symbol=b})
Submetendo evento ao motor de eventos...
O motor de eventos conseguiu processar o evento informado.
[4] > { 'symbol' : 'a' }
(HASHMAP, {symbol=a})
Submetendo evento ao motor de eventos...
O motor de eventos conseguiu processar o evento informado.
[5] > configuration
(CONFIGURATION, Configuração: {
        - state -> 2
})
[6] > :quit
Isso é tudo, pessoal!
```

Fonte: autor.

APÊNDICE D

UM EXPANSOR DE MACROS IMPLEMENTADO NA LINGUAGEM JAVA

> A man provided with paper,
> pencil, and rubber, and
> subject to strict discipline, is
> in effect a universal machine.
>
> ALAN TURING

Este apêndice apresenta um expansor de macros de propósito geral, implementado utilizando a linguagem Java, com padrão sintático dependente de contexto (tipo 1 na hierarquia de Chomsky). Complementarmente, tal expansor disponibiliza uma interface gráfica do usuário e um conjunto de primitivas de componentes visuais.

D.1 CONJUNTO DE PRIMITIVAS

O expansor de macros de propósito geral proposto (referenciado como *E* no escopo deste apêndice) segue a estrutura apresentada na Seção A.2 (página 139), com delimitadores dependentes de contexto, resolução de capturas como macros de escopo local (Subseção A.3.2, página 158) e com suporte a transformações algorítmicas (Subseção A.3.3, página 163). A Tabela D.1 apresenta correspondências entre as primitivas descritas na Seção 4.1 e as disponibilizadas no expansor de macros *E*. É importante destacar que os nomes de primitivas e macros em *E* são normalizados, conforme descrito na Seção A.2.

Adicionalmente, o expansor *E* disponibiliza primitivas de componentes visuais através das classes utilitárias Swing, incluindo a implementação efetiva da primitiva `INPUT` ilustrada na Figura 4.26. Tais primitivas são descritas a seguir.

A primitiva `get input` disponibiliza uma caixa de diálogo de entrada de dados, utilizando recursos do sistema operacional e da máquina virtual Java. Opcionalmente, o texto digitado pode ser expandido através da marcação explícita da caixa de seleção disponibilizada na interface. A Figura D.1 apresenta a sintaxe e operação de tal primitiva. As capturas (com exceção do texto inicial) são devidamente expandidas antes de sua exibição na interface gráfica do usuário.

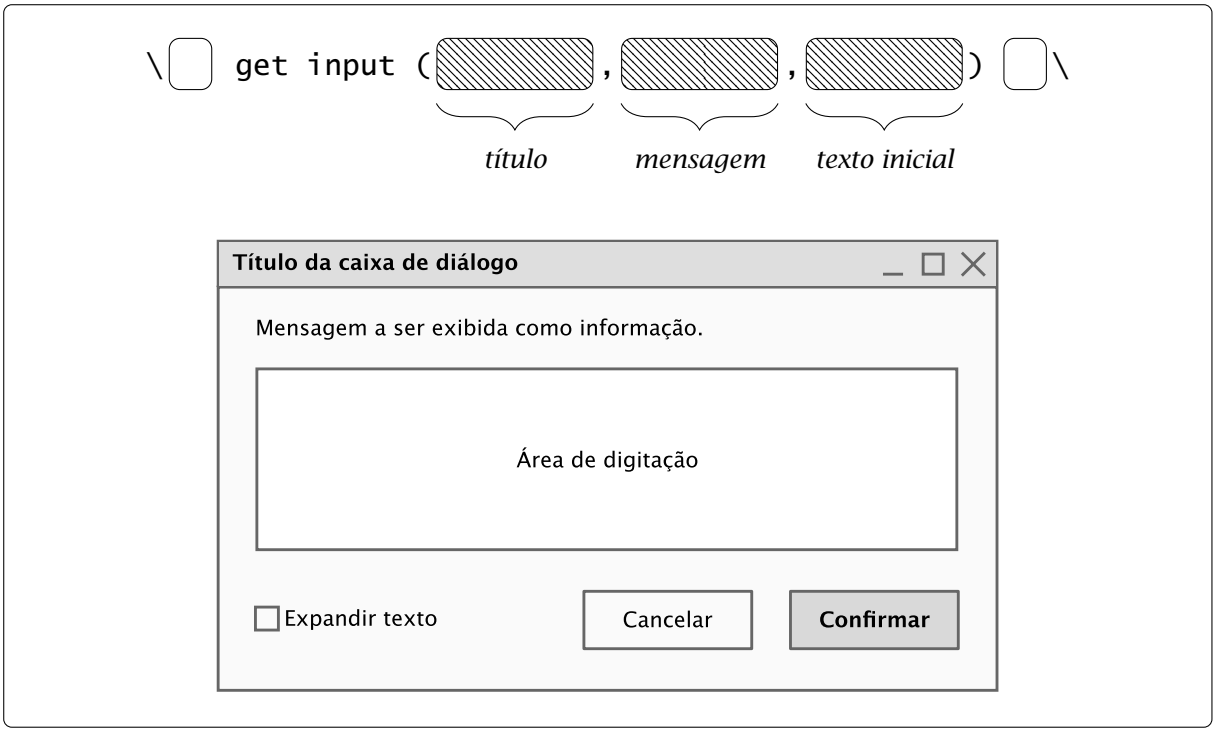
A primitiva `ask question` disponibiliza uma caixa de diálogo de confirmação, utilizando recursos do sistema operacional e da máquina virtual Java. Tal primitiva retorna um valor lógico correspondente ao botão pressionado na interface (verdadeiro para a opção positiva, e falso caso contrário). A Figura D.2 apresenta a sintaxe

Tabela D.1: Correspondências entre as primitivas descritas na Seção 4.1 e as disponibilizadas no expansor de macros *E* proposto neste apêndice.

Primitivas disponibilizadas			
Seção 4.1	Expansor *E*	Seção 4.1	Expansor *E*
`DEF`	`define`	`UNDEF`	`undefine`
`INC`	`increment`	`DEC`	`decrement`
`IF`	`if`	`UNLESS`	`unless`
`GREATER`	`greater than`	`LESS`	`less than`
`EQUAL`	`equal`	`ADD`	`add`
`SUB`	`subtract`	`MULT`	`multiply`
`DIV`	`divide`	`MOD`	`modulo`
`AND`	`and`	`OR`	`or`
`NOT`	`not`	`PRJ`	`projection`
`REPEAT`	`repeat`	`SET`	`set symbol`
`GET`	`get symbol`	`LIT`	`literal`
`EXS`	`strategy`	`WHILE`	`while`
`INSERT`	`insert`	`REMOVE`	`remove`
`QUERY`	`query`		

Fonte: autor.

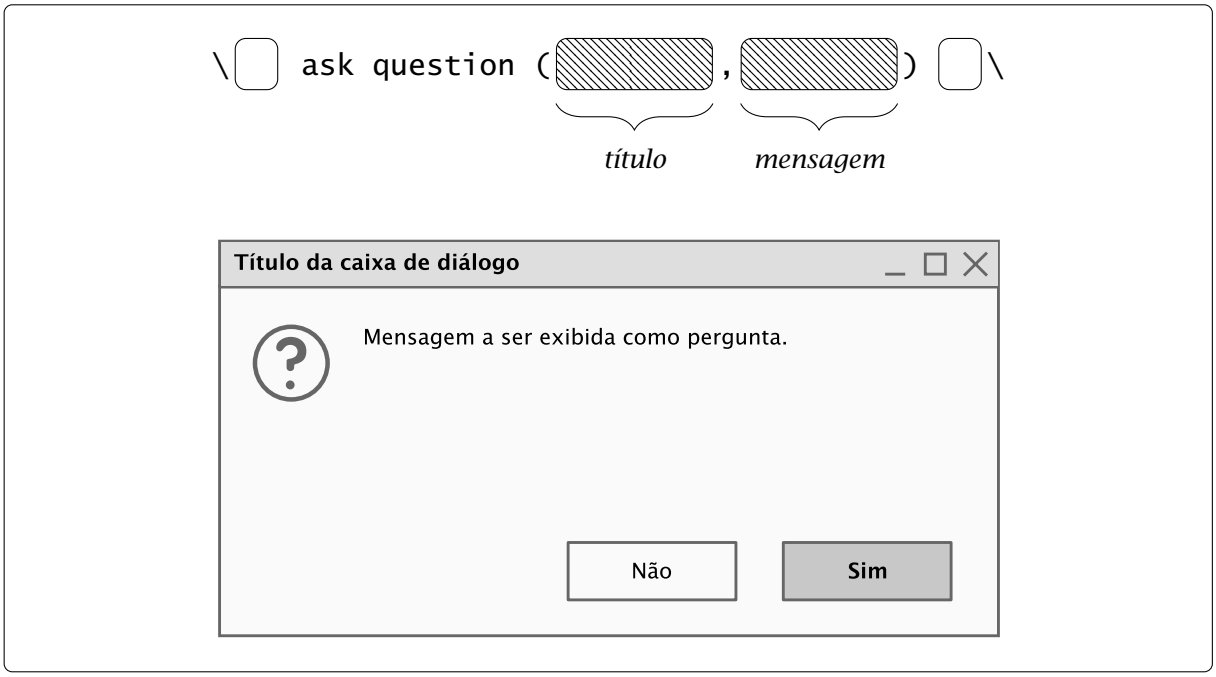
Figura D.1: Sintaxe e operação da primitiva `get input` de exibição de uma caixa de diálogo de entrada de dados, utilizando recursos do sistema operacional.



Fonte: autor.

e operação de tal primitiva. As capturas são devidamente expandidas utilizando a estratégia de pré-varredura (Definição 57).

Figura D.2: Sintaxe e operação da primitiva `ask question` de exibição de uma caixa de diálogo de confirmação, utilizando recursos do sistema operacional.



Fonte: autor.

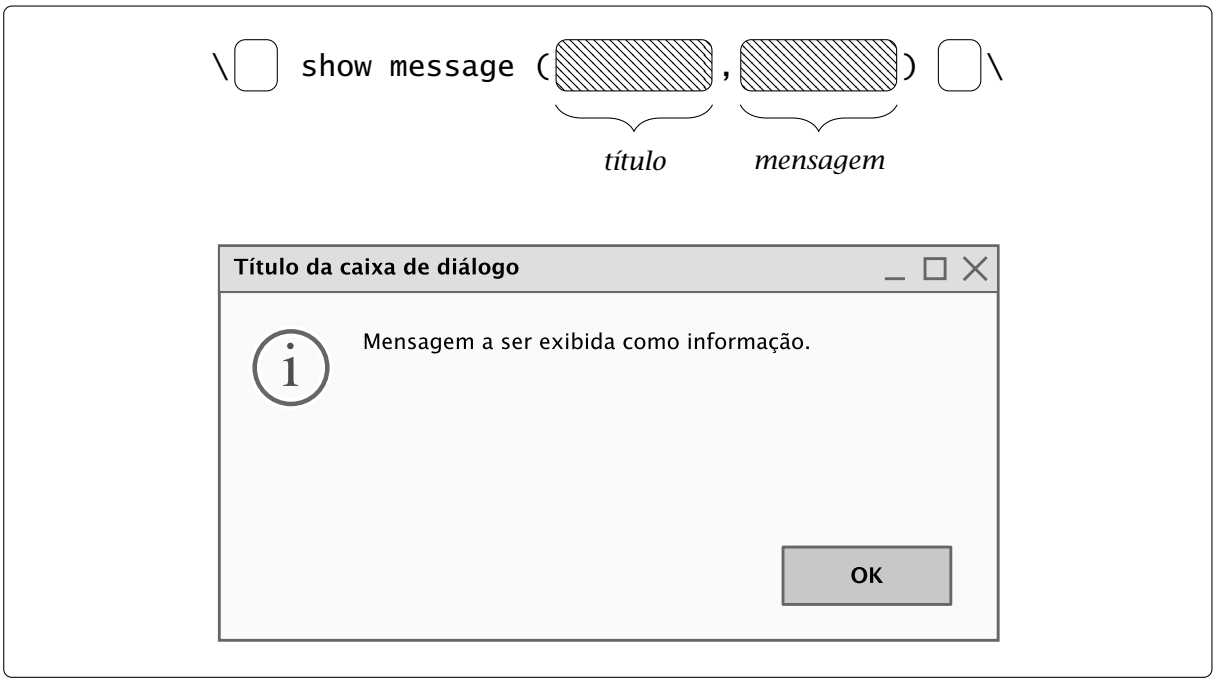
Finalmente, a primitiva `show message` disponibiliza uma caixa de diálogo de mensagem, utilizando recursos do sistema operacional e da máquina virtual Java. A Figura D.3 apresenta a sintaxe e operação de tal primitiva. Como a primitiva `show message` é informativa, ϵ é retornado como resultado da expansão (denotando a cadeia vazia), após a exibição da caixa de diálogo propriamente dita.

O conjunto de primitivas disponibilizado para o expansor de macros *E* apresentado neste apêndice oferece subsídios para a implementação de algoritmos de propósito geral durante a expansão de macros. Adicionalmente, as primitivas de componentes visuais (Figuras D.1, D.2 e D.3) estendem as funcionalidades existentes e viabilizam estruturas convenientes para interação do usuário com o expansor de macros, de forma direta e simplificada.

D.2 INTERFACE GRÁFICA DO USUÁRIO

O expansor de macros de propósito geral *E* disponibiliza uma interface gráfica do usuário, através das classes utilitárias Swing, com o propósito de auxiliar a escrita e a transformação de textos contendo potenciais definições e instâncias de macros. Tal interface é ilustrada na Figura D.4.

Figura D.3: Sintaxe e operação da primitiva `show message` de exibição de uma caixa de diálogo de mensagem, utilizando recursos do sistema operacional.



Fonte: autor.

Figura D.4: Interface gráfica do usuário disponibilizada pelo expansor de macros *E*.

Fonte: autor.

De acordo com a Figura D.4, a interface gráfica do usuário apresenta duas áreas de digitação localizadas na parte superior e inferior, respectivamente. A primeira área permite que o usuário escreva um texto qualquer, incluindo potenciais definições e instâncias de macros. Alternativamente, é possível carregar o conteúdo de um arquivo texto localizado no sistema de arquivos para tal área de digitação através da ação associada ao botão *abrir*. Complementarmente, é possível gravar o conteúdo da área em um arquivo texto através da ação associada ao botão *salvar*. A segunda área de digitação atua em modo de leitura e disponibiliza o resultado da expansão do texto proveniente da primeira área. A expansão de macros propriamente dita ocorre através da ação associada ao botão *executar*. Finalmente, é possível reiniciar as áreas de digitação (isto é, torná-las vazias) através da ação associada ao botão *limpar*. É importante destacar que, durante a expansão de macros, os botões da interface gráfica do usuário tornam-se indisponíveis.

D.3 EXEMPLOS DE USO

Esta seção apresenta exemplos de expansão de macros viabilizados através do expansor de macros *E* (interface gráfica do usuário da Seção D.2) e do conjunto de primitivas introduzido na Seção D.1. O Exemplo 32, a seguir, ilustra o cálculo do fatorial de um número através de recursão.

Exemplo 32 (cálculo do fatorial de um número)**.** Considere a versão recursiva do cálculo do fatorial de um número (Algoritmo 1.1-a). A definição da macro `fatorial` e a expansão de algumas instâncias na interface gráfica do usuário são apresentadas nas Figuras D.5 e D.6. □

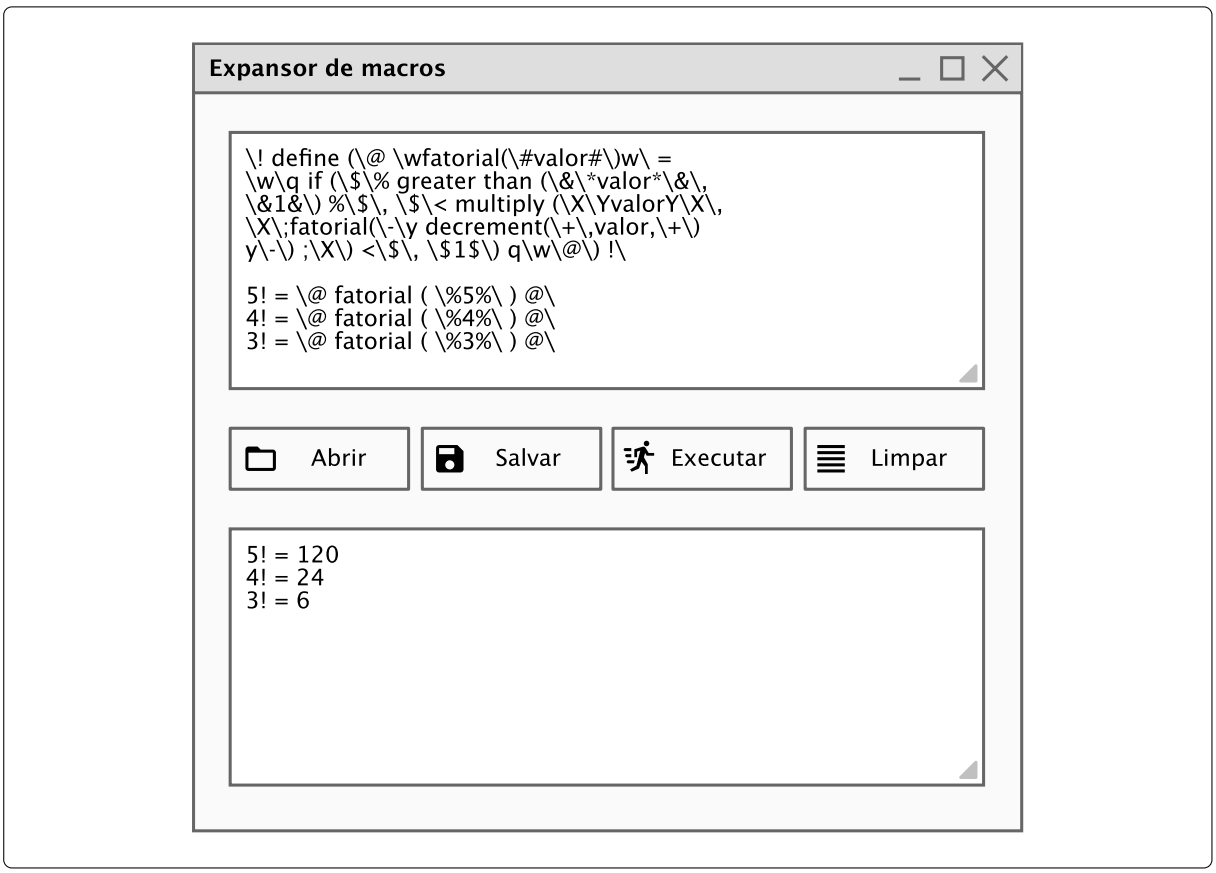
Figura D.5: Definição da macro `fatorial` que calcula recursivamente o fatorial de um número, de acordo com o Algoritmo 1.1-a.

```
\! define (\@ \wfatorial(\#valor#\)w\ =
\w\q if (\$\% greater than (\&\*valor*\&\,
\&1&\) %\$\, \$\< multiply (\X\YvalorY\X\,
\X\;fatorial(\-\y decrement(\+\,valor,\+\)
y\-\) ;\X\) <\$\, \$1$\) q\w\@\) !\
```

Fonte: autor.

O Exemplo 32 apresentou uma versão recursiva do cálculo do fatorial de um número, incluindo a expansão de algumas instâncias na interface gráfica do usuário. O Exemplo 33, a seguir, ilustra a impressão da fórmula do fatorial de um número.

Figura D.6: Expansão de algumas instâncias da macro `fatorial` (Figura D.5) na interface gráfica do usuário.



Fonte: autor.

Exemplo 33 (impressão da fórmula do fatorial de um número)**.** Considere uma versão recursiva da impressão da fórmula do fatorial de um número (sequência finita de multiplicações), inspirada no Algoritmo 1.1-a (neste caso, sem o cálculo efetivo). A definição da macro `imprime fatorial` e a expansão de algumas instâncias na interface gráfica do usuário são apresentadas nas Figuras D.7 e D.8. □

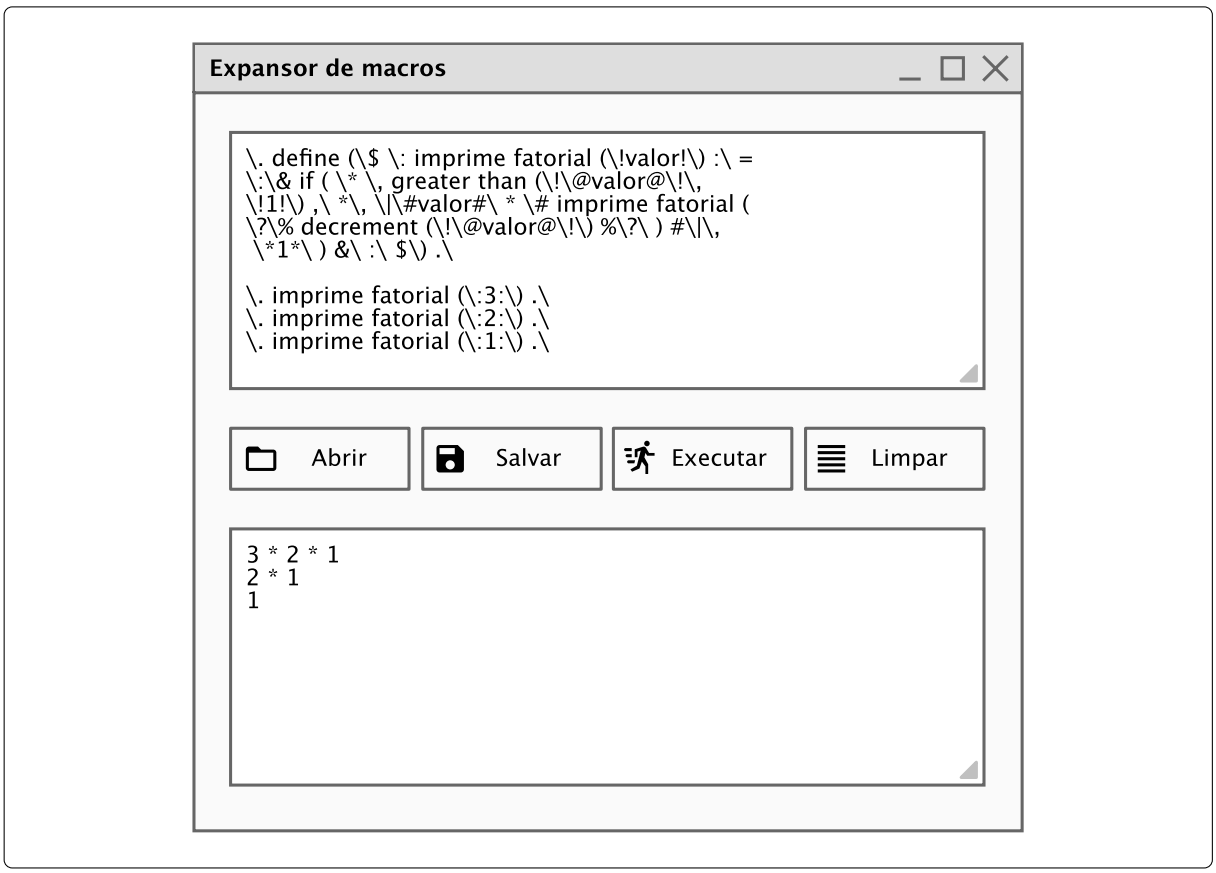
Figura D.7: Definição da macro `imprime fatorial` que imprime recursivamente a fórmula do fatorial de um número, inspirada no Algoritmo 1.1-a.

```
\. define (\$ \: imprime fatorial (\!valor!\) :\ =
\:\& if ( \* \, greater than (\!\@valor@\!\,
\!1!\) ,\ *\, \|\#valor#\ * \# imprime fatorial (
\?\% decrement (\!\@valor@\!\) %\?\ ) #\|\,
 \*1*\ ) &\ :\ $\) .\
```

Fonte: autor.

É importante destacar que os exemplos apresentados na Seção 4.2 são compatíveis com o expansor de macros *E* proposto neste apêndice, desde que as devidas

Figura D.8: Expansão de algumas instâncias da macro `imprime fatorial` (Figura D.7) na interface gráfica do usuário.



Fonte: autor.

modificações nos padrões sintáticos sejam realizadas corretamente. A interface gráfica oferece um ambiente integrado de edição e expansão de macros ao usuário, simplificando a definição e aplicação de transformações textuais.

APÊNDICE E

UM GERADOR DE SENTENÇAS LIVRES DE CONTEXTO

> If numbers aren't beautiful, I don't know what is.
>
> PAUL ERDŐS

Gramáticas e macros constituem instâncias do fenômeno de reescrita de termos. Em linhas gerais, a primeira pode ser considerada um caso particular da segunda. Este apêndice apresenta uma ferramenta para geração de sentenças a partir de gramáticas livres de contexto, implementada utilizando a linguagem Java, com a possibilidade de inclusão de restrições positivas e negativas acerca das produções de tais gramáticas. Uma restrição influencia a escolha de qual produção aplicar durante a geração de uma sentença. Complementarmente, a ferramenta proposta disponibiliza uma interface gráfica do usuário para facilitar a geração de sentenças.

E.1 REPRESENTAÇÃO TEXTUAL DAS PRODUÇÕES

A ferramenta proposta gera sentenças pertencentes à linguagem descrita por uma gramática livre de contexto. A representação textual das produções de tal gramática segue o formato descrito na Figura E.1, em notação de Wirth. Observe que `terminal` e `nonterm` denotam símbolos terminais e não-terminais, respectivamente.

Figura E.1: Representação textual das produções de uma gramática livre de contexto como formato de entrada da ferramenta de geração de sentenças.

```
grammar = { production } .
production = nonterm "=" ( ( ( nonterm | terminal )
          { nonterm | terminal } ) | "#" ) "." .
```

Fonte: autor.

De acordo com a Figura E.1, uma produção da gramática pode conter um número arbitrário de símbolos terminais e não-terminais. No caso de uma produção que resulta em uma cadeia vazia, o símbolo ϵ é denotado por `#` (em sua forma literal, sem aspas). Assim, as produções $S \rightarrow aSb$ e $S \rightarrow \epsilon$, com $a, b \in \Sigma$ (isto é, a e b são símbolos terminais), são denotadas por `S = "a" S "b" .` e `S = # .`, respectivamente. Operacionalmente, o não-terminal à esquerda do símbolo `=` da primeira produção (no exemplo, `S`) será definido como raiz da gramática. Esta notação foi adotada por

razões técnicas, limitando a representação interna dos elementos das produções a terminais e não-terminais.

E.2 INTERFACE GRÁFICA DO USUÁRIO

A ferramenta descrita neste apêndice disponibiliza uma interface gráfica do usuário, através das classes utilitárias Swing, com o propósito de auxiliar a escrita de produções gramaticais e a geração efetiva de sentenças. Adicionalmente, é possível influenciar a geração de tais sentenças através da inclusão de restrições para escolha das produções. Tal interface é ilustrada na Figura E.2.

Figura E.2: Interface gráfica do usuário disponibilizada pelo gerador de sentenças.

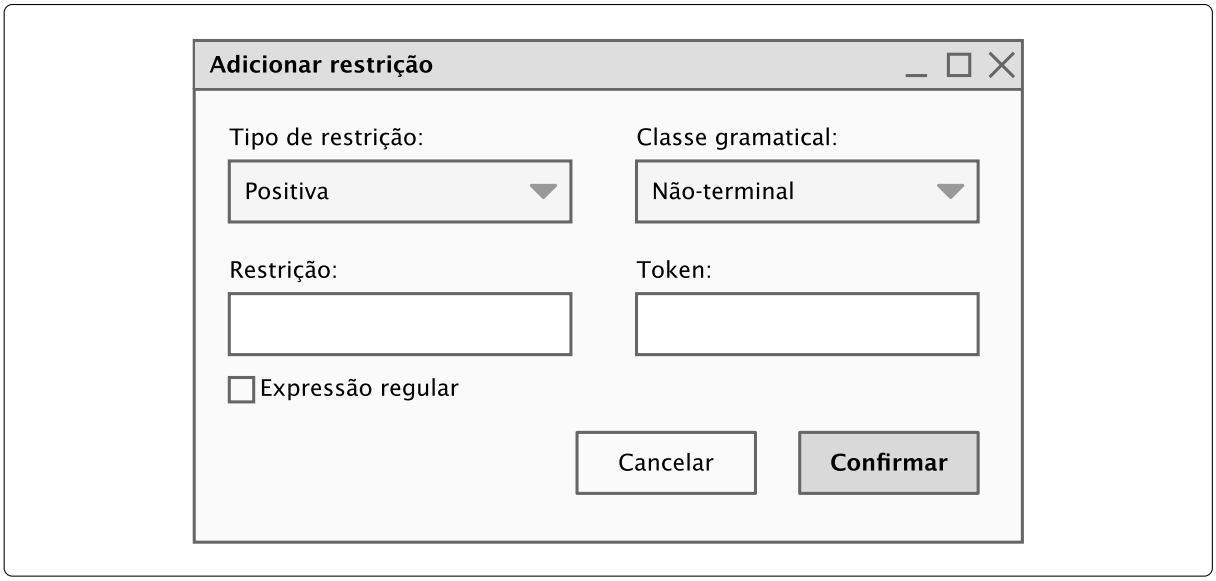
Fonte: autor.

De acordo com a Figura E.2, a interface gráfica do usuário apresenta duas áreas de digitação localizadas na parte superior e inferior, respectivamente. A primeira área permite que o usuário escreva as produções de uma gramática livre de contexto de acordo com a representação textual descrita na Seção E.1. A segunda área de digitação atua em modo de leitura e disponibiliza o resultado da geração de uma

sentença, de acordo com as produções provenientes da primeira área. A geração de uma sentença ocorre através da ação associada ao botão *gerar*, localizado na parte inferior da interface gráfica.

A região central da interface gráfica do usuário (Figura E.2) apresenta uma tabela, inicialmente vazia, e dois botões de ação. Tal região refere-se à inserção de restrições positivas e negativas consideradas durante a geração de uma sentença. Inicialmente, as produções elegíveis em uma derivação têm a mesma probabilidade de escolha. Ao introduzir uma regra de restrição, o gerador desconsidera produções não aderentes. No caso da aplicação de uma restrição positiva, as produções remanescentes contemplam os critérios determinados pela regra. Complementarmente, a aplicação de uma restrição negativa define os critérios de corte das produções elegíveis para aplicação. A Figura E.3 apresenta uma caixa de diálogo de definição de uma restrição na ferramenta de geração de sentenças, disponibilizada através da ação associada ao botão *adicionar*. Analogamente, a ação associada ao botão *remover* remove a restrição selecionada da tabela da interface gráfica.

Figura E.3: Caixa de diálogo de definição de uma restrição na ferramenta de geração de sentenças.



Fonte: autor.

De acordo com a Figura E.3, é possível especificar o tipo de restrição (positiva ou negativa) e a classe gramatical associada (terminal ou não-terminal) nas caixas de combinação correspondentes. Adicionalmente, é necessário fornecer os valores da restrição (com a opção de tratamento como expressão regular através da marcação explícita da caixa de seleção) e do *token* componente da produção.

E.3 EXEMPLOS DE USO

Esta seção apresenta exemplos de geração de sentenças a partir de gramáticas livres de contexto, com a possibilidade de inclusão de restrições positivas e negativas acerca das produções, utilizando a ferramenta proposta. O Exemplo 34, a seguir, ilustra a geração de uma sentença a partir de uma gramática livre de contexto.

Exemplo 34 (geração de sentença sem restrições)**.** Considere uma gramática livre de contexto G tal que $L(G) = \{w \in \{a,b\}^* \mid w = a^n b^n, n \geq 0\}$. A representação textual das produções de G e a geração de uma sentença na interface gráfica do usuário são apresentadas nas Figuras E.4 e E.5. □

Figura E.4: Representação textual das produções da gramática livre de contexto G, tal que $L(G) = \{w \in \{a,b\}^* \mid w = a^n b^n, n \geq 0\}$.

```
S = "a" S "b" .
S = # .
```

Fonte: autor.

O Exemplo 34 apresentou a geração de uma sentença, dada uma representação textual das produções de uma gramática livre de contexto, através da interface gráfica do usuário. O Exemplo 35 ilustra a geração de uma sentença a partir de uma gramática regular com a inclusão de duas restrições.

Exemplo 35 (geração de sentença com restrições)**.** Considere uma gramática regular H cujo conjunto de regras de produção forma sentenças em português contendo um substantivo e um verbo. A representação textual das produções de H e a geração de uma sentença na interface gráfica do usuário são apresentadas nas Figuras E.6 e E.7.

Observe que, de acordo com a Figura E.7, duas restrições são introduzidas na geração de uma sentença. A primeira restrição é negativa e descarta produções gramaticais que possuam `pedra` como valor associado ao não-terminal `NOUN` (uma ocorrência). Complementarmente, a segunda restrição é positiva e seleciona produções gramaticais cujo valor associado ao não-terminal `VERB` possui a desinência `ou`, indicado pela expressão regular correspondente (duas ocorrências). Tais restrições são verificadas em tempo de geração da cadeia, durante a análise da produção corrente. □

O Exemplo 35 apresentou a geração de uma sentença com a inclusão de restrições a partir de uma gramática regular, através da interface gráfica do usuário. O Exemplo 36 ilustra a geração de uma sentença em português contendo estruturas aninhadas a partir de uma gramática livre de contexto.

Figura E.5: Geração de uma sentença a partir da representação textual das produções da gramática livre de contexto G (Figura E.4), sem restrições, na interface gráfica do usuário.

Fonte: autor.

Figura E.6: Representação textual das produções da gramática regular H.

```
S = NOUN VERB .
VERB = "casou" .
VERB = "andou" .
VERB = "sobe" .
VERB = "subiu" .
NOUN = "pedra" .
NOUN = "menino" .
```

Fonte: autor.

Figura E.7: Geração de uma sentença a partir da representação textual das produções da gramática regular H (Figura E.6), com restrições, na interface gráfica do usuário.

Fonte: autor.

Exemplo 36 (geração de uma sentença em português contendo estruturas aninhadas)**.** Considere uma gramática livre de contexto I cujo conjunto de regras de produção forma sentenças em português contendo estruturas aninhadas. A representação textual das produções de I e a geração de uma sentença na interface gráfica do usuário são apresentadas nas Figuras E.8 e E.9. □

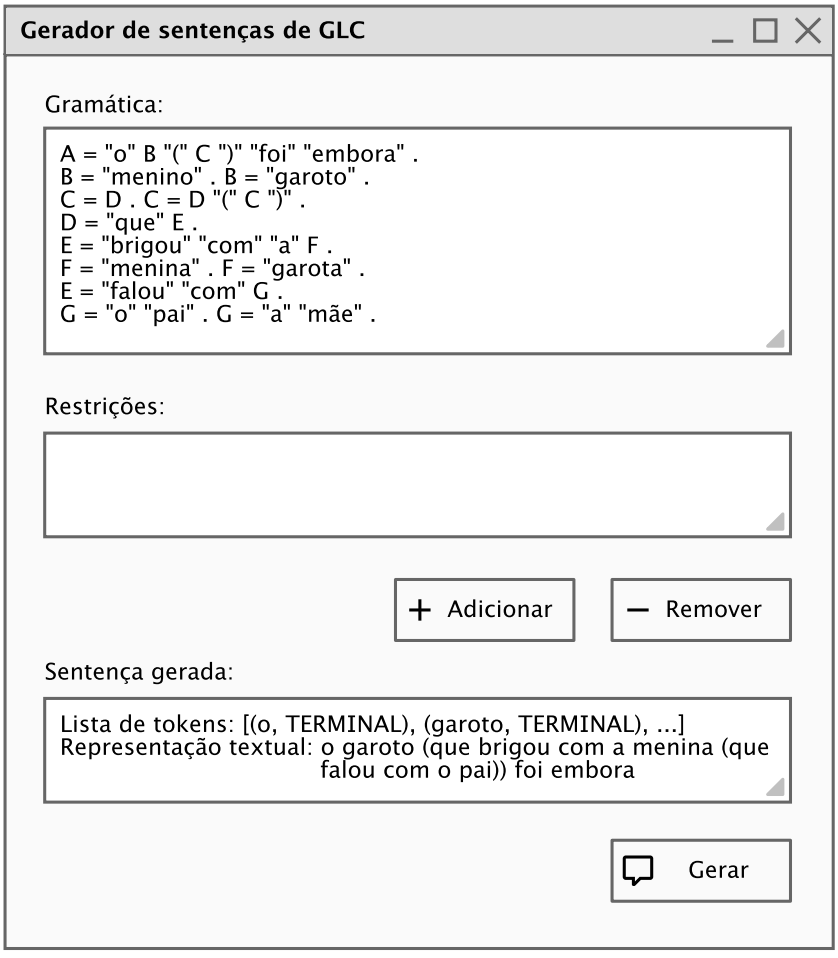
A ferramenta de geração de sentenças a partir de gramáticas livres de contexto apresentada neste apêndice pode ser estendida para oferecer suporte a componentes visuais (caixas de diálogo de entrada de dados, seleção e mensagem), proporcionando maior controle do usuário sobre a escolha das produções elegíveis e oferecendo manipulação direta dos valores associados a terminais e não-terminais.

Figura E.8: Representação textual das produções da gramática livre de contexto *I*.

```
A = "o" B "(" C ")" "foi" "embora" .
B = "menino" .
B = "garoto" .
C = D .
C = D "(" C ")" .
D = "que" E .
E = "brigou" "com" "a" F .
F = "menina" .
F = "garota" .
E = "falou" "com" G .
G = "o" "pai" .
G = "a" "mãe" .
```

Fonte: autor.

Figura E.9: Geração de uma sentença a partir da representação textual das produções da gramática livre de contexto *I* (Figura E.8) na interface gráfica do usuário.



Fonte: autor.

GLOSSÁRIO

abstração

Processo que extrai e preserva alguma propriedade de um artefato. Essa propriedade serve como base no qual o intelecto forma uma imagem ou conceito cognitivo de tal artefato (MARYNIARCZYK, 2000). ☞ *Def. 1, p. 26*

abstração textual

Processo de abstração que identifica que determinados fragmentos de texto podem ser retirados de seus contextos. Esses fragmentos são comparados de tal forma que o resultado seja uma descrição de suas estruturas comuns (KOHLBECKER, 1986). ☞ *Def. 9, p. 33*

adaptatividade

Manifestação do fenômeno no qual um dispositivo modifica seu próprio comportamento espontaneamente, em resposta ao seu histórico de operação e aos dados de entrada (JOSÉ NETO, 1993, 1994, 2001). ☞ *Def. 7, p. 30*

alfabeto

Conjunto finito e não-vazio de símbolos (HOPCROFT; ULLMAN; MOTWANI, 2003; RAMOS; JOSÉ NETO; VEGA, 2009). ☞ *Def. 11, p. 34*

autômato adaptativo

Extensão do formalismo do autômato de pilha estruturado que permite o reconhecimento de linguagens recursivamente enumeráveis. O termo *adaptativo*, neste contexto, pode ser definido como a capacidade de um dispositivo em alterar seu comportamento de forma espontânea. ☞ *Def. 34, p. 45*

autômato de pilha estruturado

Tipo de autômato de pilha formado por máquinas de estados finitos mutuamente recursivas, a cada um dos quais cabe a tarefa de efetuar o reconhecimento de uma das diferentes classes de subcadeias que compõem uma cadeia de entrada em análise (RAMOS; JOSÉ NETO; VEGA, 2009). ☞ *Def. 29, p. 43*

cadeia

Conjunto finito e não-vazio de símbolos (SIPSER, 2012; HOPCROFT; ULLMAN; MOTWANI, 2003). ☞ *Def. 12, p. 34*

escopo

Região de um programa no qual associações (do inglês *bindings*) entre nomes e variáveis são válidas. ☞ *Def. 48, p. 57*

expansão ansiosa

Estratégia que consiste em expandir todas as ocorrências de macros internas em tempo de definição da macro mais externa (BRABRAND; SCHWARTZBACH, 2002; BRABRAND; MØLLER; SCHWARTZBACH, 2002). ☞ *Def. 53, p. 58*

expansão de múltiplos passos

Estratégia de aplicar múltiplos passos de reescrita na expansão de instâncias de macros, tal que eventuais macros resultantes do processo de reescrita sejam expandidas sequencialmente até que cadeia de saída não contenha mais ocorrências. ☞ *Def. 61, p. 60*

expansão de um passo

estratégia de aplicar apenas um passo de reescrita na expansão de instâncias de macros. Eventualmente, outras macros podem resultar deste processo de reescrita, sendo reproduzidas literalmente (isto é, sem expansão) na cadeia de saída. ☞ *Def. 60, p. 60*

expansão mais externa

Estratégia que consiste em expandir ocorrências de macros internas tão somente no momento de uso da captura da macro mais externa na sequência de transformação (BRABRAND; SCHWARTZBACH, 2002). ☞ *Def. 56, p. 58*

expansão mais interna

Estratégia que consiste em expandir toda e qualquer ocorrência de macro existente em cada parâmetro da macro mais externa até que a sequência resultante não apresente mais ocorrências de macros, mesmo que a captura em questão não seja utilizada na sequência de transformação (BRABRAND; SCHWARTZBACH, 2002). ☞ *Def. 55, p. 58*

expansão preguiçosa

Estratégia que consiste em expandir ocorrências de macros internas somente no momento de chamada da macro mais externa (BRABRAND; SCHWARTZBACH, 2002). ☞ *Def. 54, p. 58*

gerador

Tipo especial de variável preenchida uma única vez, ao início da execução da função adaptativa com valores unívocos que referenciam estados recém-incorporados ao modelo do autômato. ☞ *Def. 37, p. 46*

gramática

Sistema formal no qual é possível sintetizar, de forma exaustiva, através de regras de substituição, o conjunto das cadeias que compõem uma determinada linguagem (RAMOS; JOSÉ NETO; VEGA, 2009).

homomorfismo

Substituição em que $h\colon \Sigma_1 \mapsto \Sigma_2^*$, isto é, h é uma função que mapeia cada símbolo de um alfabeto Σ_1 em uma cadeia única contida em uma linguagem Σ_2^* (RAMOS; JOSÉ NETO; VEGA, 2009). ☞ *Def. 18, p. 34*

linguagem

Conjunto arbitrário, finito ou infinito, de cadeias específicas sobre um alfabeto Σ. Se $L \subseteq \Sigma^*$, então L é uma linguagem sobre Σ (HOPCROFT; ULLMAN; MOTWANI, 2003). ☞ *Def. 16, p. 34*

linguagem de macro de propósito específico

Linguagem de macro vinculada a um sistema hospedeiro distinto (KOHLBECKER, 1986). ☞ *Def. 46, p. 57*

linguagem de macro de propósito geral

Linguagem de macro projetada para integração com um sistema hospedeiro qualquer (KOHLBECKER, 1986). ☞ *Def. 45, p. 57*

macro

Padrão sintático associado a uma transformação. Tal padrão pode ser classificado de acordo com as propriedades e características estruturais mais significativas deste, conforme as classes distintas de linguagens definidas na hierarquia de Chomsky (CHOMSKY, 1956, 1957). ☞ *Def. 41, p. 51*

macro com recursão direta

Macro cuja definição contém uma chamada explícita a si mesma. Construtos semânticos que determinem uma condição de parada podem ser utilizados, mas nem sempre há garantias de término da recursão (KOHLBECKER, 1986; BRABRAND; SCHWARTZBACH, 2002). ☞ *Def. 51, p. 58*

macro com recursão indireta

Macro que cria chamadas de macro como parte do resultado de sua própria expansão, e as chamadas de macros assim criadas são processadas em seguida. Em macros com recursão indireta, ao menos uma dessas chamadas de macro criadas refere-se à própria macro em expansão. Não há garantias de término tampouco (BRABRAND; SCHWARTZBACH, 2002). ☞ *Def. 52, p. 58*

macro higiênica

Macro cuja transformação trata da renomeação de ligações (chamadas de *bindings*) em modelos de código (KOHLBECKER et al., 1986). ☞ *Def. 58, p. 59*

macro textual

Abreviatura aplicada em textos, abstraindo fragmentos e representando-os por construtos equivalentes (KOHLBECKER, 1986). ☞ *Def. 25, p. 36*

mecanismo de abstração

Nome dado à forma de representação e tratamento de abstrações (CAMPBELL, 1987). ☞ *Def. 2, p. 27*

motor de eventos

Modelo de implementação para aplicações fundamentadas na simulação dirigida por estímulos e respostas correspondentes (TANENBAUM; BOS, 2014).

notação de Wirth

Metalinguagem para para descrição sintática de aproximações livres de contexto de linguagens de programação (WIRTH, 1977). ☞ *Def. 28, p. 41*

opacidade referencial

Característica de uma unidade que não é referencialmente transparente (FØLLESDAL, 2009; SØNDERGAARD; SESTOFT, 1990). ☞ *Def. 4, p. 29*

ponto léxico

Par ordenado (x, y), tal que $x, y \in \mathbb{N}$ representam, respectivamente, os índices de início e término do padrão sintático em um texto. ☞ *Def. 47, p. 57*

pré-varredura de parâmetro

Estratégia semelhante à expansão mais interna, com a adição de um processamento da sequência de transformação em busca de ocorrências de macros que, eventualmente, tenham sido produzidas como resultado da expansão de outras macros (BRABRAND; MØLLER; SCHWARTZBACH, 2002). ☞ *Def. 57, p. 59*

primitiva

Construto elementar (ou seja, pré-implementado, que dispensa uma definição a partir de outros conceitos mais elementares) embutido em uma linguagem de macro (KOHLBECKER, 1986). ☞ *Def. 44, p. 56*

regras de escopo de dois passos

Regras de resolução das definições de macros. No primeiro passo, o expansor coleta todas as definições em um conjunto e, no segundo passo, todas as ocorrências de chamadas de macros são efetivamente expandidas (BURMAKO, 2012). ☞ *Def. 50, p. 57*

regras de escopo de um passo

Regras de resolução das definições de macros que torna uma macro visível na sequência de símbolos a partir de seu ponto léxico de definição em diante (KOHLBECKER, 1986). ☞ *Def. 49, p. 57*

símbolo

Representação gráfica única e indivisível em um determinado nível de granularidade de uma entidade abstrata (RAMOS; JOSÉ NETO; VEGA, 2009; SIPSER, 2012). ☞ *Def. 10, p. 33*

sistema abstrato de redução

Estrutura $R = (A, I)$, no qual A é um conjunto de elementos e I é uma sequência de relações binárias $\rightarrow_\alpha$ sobre A, também chamadas de relações de redução ou reescrita (KLOP, 1992). ☞ *Def. 8, p. 32*

submáquina do autômato adaptativo

Elemento pertencente ao conjunto de máquinas de estados finitos mutuamente recursivas do autômato adaptativo que efetua o reconhecimento de uma das diferentes classes de subcadeias que compõem uma cadeia de entrada em análise. ☞ *Def. 35, p. 45*

submáquina do autômato de pilha estruturado

Elemento pertencente ao conjunto de máquinas de estados finitos mutuamente recursivas do autômato de pilha estruturado que efetua o reconhecimento de uma das diferentes classes de subcadeias que compõem uma cadeia de entrada em análise. ☞ *Def. 30, p. 43*

substituição

Função s que mapeia os elementos de um alfabeto Σ_1 em linguagens sobre um alfabeto Σ_2, tal que $s\colon \Sigma_1 \mapsto 2^{\Sigma_2^*}$. ☞ *Def. 17, p. 34*

transformação

Aplicação de um sistema de reescrita de termos R sobre uma sequência s, tal que $\rho(R, s) = s'$, no qual todos os elementos da sequência resultante s' estão na forma normal. ☞ *Def. 27, p. 37*

transformação algorítmica

Transformação que não é simbólica, isto é, ocorre interpretação ou cálculo das unidades sintáticas da sequência de substituição. ☞ *Def. 43, p. 52*

transformação simbólica

Transformação na qual a sequência de substituição é reproduzida *ipsis litteris*, sem interpretação ou cálculo das unidades sintáticas. ☞ *Def. 42, p. 51*

transparência referencial

Propriedade que garante que a substituição de uma unidade por seu valor correspondente não causa alterações no comportamento do programa (SØNDERGAARD; SESTOFT, 1990). ☞ *Def. 3, p. 28*

unidade recursiva

Unidade que, direta ou indiretamente, chama-se a si mesma, resolvendo partes da tarefa principal e combinando resultados intermediários até o término de todas as chamadas, retornando, enfim, o resultado final. ☞ *Def. 5, p. 29*

variável

Objeto definido no escopo da função adaptativa e utilizado para, uma única vez durante toda a execução de uma ação adaptativa, armazenar valores resultantes de ações adaptativas elementares de inspeção (no momento em que uma variável é preenchida, seu conteúdo fica protegido e não mais recebe alterações até o final da execução da função adaptativa). ☞ *Def. 37, p. 46*