



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Evandro Coan

**UMA FERRAMENTA DE FORMATAÇÃO
PROGRAMÁVEL POR GRAMÁTICAS**

Florianópolis, Santa Catarina – Brasil
2019

Evandro Coan

**UMA FERRAMENTA DE FORMATAÇÃO
PROGRAMÁVEL POR GRAMÁTICAS**

Trabalho de Conclusão de Curso submetido
ao Programa de Graduação em Ciências da
Computação da Universidade Federal de Santa
Catarina para a obtenção do Grau de Bacharel em
Ciências da Computação.

Orientador: Rafael de Santiago, Dr.

Florianópolis, Santa Catarina – Brasil
2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Coan, Evandro

Uma ferramenta de formatação programável por gramáticas
/ Evandro Coan ; orientador, Rafael de Santiago,
2019.
178 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Ciências da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciências da Computação. 2. Formatador do texto. 3.
Gramáticas Livre de Contexto. 4. Embelezador de código
fonte. 5. Impressão bonita. I. Santiago, Rafael de. II.
Universidade Federal de Santa Catarina. Graduação em
Ciências da Computação. III. Título.

Evandro Coan

**UMA FERRAMENTA DE FORMATAÇÃO
PROGRAMÁVEL POR GRAMÁTICAS**

Este(a) Trabalho de Conclusão de Curso foi julgado adequado(a) para obtenção do Título de Bacharel em Ciências da Computação, na área de concentração de Linguagens Formais, e foi aprovado em sua forma final pelo Programa de Graduação em Ciências da Computação do INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 25 de Novembro de 2019.

**José Francisco Danilo De Guadalupe
Correa Fletes, Dr.**
Coordenador do Programa de Graduação
em Ciências da Computação

Banca Examinadora:

Rafael de Santiago, Dr.
Orientador
Universidade Federal de Santa
Catarina – UFSC

Prof. Jerusa Marchi, Dr.
Universidade Federal de Santa Catarina –
UFSC

Prof. Álvaro Junio Pereira Franco, Dr.
Universidade Federal de Santa Catarina –
UFSC

RESUMO

Softwares Formatadores de Código-Fonte atuais, também conhecidos como *Source Code Beautifiers*, são limitados a um conjunto similar, ou mesmo à uma única linguagem de programação, além de muitos serem limitados no que eles podem fazer ao formatar o código-fonte. Nesse contexto, propõe-se uma ferramenta que permita, por meio de gramáticas, a especificação de quais linguagens de programação deseja-se realizar a formatação. Utilizando um analisador já existente, foi desenvolvido uma metagramática utilizando o Analisador Lark e então construído um analisador semântico para a nova metalinguagem. Por fim, dois protótipos de ferramentas foram desenvolvidos sobre a nova metalinguagem. Um formatador de código-fonte e uma ferramenta de adição de cores (também conhecida como *Source Code Highlighters*). Com ambas as ferramentas, é possível trabalhar com qualquer linguagem cuja a gramática foi especificada (seguindo as regras da metalinguagem desenvolvida neste trabalho). Enquanto a ferramenta de adição de cores já pode ser considerada completa (porque o processo de adição de cores em si é simples), a ferramenta de formatação de código-fonte é uma implementação simplificada e no futuro precisará ser completada, para adequar-se propriamente a qualquer processo de formatação de código-fonte.

Palavras-chaves: Formatador do texto. Embelezador de código-fonte. Impressão-bonita. Gramáticas Livre de Contexto. Sintaxe de Linguagens de Programação.

ABSTRACT

Cutting edge Source Code Formatting Softwares, also known as Source Code Beautifiers, are limited to a common set, or even to a single programming language, and many formatters are limited in what they can do. In this context, it is also proposed a new Source Code Formatting Tool, allowing users to input their preferred language grammars. Using the Lark Parser, a metagrammar was developed and then a semantic parser was built for the new metalanguage. Finally, two tool prototypes were developed with the new metalanguage. A source code formatter and a source code highlighter. With both tools, it is possible to work with any language in which its grammar was specified (following the rules of metalanguage developed in this paper). While the color addition tool can already be considered complete (because the color addition process itself is simple), the source code formatting tool is a simplified implementation and in the future will need to be completed, to suit any source code formatting requirements.

Keywords: Text Formatter. Source Code Beautifier. Pretty-printing. Context-free Grammars. Programming Languages Syntax.

LISTA DE FIGURAS

Figura 1	– Processo de Tradução	16
Figura 2	– Árvore do Programa “token sem nome”	19
Figura 3	– Árvore do Programa “token com nome”	20
Figura 4	– Árvore do Programa “token com outro nome”	20
Figura 5	– Árvore da palavra “12”	22
Figura 6	– Árvore da palavra “1”	22
Figura 7	– Árvore da palavra “2”	23
Figura 8	– Árvore da palavra “3”	23
Figura 9	– Hierarquia de Chomsky	25
Figura 10	– Gramáticas Determinísticas <i>versus</i> suas Linguagens	28
Figura 11	– Exemplo da tela de configuração de UniversalIndentGUI	39
Figura 12	– Árvore Sintática do Código 10 criada pelo Código 11	50
Figura 13	– Fluxo de uso comum de um analisador	53
Figura 14	– Uso feito pela nova ferramenta de Formatação de Código	54
Figura 15	– Relacionamentos entre os diferentes públicos deste projeto	56
Figura 16	– Relação entre Metagramáticas, Metacompiladores e Metaprogramas	57
Figura 17	– Exemplo de classificação de código-fonte com múltiplos escopos	60
Figura 18	– Diagrama das principais classes	64
Figura 19	– Árvore Sintática “main_formatter_syntax_tree.png”	98
Figura 20	– Árvore Sintática Abstrata “main_formatter_abstract_syntax_tree.png”	99
Figura 21	– Árvore Sintática “main_highlighter_syntax_tree.png”	103
Figura 22	– Árvore Sintática Abstrata “main_highlighter_abstract_syntax_tree.png”	104

LISTA DE QUADROS

Quadro 1 – Exemplo de Ambiguidade Linguística	32
Quadro 2 – Exemplo de Ofuscador de Código	40

LISTA DE CÓDIGOS

Código 1	- Exemplo de gramática utilizada pelo Analisador Lark	18
Código 2	- Exemplo de gramática com uma Estrutura de Sintaxe	21
Código 3	- Trecho do Arquivo de Configuração de Uncrustify	35
Código 4	- Antes do ofuscamento	40
Código 5	- Depois do ofuscamento	40
Código 6	- Exemplo de um arquivo “.sublime-syntax”	43
Código 7	- Exemplo de um arquivo “.sublime-color-scheme”	43
Código 8	- Exemplo de um programa na linguagem Java	47
Código 9	- Exemplo mínimo da metagramática	48
Código 10	- Exemplo de gramática pelas regras da mínima metagramática .	48
Código 11	- Exemplo de uso da metagramática e uma gramática	49
Código 12	- Resultado do reconhecimento do programa Java pela gramática	50
Código 13	- Exemplo de configuração utilizada mínima do Formatador de Código	51
Código 14	- Exemplo mínimo de Formatador de Código	52
Código 15	- Resultado do Formatador de Código para Java	52
Código 16	- Símbolo Inicial da Metagramática “ObjectBeauty”	58
Código 17	- Exemplo de Gramática, Símbolo Inicial	60
Código 18	- Exemplo de Gramática, Contextos	61
Código 19	- Exemplo de Gramática, Grupos de Captura	62
Código 20	- Exemplo de Gramática, Tipos numéricos	62
Código 21	- Exemplo de Gramática, Reconhecimento de Erros	63
Código 22	- Construtor do Analisador Semântico	63
Código 23	- Construtor do Formatador	67
Código 24	- Construtor de “AbstractFormatter”	68
Código 25	- Arquivo “source/requirements.txt”	91
Código 26	- Arquivo “source/main_formatter.py”	92
Código 27	- Arquivo “source/main_highlighter.py”	94
Código 28	- Arquivo “source/utilities.py”	96
Código 29	- Arquivo HTML gerado pelo programa de exemplo “main_formatter.py”	97
Código 30	- Resultado da execução do arquivo “source/main_formatter.py” .	97
Código 31	- Arquivo HTML gerado pelo programa de exemplo “main_highlighter.py”	102
Código 32	- Resultado da execução do arquivo “source/main_highlighter.py”	102
Código 33	- Resultado da execução dos Testes de Unidade	106
Código 34	- Arquivo “source/unit_tests.py”	106
Código 35	- Arquivo “source/semantic_analyzer.py”	131

Código 36 – Arquivo “source/code_highlighter.py”	146
Código 37 – Arquivo “source/code_formatter.py”	156
Código 38 – Arquivo “source/grammars_grammar.pushdown”	166

SUMÁRIO

1	INTRODUÇÃO	12
1.1	MOTIVAÇÃO	13
1.2	OBJETIVOS	14
1.2.1	Objetivo Geral	14
1.2.2	Objetivos Específicos	14
1.3	MÉTODO DE PESQUISA	14
1.4	ESTRUTURAÇÃO DO TEXTO	15
2	FUNDAMENTAÇÃO TEÓRICA	16
2.1	COMPILADORES E TRADUTORES	16
2.2	GRAMÁTICAS	17
2.2.1	Hierarquia de Chomsky	23
2.2.2	Gramáticas Regulares	24
2.2.3	Gramáticas Livres de Contexto	26
2.2.4	Gramáticas Sensíveis ao Contexto	26
2.2.5	Gramáticas Irrestritas	26
2.3	ANALISADORES SINTÁTICOS	27
2.3.1	Gramáticas <i>versus</i> Linguagens	27
2.3.2	Reduções e Derivações	29
2.3.3	Analisadores LR(K)	30
2.3.4	Análise Semântica	31
2.3.5	Alterações nos Analisadores Sintáticos	33
2.4	COMPILADORES E CLASSES DE COMPLEXIDADE	34
3	ESTADO DA ARTE	35
3.1	FORMATADORES DE CÓDIGO	35
3.2	QUAL A UTILIDADE DE FORMATADORES?	37
3.3	TRABALHOS RELACIONADOS	38
3.4	OFUSCADORES	40
3.5	ADIÇÃO DE CORES	41
3.5.1	Gramáticas	42
4	FORMATADOR DESENVOLVIDO	46
4.1	VISÃO GERAL	47
4.2	INTRODUÇÃO À METAGRAMÁTICA	53
4.3	ESPECIFICAÇÃO DA METALINGUAGEM	58
4.4	ANALISADOR SEMÂNTICO	63
4.5	FORMATADOR DE CÓDIGO	66

5	CONCLUSÃO	71
5.1	COMPARAÇÃO COM OUTROS TRABALHOS	72
5.2	TRABALHOS FUTUROS	73
	REFERÊNCIAS	76
	APÊNDICE A – MANUAL DO FORMATADOR	90
	APÊNDICE B – MAIN_FORMATTER.PY	97
	APÊNDICE C – MAIN_HIGHLIGHTER.PY	102
	APÊNDICE D – TESTES DE UNIDADE	106
	APÊNDICE E – ANALISADOR SEMÂNTICO	130
	APÊNDICE F – CÓDIGO DE ADIÇÃO DE CORES	146
	APÊNDICE G – CÓDIGO DO FORMATADOR	156
	APÊNDICE H – CÓDIGO DA METAGRAMÁTICA	166
	APÊNDICE I – ARTIGO SOBRE O TCC	169

1 INTRODUÇÃO

Com a criação de sistemas computacionais cada vez mais complexos, torna-se um desafio ter, em um único projeto, diversos programadores, pois cada um possui características pessoais distintas, e entende melhor o código-fonte escrito de acordo com seu costume, já que previamente sabe como localizar-se e entender melhor os seus elementos (JBARA; FEITELSON, 2015).

Por exemplo, quanto a indentação, há programadores que preferem usar 2, 4, ou 8 espaços; quanto legibilidade, há programadores que preferem o uso de linhas em branco antes de estruturas de controle (ALLAMANIS; BARR; SUTTON, 2014). Quando diferentes programadores cooperam em grandes projetos, como unir pessoas com experiências e preferências diferentes? Uma barreira adicional ao desenvolvimento do projeto será concluir um trabalho com diferentes colegas de equipe, dado que a tarefa de codificação é complexa e as etapas de projeto podem ser realizadas de diferentes formas.

Uma dessas diferenças é a maneira na qual cada programador organiza a estrutura do código-fonte de acordo com seu estilo de formatação (BAGGE *et al.*, 2003). Por exemplo, imagine que um certo “Programador A” tem o costume de deixar uma linha em branco antes de cada estrutura de controle. Já um outro “Programador B”, possui o costume oposto, ele jamais deixa uma linha em branco antes dessas estruturas de controle. Então, quais problemas podem acontecer quando ambos os programadores “A” e “B” trabalham em um mesmo projeto?

O problema mais provável é que o código-fonte esteja escrito em dois estilos: do Programador A e B. Então, que tipos de problema isso poderia causar? Uma vez que o código-fonte não está mais todo escrito sobre um mesmo padrão, pode haver uma dificuldade na leitura do código-fonte como algo homogêneo. Os próprios Programadores A e B, autores do código-fonte, teriam dificuldades, pois o código-fonte segue um estilo diferente ao que A e B estão habituados. Um exemplo de padrão que estaria prejudicado seria o de rapidamente identificar onde pode estar uma estrutura de controle, de acordo com a presença ou não de uma linha em branco antes dela, já que a presença de uma linha em branco faz um destaque maior.

Indo além de como programadores leem código-fonte enquanto estão escrevendo o código-fonte, também tem que se preocupar como o código-fonte será salvo no sistema de arquivos, em seu formato de texto simples. Já que para compartilhar código-fonte e trabalhar em times de forma eficiente, é essencial utilizar um sistema de controle de versões (versionamento) (OLEVSKY, 2013). Tal ferramenta deve permitir rastrear mudanças (LASTER, 2016), habilitando que se entenda melhor as atividades de cada programador (YENER; DUNDAR, 2016b). Permitindo que gerentes de projetos e os próprios programadores, tenham o controle das (e de suas) mudanças no código-fonte

(ROSSO; JACKSON, 2016).

Há duas razões principais para impor um formato de código-fonte único em um projeto. A primeira razão tem a ver com o controle de versão: quando todo mundo formata o código-fonte de forma idêntica, todas as alterações nos arquivos têm a garantia de terem algum significado. Nada mais de coisas como adicionar ou remover um espaço aleatoriamente, muito menos formatar um arquivo inteiro como um “efeito colateral” de alterar apenas uma linha ou duas (GEUKENS, 2013, tradução nossa)¹.

Ao trabalhar com um sistema de controle de versões como *git*, recomenda-se manter todo o código-fonte sobre um único estilo, devido a problemas que podem ser gerados ao permitir que cada programador escreva códigos-fonte livremente em seu estilo. Ao permitir que cada programador escreva códigos-fonte livremente em seu estilo, isso pode gerar problemas como ruídos de formatação, dificultando a revisão do código-fonte, uma vez que cada programador precisa entender diferentes estilos de formatação aplicados em um único arquivo ou projeto (ARNDT; RADTKE, 2016).

Caso todos os programadores decidam re-escrever o histórico, fazendo mudanças insignificantes como inserindo novas linhas antes de cada “if”, os programadores precisam dedicar tempo de projeto para formatar o código, o que poderia ser evitado e utilizado para escrever código-fonte. No fim, o histórico *git* irá conter ruídos desnecessários. Um bom uso de um sistema de controle de versões é somente conter mudanças que sejam significativas, como criações de novas funções, correção de bugs (BENDIK; BENES; CERNA, 2017; STEINERT *et al.*, 2009), etc. Ferramentas de controle de versões auxiliam na manutenção, rastreabilidade de responsabilidades e alterações, mas não apoiam na manutenção de um estilo de formatação único para uma equipe de programadores.

Assim, este trabalho tem por objetivo estudar e desenvolver uma ferramenta de reorganização do código-fonte nos aspectos estruturais ou sintáticos (veja o último exemplo da Seção 2.2: Gramáticas). Estas ferramentas também são conhecidas como “*Source Code Beautifiers*” ou “Formatadores de Código”. Com isso, espera-se que o desenvolvimento de códigos-fonte demande menos trabalho, pois o desenvolvedor pode focar mais seus esforços pensando sobre o problema que está sendo resolvido, ao contrário de tentar decifrar o que está escrito em estilo incomum a sua experiência (ATKINS *et al.*, 2002).

1.1 MOTIVAÇÃO

Ter que aprender e configurar muitas ferramentas de formatação de código-fonte é um tarefa cansativa (Seção 3.1: Formatadores de código). Mais ainda, é escrever

¹ I'd say there are two main reasons to enforce a single code format in a project. First has to do with version control: with everybody formatting the code identically, all changes in the files are guaranteed to be meaningful. No more just adding or removing a space here or there, let alone reformatting an entire file as a “side effect” of actually changing just a line or two.

uma nova ferramenta de formatação de código-fonte para cada nova linguagem de programação que surgir. Um programador da linguagem C++, pode utilizar formatador de código-fonte que oferece até 600 opções de configuração. Mas este programador, ao migrar para uma linguagem como Python, pode somente encontrar um formatador que suporta no máximo 20 opções de configuração. O que pode tornar frustrante migrar de uma linguagem de programação para outra, pela perda de controle da ferramenta de formatação de código-fonte.

1.2 OBJETIVOS

Esta seção apresenta sucintamente os objetivos deste trabalho, divididos em Objetivo Geral e Objetivos Específicos.

1.2.1 Objetivo Geral

Desenvolver um formatador de código-fonte, extensível a diversas linguagens de programação por meio de uma nova definição gramáticas, especificada por uma metagramática.

1.2.2 Objetivos Específicos

1. Analisar a fundamentação teórica de linguagens formais e compiladores;
2. Analisar o estado da arte das ferramentas que fazem formatação de código-fonte;
3. A partir dos pontos fracos e fortes do estado da arte de formatadores, propor uma nova ferramenta de formatação de código-fonte.
4. Avaliar como esta nova ferramenta difere das demais já existentes.

Destes objetivos específicos, todos eles foram alcançados como pode ser observado dos Capítulos 2 a 5: Fundamentação Teórica, Estado da Arte, Formatador Desenvolvido e Conclusão.

1.3 MÉTODO DE PESQUISA

Foram realizados pesquisas em artigos científicos, livros e sites para obter-se referências e resolver dúvidas pendentes sobre quaisquer conteúdos. Foi utilizado como guia as notas de aula retiradas das disciplinas relacionadas à construção deste trabalho. Como palavras-chave de pesquisa para encontrar trabalhos relacionados, foi utilizado termos como: 1) *pretty-printing*; 2) *beautifying*; 3) *source code formatting* e; 4) *source code highlighting*. Estes termos foram utilizados em sites como: 1) <https://dl.acm.org/>; 2) <https://ieeexplore.ieee.org> e; 3) <https://arxiv.org/>.

Foi escolhido explicar sobre um formatador de código-fonte configurável por arquivos de texto e outro configurável por uma interface gráfica com o usuário porque estes dois tipos de exemplo definem bem a classe de formatadores. Isto é, não existem grandes diferenças de uso e implementação entre estes dois formatadores e os demais.

Por fim, é proposto um novo formatador de código-fonte configurável por arquivos de texto e extensível para novas linguagem de programação através da especificação de suas gramáticas, utilizando o Analisador Lark (SHINAN, 2014). Para que as gramáticas de novas linguagens de programação possam ser interpretadas, também foi proposto uma nova metagramática (SCHILD; HERZOG, 1993). Esta nova metalinguagem (BOOK; SHORRE; SHERMAN, 1970) é então utilizada pela nova ferramenta de formatação para realizar as alterações nos códigos-fonte escolhidos pelo usuário. Para construção desta ferramenta, foi utilizado a linguagem Python e testes de unidade e integração (YENER; DUNDAR, 2016a).

1.4 ESTRUTURAÇÃO DO TEXTO

A criação deste trabalho dividi-se em três áreas distintas. No Capítulo 2: Fundamentação Teórica, é realizado uma fundamentação teórica sobre que são gramáticas na teoria da computação e os conceitos necessários para desenvolvimento de compiladores. No Capítulo 3: Estado da Arte, é explicado sobre dois tipos distintos de formatadores de código-fonte. Um formatador de código-fonte configurável por arquivos de texto (Uncrustify) e outro configurável por uma interface gráfica com o usuário (UniversalIndentGUI). Também explicado um pouco sobre as ferramentas de adição de cores ao código-fonte sendo visualizado, utilizadas por editores de texto.

No Capítulo 4: Formatador Desenvolvido, é apresentado a proposta e implementação de uma ferramenta de formatação de código-fonte. Por fim, no Capítulo 5: Conclusão, é apresentado quais serão os desafios e trabalhos futuros que podem ser feitos sobre esta nova ferramenta, e como ela difere das demais ferramentas de formatação de código-fonte. No Apêndice A: Manual do Formatador, é apresentado como utilizar a implementação realizado neste trabalho, seja para adicionar cores ou formatar código-fonte.

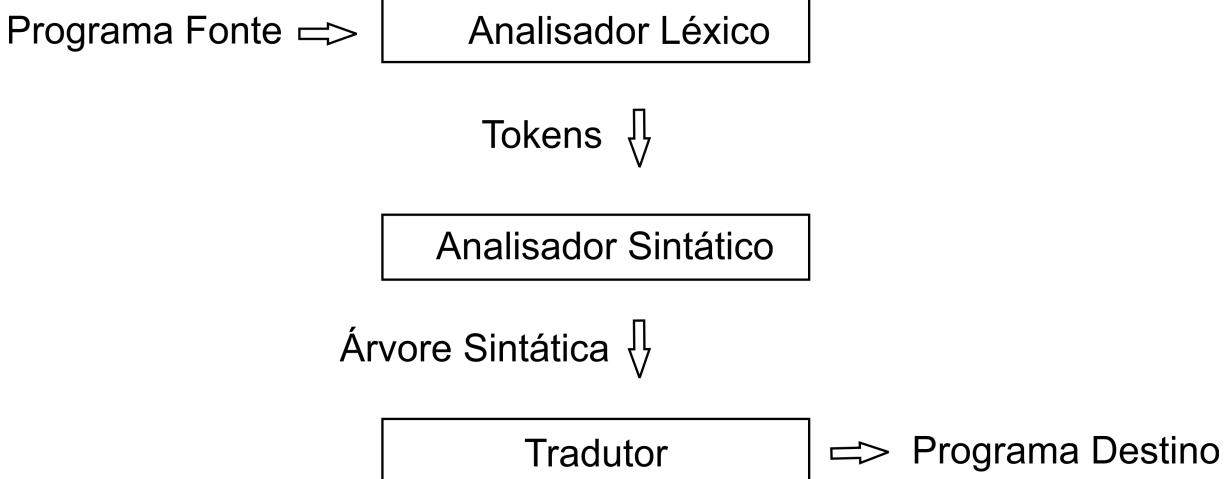
2 FUNDAMENTAÇÃO TEÓRICA

De acordo com o funcionamento de muitos Formatadores de Código (Capítulo 3: Estado da Arte) e da ferramenta que foi desenvolvida (Capítulo 4: Formatador Desenvolvido), foi explicado resumidamente toda a teoria necessária para construção de compiladores, incluindo as classes de complexidade para medidas da eficiência dos compiladores. Seguindo um currículo acadêmico de um curso de Ciências da Computação, este resumo seria o equivalente as disciplinas de: 1) Teoria da Computação; 2) Linguagens Formais, e; 3) Construção de Compiladores.

2.1 COMPILADORES E TRADUTORES

Em linguagens formais, tradutores são ferramentas que operam realizando a transformação de um programa de entrada, em um programa de saída (MURPHREE; FENVES, 1970). Diferente de um compilador, a linguagem de destino da “tradução” é do **mesmo nível** que a linguagem de origem. Por exemplo, dado um programa de entrada em C++ e um programa de saída em Java, tem-se um processo de tradução (Figura 1). A tradução é diferente de um processo de compilação, que é dotado de mais etapas (AI HUA WU; PAQUET, 2004).

Figura 1 – Processo de Tradução



Fonte: Própria, baseado em Aho, Lam *et al.* (2006)

No processo de compilação ou tradução, um Analisador Léxico cria múltiplos *tokens*. Um *token* é composto por diversos atributos como a posição e o *lexema*, i.e., a sequência de caracteres que este *token* representa no programa de entrada. O *lexema* também pode ser conhecido como os símbolos terminais da gramática. Uma vez que o programa é “*tokenizado*” pelo Analisador Léxico, o Analisador Sintático pode construir a Árvore Sintática do programa.

Utilizando a Árvore Sintática do programa de entrada, o tradutor constrói uma nova Árvore Sintática correspondente a Árvore Sintática da linguagem do programa destino, utilizada para construir o código-fonte do programa destino. Em um processo de compilação, não é necessário criar uma nova Árvore Sintática como no processo de tradução, mas sim a geração de código objeto ou binário (AHO; LAM *et al.*, 2006).

Analisadores Sintáticos podem ser Ascendentes¹ ou Descendentes². Devido a essa característica ambos possuem as suas vantagens e desvantagens. Um Analisador Ascendente realiza a construção da Árvore Sintática das folhas até a raiz, o contrário de um Analisador Descendente que realiza a construção da Árvore Sintática a partir da raiz até as folhas³.

Uma vantagem de um Analisador Ascendente é o suporte de uma maior classe de Gramáticas Determinísticas Livres de Contexto. Uma vantagem de um Analisador Descendente é a facilidade da recuperação de erros em relação aos Analisadores Ascendentes⁴ (SIPPU; SOISALON-SOININEN, 1982; SORKIN; DONOVAN, 2011; DAIN, 1985; CERECKE, 2002; GROSCH, 2005; CORCHUELO *et al.*, 2002). Existem tanto linguagens que Analisadores Ascendentes reconhecem, mas que Analisadores Descendentes não reconhecem e vice-versa (Figura 10: Gramáticas Determinísticas *versus* suas Linguagens).

2.2 GRAMÁTICAS

Gramáticas são conjuntos de regras que definem uma linguagem. Uma gramática é definida por quatro componentes, como serão apresentados a seguir. Para estes componentes, considere que os símbolos: 1) V_n representa o conjunto de não-terminais; 2) V_t representa o conjunto de terminais e; 3) $V = V_n \cup V_t$.

- O conjunto V_t de símbolos terminais (também chamados de *tokens* ou símbolos do alfabeto da linguagem). Cada terminal corresponde a um símbolo presente na linguagem. Durante a Análise Léxica, os símbolos terminais serão utilizados para definir os lexemas que são a base principal dos *tokens*. Na composição da Árvore Sintática, os “*tokens*” ou símbolos terminais, serão usualmente as folhas da Árvore Sintática (a não ser em casos específicos, como por exemplo, quando a gramática possui símbolos inúteis (HOPCROFT; MOTWANI; ULLMAN, 2006));

¹ Do inglês, *Bottom-Up*.

² Do inglês, *Top-Down*.

³ Como pode ser observado em seus nomes, ambos os analisadores tanto da família LL (Descendentes, *Left-to-right, Leftmost derivation*) ou LR (Ascendentes, *Left-to-right, Rightmost derivation*) fazem a leitura do programa de entrada da esquerda para a direita.

⁴ Conceito abordado na Seção 2.3: Analisadores Sintáticos.

2. O conjunto V_n de símbolos não-terminais (algumas vezes chamados de “variáveis sintáticas”), servem para agrupar vários não-terminais e/ou terminais. Na composição da Árvore Sintática, os símbolos não-terminais usualmente serão os nós internos da Árvore Sintática⁵. Como restrição, para evitar ambiguidades entre quais são os símbolos terminais e não-terminais, a intersecção entre o conjunto de símbolos terminais e não-terminais é sempre vazia, i.e., $V_n \cap V_t = \emptyset$;
3. Um conjunto de produções P . Uma produção consiste em uma dupla elementos. O primeiro elemento é a cabeça ou lado esquerdo e representa a substituição ou consumo que será feito no programa de entrada. O segundo elemento é obrigatoriamente constituído no mínimo um símbolo não-terminal e zero ou mais terminais ou não-terminais. O segundo elemento é a cauda ou lado direito da produção, composto de terminais e/ou não-terminais. Formalmente define-se o conjunto de produções de uma gramática pela seguinte regra, onde “*” representa o operador de fechamento do conjunto (HOPCROFT; MOTWANI; ULLMAN, 2006). Independente do tipo de gramática, está é a definição formal do que é uma gramática em teoria da computação e linguagens formais, sendo o tipo mais genérico de gramática:

$$P = \{ \alpha ::= \beta \mid \alpha \in V^*V_nV^* \wedge \beta \in V^* \}$$

4. Um símbolo inicial selecionado a partir do conjunto de símbolos não-terminais. O símbolo inicial é utilizado para definir qual será a raiz da Árvore Sintática, e.g., o símbolo inicial participará de uma das últimas regras de produção utilizada para terminar o reconhecimento do programa de entrada em um Analisador Ascendente, e também participará de uma das primeiras regras de produção utilizada em um Analisador Descendente ou gerar-se palavras desta linguagem.

No Código 1, é possível ver um exemplo real de gramática aceito pelo Analisador Lark. Esta é uma gramática de uma linguagem finita que aceita 3 palavras: 1) “token sem nome”; 2) “token com nome” e; 3) “token com outro nome”. No Analisador Lark, quando um token é declarado diretamente como “token sem nome”, ele irá ser descartado da Árvore Sintática. Como pode ser visto na Figura 2, a árvore sintática do programa “token sem nome”, irá conter somente o símbolo inicial da gramática. Nas Figuras 2 a 4, pode ser visto a Árvore Sintática de cada uma das 3 palavras da linguagem do Código 1.

Código 1 – Exemplo de gramática utilizada pelo Analisador Lark

⁵ Por exemplo, quando a gramática da linguagem não contém símbolos inúteis, i.e., todos os símbolos da gramática são fértiles e permitem a geração de palavras além do conjunto vazio \emptyset (HOPCROFT; MOTWANI; ULLMAN, 2006).

```

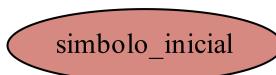
1 import pushdown
2
3 parser = pushdown.Lark(
4     """
5         simbolo_inicial: "token sem nome" | TOKEN_COM_NOME | nodo_da_arvore
6         TOKEN_COM_NOME: "token com nome"
7         nodo_da_arvore: TOKEN_COM_OUTRO_NOME
8         TOKEN_COM_OUTRO_NOME: "token com outro nome"
9     """,
10    start='simbolo_inicial',
11    parser="lalr",
12 )
13
14 def parseit(program, filename):
15     pushdown.tree.pydot__tree_to_png(
16         parser.parse( program ), filename, "TB", debug=1, dpi=600 )
17
18 parseit( 'token sem nome', "token_sem_nome.png" )
19 parseit( 'token com nome', "token_com_nome.png" )
20 parseit( 'token com outro nome', "token_com_outro_nome.png" )

```

A sintaxe da gramática utilizada no Código 1, é uma adaptação especial do padrão EBNF⁶ (PATTIS, 1994; PARR, 2013; SHINAN, 2019b; 2018) que o Analisador Lark faz. Símbolos que possuem todas as letras em “MAIÚSCULA” definem os símbolos terminais da gramática que aparecerão na Árvore Sintática. Enquanto símbolos escritos todos em letras “minúsculas” definem os símbolos não-terminais da gramática.

A Figura 2, representa a Árvore Sintática completa da palavra “token sem nome”, que a linguagem da gramática do Código 1 aceita. Como pode ser facilmente percebido, ela é uma Árvore Sintática estranha. Cadê o *token*? O *token* da árvore foi descartado pelo Analisador Lark, porque ele não está dentro de uma produção de *token* com nome em letras “MAIÚSCULAS”. Este é um recurso do Analisador Lark, que permite descartarmos da Árvore Sintática *tokens* que julgam-se desnecessários aparecem na Árvore Sintática.

Figura 2 – Árvore do Programa “token sem nome”

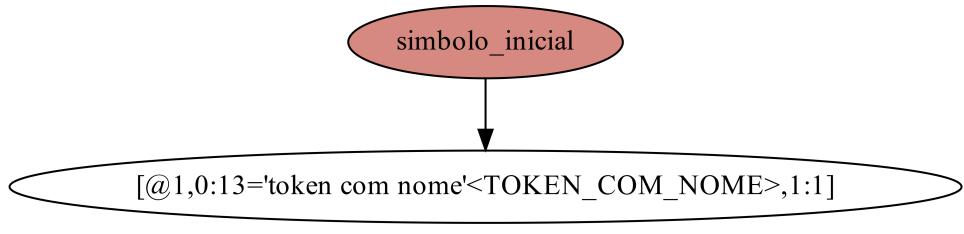


Fonte: Própria

⁶ Do inglês, *Extended Backus–Naur Form* uma extensão do padrão BNF (*Backus–Naur Form*).

A Figura 3, representa a Árvore Sintática completa da palavra “token com nome”, que a linguagem da gramática do Código 1 aceita. Diferente da Árvore Sintática da Figura 3, a Árvore Sintática da Figura 3 possui como nó folha a palavra “token com nome”. Porque na gramática da linguagem, foi criado especificamente a produção “TOKEN_COM_NOME : “token com nome””, fazendo com que o Analisador Lark não descarte *token* da Árvore Sintática.

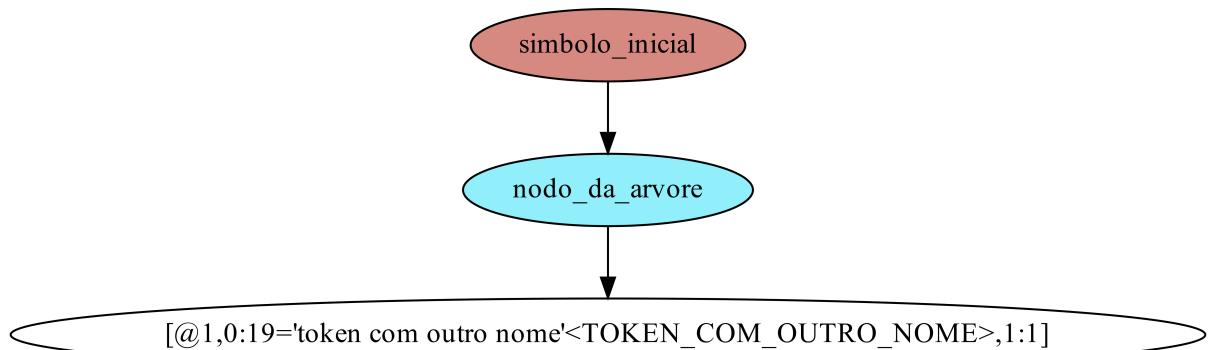
Figura 3 – Árvore do Programa “token com nome”



Fonte: Própria

A Figura 4, representa a Árvore Sintática completa da palavra “token com outro nome”, que a linguagem da gramática do Código 1 aceita. Diferente da Árvore Sintática das Figuras 2 e 3, a Árvore Sintática da Figura 4, apresenta um nó intermediário chamado de “nodo_da_arvore”. O nó intermediário “nodo_da_arvore” é um exemplo de símbolo não-terminal da gramática. Usualmente, os símbolos não-terminais não são nós-folhas da Árvore Sintática. Entretanto, caso a gramática de entrada possua símbolos inúteis (HOPCROFT; MOTWANI; ULLMAN, 2006), podem existir ramificações da Árvore Sintática na qual símbolos não-terminais são nós-folhas. Reveja também a Figura 2, nela temos outro exemplo onde um não-terminal foi um “nó-folha” da Árvore Sintática.

Figura 4 – Árvore do Programa “token com outro nome”



Fonte: Própria

No Código 2, pode ser encontrado outro exemplo de gramática aceita pelo Analisador Lark. Nas Figuras 5 a 8, pode ser encontrado as 4 palavras da linguagem

que esta gramática aceita. Nas Figuras 6 a 8, pode-se ver as palavras da linguagem (gramática do Código 2), na qual os *tokens* correspondem exatamente aos símbolos do alfabeto da linguagem. Já a Figura 5, mostra-se como o conjunto de produções da gramática pode ser utilizado para montar a estrutura (sintaxe) da linguagem, combinando os símbolos do alfabeto da linguagem.

Código 2 – Exemplo de gramática com uma Estrutura de Sintaxe

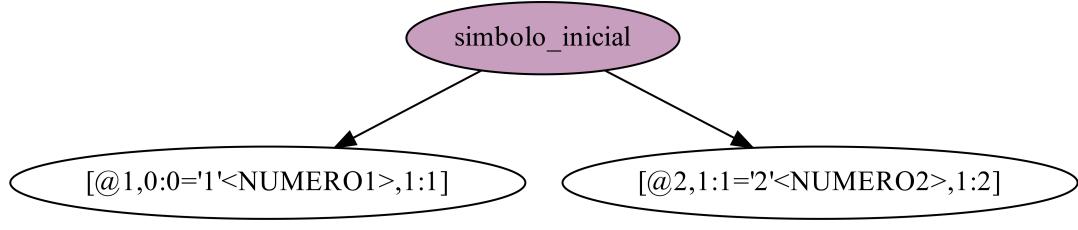
```

1 import pushdown
2
3 parser = pushdown.Lark(
4     r"""
5         simbolo_inicial: NUMERO1 NUMERO2 |  NUMERO3  | nao_terminal
6         nao_terminal: NUMERO1 | NUMERO2
7         NUMERO1: "1"
8         NUMERO2: "2"
9         NUMERO3: "3"
10        """,
11        start='simbolo_inicial',
12        parser="lalr",
13    )
14
15 def parseit(program, filename):
16     pushdown.tree.pydot__tree_to_png(
17         parser.parse( program ), filename, "TB", debug=1, dpi=600 )
18
19 parseit( '1', "palavra1.png" )
20 parseit( '2', "palavra2.png" )
21 parseit( '3', "palavra3.png" )
22 parseit( '12', "palavra12.png" )

```

A Figura 5, representa a Árvore Sintática completa da palavra “12”, que a linguagem da gramática do Código 2 aceita. A Figura 5, é um bom exemplo para demonstrar a diferença entre sintaxe e semântica. Gramáticas Livre de Contexto somente representam a estrutura de um programa (ou palavra) de uma linguagem. O que é mostrado na Figura 5, é a estrutura da palavra “12”, isto é, como os símbolos “1” e “2” do alfabeto da linguagem são estruturados. Já semanticamente (o significado), da estrutura formada pelos símbolos “1” e “2” do alfabeto, pode ser interpretado como o número “12” (doze), representado estruturalmente pela Árvore Sintática da Figura 5.

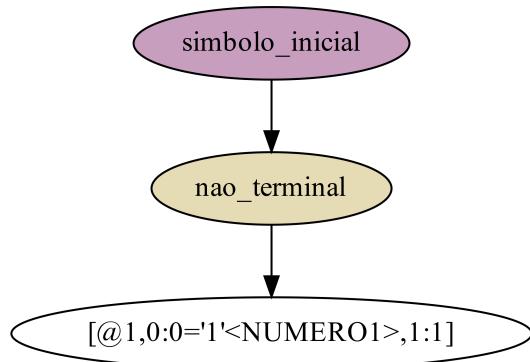
Figura 5 – Árvore da palavra “12”



Fonte: Própria

A Figura 6, representa a Árvore Sintática completa da palavra “1”, que a linguagem da gramática do Código 2 aceita. A Árvore Sintática da Figura 6, apresenta a definição de um *token* com o nome “NUMERO1” e o um uso de um único símbolo do alfabeto, o símbolo “1” (implicitamente derivado a partir das produções da gramática pelo Analisador Lark).

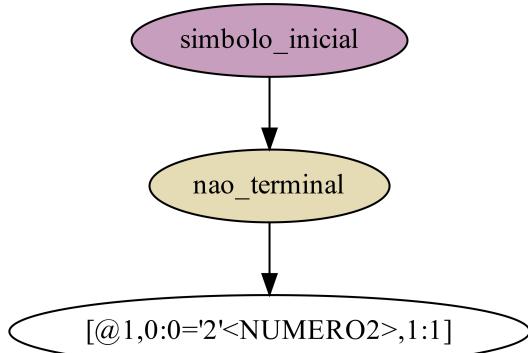
Figura 6 – Árvore da palavra “1”



Fonte: Própria

A Figura 7, representa a Árvore Sintática completa da palavra “2”, que a linguagem da gramática do Código 2 aceita. A Árvore Sintática da Figura 7, apresenta a definição de um *token* com o nome “NUMERO2” e o um uso de um único símbolo do alfabeto, o símbolo “2” (implicitamente derivado a partir das produções da gramática pelo Analisador Lark).

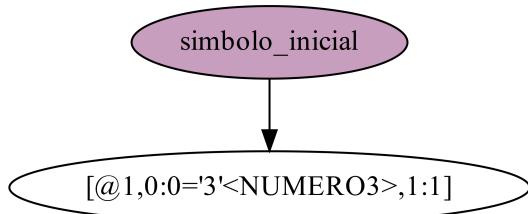
Figura 7 – Árvore da palavra “2”



Fonte: Própria

A Figura 8, representa a Árvore Sintática completa da palavra “3”, que a linguagem da gramática do Código 2 aceita. A Árvore Sintática da Figura 8, apresenta a definição de um *token* com o nome “NUMERO3” e o uso de um único símbolo do alfabeto, o símbolo “3” (implicitamente derivado a partir das produções da gramática pelo Analisador Lark).

Figura 8 – Árvore da palavra “3”



Fonte: Própria

Veja o Apêndice A: Manual do Formatador para uma descrição mais detalhada das informações que apareceram nos *tokens* das Árvores Sintáticas nos exemplos anteriores (Figuras 2 a 8). As gramáticas recém-apresentadas e aceitas pelo Analisador Lark são Gramáticas Livre de Contexto. Na próxima seção será falado um pouco mais sobre os diferentes tipos de gramáticas.

2.2.1 Hierarquia de Chomsky

Todas as gramáticas que existem são no mínimo⁷ Gramáticas Tipo 0 (AHO; ULLMAN, 1972; CHOMSKY, 1956), também conhecidas como Gramáticas Irrestritas

⁷ Caso contrário não serão gramáticas, mas qualquer outra definição na qual a Teoria de Linguagens Formais e Compiladores pode não se aplicar.

porque não possuem nenhuma restrição de complexidade de tempo⁸, como os tipos de gramáticas a serem descritos nas próximas seções. A partir da adição de restrições sobre a definição formal de gramática recém-apresentada, também pode-se compreender a hierarquia de Chomsky (1956), onde uma linguagem pode ser classificada como Regular, Livre de Contexto, Sensível ao Contexto e Irrestrita (Figura 9).

Toda Gramática Regular ou Livre de Contexto, é também uma Gramática Irrestrita ou Sensível ao Contexto, uma vez que Gramáticas Livres de Contexto ou Regulares são um subconjunto das Gramáticas Irrestritas ou Sensíveis ao Contexto, como apresentado na Figura 9. Por isso, também pode-se chamar uma dada Gramática Regular de Irrestrita ou Livre de Contexto. Isto é: 1) qualquer Gramática Regular é Livre de Contexto; 2) qualquer Gramática Livre de Contexto também é Sensível ao Contexto e; 3) qualquer Gramática Sensível ao Contexto também é Irrestrita.

Quando diz-se que existe uma Gramática Livre de Contexto para uma dada linguagem, pode-se ter a impressão de que este é o melhor tipo de gramática, i.e., o tipo mais eficiente em tempo computational⁹ na qual uma dada linguagem pode ser representada. Entretanto, precisa-se tomar cuidado quando se fala sobre gramáticas e linguagens.

Não se pode dizer que uma dada Linguagem é Livre de Contexto simplesmente porque existe uma Gramática Livre de Contexto para dada linguagem. Pois também é preciso que esta gramática seja o tipo mínimo na qual esta linguagem pode ser escrita. Sempre se pode escrever uma gramática menos eficiente do que o tipo mínimo de gramática que uma linguagem pode ser escrita. Para saber se este tipo de gramática é o mínimo, utiliza-se o Lema do Bombeamento¹⁰ para determinar e provar formalmente que dada gramática é o tipo mínimo de gramática (na teoria de linguagens formais e compiladores) para dada linguagem.

2.2.2 Gramáticas Regulares

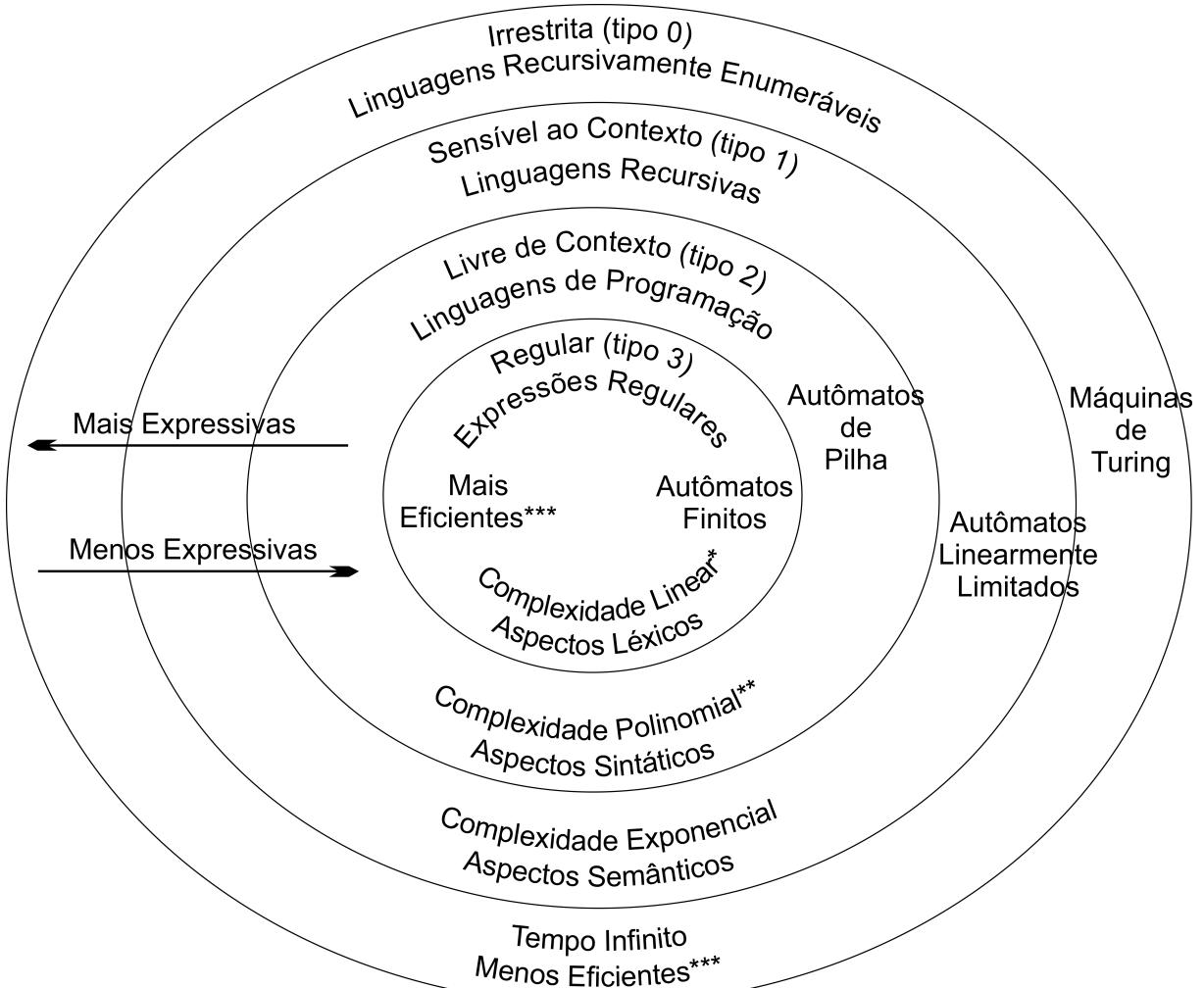
Gramáticas Regulares (também conhecidas como Tipo 3) são todas aquelas reconhecidas por Autômatos Finitos Determinísticos e/ou Não-Determinísticos. Autômatos reconhecem linguagens e gramáticas geram palavras de sua linguagem (Seção 2.3.2: Reduções e Derivações). Para toda linguagem reconhecida por Autômato Finito, existe uma Gramática Regular que gera palavras dessa linguagem. Gramáticas de Lingua-

⁸ Veja a Figura 9 e perceba que Gramáticas Irrestritas são equivalentes a Máquinas de Turing, portanto podem em pior caso levar tempo infinito para responder. Aprenda mais a respeito procurando sobre o problema da parada (ROYER, 2015; SIPSER, 2012) ou responda a pergunta: Quanto tempo levaria para a seguinte instrução de um programa na Linguagem C terminar de executar: “for(;;);”?

⁹ Tempo computational refere-se ao número de passos ou instruções necessários para executar determinado procedimento (Seção 2.4: Compiladores e Classes de Complexidade).

¹⁰ Do inglês, *Pumping Lemma* (HOPCROFT; MOTWANI; ULLMAN, 2006; SIPSER, 2012).

Figura 9 – Hierarquia de Chomsky



*Para Gramáticas Regulares Determinísticas, complexidade linear ao tamanho da palavra de entrada para determinar se uma dada palavra pertence ou não à linguagem. Para Gramáticas Regulares Não-Determinísticas, complexidade polinomial para construir algumas Árvores de Derivações e determinar se dada palavra pertence ou não a linguagem, com algoritmos como CYK ou Earley (Seção 2.4: Compiladores e Classes de Complexidade). Por fim, para Autômatos Finitos Não-Determinísticos ou Analisadores com Backtracking, tempo exponencial.

**Para Gramáticas Livres de Contexto Determinísticas, também conhecidas como LR(K), complexidade linear ao tamanho da palavra de entrada (Seção 2.3.1: Gramáticas *versus* Linguagens). Para Gramáticas Livre de Contexto Não-Determinísticas, vale o mesmo que para Linguagens Regulares Não-Determinísticas logo acima, mas no lugar de Autômatos Finitos Não-Determinísticos, utilizam-se Máquinas de Pilha Não-Determinísticas.

***Para verificar se uma dada sentença pertence ou não a linguagem.

Fonte: Própria, baseado em Sipser (2012), Aho e Ullman (1972), Lang (1974) e Cocke (1969)

Nota: As complexidades Linear, Polinomial, Exponential e Infinita são explicados na Seção 2.4: Compiladores e Classes de Complexidade.

Nota: Aprenda o que são Gramáticas Determinísticas ou Não-Determinísticas lendo a Seção 2.3.1: Gramáticas *versus* Linguagens.

gens Regulares pela definição formal, são todas aquelas nas quais todas as Produções P da gramática possuem a seguinte forma:

$$P = \{ \alpha ::= a\beta \mid \alpha \in V_n \wedge a \in V_t \wedge \beta \in \{ V_n \cup \varepsilon \} \}$$

2.2.3 Gramáticas Livres de Contexto

Gramáticas Livres de Contexto (também conhecidas como Tipo 2) (HOPCROFT; MOTWANI; ULLMAN, 2006) são todas aquelas reconhecidas por Autômatos de Pilha Não-Determinísticos. Gramáticas de Linguagens Livre de Contexto pela definição formal, são todas aquelas nas quais todas as Produções P da gramática possuem a seguinte forma:

$$P = \{ \alpha ::= \beta \mid \alpha \in V_n \wedge \beta \in V^* \}$$

2.2.4 Gramáticas Sensíveis ao Contexto

Gramáticas Sensíveis ao Contexto (também conhecidas como Tipo 1) são todas aquelas reconhecidas por Autômatos Linearmente Limitados¹¹, que tratam-se somente de Máquinas de Turing (SIPSER, 2012) com Fita (ou memória) Finita. Gramáticas de Linguagens Sensíveis ao Contexto pela definição formal, são todas aquelas nas quais todas as Produções P da gramática possuem a seguinte forma. Onde os símbolos $|\alpha|$ e $|\beta|$ significam a quantidade (contável) de elementos dentro de α e β , respectivamente:

$$P = \{ \alpha ::= \beta \mid \alpha \in V^*V_nV^* \wedge \beta \in V^* \wedge |\alpha| \leq |\beta| \}$$

2.2.5 Gramáticas Irrestritas

Por fim, as Gramáticas Irrestritas (também conhecidas como Tipo 0), possuem a mesma definição do que a definição formal de gramática (apresentado anteriormente no Item 3: Gramáticas). Gramáticas Irrestritas são reconhecidas somente por Máquinas de Turing¹², e diferente das Gramáticas Sensíveis ao Contexto, a Máquina de Turing não possui parada garantida.

Linguagens do Tipo 0 (ou Irrestritas) representam problemas indecidíveis e que podem ser representados por procedimentos (SIPSER, 2012). Já Linguagens do Tipo 1 (ou Sensíveis ao Contexto), representam todos os problemas decidíveis e sua implementação pode ser representada por algoritmos, pois possuem parada garantida, apesar de terem em pior caso, tempo exponencial ao contrário de tempo infinito¹³, como nas Linguagens Irrestritas.

¹¹ Do inglês, Linear Bounded Automata (DAVIS; SIGAL; WEYUKER, 1994).

¹² Máquinas de Turing possuem por definição fita (ou memória) ilimitada, mas não infinita, pois em um dado momento, somente uma quantidade finita de símbolos podem estar na fita, que continuamente pode crescer ilimitadamente.

¹³ Porque podem em pior caso levar tempo infinito para responder, imagine o seguinte trecho de código-fonte na linguagem C “for(; ;);”.

2.3 ANALISADORES SINTÁTICOS

Analisadores são equivalentes à Mecanismos Reconhecedores como Autômatos Finitos, Autômatos de Pilha ou Máquinas de Turing. No caso de outros mecanismos como Autômatos Finitos, o reconhecimento é feito a partir da especificação ou construção do autômato que reconhece palavras de dada linguagem. Por exemplo, ambos Gramáticas Regulares e Autômatos Finitos são equivalentes e existem algoritmos de conversão entre um e outro (HOPCROFT; MOTWANI; ULLMAN, 2006).

Analisador Sintático é um nome dado para analisadores que recebem como entrada uma Gramática Livre de Contexto que representa os aspectos estruturais de uma linguagem, i.e., sua sintaxe (AHO; LAM *et al.*, 2006). Analisadores Sintáticos possuem muito mais utilidade do que somente checar se a sintaxe do programa de entrada está correta, uma vez que eles também podem gerar a Árvore Sintática do programa¹⁴ que é utilizada para realizar a Análise Semântica e geração de código.

2.3.1 Gramáticas *versus* Linguagens

É importante fazer a distinção entre Gramáticas Livre de Contexto e as Linguagens Livre de Contexto. Parikh (1966), provou que existem linguagens para as quais não existe Gramática Não-Ambígua que as representem. Tais linguagens são conhecidas como Linguagens Inerentemente Ambíguas¹⁵ onde não existe Gramática Livre de Contexto Determinística capaz de representá-las e tais Linguagens somente podem ser reconhecidas por Analisadores com Backtracking (AHO; LAM *et al.*, 2006), Autômatos de Pilha Não-Determinísticos ou algoritmos de análise como Earley e CYK (Seção 2.4: Compiladores e Classes de Complexidade).

A maior classe de Gramáticas Determinísticas suportadas por Analisadores Sintáticos são as Gramáticas LR(K)¹⁶. Analisadores LR(K) (AHO; LAM *et al.*, 2006) são Ascendentes e reconhecem um subconjunto das Linguagens Livre de Contexto (Figura 10). Já os Analisadores LL(K)¹⁷ são Descendentes (PARR, 2013; PARR; FISHER, 2011; PARR; HARWELL; FISHER, 2014) e reconhecem somente um subconjunto das Linguagens LR(K)¹⁸.

Na Figura 10, encontra-se um Diagrama de Venn (KESTLER *et al.*, 2004) com

¹⁴ Como visto no começo desde capítulo na Seção 2.1: Compiladores e Tradutores.

¹⁵ Do inglês, *Inherently Ambiguous Languages*. Veja o que significa ambiguidade na Seção 2.3.4: Análise Semântica e Quadro 1.

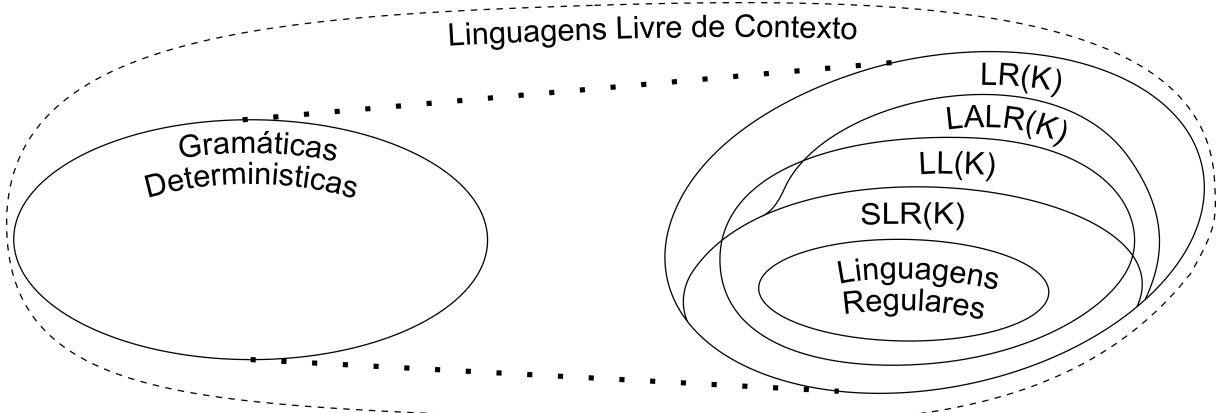
¹⁶ Do inglês, *Left-to-right, Rightmost derivation*, em reverso com K símbolos de lookahead. *Rightmost* significa que ao realizar as derivações, escolhe-se sempre o não-terminal mais a direita.

¹⁷ Do inglês, *Left-to-right, Leftmost derivation*, com K símbolos de lookahead. *Leftmost* significa que ao realizar as derivações, escolhe-se sempre o não-terminal mais a esquerda.

¹⁸ Diz-se que uma linguagem é LR(K) ou LL(K) quando ela é reconhecida por este analisador.

a relação entre as classes de linguagens de acordo com os analisadores que as reconhecem. Nela encontra-se as gramáticas que são mais importantes, as Gramáticas Determinísticas¹⁹, que podem ser classificadas como $LR(K)$, $LL(K)$, etc, i.e., de acordo com o tipo de analisador que pode ser construído sem conflitos em sua Tabela de Análise²⁰.

Figura 10 – Gramáticas Determinísticas *versus* suas Linguagens



Fonte: Própria, baseado em Brink (2013), Rici (2019), Beatty (1982), Aho, Lam *et al.* (2006) e Johnson e Zelenski (2009)

Nota: Apesar da fonte da figura ser o próprio autor deste texto, o conhecimento requerido para construir tão complexa figura não, requerendo pesquisa em várias fontes.

É importante notar que usualmente o processo de análise por um analisador, seja ele $LR(K)$ ou $LL(K)$, acontece em duas etapas. Com a exceção dos Analisadores $LL(K)$ que também podem ser facilmente construídos programaticamente²¹, i.e., com o programador construindo manualmente como deve acontecer cada transição de estado do analisador (AHO; LAM *et al.*, 2006).

Na primeira etapa de um analisador como $LL(K)$ utilizam-se algoritmos de construção da Tabela de Análise. Quando a tabela está construída sem conflitos (este analisador portanto, é Determinístico), entra em cena o algoritmo de análise na segunda etapa, que utilizando a Tabela de Análise, realiza o reconhecimento do programa de entrada. A única diferença entre os Analisadores $LR(K)$, $LALR(K)$ e $SLR(K)$ é a construção da Tabela de Análise. Ambos utilizam o mesmo algoritmo de análise, e somente trocando o algoritmo de geração Tabela de Análise. Assim, ambos os analisadores possuem a mesma complexidade de tempo computational para determinar se

¹⁹ As Gramáticas Determinísticas representam o conjunto de Linguagens que podem ser Analisadas Deterministicamente e tais Linguagens também podem ser conhecidas como $LR(K)$. Reveja os parágrafos após a Figura 9.

²⁰ Do inglês, *Parsing Table*, (AHO; LAM *et al.*, 2006).

²¹ Neste caso, o analisador pode ser conhecido como Descendente Recursivo, (do inglês, Recursive Descent).

um programa pertence ou não a linguagem (análise linear ao tamanho do programa²² de entrada que será analisado (KNUTH, 1965; HOLZER; LANGE, 1993; LEO, 1991)).

No caso de conflitos na Tabela de Análise, a gramática não pode ser analisada deterministicamente e algoritmos de análise com backtracking ou algoritmos de tempo polinomial como Earley e CYK (Seção 2.4: Compiladores e Classes de Complexidade), precisam ser utilizados para construção da Árvore Sintática. Analisadores com Backtracking também funcionam em pior caso, com tempo exponencial e podem escolher uma estratégia como Busca em Profundidade (CORMEN *et al.*, 2009) para executar os Ramos de Computação Não-Determinística.

2.3.2 Reduções e Derivações

Diferente de máquinas específicas como Autômatos Finitos, analisadores recebem diretamente como entrada uma gramática de uma dada linguagem. Mas diferente de Gramáticas e Analisadores LL(K), Analisadores LR(K) especificamente funcionam de modo contrário. Analisadores LR(K) operam por meio de Reduções ao contrário de Derivações como no caso das Gramáticas e Analisadores LL(K) (AHO; LAM *et al.*, 2006).

Uma Derivação acontece quando uma regra de produção como “ $S \Rightarrow aa$ ” de uma gramática expande e tem-se como resultado “ aa ” a partir do símbolo de origem “ S ”. Já uma Redução acontece quanto a dada regra de produção como “ $S \Rightarrow aa$ ” de uma gramática reduz e tem-se como resultado “ S ” a partir do símbolo de origem “ aa ”. Tanto Derivações quanto Reduções podem ser descritas em termos de quantos passos são necessários para que se possa sair de um ponto até outro:

1. Quando uma derivação é denotada como “ $S \Rightarrow aa$ ”, isso significa que somente um passo é necessário para sair do símbolo inicial “ S ” e chegar no símbolo final “ aa ”;
2. Quando uma derivação é denotada como “ $S \xrightarrow{*} aa$ ”, isso significa que são necessários, desde zero (nenhum) até infinitos passos para sair do símbolo inicial “ S ” e chegar no símbolo final “ aa ”;
3. Quando uma derivação é denotada como “ $S \xrightarrow{+} aa$ ”, isso significa que são necessários, desde um passo até infinitos passos para sair do símbolo inicial “ S ” e chegar no símbolo final “ aa ”.

²² Quando refere-se a programa, fala-se da *string* ou texto que será analisado e decidir se tal programa é um programa da linguagem que se está analisando. Um ponto curioso, caso o programa não seja aceito pelo analisador, ele não é um programa com erros, mas um programa inválido, i.e., de uma outra linguagem, que não é a linguagem que está sendo analisada. Comumente ou informalmente, chamamos estes programas como programas com erros (de sintaxe).

Para reduções, estas mesmas condições se aplicam, mas em ordem reversa, i.e., \Leftarrow , \Leftarrow^* e \Leftarrow^\dagger , ao invés de \Rightarrow , \Rightarrow^* e \Rightarrow^\dagger (AHO; LAM *et al.*, 2006). Enquanto gramáticas são geradores de palavras que partem do símbolo inicial da gramática até gerarem uma palavra da linguagem, Analisadores Ascendentes são reconhecedores de palavras, para uma dada gramática de entrada.

Diferente de gramáticas, Analisadores Ascendentes como LR(K) partem de uma palavra da linguagem até chegarem no símbolo inicial da gramática, consumindo toda a palavra de entrada e chegando em um Estado de Aceitação. Já Analisadores Descendentes como LL(K), partem do símbolo inicial da gramática até consumirem toda palavra de entrada, também chegando um em Estado de Aceitação.

Ambos os Analisadores Ascendentes ou Descendentes, terminam no final do processo, gerando toda a Árvore de Derivação (Árvore Sintática). Entretanto, caso no final do processo reconhecimento de um Analisador Descendente, não se chegue em um Estado de Aceitação (AHO; LAM *et al.*, 2006), tem-se somente a construção de uma Árvore de Derivação parcial. Já no caso dos Analisadores Ascendentes, será uma floresta (de árvores), porque somente no final da análise, com a chegada ao símbolo inicial da gramática, o Analisador Ascendente completa a “costura” de todas a árvores que foram parcialmente construídas durante o processo de análise (*Bottom-Up*).

No caso dos Analisadores Descendentes, não existe uma floresta de árvores. Como parte-se diretamente do símbolo inicial da gramática, a Árvore de Derivação desde o começo é construída como sendo uma única árvore (*Top-Down*). Em caso de erros na construção da Árvore Sintática, ela terminará somente com alguns nós-folhas faltando.

2.3.3 Analisadores LR(K)

Como pode ser observado na Figura 10, existem Gramáticas SLR(K) que não são Gramáticas LL(K) porque para uma gramática ser LL(K), ela precisa respeitar 3 propriedades: 1) Não possuir Recursão à Esquerda; 2) Estar fatorada e; 3) $\forall A \in V_n \mid A \xrightarrow{*} \varepsilon \wedge First(A) \cap Follow(A) = \emptyset$ (AHO; LAM *et al.*, 2006).

Entretanto, Gramáticas LR(K), LALR(K) e SLR(K) não precisam de nenhuma dessas restrições. No caso da Recursão à Esquerda, o algoritmo de criação da Tabela de Análise Sintática da Gramática LR(K), LALR(K) ou SLR(K), não possui o problema de entrar em um loop infinito assim como acontecem com as Gramáticas LL(K), portanto aceitando-se Gramáticas com Recursão à Esquerda (AHO; LAM *et al.*, 2006).

Analisadores LR(K) requerem grandes quantidades de memória, proporcional ao tamanho da gramática de entrada (HUNT; SZYMANSKI; ULLMAN, 1975). Por isso, DeRemer e Pennello (1982), criaram os Analisadores LALR(K)²³ e SLR(K)²⁴ com o

²³ Do inglês, *Look-Ahead LA(K) LR(0)*, onde LR(0) é um Analisador LR(K) com $K = 0$.

²⁴ Do inglês, *Simple LR(K) parser*.

objeto de viabilizar a implementação de Analisadores Ascendentes Determinísticos.

Gramáticas de Linguagens Determinísticas são chamadas de LR, porque todas as Linguagens Determinísticas são reconhecidas por Analisadores LR(K), uma vez que Knuth (1965), provou que todas as Gramáticas Determinísticas são aceitas por um Analisador LR(K). Assim, além da Hierarquia de Chomsky (Figura 9), também classifica-se as gramáticas de acordo com o tipo de analisador que reconhece as linguagens representadas por elas. Como mostrado na Figura 10, nem todas as Gramáticas Livre de Contexto são Gramáticas Determinísticas e uma gramática é Determinística somente se ela pode ser reconhecida por um Analisador LR(K).

Portanto, uma maneira fácil de decidir se uma dada gramática é Determinística ou não, é tentar construir a sua Tabela de Análise para um Analisador LR(K). Caso se consiga construir com sucesso (sem conflitos) a Tabela de Análise Sintática (AHO; LAM *et al.*, 2006), a gramática é LR(K) e Determinística, caso contrário a gramática não é Determinística. A mesma técnica pode ser aplicada no caso de analisadores menos poderosos como LALR(K), entretanto, uma vez que não se consiga construir a Tabela de Análise Sintática, não se pode ter certeza se dada gramática é ou não Determinística.

2.3.4 Análise Semântica

Usualmente²⁵, somente depois que a Árvore Sintática é construída, realiza-se o processo de Análise Semântica (AHO; LAM *et al.*, 2006), i.e., a verificação da corretude do programa escrito em relação os aspectos não-estruturais (Código 2). Por exemplo, é sintaticamente correto escrever a declaração de uma mesma variável duas vezes ou mais em um mesmo escopo. Entretanto, para algumas linguagens é semanticamente errado redeclarar uma variável duas vezes ou mais em um mesmo escopo.

O Analisador Sintático representado por uma Gramática Livre de Contexto não tem poder suficiente para realizar verificações de significado, devido as limitações desse tipo de gramática, que se restringem a estrutura do programa e não ao seu significado (semântica).

Nem todas as linguagens são analisadas completamente em diferentes etapas, como Análise Léxica, Sintática e Semântica. Muitas vezes, estas três etapas acontecem em paralelo como realizado na implementação do compilador da Linguagem C (JOURDAN; POTTIER, 2017; BAXTER, 2009). A Gramática da Linguagem C não é implementada utilizando um Analisador geral como LR(K) para simplificar a construção de seu Analisador Semântico. Na estrutura ou sintaxe de um programa C existem

²⁵ Como será apresentado mais a frente nesta seção: 1) o processo de Análise Semântica pode acontecer ao mesmo tempo que a Análise Sintática; 2) por fim, a Árvore Sintática também pode não ser gerada, ocorrendo diretamente Análise Semântica seguido da Geração de Código.

ambiguidades²⁶ (Quadro 1) semânticas como a expressão “ $x * y;$ ”. Tal sentença pode ser ou a declaração de um ponteiro chamado “ y ” do tipo “ x ”, ou a multiplicação de dois números armazenados nas variáveis “ x ” e “ y ”.

Quadro 1 – Exemplo de Ambiguidade Linguística

SOCORRO!
UM TIGRE DE BENGALA ESTÁ ME ATACANDO!

1. Um tigre que está utilizando uma bengala para se locomover está atrás de você;
2. Um tigre que veio da grande área metropolitana de Bengala na Ásia está atrás de você;
3. Um torcedor do Criciúma (tigre) que usa sua bengala, depois que seu time perdeu de 7 à 1 está atrás de você.

Como a ambiguidade de “ $x * y;$ ” não pode ser resolvida pela sintaxe ou estrutura de um programa (sua Gramática Livre de Contexto), será necessário validar se “ $x * y;$ ” trata-se da declaração de um ponteiro ou a multiplicação de duas variáveis na etapa de Análise Semântica (verificação de significado, equivalente uma Gramática Sensível ao Contexto), pois essa característica da linguagem C é um aspecto Sensível ao Contexto. Portanto, para simplificar a implementação do Analisador Semântico, escolheu-se programaticamente²⁷ construir o analisador da linguagem C.

Assim, o compilador da linguagem C consegue fazer a “Análise Determinística” da linguagem verificando o significado de uso símbolo asterisco ou estrela “*” como operador de multiplicação ou declaração de variável do tipo ponteiro com a realização “simultânea” da Análise Léxica, Sintática e Semântica. Uma vez que um novo *token* é reconhecido, ele é enviado ao Analisador Sintático que atualiza a Tabela de Símbolos (AHO; LAM *et al.*, 2006). Durante o processo de “Análise Sintática” então se utiliza da Tabela de Símbolos para eliminar a ambiguidade “ $x * y;$ ”.

Requer-se cuidado sobre como as alterações do Analisador são feitas, pois pode-se pensar que todas as gramáticas de todas as linguagens de programação são “Livres de Contexto” e Determinísticas, o que não é verdade (veja a próxima Seção 2.3.5: Alterações nos Analisadores Sintáticos). Uma vez que a gramática não é mais Livre de Contexto ou Determinística, pode-se mover Aspectos Sensíveis ao Contexto para o Analisador Semântico, assim, deixando a gramática somente com aspectos Determinísticos.

Como também poderia ter sido feito no caso da implementação da linguagem C,

²⁶ Conhecido também como Não-Determinismo.

²⁷ Como o Compilador da Linguagem C é escrito programaticamente, ele é equivalente a uma Máquina de Turing e portanto possui em pior caso, tempo infinito de execução, reveja a Figura 9: Hierarquia de Chomsky.

ao contrário de criar programaticamente um analisador que verifica aspectos semânticos durante a Análise Sintática, poderia-se resolver os aspectos Sensíveis ao Contexto de “ $x * y;$ ” para a etapa de Análise Semântica, e então utilizar-se um Analisador como LR(K). Com o ônus de ter que se criar algoritmos de Análise Semântica, que trabalham sobre a Árvore Sintática gerada pelo Analisador (LR(K)) para verificar os aspectos semânticos (de significado da estrutura).

2.3.5 Alterações nos Analisadores Sintáticos

Dependendo de como o Analisador Sintático de Gramáticas Livre de Contexto é alterado, o conjunto de gramáticas aceitas por tal analisador pode deixar de ser Livre de Contexto. As gramáticas somente continuarão Livre de Contexto caso estas alterações sejam somente mover checagens da etapa de Análise Sintática para a etapa de Análise Semântica sem realizar alterações no Analisador Sintático.

Quando se adiciona suporte a Aspectos Sensíveis ao Contexto (WOODS, 1970) à Gramáticas Livre de Contexto por meio de alterações do Analisador Sintático, como feito no Analisador da Linguagem C, o analisador da gramática deixa de ser Livre de Contexto, suportando assim, algumas Gramáticas Sensíveis ao Contexto e/ou também algumas Gramáticas Não-Determinísticas.

Note que, apesar disso não impede-se que a gramática da Linguagem C, como mostrado na seção anterior (Seção 2.3.4: Análise Semântica), seja analisada com eficiência. Mas isso deixa brechas para que ela possa não ser analisada com eficiência. A diferença para um analisador onde a gramática é inteiramente Livre de Contexto Determinística, é que ela tem desempenho *garantido* pela sua Classe de Complexidade (Seção 2.4: Compiladores e Classes de Complexidade).

Sintaxe e Semântica de Linguagens são completamente ortogonais. Gramáticas de Linguagens Irrestritas²⁸ podem ser Turing Completas²⁹ devido a sua equivalência com Máquinas de Turing e são capazes de realizar qualquer operação computacional. Mas, isso não pode ser confundido com as *Strings* ou Programas gerados por essas gramáticas (MICHAELSON, 2016; ZERVOUDAKIS *et al.*, 2013). Tais programas podem ou não ser Turing Completos. Do lado oposto, até Linguagens Regulares podem gerar programas que são Turing Completos, mesmo que seu dispositivo reconhecedor equivalente, os Autômatos Finitos, não tenham Turing Completude³⁰ (KIRPICHOV, 2014; FENNER, 2019).

²⁸ Não a linguagem que elas representam, mas a própria gramática em si (FENNER, 2019).

²⁹ A Turing Completude acontece quando uma dada linguagem pode simular o funcionamento completo de uma Máquina de Turing (MICHAELSON, 2016).

³⁰ Caso isso esteja confuso, reveja a Figura 9 e note que de todas as Linguagens, quem tem Turing Completude são as Linguagens Irrestritas, enquanto Autômatos Finitos são um subconjunto das Máquinas de Turing (FENNER, 2019).

2.4 COMPILADORES E CLASSES DE COMPLEXIDADE

Como um todo, o conjunto de Linguagens Regulares pode ser considerado com complexidade linear³¹ em tempo computacional para determinar de dada palavra pertence ou não a linguagem, porque toda Gramática Regular Não-Determinística pode ser convertida em uma Gramática Regular Determinística (SIPSER, 2012).

Infelizmente isso não é verdade para Gramáticas Livres de Contexto, porque Gramáticas Livre de Contexto Determinísticas e Não-Determinísticas não são equivalentes e uma não pode ser convertida em outra. Gramáticas Não-Determinísticas possuem complexidade exponencial, quando analisadas por um Analisador com Backtracking. Em contra-partida, qualquer Gramática Livre de Contexto também pode ser analisada em tempo polinomial com algoritmos de análise como CYK ou Earley (SHINAN, 2014).

O algoritmo de análise CYK (HOPCROFT; MOTWANI; ULLMAN, 2006; SHINAN, 2014), possui complexidade de tempo $\Theta(n^3 \cdot |G|)$, onde n é o tamanho do programa de entrada e G é o tamanho da gramática de entrada na Forma Normal de Chomsky (HOPCROFT; MOTWANI; ULLMAN, 2006). Em tempo $\Theta(n^3)$ o algoritmo CYK utilizando a gramática e o programa de entrada, constrói a Tabela de Análise que diz se dada palavra pertence ou não a linguagem.

Earley (1970), publicou um artigo com a especificação do algoritmo de análise Earley: 1) capaz de analisar qualquer Gramática Livre de Contexto ambígua em tempo $\Theta(n^3)$; 2) capaz de analisar qualquer Gramática Livre de Contexto não-ambígua em tempo $\Theta(n^2)$ e; 3) capaz de analisar qualquer Gramática Livre de Contexto LR(K) em tempo $\Theta(n^1)$. Quando faz-se a Análise de uma Gramática Livre de Contexto Não-Determinística, tem-se como resultado várias possíveis Árvores de Derivação³².

Neste capítulo, foi apresentado um pouco sobre a Teoria da Computação e Linguagens Formais. Tal conteúdo, é ligado fortemente com matemática e provas formais. Diferente do conteúdo do próximo capítulo, que está mais ligado com às últimas mudanças do estado da arte para editores de texto e ferramentas de auxílio ao desenvolvimento de software.

³¹ Complexidade linear é um caso particular de complexidade polinomial onde o grau do Polinômio é 1, i.e., $\Theta(n)$.

³² Como a gramática é Não-Determinística, existem muitas possíveis Árvores de Derivação, (devido à ambiguidade da gramática, Quadro 1).

3 ESTADO DA ARTE

Programar pode ser considerado pela maioria das pessoas como escrever um livro, e para tal, a facilidade de leitura é desejável. Existem livros difíceis de ler: fontes inadequadas ou pequenas, com folhas que brilham contra a luz. No mínimo espera-se algumas características chave ao comprar-se um livro (VELAZQUEZ-ITURBIDE; PRESA-VAZQUEZ, 1999). Por exemplo, quando compra-se um livro, espera-se que (YENIGALLA *et al.*, 2016):

1. Seu conteúdo esteja bem organizado, para que o leitor não se perca durante a leitura;
2. Que suas cores sejam propriamente escolhidas e utilizadas, para que elas não distraiam o leitor ou tirem o foco do principal, o conteúdo do livro;
3. Que o espaçamento entre os parágrafos, palavras, capítulos, seções, subseções, etc, estejam propriamente ajustados, e não aglomerado ou desordenado em um único parágrafo, frase, capítulo.

3.1 FORMATADORES DE CÓDIGO

Conhecido como “*pretty-printing*” ou embelezadores¹ (DE JONGE, 2002), uma ferramenta de formatação pode ser complicada de se utilizar, somente com um conjunto básico de definições. Por exemplo, para permitir um melhor controle do usuário, a ferramenta de formatação pode permitir que exista uma configuração específica para cada aspecto da linguagem.

Uma possível implementação para tal ajuste fino, pode ser uma configuração específica como por exemplo a entrada booleana “use_spaces_after_if”, em um arquivo de configuração para definir caso deva-se ou não adicionar espaços após cada “if” ao fazer a formatação da linguagem, i.e., “if (var)” ao contrário de “if(var)”.

Como pode ser percebido, uma lista contendo todas as configurações de formatação para cada aspecto da linguagem, ficará muito grande quando todos os aspectos das linguagens mais complexas como C ou C++ forem implementados. Em softwares como Uncrustify (BERG; GARDNER; MAUREL, 2005b), encontram-se mais de 500 configurações² tais como o Código 3. No Código 3, deve ser observado uma amostra das mais de 500 opções que um formatador de código-fonte configurável por arquivos de configuração pode ter. Também o quão difícil pode ser escrever e entender o que cada uma dessas incontáveis opções significam.

¹ Do inglês, *Source Code Beautifiers*.

² Veja mais sobre configurações na Seção 3.3: Trabalhos Relacionados.

Código 3 – Trecho do Arquivo de Configuração de Uncrustify

```
1 ...
2
3 # Add or remove space before ','
4 sp_before_comma = remove # ignore/add/remove/force
5
6 # Add or remove space between an open paren
7 # and comma: '(', ' vs '(', ,
8 sp_paren_comma = add # ignore/add/remove/force
9
10 # Add or remove space after class ':'
11 sp_after_class_colon = ignore # ignore/add/remove/force
12
13 # Add or remove space before class ':'
14 sp_before_class_colon = ignore # ignore/add/remove/force
15
16 ...
17 # Whether the 'extern "C"' body is indented
18 indent_extern = false      # false/true
19
20 # Whether the 'class' body is indented
21 indent_class = true       # false/true
22
23 # Whether to indent the stuff after a
24 # leading base class colon
25 indent_class_colon = false # false/true
26
27 # Whether to indent the stuff after a
28 # leading class initializer colon
29 indent_constr_colon = false # false/true
30
31 ...
```

Mesmo que se passe por todas as configurações do formador de código-fonte (amostrado no Código 3), realizando os ajustes de preferência do usuário, isso levará um bom tempo. Infelizmente com todo esse processo, somente será realizado a configuração de uma e talvez algumas linguagens fortemente relacionadas, dependendo do suporte que a ferramenta de formatação de código-fonte oferece.

No caso da ferramenta Uncrustify, serão configuradas linguagens como “C”, “C++”, “C” e “ObjectiveC” (veja mais linguagens na Seção 3.3: Trabalhos Relacionados). Para todas as outras linguagens, ainda será preciso encontrar outra ferramenta de formatação, que será certamente mais limitada no que ela pode customizar (STEPHEN, 2017;

MAXDAX, 2019), uma vez que Uncrustify já é uma das mais completas.

Ao realiza-se a migração de uma ferramenta de formatação como Uncrustify para outra, precisa-se configurar novamente todas as novas opções já definidas para Uncrustify. Outra forte desvantagem do arquivo de configurações do Uncrustify segue na dificuldade de entender e visualizar o que está sendo configurado para que se possa realizar a migração.

3.2 QUAL A UTILIDADE DE FORMATADORES?

Pode-se compreender uma frase sem pontuação, mas é mais fácil fazer essa operação utilizando a pontuação. Por exemplo:

umaboapontuacaocomcertezatornaascoisasmaisfaceisdeseler

Um leitor hábil pode compreender facilmente a oração acima, mas a pontuação tornaria a tarefa mais fácil (WICKHAM, 2017).

Não pôde-se encontrar nenhuma forte evidência³ ou relação sobre a compreensão de códigos-fonte e formatadores de código. Alguns estudos como Scalabrino *et al.* (2016), implementam modelos de Inteligência Artificial utilizados para classificar códigos-fonte como “bem legíveis” ou “mal legíveis”. Inicialmente estes estudos começam coletando dados de pessoas classificando códigos-fonte como legíveis ou não, para então treinar a Inteligência Artificial para classificar códigos-fonte.

Além desse estudo científico para classificar códigos-fonte, encontram-se em outros lugares como Rossum, Warsaw e Coghlan (2001), dicas ou notas explicando quais características de formatação de código-fonte ajudam ou prejudicam a legibilidade de código.

Um dos melhores métodos de destruição de equipes de desenvolvedores de software, é engajar-se em guerras de formatação passiva-agressiva. Elas desestimulam os relacionamentos entre colegas, e dependendo do tipo de formatação feita, também podem prejudicar a capacidade de comparar efetivamente as revisões no sistema de controle de versão, o que é realmente assustador. Não pode-se nem imaginar o quanto ruim seria caso a liderança das equipes fosse a culpada por esse comportamento. Isso que é liderar por exemplo. Por mau exemplo. (ATWOOD, 2007, tradução nossa⁴).

Em Miara *et al.* (1983), foi analisado o nível de compreensão do programa que pode ser obtido pela indentação e foi constatado que os níveis de indentação de 2 e 4 espaços provaram ter os melhores níveis de compreensão do que outros níveis.

³ Com forte evidência refere-se a afirmações absolutas como: “–Ao utilizar formatadores de código, é garantido que seu projeto de desenvolvimento de código-fonte irá sofrer ganhos em tempo de desenvolvimento e qualidade de software”.

⁴ One of absolute worst, worst methods of teamicide for software developers is to engage in these kinds of passive-aggressive formatting wars. I know because I've been there. They destroy peer relationships, and depending on the type of formatting, can also damage your ability to effectively compare revisions in source control, which is really scary. I can't even imagine how bad it would get if the lead was guilty of this behavior. That's leading by example, all right. Bad example.

Por mais absurdo que possa parecer, lutar sobre espaços em branco e outras questões aparentemente triviais de layout de código-fonte é realmente justificado. Desde que feito com moderação, abertamente, de uma forma justa e com construção de consenso, sem esfaquear companheiros de equipe ao longo do caminho. (ATWOOD, 2007, tradução nossa⁵).

No fim, como não se pode dizer certamente que ao utilizar ferramentas de formatação de código-fonte se irá ter um melhor desempenho, cabe a cada desenvolvedor ou time de desenvolvimento decidir por sua experiência, se existe a necessidade de utilizar uma ferramenta de formatação de código-fonte.

3.3 TRABALHOS RELACIONADOS

Nesta seção, são relatados os trabalhos relacionados obtidos através de pesquisa na literatura utilizando a metodologia especificada no Capítulo 1: Introdução. Segundo Berg, Gardner e Maurel (2005b), Uncrustify é capaz de formatar as linguagens: 1) C; 2) C++; 3) C#; 4) ObjectiveC; 5) D; 6) Java; 7) Pawn e; 8) VALA. De acordo com o site (*Uncrustify*), a versão “0.69.0” possui 671 configurações para personalizar a formatação do código-fonte como mostrado no Código 3.

Existem muitas ferramentas e/ou editores de texto, e cada uma dessas ferramentas possui suas vantagens e desvantagens (DINESH; USKUDARH, 1997; CAMERON, 1988; KIM *et al.*, 2016). Por isso, alguns desenvolvedores de software podem utilizar simultaneamente duas ou mais ferramentas dependendo de suas especialidades e da tarefa que o desenvolvedor está realizando no momento.

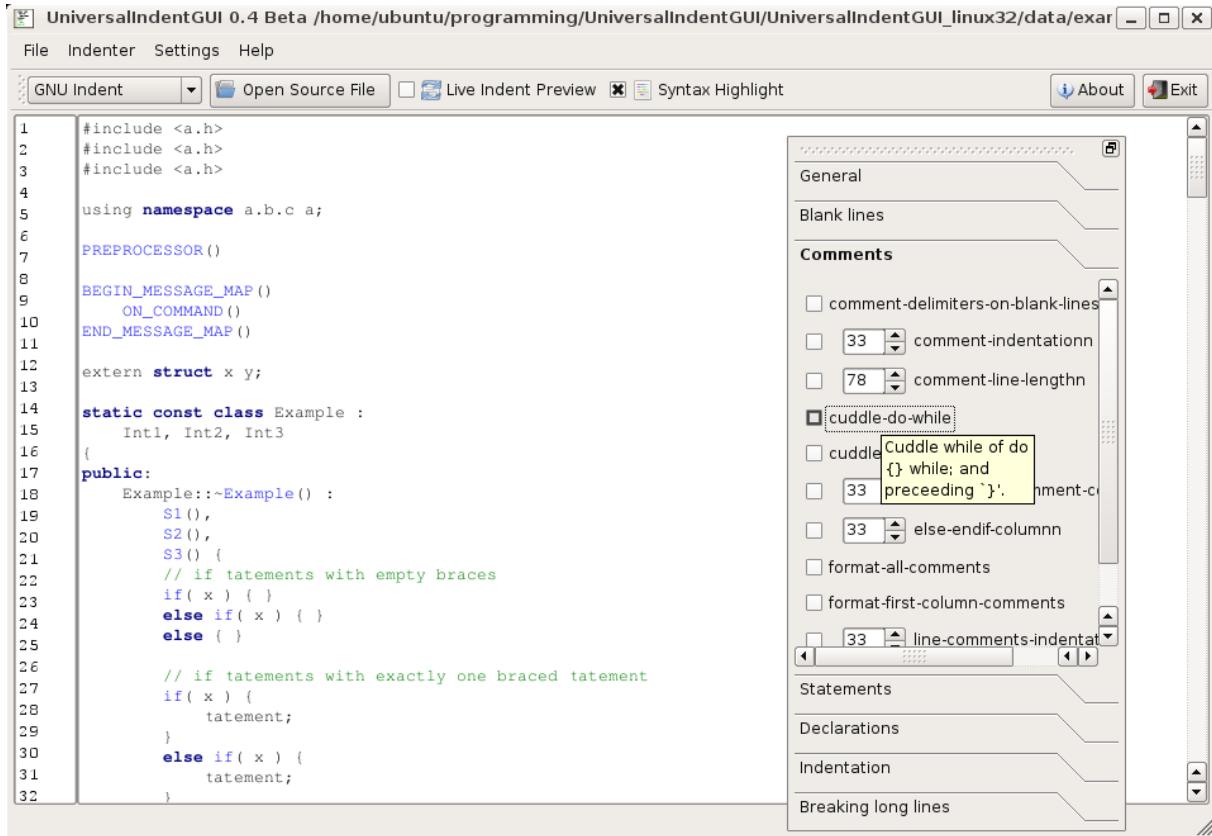
Trabalhos como Hunner e Xu (2012), não são diretamente um formatador de código-fonte. Mas sua relação com este trabalho é seu propósito de servir como uma ponte de compartilhamento de configurações entre todos os editores de texto. Com isso, facilitando a troca de um editor pelo outro durante o desenvolvimento.

Schweitzer (2006), é outra ferramenta que não é um formatador de código-fonte, mas sim uma interface comum às várias e diferentes ferramentas de formatação de código-fonte. Na Figura 11, pode-se ver como é a interface com o usuário que esta ferramenta proporciona.

A interface da Figura 11, permite que se escolha qual será o formatador de código que será utilizado, e permite que se carregue um arquivo de código-fonte para ser formatado, e na medida em que se seleciona as opções pela interface gráfica, pode-se visualizar as mudanças no código-fonte exemplo carregado. Entretanto, a ferramenta proposta por Schweitzer (2006) enfrenta problemas como as limitações de sua interface gráfica. Como já mencionado na Seção 3.3: Trabalhos Relacionados, Uncrustify na sua versão “0.69.0” possui 671 configurações de formatação de código-

⁵ So yes, absurd as it may sound, fighting over whitespace characters and other seemingly trivial issues of code layout is actually justified. Within reason of course – when done openly, in a fair and concensus building way, and without stabbing your teammates in the face along the way.

Figura 11 – Exemplo da tela de configuração de UniversalIndentGUI



Fonte: Schweitzer (2007)

Nota: O nome da ferramenta é UniversalIndentGUI.

fonte, mas a interface gráfica de Schweitzer (2006), não possui tal número de opções de configuração.

Todas as opções que não aparecem na interface gráfica de Schweitzer (2006), não podem ser utilizadas ao realizar a formatação do código-fonte porque os arquivos de configuração gerados e lidos pela ferramenta não conterão estas opções. Qualquer outra opção adicional inserida manualmente pelo usuário será apagada, assim não podendo ser utilizadas mais opções além das quais a interface gráfica apresenta.

Outra desvantagem da visualização das mudanças, como mostrado na Figura 11, acontece quando o código-fonte de amostra carregado não contém partes que são formatadas pela configuração utilizada na interface gráfica. Nesse caso, não será apresentada nenhuma alteração no código-fonte de visualização, o que pode causar a impressão de que a ferramenta não funciona.

Com o que foi apresentado sobre as ferramentas de formatação de código-fonte, já se pode ter uma visão sobre como funcionam e são as ferramentas de formatação. Ou elas serão configuradas diretamente pelo usuário, editando arquivos de configuração (BERG; GARDNER; MAUREL, 2005b), ou elas terão uma interface gráfica que permita ao usuário visualizar as suas alterações diretamente em uma amostra de

código-fonte (SCHWEITZER, 2006).

Quanto ao seu funcionamento interno, formatadores de código-fonte seguem o processo de tradução como apresentado na Seção 2.1: Compiladores e Tradutores. Diferentemente do usual para um tradutor, um formatador de código-fonte realiza um processo de tradução onde o nível da linguagem de origem e destino é o mesmo. Por exemplo, formatadores de código-fonte realizam a “tradução” de um programa em “C++” para a própria linguagem novamente “C++”.

3.4 OFUSCADORES

Ofuscadores⁶ são formatadores de código-fonte que funcionam com objetivo contrário ao dos *Beautifiers*, veja o Quadro 2. Em vez de melhorar a leitura do código-fonte, sua função é impossibilitar que o código-fonte seja compreendido ou eliminar caracteres desnecessários do código-fonte. Tais técnicas e utilidades para estes tipos de software podem ser encontradas com mais detalhes em Ceccato *et al.* (2008).

Quadro 2 – Exemplo de Ofuscador de Código (DESIGNS, 2003)

Código 4 – Antes do ofuscamento

```

1  for (i=0; i < M.length; i++)
2  {
3      // Adjust position of clock hands
4      var ML = (ns) ? document.layers['nsMinutes' + i]:ieMinutes[i].style;
5
6      ML.top = y[i] + HandY + ( i*HandHeight ) * Math.sin(min) + scroll;
7      ML.left = x[i] + HandX + ( i*HandWidth ) * Math.cos(min);
8  }

```

Código 5 – Depois do ofuscamento

```

1  for(079=0;079<16x.length;079++){var 063=(170)?
2  document.layers["nsM\151\156u\164\145s"+079]:
3  ieMinutes[079].style;063.top=161[079]+076+(079*075)
4  *Math.sin(051)+173;063.left=175[079]+177+(079*176)
5  *Math.cos(051);}

```

Dentre as utilidades dos Ofuscadores está minimizar a quantidade de dados transmitidos entre um Servidor e um Cliente Web ao reduzir o tamanho dos arquivos de linguagens como JavaScript, quando eles são baixados pelos Navegadores de Internet ao acessar um site que utiliza essas linguagens de script. Como exemplo, pode-se imaginar um arquivo com grande volume de comentários. Nesse arquivo, a

⁶ Do inglês, *Obfuscator*, (*Obfuscation (software)*).

transmissão dos comentários pode ser considerada inútil e consequentemente podem removidos. Portanto, fazendo seu ofuscamento pode-se reduzir seu tamanho em até metade ou mais.

Eventualmente, existem linguagens onde o uso de ofuscadores apresenta limitações. Por exemplo, diferente da linguagem JavaScript utilizada no Quadro 2, a sintaxe da linguagem YAML⁷ (ZERVOUDAKIS *et al.*, 2013) obrigatoriamente deve utilizar espaços para indentação e linhas novas para separar blocos de informação. Tais características enfraquecem o poder do ofuscamento uma vez que ele possui um maior efeito quando pode-se introduzir maior caos no documento, removendo-se as indentações e linhas novas. Entretanto, mesmo sem esses elementos chave, ainda pode-se introduzir caos removendo todos os comentários e renomeando variáveis para nomes sem significado de uso como “A88”.

3.5 ADIÇÃO DE CORES

Ao pesquisar sobre formatadores de texto com os recém-apresentados, é comum encontrar trabalhos conhecidos como *Pretty-Printing* ou *Source Code Highlighters*, que fazem a adição de cores à código-fonte para serem melhores visualizados: 1) fazer maior destaque aos elementos mais importantes do código-fonte; 2) fazer menor destaque aos elementos considerados menos importantes; 3) fazer que elementos relacionados ou iguais possam ser facilmente encontrados, fazendo com que eles tenham a mesma cor, ou uma cor muito similar (além de estilos como negrito, itálico e sublinhado).

Editores de texto como Skinner (2015c), permitem que seus usuários escrevam as gramáticas das linguagens nas quais se quer editar seu código-fonte, dentro do editor com suporte a adição de cores ao elementos do texto. Este processo é separado por dois “arquivos de configuração”: 1) o arquivo “.sublime-syntax” que contém propriamente a gramática da linguagem; 2) o arquivo “.sublime-color-scheme” que contém as configurações de estilo para serem aplicados à uma ou mais gramáticas de diversas linguagens. Seguindo convenções de na escrita dos arquivos “.sublime-syntax” (SKINNER, 2015b) e “.sublime-color-scheme” (SKINNER, 2015a), é possível utilizar um único arquivo “.sublime-color-scheme” com diversas gramáticas diferentes (“.sublime-syntax”). Também é possível utilizar um único arquivo de gramática (“.sublime-syntax”) com diversas configurações de cores ou estilos diferentes (“.sublime-color-scheme”).

Como o código-fonte do editor Skinner (2015c) é proprietário, Hume (2016) fez a implementação de uma versão de código-aberto que reconhece os arquivos de gramática (“.sublime-syntax”) e configurações de estilo (“.sublime-color-scheme”) do editor Skinner (2015c). A iniciativa de Hume (2016) aconteceu porque mesmo o editor de

⁷ Do inglês, *YAML Ain't Markup Language*, i.e., a sigla de YAML é uma definição recursiva.

texto Skinner (2015c) sendo de código-fechado, ele possui um grande número gramáticas e arquivos de estilo criados pelos seus usuários. Assim, esse grande número de especificações poderia ser utilizado por outros editores de texto, que gostariam de reutilizar a grande base de gramáticas e estilos já escritas para Skinner (2015c). Hume (2016) diferente de Skinner (2015c), é somente uma biblioteca que pode ser importada por outros editores de texto.

Dima (2017) diferente de Skinner (2015c), é um editor de texto de código-aberto que faz uso do mesmo esquema de gramáticas e arquivos de estilo. Entretanto, os arquivos de configuração e estilo utilizados por Dima (2017) e Skinner (2015c) não são completamente compatíveis, eles são similares. Ambos os trabalhos foram construídos em cima da mesma base, o editor de texto Odgaard (2004). Cada um desses editores seguiu um rumo diferente do original Odgaard (2004) conforme eles foram sendo desenvolvidos.

Skinner (2015c) continua compatível com os arquivos de gramática e estilo originais de Odgaard (2004), mas também fez a criação de novos operadores para especificação das gramáticas das linguagens (para poder especificar mais tipos linguagens de programação). Já Dima (2017), não fez a adição de novos recursos como Skinner (2015c), contudo, ele ainda não é completamente compatível com os arquivos de gramáticas originais devido a *bugs* em sua implementação (AESCHLIMANN, 2017).

3.5.1 Gramáticas

Apesar das suas diferenças, tanto Skinner (2015c), quanto Dima (2017) ou Odgaard (2004), fazem uso do mesmo núcleo de funcionamento para especificação das gramáticas e estilos. O que muda de um para outro é o suporte a recursos mais avançados (como Skinner (2015c) faz) e o formato de representação das informações. Enquanto Odgaard (2004) e Dima (2017) utilizam arquivos no formato XML⁸ (MARTENS *et al.*, 2006) para representar as gramáticas e estilos, Skinner (2015c) faz uso do formato YAML (ZERVOUDAKIS *et al.*, 2013) para gramáticas e JSON para estilos (“.sublime-color-scheme”).

Tanto YAML quanto XML são equivalentes e podem representar a mesma informação sobre as gramáticas especificadas, portanto, tradutores (Seção 2.1: Compiladores e Tradutores) como King e López-Anglada (2011) podem ser escritos sem dificuldade para converter uma gramática de Odgaard (2004) para Skinner (2015c). Mas nem sempre no sentido contrário, uma vez que Skinner (2015c) faz a adição de novos operadores para especificação de suas gramáticas.

Escolheu-se mostrar um exemplo de gramática e estilo utilizados por Skinner (2015c) porque dos três trabalhos: 1) ele é consegue especificar mais diferentes tipos de gramáticas (pelos novos operadores adicionados, como “set” e “branch” (SMITH,

⁸ Do inglês, *Extensible Markup Language*.

2019)) e; 2) utiliza YAML para representar suas gramáticas, tornando a leitura dos arquivos muito mais amigável do que XML, como utilizado pelos outros trabalhos (OD-GAARD, 2004; DIMA, 2017). No Código 6, pode ser encontrado um exemplo funcional de gramática aceita por Skinner (2015c). Mesmo não sendo capaz de descrever uma linguagem em todos os seus detalhes, este arquivo já é suficiente para encontrar as palavras-chaves: 1) “if”; 2) “else”; 3) “for”; 4) e “while”, com o escopo “keyword.control.c” de uma certa linguagem chamada “c”, onde os arquivos dela possuem as extensões “.c” e “.h”.

Código 6 – Exemplo de um arquivo “.sublime-syntax”

```

1 %YAML 1.2
2 ---
3 name: C
4 file_extensions: [c, h]
5 scope: source.c
6
7 contexts:
8   main:
9     - match: \b(if|else|for|while)\b
10    scope: keyword.control.c

```

Fonte: Skinner (2015c)

Uma vez que o Código 6 classificou as palavras-chave com o escopo “keyword.control.c”, um editor de texto precisa somente mais de um arquivo de configuração para determinar como estas palavras-chave devem ser exibidas. No Código 7, encontra-se um exemplo de arquivo JSON de configuração de estilos. Nele, há a definição de três tipos de estilo diferentes. O último deles aplica ao nosso escopo “keyword.control.c”, a cor de texto “#7F00FF” (RGB hexadecimal), com o estilo de fonte “*italico*”.

As demais especializações de escopos do arquivo (Código 7) não foram utilizadas porque a gramática (Código 6), não fez sua utilização. Como arquivos de estilo podem ser utilizados com muitas gramáticas diferentes, e uma gramática pode ser utilizada com muitos arquivos de estilo diferentes, sempre poderão acontecer casos onde nem todas os escopos da gramática serão utilizados e nem todos os escopos dos arquivos de estilo serão utilizados.

Código 7 – Exemplo de um arquivo “.sublime-color-scheme”

```

1 {
2   "name": "Example Color Scheme",
3   "variables":
4   {

```

```

5      "green": "hsla(153, 80%, 40%, 1)",
6      "black": "#111",
7      "white": "rgb(242, 242, 242)"
8  },
9  "globals":
10 {
11      "background": "var(black)",
12      "foreground": "var(white)",
13      "caret": "color(var(white) alpha(0.8))"
14 },
15  "rules":
16 [
17  {
18      "name": "Comment",
19      "scope": "comment",
20      "foreground": "#888888"
21 },
22  {
23      "name": "String",
24      "scope": "string",
25      "foreground": "hsla(50, 100%, 50%, 1)",
26 },
27  {
28      "name": "Operators",
29      "scope": "keyword.control.c",
30      "foreground": "#7F00FF",
31      "font_style": "italic",
32  }
33 ]
34 }
```

Fonte: Skinner (2015a)

Em casos de estrema incompatibilidade, podem acontecer casos onde nenhum estilo pode ser aplicado, porque a gramática e o arquivo de estilo não concordam em nenhum nome de escopo. Por essas e outras, foram criados guias de estilo (SKINNER, 2015b) onde criam-se convenções de nomes de escopos para serem utilizados tanto por quem cria gramáticas, quanto por quem cria arquivos de estilo para editores de texto.

Mas porque existem arquivos de estilo? Para que se possam trocar algumas, senão todas as cores que são exibidas pelo editor de texto. Por exemplo, durante o dia pode-se gostar de utilizar cores de texto mais claras com um fundo branco, enquanto durante a noite, um fundo mais escuro com cores de texto mais claras. Ou ainda,

diferentes programadores possuem diferentes preferências de cores. Assim, todos podem utilizar o mesmo editor de texto somente configurando o editor de texto com as suas preferências de cores.

Neste capítulo, foi apresentado um pouco sobre os formatadores de código-fonte e ferramentas de adição de cores. No próximo capítulo será apresentado um pouco sobre a tentativa de união do estado da arte dos formatadores de código-fonte, junto com o estado da arte das ferramentas de adição de cores recém-apresentadas, na tentativa da criação de formatadores de código-fonte com a flexibilidade de extensão das ferramentas de adição de cores.

4 FORMATADOR DESENVOLVIDO

Neste capítulo, será explicado o funcionamento e implementação de uma nova ferramenta de formatação. A proposta desta nova ferramenta é permitir que usuários possam entrar com a gramática de qualquer linguagem, por meio de uma metagramática¹ para então formatar o código-fonte da linguagem descrita pela gramática.

Para desenvolver este trabalho foi utilizada a linguagem Python (ORTIZ, 2012), porque Python é uma linguagem simples de entender, e permite que se escreva códigos com maior velocidade. Em contra-partida, Python como sendo uma linguagem interpretada, apresentada menor eficiência do que linguagens compiladas como C++. Entretanto, não é objetivo deste trabalho ter eficiência em tempo de execução. Sómente apresentar uma prototipação rápida de algoritmos para uma nova proposta de formatadores de código-fonte.

Como Python é a linguagem de desenvolvimento deste trabalho, escolheu-se utilizar o Analisador Lark porque ele é um dos analisadores escritos inteiramente na linguagem Python. Dentre os motivos para se utilizar o Analisador Lark, pode-se citar: 1) implementa vários algoritmos de análise como LALR (DEREMER; PENNELLO, 1982), Earley (EARLEY, 1970) e CYK (HOPCROFT; MOTWANI; ULLMAN, 2006), podendo reconhecer todas as gramáticas livre de contexto (Seção 2.4: Compiladores e Classes de Complexidade); 2) não possui dependência de nenhuma outra biblioteca; 3) tanto Analisador Léxico quanto Analisador Sintática são totalmente integrados na construção da gramática no formato EBNF, sem a necessidade de escrever nenhum código de programação (Python) adicional para fazer a Análise Léxica ou Sintática; 4) é um projeto ativo de desenvolvimento (SHINAN, 2019a), constantemente recebendo correções de problemas (*bugs*, Bendik, Benes e Cerna (2017) e Steinert *et al.* (2009)) e adição de novos recursos (*features*). No fim, qualquer analisador que atenda estas características de flexibilidade pode ser utilizado.

Para este trabalho, foi realizado um *fork*² (REN; ZHOU; KÄSTNER, 2018; BIAZZINI; BAUDRY, 2014; CELIŃSKA, 2018) do Analisador Lark, renomeado o Analisador Lark (SHINAN, 2014) para “pushdown”³. Foi realizado um *fork* do Analisador Lark para poder realizar pequenas alterações que facilitam o entendimento do funcionamento interno da ferramenta como adição de logs e alterações nos algoritmos de navegação sobre as árvores geradas pela ferramenta. Por isso, em alguns lugares do código-fonte é encontrado o nome “pushdown” ao invés de “lark”. Já em outros, Lark continua sendo

¹ Em Ciências da Computação, quando algo é prefixado com “meta”, isso significa que ele refere-se sobre o seu tipo ou categoria (SCHILD; HERZOG, 1993). Por exemplo, “metadata” são dados sobre os dados.

² Uma cópia do código-fonte, usualmente feita para realizar modificações no código-fonte original.

³ O código-fonte do *fork* pode ser encontrado em <https://github.com/evandrocoan/pushdownparser>.

chamado de Lark para simplificar a retrocompatibilidade com a biblioteca original e facilitar a realização da integração de novas atualizações vindos do repositório original do Analisador Lark para o *fork* realizado.

4.1 VISÃO GERAL

Nesta seção, será simplificadamente explicado como o algoritmo de formatação de código-fonte deste trabalho funciona. Para mais tarde nas próximas seções, poder-se entrar com mais detalhes sobre partes específicas do funcionamento, sem se preocupar com detalhes básicos do funcionamento do formatador de código-fonte. No Código 8, é apresentado um programa completo e funcional na linguagem Java, que imprime “Hello World!” quando chamado sem nenhum argumento de linha de comando (CLI⁴, (SINGER, 2017)). Quando este mesmo programa Java é chamado com qualquer número de argumentos pela linha de comando, ele imprime “Bye World!” na saída padrão.

Código 8 – Exemplo de um programa na linguagem Java

```

1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         if(args.length > 0)
6         {
7             System.out.println("Bye World!");
8         }
9         else
10        {
11             System.out.println("Hello World!");
12        }
13    }
14 }
```

Tendo como o exemplo o código-fonte em Java apresentado no Código 8, irá ser introduzido o algoritmo de formatação proposto de forma “simples⁵”, capaz que pegar algum elemento com uma estrutura como “if(args.length > 0)”, de um programa de entrada em uma linguagem qualquer e realizar a sua formatação. No Código 9, inicia-se com a apresentação de uma simplificação da metagramática utilizada neste trabalho.

⁴ Do inglês, *Command Line Interface*.

⁵ Com “simples”, refere-se a simplificação da Metagramática e Gerador de Formatadores utilizados nesta seção. Para uma apresentação mais complexa, confira a Seção 4.3: Especificação da Metalínguagem.

A metagramática completa pode encontrada no Apêndice H: Código da Metagramática. Ela segue as mesmas regras do Analisador Lark explicadas na Seção 2.2: Gramáticas. Uma melhor descrição do uso dos operadores “match” e “scope” definidos por essa metagramática, podem ser encontrados mais tarde na Seção 4.3: Especificação da Metalinguagem.

Código 9 – Exemplo mínimo da metagramática

```

1 language_syntax: _NEWLINE? ( match_statement | scope_name_statement )+ _NEWLINE?
2
3 match_statement: "match" ":" / .+/ _NEWLINE
4 scope_name_statement: "scope" ":" / .+/ _NEWLINE
5
6 CR: "/r"
7 LF: "/n"
8
9 _NEWLINE: ( /\r?\n[\t ]*/ )+
10 SPACES: /[\t \f]+/
11
12 %ignore SPACES

```

No Código 10, é apresentado um exemplo de gramática válida definida pelas regras da metagramática apresentada no Código 9. Com a gramática do Código 10 é possível reconhecer qualquer programa em que exista a estrura “if” com a seguinte forma “if(alguma coisa)” ou somente “if()”. Uma vez que esses “if’s” sejam reconhecidos, eles serão atribuídos ao escopo “if.statement.body”, que representa a região do código-fonte onde encontra-se o conteúdo do “if”. No exemplo do Código 8 o escopo “if.statement.body” seria equivalente ao trecho de código-fonte “args.length > 0”.

Código 10 – Exemplo de gramática pelas regras da mínima metagramática

```

1 match: (?<=if\().*(?=\\))
2 scope: if.statement.body

```

Agora, o quê pode ser feito com a gramática do Código 10? Ela junto com a metagramática (Código 9) devem ser entrada do Analisador Lark (Código 11), que irá devolver como resultado a Árvore Sintática (Seção 2.1: Compiladores e Tradutores). Então envia-se a Árvore Sintática para o Analisador Semântico (Seções 2.3 e 2.3.4: Analisadores Sintáticos e Análise Semântica) verificar se o programa (gramática) de entrada está semanticamente correta, para então devolver como resultado a Árvore Sintática Abstrata⁶ (Seções 4.4 e 4.5: Formatador de Código e Analisador Semântico).

A Árvore Sintática Abstrata representa a estrutura do programa de uma lingua-

⁶ Do inglês (AST), Abstract Syntax Tree.

gem de programação (KIM *et al.*, 2016; ELMAS *et al.*, 2009). Ela é originalmente construída a partir da Árvore Sintática, adicionado-se informações adicionais sobre o significado programa e removendo-se elementos textuais como parenteses de procedência de operadores, que podem ser representados pela estrutura da árvore. Veja no Apêndice B: main_formatter.py um exemplo de uma Árvore Sintática e uma Árvore Sintática Abstrata e as compare.

No Código 11, é apresentado um programa em Python que faz uso do Analisador Lark recebendo a metagramática e gramática dos Códigos 9 e 10. Este exemplo de uso do Analisador Lark é igual ao já apresentado na Seção 2.2: Gramáticas, com exceção da gramática (aqui representando um metagramática) e o programa de entrada (aqui representando uma gramática).

Código 11 – Exemplo de uso da metagramática e uma gramática

```
1 import pushdown
2
3 parser = pushdown.Lark(
4     r"""
5         language_syntax: _NEWLINE? ( match_statement | scope_name_statement )+
6         → _NEWLINE?
7
8         match_statement: "match" ":" " .+/" _NEWLINE
9         scope_name_statement: "scope" ":" " .+/" _NEWLINE
10
11         CR: "/r"
12         LF: "/n"
13
14         _NEWLINE: ( /\r?\n[\t ]*/ )+
15         SPACES: /[ \t \f]+/
16
17         %ignore SPACES
18         """
19         start='language_syntax',
20         parser="lalr",
21     )
22
23 def parseit(program, filename):
24     pushdown.tree.pydot__tree_to_png(
25         parser.parse( program ), filename, "TB", debug=1, dpi=600 )
26
27 parseit(
28     r""""
```

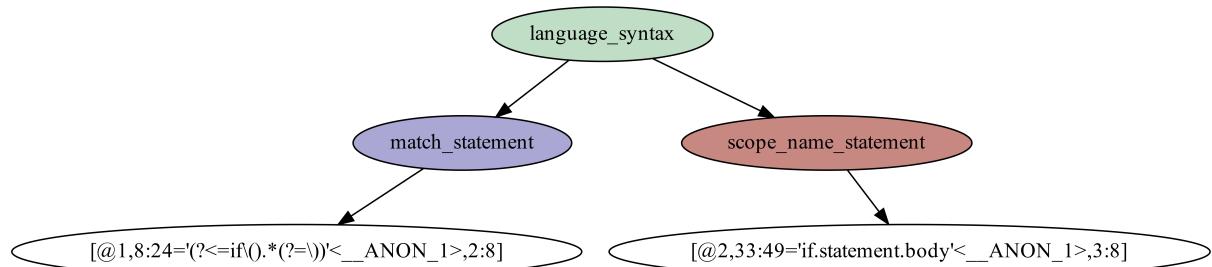
```

28     match: (?<=if\().*(?=\\))
29     scope: if.statement.body
30     """,
31     "arvore_da_gramatica.png"
32 )

```

Na Figura 12 é apresentado a Árvore Sintática da gramática construída no programa Python de exemplo no Código 11. Com uma Árvore Sintática agora pode-se programaticamente navegar pelos nós da árvore e fazer uso do conteúdo dos nós em algum algoritmo específico.

Figura 12 – Árvore Sintática do Código 10 criada pelo Código 11



Fonte: Própria

Assumindo que neste ponto, o programa (a gramática de entrada) já está semanticamente validado (e a Árvore Sintática Abstrata já foi criada), utiliza-se a Árvore Sintática Abstrata da gramática da linguagem do código-fonte (Código 8) para construir algum algoritmo ou estratégia que possa ser utilizada para realizar a formatação do código-fonte inicialmente apresentado no Código 8.

Seguindo a mesma estratégia utilizada por editores de texto em suas gramáticas (Seção 3.5.1: Gramáticas), realizar-se o consumo do programa de entrada, removendo dele as estruturas descritas pela gramática no Código 10, até que não se possa mais remover nenhum elemento novo, terminando a reconhecimento do programa de entrada pela gramática utilizada.

Ao fazer o processo de reconhecimento do programa de entrada, remove-se os caracteres reconhecidos, substituindo eles por algum outro caractere de marcação (como “§”, símbolo de seção⁷) para que não altere-se o tamanho original do programa de entrada. No Código 12 se pode ver como o programa original (Código 8) ficou ao final do processo de reconhecimento pela gramática no Código 10. Essa foi uma estratégia utilizada para se possa no final do processo de formatação se possa facilmente reintroduzir no programa os novos trechos de código-fonte que foram formatados.

Código 12 – Resultado do reconhecimento do programa Java pela gramática

⁷ Do inglês, section sign. Também conhecido sinal de seção ou sinal de corte.

```

1  {
2      public static void main(String[] args)
3      {
4          if($$$$$$$$$$$$$)
5          {
6              System.out.println("Bye World!");
7          }
8          else
9          {
10             System.out.println("Hello World!");
11         }
12     }
13 }
```

Como pode ser percebido, a gramática de entrada no Código 10 não consome todo o programa de entrada (Código 8) no final do processo de reconhecimento (Código 12). Esta é uma característica importante dos formatadores de código deste trabalho. Todo texto que não é consumido, ou pela gramática de entrada, ou pelo formatador de código ou adição de cores, será mantido intacto no final do processo de formatação ou adição de cores. Assim, pode-se ter o formatador de código já em funcionamento com gramática mais simples possível, ou que já atenda as características mínimas que se deseja formatar ou adicionar de cores.

Até este ponto, ainda não se mostrou como a parte mais importante deste trabalho deve acontecer, a formatação de código-fonte. A estratégia deste trabalho foi realizar a formatação de código-fonte ao reconhecer o programa de entrada (Código 8) “removendo” os caracteres reconhecidos, atribuindo escopos (Seção 3.5.1: Gramáticas) para cada um deles. Uma vez que o trecho de código-fonte “removido” possui um escopo atribuído, um arquivo de configurações JSON como o Código 13 é consultado. Caso exista uma “configuração” relacionada ao escopo recém-reconhecido, o trecho de código-fonte é enviado para um formatador especializado de código-fonte como o Código 14.

Código 13 – Exemplo de configuração utilizada mínima do Formatador de Código

```

1  {
2      "if.statement.body" : 2,
3  }
```

O Código 14 é uma especialização de uma classe maior responsável por navegar pela Árvore Sintática Abstrata das gramáticas das linguagens sendo formatadas. A sua função “format_text” será chamada sempre que um escopo for encontrado pela metagramática. Os valores dos parâmetros “code_to_format” e “setting_value” serão o

nome do escopo encontrado pela gramática e o valor da “configuração” correspondente encontrado no arquivo do Código 13.

Código 14 – Exemplo mínimo de Formatador de Código

```

1 class SingleSpaceFormatter(AbstractFormatter):
2
3     def format_text(self, code_to_format, setting_value):
4         code_to_format = code_to_format.strip( " " )
5
6         if setting_value:
7             return " " * setting_value + code_to_format + " " * setting_value
8         else:
9             return code_to_format

```

Por fim, no Código 15 encontra-se o resultado da formatador para comparação com o código-fonte original (Código 8). Como pode-se perceber, esta formatação realizada não foi muito significativa. Entretanto, está-se trabalhando várias simplificações de implementação. Para trabalhos futuros a este (Seção 5.2: Trabalhos Futuros) é necessário criar maiores abstrações que permitam trabalhar com gramáticas de linguagens utilizando uma pilha de múltiplos contextos (Seção 4.3: Especificação da Metalinguagem), com já feito em editores de texto (Seção 3.5: Adição de Cores) que serviram de inspiração a este trabalho.

Código 15 – Resultado do Formatador de Código para Java

```

1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         if( args.length > 0 )
6         {
7             System.out.println("Bye World!");
8         }
9         else
10        {
11             System.out.println("Hello World!");
12        }
13    }
14 }

```

Nas próximas seções, será explicado em pouco mais de detalhes o mesmo funcionamento recém-apresentado brevemente. Para maiores detalhes, consulte a

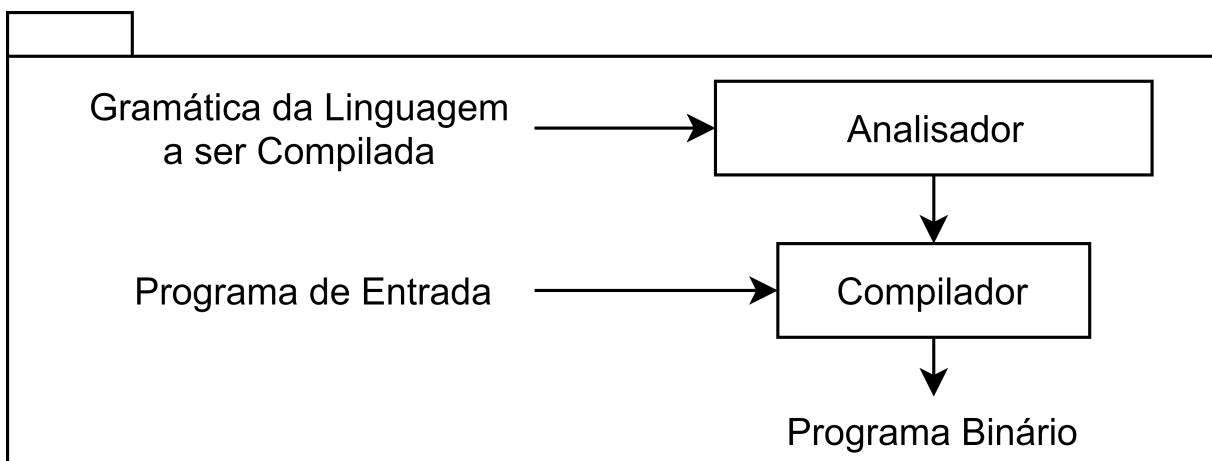
implementação e exemplos de uso destes algoritmos nos Apêndices B e G: main_formatter.py e Código do Formatador.

4.2 INTRODUÇÃO À METAGRAMÁTICA

Na Seção 2.2: Gramáticas, foi explicado o que são gramáticas. Mas, como gramáticas podem ser expressadas? Isso depende de como seu analisador foi implementado. Analisadores seguem uma notação comum como EBNF (PATTIS, 1994; PARR, 2013; SHINAN, 2019b; 2018), que diverge de acordo com detalhes de implementação.

Para realizar a implementação da nova ferramenta de formatação de código-fonte, foi realizado a construção de uma nova gramática de gramáticas de uma nova linguagem chamada de “ObjectBeauty”, uma metalinguagem (BOOK; SHORRE; SHERMAN, 1970). Na Figura 13, é apresentado o fluxo de uso comum para um analisador. Neste processo, o desenvolvedor da linguagem escreve a gramática de especificação que é entregue a algum analisador e gera-se um compilador para tal linguagem.

Figura 13 – Fluxo de uso comum de um analisador



Fonte: Própria, traduzido e adaptado de Coan (2018)

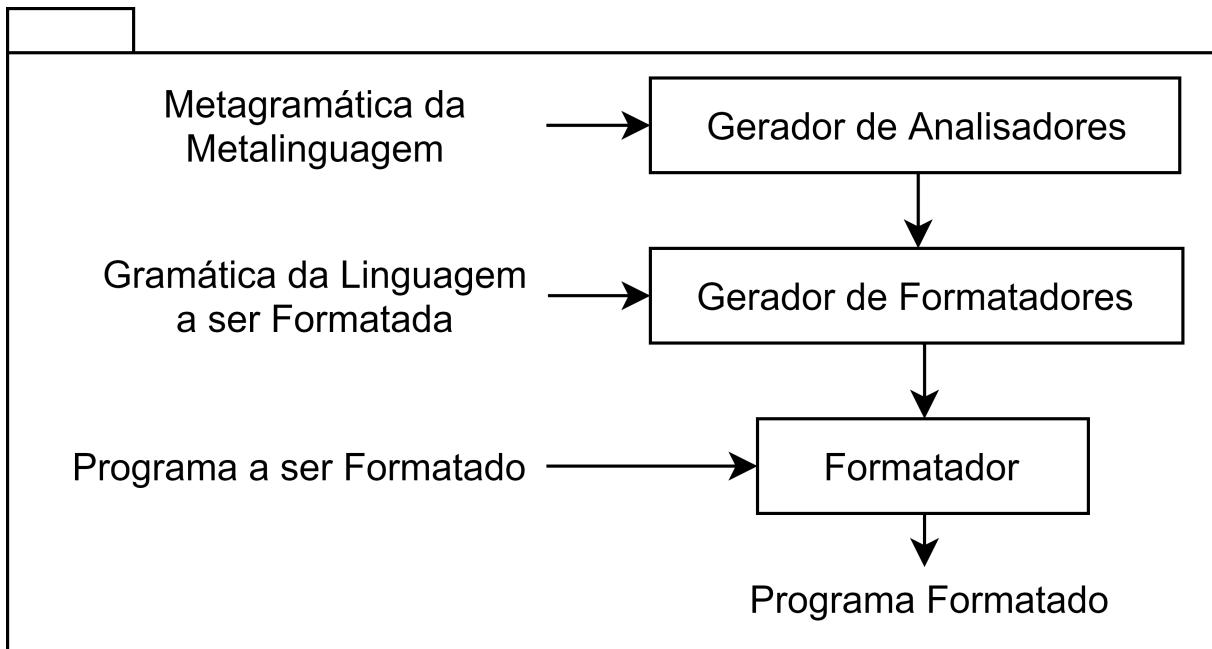
Nota: Própria, traduzido e adaptado, são figuras desenvolvidas pelo autor, mas inicialmente publicadas em outro local, idioma e com o conteúdo um pouco diferente.

Com algumas exceções, compiladores são usualmente construídos utilizando um Compilador de Compiladores (ou Analisadores) (BOOK; SHORRE; SHERMAN, 1970). Utilizando um Analisador (Figura 13), é escrita a gramática da linguagem para qual se quer construir um compilador (Seção 2.2: Gramáticas). Após isso, o Analisador gera o código-fonte do analisador léxico e sintático que aceita como entrada o programas da linguagem a ser compilada. Então, uma vez que os analisadores léxico e sintático terminam seu trabalho, tem-se como resultado a Árvore Sintática do programa de entrada.

A Árvore Sintática é utilizada então para realizar a Análise Semântica e geração de código, completando-se a construção de um compilador. Para o processo de

construção de um compilador, não basta somente utilizar-se de um Analisador, mas também é necessário a escrita do código-fonte do Analisador Semântico e a geração de código binário executável (Seção 2.1: Compiladores e Tradutores). Este trabalho faz um uso diferente. Como mostrado na Figura 14, primeiro especifica-se uma metalinguagem que será utilizada pelos usuários da nova ferramenta de Formatação de Código. Para escrever esta nova metalinguagem, utilizou-se o Analisador Lark.

Figura 14 – Uso feito pela nova ferramenta de Formatação de Código



Fonte: Própria, traduzido e adaptado de Coan (2018)

Diferente da Figura 13, na Figura 14 é adicionado mais uma etapa ao uso do Analisador. Antes existia somente o fluxo de comunicação “Analizador -> Compilador”, agora o fluxo de informações segue como “Gerador de Analisadores -> Gerador de Formatadores -> Formatador”. Neste novo fluxo, “Gerador de Analisadores” é o equivalente ao antigo “Analizador” e “Formatador” é o equivalente ao antigo “Compilador”. Entre eles, foi adicionado o “Gerador de Formatadores”, que é a criação de um novo tipo de Analisador de propósito específico, para uma DSL⁸ (MICHAELSON, 2016; ZERVOUDAKIS *et al.*, 2013), uma linguagem de propósito específico (neste caso, a linguagem de uma gramática, uma metalinguagem).

Esta nova DSL trata-se de uma especificação de gramáticas⁹, baseada nas gramáticas já utilizadas por editores de texto para adicionar cores aos códigos-fontes exibidos em sua interface gráfica (Seção 3.5.1: Gramáticas). Diferente das gramáticas

⁸ Do inglês, *Domain-Specific Language*.

⁹ Assim como DSL's, estas gramáticas também são de propósito específico e não poderiam ser utilizadas para descrever linguagens de propósito geral, i.e., a criação de outras metagramáticas de propósito geral (KIRPICHOV, 2014).

já utilizadas por editores de texto para adicionar cores, é criada uma nova especificação de gramáticas para que no futuro ela possa conter recursos específicos para formatação de código-fonte¹⁰ (e não adição de cores), e para que ela não seja dependente de características de linguagens como YAML ou XML utilizada pelos editores de texto para especificação de suas gramáticas (Seção 3.5: Adição de Cores).

Usualmente, o Analisador Lark é utilizado somente como um gerador de compiladores (Figura 13), entretanto, neste contexto Lark é utilizado como um “Compilador de Compiladores” (Figura 14). A vantagem de utilizar outro analisador (Lark) para criar outro analisador (Gerador de Formatadores), é que a gramática de especificação do Gerador de Formatadores pode usufruir da flexibilidade que a utilização que analisadores trazem na especificação das gramáticas de uma linguagem. A facilidade em alterar a gramática da linguagem utilizada, neste caso, a gramática de gramáticas (ou metagramática).

Na Figura 15, pode-se encontrar uma relação entre o funcionamento das diversas partes da ferramenta de Formatação de Código e a audiência alvo. Basicamente existem três grupos distintos de usuários ou audiência: 1) quem escreve ou desenvolve a ferramenta de Formatação de Código proposta por este trabalho e define as regras da metalinguagem (especificada pela sua metagramática, i.e., a gramática de gramáticas); 2) quem escreve ou desenvolve gramáticas de linguagens para serem formatadas de acordo com as regras da metalinguagem e; 3) quem escreve ou desenvolve programas de computador e deseja realizar a formatação de seus códigos-fonte.

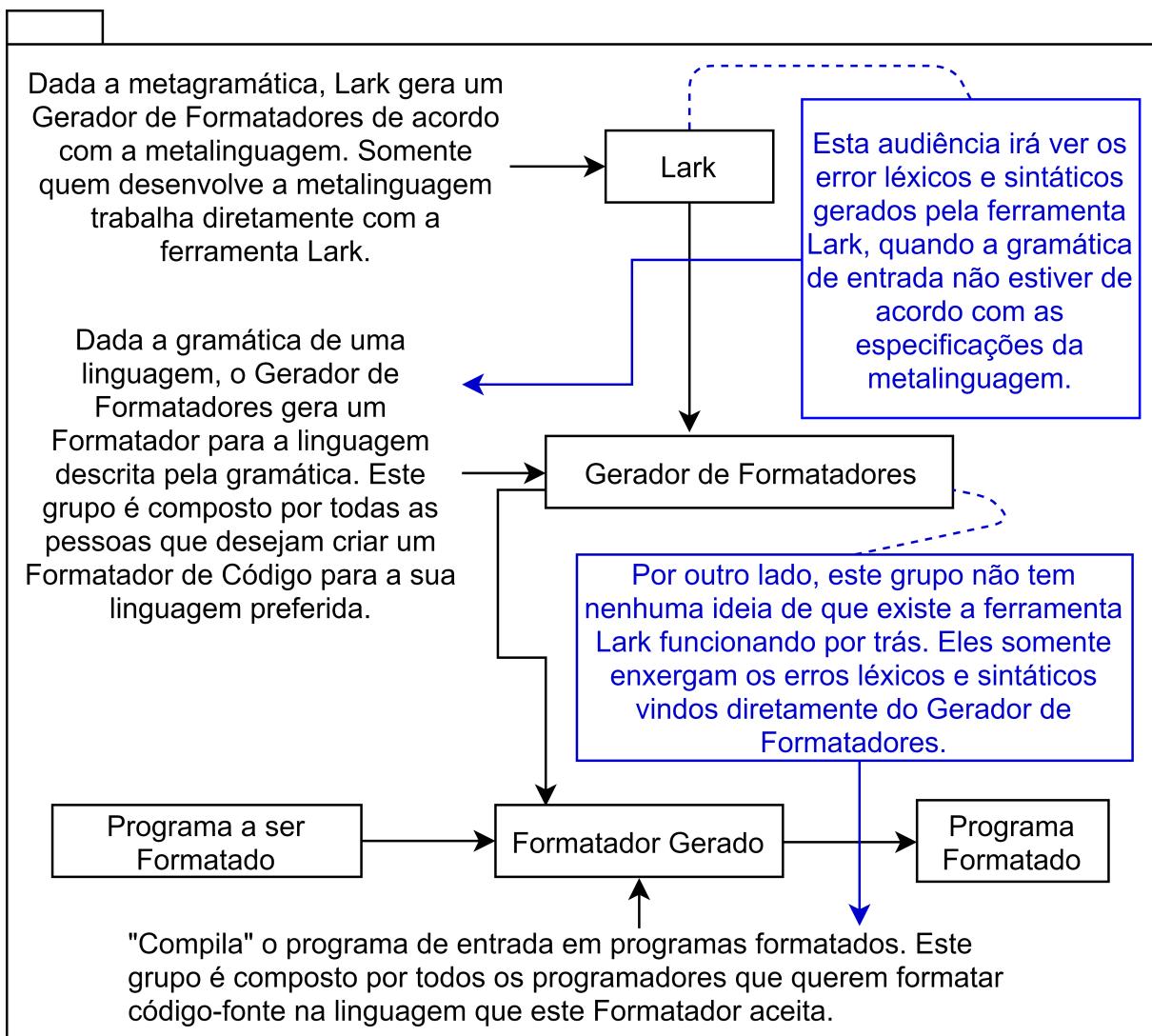
Até este ponto, já falou-se de metagramática e metalinguagem com a exceção dos metaprogramas (VELDHUIZEN, 2006). Nas Figuras 14 e 15, por simplificação foram omitidos o relacionamento dos metaprogramas com a metagramática e metalinguagem. Metaprogramas fazem parte da entrada do metacompilador (Figura 16) junto com a metagramática para gerar um novo compilador (ou Formatador de Código). Neste trabalho, os metaprogramas serão as gramáticas que serão utilizadas pelos formatadores de código-fonte.

Os metaprogramas (ou gramáticas) são entradas diretas do metacompilador, o Analisador Lark na Figura 14, utilizado para analisar uma gramática LALR(1) (Apêndice H: Código da Metagramática). Na Figura 15, não pode-se ver diretamente que as gramáticas das linguagens serão os metaprogramas, mas o quadro em azul mais à esquerda ligado por linhas pontilhadas explica-se que os erros léxicos e sintáticos nas gramáticas de entrada serão mostrados pelo Analisador Lark. Isso acontece porque as gramáticas (ou metaprogramas) são entradas diretamente no Analisador Lark.

Na Figura 16, encontra-se uma extensão da Figura 14, e pode-se ver claramente as relações entre Metagramáticas, Metacompiladores e Metaprogramas. Por simplifi-

¹⁰ Não é objetivo deste trabalho fazer a análise completa de programas, pela sua sintaxe, semântica, e gerar código-binário executável (Seção 2.1: Compiladores e Tradutores).

Figura 15 – Relacionamentos entre os diferentes públicos deste projeto

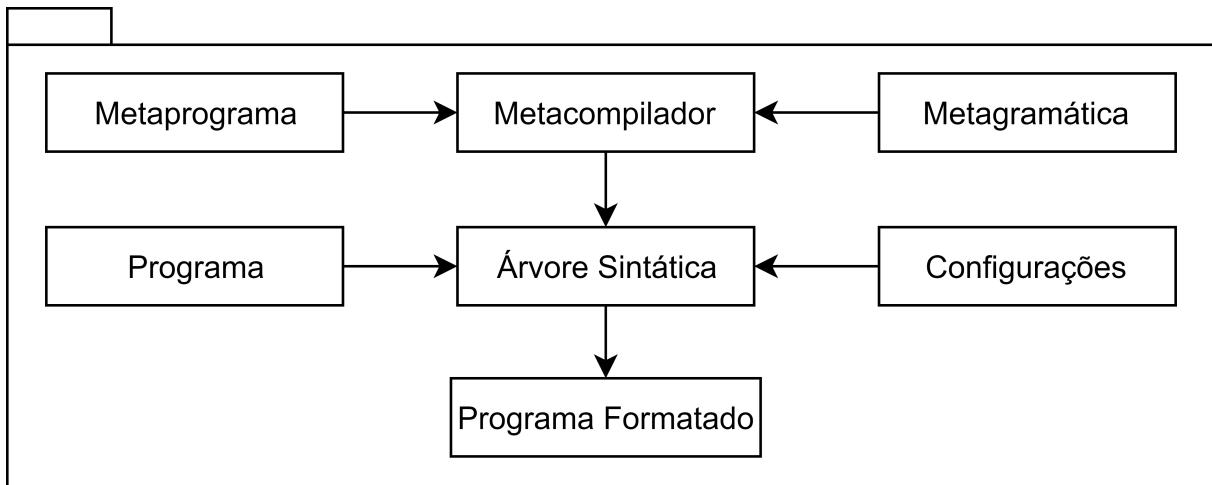


Fonte: Própria, traduzido e adaptado de Coan (2018)

cação, mostra-se o nó “Árvore Sintática” sem explicitamente falar sobre sua Análise Semântica e propriamente a construção do Compilador (ou do Formatador de Código). Vale lembrar que trata-se de um Compilador de Analisadores de uma DSL, e não um Compilador de Analisadores de propósito geral. Isso significa que as gramáticas aceitas pelos analisadores gerados não são capazes de gerar analisadores de linguagens de propósito geral ou específico.

Esta não é a primeira vez que uma metagramática com simplificações foi escrita. Em trabalhos relacionados com esta implementação de metagramática (Seção 3.5: Adição de Cores), foram realizadas as mesmas simplificações aqui apresentadas. Existem algumas diferenças técnicas da metagramática deste trabalho com as dos recém-apresentados. Como por exemplo, a implementação da metagramática realizada ainda não suporta a classificação do mesmo trecho de código-fonte por múltiplos tipos de escopo (DIMA, 2017).

Figura 16 – Relação entre Metagramáticas, Metacompiladores e Metaprogramas



Fonte: Própria, traduzido e adaptado de Coan (2018)

Foi escolhida a criação de uma nova metagramática porque as implementações de metagramáticas já existentes como Hume (2016) e Dima (2017): 1) não utilizam explicitamente nenhum analisador, realizando a programação das produções da gramática diretamente no código-fonte (Seção 2.3.1: Gramáticas *versus* Linguagens); 2) não são capazes de reconhecer todas as características de todas as linguagens de programação (devido a optimizações para maior desempenho); 3) não possuem sintaxe própria, i.e., utilizam-se de outras linguagens como YAML, XML e JSON¹¹ (PEZOA *et al.*, 2016) para fazer a especificação da metagramática. Fazendo a especificação de uma nova metagramática, é possível adaptar-se a especificação da sintaxe das gramáticas de acordo com as necessidades específicas sem ter que depender de características de outras linguagens como YAML, XML ou JSON.

Na Figura 16, percebe-se que os Metaprogramas (ou gramáticas) são entradas diretas dos Metacompilador, e não do Formatador de Código como mostrado nas figuras anteriores (Figuras 13 a 15). Ambas as figuras estão corretas, entretanto, a Figura 16 é uma versão mais resumida de como o processo de análise léxica e sintática acontece. Analisadores sempre precisam como entrada ter uma gramática e a palavra de uma dada linguagem (que pode ou não ser aceita pelo analisador).

Como já explicado no início desta seção (Seção 4.2: Introdução à Metagramática), a ligação direta entre a gramática e a palavra da linguagem não precisa acontecer diretamente com o analisador da Figura 16. Caso o analisador suporte, ao contrário de pedir para que o analisador gere o código-fonte de um Analisador Léxico e Sintático (e funcione como um Compilador de Compiladores), ele pode ser utilizado diretamente recebendo como entrada a gramática da linguagem e uma palavra para análise. Retornando uma Árvore Sintática como resultado do processo ou erro de análise caso a

¹¹ Do inglês, *JavaScript Object Notation*.

palavra de entrada não pertença a linguagem.

A existência do processo de receber somente como entrada a gramática de uma linguagem e gerar código-fonte de um Analisador Léxico e Sintático, capaz de receber um programa de entrada e gerar uma Árvore Sintática, é o que caracteriza um “Analizador” como um “Compilador de Compiladores”. Quando um “Analizador” não é capaz de gerar código-fonte de um Analisador Léxico e Sintático somente a partir de uma gramática de entrada, ele não é um “Compilador de Compiladores”. Trata-se somente de um “Analizador”, que então precisa receber como entrada diretamente a gramática e a palavra da linguagem a ser analisada¹². Para então devolver como resultado a Árvore Sintática do programa de entrada ou um erro de análise, caso programa de entrada não pertença a linguagem da gramática.

4.3 ESPECIFICAÇÃO DA METALINGUAGEM

Como já explicado na seção anterior (Seção 4.2: Introdução à Metagramática), uma metagramática é gramática de gramáticas e foi utilizado o Analisador Lark como um “Metacompilador” ou “Compilador de Compiladores”. Nesta seção será discutido como a metalinguagem (especificada pela metagramática) utilizada foi construída, começando com o seu símbolo inicial. No Código 16, a metagramática define que o programa é constituído de três grandes áreas, que devem acontecer uma em sequência da outra:

1. A produção “preamble_statements” define características globais da gramática como um nome, e um escopo que será atribuído a toda gramática;
2. A produção “language_construct_rules” define qual será o símbolo inicial da gramática. Em comparação com linguagens de programação como “C”, ele pode ser considerado similar ao método “main”;
3. A produção “miscellaneous_language_rules” permite a definição de diversos contextos¹³ com grupos de produções da gramática (Item 3: Gramáticas), que podem ser incluídos a partir do símbolo inicial da gramática definido no item “language_construct_rules”.

Código 16 – Símbolo Inicial da Metagramática “ObjectBeauty” (Apêndice H)

¹² A título de curiosidade, o Analisador Lark deste trabalho foi utilizado como um Analisador, entretanto Lark é capaz de funcionar tanto como um Analisador quanto como um “Compilador de Compiladores” (PARR, 2013).

¹³ Na regras da gramática, contexto refere-se a um bloco de operadores ou conjunto de instruções como “include” e “match”.

```

1 language_syntax: _NEWLINE? preamble_statements _NEWLINE?
2                     language_construct_rules _NEWLINE?
3                     ( miscellaneous_language_rules _NEWLINE? )* _NEWLINE?
4
5 preamble_statements: ( (
6                         target_language_name_statement
7                         | master_scope_name_statement
8                         | constant_definition
9                     ) _NEWLINE )+
10
11 language_construct_rules: "contexts" ":" indentation_block
12 miscellaneous_language_rules: /[^\n]+/ ":" indentation_block
13
14 target_language_name_statement: "name" ":" free_input_string
15 master_scope_name_statement: "scope" ":" free_input_string
16
17 indentation_block: enter_block _NEWLINE ( statements_list _NEWLINE )+ leave_block
18 statements_list: match_statement | include_statement | push_statement
19             | pop_statement | constant_definition | scope_name_statement
20             | capturing_block | meta_scope_statement
21
22 enter_block: OPEN_BRACE
23 leave_block: CLOSE_BRACE
24 OPEN_BRACE: "{"
25 CLOSE_BRACE: "}"
26
27 ...

```

Entre os Códigos 17 a 20, encontram-se pequenos exemplos de gramáticas escritas na metalinguagem “ObjectBeauty” brevemente apresentada. No Código 17, encontra-se a definição do símbolo inicial da gramática da linguagem sendo descrita (pela metagramática) e pode-se ver a metalinguagem sendo utilizada para definir uma linguagem chamada de “Abstract Machine Language”. Por padrão, toda gramática “ObjectBeauty” precisa ter um contexto inicial ou símbolo inicial chamado de “contexts”.

Como já inicialmente apresentado na Seção 3.5.1: Gramáticas, na Figura 17 é mostrado na primeira linha o trecho de código-fonte “function f1 () {}” e nas demais linhas são apresentados as diversas classificações de escopo aplicados a cada um dos trechos do código-fonte de amostra. Por exemplo, a palavra “function” possui simultaneamente a pilha de escopos: 1) “source.js”; 2) “meta.function.js” e; 3) “storage.type.function.js”. Enquanto o caractere “{” (chave de abertura de bloco) possui simultaneamente a pilha de escopos: 1) “source.js”; 2) “meta.function.js”; 3) “meta.block.js” e; 4) “punctuation.definition.block.js”.

Figura 17 – Exemplo de classificação de código-fonte com múltiplos escopos

function	f1	()	{
source.js			
storage.type.function.js	meta.definition.function.js	meta.parameters.js	meta.block.js
entity.name.function.js	punctuation.definition.parameters.js	punctuation.definition.block.js	

Fonte: Dima (2017)

Os nomes utilizados na Figura 17, podem ser qualquer texto que usuário especificador daquela gramática atribuiu. Entretanto, pode-se perceber que o nome dos escopos recém apresentados parecem seguir um padrão. Por conversão, desenvolvedores de gramáticas para os editores de texto como Skinner (2015c) e Dima (2017), seguem uma conversão de nomes para que as utilizações dos escopos gerados pelas gramáticas sejam compatíveis entre si.

Fazendo o uso de uma conversão para nomes de escopos, as gramáticas ficam compatíveis com um maior número de arquivos de temas (ou configurações de cores), onde são especificados os nomes dos escopos serão utilizados para especificar as cores a serem utilizadas pelo editor de texto (Seção 3.5.1: Gramáticas).

O Código 17, faz uso dos operadores “include” e “match”. O operador “include” serve para incluir partes de outras gramáticas ou mesmo gramáticas inteiras no contexto da gramática atual. Entretanto, a implementação de “include” realizada neste trabalho somente consegue realizar includes de contextos definidos no mesmo arquivo. No exemplo do Código 17, o operador “include” está incluindo contextos (Seção 4.3: Especificação da Metalinguagem) da gramática atual que serão definidos mais tarde neste mesmo arquivo. Já o operador “match” utilizado no final serve para realizar propriamente o reconhecimento do programa de entrada e atribuir a ele o escopo “constant.boolean.language.pawn”.

Mais tarde, as informações de escopo atribuídas por operadores como “match” e “captures” serão utilizadas pelo formatador de código-fonte. Com estas informações, o Formatador de Código será capaz de realizar as operações de formatação somente sobre os trechos de código-fonte que o usuário definir. Realizando assim, a formatação seletiva de código-fonte, contrário da formatação total de código-fonte como acontece nos demais trabalhos (Seção 4.5: Formatador de Código).

Código 17 – Exemplo de Gramática, Símbolo Inicial

```

1 name: Abstract Machine Language
2 scope: source.sma
3
4 contexts: {
5     include: parens
6     include: numbers

```

```

7     include: check_brackets
8
9     match: (true|false) {
10         scope: constant.boolean.language.pawn
11     }
12 }
```

No Código 18, é introduzido o uso dos operadores “push”, “meta_scope” e “pop”. O operadores “push” e “pop” são responsáveis por manter uma pilha de contextos que permite aplicar um mesmo escopo por várias linhas utilizando o operador “meta_scope”. A diferença entre o operador “scope” e “meta_scope” é que o operador “scope” atribuí o escopo diretamente ao texto reconhecido pelo operador “match”. Já o operador “meta_scope” permite aplicar o escopo a todo o texto desde o primeiro até o último “match”, que desempilha com o operador “pop”, o contexto empilhado inicialmente com o operador “push”.

Código 18 – Exemplo de Gramática, Contextos

```

1 parens: {
2     match: \( {
3         scope: parens.begin.pawn
4         push: {
5             meta_scope: meta.group.pawn
6             match: \) {
7                 scope: parens.end.pawn
8                 pop: true
9             }
10            include: numbers
11        }
12    }
13 }
```

No Código 19, é introduzido o uso do operador “captures”. O operador “captures” atribuí simultaneamente diversos escopos com uma única expressão regular. Cada um dos números listados equivalem a um dos grupos de captura da expressão regular utilizada no operador “captures”. O operador “scope” pode ser considerado um caso especial do operador “captures” quando utiliza-se o Grupo de Captura 0.

Motores de expressões regulares geralmente suportam um recurso chamado de Grupos de Captura (YAMAGUCHI; KURAMITSU, 2019). Por exemplo, a expressão regular “foo(bar)zoo(car)” possuí 3 grupos de captura quando analisado o texto de entrada “foobarzoocar”: 0) foobarzoocar; 1) bar; 2) car; onde o grupo de captura 0 refere-se a toda a expressão regular encontrada. Portanto, ao invés de utilizar-se o operador “scope: constant.numeric.pawn”, poderia-se utilizar equivalentemente o operador

“captures: 0. constant.numeric.pawn”.

Código 19 – Exemplo de Gramática, Grupos de Captura

```

1 numbers: {
2     match: '(\d+)(\.\{2\})(\d+)' {
3         captures: {
4             0: constant.numeric.pawn
5             1: constant.numeric.int.pawn
6             2: keyword.operator.switch-range.pawn
7             3: constant.numeric.int.pawn
8         }
9         include: numeric
10    }

```

No Código 20, são apresentados mais alguns exemplos de uso do operador “match” classificando diversos tipos de numéricos (da linguagem sendo descrita pela gramática). É importante notar que a ordem na qual os operadores como “match” aparecem é importante. Ao realizar o reconhecido o programa de entrada utilizando esta gramática, a Árvore Sintática Abstrata (AHO; LAM *et al.*, 2006) será interpretada diversas vezes, partindo do símbolo inicial até chegar ao último símbolo da gramática.

O processo de interpretação irá reiniciar indefinidamente até que nenhum texto seja mais consumido por nenhum dos operadores da gramática. Assim, uma vez que um trecho de código-fonte já foi classificado, ele será ignorado quando os próximos operadores forem aplicados, evitando assim que o programa execute infinitamente.

Código 20 – Exemplo de Gramática, Tipos numéricos

```

1 numeric: {
2     match: ([-]?0x[\da-f]+) {
3         scope: constant.numeric.hex.pawn
4     }
5     match: \b(\d+\.\d+)\b {
6         scope: constant.numeric.float.pawn
7     }
8     match: \b(\d+)\b {
9         scope: constant.numeric.int.pawn
10    }
11 }

```

Por fim, no Código 21, é apresentado um exemplo não relacionado com formatação de código-fonte. A construção utilizada é comum para gramáticas que serão utilizadas para realizar a aplicação de cores em editores de texto (DIMA, 2017). Com ela é possível colorir o código-fonte, destacando-o como inválido no editor de texto, uma

vez que uma inconsistência sintática foi encontrada na linguagem sendo analisada.

Construções como a do Código 21, funcionam usualmente quando elas são a última regra da gramática. Uma vez que todas as regras que consomem o programa de entrada e o classifica em escopos terminam seu trabalho, não deveria existir mais nenhum texto ser reconhecido. Caso exista, ou a gramática não estava preparada para reconhecer todo o programa de entrada, ou estes trechos de código-fonte são frutos de algum erro no programa de entrada.

Código 21 – Exemplo de Gramática, Reconhecimento de Erros

```

1 check_brackets: {
2     match: \) {
3         scope: invalid.illegal.stray-bracket-end
4     }
5 }
```

4.4 ANALISADOR SEMÂNTICO

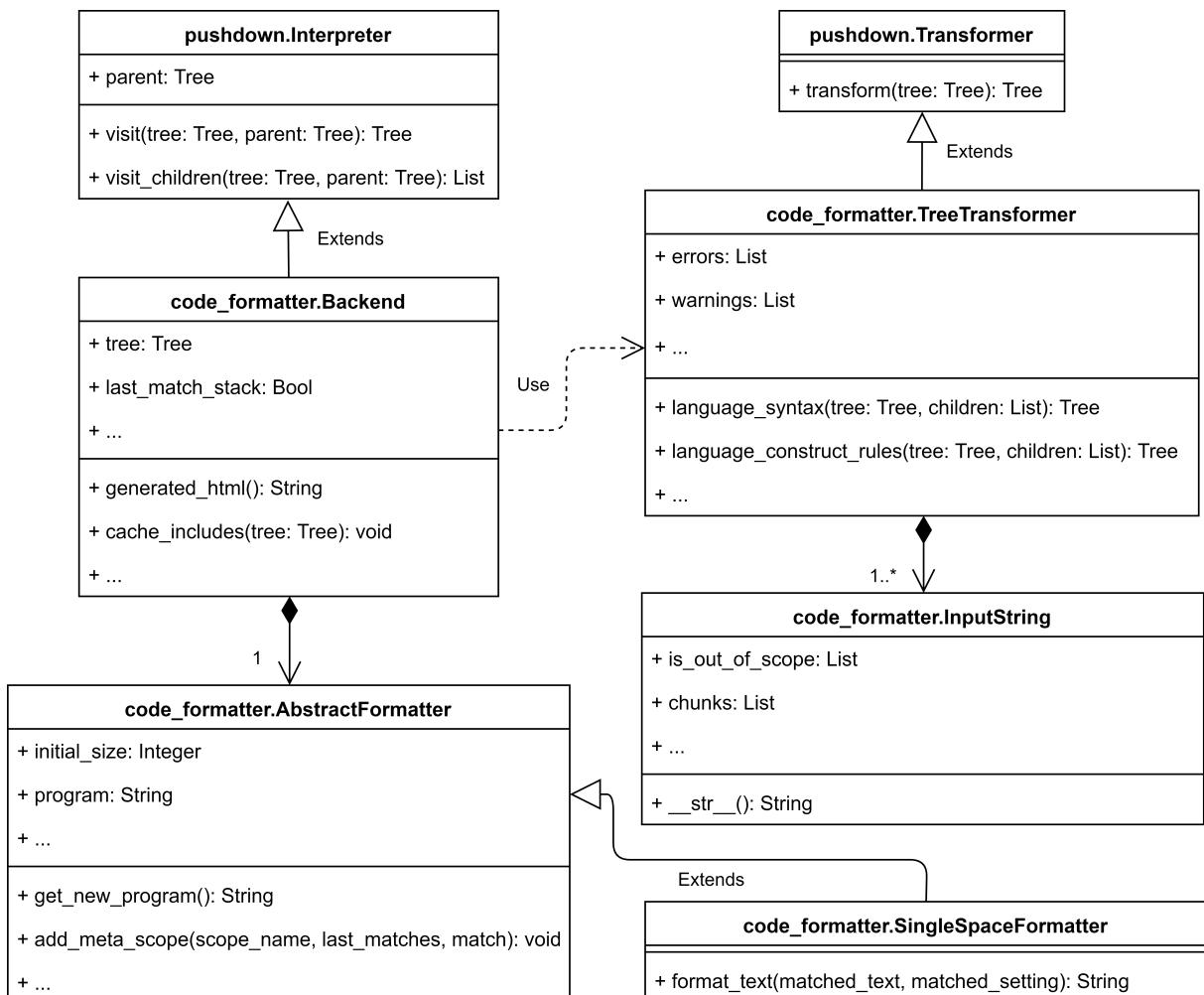
Depois que um metaprograma da metalinguagem apresentada na seção anterior (Seção 4.3: Especificação da Metalinguagem) é reconhecido pelo Analisador Lark, o Analisador Lark entrega a Árvore Sintática da gramática da linguagem sendo descrita pelo metaprograma (Figura 16). Todas as verificações de corretude da sintaxe da gramática são executadas pelo Analisador Lark, com base na metagramática da metalinguagem. Portanto, somente resta ser implementado o Analisador Semântico para verificar se a linguagem descrita respeita as regras semânticas da metalinguagem explicadas na seção anterior (Seção 4.3: Especificação da Metalinguagem).

Um Diagrama de Classes é apresentado na Figura 18. Nele, o Analisador Semântico que deriva de “Transformer” recebe como entrada a Árvore Sintática do programa de entrada, e uma vez que o Analisador Semântico termina seu trabalho, ele devolve Árvore Sintática Abstrata completa. Então, utilizando a Árvore Sintática Abstrata, “Backend” que deriva de “Interpreter”, realiza a formatação de código-fonte recebendo um programa de entrada e as configurações do formatador (Figuras 15 e 16).

No Código 22, pode-se ver o construtor do Analisador Semântico. Pode-se notar que seu construtor não recebe como parâmetro a Árvore Sintática. Entretanto, a ela não é passada pelo construtor mas sim por um método chamado “transform(tree)”. Esta é uma característica do Analisador Lark utilizado. A função “transform(tree)” do Analisador Lark simplesmente inicia a análise do programa visitando todos os nós da Árvore Sintática, partindo das folhas até chegar na raiz (Seção 2.1: Compiladores e Tradutores).

Código 22 – Construtor do Analisador Semântico (Apêndice E)

Figura 18 – Diagrama das principais classes



Fonte: Própria

```

1  class TreeTransformer(pushdown.Transformer):
2      """
3          Transforms the Derivation Tree nodes into meaningful string representations,
4          allowing simple recursive parsing and conversion to Abstract Syntax Tree.
5      """
6
7      def __init__(self):
8          ## Saves all the semantic errors detected so far
9          self.errors = []
10
11         ## Saves all warnings noted so far
12         self.warnings = []
13
14         ## Whether the mandatory/obligatory global scope name statement was declared
15         self.is_master_scope_name_set = False
16
  
```

```

17     ## Whether the mandatory/obligatory global language name statement was
18     ## declared
19     self.is_target_language_name_set = False
20
21     ## Can only be one scope called `contexts`
22     self.has_called_language_construct_rules = False
23
24     ## Pending constants declarations
25     self.constant_usages = {}
26
27     ## Pending constants usages
28     self.constant_definitions = {}
29
30     ## A list of miscellaneous_language_rules include contexts defined for
31     ## duplication checking
32     self.defined_includes = {}
33
34     ## A list of required includes to check for missing includes
35     self.required_includes = {}
36
37     ## A list of regular expressions used on match statements,
38     ## for validation when the constants definitions are completely known
39     self.pending_match_statements = []
40
41     ## Responsible for calculating all open and close commands scoping
42     self.open_blocks = {}
43     self.indentation_level = 0
44     self.indentation_blocks = []

```

A visita dos nós da Árvore Sintática pela função “transform(tree)” acontece simplesmente chamando os métodos que a classe “TreeTransformer” (Código 22) define e que possuem os mesmos nomes que os símbolos não-terminais definidos pela metagramática. Então, cada nó ou função deve devolver qual será o novo nó que irá substituir na Árvore Sintática. Assim, no final do processo, um a um, cada nó da Árvore Sintática será convertido para um nó da Árvore Sintática Abstrata.

Um jeito fácil de excluir um nó da Árvore Sintática é simplesmente definir uma função com o nome de seu não-terminal que devolve “null”. Por exemplo, o trecho da metagramática apresentado no Código 16, possui alguns símbolos não-terminais como “preamble_statements” e “language_construct_rules”. Então, para estes dados símbolos, serão chamados os métodos da classe “TreeTransformer” que possuem os nomes “preamble_statements” e “language_construct_rules”.

Caso não existam os métodos “preamble_statements” e “language_construct_rules”

na classe “TreeTransformer”, os nós “preamble_statements” e “language_construct_rules” da Árvore Sintática não serão visitados e “poderão” ser excluídos da Análise Semântica (mas não da Árvore Sintática Abstrata). Nós não analisados pela classe “TreeTransformer” não serão excluídos da Árvore Sintática Abstrata, eles serão mantidos intactos, a não ser que algum outro nó os altere diretamente na Árvore Sintática.

Mesmo que na classe “TreeTransformer” não existam definidos os métodos “preamble_statements” e “language_construct_rules”, eles também podem ser visitados diretamente a partir de algum nó pai ou até de algum de seus filhos. Inclusive, esta foi uma das alterações realizadas no fork “pushdown” do Analisador Lark. Por padrão, ao navegar pela árvore, o Analisador Lark somente passa como parâmetro da função (de “TreeTransformer”) o nó correspondente ao método atualmente sendo chamado e uma lista de nós-filhos. Entretanto, “pushdown” agora também passa um terceiro parâmetro que é uma referência para o nó raiz da árvore e mantém a variável “parent” (que aponta para o pai do nó atual da árvore), acessível como um atributo da classe “TreeTransformer”.

4.5 FORMATADOR DE CÓDIGO

O Formatador de Código não é composto somente por uma parte única e altamente acoplada. Mas pelo contrário, um conjunto de partes altamente coesas e completamente independentes onde cada uma dessas partes recebe a Árvore Sintática Abstrata, para então realizar a formatação do programa de entrada junto com as configurações que esta parte aceita.

Continuamente chamar diversos algoritmos independentes acarreta uma perda de desempenho em comparação com os formatadores de código-fonte apresentados no Capítulo 3: Estado da Arte. Estes formatadores tem como principal características realizar a formatação de código-fonte em uma única passada, i.e., a Árvore Sintática de programa de entrada é completamente reconstruída, para então ser serializada novamente em texto de acordo com as configurações de formatador.

Este trabalho permite que o usuário entre com a gramática da linguagem a ser formatada, diferente de outros trabalhos onde a gramática já é incluída ao código-fonte do formatador. Formadores de código-fonte como apresentados na Capítulo 3: Estado da Arte são construídos programando-se as produções da gramática diretamente no código-fonte do formatador (Descendentes Recursivos, reveja a Seção 2.3.1: Gramáticas *versus* Linguagens).

Ao não permitir-se que o usuário possa entrar com a gramática do programa, restringe-se o formatador a somente funcionar com as gramáticas que foram programadas dentro do seu código-fonte. Uma vez que o usuário da ferramenta precisa programar o formatador de código para ter suporte a sua linguagem, isso dificulta a adição do suporte de novas linguagens ao formatador, pois precisa-se programar as suas gramáticas

diretamente dentro do código-fonte do formatador.

Um formatador pode ter pré-configurado algumas gramáticas de algumas linguagens de programação, como acontece com o formatador Uncrustify (Seção 3.3: Trabalhos Relacionados). O que Uncrustify consegue fazer é formatar um conjunto de linguagens onde sua gramática é parecida na maior parte dos aspectos. Caso fosse necessário expandir o formatador Uncrustify a suportar linguagens pouco relacionadas com as linguagens já existentes, o código-fonte da implementação atual do formatador seria muito mais complexa.

No Código 23, pode ser encontrado o construtor do formatador de código-fonte. Comparando-o com o construtor do Analisador Semântico (Código 22), pode-se notar algumas diferenças. Em ambos os casos, o processo todo se completará ao percorrer toda a árvore. No caso do Analisador Semântico, a Árvore Sintática, e no caso do Formatador de Código, a Árvore Sintática Abstrata.

Código 23 – Construtor do Formatador (Apêndice G)

```
1 class Backend(pushdown.Interpreter):
2
3     def __init__(self, formatter, tree, program, settings):
4         super().__init__()
5         self.tree = tree
6         self.program = formatter( program, settings )
7
8         ## A list of lists, where each list saves all the matches performed by
9         ## the last match_statement on scope_name_statement
10        self.last_match_stack = []
11
12        ## This is set to False every push statement, and set to True, after
13        ## every match statement. This way we can know whether there is a match
14        ## statement after a push statement.
15        self.is_there_push_after_match = False
16        self.is_there_scope_after_match = False
17
18        self.cached_includes = {}
19        self.cache_includes( tree )
20
21        self.visit( tree )
22        log( 4, "Tree: \n%s", tree.pretty( debug=0 ) )
23
24    def target_language_name_statement(self, tree):
25        target_language_name = tree.children[0]
```

```

27     self.target_language_name = target_language_name
28     log( 4, "target_language_name: %s", target_language_name )
29
30     def master_scope_name_statement(self, tree):
31         master_scope_name = tree.children[0]
32
33         self.master_scope_name = master_scope_name
34         log( 4, "master_scope_name: %s", master_scope_name )
35
36     def include_statement(self, tree):
37         include_statement = str( tree.children[0] )
38
39         log( 4, "include_statement: %s", include_statement )
40         self.visit_children( self.cached_includes[include_statement] )

```

Diferentemente do Analisador Semântico, o Formatador de Código faz herança do tipo “Interpreter” em vez de “Transformer” (Figura 18). A diferença é simples, “Interpreter” visita a árvore partindo das folhas até chegar na raiz visitando todos os nós filhos (Seção 2.1: Compiladores e Tradutores), já “Transformer” visita a árvore partindo da raiz até chegar nos nós pais, i.e., ele não visita os nós-filhos automaticamente como “Transformer” faz. Foi escolhido utilizar “Interpreter” ao contrário de “Transformer” para fazer a interpretação da gramática, porque ele segue naturalmente o algoritmo de consumo do programa que gramáticas utilizadas para adição de cores já seguem (Seção 3.5.1: Gramáticas). Enquanto o tipo “Transformer” segue melhor o fluxo para transformação e criação de uma nova árvore, neste caso, a Árvore Sintática Abstrata.

“Interpreter” diferente de “Transformer”, recebe diretamente no construtor qual será a árvore que ele irá navegar sobre. Nos demais aspectos, “Interpreter” funciona igual ao “Transformer”, exceto no ponto onde “Transformer” cria uma nova árvore no final do processo, enquanto “Interpreter” não cria árvore alguma. O parâmetro chamado “program”, que o construtor de “Transformer” recebe, é o programa a ser formatado pelo formatador. No final do processo, “Interpreter” terá em sua variável “self.program” o novo programa completamente formatado.

Enquanto “Interpreter” é responsável por somente “passar” pela Árvore Sintática Abstrata, a classe “AbstractFormatter” (Código 24) é responsável por realmente fazer a formatação de código-fonte de acordo com as instruções vindas da Árvore Sintática Abstrata. No final do processo, “AbstractFormatter” terá na variável “new_program” todos os pedaços do programa formatado. Uma vez que “Interpreter” termina de construir todos os pedaços de código-fonte formatado, a função “get_new_program” irá unir os pedaços e salvá-los na variável “cached_new_program”, para evitar ter que recalcular o novo programa toda vez que for solicitado a sua nova versão.

Código 24 – Construtor de “AbstractFormatter” (Apêndice G)

```

1 class AbstractFormatter(object):
2     """
3         Represents a program as chunks of data as (text_chunk_start_position,
4             text_chunk).
5     """
6
7     def __init__(self, program, settings):
8         super().__init__()
9         self.initial_size = len( program )
10        self.program = program
11        self.settings = OrderedDict( sorted( settings.items(), key=lambda item:
12            len( str( item ) ) ) )
13
14        self.new_program = []
15        self.cached_new_program = []
log( 4, "program %s: `%s`", len( str( self.program ) ), self.program )

```

A linha “sorted” realiza a ordenação das configurações sem nenhuma necessidade prática neste caso. Ela garante que a ordem no qual as configurações irão ser processadas seja sempre a mesma. Entretanto, no caso da Adição de Cores, a linha “sorted” é uma função obrigatória para garantir que os nomes das cores das configurações sejam atribuídas de acordo com a ordem correta do tema (seleção de cores (DIMA, 2017; SKINNER, 2015b)).

A linha “len(program)” não possui nenhuma influência nos resultados do programa, seja para formatação ou adição de cores. Entretanto, ela é constantemente verificada durante as operações que realizam o consumo do programa de entrada pelas regras da metagramática para garantir que o tamanho do programa original não seja alterado. É necessário que os novos trechos de programa que são formatados saibam qual era a posição deles no programa de original (antes da formatação iniciar), para que as partes do programa original (que não foram formatadas) possam ser adicionadas no novo programa formatado no final do processo de formatação¹⁴.

Durante o processo de consumo, o programa de entrada é modificado e os caracteres que foram consumidos pela gramática são substituídos por algum caractere qualquer¹⁵ e fixo como “§” (símbolo de seção) para impedir que eles sejam reconsu-

¹⁴ Assim que os algoritmos de adição de cores ou formatação estiverem propriamente testados, não seria mais necessário manter o uso da variável “len(program)”.

¹⁵ Não existe a necessidade de utilizar algum caractere em específico, mas é recomendável que este caractere já não exista no texto de entrada para evitar ambiguidades na gramática (Seção 4.3: Especificação da Metalinguagem) fornecida pelo usuário da ferramenta.

midos. A utilização do caractere “\\$” (*símbolo de seção*) foi realizada para simplificar a implementação do algoritmo de consumo. Com o caractere “\\$” (*símbolo de seção*) colocado no lugar dos caracteres que já foram consumidos, é possível reconstruir o programa de entrada no final da análise simplesmente ordenando todos os pedaços do programa formatado que estão armazenada na variável “new_program”. Como o tamanho do programa original não foi modificado, também é possível facilmente integrar no programa formatado, todas as partes do programa original que não foram formatados, no caso do formatador de código-fonte, ou não coloridas do caso da adição de cores.

Neste capítulo, foi visto um pouco sobre detalhes da implementação de meta-programas, metagramáticas e metacompiladores para uso como um formatador de código-fonte. No próximo capítulo será visto um pouco sobre o que se pode concluir sobre este trabalho e esperar dos seus trabalhos futuros.

5 CONCLUSÃO

Neste trabalho foi proposta uma implementação de um algoritmo que trabalha diretamente com a Árvore Sintática da gramática do programa de entrada (Capítulo 4: Formatador Desenvolvido). Apesar de simples, esta implementação é o primeiro passo para criação de novos formatadores de código-fonte. Servindo de base para criação de novos algoritmos voltados a utilizar Árvore Sintática da gramática da linguagem, ao contrário da Árvore Sintática do programa de entrada.

Ferramentas de formatação de código-fonte usualmente não permitem que usuários finais tenham o controle total das alterações no código-fonte. Em um primeiro momento, utilizando a proposta desenvolvida neste trabalho, é possível escrever regras de formatação que quebrem a sintaxe e semântica da linguagem que está sendo formatada. Por exemplo, na linguagem “Go” (BRIMZHANOVA *et al.*, 2019), diferentemente de todas as demais linguagens, é um erro de sintaxe adicionar a chave “{” de abertura de bloco em uma linha nova. Por isso, um formatador de código-fonte para linguagem “Golang”, não deveria permitir a configuração de colocar a chave de abertura de blocos “{” em uma nova linha.

Utilizando-se as ferramentas usuais de formatação, fica impedido que configurações do usuário cause erros sintáticos ou semânticos código-fonte da linguagem que está sendo formatada, a não ser em casos de *bugs* na ferramenta (BENDIK; BENES; CERNA, 2017; STEINERT *et al.*, 2009). As ferramentas de formatação de código-fonte em geral, tentam reconstruir a Árvore Sintática do programa de entrada a ser formatado (Seção 3.3: Trabalhos Relacionados). Neste trabalho, é realizada a construção da Árvore Sintática da gramática do programa ser formatado, e não a Árvore Sintática do programa que está sendo formatado. Com essa mudança, cresce a necessidade da criação de toda uma nova gama de algoritmos de formatação, que possam trabalhar com a Árvore Sintática da gramática dos programas que estão sendo formatados, ao contrário de trabalhar diretamente com a Árvore Sintática do programa que está sendo formatado.

A primeira vantagem de se trabalhar com a Árvore Sintática da gramática do programa que está sendo formatado, ao contrário da Árvore Sintática do programa que está sendo formatado, é a liberdade do usuário final poder escolher especificar exatamente quais são as partes da sintaxe do programa a ser formatado (como já é feito em editores de texto para adição de cores (Seção 3.5: Adição de Cores)). Com isso, é possível especificar exatamente quais estruturas do programa devem ser formatados (e quais não), além de permitir que facilmente sejam adicionados suporte a novas linguagens de programação recém criadas, reutilizando estruturas de sintaxe comuns a linguagens já existentes.

Em um modelo de trabalho onde se faz a formatação do código-fonte diretamente

com a Árvore Sintática do programa de entrada, é possível realizar verificações da semântica do programa que está sendo formatado com maior facilmente, pelo acesso direto a Árvore Sintática do programa de entrada, base para Análise Semântica (Seção 2.3.4: Análise Semântica). Por isso, os algoritmos de formatação de código-fonte desenvolvidos para este trabalho também irão precisar considerar os aspectos semânticos, como a restrição do uso da chave de abertura de blocos da linguagem “Golang”.

Estes formatadores precisarão estar cientes de qual linguagem está sendo formatada e preversar a sua semântica. Entretanto, estes algoritmos terão uma dificuldade adicional. Enquanto os algoritmos de Análise Semântica trabalham com base na Árvore Sintática do programa de entrada, os algoritmos de Análise Semântica deste trabalho precisam trabalhar com base na Árvore Sintática da gramática do programa de entrada. O que levará a necessidade da criação de novos algoritmos de Análise Semântica capazes de trabalhar com a Árvore Sintática da gramática do programa de entrada.

5.1 COMPARAÇÃO COM OUTROS TRABALHOS

Dos três tipos trabalhos relacionados apresentados no Capítulo 3: Estado da Arte, este trabalho é mais similar a dois deles. Os formatadores de código-fonte configurado por arquivos de texto e as ferramentas de adição de cores. Pela lógica de funcionamento deste trabalho, não seria muito interessante que existisse uma interface gráfica como a da ferramenta de Schweitzer (2006), porque o objetivo deste trabalho é que resolva os problemas apresentados pelas ferramentas de formatação de código-fonte hoje existentes. Enquanto que a criação de uma ferramenta de formatação com uma interface gráfica como a de Schweitzer (2006) apresenta problemas (Seção 3.3: Trabalhos Relacionados).

A relação deste trabalho com os formatadores configurados por arquivos de texto, é que ambos são inteiramente configurados por arquivos de texto. Dependendo da complexidade das gramáticas escritas, ambos os trabalhos podem apresentar o problema de compreensão, i.e., entender como determina configuração irá formatar o código-fonte. Ambas as ferramentas possuem arquivos de configuração que são simples de alterar. No exemplo apresentado (Seção 4.1: Visão Geral), o resultado da formatação é inteiramente controlado pelo valor de um número inteiro.

A atual estrutura de composição dos formatadores de código-fonte não suporta que diversos formatadores realizem a formatação simultaneamente. Somente um formatador, que estende da classe “AbstractFormatter” pode estar em funcionamento (Seção 4.4: Analisador Semântico). Entretanto, este formatador recebe como parâmetro de sua função “format_text”, o trecho de código-fonte a ser formatado e o escopo dele. Com essas informações ele poderia em tese, chamar diferentes formatadores

mais especializados de acordo com o seu parâmetro de escopo.

Assim em evoluções deste trabalho, diferente dos outros formatadores de código-fonte (Seção 3.3: Trabalhos Relacionados), espera-se que existam diversos formatadores de código-fonte (ou algoritmos de formatação), capazes de lidar com os diferentes aspectos das linguagens de programação que se deseja formatar seu código-fonte. A simplicidade de configurações (como somente um número inteiro) poderia ser alterada e propor-se configurações que sejam mais elaboradas (complexas). Permitindo que o usuário tenha maior controle sobre o processo de formatação de código-fonte, em conjunto com as gramáticas escritas para esta ferramenta.

Após estas breves comparações, na próxima seção sugere-se algumas ideias e melhorias sobre o trabalho proposto.

5.2 TRABALHOS FUTUROS

Caso exista, o melhor trabalho futuro que pode ser feito é uma reinvención do algoritmo de formação proposto (Seção 4.1: Visão Geral), que permita que as gramáticas fornecidas pelos usuários possam ser melhor utilizadas nos algoritmos de formatação de código-fonte. Ou ainda, uma nova proposta de metagramática que por algum motivo seja mais robusta e simples de utilizar. Enquanto nenhum nova proposta de algoritmo surge, sugere-se algumas melhorias sobre a implementação proposta neste trabalho.

1. Reduzir o uso de memória e otimizar o desempenho em tempo de execução, uma vez que os algoritmos e estratégias adotas não levaram estes pontos em consideração;
2. Corrigir erros de interpretação da metalinguagem ou da especificação da metagramática quando alguns operadores como “scope” que tem um uso opcional, são omitidos (Seção 4.4: Analisador Semântico);
3. Implementar operadores como “captures” e “set” para tornar o uso da metalinguagem mais fácil ou melhorar o seu desempenho em casos de uso específicos (SKINNER, 2015c);
4. Adicionar suporte à especificação de múltiplos escopos a um mesmo trecho de código (DIMA, 2017), definindo alguma estrutura de dados adequada, capaz de permitir consultas e aritméticas de escopos (AESCHLIMANN, 2017) com desempenho constante $\Theta(1)$. Permitindo que estas operações possam ser utilizadas nos arquivos de configuração, melhorando o controle do usuário sobre a formatação de código-fonte em conjunto com a gramática especificada;
5. Melhorar a legibilidade e facilitar a escrita das gramáticas, removendo a necessidade de chaves de abertura “{” e fechamento “}” de blocos (alterando a metagramática (Seção 4.3: Especificação da Metalinguagem). Fazendo com que a

- separação de blocos ser feita de acordo com a indentação como em linguagens como Python e YAML;
6. Realizar a criação de uma nova estrutura de dados para representar o programa de entrada e avaliar se esta nova estrutura de dados é capaz de substituir utilização do caractere “§” (*símbolo de seção*) em conjunto com a classe padrão de *string* da linguagem Python (aumentando a eficiência dos algoritmos de análise do programa de entrada a ser formatado (Seção 4.5: Formatador de Código)).
 7. Implementar uma interface de linha de comando (CLI, Seção 4.1: Visão Geral) que: 1) automaticamente faça o carregamento de todas as gramáticas definidas em um diretório especificado; 2) automaticamente faça o carregamento de todas as configurações definidas em outro diretório e associe estas configurações com as gramáticas correspondentes; 3) automaticamente associe os conjuntos de gramáticas e configurações com um programa de entrada passado pela linha de comando. A separação dos diretórios de gramáticas e configurações é necessário para que se possa aplicar diferentes coleções de configurações de acordo com a vontade do usuário.
 8. Implementar o carregamento automático de diferentes formatadores ou algoritmos de formatação como Código 14, permitindo que eles trabalhem em conjunto. Combinando diferentes formatadores e suas configurações com as gramáticas implementadas pelos usuários.
 9. Implementar plugins de integração com editores de texto mais utilizados como Skinner (2015c) e Dima (2017). Esta integração seria equivalente a implementação de uma linha de comando (CLI), mas com a diferença de que as funcionalidades da CLI estariam acessíveis de dentro dos editores de texto.
 10. Implementar algoritmos que permitam a formatação dinâmica de código-fonte, i.e., enquanto o usuário está escrevendo o código-fonte em um editor de texto, o texto é automaticamente formatado na medida que ele é escrito.
 11. Implementar algoritmos que permitam editores de texto editar o código-fonte de acordo com as configurações de formatação do usuário, enquanto estes mesmos arquivos são salvos no sistema de arquivos utilizando outras configurações, diferentes das configurações de visualização do código-fonte. Tal característica seria útil para permitir que times de desenvolvimento possam trabalhar com diferentes configurações de formatação, e ao mesmo tempo manter o histórico do código-fonte em um padrão fixo de formatação (Capítulo 1: Introdução).
 12. Um pouco fora do escopo deste trabalho, mas ainda relacionado com os últimos itens seria permitir a tradução dinâmica de código-fonte. Nela, o usuário seria

capaz de editar um arquivo de uma linguagem como XML, visualizado ele com um arquivo YAML. Ou seja, ao abrir um arquivo XML, ele é dinamicamente convertido para XML ao ser visualizado no editor de texto. Por fim, ao salvar o arquivo, ele é escrito novamente no sistema de arquivos com o seu formato original, XML. Tal característica seria útil porque algumas linguagens como XML podem ser difíceis de se entender, enquanto ao trabalhar com um formato equivalente como YAML pode ser de mais fácil entendimento (Seção 3.5 e Capítulo 1: Adição de Cores e Introdução).

Com certeza estas não são todas as melhorias ou trabalhos futuros que podem ser feitos. Mas espera-se que elas já sejam o suficiente para despertar a curiosidade e motivar a criação de novos trabalhos sobre formatadores de código-fonte.

REFERÊNCIAS

- AESCHLIMANN, Martin. **TextMate scope selectors: scope exclusion is not implemented.** 2017. Disponível em: <<https://github.com/Microsoft/vscode-textmate/issues/52>>. Acesso em: 21 jul. 2019. Citado nas pp. 42, 73.
- AHO, Alfred V.; LAM, Monica S. *et al.* **Compilers: Principles, Techniques, and Tools (2Nd Edition).** Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811. Citado nas pp. 16, 17, 27–32, 62.
- AHO, Alfred V.; ULLMAN, Jeffrey D. **The Theory of Parsing, Translation, and Compiling.** Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1972. ISBN 0-13-914556-7. Citado nas pp. 23, 25.
- AI HUA WU; PAQUET, J. The translator generation in the general intensional programming compilier. *In:* 8TH International Conference on Computer Supported Cooperative Work in Design. [S.I.: s.n.], maio 2004. 668–672 vol.2. DOI: 10.1109/CACWD.2004.1349274. Citado na p. 16.
- ALLAMANIS, Miltiadis; BARR, Earl T.; SUTTON, Charles. Learning Natural Coding Conventions. **Proceeding FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**, ACM, Hong Kong, China, v. 23, p. 281–293, nov. 2014. DOI: 10.1145/2635868.2635883. Disponível em: <https://www.researchgate.net/publication/260250447_Learning_Natural_Coding_Conventions>. Acesso em: 27 out. 2017. Citado na p. 12.
- ARNDT, Natanael; RADTKE, Norman. Quit Diff: Calculating the Delta Between RDF Datasets Under Version Control. *In:* PROCEEDINGS of the 12th International Conference on Semantic Systems. Leipzig, Germany: ACM, set. 2016. (SEMANTiCS 2016), p. 185–188. DOI: 10.1145/2993318.2993349. Disponível em: <https://www.researchgate.net/publication/309430151_Quit_Diff_Calculating_the_Delta_Between_RDF_Datasets_Under_Version_Control>. Acesso em: 3 nov. 2017. Citado na p. 13.
- ATKINS, D. L. *et al.* Using version control data to evaluate the impact of software tools: a case study of the Version Editor. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Oregon Univ., Eugene, OR, USA, v. 28, n. 7, p. 625–637, jul. 2002. ISSN 0098-5589. DOI: 10.1109/TSE.2002.1019478. Disponível em: <<http://ieeexplore.ieee.org/document/1019478/>>. Acesso em: 3 nov. 2017. Citado na p. 13.

ATWOOD, Jeff. **Death to the Space Infidels!** 2007. Disponível em: <<http://www.codinghorror.com/blog/2009/04/death-to-the-space-infidels.html>>. Acesso em: 1 mar. 2017. Citado nas pp. 37, 38.

BAGGE, O. S. *et al.* Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In: PROCEEDINGS Third IEEE International Workshop on Source Code Analysis and Manipulation. [S.l.: s.n.], set. 2003. p. 65–74. DOI: 10.1109/SCAM.2003.1238032. Citado na p. 12.

BAXTER, Ira. **Why can't C++ be parsed with a LR(1) parser?** 2009. Disponível em: <<https://stackoverflow.com/questions/243383/why-cant-c-be-parsed-with-a-lr1-parser>>. Acesso em: 6 jun. 2019. Citado na p. 31.

BEATTY, John C. On the Relationship Between LL(1) and LR(1) Grammars. **J. ACM**, ACM, New York, NY, USA, v. 29, n. 4, p. 1007–1022, out. 1982. ISSN 0004-5411. DOI: 10.1145/322344.322350. Disponível em: <<http://doi.acm.org/10.1145/322344.322350>>. Citado na p. 28.

BENDIK, Jaroslav; BENES, Nikola; CERNA, Ivana. Finding Regressions in Projects under Version Control Systems. **Computing Research Repository**, Cornell University Library, Masaryk University, Brno, Czech Republic, abs/1708.06623, out. 2017. arXiv: 1708.06623. Disponível em: <<http://arxiv.org/abs/1708.06623>>. Acesso em: 3 nov. 2017. Citado nas pp. 13, 46, 71.

BERG, André; GARDNER, Ben; MAUREL, Guy. **Uncrustify**. 2005. Disponível em: <<http://uncrustify.sourceforge.net/>>. Acesso em: 26 jul. 2019. Citado na p. 38.

BERG, André; GARDNER, Ben; MAUREL, Guy. **Uncrustify – A source code beautifier for C, C++, C#, ObjectiveC, D, Java, Pawn and VALA**. 2005. Disponível em: <<https://github.com/uncrustify/uncrustify>>. Acesso em: 9 jul. 2019. Citado nas pp. 35, 38, 39.

BIAZZINI, Marco; BAUDRY, Benoit. "May the Fork Be with You": Novel Metrics to Analyze Collaboration on GitHub. In: PROCEEDINGS of the 5th International Workshop on Emerging Trends in Software Metrics. Hyderabad, India: ACM, 2014. (WETSoM 2014), p. 37–43. DOI: 10.1145/2593868.2593875. Disponível em: <<http://doi.acm.org/10.1145/2593868.2593875>>. Citado na p. 46.

BOOK, Erwin; SHORRE, Dewey Val; SHERMAN, Steven J. The CWIC/36O System, a Compiler for Writing and Implementing Compilers. **SIGPLAN Not.**, ACM, New York, NY,

USA, v. 5, n. 6, p. 11–29, jun. 1970. ISSN 0362-1340. DOI: 10.1145/954344.954345. Disponível em: <<http://doi.acm.org/10.1145/954344.954345>>. Citado nas pp. 15, 53.

BRIMZHANOVA, S. S. *et al.* Cross-platform Compilation of Programming Language Golang for Raspberry Pi. In: PROCEEDINGS of the 5th International Conference on Engineering and MIS. Astana, Kazakhstan: ACM, 2019. (ICEMIS '19), 10:1–10:5. DOI: 10.1145/3330431.3330441. Disponível em: <<http://doi.acm.org/10.1145/3330431.3330441>>. Citado na p. 71.

BRINK, Alex ten. **Language theoretic comparison of LL and LR grammars**. 2013. Disponível em: <<https://cs.stackexchange.com/q/43>>. Acesso em: 1 jun. 2019. Citado na p. 28.

CAMERON, R. D. An abstract pretty printer. **IEEE Software**, v. 5, n. 6, p. 61–67, nov. 1988. ISSN 0740-7459. DOI: 10.1109/52.10004. Citado na p. 38.

CECCATO, Mariano *et al.* Towards Experimental Evaluation of Code Obfuscation Techniques. In: PROCEEDINGS of the 4th ACM Workshop on Quality of Protection. Alexandria, Virginia, USA: ACM, 2008. (QoP '08), p. 39–46. DOI: 10.1145/1456362.1456371. Disponível em: <<http://doi.acm.org/10.1145/1456362.1456371>>. Citado na p. 40.

CELIŃSKA, Dorota. Coding Together in a Social Network: Collaboration Among GitHub Users. In: PROCEEDINGS of the 9th International Conference on Social Media and Society. Copenhagen, Denmark: ACM, 2018. (SMSociety '18), p. 31–40. DOI: 10.1145/3217804.3217895. Disponível em: <<http://doi.acm.org/10.1145/3217804.3217895>>. Citado na p. 46.

CERECKE, Carl. Repairing Syntax Errors in LR-based Parsers. In: PROCEEDINGS of the Twenty-fifth Australasian Conference on Computer Science - Volume 4. Melbourne, Victoria, Australia: Australian Computer Society, Inc., 2002. (ACSC '02), p. 17–22. Disponível em: <<http://d1.acm.org/citation.cfm?id=563801.563804>>. Citado na p. 17.

CHOMSKY, N. Three models for the description of language. **IRE Transactions on Information Theory**, v. 2, n. 3, p. 113–124, set. 1956. ISSN 0096-1000. DOI: 10.1109/TIT.1956.1056813. Citado nas pp. 23, 24.

- COAN, Evandro. **Alternate simplified logging support and general utilities functions.** 2016. Disponível em: <<https://github.com/evandrocoan/debugtools>>. Acesso em: 8 dez. 2019. Citado na p. 91.
- COAN, Evandro. **Error recovery: Can I add some error recovery strategy?** 2018. Disponível em: <<https://github.com/lark-parser/lark/issues/227>>. Acesso em: 21 jul. 2019. Citado nas pp. 53, 54, 56, 57.
- COCKE, John. **Programming Languages and Their Compilers: Preliminary Notes.** New York, NY, USA: New York University, 1969. ISBN B0007F4UOA. Citado na p. 25.
- CONTRIBUTORS, Wikipedia. **Obfuscation (software).** 2008. Disponível em: <[https://en.wikipedia.org/w/index.php?title=Obfuscation_\(software\)&oldid=905604987](https://en.wikipedia.org/w/index.php?title=Obfuscation_(software)&oldid=905604987)>. Acesso em: 26 jul. 2019. Citado na p. 40.
- COOK, William R. On Understanding Data Abstraction, Revisited. In: PROCEEDINGS of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications. Orlando, Florida, USA: ACM, 2009. (OOPSLA '09), p. 557–572. DOI: 10.1145/1640089.1640133. Disponível em: <<http://doi.acm.org/10.1145/1640089.1640133>>. Citado na p. 106.
- CORCHUELO, Rafael *et al.* Repairing syntax errors in LR parsers. **ACM Trans. Program. Lang. Syst.**, v. 24, p. 698–710, nov. 2002. DOI: 10.1145/586088.586092. Disponível em: <https://www.researchgate.net/publication/220404285_Repairing_syntax_errors_in_LR_parsers>. Citado na p. 17.
- CORMEN, Thomas H. *et al.* **Introduction to Algorithms, Third Edition.** 3rd. [S.I.]: The MIT Press, 2009. ISBN 0262033844, 9780262033848. Citado na p. 29.
- DAIN, Julia Anne. Error recovery for YACC parsers. University of Warwick. Department of Computer Science, Coventry, UK, Number 73, out. 1985. Disponível em: <<http://wrap.warwick.ac.uk/60772/>>. Citado na p. 17.
- DAVIS, Martin D.; SIGAL, Ron; WEYUKER, Elaine J. **Computability, Complexity, and Languages (2Nd Ed.): Fundamentals of Theoretical Computer Science.** San Diego, CA, USA: Academic Press Professional, Inc., 1994. ISBN 0-12-206382-1. Citado na p. 26.

DE JONGE, M. Pretty-printing for software reengineering. In: INTERNATIONAL Conference on Software Maintenance, 2002. Proceedings. [S.I.: s.n.], out. 2002. p. 550–559. DOI: 10.1109/ICSM.2002.1167816. Citado na p. 35.

DEREMER, Frank; PENNELLO, Thomas. Efficient Computation of LALR(1) Look-Ahead Sets. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 4, n. 4, p. 615–649, out. 1982. ISSN 0164-0925. DOI: 10.1145/69622.357187. Disponível em: <<http://doi.acm.org/10.1145/69622.357187>>. Citado nas pp. 30, 46.

DESIGNS, Semantic. **Thicket™ Family of Source Code Obfuscators**. 2003. Disponível em: <<http://www.semdesigns.com/Products/Obfuscators/index.html>>. Acesso em: 24 jul. 2019. Citado na p. 40.

DIMA, Alexandru. **Optimizations in Syntax Highlighting**. 2017. Disponível em: <<https://code.visualstudio.com/blogs/2017/02/08/syntax-highlighting-optimizations>>. Acesso em: 16 set. 2019. Citado nas pp. 42, 43, 56, 57, 60, 62, 69, 73, 74.

DINESH, T. B.; USKUDARH, S. M. Pretty-printing of visual sentences. In: PROCEEDINGS. 1997 IEEE Symposium on Visual Languages (Cat. No.97TB100180). [S.I.: s.n.], set. 1997. p. 242–243. DOI: 10.1109/VL.1997.626589. Citado na p. 38.

EARLEY, Jay. An Efficient Context-free Parsing Algorithm. **Commun. ACM**, ACM, New York, NY, USA, v. 13, n. 2, p. 94–102, fev. 1970. ISSN 0001-0782. DOI: 10.1145/362007.362035. Disponível em: <<http://doi.acm.org/10.1145/362007.362035>>. Citado nas pp. 34, 46.

ELMAS, Tayfun *et al.* An Annotation Assistant for Interactive Debugging of Programs with Common Synchronization Idioms. ACM, Chicago, Illinois, 10:1–10:11, 2009. DOI: 10.1145/1639622.1639632. Disponível em: <<http://doi.acm.org/10.1145/1639622.1639632>>. Acesso em: 30 out. 2017. Citado nas pp. 49, 91.

FENNER, Curtis. **Are Finite Automata Turing Complete?** Disponível em: <<https://cs.stackexchange.com/q/110998>>. Acesso em: 22 jun. 2019. Citado na p. 33.

GEUKENS, Stijn. **Is imposing the same code format for all developers a good idea?** 2013. Disponível em: <<https://softwareengineering.stackexchange.com/questions/189274/is-imposing-the-same-code-format-for-all-developers-a-good-idea>>. Acesso em: 1 mar. 2017. Citado na p. 13.

GROSCH, Josef. Lark-An LALR(2) parser generator with backtracking. Online, jun. 2005. Disponível em: <<http://www.cocolab.com/products/cocktail/doc.pdf/lark.pdf>>. Acesso em: 6 jun. 2019. Citado na p. 17.

HOLZER, Markus; LANGE, Klaus -Jörn. On the complexities of linear LL(1) and LR(1) grammars. In: ÉSIK, Zoltán (Ed.). **Fundamentals of Computation Theory**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 299–308. Citado na p. 29.

HOPCROFT, John E.; MOTWANI, Rajeev; ULLMAN, Jeffrey D. Introduction to Automata Theory, Languages, and Computation. In: New York, NY, USA: Pearson Education, 2006. ISBN 978-0321455369. DOI: 10.1145/568438.568455. Disponível em: <<http://doi.acm.org/10.1145/568438.568455>>. Citado nas pp. 17, 18, 20, 24, 26, 27, 34, 46.

HUME, Tristan. **Rust library for syntax highlighting using Sublime Text syntax definitions**. 2016. Disponível em: <<https://github.com/trishume/syntect>>. Acesso em: 16 set. 2019. Citado nas pp. 41, 42, 57.

HUNNER, Trey; XU, Hong. **Editor Config**. 2012. Disponível em: <<https://github.com/editorconfig/editorconfig>>. Acesso em: 23 jul. 2019. Citado na p. 38.

HUNT III, Harry B.; SZYMANSKI, Thomas G.; ULLMAN, Jeffrey D. On the Complexity of LR(K) Testing. **Commun. ACM**, ACM, New York, NY, USA, v. 18, n. 12, p. 707–716, dez. 1975. ISSN 0001-0782. DOI: 10.1145/361227.361232. Disponível em: <<http://doi.acm.org/10.1145/361227.361232>>. Citado na p. 30.

JBARA, A.; FEITELSON, D. G. How Programmers Read Regular Code: A Control-led Experiment Using Eye Tracking. In: 2015 IEEE 23rd International Conference on Program Comprehension. Florence, Italy: [s.n.], maio 2015. p. 244–254. DOI: 10.1109/ICPC.2015.35. Disponível em: <https://www.researchgate.net/publication/281579264_How_Programmers_Read-Regular_Code_A_Controlled_Experiment_Using_Eye_Tracking>. Acesso em: 31 out. 2017. Citado na p. 12.

JOHNSON, Maggie; ZELENSKI, Julie. **What about these grammars and the minimal parser to recognize it?** 2009. Disponível em: <<https://stackoverflow.com/questions/6379937>>. Acesso em: 20 jul. 2019. Citado na p. 28.

JOURDAN, Jacques-Henri; POTTIER, François. A Simple, Possibly Correct LR Parser for C11. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 39, n. 4, p. 1–36, set. 2017. DOI: 10.1145/3064848. Disponível em: <<https://hal.archives-ouvertes.fr/hal-01633123>>. Citado na p. 31.

- KESTLER, Hans A. *et al.* Generalized Venn diagrams: a new method of visualizing complex genetic set relations. **Bioinformatics**, v. 21, n. 8, p. 1592–1595, nov. 2004. ISSN 1367-4803. DOI: 10.1093/bioinformatics/bti169. eprint: <http://oup.prod.sis.lan/bioinformatics/article-pdf/21/8/1592/691813/bti169.pdf>. Disponível em: <<https://doi.org/10.1093/bioinformatics/bti169>>. Citado na p. 27.
- KIM, J. *et al.* Improving Refactoring Speed by 10X. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). [S.I.: s.n.], maio 2016. p. 1145–1156. DOI: 10.1145/2884781.2884802. Citado nas pp. 38, 49.
- KING, Zachary; LÓPEZ-ANGLADA, Guillermo. **PackageDev - Tools to ease the creation of snippets, syntax definitions for Sublime Text**. 2011. Disponível em: <<https://github.com/SublimeText/PackageDev>>. Acesso em: 30 nov. 2019. Citado na p. 42.
- KIRPICHOV, Eugene. **Can regular languages be Turing complete?** 2014. Disponível em: <<https://cs.stackexchange.com/questions/33666>>. Acesso em: 13 jun. 2019. Citado nas pp. 33, 54.
- KNUTH, Donald E. On the translation of languages from left to right. **Information and Control**, v. 8, n. 6, p. 607–639, 1965. ISSN 0019-9958. DOI: [https://doi.org/10.1016/S0019-9958\(65\)90426-2](https://doi.org/10.1016/S0019-9958(65)90426-2). Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0019995865904262>>. Citado nas pp. 29, 31.
- LANG, Bernard. Deterministic Techniques for Efficient Non-Deterministic Parsers. In: PROCEEDINGS of the 2Nd Colloquium on Automata, Languages and Programming. Berlin, Heidelberg: Springer-Verlag, 1974. p. 255–269. DOI: 10.1007/3-540-06841-4_65. Disponível em: <<http://dl.acm.org/citation.cfm?id=646230.681872>>. Citado na p. 25.
- LASTER, Brent. Getting Productive. In: PROFESSIONAL Git. Cary, North Carolina, USA: John Wiley & Sons, Inc., nov. 2016. p. 73–97. ISBN 9781119285021. DOI: 10.1002/9781119285021.ch5. Disponível em: <https://www.researchgate.net/publication/311666005_Getting_Productive>. Acesso em: 2 nov. 2017. Citado na p. 12.
- LEO, Joop M.I.M. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. **Theoretical Computer Science**, v. 82, n. 1, p. 165–176, 1991. ISSN 0304-3975. DOI: [https://doi.org/10.1016/0304-3975\(91\)90180-A](https://doi.org/10.1016/0304-3975(91)90180-A). Disponível em: <<http://www.sciencedirect.com/science/article/pii/030439759190180A>>. Citado na p. 29.

MARTENS, Wim *et al.* Expressiveness and Complexity of XML Schema. **ACM Trans. Database Syst.**, ACM, New York, NY, USA, v. 31, n. 3, p. 770–813, set. 2006. ISSN 0362-5915. DOI: 10.1145/1166074.1166076. Disponível em: <<http://doi.acm.org/10.1145/1166074.1166076>>. Citado na p. 42.

MAXDAX, S. **Unable to meet our code style and other issues**. 2019. Disponível em: <<https://github.com/uncrustify/uncrustify/issues/2338>>. Acesso em: 10 jul. 2019. Citado na p. 36.

MIARA, Richard J. *et al.* Program indentation and comprehensibility. **Communications of the ACM**, ACM, Online, v. 26, p. 861–867, nov. 1983. DOI: 10.1145/182.358437. Disponível em: <https://www.researchgate.net/publication/234809222_Program_indentation_and_comprehensibility>. Acesso em: 7 set. 2017. Citado na p. 37.

MICHAELSON, Greg. Are There Domain Specific Languages? In: PROCEEDINGS of the 1st International Workshop on Real World Domain Specific Languages. Barcelona, Spain: ACM, 2016. (RWDSL '16), 1:1–1:3. DOI: 10.1145/2889420.2892271. Disponível em: <<http://doi.acm.org/10.1145/2889420.2892271>>. Citado nas pp. 33, 54.

MURPHREE, E. L.; FENVES, S. J. A technique for generating interpretive translators for problem-oriented languages. **BIT Numerical Mathematics**, v. 10, n. 3, p. 310–323, set. 1970. ISSN 1572-9125. DOI: 10.1007/BF01934200. Disponível em: <<https://doi.org/10.1007/BF01934200>>. Citado na p. 16.

ODGAARD, Allan. **TextMate - A graphical text editor for macOS**. 2004. Disponível em: <<https://github.com/textmate/textmate>>. Acesso em: 30 nov. 2019. Citado nas pp. 42, 43.

OLEVSKY, Ilya. **Why Version Control Is Critical To Your Success**. 2013. Disponível em: <<https://web.archive.org/web/20180423062356/http://www.codeservedcold.com/version-control-importance/>>. Acesso em: 4 jul. 2019. Citado na p. 12.

ORTIZ, Ariel. Web Development with Python and Django (Abstract Only). In: PROCEEDINGS of the 43rd ACM Technical Symposium on Computer Science Education. Raleigh, North Carolina, USA: ACM, 2012. (SIGCSE '12), p. 686–686. DOI: 10.1145/2157136.2157353. Disponível em: <<http://doi.acm.org/10.1145/2157136.2157353>>. Citado nas pp. 46, 91.

PARIKH, Rohit J. On Context-Free Languages. **J. ACM**, ACM, New York, NY, USA, v. 13, n. 4, p. 570–581, out. 1966. ISSN 0004-5411. DOI: 10.1145/321356.321364. Disponível em: <<http://doi.acm.org/10.1145/321356.321364>>. Citado na p. 27.

PARR, Terence. **The Definitive ANTLR 4 Reference**. 2nd. [S.l.]: Pragmatic Bookshelf, 2013. ISBN 1934356999, 9781934356999. Citado nas pp. 19, 27, 53, 58, 90.

PARR, Terence; FISHER, Kathleen. LL(*): The Foundation of the ANTLR Parser Generator. In: PROCEEDINGS of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. San Jose, California, USA: ACM, 2011. (PLDI '11), p. 425–436. DOI: 10.1145/1993498.1993548. Disponível em: <<http://doi.acm.org/10.1145/1993498.1993548>>. Citado na p. 27.

PARR, Terence; HARWELL, Sam; FISHER, Kathleen. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In: PROCEEDINGS of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. Portland, Oregon, USA: ACM, 2014. (OOPSLA '14), p. 579–598. DOI: 10.1145/2660193.2660202. Disponível em: <<http://doi.acm.org/10.1145/2660193.2660202>>. Citado na p. 27.

PATTIS, Richard E. Teaching EBNF First in CS 1. **SIGCSE Bull.**, ACM, New York, NY, USA, v. 26, n. 1, p. 300–303, mar. 1994. ISSN 0097-8418. DOI: 10.1145/191033.191155. Disponível em: <<http://doi.acm.org/10.1145/191033.191155>>. Citado nas pp. 19, 53.

PEZOA, Felipe *et al.* Foundations of JSON Schema. In: PROCEEDINGS of the 25th International Conference on World Wide Web. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016. (WWW '16), p. 263–273. DOI: 10.1145/2872427.2883029. Disponível em: <<https://doi.org/10.1145/2872427.2883029>>. Citado na p. 57.

REN, Luyao; ZHOU, Shurui; KÄSTNER, Christian. Forks Insight: Providing an Overview of GitHub Forks. In: PROCEEDINGS of the 40th International Conference on Software Engineering: Companion Proceedings. Gothenburg, Sweden: ACM, 2018. (ICSE '18), p. 179–180. DOI: 10.1145/3183440.3195085. Disponível em: <<http://doi.acm.org/10.1145/3183440.3195085>>. Citado na p. 46.

RICI, Stack. **Is there a LL(K) Grammar which is not LALR(K) Grammar?** 2019. Disponível em: <<https://cs.stackexchange.com/q/110775>>. Acesso em: 1 jun. 2019. Citado na p. 28.

RONACHER, Armin. **Flask - The Python micro framework for building web applications**. Disponível em: <<https://github.com/pallets/flask>>. Acesso em: 29 nov. 2019. Citado na p. 91.

ROSSO, Santiago Perez De; JACKSON, Daniel. Purposes, Concepts, Misfits, and a Redesign of Git. **SIGPLAN Not.**, ACM, Massachusetts Institute of Technology, USA, v. 51, n. 10, p. 292–310, out. 2016. ISSN 0362-1340. DOI: 10.1145/3022671.2984018. Disponível em: <https://www.researchgate.net/publication/311477527_Purposes_Concepts_misfits_and_a_redesign_of_git>. Acesso em: 30 out. 2017. Citado na p. 13.

ROSSUM, Guido van; WARSAW, Barry; COGHLAN, Nick. **Should a Line Break Be-fore or After a Binary Operator?** 2001. Disponível em: <<https://www.python.org/dev/peps/pep-0008/#should-a-line-break-before-or-after-a-binary-operator>>. Acesso em: 8 jul. 2019. Citado na p. 37.

ROYER, Tiago. **MÁQUINAS DE TURING NÃO-DETERMINÍSTICAS COMO COMPUTADORES DE FUNÇÕES**. 2015. Florianópolis, Santa Catarina, Brazil. [Departamento de Informática e Estatística]. Disponível em: <<https://github.com/royertiago/tcc>>. Acesso em: 20 jun. 2019. Citado na p. 24.

SCALABRINO, S. *et al.* Improving code readability models with textual features. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). Austin, TX, USA: IEEE Computer Society, maio 2016. p. 1–10. DOI: 10.1109/ICPC.2016.7503707. Disponível em: <https://www.researchgate.net/publication/301685380_Improving_Code_Readability_Models_with_Textual_Features>. Acesso em: 1 nov. 2017. Citado na p. 37.

SCHILD, Uri J.; HERZOG, Shai. The Use of Meta-rules in Rule Based Legal Computer Systems. In: PROCEEDINGS of the 4th International Conference on Artificial Intelligence and Law. Amsterdam, The Netherlands: ACM, 1993. (ICAIL '93), p. 100–109. DOI: 10.1145/158976.158989. Disponível em: <<http://doi.acm.org/10.1145/158976.158989>>. Citado nas pp. 15, 46.

SCHWEITZER, Thomas. **Powerful code indenter front-end, UniversalIndentGUI**. <http://universalindent.sourceforge.net/index.php>. 2006. Disponível em: <<https://github.com/danblakemore/universal-indent-gui>>. Acesso em: 1 mar. 2017. Citado nas pp. 38–40, 72.

SCHWEITZER, Thomas. **UniversalIndentGUI**. <http://universalindent.sourceforge.net/screenshots.php>. Jul. 2007. Disponível em: <<http://universalindent.sourceforge.net/images/screenshot4.png>>. Acesso em: 31 jul. 2019. Citado na p. 39.

SHINAN, Erez. **Commit history for Lark on GitHub**. 2019. Disponível em: <<https://github.com/lark-parser/lark/commits/master>>. Acesso em: 1 dez. 2019. Citado na p. 46.

SHINAN, Erez. **Design a new Lark cheatsheet**. 2018. Disponível em: <<https://github.com/lark-parser/lark/issues/195>>. Acesso em: 30 set. 2019. Citado nas pp. 19, 53.

SHINAN, Erez. **Grammar Reference**. 2019. Disponível em: <<https://lark-parser.readthedocs.io/en/latest/grammar/>>. Acesso em: 30 set. 2019. Citado nas pp. 19, 53.

SHINAN, Erez. **Lark implements the following parsing algorithms: Earley, LALR(1), and CYK**. 2014. Disponível em: <<https://lark-parser.readthedocs.io/en/latest/parsers/>>. Acesso em: 30 jun. 2019. Citado nas pp. 15, 34, 46.

SINGER, Adam B. The Command Line Interface. In: PRACTICAL C++ Design. 1. ed. Online: Apress, 2017. p. 97–113. ISBN 978-1-4842-3056-5. DOI: 10.1007/978-1-4842-3057-2_5. Disponível em: <https://www.researchgate.net/publication/320120365_The_Command_Line_Interface>. Acesso em: 10 out. 2017. Citado na p. 47.

SIPPU, Seppo; SOISALON-SOININEN, Eljas. Practical Error Recovery in LR Parsing. In: PROCEEDINGS of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Albuquerque, New Mexico: ACM, 1982. (POPL '82), p. 177–184. DOI: 10.1145/582153.582173. Disponível em: <<http://doi.acm.org/10.1145/582153.582173>>. Citado na p. 17.

SIPSER, Michael. Introduction to the Theory of Computation. In: New York, NY, USA: Pearson Education, 2012. ISBN 978-1133187790. DOI: 10.1145/230514.571645. Disponível em: <<http://doi.acm.org/10.1145/230514.571645>>. Citado nas pp. 24–26, 34.

SKINNER, Jon. **SUBLIME TEXT 3 DOCUMENTATION, Color Schemes**. 2015. Disponível em: <https://www.sublimetext.com/docs/3/color_schemes.html>. Acesso em: 30 nov. 2019. Citado nas pp. 41, 44.

SKINNER, Jon. **SUBLIME TEXT 3 DOCUMENTATION, Scope Naming**. 2015. Disponível em: <http://www.sublimetext.com/docs/3/scope_naming.html>. Acesso em: 17 set. 2019. Citado nas pp. 41, 44, 69.

SKINNER, Jon. **SUBLIME TEXT 3 DOCUMENTATION, Syntax Definitions**. 2015. Disponível em: <<https://www.sublimetext.com/docs/3/syntax.html>>. Acesso em: 1 mar. 2017. Citado nas pp. 41–43, 60, 73, 74.

SMITH, Thomas. **branch temporarily highlights wrong 3042**. 2019. Disponível em: <https://github.com/sublimehq/sublime_text/issues/3042>. Acesso em: 30 nov. 2019. Citado na p. 42.

SORKIN, Arthur; DONOVAN, Peter. LR(1) Parser Generation System: LR(1) Error Recovery, Oracles, and Generic Tokens. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 36, n. 2, p. 1–5, mar. 2011. ISSN 0163-5948. DOI: 10.1145/1943371.1943391. Disponível em: <<http://doi.acm.org/10.1145/1943371.1943391>>. Citado na p. 17.

STEINERT, Bastian *et al.* Debugging into Examples. In: TESTING of Software and Communication Systems: 21st IFIP WG 6.1 International Conference , November 2-4, 2009. Proceedings. Eindhoven, The Netherlands: Springer, 2009. p. 235–240. ISBN 978-3-642-05031-2. DOI: 10.1007/978-3-642-05031-2_18. Disponível em: <https://www.researchgate.net/publication/221047094_Debugging_into_Examples>. Acesso em: 31 out. 2017. Citado nas pp. 13, 46, 71, 91.

STEPHEN, C. **How can I get eclipse to wrap lines after a period instead of before**. 2017. Disponível em: <<https://stackoverflow.com/questions/31438377>>. Acesso em: 26 jun. 2019. Citado na p. 36.

VELAZQUEZ-ITURBIDE, J. A.; PRESA-VAZQUEZ, A. Customization of visualizations in a functional programming environment. In: FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011. [S.I.: s.n.], nov. 1999. 12b3/22–12b3/28 vol.2. DOI: 10.1109/FIE.1999.841580. Citado na p. 35.

VELDHUIZEN, Todd L. Tradeoffs in Metaprogramming. In: PROCEEDINGS of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. Charleston, South Carolina: ACM, 2006. (PEPM '06), p. 150–159. DOI: 10.1145/1111542.1111569. Disponível em: <<http://doi.acm.org/10.1145/1111542.1111569>>. Citado na p. 55.

WICKHAM, Hadley. **The tidyverse style guide**. 2017. Disponível em: <<https://style.tidyverse.org/>>. Acesso em: 9 jul. 2019. Citado na p. 37.

WOODS, William A. Context-sensitive Parsing. **Commun. ACM**, ACM, New York, NY, USA, v. 13, n. 7, p. 437–445, jul. 1970. ISSN 0001-0782. DOI: 10.1145/362686.362695. Disponível em: <<http://doi.acm.org/10.1145/362686.362695>>. Citado na p. 33.

YAMAGUCHI, Daisuke; KURAMITSU, Kimio. CPEG: A Typed Tree Construction from Parsing Expression Grammars with Regex-like Captures. In: PROCEEDINGS of the 34th ACM/SIGAPP Symposium on Applied Computing. Limassol, Cyprus: ACM, 2019. (SAC '19), p. 1526–1533. DOI: 10.1145/3297280.3297433. Disponível em: <<http://doi.acm.org/10.1145/3297280.3297433>>. Citado na p. 61.

YENER, Murat; DUNDAR, Onur. Continuous Integration. In: EXPERT Android Studio. San Jose, California: John Wiley & Sons, Inc., out. 2016. p. 281–307. ISBN 9781119419310. DOI: 10.1002/9781119419310.ch9. Disponível em: <https://www.researchgate.net/publication/316657029_Using_Source_Control>. Acesso em: 3 nov. 2017. Citado na p. 15.

YENER, Murat; DUNDAR, Onur. Using Source Control. In: EXPERT Android Studio. San Jose, California: John Wiley & Sons, Inc., out. 2016. p. 245–279. ISBN 9781119419310. DOI: 10.1002/9781119419310.ch9. Disponível em: <https://www.researchgate.net/publication/316657029_Using_Source_Control>. Acesso em: 3 nov. 2017. Citado na p. 12.

YENIGALLA, Leelakrishna *et al.* How Novices Read Source Code in Introductory Courses on Programming: An Eye-Tracking Experiment. In: **Foundations of Augmented Cognition: Neuroergonomics and Operational Neuroscience: 10th International Conference, Proceedings, Part II**. Edição: Dylan D. Schmorow e Cali M. Fidopiastis. Toronto, ON, Canada: Springer-Verlag, jul. 2016. p. 120–131. ISBN 978-3-319-39952-2. DOI: 10.1007/978-3-319-39952-2_13. Disponível em: <https://www.researchgate.net/publication/304190149_How_Novices_Read_Source_Code_in_Introductory_Courses_on_Programming_An_Eye-Tracking_Experiment>. Acesso em: 2 nov. 2017. Citado na p. 35.

ZERVOUDAKIS, Fokion *et al.* Cascading Verification: An Integrated Method for Domain-specific Model Checking. In: PROCEEDINGS of the 2013 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg, Russia: ACM, 2013. (ESEC/FSE 2013), p. 400–410. DOI: 10.1145/2491411.2491454. Disponível em: <<http://doi.acm.org/10.1145/2491411.2491454>>. Citado nas pp. 33, 41, 42, 54.

ZHAO, Zhijia *et al.* HPar: A Practical Parallel Parser for HTML–taming HTML Complexities for Parallel Parsing. **ACM Trans. Archit. Code Optim.**, ACM, New York, NY, USA, v. 10, n. 4, 44:1–44:25, dez. 2013. ISSN 1544-3566. DOI: 10.1145/2555289.2555301. Disponível em: <<http://doi.acm.org/10.1145/2555289.2555301>>. Citado na p. 90.

APÊNDICE A – MANUAL DO FORMATADOR DE CÓDIGO

Esse apêndice apresenta um manual da ferramenta. Não recomenda-se que usuários que não possuam conhecimentos sobre a semântica das linguagens de programação utilizem esta ferramenta para realizar a formatação de código-fonte. Neste caso, o mais indicado é que sejam utilizados as demais ferramentas de formatação, que hoje são mais específicas para cada linguagem individualmente e possuem inherentemente os conhecimentos específicos da sintaxe e semântica da linguagem a ser formatada.

Não foi criada nenhuma interface gráfica ou de linha de comando que faça a entrada do programa a ser formatado e das configurações do formatador de código-fonte. Existem duas implementações que utilizam a metalinguagem (Código 38). Uma utiliza a metalinguagem para adicionar cores (Código 36), como feito em editores de texto e a outra realiza a formatação de código-fonte (Código 37).

Nos Códigos 26 a 28, encontra-se um exemplo simples de programa que pode ser construído para executar o Formatador de Código e a Adição de Cores. Sua construção é a mesma utilizada para a criação dos testes de unidade (Código 34). Para manter a implementação simples, tanto o Código 26 quanto o Código 27 geram como resultado arquivos HTML (ZHAO *et al.*, 2013), contendo como conteúdo o resultado de seu trabalho, i.e., respectivamente, o código-fonte formatado ou com a adição de cores. Também, gerando páginas HTML para o formatador de código-fonte será possível observar com mais facilidade o código-fonte original e formatado, e as metainformações atribuídas pela gramática da linguagem como atributos das tags HTML.

Nos Códigos 30 e 32, serão encontrados como nós-folhas das árvores, os *tokens* criados pelo analisador léxico. Eles seguem o mesmo padrão de notação dos *tokens* utilizados pelo Analisador ANTLR (PARR, 2013). A implementação da exibição dos *tokens* no formato do analisador ANTLR 4 foi uma das implementações feitas no *fork “pushdown”* do Analisador Lark (Seção 4.2: Introdução à Metagramática).

Um *token* como “[@7,151:166='comment.line.sma'<TEXT CHUNK END_>,7:25]” segue o modelo de representação do ANTLR 4 (PARR, 2013) e contém as seguintes informações: 1) “@7” significa que ele é o sétimo *token*; 2) “151:166” é o início e fim do lexema do *token* no programa de entrada, contado a partir de 0 até o fim do programa; 3) “='comment.line.sma'” é o lexema (Seção 2.1: Compiladores e Tradutores) ou conteúdo do *token* propriamente dito; 4) “<TEXT CHUNK END_>” é o nome do *token* atribuído pela gramática de entrada, por fim; 5) “7:25” é a linha e coluna do *token* no programa de entrada.

Tanto para a escrita do Código 36, quanto para a escrita do Código 37 por consequência, foi utilizado em um primeiro momento a biblioteca “dominate” para realizar a construção da página HTML com a adição de cores ou código-formatado. Entretanto,

devido a *bugs* na biblioteca “dominate”, a página HTML era gerada de forma errada. Por fim, optou-se em escrever o código necessário para gerar a páginas HTML, fazendo o uso da biblioteca “dominate” somente para realizar a conversão de caracteres especiais para seus correspondes em HTML. Por exemplo, o caractere “<” em HTML é escrito como “<”.

Não foi escolhido procurar uma outra biblioteca para fazer a criação de páginas HTML, porque ao pesquisar por bibliotecas que tinham somente esta função, não foi encontrado nenhuma similar. Para realizar a pesquisa por uma nova biblioteca de geração de HTML, foram utilizados os seguintes critérios: 1) ser um biblioteca de código-aberto (*open-source*); 2) ser ativa, i.e., possuir novos recursos (*features*) ou *bugs* sendo reportados e corrigidos no último ano.

Somente foram encontrados bibliotecas com Ronacher (2019) conseguem gerar páginas HTML, mas que também são servidores HTML ou Web, que vão muito além das necessidades deste projeto. Portanto, optou-se por fazer todo o código necessário para geração de páginas HTML. Dispensando a necessidade de escolher outra ferramenta. Não existe a necessidade explícita de utilizar um biblioteca HTML para gerar páginas HTML. Pode-se tranquilamente escrever o código HTML necessário para as necessidades deste projeto, sem o uso de um biblioteca HTML. A vantagem de utilizar uma biblioteca HTML é uma melhor legibilidade do código escrito em Python, que será mais parecido com o código HTML (ORTIZ, 2012).

Para se realizar a execução de qualquer arquivo deste projeto, é necessário ter um interpretador “Python 3.6” instalado, junto com as bibliotecas “pip3”, “debug_tools”, “dominate” e “pushdown”: 1) “pip3” é o gerenciador de pacotes da linguagem Python, responsável fazer a instalação automatizada de pacotes ou bibliotecas da linguagem Python e permite que os outros pacotes de dependência deste projeto seja instalados facilmente. 2) “debug_tools” é uma biblioteca de gerenciamento mensagens de “debug” ou depuração (STEINERT *et al.*, 2009; ELMAS *et al.*, 2009) criada também pelo autor deste projeto (COAN, 2016), “debug_tools” foi utilizada porque qualquer problema (*bug*) ou recurso novo (*feature*) necessário na ferramenta pode ser facilmente corrigido pelo autor. Em uma instalação tradicional “Ubuntu”, os pacotes requeridos por este projeto podem ser instalados com os seguintes comandos:

- 1) sudo apt-get install python3 python3-pip
- 2) pip3 install -r requirements.txt (Código 25)
- 3) python3 main_formatter.py (Código 26)
- 4) python3 main_highlighter.py (Código 27)

```
1 setuptools
2 wheel
3 pushdown
4 debug_tools
5 dominate
```

Código 26 – Arquivo “source/main_formatter.py”

```
1 import os
2 import sys
3
4 import code_formatter
5 import semantic_analyzer
6
7 from pushdown import Lark
8 from debug_tools import getLogger
9 from debug_tools.testing_utilities import wrap_text
10
11 program_name = "main_formatter"
12 log = getLogger(__name__)
13
14 def assert_path(module):
15     if module not in sys.path:
16         sys.path.append( module )
17
18 assert_path( os.path.realpath( __file__ ) )
19 from utilities import make_png
20 from debug_tools.utilities import get_relative_path
21
22 ## The relative path the the pushdown grammar parser file from the current file
23 metagrammar_path = get_relative_path( "grammars_grammar.pushdown", __file__ )
24
25 ## The parser used to build the syntax tree and parse the input text
26 with open( metagrammar_path, "r", encoding='utf-8' ) as file:
27     my_parser = Lark( file.read(), start='language_syntax', parser='lalr',
28                      lexer='contextual')
29 example_grammar = wrap_text(
30 r"""
31     scope: source.sma
32     name: Abstract Machine Language
33     contexts: {
```

```
34     match: if\(\ {
35         scope: if.statement.definition
36         push: {
37             meta_scope: if.statement.body
38             match: \) \{
39                 scope: if.statement.definition
40                 pop: true
41             }
42         }
43     }
44 }
45 """ )
46
47 def generate_image(tree, tree_name):
48     log.clean( "Generating '%s'...", tree_name )
49     make_png( tree, get_relative_path( tree_name, __file__ ), debug=0, dpi=300 )
50
51 syntax_tree = my_parser.parse( example_grammar )
52 log.clean( "Syntax Tree\n%s", syntax_tree.pretty( debug=1 ) )
53
54 abstract_syntax_tree = semantic_analyzer.TreeTransformer().transform( syntax_tree )
55 log.clean( "Abstract Syntax Tree\n%s", abstract_syntax_tree.pretty( debug=1 ) )
56
57 example_program = wrap_text(
58 r"""
59 if(something) bar
60 """
61 )
62
63 example_settings = {
64     "if.statement.body" : 2,
65 }
66
67 backend = code_formatter.Backend( code_formatter.SingleSpaceFormatter,
68     ↳ abstract_syntax_tree, example_program, example_settings )
69 generated_html = backend.generated_html()
70
71
72 html_file_name = "%s.html" % program_name
73 log.clean( "Generating '%s'...", html_file_name )
74
75
76 with open( html_file_name, 'w', newline='\n', encoding='utf-8' ) as output_file:
77     output_file.write( generated_html )
78     output_file.write("\n")
```

```
75  
76 generate_image( syntax_tree, "%s_syntax_tree.png" % program_name )  
77 generate_image( abstract_syntax_tree, "%s_abstract_syntax_tree.png" % program_name  
↪ )
```

Código 27 – Arquivo “source/main_highlighter.py”

```
1 import os  
2 import sys  
3  
4 import code_highlighter  
5 import semantic_analyzer  
6  
7 from pushdown import Lark  
8 from debug_tools import getLogger  
9 from debug_tools.testing_utilities import wrap_text  
10  
11 program_name = "main_highlighter"  
12 log = getLogger(__name__)  
13  
14 def assert_path(module):  
15     if module not in sys.path:  
16         sys.path.append( module )  
17  
18 assert_path( os.path.realpath( __file__ ) )  
19 from utilities import make_png  
20 from debug_tools.utilities import get_relative_path  
21  
22 ## The relative path the the pushdown grammar parser file from the current file  
23 metagrammar_path = get_relative_path( "grammars_grammar.pushdown", __file__ )  
24  
25 ## The parser used to build the syntax tree and parse the input text  
26 with open( metagrammar_path, "r", encoding='utf-8' ) as file:  
27     my_parser = Lark( file.read(), start='language_syntax', parser='lalr',  
↪     lexer='contextual')  
28  
29 example_grammar = wrap_text(  
30 r"""  
31     scope: source.sma  
32     name: Abstract Machine Language  
33     contexts: {  
34         match: // {
```

```
35         scope: comment.start.sma
36     push: {
37         meta_scope: comment.line.sma
38         match: \n|\$ {
39             pop: true
40         }
41     }
42 }
43 }
44 """
45
46 def generate_image(tree, tree_name):
47     log.clean( "Generating '%s'...", tree_name )
48     make_png( tree, get_relative_path( tree_name, __file__ ), debug=0, dpi=300 )
49
50 syntax_tree = my_parser.parse( example_grammar )
51 log.clean( "Syntax Tree\n%s", syntax_tree.pretty( debug=1 ) )
52
53 abstract_syntax_tree = semantic_analyzer.TreeTransformer().transform( syntax_tree )
54 log.clean( "Abstract Syntax Tree\n%s", abstract_syntax_tree.pretty( debug=1 ) )
55
56 example_program = wrap_text(
57 r"""
58 // Example single line commentary
59 """
60
61 example_theme = {
62     "comment" : "#FF0000",
63     "comment.line" : "#00FF00",
64 }
65
66 backend = code_highlighter.Backend( abstract_syntax_tree, example_program,
67     ↳ example_theme )
68 generated_html = backend.generated_html()
69
70 html_file_name = "%s.html" % program_name
71 log.clean( "Generating '%s'...", html_file_name )
72 with open( html_file_name, 'w', newline='\n', encoding='utf-8' ) as output_file:
73     output_file.write( generated_html )
74     output_file.write("\n")
75
```

```
76 generate_image( syntax_tree, "%s_syntax_tree.png" % program_name )
77 generate_image( abstract_syntax_tree, "%s_abstract_syntax_tree.png" % program_name
    ↵ )
```

Código 28 – Arquivo “source/utilities.py”

```
1 import pushdown
2
3 from debug_tools import getLogger
4 log = getLogger( 127, __name__ )
5
6
7 def make_png(pushdown_tree, output_file, debug=False, **kwargs):
8     pushdown.tree.pydot__tree_to_png( pushdown_tree, output_file, "TB",
    ↵     debug=debug, **kwargs )
```

APÊNDICE B – EXECUÇÃO DE “MAIN_FORMATTER.PY”

O programa do Código 26, faz a criação de 5 artefatos de resultado. Duas árvores em forma de figura, duas árvores em forma de texto e uma arquivo HTML. No Código 29, encontra-se o arquivo HTML gerado pelo programa de exemplo (Código 26). Nas Figuras 19 e 20, encontram-se as imagens geradas pelo programa “main_formatter.py” (Código 26). No Código 30, encontra-se a saída da linha de comando que se obtém ao realizar a execução do programa de exemplo (Código 26).

Ambas as Figuras 19 e 20 quanto o Código 30 representam os mesmos dados, mas mostrados de formas diferentes (figura *versus* textual) e com níveis de detalhe diferentes. No Código 30, são mostrado os nós-folhas das árvores com maior nível de detalhe. Já nas Figuras 19 e 20, os nós-folhas são mostrados com simplificações para que a figura possa ser vista em uma única tela.

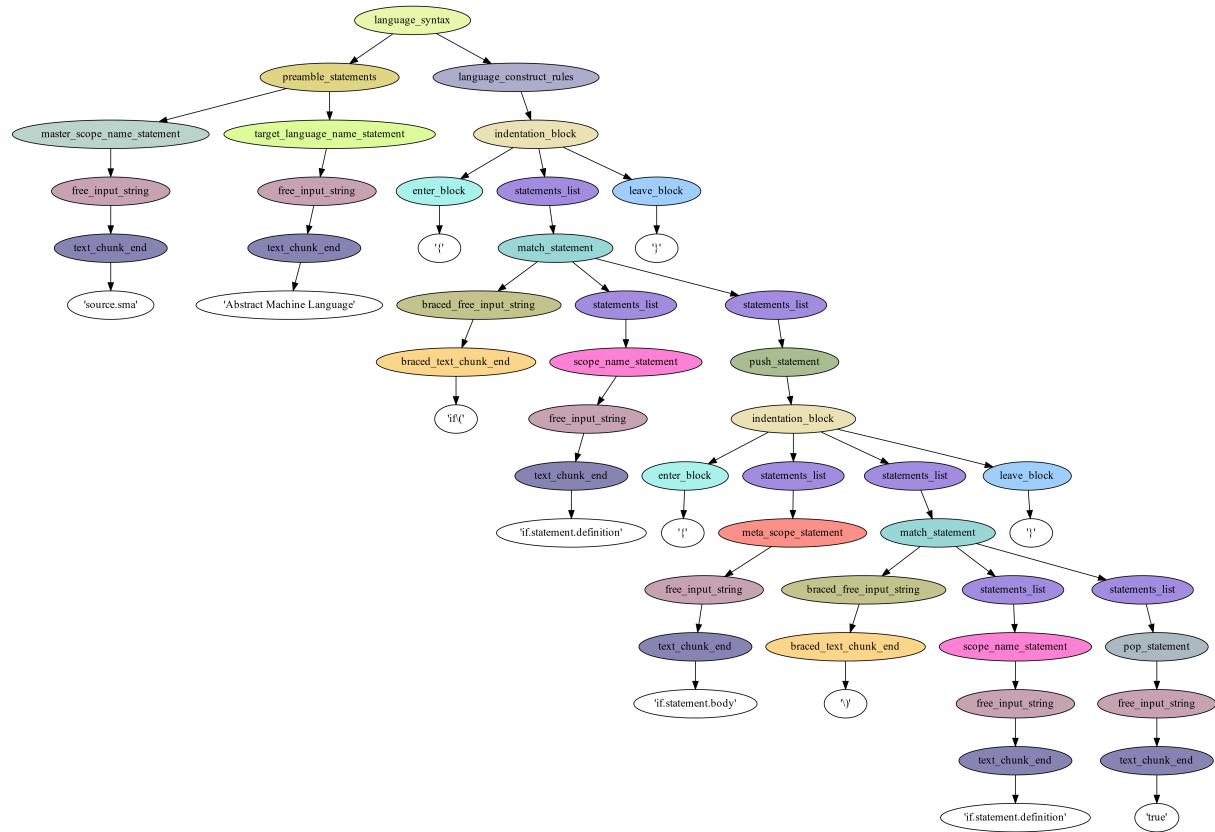
Código 29 – Arquivo HTML gerado pelo programa de exemplo “main_formatter.py”

```
1 <!DOCTYPE html><html><head><title>Abstract Machine Language -  
2   source.sma</title></head>  
3 <body style="white-space: pre; font-family: monospace;"><span setting="unformatted"  
4   grammar_scope="if.statement.definition" setting_scope=""  
5   original_program="if(>if(</span><span setting="2"  
6   grammar_scope="if.statement.body" setting_scope="if.statement.body"  
7   original_program="something"> something </span><span setting="unformatted"  
8   grammar_scope="if.statement.definition" setting_scope=""  
9   original_program=")">)</span><span grammar_scope="none" setting_scope="none">  
10  bar</span></body></html>
```

Código 30 – Resultado da execução do arquivo “source/main_formatter.py”

```
1 $ python3 main_formatter.py  
2 Syntax Tree  
3 language_syntax  
4 preamble_statements  
5   master_scope_name_statement  
6     free_input_string  
7       text_chunk_end  [@1,7:16='source.sma'<TEXT_CHUNK_END_>,1:8]  
8   target_language_name_statement  
9     free_input_string  
10    text_chunk_end  [@2,24:48='Abstract Machine Language'<TEXT_CHUNK_END_>,2:7]  
11 language_construct_rules  
12   indentation_block  
13     enter_block      [@3,60:60='{ '<OPEN_BRACE>,3:11]
```

Figura 19 – Árvore Sintática “main_formatter_syntax_tree.png”



Fonte: Própria

```

14     statements_list
15         match_statement
16             braced_free_input_string
17                 braced_text_chunk_end
18                     →  [@4,73:76='if\\(<BRACED_TEXT_CHUNK_END_>,4:12]
19             statements_list
20                 scope_name_statement
21                     free_input_string
22                         text_chunk_end
23                             →  [@5,95:117='if.statement.definition'<TEXT_CHUNK_END_>,5:16]
24             statements_list
25                 push_statement
26                     indentation_block
27                         enter_block      [@6,133:133='{ '<OPEN_BRACE>,6:15]
28                         statements_list
29                             meta_scope_statement
30                                 free_input_string
31                                 text_chunk_end
32                                     →  [@7,159:175='if.statement.body'<TEXT_CHUNK_END_>,7:25]

```

Figura 20 – Árvore Sintática Abstrata “main_formatter_abstract_syntax_tree.png”



Fonte: Própria

```

30     statements_list
31         match_statement
32             braced_free_input_string
33             braced_text_chunk_end
34                 ←  [@8,196:197='\\\'<BRACED_TEXT_CHUNK_END_>,8:20]
35             statements_list
36                 scope_name_statement
37                 free_input_string
38                 text_chunk_end      [@9,224:246='if.statement.definitio_']
39                     ←  n'<TEXT_CHUNK_END_>,9:24]
40             statements_list
41                 pop_statement
42                 free_input_string
43                 text_chunk_end
44                     ←  [@10,269:272='true'<TEXT_CHUNK_END_>,10:22]
45             leave_block      [@11,296:296='}'<CLOSE_BRACE>,12:9]
46             leave_block      [@12,304:304='}'<CLOSE_BRACE>,14:1]
  
```

```
44
45 Abstract Syntax Tree
46 language_syntax
47     preamble_statements
48         master_scope_name_statement InputString str: , is_out_of_scope: [], chunks:
49             ↳ [source.sma], definitions: {}, errors: [], indentations: [<indent 2, open
50                 ↳ 133, close 296>, <indent 1, open 60, close 304>], is_resolved: False;
51         target_language_name_statement      InputString str: , is_out_of_scope: [],
52             ↳ chunks: [Abstract Machine Language], definitions: {}, errors: [],
53                 indentations: [<indent 2, open 133, close 296>, <indent 1, open 60, close
54                     ↳ 304>], is_resolved: False;
55     language_construct_rules
56         indentation_block
57             statements_list
58                 match_statement
59                     InputString str: if\(), is_out_of_scope: [], chunks: [if\(), definitions:
60                         ↳ {}, errors: [], indentations: [<indent 2, open 133, close 296>,
61                             ↳ <indent 1, open 60, close 304>], is_resolved: True;
62             statements_list
63                 scope_name_statement      InputString str: , is_out_of_scope: [],
64                     ↳ chunks: [if.statement.definition], definitions: {}, errors: [],
65                         indentations: [<indent 2, open 133, close 296>, <indent 1, open 60,
66                             ↳ close 304>], is_resolved: False;
67             statements_list
68                 push_statement
69                     indentation_block
70                         statements_list
71                             meta_scope_statement  InputString str: , is_out_of_scope: [],
72                                 ↳ chunks: [if.statement.body], definitions: {}, errors: [],
73                                     indentations: [<indent 2, open 133, close 296>, <indent 1,
74                                         ↳ open 60, close 304>], is_resolved: False;
75                         statements_list
76                             match_statement
77                                 InputString str: \), is_out_of_scope: [], chunks: [\)],
78                                     ↳ definitions: {}, errors: [], indentations: [<indent 2,
79                                         ↳ open 133, close 296>, <indent 1, open 60, close 304>],
80                                         ↳ is_resolved: True;
81                         statements_list
```

```
66             scope_name_statement      InputString str: , is_out_of_scope:  
67                 ↵  [], chunks: [if.statement.definition], definitions: {},  
68                 ↵  errors: [], indentations: [<indent 2, open 133, close  
69                 ↵  296>, <indent 1, open 60, close 304>], is_resolved:  
70                 ↵  False;  
71             statements_list  
72             pop_statement      InputString str: , is_out_of_scope: [],  
73                 ↵  chunks: [true], definitions: {}, errors: [],  
74                 ↵  indentations: [<indent 2, open 133, close 296>, <indent  
75                 ↵  1, open 60, close 304>], is_resolved: False;  
76  
77 Generating 'main_formatter.html'...  
78 Generating 'main_formatter_syntax_tree.png'...  
79 Generating 'main_formatter_abstract_syntax_tree.png'...
```

APÊNDICE C – EXECUÇÃO DE “MAIN_HIGHLIGHTER.PY”

O programa do Código 27, faz a criação de 5 artefatos de resultado. Duas árvores em forma de figura, duas árvores em forma de texto e uma arquivo HTML. No Código 31, encontra-se o arquivo HTML gerado pelo programa exemplo (Código 27). Nas Figuras 21 e 22, encontram-se as imagens geradas pelo programa “main_highlighter.py” (Código 27). No Código 32, encontra-se a saída da linha de comando que se obtém ao realizar a execução do programa de exemplo (Código 27).

Ambas as Figuras 21 e 22 quanto o Código 32 representam os mesmos dados, mas mostrados de formas diferentes (figura *versus* textual) e com níveis de detalhe diferentes. No Código 32 são mostrado os nós-folhas das árvores com maior nível de detalhe. Já nas Figuras 21 e 22, os nós-folhas são mostrados com simplificações para que a figura possa ser vista em uma única tela.

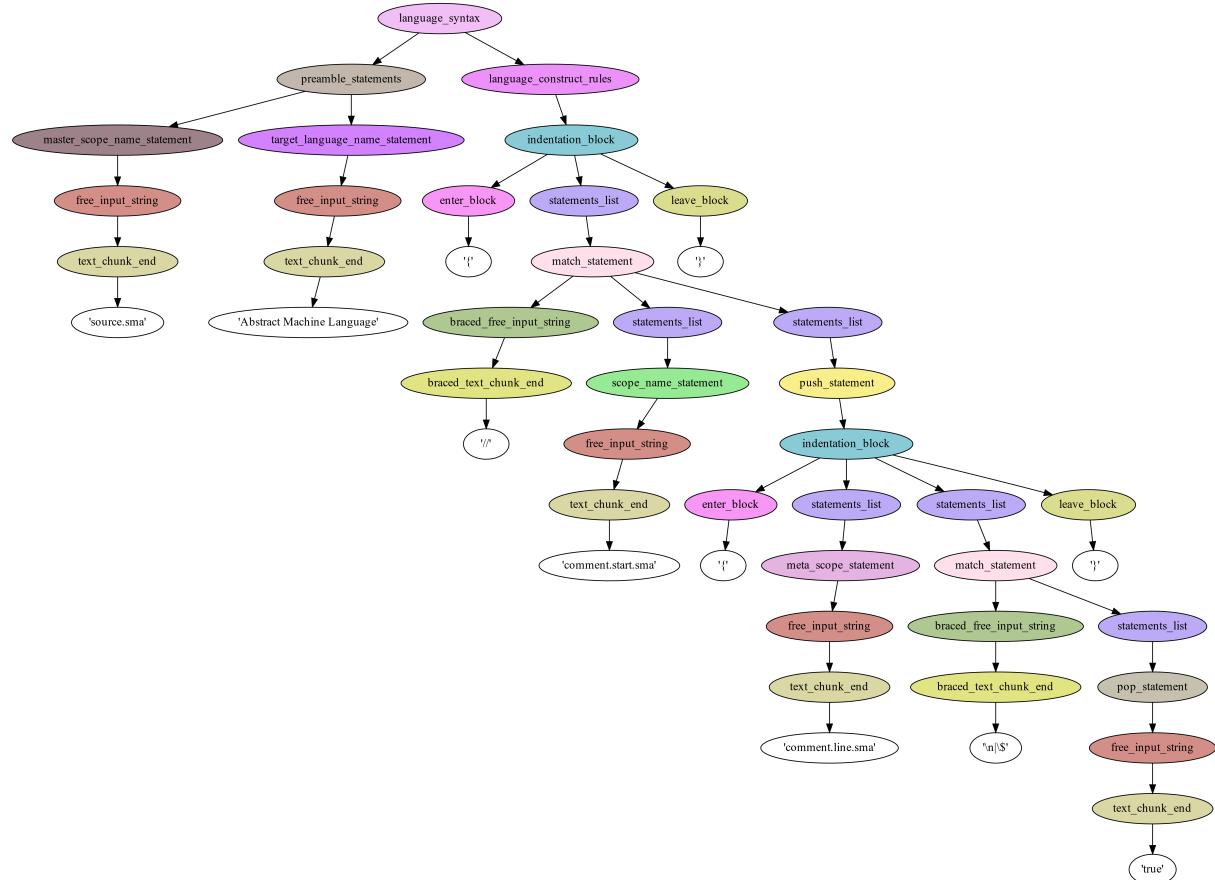
Código 31 – Arquivo HTML gerado pelo programa de exemplo “main_highlighter.py”

```
1 <!DOCTYPE html><html><head><title>Abstract Machine Language -  
2   source.sma</title></head>  
3 <body style="white-space: pre; font-family: monospace;"><font color="#FF0000"  
4   grammar_scope="comment.start.sma" theme_scope="comment">//</font><font  
5   color="#00FF00" grammar_scope="comment.line.sma" theme_scope="comment.line">  
6   Example single line commentary</font></body></html>
```

Código 32 – Resultado da execução do arquivo “source/main_highlighter.py”

```
1 $ python3 main_highlighter.py  
2 Syntax Tree  
3 language_syntax  
4   preamble_statements  
5     master_scope_name_statement  
6       free_input_string  
7         text_chunk_end  [@1,7:16='source.sma'<TEXT_CHUNK_END_>,1:8]  
8     target_language_name_statement  
9       free_input_string  
10      text_chunk_end  [@2,24:48='Abstract Machine Language'<TEXT_CHUNK_END_>,2:7]  
11 language_construct_rules  
12   indentation_block  
13     enter_block      [@3,60:60='{ '<OPEN_BRACE>,3:11]  
14     statements_list  
15     match_statement  
16     braced_free_input_string  
17       braced_text_chunk_end      [@4,73:74='//<BRACED_TEXT_CHUNK_END_>,4:12]
```

Figura 21 – Árvore Sintática “main_highlighter_syntax_tree.png”



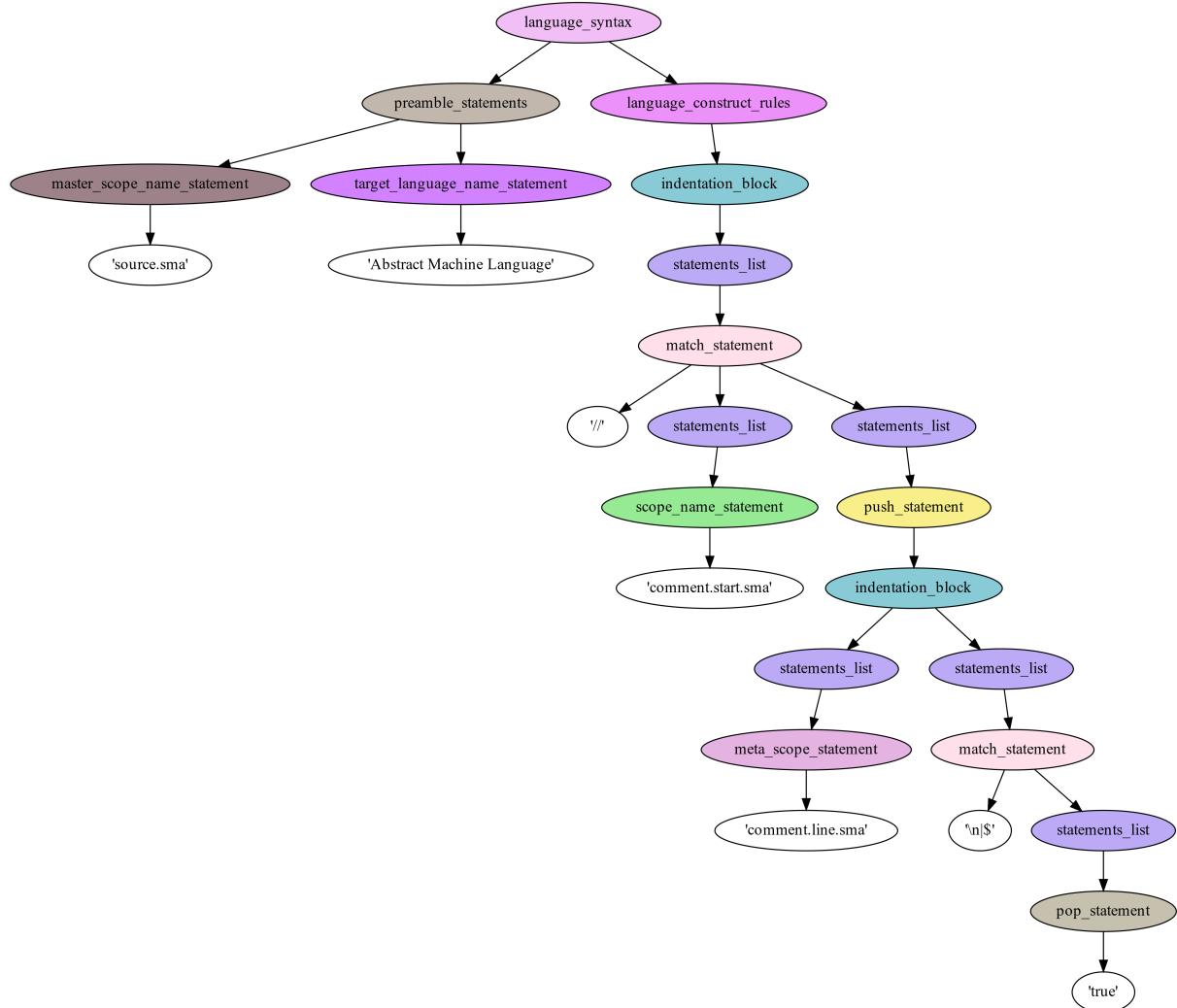
Fonte: Própria

```

18     statements_list
19         scope_name_statement
20             free_input_string
21                 text_chunk_end  [@5,93:109='comment.start.sma'<TEXT_CHUNK_END_>,5:16]
22     statements_list
23         push_statement
24             indentation_block
25                 enter_block      [@6,125:125='{ '<OPEN_BRACE>,6:15]
26                 statements_list
27                     meta_scope_statement
28                         free_input_string
29                             text_chunk_end
30                                 ←  [@7,151:166='comment.line.sma'<TEXT_CHUNK_END_>,7:25]
31     statements_list
32         match_statement
33             braced_free_input_string
34                 braced_text_chunk_end
35                     ←  [@8,187:191='\\n|\\$'<BRACED_TEXT_CHUNK_END_>,8:20]

```

Figura 22 – Árvore Sintática Abstrata “main_highlighter_abstract_syntax_tree.png”



Fonte: Própria

```

34         statements_list
35         pop_statement
36         free_input_string
37         text_chunk_end
38             ↳  [@9,216:219='true'<TEXT_CHUNK_END_>,9:22]
39         leave_block      [@10,243:243='}'<CLOSE_BRACE>,11:9]
40         leave_block      [@11,251:251='}'<CLOSE_BRACE>,13:1]
41
42 Abstract Syntax Tree
43 language_syntax
44 preamble_statements
        master_scope_name_statement InputString str: , is_out_of_scope: [], chunks:
        ↳  [source.sma], definitions: {}, errors: [], indentations: [<indent 2, open
        ↳  125, close 243>, <indent 1, open 60, close 251>], is_resolved: False;
  
```

```
45     target_language_name_statement      InputString str: , is_out_of_scope: [],
        ↵   chunks: [Abstract Machine Language], definitions: {}, errors: [],
        ↵   indentations: [<indent 2, open 125, close 243>, <indent 1, open 60, close
        ↵   251>], is_resolved: False;
46     language_construct_rules
47         indentation_block
48             statements_list
49                 match_statement
50                     InputString str: //, is_out_of_scope: [], chunks: [//], definitions: {},
        ↵   errors: [], indentations: [<indent 2, open 125, close 243>, <indent
        ↵   1, open 60, close 251>], is_resolved: True;
51             statements_list
52                 scope_name_statement      InputString str: , is_out_of_scope: [],
        ↵   chunks: [comment.start.sma], definitions: {}, errors: [],
        ↵   indentations: [<indent 2, open 125, close 243>, <indent 1, open 60,
        ↵   close 251>], is_resolved: False;
53             statements_list
54                 push_statement
55                     indentation_block
56                         statements_list
57                             meta_scope_statement  InputString str: , is_out_of_scope: [],
        ↵   chunks: [comment.line.sma], definitions: {}, errors: [],
        ↵   indentations: [<indent 2, open 125, close 243>, <indent 1,
        ↵   open 60, close 251>], is_resolved: False;
58             statements_list
59                 match_statement
60                     InputString str: \n|$, is_out_of_scope: [], chunks: [\n|\$],
        ↵   definitions: {}, errors: [], indentations: [<indent 2,
        ↵   open 125, close 243>, <indent 1, open 60, close 251>],
        ↵   is_resolved: True;
61             statements_list
62                 pop_statement      InputString str: , is_out_of_scope: [],
        ↵   chunks: [true], definitions: {}, errors: [],
        ↵   indentations: [<indent 2, open 125, close 243>, <indent
        ↵   1, open 60, close 251>], is_resolved: False;
63
64 Generating 'main_highlighter.html'...
65 Generating 'main_highlighter_syntax_tree.png'...
66 Generating 'main_highlighter_abstract_syntax_tree.png'...
```

APÊNDICE D – CÓDIGO DOS TESTES DE UNIDADE

Primeiro foi desenvolvida a ferramenta de Adição de Cores para facilitar os testes da metalinguagem e do Analisador Semântico. Uma vez que se comprovou o funcionamento da metalinguagem, foi realizada uma implementação mínima de um formatador de código-fonte. Tanto as implementações do módulo de Adição de Cores quanto do Formatador de Código são somente uma prova de conceito do que pode ser feito com a metalinguagem desenvolvida.

No Código 34, a classe “TestingGrammarUtilities” é uma classe abstrata (COOK, 2009) que contém características necessárias a todos os tipos de testes de unidades implementados, tanto os testes de unidade de Adição de Cores quanto os testes de unidade do Formatador de Código. No total existem 22 testes: i) 16 testes do Analisador Semântico (Código 35) na classe testes “TestSemanticRules”; ii) 5 testes de Adição de Cores (Código 36) na classe testes “TestCodeHighlighterBackEnd”; iii) 1 teste de Formatação de Código (Código 37) na classe testes “TestCodeFormatterBackEnd”.

A ferramenta de Adição de Cores foi a primeira desenvolvida, portanto obteve a criação de mais testes de unidade para verificar a implementação do Analisador Semântico (Código 35). Para se executar os testes de unidade, basta executar o comando “python3 unit_tests.py”. No Código 33, pode-se encontrar um exemplo de execução dos testes de unidade apresentados no Código 34.

Código 33 – Resultado da execução dos Testes de Unidade

```
1 $ python3 unit_tests.py
2 00:02:41:738.736391 1.52e-04 - source.<module>:52 - Importing __main__
3 .....
4 -----
5 Ran 22 tests in 2.086s
6
7 OK
```

Código 34 – Arquivo “source/unit_tests.py”

```
1 import os
2 import sys
3
4 import pushdown
5 import semantic_analyzer
6 import code_formatter
7 import code_highlighter
8
9 import unittest
```

```
10
11 from debug_tools import getLogger
12
13 log = getLogger( 127, os.path.basename( os.path.dirname( os.path.abspath( __file__
14     → ) ) ) )
14 log( 1, "Importing " + __name__ )
15
16 def assert_path(module):
17     if module not in sys.path:
18         sys.path.append( module )
19
20 assert_path( os.path.realpath( __file__ ) )
21
22 from debug_tools.utilities import get_relative_path
23 from debug_tools.testing_utilities import TestingUtilities
24 from debug_tools.testing_utilities import wrap_text
25
26 def main():
27     # https://stackoverflow.com/questions/6813837/stop-testsuite-if-a-testcase-fin_
28     → d-an-error
28     unittest.main( failfast=True )
29
30 def findCaller():
31     return log.findCaller()
32
33 def getCallerName():
34     return findCaller()[2]
35
36
37 # Things to improve:
38 # 1. Remove the open and close braces for block opening and closing, and make
39     → blocks indentation based
40 # 2. Implement the captures, set statements and other more statements required for
41     → performace or easy of use
42 # 3. Reimplement the whole matching logic, fixing the ordering/sequence issues of
43     → interpreting
44 # 4. Implement the new missing semantic rules on the semantic_analyzer.py
45 # 5. Reduce/decrease memory consuption and optime runtime execution performance
46 # 6. Theme operator scope matching arithmetics, i.e., function.block.c++ - block
47 class TestingGrammarUtilities(TestingUtilities):
48
49     def _getParser(self, log_level):
```

```
47
48     ## The relative path the the pushdown grammar parser file from the current
49     ## file
50     grammar_file_path = get_relative_path( "grammars_grammar.pushdown",
51     ##                                     __file__ )
52
53     ## The parser used to build the Abstract Syntax Tree and parse the input text
54     with open( grammar_file_path, "r", encoding='utf-8' ) as file:
55         my_parser = pushdown.Lark( file.read(), start='language_syntax',
56         ##                               parser='lalr', lexer='contextual', debug=log_level)
57         return my_parser
58
59
60     def _getError(self, example_grammar, return_tree=False, log_level=0):
61         example_grammar = wrap_text( example_grammar )
62
63         my_parser = self._getParser(log_level)
64         tree = my_parser.parse(example_grammar)
65
66         # function_file = get_relative_path( "examples/%s.png" % getCallerName(),
67         ##                                     __file__ )
68         # from utilities import make_png
69         # make_png( tree, get_relative_path( function_file, __file__ ) )
70
71         # https://stackoverflow.com/questions/5067604/determine-function-name-from_
72         ## -within-that-function-without-using-traceback
73         # import inspect
74         # log( 1, "%s", getCallerName() )
75         # log( 1, "%s", inspect.stack()[1][3] )
76
77
78         if return_tree:
79             new_tree = semantic_analyzer.TreeTransformer().transform( tree )
80
81             # log( 1, 'tree: \n%s', tree.pretty() )
82             # log( 1, 'tree: \n%s', new_tree.pretty() )
83             return new_tree
84
85
86         else:
87             with self.assertRaises( semantic_analyzer.SemanticErrors ) as error:
88                 new_tree = semantic_analyzer.TreeTransformer().transform( tree )
89
90             return error
```

```
84
85 class TestSemanticRules(TestingGrammarUtilities):
86
87     def test_duplicatedContext(self):
88         example_grammar = wrap_text(
89             r"""
90                 name: Abstract Machine Language
91                 scope: source.sma
92                 contexts: {
93                     match: (true|false) {
94                         scope: constant.language
95                     }
96                 }
97
98                 contexts: {
99                     match: (true|false) {
100                         scope: constant.language
101                     }
102                 }
103             """
104         )
105
106         error = self._getError( example_grammar )
107
108         self.assertTextEqual(
109             r"""
110                 + 1. Extra `contexts` rule defined in your grammar on
111                 ↳ [ @-1,130:137='contexts'<__ANON_1>, 9:1 ]
112                 """
113             , error.exception )
114
115     def test_duplicatedIncludes(self):
116         example_grammar = wrap_text(
117             r"""
118                 name: Abstract Machine Language
119                 scope: source.sma
120                 contexts: {
121                     match: (true|false) {
122                         scope: constant.language
123                     }
124                     include: duplicate
125                 }
126
127                 duplicate: {
128                     match: (true|false) {
```

```
125             scope: constant.language
126         }
127     }
128
129     duplicate: {
130         match: (true|false) {
131             scope: constant.language
132         }
133     }
134 """
135     error = self._getError( example_grammar )
136
137     self.assertTextEqual(
138         r"""
139             + 1. Duplicated include `duplicate` defined in your grammar on
140             ↪  [@-1,234:242='duplicate'<__ANON_1>,16:1]
141             "", error.exception )
142
143     def test_missingIncludeDetection(self):
144         example_grammar = wrap_text(
145             r"""
146                 name: Abstract Machine Language
147                 scope: source.sma
148                 contexts: {
149                     match: (true|false) {
150                         scope: constant.language
151                     }
152                     include: missing_include
153                 }
154             """
155             error = self._getError( example_grammar )
156
157             self.assertTextEqual(
158                 r"""
159                     + 1. Missing include `missing_include` defined in your grammar on
160                     ↪  [@-1,140:154='missing_include'<TEXT_CHUNK_END_>,7:14]
161                     "", error.exception )
162
163     def test_invalidRegexInput(self):
164         example_grammar = wrap_text(
165             r"""
166                 name: Abstract Machine Language
```

```
165         scope: source.sma
166         contexts: {
167             match: (true|false {
168                 scope: constant.language
169             }
170             )
171             """
172             error = self._getError( example_grammar )
173
174             self.assertTextEqual(
175                 r"""
176                 + 1. Invalid regular expression `(true|false` on match statement:
177                 ↳ missing ), unterminated subpattern at position 0
178                 """
179             , error.exception )
180
181
182     def test_duplicatedGlobalNames(self):
183         example_grammar = wrap_text(
184             r"""
185                 name: Abstract Machine Language
186                 name: Abstract Machine Language
187                 scope: source.sma
188                 scope: source.sma
189                 contexts: {
190                     match: (true|false) {
191                         }
192                         )
193                         """
194                         error = self._getError( example_grammar )
195
196                         self.assertTextEqual(
197                             r"""
198                             + 1. Duplicated target language name defined in your grammar on
199                             ↳ [@-1,38:62='Abstract Machine Language'<TEXT_CHUNK_END_>,2:7]
200                             + 2. Duplicated master scope name defined in your grammar on
201                             ↳ [@-1,89:98='source.sma'<TEXT_CHUNK_END_>,4:8]
202                             """
203                             , error.exception )
204
205
206     def test_missingScopeGlobalName(self):
207         example_grammar = wrap_text(
208             r"""
209                 name: Abstract Machine Language
210                 contexts: {
```

```
204             match: (true|false) {
205                 }
206             }
207         """
208     error = self._getError( example_grammar )
209
210     self.assertTextEqual(
211         r"""
212         + 1. Missing master scope name in your grammar preamble.
213         """, error.exception )
214
215     def test_missingNameGlobal(self):
216         example_grammar = wrap_text(
217             r"""
218             scope: source.sma
219             contexts: {
220                 match: (true|false) {
221                     }
222                     }
223                 """
224         error = self._getError( example_grammar )
225
226         self.assertTextEqual(
227             r"""
228             + 1. Missing target language name in your grammar preamble.
229             """, error.exception )
230
231     def test_unusedInclude(self):
232         example_grammar = wrap_text(
233             r"""
234             scope: source.sma
235             name: Abstract Machine Language
236             contexts: {
237                 match: (true|false) {
238                     }
239                 }
240
241             unused: {
242                 match: (true|false) {
243                     }
244                     }
245                 """

```

```
246     error = self._getError( example_grammar )
247
248     self.assertTextEqual(
249         r"""
250         +  Warnings:
251             + 1. Unused include `unused` defined in your grammar on
252                 → [ @-1,101:106='unused'<__ANON_1>,8:1]
253             "", error.exception )
254
255     def test_unusedConstantDeclaration(self):
256         example_grammar = wrap_text(
257             r"""
258                 scope: source.sma
259                 name: Abstract Machine Language
260                 $constant: test
261                 contexts: {
262                     match: (true|false) {
263                         }
264                     }
265                 """
266         )
267         error = self._getError( example_grammar )
268
269         self.assertTextEqual(
270             r"""
271             +  Warnings:
272                 + 1. Unused constant `'$constant:'` defined in your grammar on
273                     → [ @-1,50:59='$constant:'<CONSTANT_NAME_>,3:1]
274             "", error.exception )
275
276     def test_constantUsage(self):
277         example_grammar = wrap_text(
278             r"""
279                 scope: source.sma
280                 name: Abstract Machine Language
281                 $constant: test
282                 contexts: {
283                     match: (true$constant:$false)$constant: {
284                         }
285                     }
286                 """
287         )
288         tree = self._getError( example_grammar, True )
```

```
286     self.assertTextEqual(
287         r"""
288             + language_syntax
289             + preamble_statements
290             + master_scope_name_statement source.sma
291             + target_language_name_statement Abstract Machine Language
292             + test
293             + language_construct_rules
294             + indentation_block
295             + statements_list
296             + match_statement (true test|false) test
297         """", tree.pretty(debug=0) )
298
299     def test_isolatedConstantUsage(self):
300         my_parser = pushdown.Lark(
301             r"""
302                 free_input_string: ( constant_usage | text_chunk )* ( constant_usage_end
303                 ← | text_chunk_end )
304                 constant_usage: CONSTANT_USAGE_
305                 text_chunk: TEXT_CHUNK_
306                 constant_usage_end: CONSTANT_USAGE_END_
307                 text_chunk_end: TEXT_CHUNK_END_
308
309                 CONSTANT_USAGE_: /\$[^\n\$\:]+\:/
310                 TEXT_CHUNK_: /(\{\|\}|\$\|[^$\{]\$)+/
311                 CONSTANT_USAGE_END_: /(?:\$[^\n\$\:]+\:)(?=(:\n\$))/
312                 TEXT_CHUNK_END_: /(\{\|\}|\$\|[^$\{]\$)+(?=(:\n\$))/
313             """,
314             start='free_input_string', parser='lalr', lexer='contextual' )
315         tree = my_parser.parse( r"true$constant:\$|false" )
316
317         self.assertTextEqual(
318             r"""
319                 + free_input_string
320                 + text_chunk  [@1,0:3='true'<TEXT_CHUNK_>,1:1]
321                 + constant_usage  [@2,4:13='$constant:<CONSTANT USAGE_>,1:5]
322                 + text_chunk_end  [@3,14:21='\\$|false'<TEXT_CHUNK_END_>,1:15]
323             """", tree.pretty(debug=True) )
324
325     def test_isolatedBracedEnd(self):
326         my_parser = pushdown.Lark(
327             r""""
```

```
327         start: braced_free_input_string " {"
328
329         constant_usage: CONSTANT_USAGE_
330         text_chunk: TEXT_CHUNK_
331         CONSTANT_USAGE_: /\$[^n$:]++:/#
332         TEXT_CHUNK_: /(\\{|\\}|\\$|[^n{}$])+/
333
334         braced_free_input_string: ( constant_usage | text_chunk )* (
335             → braced_constant_usage_end | braced_text_chunk_end )
336             braced_constant_usage_end: BRACED_CONSTANT_USAGE_END_
337             braced_text_chunk_end: BRACED_TEXT_CHUNK_END_
338             BRACED_CONSTANT_USAGE_END_: /(?:\$[^n$:]++:)=(?:\u007b))/#
339             BRACED_TEXT_CHUNK_END_: /(\\{|\\}|\\$|[^n{}$])+(?:\u007b))/#
340             """
341             start='start', parser='lalr', lexer='contextual' )
342             tree = my_parser.parse( r"true$constant:$|false {" )
343
344             self.assertTextEqual(
345                 r"""
346                 + start
347                 + braced_free_input_string
348                 + text_chunk  [@1,0:3='true'<TEXT_CHUNK_>,1:1]
349                 + constant_usage  [@2,4:13='$constant:'<CONSTANT_USAGE_>,1:5]
350                 + braced_text_chunk_end
351             → [@3,14:21='\\$|false'<BRACED_TEXT_CHUNK_END_>,1:15]
352             """
353             , tree.pretty(debug=True) )
354
355     def test_redefinedConst(self):
356         example_grammar = wrap_text(
357             r"""
358             scope: source.sma
359             name: Abstract Machine Language
360             $constant: test
361             contexts: {
362                 $constant: test
363                 match: (true$constant:$|false) {
364                     }
365                 }
366             """
367             )
368             error = self._getError( example_grammar )
369
370             self.assertTextEqual(
```

```
367         r"""
368             + 1. Constant redefinition on
369             ↳ [-1,82:91='$constant:<CONSTANT_NAME_>,5:5]
370                 "", error.exception )
371
372     def test_recursiveConstantDefinition(self):
373         example_grammar = wrap_text(
374             r"""
375                 scope: source.sma
376                 name: Abstract Machine Language
377                 $constant: test$constant:
378                 contexts: {
379                     match: (true$constant:|false) {
380                         }
381                     }
382                 """
383         error = self._getError( example_grammar )
384
385         self.assertTextEqual(
386             r"""
387                 + Warnings:
388                 + 1. Recursive constant definition on
389                 ↳ [-1,50:59='$constant:<CONSTANT_NAME_>,3:1]
390                     "", error.exception )
391
392     def test_usingConstOutOfScope(self):
393         example_grammar = wrap_text(
394             r"""
395                 scope: source.sma
396                 name: Abstract Machine Language
397                 contexts: {
398                     match: (true$constant:|false) {
399                         }
400                     $constant: test
401                     }
402                 """
403         error = self._getError( example_grammar )
404
405         self.assertTextEqual(
406             r"""
407                 + 1. Using constant `$constant:` out of scope on
408                 + [-1,82:91='$constant:<CONSTANT_USAGE_>,4:21] from
```

```
407         +      [ @-1,112:121= '$constant: '<CONSTANT_NAME_>, 6:5]
408         """", error.exception )
409
410     def test_usingConstOutOfBlockDefinition(self):
411         example_grammar = wrap_text(
412             r"""
413             scope: source.sma
414             name: Abstract Machine Language
415             contexts: {
416                 $constant: test:
417                 match: (true $constant:|false) {
418                     include: block
419                 }
420             }
421
422             block: {
423                 match: ($constant:) {
424                 }
425             }
426         """
427         )
428         error = self._getError( example_grammar )
429
430         self.assertTextEqual(
431             r"""
432                 + 1. Using constant `'$constant:'` out of block on
433                 +      [ @-1,173:182= '$constant: '<CONSTANT_USAGE_>, 11:13] from
434                 +      [ @-1,66:75= '$constant: '<CONSTANT_NAME_>, 4:5]
435                 """
436         , error.exception )
437
438     class TestCodeHighlighterBackEnd(TestingGrammarUtilities):
439
440         def _getBackend(self, example_grammar, example_program, example_theme):
441             function_file = get_relative_path( "examples/%s.html" % getCallerName(),
442                 → __file__ )
443             # log( 1, "function_file: %s", function_file )
444
445             tree = self._getError( example_grammar, True )
446             backend = code_highlighter.Backend(tree, example_program, example_theme)
447             generated_html = backend.generated_html()
```

```

447     with open( function_file, 'w', newline='\n', encoding='utf-8' ) as
448         output_file:
449             output_file.write( generated_html )
450             output_file.write("\n")
451
452     return generated_html
453
454 def test_simpleMatchStatement(self):
455     example_grammar = wrap_text(
456         r"""
457             scope: source.sma
458             name: Abstract Machine Language
459             contexts: {
460                 match: (true|false) {
461                     scope: boolean.sma
462                 }
463             """
464     )
465
466     example_program = wrap_text(
467         r"""true"""
468 )
469
470     example_theme = \
471     {
472         "boolean" : "#FF0000",
473     }
474
475     generated_html = self._getBackend(example_grammar, example_program,
476                                         example_theme)
477
478     self.assertTextEqual(
479         r"""
480             + <!DOCTYPE html><html><head><title>Abstract Machine Language -
481             source.sma</title></head>
482                 + <body style="white-space: pre; font-family: monospace;"><font
483                 color="#FF0000" grammar_scope="boolean.sma"
484                 theme_scope="boolean">true</font></body></html>
485             """, generated_html )
486
487     def test_unmatchedProgramCompletionAtEnd(self):
488         example_grammar = wrap_text(
489             r"""

```

```
484     scope: source.sma
485     name: Abstract Machine Language
486     contexts: {
487         match: // {
488             scope: comment.line.start.sma
489         }
490     }
491     """
492 )
493
494     example_program = wrap_text(
495     r"""
496         // Example single line commentary
497     """
498
499     example_theme = \
500     {
501         "comment" : "#FF0000",
502     }
503
504     generated_html = self._getBackend(example_grammar, example_program,
505                                     example_theme)
506
507     self.assertTextEqual(
508     r"""
509         + <!DOCTYPE html><html><head><title>Abstract Machine Language -
510             source.sma</title></head>
511             + <body style="white-space: pre; font-family: monospace;"><font
512                 color="#FF0000" grammar_scope="comment.line.start.sma"
513                 theme_scope="comment">//</font><span grammar_scope="none" theme_scope="none">
514                 Example single line commentary</span></body></html>
515             """
516             , generated_html )
517
518     def test_unmatchedProgramCompletionAtMiddle(self):
519         example_grammar = wrap_text(
520             r"""
521                 scope: source.sma
522                 name: Abstract Machine Language
523                 contexts: {
524                     match: // {
525                         scope: comment.line.start.sma
526                     }
527                     match: single {
```

```
521             scope: comment.middle.start.sma
522         }
523     }
524     """
525 )
526
527     example_program = wrap_text(
528         r"""
529             // Example single line commentary
530         """
531
532     example_theme = \
533     {
534         "comment" : "#FF0000",
535     }
536
537     generated_html = self._getBackend(example_grammar, example_program,
538         example_theme)
539
540     self.assertTextEqual(
541         r"""
542             + <!DOCTYPE html><html><head><title>Abstract Machine Language -
543             source.sma</title></head>
544             + <body style="white-space: pre; font-family: monospace;"><font
545             color="#FF0000" grammar_scope="comment.line.start.sma"
546             theme_scope="comment">//</font><span grammar_scope="none" theme_scope="none">
547             Example </span><font color="#FF0000" grammar_scope="comment.middle.start.sma"
548             theme_scope="comment">single</font><span grammar_scope="none"
549             theme_scope="none"> line commentary</span></body></html>
550         """
551         , generated_html )
552
553     def test_simplePushPopStatement(self):
554         example_grammar = wrap_text(
555             r"""
556                 scope: source.sma
557                 name: Abstract Machine Language
558                 contexts: {
559                     match: // {
560                         scope: comment.start.sma
561                         push: {
562                             meta_scope: comment.line.sma
563                             match: \n|\$ {
564                                 pop: true
565                             }
```

```
556         }
557     }
558 }
559 }
560 """ )
561
562     example_program = wrap_text(
563     r"""
564         // Example single line commentary
565     """ )
566
567     example_theme = \
568     {
569         "comment" : "#FF0000",
570         "comment.line" : "#00FF00",
571     }
572
573     generated_html = self._getBackend( example_grammar, example_program,
574                                     example_theme )
575
576     self.assertTextEqual(
577     r"""
578         + <!DOCTYPE html><html><head><title>Abstract Machine Language -
579         source.sma</title></head>
580             + <body style="white-space: pre; font-family: monospace;"><font
581             color="#FF0000" grammar_scope="comment.start.sma"
582             theme_scope="comment">//</font><font color="#00FF00"
583             grammar_scope="comment.line.sma" theme_scope="comment.line"> Example single
584             line commentary</font></body></html>
585         """ , generated_html )
586
587     def test_complexGrammarFile(self):
588         example_grammar = wrap_text(
589         r"""
590             scope: source.sma
591             name: Abstract Machine Language
592             contexts: {
593                 include: pawn_keywords
594                 include: pawn_comment
595                 include: pawn_boolean
596                 include: pawn_preprocessor
597                 include: pawn_string
```

```
592         include: pawn_function
593         include: pawn_numbers
594     }
595     pawn_boolean: {
596         match: (true|false) {
597             scope: boolean.sma
598         }
599     }
600     pawn_comment: {
601         match: /\*/ {
602             scope: comment.begin.sma
603             push: {
604                 meta_scope: comment.sma
605                 match: \*/ {
606                     pop: true
607                 }
608             }
609         }
610         match: // {
611             push: {
612                 meta_scope: comment.documentation.sma
613                 match: \n {
614                     pop: true
615                 }
616             }
617         }
618     }
619     pawn_preprocessor: {
620         match: \s*#define {
621             scope: function.definition.sma
622             push: {
623                 meta_scope: meta.preprocessor.sma
624                 match: \n {
625                     pop: true
626                 }
627             }
628         }
629     }
630     pawn_string: {
631         match: "(?=.*)" {
632             scope: punctuation.definition.string.begin.sma
633             push: {
```

```
634         meta_scope: string.quoted.double.sma
635         match: "(?!.*)"
636             scope: punctuation.definition.string.end.sma
637             pop: true
638         }
639     }
640 }
641 }
642 pawn_function: {
643     match: (\w+)\s*(\(.*\))
644         scope: function.call.sma
645     }
646 }
647 pawn_numbers: {
648     match: \d+\.\?\d*
649         scope: constant.numeric.sma
650     }
651 }
652 pawn_keywords: {
653     match: \b(sizeof|charsmax|assert|break|case|continue|default|do|in|els_
654     ↪ e|exit|for|goto|if|return|switch|while)\b
655     ↪ {
656         scope: keyword.control.sma
657     }
658     match: \b(var|new)\b
659         scope: keyword.new.sma
660     "}
661     """
662     example_program = wrap_text(
663         r"""
664         /* Commentary example */ true or false
665         #define GLOBAL_CONSTANT
666         var string = "My string definition"
667         void function() {
668
669             // Single line commentary
670             var number = 100.0
671             for var index = 5999; index < sizeof number; ++index:
672
673                 /* More multiline
```

```
674         comments
675         with a bunch of
676         lines */
677         for variable in list:
678             print( variable )
679
680
681     #define MORE_CRAZYNES 100000
682     if index == 0:
683         // More singleline comments
684         // with incredicle single linearity
685         var string = "Cool beatiful String"
686
687         while string in list:
688             exit( string )
689
690     }
691 """
692
693 example_theme = \
694 {
695     "boolean" : "#FF0000",
696     "comment" : "#00FF00",
697     "function" : "#DDB700",
698     "keyword.new" : "#FF00FF",
699     "meta" : "#0000FF",
700     "storage" : "#8000FF",
701     "string" : "#808080",
702     "punctuation" : "#FF0000",
703     "constant" : "#99CC99",
704     "keyword" : "#804000",
705     "comment.documentation" : "#248591",
706 }
707
708 generated_html = self._getBackend(example_grammar, example_program,
709                                     example_theme)
710
711 self.assertTextEqual(
712     r"""
713         + <!DOCTYPE html><html><head><title>Abstract Machine Language -
714         source.sma</title></head>
```

```

713      + <body style="white-space: pre; font-family: monospace;"><font
    ↵  color="#00FF00" grammar_scope="comment.begin.sma"
    ↵  theme_scope="comment">/*</font><font color="#00FF00"
    ↵  grammar_scope="comment.sma" theme_scope="comment"> Commentary example
    ↵  */</font><span grammar_scope="none" theme_scope="none"> </span><font
    ↵  color="#FF0000" grammar_scope="boolean.sma"
    ↵  theme_scope="boolean">true</font><span grammar_scope="none"
    ↵  theme_scope="none"> or </span><font color="#FF0000"
    ↵  grammar_scope="boolean.sma" theme_scope="boolean">false</font><font
    ↵  color="#DDB700" grammar_scope="function.definition.sma"
    ↵  theme_scope="function"><br />#define</font><font color="#0000FF"
    ↵  grammar_scope="meta.preprocessor.sma" theme_scope="meta"> GLOBAL_CONSTANT<br
    ↵  /></font><font color="#FF00FF" grammar_scope="keyword.new.sma"
    ↵  theme_scope="keyword.new">var</font><span grammar_scope="none"
    ↵  theme_scope="none"> string = </span><font color="#FF0000"
    ↵  grammar_scope="punctuation.definition.string.begin.sma"
    ↵  theme_scope="punctuation">"</font><font color="#808080"
    ↵  grammar_scope="string.quoted.double.sma" theme_scope="string">My string
    ↵  definition</font><font color="#FF0000"
    ↵  grammar_scope="punctuation.definition.string.end.sma"
    ↵  theme_scope="punctuation">"</font><span grammar_scope="none"
    ↵  theme_scope="none"><br />void </span><font color="#DDB700"
    ↵  grammar_scope="function.call.sma"
    ↵  theme_scope="function">function()</font><span grammar_scope="none"
    ↵  theme_scope="none"> {<br /><br /> //</span><font color="#248591"
    ↵  grammar_scope="comment.documentation.sma"
    ↵  theme_scope="comment.documentation"> Single line commentary<br /></font><span
    ↵  grammar_scope="none" theme_scope="none"> </span><font color="#FF00FF"
    ↵  grammar_scope="keyword.new.sma" theme_scope="keyword.new">var</font><span
    ↵  grammar_scope="none" theme_scope="none"> number = </span><font """

```

714

715 # Break this unit test result into two parts otherwise Latex cannot
 ↵ including this file: `Dimension too large`

```

716
+ r"""color="#99CC99" grammar_scope="constant.numeric.sma"
→ theme_scope="constant">100.0</font><span grammar_scope="none"
→ theme_scope="none"><br />      </span><font color="#804000"
→ grammar_scope="keyword.control.sma"
→ theme_scope="keyword">for</font><span grammar_scope="none"
→ theme_scope="none"> </span><font color="#FF00FF"
→ grammar_scope="keyword.new.sma"
→ theme_scope="keyword.new">var</font><span grammar_scope="none"
→ theme_scope="none"> index = </span><font color="#99CC99"
→ grammar_scope="constant.numeric.sma"
→ theme_scope="constant">5999</font><span grammar_scope="none"
→ theme_scope="none">; index &lt; </span><font color="#804000"
→ grammar_scope="keyword.control.sma"
→ theme_scope="keyword">sizeof</font><span grammar_scope="none"
→ theme_scope="none"> number; ++index:<br /><br />
→ </span><font color="#00FF00" grammar_scope="comment.begin.sma"
→ theme_scope="comment">/*</font><font color="#00FF00"
→ grammar_scope="comment.sma" theme_scope="comment"> More
→ multiline<br />           comments<br />           with a bunch
→ of<br />           lines *</font><span grammar_scope="none"
→ theme_scope="none"><br />           </span><font color="#804000"
→ grammar_scope="keyword.control.sma"
→ theme_scope="keyword">for</font><span grammar_scope="none"
→ theme_scope="none"> variable </span><font color="#804000"
→ grammar_scope="keyword.control.sma"
→ theme_scope="keyword">in</font><span grammar_scope="none"
→ theme_scope="none"> list:<br />           </span><font
→ color="#DDB700" grammar_scope="function.call.sma"
→ theme_scope="function">print( variable )</font><span
→ grammar_scope="none" theme_scope="none"><br />     }</span><font
→ color="#DDB700" grammar_scope="function.definition.sma"
→ theme_scope="function"><br /><br /> #define</font><font
→ color="#0000FF" grammar_scope="meta.preprocessor.sma"
→ theme_scope="meta"> MORE_CRAZINESS 10000<br /></font><span
→ grammar_scope="none" theme_scope="none">     </span><font """

```

717

```

718 # Break this unit test result into two parts otherwise Latex cannot
→ including this file: `Dimension too large`
```

```
719 + r"""color="#804000" grammar_scope="keyword.control.sma"
    ↳ theme_scope="keyword">if</font><span grammar_scope="none"
    ↳ theme_scope="none"> index == </span><font color="#99CC99"
    ↳ grammar_scope="constant.numeric.sma"
    ↳ theme_scope="constant">0</font><span grammar_scope="none"
    ↳ theme_scope="none">:<br />           //</span><font color="#248591"
    ↳ grammar_scope="comment.documentation.sma"
    ↳ theme_scope="comment.documentation"> More singleline comments<br
    ↳ /></font><span grammar_scope="none" theme_scope="none">
    ↳ //</span><font color="#248591"
    ↳ grammar_scope="comment.documentation.sma"
    ↳ theme_scope="comment.documentation"> with incredicle single
    ↳ linearity<br /></font><span grammar_scope="none"
    ↳ theme_scope="none">           </span><font color="#FF00FF"
    ↳ grammar_scope="keyword.new.sma"
    ↳ theme_scope="keyword.new">var</font><span grammar_scope="none"
    ↳ theme_scope="none"> string = </span><font color="#FF0000"
    ↳ grammar_scope="punctuation.definition.string.begin.sma"
    ↳ theme_scope="punctuation">"</font><font color="#808080"
    ↳ grammar_scope="string.quoted.double.sma"
    ↳ theme_scope="string">Cool beatiful String</font><font
    ↳ color="#FF0000"
    ↳ grammar_scope="punctuation.definition.string.end.sma"
    ↳ theme_scope="punctuation">"</font><span grammar_scope="none"
    ↳ theme_scope="none"><br /><br />           </span><font color="#804000"
    ↳ grammar_scope="keyword.control.sma"
    ↳ theme_scope="keyword">while</font><span grammar_scope="none"
    ↳ theme_scope="none"> string </span><font color="#804000"
    ↳ grammar_scope="keyword.control.sma"
    ↳ theme_scope="keyword">in</font><span grammar_scope="none"
    ↳ theme_scope="none"> list:<br />           </span><font
    ↳ color="#804000" grammar_scope="keyword.control.sma"
    ↳ theme_scope="keyword">exit</font><span grammar_scope="none"
    ↳ theme_scope="none">(<span grammar_scope="none"> string </span>)<br /> }<br
    ↳ />}</span></body></html>
720     """", generated_html )
721
722
723 class TestCodeFormatterBackEnd(TestingGrammarUtilities):
724
725     def _getBackend(self, example_grammar, example_program, example_settings):
```

```
726     function_file = get_relative_path( "examples/%s.html" % getCallerName(),
    ↵     __file__ )
727     # log( 1, "function_file: %s", function_file )
728
729     tree = self._getError( example_grammar, True )
730     backend = code_formatter.Backend( code_formatter.SingleSpaceFormatter,
    ↵     tree, example_program, example_settings )
731     generated_html = backend.generated_html()
732
733     with open( function_file, 'w', newline='\n', encoding='utf-8' ) as
    ↵     output_file:
        output_file.write( generated_html )
        output_file.write("\n")
736
737     return generated_html
738
739 def test_singleIfStatement(self):
740     example_grammar = wrap_text(
741         r"""
742             scope: source.sma
743             name: Abstract Machine Language
744             contexts: {
745                 match: if\(
746                     scope: if.statement.definition
747                     push: {
748                         meta_scope: if.statement.body
749                         match: \) {
750                             scope: if.statement.definition
751                             pop: true
752                         }
753                     }
754                 }
755             }
756         """
757
758     example_program = wrap_text(
759         r"""if(something) bar"""
760     )
761
762     example_settings = {
763         "if.statement.body" : 2,
764     }
```

```
765     generated_html = self._getBackend(example_grammar, example_program,
    ↵   example_settings)
766
767     self.assertTextEqual(
768       r"""
769         + <!DOCTYPE html><html><head><title>Abstract Machine Language -
    ↵   source.sma</title></head>
770         + <body style="white-space: pre; font-family: monospace;"><span
    ↵   setting="unformatted" grammar_scope="if.statement.definition" setting_scope=""
    ↵   original_program="if(>if(</span><span setting="2"
    ↵   grammar_scope="if.statement.body" setting_scope="if.statement.body"
    ↵   original_program="something"> something </span><span setting="unformatted"
    ↵   grammar_scope="if.statement.definition" setting_scope=""
    ↵   original_program=")">)</span><span grammar_scope="none" setting_scope="none">
    ↵   bar</span></body></html>
771       "", generated_html )
772
773
774 if __name__ == "__main__":
775     main()
```

APÊNDICE E – CÓDIGO DO ANALISADOR SEMÂNTICO

De todos os códigos-fonte criados neste trabalho, o Analisador Semântico (Código 35) é o maior deles. Sua implementação é utilizada diretamente pelo módulo de Adição de Cores e Formatação de Código. As regras do Analisador Semântico estão divididas entre dois tipos, erros e alertas. Um erro é algo que impede completamente a gramática final de funcionar. Um alerta é algo que pode precisar ser revisado ou ignorado.

A seguir, pode-se encontrar algumas regras semânticas implementadas pelo Analisador Semântico (Código 35). A classe “`TestSemanticRules`” (Código 34), apresenta testes automatizados do Analisador Semântico. O nome de cada uma das regras a seguir começa com o prefixo “`test_`” que corresponde ao nome do teste de unidade automatizado, criado para verificar tal regra semântica na classe de testes “`TestSemanticRules`” (Código 34).

`test_duplicatedContext 1)` Detecção de contextos duplicados e emissão de um erro semântico;

`test_duplicatedIncludes 2)` Detecção de inclusões duplicadas e emissão de um erro semântico;

`test_invalidRegexInput 3)` Detecção de expressões regulares inválidas e emissão de um erro semântico;

`test_missingIncludeDetection 4)` Detecção da inclusão um bloco inexistente e emissão de um erro semântico;

`test_duplicatedGlobalNames 5)` Detecção de múltiplas definições do nome da gramática e emissão de um erro semântico;

`test_missingScopeGlobalName 6)` Detecção da falta da definição do nome do escopo global da gramática e emissão de um erro semântico;

`test_missingNameGlobal 7)` Detecção de esquecer de definir o nome da gramática e emissão de um erro semântico;

`test_unusedInclude 8)` Detecção de criação de um contexto e em esquecer de utilizar ele e emissão de um alerta;

`test_redefinedConst 9)` Detecção de redefinir um valor constante e emissão de um erro semântico;

`test_unusedConstantDeclaration 10)` Detecção de definir um valor constante e esquecer de utilizar ele e emissão de um alerta;

test_usingConstOutOfScope 11) Detecção de tentativa de usar um valor constante fora do escopo dele e emissão de um erro semântico.

Código 35 – Arquivo “source/semantic_analyzer.py”

```
1 import re
2 import pushdown
3
4 from pushdown import Tree, LarkError, Token, Discard
5
6 from debug_tools import getLogger
7 from debug_tools.utilities import get_representation
8
9 log = getLogger(__name__)
10
11
12 class SemanticErrors(LarkError):
13     def __init__(self, warnings, errors):
14         if warnings:
15             self.warnings = self._build_messages(warnings)
16         else:
17             self.warnings = None
18
19         if errors:
20             self.errors = self._build_messages(errors)
21         else:
22             self.errors = None
23
24     def _build_messages(self, exceptions):
25         # https://stackoverflow.com/questions/16625068/combine-lists-by-joining-st
26         # rings-with-matching-index-values
27         messages = map( lambda a, b: "%s. %s" % (a, b), range( 1, len(exceptions) +
28             1 ), exceptions )
29         return "\n".join( message for message in messages )
30
31     def __str__(self):
32         return ( "\n%s\n" % self.errors if self.errors else "" ) + \
33             ( "\n  Warnings:\n%s" % self.warnings if self.warnings else "" )
34
35 class CommandBlock(object):
```

```
36     def __init__(self, indentation, open_position):
37         super(CommandBlock, self).__init__()
38         self.indentation = indentation
39         self.open_position = open_position
40         self.close_position = 0
41
42     def __repr__(self):
43         return "<indent %s, open %s, close %s>" % ( self.indentation,
44             ↪ self.open_position, self.close_position )
45
46     __str__ = __repr__
47
48 class UndefinedInput(object):
49
50     def __init__(self):
51         super(UndefinedInput, self).__init__()
52         self.str = ""
53
54     def resolve(self, value):
55         raise NotImplementedError( "%s is an abstract class" % self.__name__ )
56
57     def __repr__(self):
58         return get_representation(self)
59
60     def __str__(self):
61
62         if self.str:
63             return self.str
64
65         return self.__repr__()
66
67
68 class ConstantUsage(UndefinedInput):
69     """
70         Represents a constant which is used somewhere, but its definition is yet
71         ↪ unknown.
72
73         As soon as this constant definition is known, this object will return the
74         constant complete representation.
75         """
76
77     def __init__(self, token):
```

```
76     super(ConstantUsage, self).__init__()
77     self.name = str( token )
78     self.token = token
79
80     def __repr__(self):
81         return self.name
82
83     def resolve(self, value):
84
85         if self.str:
86             return False
87
88         self.str = str( value )
89         return True
90
91
92 class ConstantDefinition(UndefinedInput):
93
94     def __init__(self, token, input_string ):
95         super(ConstantDefinition, self).__init__()
96         self.token = token
97         self.input_string = input_string
98
99     def __repr__(self):
100        return str(self.input_string)
101
102    def resolve(self):
103
104        if self.str:
105            raise RuntimeError("You cannot resolve a constant declaration twice!")
106
107        input_string = str( self.input_string )
108
109        if self.input_string.str:
110            self.str = input_string
111            return True
112
113        return False
114
115
116 class InputString(UndefinedInput):
117     """
```

```
118     Always recalculate itself when asked for its string form because it is not
119     → always beforehand know.
120     """
121
122     unescape_control_characters = re.compile( r"\\([${}])" )
123
124     def __init__(self, chunks, definitions, errors, indentations):
125         super(InputString, self).__init__()
126         self.is_out_of_scope = []
127
128         self.chunks = chunks
129         self.definitions = definitions
130         self.errors = errors
131         self.indentations = indentations
132         self.is_resolved = False
133
134     def __str__(self):
135         """
136             @param `definitions` a dictionary with all completely known
137             constants. For example { "$varrrr": " varrrr " }
138
139         if self.is_resolved: return self.str
140
141         scope_usage_error = ""
142         is_resolved = True
143         resolutions = []
144
145         # log( 1, 'self.chunks %s', self.chunks )
146         # log( 1, 'self.definitions %s', self.definitions )
147
148         for usage in self.chunks:
149             # log( 1, 'usage %s', usage )
150             # log( 1, 'usage %s', type(usage) )
151             # log( 1, 'usage.name %s', usage.name )
152             # log( 1, 'usage.name %s', type(usage.name) )
153             usage_name_in_self_definitions = usage.name in self.definitions
154
155             if usage.str or usage_name_in_self_definitions:
156                 definition = usage
157
158                 if usage_name_in_self_definitions:
159                     definition = self.definitions[usage.name]
```

```
159     usage_position = usage.token.pos_in_stream
160     definition_position = definition.token.pos_in_stream
161
162     # log(1, 'usage_position', usage_position, usage.token.pretty())
163     # log(1, 'definition_position', definition_position,
164           ↪ definition.token.pretty())
165     if definition_position > usage_position:
166         scope_usage_error = "Using constant `%s` out of scope on\n    %s"
167             ↪ from\n    %s" % (
168                 definition.token, usage.token.pretty(),
169                 ↪ definition.token.pretty() )
170
171
172     else:
173         definition_block = CommandBlock(-1, 0)
174         # log('definition_position', definition_position)
175
176         for command in self.indentations:
177             indent, open_position, close_position = command.indentation,
178                 ↪ command.open_position, command.close_position
179             # log('indent', indent, 'open_position', open_position,
180                 ↪ 'close_position', close_position)
181
182             if definition_position > open_position:
183
184                 if definition_position < close_position:
185
186                     if indent > definition_block.indentation:
187                         definition_block = command
188                         # log('setting definition_block', definition_block)
189
190                 else:
191
192                     # possible global token
193                     if definition_block.indentation < 0:
194                         continue
195
196
197                 else:
198
199                     break
200
201
202             # log('resolution for definition_block', definition_block)
203             if definition_block.indentation > -1:
```

```
196         if not ( usage_position > definition_block.open_position and
197             ↪   usage_position < definition_block.close_position ):
198             scope_usage_error = "Using constant `%s` out of block
199             ↪   on\n    %s from\n    %s" % (
200                 definition.token, usage.token.pretty(),
201                 ↪   definition.token.pretty() )
202
203             resolutions.append( str( definition ) )
204
205     else:
206         # log( 1, 'is_resolved False' )
207         is_resolved = False
208         resolutions.append( str( usage.name ) )
209
210     if is_resolved:
211         self.is_resolved = True
212         self.str = self.unescape_control_characters.sub( "\\\\"1", "" .join(
213             ↪   resolutions ) )
214         if scope_usage_error: self.errors.append( scope_usage_error )
215
216     return self.str
217
218
219
220
221
222 class TreeTransformer(pushdown.Transformer):
223     """
224
225         Transforms the Derivation Tree nodes into meaningful string representations,
226         allowing simple recursive parsing and conversion to Abstract Syntax Tree.
227     """
228
229
230     def __init__(self):
231         ## Saves all the semantic errors detected so far
232         self.errors = []
233
234         ## Saves all warnings noted so far
235         self.warnings = []
236
237         ## Whether the mandatory/obligatory global scope name statement was declared
238         self.is_master_scope_name_set = False
239
240         ## Whether the mandatory/obligatory global language name statement was
241         ↪   declared
```

```
233     self.is_target_language_name_set = False
234
235     ## Can only be one scope called `contexts`
236     self.has_called_language_construct_rules = False
237
238     ## Pending constants declarations
239     self.constant_usages = {}
240
241     ## Pending constants usages
242     self.constant_definitions = {}
243
244     ## A list of miscellaneous_language_rules include contexts defined for
245     #→ duplication checking
246     self.defined_includes = {}
247
248     ## A list of required includes to check for missing includes
249     self.required_includes = {}
250
251     ## A list of regular expressions used on match statements,
252     ## for validation when the constants definitions are completely known
253     self.pending_match_statements = []
254
255     ## Responsible for calculating all open and close commands scoping
256     self.open_blocks = {}
257     self.indentation_level = 0
258     self.indentation_blocks = []
259
260     def language_syntax(self, tree, children):
261         """
262             This is the grammar start symbol and will be called by last with all
263             partial subtrees the start symbol derivates.
264
265             This is a great place to check whether global grammar properties
266             where set.
267
268             if not self.has_called_language_construct_rules:
269                 self.errors.append( "You must to define the `contexts` block in your
270                     #→ grammar!" )
271
272                 self._resolve_constants_definitions()
273                 self._check_includes_definitions()
```

```
273     self._check_for_main_rules()
274
275     if self.errors or self.warnings:
276         # log('tree\n', children[1].pretty())
277         raise SemanticErrors(self.warnings, self.errors)
278
279     return self.__default__(tree, children)
280
281 def language_construct_rules(self, tree, children):
282     self.has_called_language_construct_rules = True
283     return self.__default__(tree, children)
284
285 def miscellaneous_language_rules(self, tree, children):
286     first_token = children[0]
287     include_name = str(first_token)
288     assert tree.data == 'miscellaneous_language_rules', "Just documenting what
289     ↪ the data attribute has."
290     assert isinstance(first_token, Token), "The first children must be a
291     ↪ Token, while the second is a subtree."
292
293     if include_name in self.defined_includes:
294         self.errors.append("Duplicated include `%s` defined in your grammar on
295     ↪ %s" % (include_name, first_token.pretty() ) )
296
297     if include_name == 'contexts':
298         self.errors.append("Extra `contexts` rule defined in your grammar on
299     ↪ %s" % first_token.pretty() )
300
301     else:
302         self.defined_includes[include_name] = first_token
303
304     return self.__default__(tree, children)
305
306 def target_language_name_statement(self, tree, children):
307     input_string = children[0]
308     # log(1, 'tree: \n%s', tree)
309     # log(1, 'children: %s', children)
310     # log(1, 'input_string: %s', type(input_string))
311     # log(1, 'input_string: %s', input_string)
312     if self.is_target_language_name_set:
313         self.errors.append("Duplicated target language name defined in your
314     ↪ grammar on %s" % (input_string.chunks[0].token.pretty() ) )
```

```
310
311     self.is_target_language_name_set = True
312     return self.__default__(tree, children)
313
314 def master_scope_name_statement(self, tree, children):
315     input_string = children[0]
316     if self.is_master_scope_name_set:
317         self.errors.append( "Duplicated master scope name defined in your
318         → grammar on %s" % ( input_string.chunks[0].token.pretty() ) )
319
320     self.is_master_scope_name_set = True
321     return self.__default__(tree, children)
322
323 def enter_block(self, tree, children):
324     self.indentation_level += 1
325     token = children[0]
326     self.open_blocks[self.indentation_level] = CommandBlock(
327         self.indentation_level, token.pos_in_stream )
328
329     # log('indentation_level', self.indentation_level, ', children', children)
330     # log( 'indentation_blocks', self.indentation_blocks )
331     raise Discard()
332
333 def leave_block(self, tree, children):
334     token = children[0]
335     command_block = self.open_blocks[self.indentation_level]
336     command_block.close_position = token.pos_in_stream
337     self.indentation_blocks.append( command_block )
338
339     self.indentation_level -= 1
340     # log('indentation_level', self.indentation_level, ', children', children)
341     # log( 'indentation_blocks', self.indentation_blocks )
342     raise Discard()
343
344 def include_statement(self, tree, children):
345     input_string = children[0]
346     include_name = str(input_string)
347
348     self.required_includes[include_name] = input_string
349     return self.__default__(tree, children)
350
351 def constant_definition(self, tree, children):
```

```
350     # log(1, 'tree: \n%s', tree.pretty(debug=1))
351     # log(1, 'children: \n%s', children)
352     #
353     # [
354     #   Tree(constant_name, [Token(CONSTANT_NAME_, '$constant:')]),
355     #   [
356     #     Token(TEXT_CHUNK, ' test'),
357     #     Token(CONSTANT_USAGE_, '$varrrr:'),
358     #     Token(TEXT_CHUNK_END, 'test')
359     #   ]
360   # ]
361   constant_name = children[0]
362   constant_value = children[1]
363
364   constant_definition = ConstantDefinition( constant_name, constant_value )
365   constant_name_str = str( constant_name )
366
367   # log( 'constant_name:', constant_name )
368   # log( 'constant_name_str:', constant_name_str )
369   # log( 'constant_value:', constant_value )
370   # log( 'constant_definition:', constant_definition )
371   # log( 'constant_value.chunks:', constant_value.chunks )
372   if constant_name_str in str(constant_value):
373       constant_value.chunks = [ chunk for chunk in constant_value.chunks
374                               if chunk.name != constant_name_str ]
375
376       # log( 'constant_value.chunks:', constant_value.chunks )
377       self.warnings.append( "Recursive constant definition on %s" % (
378                           constant_name.pretty() ) )
379
380   if constant_name_str in self.constant_definitions:
381       self.errors.append( "Constant redefinition on %s" % (
382                           constant_name.pretty() ) )
383
384   self.constant_definitions[constant_name_str] = constant_definition
385   return constant_definition
386
387   def constant_name(self, tree, children):
388       # log(1, 'tree: \n%s', tree.pretty(debug=1))
389       # log(1, 'children: \n%s', children)
390       token = children[0]
```

```
390     # Trim trailing obligatory white space by the grammar
391     token.value = token.value[:-1]
392
393     return token
394
395 def braced_constant_usage_end(self, tree, children):
396     return self.constant_usage( tree, children )
397
398 def constant_usage_end(self, tree, children):
399     return self.constant_usage( tree, children )
400
401 def constant_usage(self, tree, children):
402     token = children[0]
403     constant_name = str( token )
404
405     undefined_constant = ConstantUsage( token )
406     self.constant_usages[constant_name] = undefined_constant
407
408     # log( 'constant_name:', constant_name )
409     # log( undefined_constant )
410
411     # log(1, 'tree: \n%s', tree.pretty(debug=1))
412     return undefined_constant
413
414 def match_statement(self, tree, children):
415     include_name = children[0]
416     # log('include_name', type(include_name))
417
418     self.pending_match_statements.append( include_name )
419     return self.__default__(tree, children)
420
421 def braced_free_input_string(self, tree, children):
422     return self.free_input_string( tree, children )
423
424 def free_input_string(self, tree, children):
425     # log(1, 'tree: \n%s', tree.pretty(debug=1))
426     # log(1, 'children: \n%s', tree.children)
427     #
428     # Tree
429     # (
430     #   free_input_string,
431     #   [
```

```
432         #     Token(TEXT_CHUNK_END, 'source.sma')
433         # ]
434         #
435         constant_body = children
436         input_string = InputString( constant_body, self.constant_definitions,
437                                     ↳ self.errors, self.indentation_blocks )
438
439         # log( 'constant_body:', constant_body )
440         # log( 'input_string: %s', input_string )
441         return input_string
442
443     def text_chunk(self, tree, children):
444         # log(1, 'tree: \n%s', tree.pretty(debug=1))
445         token = children[0]
446         constant_name = str( token )
447         defined_chunk = ConstantUsage( token )
448         defined_chunk.resolve( constant_name )
449
450         # log( defined_chunk )
451         # log( 'constant_name:', constant_name )
452         return defined_chunk
453
454     def text_chunk_end(self, tree, children):
455         return self.text_chunk(tree, children)
456
457     def braced_text_chunk_end(self, tree, children):
458         return self.text_chunk(tree, children)
459
460     def _check_for_main_rules(self):
461         """
462             Look for missing required main rules on the grammar preamble statement.
463         """
464
465         if not self.is_master_scope_name_set:
466             self.errors.append( "Missing master scope name in your grammar
467                                 ↳ preamble." )
468
469         if not self.is_target_language_name_set:
470             self.errors.append( "Missing target language name in your grammar
471                                 ↳ preamble." )
472
473         # log.newline()
474         for include_name in self.pending_match_statements:
```

```
471         # log('include_name', type(include_name))
472         # log('include_name', include_name)
473
474     try:
475         re.compile(str(include_name))
476
477     except re.error as error:
478         self.errors.append( "Invalid regular expression `\\%s` on match
479                           → statement: %s" % ( include_name, error ) )
480
481     def _check_includes_definitions(self):
482         """
483             Resolve all pending include usages across the tree.
484         """
485
486         # Look for missing required main rules on the grammar preamble statement.
487         for include_name, include_token in self.defined_includes.items():
488             if include_name not in self.required_includes:
489                 self.warnings.append( "Unused include `\\%s` defined in your grammar
490                               → on %s" % ( include_name, include_token.pretty() ) )
491
492         # Look for missing required includes by the `include` statement.
493         for include_name, include_token in self.required_includes.items():
494             if include_name not in self.defined_includes:
495                 self.errors.append( "Missing include `\\%s` defined in your grammar on
496                               → %s" % ( include_name, include_token.chunks[0].token.pretty() )
497                               → )
498
499     def _resolve_constants_definitions(self):
500         """
501             Resolve all pending constant usages across the tree.
502         """
503
504         # Checks for undefined constants usage
505         for name, constant in self.constant_usages.items():
506             if name not in self.constant_definitions:
507                 self.errors.append( "Missing constant `\\%s` defined in your grammar
508                               → on %s" % ( name, constant.token.pretty() ) )
509
510         # Checks for unused constants
511         for name, constant in self.constant_definitions.items():
512             # log(1, 'name %s', name)
513             # log(1, 'constant %s', repr(constant))
```

```
508     # log(1, 'constant %s', type(constant))
509     if name not in self.constant_usages:
510         self.warnings.append( "Unused constant `%s` defined in your grammar
511             → on %s" % ( name, constant.token.pretty() ) )
512
513     revolved_count = 1
514     last_resolution = 0
515     pending = {}
516     resolved_constants = {}
517
518     # work resolving the constants usages on `self.constant_usages` until it
519     # there is no new progress
520     while revolved_count != last_resolution:
521         # log('revolved_count', revolved_count, ', last_resolution',
522         #      → last_resolution)
523         revolved_count = last_resolution
524         just_resolved = []
525
526         # Updates all constants definitions with the constants contents values
527         for name, constant in self.constant_definitions.items():
528             # log( 1, 'Trying to resolve name %s, constant %s', name, constant )
529             if constant.resolve():
530                 # log( 1, 'Resolved constant to %s', constant )
531                 just_resolved.append(name)
532                 resolved_constants[name] = constant
533
534             # When a constant_definitions has inner unresolved constants, it can
535             # only be resolved later
536             for constant in just_resolved:
537                 # log( 1, 'Deleting just resolved %s, value %s', constant,
538                 #          → self.constant_definitions[constant] )
539                 del self.constant_definitions[constant]
540
541
542             # Resolve all pending constants
543             for name, constant in self.constant_usages.items():
544                 # log( 1, 'Trying to resolve pending name %s, constant %s', name,
545                 #          → constant )
546                 if name in resolved_constants:
547                     resolution = resolved_constants[name]
548
549                     if constant.resolve( resolution.str ):
550                         # log( 1, 'Resolved constant to %s', constant )
```

```
544         last_resolution += 1
545
546     # log.newline()
547     # log(1, 'constant_usages %s', self.constant_usages)
548     # log(1, 'resolved_constants %s', resolved_constants)
549
550     # if the resolution count does not reach 0, something went wrong
551     if len( self.constant_definitions ) > 0:
552         self.errors.append( "The following constants could not be resolved:\n"
553                             ↪ `"%s"` % ( self.constant_definitions ) )
554
555     self.constant_definitions.update( resolved_constants )
556
557     def _call_userfunc(self, tree, new_children=None):
558         """
559             Overrides the default behavior of TreeTransformer, to send the whole
560             tree to its children instead of only trees children.
561         """
562
563         # Assumes tree is already transformed
564         children = tree.children if new_children is None else new_children
565         try:
566             f = getattr(self, tree.data)
567         except AttributeError:
568             return self.__default__(tree, children)
569         else:
570             return f(tree, children)
```

APÊNDICE F – CÓDIGO DE ADIÇÃO DE CORES

Neste capítulo, pode-se ver na íntegra o arquivo do formatador de código-fonte (Código 36), que faz uso da metagramática no Código 38. No Código 27, pode-se encontrar um exemplo explícito de utilização do Código 36.

Código 36 – Arquivo “source/code_highlighter.py”

```
1 import re
2 import pprint
3 import pushdown
4 import dominate
5 import debug_tools
6
7 from pushdown import Tree
8 from collections import OrderedDict
9
10 from debug_tools import getLogger
11 log = getLogger(1, __name__, time=0, tick=0, msecs=0)
12
13 def escape_html(input_text):
14     return dominate.util.escape( input_text, quote=False ).replace("\n", "<br /> ")
15
16 def get_div_doc(input_text):
17     return '<span grammar_scope="none" theme_scope="none">%s</span>' % escape_html(
18         input_text )
19
20 def get_font_doc(input_text, color, grammar_scope, theme_scope):
21     return '<font color="%s" grammar_scope="%s" theme_scope="%s">%s</font>' % (
22         color, grammar_scope, theme_scope, escape_html( input_text ))
23
24 def get_html_header(title):
25     return debug_tools.utilities.wrap_text(
26         r"""
27             <!DOCTYPE html><html><head><title>%s</title></head>
28             <body style="white-space: pre; font-family: monospace;">
29             """ % ( title )
30     )
31
32 def get_html_footer():
33     return debug_tools.utilities.wrap_text(
34         r""""
```

```
35         </body></html>
36     """
37 )
38
39
40 class ParsedProgram(object):
41     """
42         Represents a program as chunks of data as (text_chunk_start_position,
43         text_chunk).
44     """
45
46     def __init__(self, program, theme):
47         super().__init__()
48         self.initial_size = len( program )
49         self.program = program
50         self.theme = OrderedDict( sorted( theme.items(), key=lambda item: len( str(
51             item ) ) ) )
52
53         self.new_program = []
54         self.cached_new_program = []
55         log( 4, "program %s: `%s`", len( str( self.program ) ), self.program )
56
57     def __str__(self):
58         """
59             Returns the current version of the program,
60             after being cut by add_match().
61         """
62
63         return self.program
64
65     __repr__ = __str__
66
67     def get_new_program(self):
68         """
69             Sorts the list of (text_chunk_start_position, text_chunk) accordingly to
70             `text_chunk_start_position` and return the new program as full string.
71         """
72
73         if self.cached_new_program: return self.cached_new_program
74
75         fixed_program = sorted( self.new_program, key=lambda item: item[0] )
76         fixed_program_len = len(fixed_program)
77         assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
78             self.initial_size, len(self.program) )
```

```
75
76     # Copy the unmatched chunks of text into the final program on
    ↪   self.new_program
77     for index in range( 0, fixed_program_len ):
78
79         if index < fixed_program_len - 1:
80             current_chunk = fixed_program[index]
81             next_chunk = fixed_program[index+1]
82
83             if current_chunk[1] < next_chunk[0]:
84                 doc = get_div_doc( self.program[current_chunk[1]:next_chunk[0]] )
85                 self.new_program.append( ( current_chunk[1], next_chunk[0], doc
    ↪   ) )
86
87         else:
88             current_chunk = fixed_program[index]
89
90             if current_chunk[1] < len( self.program ):
91                 doc = get_div_doc( self.program[current_chunk[1]:] )
92                 self.new_program.append( ( current_chunk[1], len( self.program
    ↪   ), doc ) )
93
94     # At the end of the process, we must to preserve the program size on
    ↪   self.program
95     # for the correct merge with the text chunks processed
96     assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
    ↪   self.initial_size, len(self.program) )
97     fixed_program = sorted( self.new_program, key=lambda item: item[0] )
98     fixed_program_len = len(fixed_program)
99
100    # for index in range( 0, fixed_program_len - 1 ):
101        #     current_chunk = fixed_program[index]
102        #     next_chunk = fixed_program[index+1]
103
104        #     if current_chunk[1] > next_chunk[0]:
105        #         fixed_program[index], fixed_program[index+1] =
    ↪   fixed_program[index+1], fixed_program[index]
106
107    log( 4, "fixed_program:\n%s", pprint.pformat( [ ( item[0], item[1],
    ↪   str(item[2]) ) for item in fixed_program ], indent=2, width=200 ) )
108    log( 4, "cached_new_program: %s", self.cached_new_program )
109
```

```
110     self.cached_new_program = [ item[2] for item in fixed_program ]
111     return self.cached_new_program
112
113     def get_theme(self, scope_name):
114         program_scopes = scope_name.split( '.' )
115
116         def select():
117             for index in range( len(program_scopes), 0, -1 ):
118                 program_scope = ".".join( [program_scopes[inner_index] for
119                     inner_index in range( 0, index )] )
120
121                 for theme_scopes, theme_color in self.theme.items():
122                     theme_scopes = theme_scopes.split( '.' )
123
124                     for index in range( 1, len(theme_scopes) + 1 ):
125                         theme_scope = ".".join( [theme_scopes[inner_index] for
126                             inner_index in range( 0, index )] )
127                         # log( 4, "theme_color: %s, comparing `%s` with `%s`",
128                             theme_color, program_scope, theme_scope )
129
130                         if theme_scope == program_scope:
131                             # log( 4, "Selecting %s with %s", theme_scope,
132                                 theme_color )
133                             return theme_color, theme_scope
134
135             return "", ""
136
137         first_matched_color, theme_scope = select()
138
139         # log( 4, "first_matched_color: %s", first_matched_color )
140         return first_matched_color, theme_scope
141
142     def add_match(self, scope_name, last_match_stack, match):
143         log( 4, "add_match: %s", match )
144         last_match_stack[-1].append(match)
145         match_start = match.start(0)
146         match_end = match.end(0)
147         match_start, match_end = match_start, match_end
148
149         self.program = self.program[:match_start] + "$" * ( match_end - match_start
150             ) + self.program[match_end:]
```

```
146     assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
147         self.initial_size, len(self.program) )
148
149     self._generate_chunk_html( scope_name, match[0], match_start, match_end )
150
151     def add_meta_scope(self, scope_name, last_matches, match,
152         ignore_last_match=False):
153         """ ignore_last_match is set when the last_match value was already scoped,
154             and as we only support scoping text one time, we have to ignore the
155             last_match """
156
157         log( 4, "add_meta_scope: %s", match )
158         last_match = last_matches.pop() if last_matches else None
159         match_end = match.end(0)
160
161         if last_match:
162             log( 4, "match.start: %s, match.end: %s", match.start(0), match.end(0) )
163             log( 4, "last_match.start: %s, last_match.end: %s", last_match.start(0),
164                 last_match.end(0) )
165
166             # match.start: 134, match.end: 135
167             # last_match.start: 113, last_match.end: 114
168             if ignore_last_match and match.start(0) > last_match.start(0):
169                 match_end = match.start(0)
170
171             match_start = last_match.end(0)
172             match_start, match_end = match_start, match_end
173
174             self._generate_chunk_html( scope_name,
175                 self.program[match_start:match_end], match_start, match_end )
176             self.program = self.program[:match_start] + "$" * ( match_end -
177                 match_start ) + self.program[match_end:]
178
179     else:
180         match_start = match.start(0)
181         match_start, match_end = match_start, match_end
182
183         self._generate_chunk_html( scope_name,
184             self.program[match_start:match_end], match_start, match_end )
185         self.program = self.program[:match_start] + "$" * ( match_end -
186             match_start ) + self.program[match_end:]
```

```
179     assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
180         self.initial_size, len(self.program) )
181
182     def _generate_chunk_html(self, grammar_scope, matched_text, match_start,
183         ↵ match_end):
184         first_matched_color, theme_scope = self.get_theme(grammar_scope)
185         doc = get_font_doc( matched_text, first_matched_color, grammar_scope,
186             ↵ theme_scope )
187
188         self.new_program.append( ( match_start, match_end, doc ) )
189         log( 4, "formatted_text: %s", doc )
190         log( 4, "program %s: `%"s`", len( str( self.program ) ), self.program ) )
191
192     class Backend(pushdown.Interpreter):
193
194         def __init__(self, tree, program, theme):
195             super().__init__()
196             self.tree = tree
197             self.program = ParsedProgram( program, theme )
198
199             ## A list of lists, where each list saves all the matches performed by
200             ## the last match_statement on scope_name_statement
201             self.last_match_stack = []
202
203             ## This is set to False every push statement, and set to True, after
204             ## every match statement. This way we can know whether there is a match
205             ## statement after a push statement.
206             self.is_there_push_after_match = False
207             self.is_there_scope_after_match = False
208
209             self.cached_includes = {}
210             self.cache_includes( tree )
211
212             self.visit( tree )
213             log( 4, "Tree: \n%"s, tree.pretty( debug=0 ) )
214
215             def cache_includes(self, tree):
216                 log( 4, "tree.data: ", tree.data )
217
218                 for child in tree.children:
```

```
218     if isinstance(child, Tree) and child.data ==  
219         "miscellaneous_language_rules":  
220             include_name = child.children[0]  
221             include_tree = child.children[1]  
222  
223             log( 4, "include_name: ", include_name )  
224             self.cached_includes[include_name] = include_tree  
225  
226     def target_language_name_statement(self, tree):  
227         target_language_name = tree.children[0]  
228  
229         self.target_language_name = target_language_name  
230         log( 4, "target_language_name: %s", target_language_name )  
231  
232     def master_scope_name_statement(self, tree):  
233         master_scope_name = tree.children[0]  
234  
235         self.master_scope_name = master_scope_name  
236         log( 4, "master_scope_name: %s", master_scope_name )  
237  
238     def include_statement(self, tree):  
239         include_statement = str( tree.children[0] )  
240  
241         log( 4, "include_statement: %s", include_statement )  
242         self.visit_children( self.cached_includes[include_statement] )  
243  
244     def miscellaneous_language_rules(self, tree):  
245         miscellaneous_language_rules = tree.children[0]  
246         log( 4, "miscellaneous_language_rules: %s", miscellaneous_language_rules )  
247  
248     ## match_statement -> scope_name_statement -> push_statement ->  
249     ##                                         match_statement ->  
250     ##                                         pop_statement  
251     ##                                         match_statement ->  
252     ##                                         scope_name_statement -> pop_statement  
253     ##                                         match_statement ->  
254     ##                                         meta_scope_statement -> match_statement -> pop_statement  
255  
256     def match_statement(self, tree):  
257         log( 4, "is_there_push_after_match: %s", self.is_there_push_after_match )  
258  
259         ## Discards the last_match_stack because we do not need the stack when  
260         ## there are 2 matches consecutives without a new push
```

```
256     if not self.is_there_push_after_match and self.last_match_stack:
257         self.last_match_stack.pop()
258
259         self.is_there_push_after_match = False
260         self.is_there_scope_after_match = False
261
262         match = tree.children[0]
263         self.match = re.compile( str(match) )
264
265         log( 4, "match: %s", self.match.pattern )
266         # log( 4, "tree: %s", tree )
267         self.visit_children( tree )
268
269     def push_statement(self, tree):
270         self.is_there_push_after_match = True
271         log( 4, "push_stack: %s", self.last_match_stack )
272
273         if not self.is_there_scope_after_match:
274             self.last_match_stack.append( [ match for match in self.match.finditer(
275                 str(self.program) ) ] )
276
277         # log( 4, "tree: %s", tree )
278         self.visit_children( tree )
279
280     def meta_scope_statement(self, tree):
281         meta_scope = tree.children[0]
282         self.meta_scope = meta_scope
283
284         log( 4, "pattern: %s", self.match.pattern )
285         log( 4, "meta_scope: %s", meta_scope )
286
287     def pop_statement(self, tree):
288         """ Used the saved self.meta_scope and self.last_match_stack to process the
289             input program. """
290
291         log( 4, "pop_statement: %s", self.meta_scope )
292         log( 4, "last_match_stack: %s", self.last_match_stack )
293         log( 4, "is_there_push_after_match: %s", self.is_there_push_after_match )
294         log( 4, "is_there_scope_after_match: %s", self.is_there_scope_after_match )
295
296         # When there is a scope_name_statement after a push_statement, it means that
297         # the self.match regular expression was already evaluated, therefore, we
298         # must to reuse the cached result on the top of the stack and ignore the
299         # penultimate matches positions
300
301         if not self.is_there_push_after_match and self.is_there_scope_after_match:
302             last_matches = self.last_match_stack.pop()
```

```
296     matches = self.last_match_stack.pop()
297     reversed_matches = list( reversed( matches ) )
298
299     for last_match in last_matches:
300         self.program.add_meta_scope( str(self.meta_scope), reversed_matches,
301             ↪ last_match, True )
302
303 else:
304     last_matches = self.last_match_stack.pop()
305     reversed_last_matches = list( reversed( last_matches ) )
306
307     for last_match in last_matches:
308         match = self.match.search( str( self.program ), last_match.end(0) )
309         # log( "match: '%s'", match )
310         # log( "pattern: '%s' last_match.end", self.match.pattern,
311             ↪ last_match.end(0) )
312
313     if match is None:
314         log( 1, "Error: Could not find the pop end statement! Skipping
315             ↪ highlighting... match '%s' pattern '%s' end '%s'",
316             match, self.match.pattern, last_match.end(0) )
317
318     match = last_match
319
320     self.program.add_meta_scope( str(self.meta_scope),
321             ↪ reversed_last_matches, match )
322
323 def scope_name_statement(self, tree):
324     """ Used the saved self.match to process the input program. """
325     self.is_there_scope_after_match = True
326     scope_name = tree.children[0]
327     self.scope_name = scope_name
328
329     log( 4, "pattern: %s", self.match.pattern )
330     log( 4, "scope_name: %s", scope_name )
331     self.last_match_stack.append([])
332     self.match.sub( lambda match: self.program.add_match( str(scope_name),
333                     self.last_match_stack, match ), str( self.program ) )
334
335 def generated_html(self):
336     log( 4, "...")
```

```
333     document = [ get_html_header( "%s - %s" % ( self.target_language_name,
334                                     ↪ self.master_scope_name ) ) ]
335
335     for item in self.program.get_new_program():
336         document.append( str(item) )
337
338     document.append( get_html_footer() )
339
339     return "".join( document )
```

APÊNDICE G – CÓDIGO DO FORMATADOR

Neste capítulo, pode-se ver na íntegra o arquivo do formatador de código-fonte (Código 37) que faz uso da metagramática no Código 38. No Código 26, pode-se encontrar um exemplo explícito de utilização do Código 37.

Código 37 – Arquivo “source/code_formatter.py”

```
1 import re
2 import pprint
3 import pushdown
4 import dominate
5 import debug_tools
6
7 from pushdown import Tree
8 from collections import OrderedDict
9
10 from debug_tools import getLogger
11 log = getLogger(1, __name__, time=0, tick=0, msecs=0)
12
13 def escape_html(input_text):
14     return dominate.util.escape( input_text, quote=False ).replace("\n", "<br />" )
15
16 def get_div_doc(original_program):
17     return '<span grammar_scope="none" setting_scope="none">%s</span>' %
18         escape_html( original_program )
19
20 def get_font_doc(original_program, formatted_text, setting, grammar_scope,
21     ↪ setting_scope):
22     return '<span setting="%s" grammar_scope="%s" setting_scope="%s"
23             ↪ original_program="%s">%s</span>' %
24                 setting, grammar_scope, setting_scope, original_program, escape_html(
25                     ↪ formatted_text )
26
27     )
28
29 def get_html_header(title):
30     return debug_tools.utilities.wrap_text(
31         r"""
32             <!DOCTYPE html><html><head><title>%s</title></head>
33             <body style="white-space: pre; font-family: monospace;">
34             """ % ( title )
35     )
```

```
32 def get_html_footer():
33     return debug_tools.utilities.wrap_text(
34         r"""
35             </body></html>
36         """
37     )
38
39
40 class AbstractFormatter(object):
41     """
42         Represents a program as chunks of data as (text_chunk_start_position,
43         text_chunk).
44     """
45
46     def __init__(self, program, settings):
47         super().__init__()
48         self.initial_size = len( program )
49         self.program = program
50
51         ## No need to sort the settings other than always having the some logs output
52         self.settings = OrderedDict( sorted( settings.items(), key=lambda item:
53             item ) )
54
55         self.new_program = []
56         self.cached_new_program = []
57         log( 4, "program %s: `%s`", len( str( self.program ) ), self.program )
58
59     def __str__(self):
60         """
61             Returns the current version of the program,
62             after being cut by add_match().
63         """
64
65         return self.program
66
67     __repr__ = __str__
68
69     def get_new_program(self):
70         """
71             Sorts the list of (text_chunk_start_position, text_chunk) accordingly to
72             `text_chunk_start_position` and return the new program as full string.
73         """
74
75         if self.cached_new_program: return self.cached_new_program
```

```
73
74     fixed_program = sorted( self.new_program, key=lambda item: item[0] )
75     fixed_program_len = len(fixed_program)
76     assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
77         self.initial_size, len(self.program) )
78
79     # Copy the unmatched chunks of text into the final program on
80     # → self.new_program
81     for index in range( 0, fixed_program_len ):
82
83         if index < fixed_program_len - 1:
84             current_chunk = fixed_program[index]
85             next_chunk = fixed_program[index+1]
86
87             if current_chunk[1] < next_chunk[0]:
88                 doc = get_div_doc( self.program[current_chunk[1]:next_chunk[0]] )
89                 self.new_program.append( ( current_chunk[1], next_chunk[0], doc
90                     ) )
91
92             else:
93                 current_chunk = fixed_program[index]
94
95             if current_chunk[1] < len( self.program ):
96                 doc = get_div_doc( self.program[current_chunk[1]:] )
97                 self.new_program.append( ( current_chunk[1], len( self.program
98                     ), doc ) )
99
100            # At the end of the process, we must to preserve the program size on
101            # → self.program
102            # for the correct merge with the text chunks processed
103            assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
104                self.initial_size, len(self.program) )
105            fixed_program = sorted( self.new_program, key=lambda item: item[0] )
106            fixed_program_len = len(fixed_program)
107
108            log( 4, "fixed_program:\n%s", pprint.pformat( [ ( item[0], item[1],
109                str(item[2]) ) for item in fixed_program ], indent=2, width=200 ) )
110            log( 4, "cached_new_program: %s", self.cached_new_program )
111
112            self.cached_new_program = [ item[2] for item in fixed_program ]
113        return self.cached_new_program
114
```

```
108     def get_theme(self, scope_name):
109
110         def select():
111
112             for setting_scopes, setting_value in self.settings.items():
113                 setting_scopes = setting_scopes.split( '.' )
114
115                 for index in range( 1, len(setting_scopes) + 1 ):
116                     setting_scope = ".".join( [ setting_scopes[inner_index] for
117                                     ↪ inner_index in range( 0, index ) ] )
118                     log( 8, "setting_value: %s, comparing `%s` with `%s`",
119                         ↪ setting_value, scope_name, setting_scope )
120
121                     if setting_scope == scope_name:
122                         log( 8, "Selecting %s with %s", setting_scope, setting_value )
123                         return setting_value, setting_scope
124
125
126             return "", ""
127
128         matched_setting, setting_scope = select()
129
130         log( 8, "matched_setting: '%s' (on %s)", matched_setting, setting_scope )
131         return matched_setting, setting_scope
132
133     def add_match(self, scope_name, last_match_stack, match):
134
135         log( 4, "add_match: %s", match )
136         last_match_stack[-1].append(match)
137         match_start = match.start(0)
138         match_end = match.end(0)
139         match_start, match_end = match_start, match_end
140
141         self.program = self.program[:match_start] + "$" * ( match_end - match_start
142                                     ↪ ) + self.program[match_end:]
143         assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
144                                     ↪ self.initial_size, len(self.program) )
145
146         self._generate_chunk_html( scope_name, match[0], match_start, match_end )
147
148     def add_meta_scope(self, scope_name, last_matches, match,
149                      ↪ ignore_last_match=False):
150         """ ignore_last_match is set when the last_match value was already scoped,
```

```
144         and as we only support scoping text one time, we have to ignore the
145         ↵ last_match """
146         log( 4, "add_meta_scope: %s", match )
147         last_match = last_matches.pop() if last_matches else None
148         match_end = match.end(0)
149
150         if last_match:
151             log( 4, "match.start: %s, match.end: %s", match.start(0), match.end(0) )
152             log( 4, "last_match.start: %s, last_match.end: %s", last_match.start(0),
153                 ↵ last_match.end(0) )
154
155             # match.start: 134, match.end: 135
156             # last_match.start: 113, last_match.end: 114
157             if ignore_last_match and match.start(0) > last_match.start(0):
158                 match_end = match.start(0)
159
160             match_start = last_match.end(0)
161             match_start, match_end = match_start, match_end
162
163             self._generate_chunk_html( scope_name,
164                 ↵ self.program[match_start:match_end], match_start, match_end )
165             self.program = self.program[:match_start] + "$" * ( match_end -
166                 ↵ match_start ) + self.program[match_end:]
167
168         else:
169             match_start = match.start(0)
170             match_start, match_end = match_start, match_end
171
172             self._generate_chunk_html( scope_name,
173                 ↵ self.program[match_start:match_end], match_start, match_end )
174             self.program = self.program[:match_start] + "$" * ( match_end -
175                 ↵ match_start ) + self.program[match_end:]
176
177             assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
178                 ↵ self.initial_size, len(self.program) )
179
180             def _generate_chunk_html(self, grammar_scope, matched_text, match_start,
181                 ↵ match_end):
182                 matched_setting, setting_scope = self.get_theme(grammar_scope)
183
184                 if setting_scope:
185                     formatted_text = self.format_text( matched_text, matched_setting )
```

```
178     html_fragment = get_font_doc( matched_text, formatted_text,
179         ↪  matched_setting, grammar_scope, setting_scope )
180
180 else:
181     html_fragment = get_font_doc( matched_text, matched_text, "unformatted",
182         ↪  grammar_scope, setting_scope )
183
183     self.new_program.append( ( match_start, match_end, html_fragment ) )
184     log( 4, "formatted_text: %s", html_fragment )
185     log( 4, "program %s: `%" , len( str( self.program ) ), self.program )
186
187 def format_text(matched_text, matched_setting):
188     return matched_text
189
190
191 class SingleSpaceFormatter(AbstractFormatter):
192
193     def format_text(self, code_to_format, setting_value):
194         code_to_format = code_to_format.strip( " " )
195
196         if setting_value:
197             return " " * setting_value + code_to_format + " " * setting_value
198         else:
199             return code_to_format
200
201
202 class Backend(pushdown.Interpreter):
203
204     def __init__(self, formatter, tree, program, settings):
205         super().__init__()
206         self.tree = tree
207         self.program = formatter( program, settings )
208
209         ## A list of lists, where each list saves all the matches performed by
210         ## the last match_statement on scope_name_statement
211         self.last_match_stack = []
212
213         ## This is set to False every push statement, and set to True, after
214         ## every match statement. This way we can know whether there is a match
215         ## statement after a push statement.
216         self.is_there_push_after_match = False
217         self.is_there_scope_after_match = False
```

```
218
219     self.cached_includes = {}
220     self.cache_includes( tree )
221
222     self.visit( tree )
223     log( 4, "Tree: \n%s", tree.pretty( debug=0 ) )
224
225 def cache_includes(self, tree):
226     log( 4, "tree.data: ", tree.data )
227
228     for child in tree.children:
229
230         if isinstance(child, Tree) and child.data ==
231             "miscellaneous_language_rules":
232             include_name = child.children[0]
233             include_tree = child.children[1]
234
235             log( 4, "include_name: ", include_name )
236             self.cached_includes[include_name] = include_tree
237
238 def target_language_name_statement(self, tree):
239     target_language_name = tree.children[0]
240
241     self.target_language_name = target_language_name
242     log( 4, "target_language_name: %s", target_language_name )
243
244 def master_scope_name_statement(self, tree):
245     master_scope_name = tree.children[0]
246
247     self.master_scope_name = master_scope_name
248     log( 4, "master_scope_name: %s", master_scope_name )
249
250 def include_statement(self, tree):
251     include_statement = str( tree.children[0] )
252
253     log( 4, "include_statement: %s", include_statement )
254     self.visit_children( self.cached_includes[include_statement] )
255
256 def miscellaneous_language_rules(self, tree):
257     miscellaneous_language_rules = tree.children[0]
258     log( 4, "miscellaneous_language_rules: %s", miscellaneous_language_rules )
```

```
259     ## match_statement -> scope_name_statement -> push_statement ->
260     ##
261     ##                                     match_statement ->
262     ##                                         → pop_statement
263     ##                                     match_statement ->
264     ##                                         → scope_name_statement -> pop_statement
265     ##
266     ##                                         → meta_scope_statement -> match_statement -> pop_statement
267
268 def match_statement(self, tree):
269     log( 4, "is_there_push_after_match: %s", self.is_there_push_after_match )
270
271     ## Discards the last_match_stack because we do not need the stack when
272     ## there are 2 matches consecutives without a new push
273     if not self.is_there_push_after_match and self.last_match_stack:
274         self.last_match_stack.pop()
275
276         self.is_there_push_after_match = False
277         self.is_there_scope_after_match = False
278
279         match = tree.children[0]
280         self.match = re.compile( str(match) )
281
282         log( 4, "match: %s", self.match.pattern )
283         # log( 4, "tree: %s", tree )
284         self.visit_children( tree )
285
286
287 def push_statement(self, tree):
288     self.is_there_push_after_match = True
289     log( 4, "push_stack: %s", self.last_match_stack )
290
291     if not self.is_there_scope_after_match:
292         self.last_match_stack.append( [ match for match in self.match.finditer(
293             → str(self.program) ) ] )
294
295         # log( 4, "tree: %s", tree )
296         self.visit_children( tree )
297
298
299 def meta_scope_statement(self, tree):
300     meta_scope = tree.children[0]
301     self.meta_scope = meta_scope
302     log( 4, "pattern: %s", self.match.pattern )
303     log( 4, "meta_scope: %s", meta_scope )
304
305
306 def pop_statement(self, tree):
```

```
296     """ Used the saved self.meta_scope and self.last_match_stack to process the
297     ↪ input program. """
298     log( 4, "pop_statement: %s", self.meta_scope )
299     log( 4, "last_match_stack: %s", self.last_match_stack )
300     log( 4, "is_there_push_after_match: %s", self.is_there_push_after_match )
301     log( 4, "is_there_scope_after_match: %s", self.is_there_scope_after_match )
302
303     # When there is a scope_name_statement after a push_statement, it means that
304     # the self.match regular expression was already evaluated, therefore, we
305     # must to reuse the cached result on the top of the stack and ignore the
306     # penultimate matches positions
307     if not self.is_there_push_after_match and self.is_there_scope_after_match:
308         last_matches = self.last_match_stack.pop()
309         matches = self.last_match_stack.pop()
310         reversed_matches = list( reversed( matches ) )
311
312         for last_match in last_matches:
313             self.program.add_meta_scope( str(self.meta_scope), reversed_matches,
314                                         ↪ last_match, True )
315
316     else:
317         last_matches = self.last_match_stack.pop()
318         reversed_last_matches = list( reversed( last_matches ) )
319
320         for last_match in last_matches:
321             match = self.match.search( str( self.program ), last_match.end(0) )
322             # log( "match: '%s'", match )
323             # log( "pattern: '%s' last_match.end", self.match.pattern,
324             ↪ last_match.end(0) )
325
326             if match is None:
327                 log( 1, "Error: Could not find the pop end statement! Skipping
328                     ↪ highlighting... match '%s' pattern '%s' end '%s'",
329                         match, self.match.pattern, last_match.end(0) )
330
331             match = last_match
332
333             self.program.add_meta_scope( str(self.meta_scope),
334                                         ↪ reversed_last_matches, match )
335
336     def scope_name_statement(self, tree):
337         """ Used the saved self.match to process the input program. """
```

```
333     self.is_there_scope_after_match = True
334     scope_name = tree.children[0]
335     self.scope_name = scope_name
336
337     log( 4, "pattern: %s", self.match.pattern )
338     log( 4, "scope_name: %s", scope_name )
339     self.last_match_stack.append([])
340     self.match.sub( lambda match: self.program.add_match( str(scope_name),
341                     self.last_match_stack, match ), str( self.program ) )
342
343 def generated_html(self):
344     log( 4, "..." )
345     document = [ get_html_header( "%s - %s" % ( self.target_language_name,
346         self.master_scope_name ) ) ]
346
347     for item in self.program.get_new_program():
348         document.append( str(item) )
349
350     document.append( get_html_footer() )
351     return "".join( document )
```

APÊNDICE H – CÓDIGO DA METAGRAMÁTICA

Neste capítulo, pode-se ver na íntegra o arquivo da gramática (Código 38) utilizada pela adição de cores e formatação de código-fonte. Nos Códigos 26 e 27, pode-se ver o uso explícito deste arquivo de gramática (na linha referenciando o arquivo "grammars_grammar.pushdown") e seu recebimento pelo Analisador Lark (Seção 4.2: Introdução à Metagramática).

Nesta gramática, podem ser encontradas produções como `enter_block: OPEN_BRACE` que servem para fazer com que a árvore sintática tenha um nó com o nome "enter_block" e com que o analisador léxico gere um *token* com o nome "OPEN_BRACE". Estas características facilitam a manipulação da árvore sintática e operações com *tokens*.

Código 38 – Arquivo “source/grammars_grammar.pushdown”

```
1 language_syntax: _NEWLINE? preamble_statements _NEWLINE? language_construct_rules
2   → _NEWLINE? ( miscellaneous_language_rules _NEWLINE? )* _NEWLINE?
3
4 preamble_statements: ( ( target_language_name_statement
5                         | master_scope_name_statement
6                         | constant_definition ) _NEWLINE )+
7
8 language_construct_rules: "contexts" ":" " indentation_block"
9
10 miscellaneous_language_rules: /[^\n]+/ ":" " indentation_block"
11
12
13 target_language_name_statement: "name" ":" " free_input_string"
14 master_scope_name_statement: "scope" ":" " free_input_string"
15
16 indentation_block: enter_block _NEWLINE ( statements_list _NEWLINE )+ leave_block
17
18 statements_list: match_statement | include_statement | push_statement
19               | pop_statement| constant_definition | scope_name_statement
20               | capturing_block | meta_scope_statement
21
22 enter_block: OPEN_BRACE
23 leave_block: CLOSE_BRACE
24 OPEN_BRACE: "{"
25 CLOSE_BRACE: "}"
26
27 push_statement: "push" ":" " indentation_block"
28 include_statement: "include" ":" " free_input_string"
29 constant_definition: constant_name free_input_string
30 constant_name: CONSTANT_NAME_
```

```
27 CONSTANT_NAME_: /\$.+?\:/ /
28
29 free_input_string: ( constant_usage
30             | text_chunk )* ( constant_usage_end
31             | text_chunk_end )
32
33 constant_usage: CONSTANT_USAGE_
34 text_chunk: TEXT_CHUNK_
35 constant_usage_end: CONSTANT_USAGE_END_
36 text_chunk_end: TEXT_CHUNK_END_
37
38 CONSTANT_USAGE_: /\$[^\\n\\$\\:]+\\:/
39 TEXT_CHUNK_: /(?:\\{||\\}|\\\\$|[^\n{}\\$])+/
40 CONSTANT_USAGE_END_: /(?:\\$[^\\n\\$\\:]+\\:)(?=(:\\n|$))/
41 TEXT_CHUNK_END_: /(?:\\{||\\}|\\\\$|[^\n{}\\$])+(?=(:\\n|$))/
42
43 braced_free_input_string: ( constant_usage
44             | text_chunk )* ( braced_constant_usage_end
45             | braced_text_chunk_end )
46
47 braced_constant_usage_end: BRACED_CONSTANT_USAGE_END_
48 braced_text_chunk_end: BRACED_TEXT_CHUNK_END_
49 BRACED_CONSTANT_USAGE_END_: /(?:\\$[^\\n\\$\\:]+\\:)(?=(: \\u007b))/
50 BRACED_TEXT_CHUNK_END_: /(?:\\{||\\}|\\\\$|[^\n{}\\$])+(?=(: \\u007b))/
51
52 match_statement: "match" ":" braced_free_input_string "{" ( _NEWLINE
    ↳ statements_list )* _NEWLINE "}"
53 scope_name_statement: "scope" ":" free_input_string
54
55 capturing_block: "captures" ":" "{" ( _NEWLINE capturing_lines )+ _NEWLINE "}"
56 capturing_lines: INTEGER+ ":" free_input_string
57
58 pop_statement: "pop" ":" free_input_string
59 meta_scope_statement: "meta_scope" ":" free_input_string
60
61 // General terminal tokens
62 CR: "/r"
63 LF: "/n"
64 SPACES: /[\\t \\f]+/
65
66 SINGLE_LINE_COMMENT: /(?:\\#|\\/\\/)[^\\n]*/
67 MULTI_LINE_COMMENT: /\\/\\*(?:[\\s\\S]*?)\\*\\/
```

```
68
69 NEWLINE: (CR? LF)+  
70 _NEWLINE: ( /\r?\n[\t ]*/ | SINGLE_LINE_COMMENT )+  
71 INTEGER: "0".."9"  
72  
73 %ignore SPACES  
74 %ignore MULTI_LINE_COMMENT  
75 %ignore SINGLE_LINE_COMMENT  
76  
77 %declare _INDENT _DEDENT  
78 // _INDENT: " "  
79 // _DEDENT: " "
```

APÊNDICE I – ARTIGO SOBRE O TCC

Uma Ferramenta de Formatação Programável Por Gramáticas

Evandro Coan¹, Rafael de Santiago¹

¹Centro Tecnológico, Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brazil
evandrocoan@hotmail.com, r.santiago@ufsc.br

Abstract. Propõe-se uma ferramenta que permita, por meio de gramáticas, a especificação de quais linguagens de programação deseja-se realizar a formatação. Utilizando um analisador já existente, foi desenvolvido uma metagramática utilizando o Analisador Lark e então construído um analisador semântico para a nova metalínguagem. Por fim, dois protótipos de ferramentas foram desenvolvidos sobre a nova metalínguagem. Um formatador de código-fonte e uma ferramenta de adição de cores. A ferramenta de formatação de código-fonte é uma implementação simplificada e no futuro precisará ser completada, para adequar-se propriamente a qualquer processo de formatação de código-fonte.

Resumo. It proposed a new Source Code Formatting Tool, allowing users to input their preferred language grammars. Using the Lark Parser, a metagrammar was developed and then a semantic parser was built for the new metalanguage. Finally, two tool prototypes were developed with the new metalanguage. A source code formatter and a source code highlighter. While the color addition tool can already be considered complete (because the color addition process itself is simple), the source code formatting tool is a simplified implementation and in the future will need to be completed, to suit any source code formatting requirements.

1. Introdução

Ter que aprender e configurar muitas ferramentas de formatação de código-fonte é um tarefa cansativa. Mais ainda, é escrever uma nova ferramenta de formatação de código-fonte para cada nova linguagem de programação que surgir. Um programador da linguagem C++, pode utilizar formatador de código-fonte que oferece até 600 opções de configuração. Mas este programador, ao migrar para uma linguagem como Python, pode somente encontrar um formatador que suporta no máximo 20 opções de configuração. O que pode tornar frustrante migrar de uma linguagem de programação para outra, pela perda de controle da ferramenta de formatação de código-fonte.

Com isso em mente, propõem-se desenvolver um formatador de código-fonte, extensível a diversas linguagens de programação por meio de uma nova definição gramática, especificada por uma metagramática. Analisar a fundamentação teórica de linguagens formais e compiladores. Analisar o estado da arte das ferramentas que fazem formatação de código-fonte, A partir dos pontos fracos e fortes do estado da arte de

formatadores, propor uma nova ferramenta de formatação de código-fonte. E avaliar como esta nova ferramenta difere das demais já existentes.

1.1. Gramáticas

Gramáticas são conjuntos de regras que definem uma linguagem. Na teoria da computação e linguagens formais, uma gramática é definida por quatro componentes (HOPCROFT; MOTWANI; ULLMAN, 2006): 1) O conjunto de símbolos terminais (também chamados de tokens ou símbolos do alfabeto da linguagem). Cada terminal corresponde a um símbolo presente na linguagem; 2) O conjunto V n de símbolos não-terminais (algumas vezes chamados de “variáveis sintáticas”), servem para agrupar vários não-terminais e/ou terminais; 3) Um conjunto de produções P . Uma produção consiste em uma dupla elementos. O primeiro elemento é a cabeça ou lado esquerdo e representa a substituição ou consumo que será feito no programa de entrada. O segundo elemento é a cauda ou lado direito da produção, composto de terminais e/ou não-terminais; 4) Um símbolo inicial selecionado a partir do conjunto de símbolos não-terminais.

1.2. Compiladores e Tradutores

Em linguagens formais, tradutores são ferramentas que operam realizando a transformação de um programa de entrada, em um programa de saída (MURPHREE; FENVES, 1970). Diferente de um compilador, a linguagem de destino da “tradução” é do mesmo nível que a linguagem de origem. Por exemplo, dado um programa de entrada em C++ e um programa de saída em Java, tem-se um processo de tradução (Figura 1). A tradução é diferente de um processo de compilação, que é dotado de mais etapas (AI HUA WU; PAQUET, 2004).

Analisador Sintático é um nome dado para analisadores que recebem como entrada uma Gramática Livre de Contexto que representa os aspectos estruturais de uma linguagem, i.e., sua sintaxe (AHO; LAM et al., 2006). Analisadores Sintáticos possuem muito mais utilidade do que somente checar se a sintaxe do programa de entrada está correta, uma vez que eles também podem gerar a Árvore Sintática do programa 14 que é utilizada para realizar a Análise Semântica e geração de código.

Utilizando a Árvore Sintática do programa de entrada, o tradutor constrói uma nova Árvore Sintática correspondente a Árvore Sintática da linguagem do programa destino, utilizada para construir o código-fonte do programa destino. Em um processo de compilação, não é necessário criar uma nova Árvore Sintática como no processo de tradução, mas sim a geração de código objeto ou binário (AHO; LAM et al., 2006).

2. Formatadores de Código

Conhecido como “pretty-printing” ou embelezadores 1 (DE JONGE, 2002), uma ferramenta de formatação pode ser complicada de se utilizar, somente com um conjunto básico de definições. Por exemplo, para permitir um melhor controle do usuário, a ferramenta de formatação pode permitir que exista uma configuração específica para cada aspecto da linguagem.

2.1. Qual a utilidade de Formatadores de Código?

Pode-se compreender uma frase sem pontuação, mas é mais fácil fazer essa operação utilizando a pontuação. Por exemplo:

umaboapontuacaocomcertezatornaascoisasmaisfaceisdeseler

Um leitor hábil pode compreender facilmente a oração acima, mas a pontuação tornaria a tarefa mais fácil (WICKHAM, 2017).

Não pôde-se encontrar nenhuma forte evidência ou relação sobre a compreensão de códigos-fonte e formatadores de código. Alguns estudos como Scalabrino et al. (2016), implementam modelos de Inteligência Artificial utilizados para classificar códigos-fonte como “bem legíveis” ou “mal legíveis”. Inicialmente estes estudos começam coletando dados de pessoas classificando códigos-fonte como legíveis ou não, para então treinar a Inteligência Artificial para classificar códigos-fonte.

No fim, como não se pode dizer certamente que ao utilizar ferramentas de formatação de código-fonte se irá ter um melhor desempenho, cabe a cada desenvolvedor ou time de desenvolvimento decidir por sua experiência, se existe a necessidade de utilizar uma ferramenta de formatação de código-fonte.

2.2. Ofuscadores

Ofuscadores são formatadores de código-fonte que funcionam com objetivo contrário ao dos formatadores. Em vez de melhorar a leitura do código-fonte, sua função é impossibilitar que o código-fonte seja compreendido ou eliminar caracteres desnecessários do código-fonte. Tais técnicas e utilidades para estes tipos de software podem ser encontradas com mais detalhes em Ceccato et al. (2008).

2.3. Adição de Cores

Ao pesquisar sobre formatadores de texto com os recém-apresentados, é comum encontrar trabalhos conhecidos como Pretty-Printing ou Source Code Highlighters, que fazem a adição de cores à código-fonte para serem melhores visualizados: 1) fazer maior destaque aos elementos mais importantes do código-fonte; 2) fazer menor destaque aos elementos considerados menos importantes; 3) fazer que elementos relacionados ou iguais possam ser facilmente encontrados, fazendo com que eles tenham a mesma cor, ou uma cor muito similar (além de estilos como negrito, itálico e sublinhado).

Editores de texto como Skinner (2015b), permitem que seus usuários escrevam as gramáticas das linguagens nas quais se quer editar seu código-fonte, dentro do editor com suporte a adição de cores ao elementos do texto. Este processo é separado por dois “arquivos de configuração”: 1) o arquivo “.sublime-syntax” que contém propriamente a gramática da linguagem; 2) o arquivo “.sublime-color-scheme” que contém as configurações de estilo para serem aplicados à uma ou mais gramáticas de diversas linguagens. Seguindo convenções de na escrita dos arquivos “.sublime-syntax” (SKINNER, 2015a) e “.sublime-color-scheme” (SKINNER, 2015c), é possível utilizar um único arquivo “.sublime-color-scheme” com diversas gramáticas diferentes (“.sublime-syntax”). Também é possível utilizar um único arquivo de gramática

(“.sublime-syntax”) com diversas configurações de cores ou estilos diferentes (“.sublime-color-scheme”).

3. Formatador Desenvolvido

Nesta seção, será explicado o funcionamento e implementação de uma nova ferramenta de formatação. A proposta desta nova ferramenta é permitir que usuários possam entrar com a gramática de qualquer linguagem, por meio de uma metagramática para então formatar o código-fonte da linguagem descrita pela gramática.

Para desenvolver este trabalho foi utilizada a linguagem Python (ORTIZ, 2012), porque Python é uma linguagem simples de entender, e permite que se escreva códigos com maior velocidade. Em contra-partida, Python como sendo uma linguagem interpretada, apresentada menor eficiência do que linguagens compiladas como C++. Entretanto, não é objetivo deste trabalho ter eficiência em tempo de execução. Somente apresentar uma prototipação rápida de algoritmos para uma nova proposta de formatadores de código-fonte.

Na Figura 1, é apresentado um programa completo e funcional na linguagem Java, que imprime “Hello World!” quando chamado sem nenhum argumento de linha de comando (CLI, (SINGER, 2017)). Quando este mesmo programa Java é chamado com qualquer número de argumentos pela linha de comando, ele imprime “Bye World!” na saída padrão.

```
1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         if(args.length > 0)
6         {
7             System.out.println("Bye World!");
8         }
9     else
10    {
11        System.out.println("Hello World!");
12    }
13 }
14 }
```

Figura 1. Exemplo de um programa na linguagem Java

Tendo como o exemplo o código-fonte em Java apresentado na Figura 1, irá ser introduzido o algoritmo de formatação proposto de forma “simples”, capaz que pegar algum elemento com uma estrutura como “`if(args.length > 0)`”, de um programa de entrada em uma linguagem qualquer e realizar a sua formatação. Na Figura 2, inicia-se com a apresentação de uma simplificação da metagramática utilizada neste trabalho.

Na Figura 3, é apresentado um exemplo de gramática válida definida pelas regras da metagramática apresentada na Figura 2. Com a gramática da Figura 3 é possível reconhecer qualquer programa em que exista a estrutura “`if`” com a seguinte forma “`if(alguma coisa)`” ou somente “`if()`”. Uma vez que esses “`if`’s” sejam

reconhecidos, eles serão atribuídos ao escopo “if.statement.body”, que representa a região do código-fonte onde encontra-se o conteúdo do “if”. No exemplo da Figura 1 o escopo “if.statement.body” seria equivalente ao trecho de código-fonte “args.length > 0”.

```

1 language_syntax: _NEWLINE? ( match_statement | scope_name_statement )+ _NEWLINE?
2
3 match_statement: "match" ":" /.+/_NEWLINE
4 scope_name_statement: "scope" ":" /.+/_NEWLINE
5
6 CR: "/r"
7 LF: "/n"
8
9 _NEWLINE: ( /\r?\n[\t ]*/ )+
10 SPACES: /[\t \f]+/
11
12 %ignore SPACES

```

Figura 2. Exemplo mínimo da metagramática

Assumindo que neste ponto, o programa (gramática) já está semanticamente validado, utiliza-se a Árvore Sintática Abstrata da gramática da linguagem do código-fonte (Figura 1) para construir algum algoritmo ou estratégia que possa ser utilizada para realizar a formatação do código-fonte inicialmente apresentado na Figura 1.

```

1 match: (?<=if\().*(?=\
2 scope: if.statement.body

```

Figura 3. Exemplo de gramática pelas regras da mínima metagramática

Segundo a mesma estratégia utilizada por editores de texto em suas gramáticas (Seção 2.3: Adição de Cores), realizar-se o consumo do programa de entrada, removendo dele as estruturas descritas pela gramática na Figura 3, até que não se possa mais remover nenhum elemento novo, terminando a reconhecimento do programa de entrada pela gramática utilizada.

Ao fazer o processo de reconhecimento do programa de entrada, remove-se os caracteres reconhecidos, substituindo eles por algum outro caractere de marcação (como “§”, símbolo de seção) para que não altere-se o tamanho original do programa de entrada. Na Figura 4 se pode ver como o programa original (Figura 1) ficou ao final do processo de reconhecimento pela gramática na Figura 3. Essa foi uma estratégia utilizada para se possa no final do processo de formatação se possa facilmente reintroduzir no programa os novos trechos de código-fonte que foram formatados.

Como pode ser percebido, a gramática de entrada na Figura 3 não consome todo o programa de entrada (Figura 1) no final do processo de reconhecimento (Figura 4). Esta é uma característica importante dos formatadores de código deste trabalho. Todo texto que não é consumido, ou pela gramática de entrada, ou pelo formatador de código ou adição de cores, será mantido intacto no final do processo de formatação ou adição

de cores. Assim, pode-se ter o formatador de código já em funcionamento com gramática mais simples possível, ou que já atenda as características mínimas que se deseja formatar ou adicionar de cores.

```
1  {
2      public static void main(String[] args)
3      {
4          if($$$$$$$$$$$$$)
5          {
6              System.out.println("Bye World!");
7          }
8          else
9          {
10             System.out.println("Hello World!");
11         }
12     }
13 }
```

Figura 4. Resultado do reconhecimento do programa Java pela gramática

Até este ponto, ainda não se mostrou como a parte mais importante deste trabalho deve acontecer, a formatação de código-fonte. A estratégia deste trabalho foi realizar a formatação de código-fonte ao reconhecer o programa de entrada (Figura 1) “removendo” os caracteres reconhecidos, atribuindo escopos para cada um deles. Uma vez que o trecho de código-fonte “removido” possui um escopo atribuído, um arquivo de configurações JSON como a Figura 5 é consultado. Caso exista uma “configuração” relacionada ao escopo recém-reconhecido, o trecho de código-fonte é enviado para um formatador especializado de código-fonte como a Figura 6.

```
1  {
2      "if.statement.body" : 2,
3  }
```

Figura 5. Exemplo de configuração utilizada mínima do Formatador de Código

A Figura 6 é uma especialização de uma classe maior responsável por navegar pela Árvore Sintática Abstrata das gramáticas das linguagens sendo formatadas. A sua função “format_text” será chamada sempre que um escopo for encontrado pela metagramática. Os valores dos parâmetros “code_to_format” e “setting_value” serão o nome do escopo encontrado pela gramática e o valor da “configuração” correspondente encontrado no arquivo da Figura 5.

```

1 class SingleSpaceFormatter(AbstractFormatter):
2
3     def format_text(self, code_to_format, setting_value):
4         code_to_format = code_to_format.strip( " " )
5
6         if setting_value:
7             return " " * setting_value + code_to_format + " " * setting_value
8         else:
9             return code_to_format

```

Figura 6. Exemplo mínimo de Formatador de Código

Por fim, na Figura 7 encontra-se o resultado da formatador para comparação com o código-fonte original (Figura 1). Como pode-se perceber, esta formatação realizada não foi muito significativa. Entretanto, está-se trabalhando várias simplificações de implementação. Para trabalhos futuros a este é necessário criar maiores abstrações que permitam trabalhar com gramáticas de linguagens utilizando uma pilha de múltiplos contextos, com já feito em editores de texto que serviram de inspiração a este trabalho.

```

1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         if( args.length > 0 )
6         {
7             System.out.println("Bye World!");
8         }
9         else
10        {
11             System.out.println("Hello World!");
12        }
13    }
14 }

```

Figura 7. Resultado do Formatador de Código para Java

4. Conclusão

Neste trabalho foi proposta uma implementação de um algoritmo que trabalha diretamente com a Árvore Sintática da gramática do programa de entrada (Seção 3: Formatador Desenvolvido). Apesar de simples, esta implementação é o primeiro passo para criação de novos formatadores de código-fonte. Servindo de base para criação de novos algoritmos voltados a utilizar Árvore Sintática da gramática da linguagem, ao contrário da Árvore Sintática do programa de entrada.

As ferramentas de formatação de código-fonte em geral, tentam reconstruir toda a Árvore Sintática do programa de entrada a ser formatado (Seção 2: Formatadores de Código). Neste trabalho, é realizada a construção da Árvore Sintática da gramática do

programa ser formatado, e não a Árvore Sintática do programa que está sendo formatado. Com essa mudança, cresce a necessidade da criação de toda uma nova gama de algoritmos de formatação, que possam trabalhar com a Árvore Sintática da gramática dos programa que está sendo formatado, ao contrário de trabalhar diretamente com a Árvore Sintática do programa que está sendo formatado.

Assim em evoluções deste trabalho, diferente dos outros formatadores de código-fonte (Seção 2: Formatadores de Código), espera-se que existam diversos formatadores de código-fonte (ou algoritmos de formatação), capazes de lidar com os diferentes aspectos das linguagens de programação que se deseja formatar seu código-fonte. A simplicidade de configurações (como somente um número inteiro) poderia ser alterada e propor-se configurações que sejam mais elaboradas (complexas). Permitindo que o usuário tenha maior controle sobre o processo de formatação de código-fonte, em conjunto com as gramáticas escritas para esta ferramenta.

References

- MURPHREE, E. L.; FENVES, S. J. A technique for generating interpretive translators for problem-oriented languages. *BIT Numerical Mathematics*, v. 10, n. 3, p. 310–323, set. 1970. ISSN 1572-9125. DOI: 10.1007/BF01934200. Disponível em: <<https://doi.org/10.1007/BF01934200>>.
- AI HUA WU; PAQUET, J. The translator generation in the general intensional programming compilier. In: 8TH International Conference on Computer Supported Cooperative Work in Design. [S.l.: s.n.], maio 2004. 668–672 vol.2. DOI: 10.1109/CACWD.2004.1349274.
- AHO, Alfred V.; LAM, Monica S. et al. Compilers: Principles, Techniques, and Tools (2Nd Edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.
- HOPCROFT, John E.; MOTWANI, Rajeev; ULLMAN, Jeffrey D. Introduction to Automata Theory, Languages, and Computation. In: New York, NY, USA: Pearson Education, 2006. ISBN 978-0321455369. DOI: 10.1145/568438.568455 . Disponível em: <<http://doi.acm.org/10.1145/568438.568455>>.
- DE JONGE, M. Pretty-printing for software reengineering. In: INTERNATIONAL Conference on Software Maintenance, 2002. Proceedings. [S.l.: s.n.], out. 2002. p. 550–559. DOI: 10.1109/ICSM.2002.1167816.
- SCALABRINO, S. et al. Improving code readability models with textual features. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). Austin, TX, USA: IEEE Computer Society, maio 2016. p. 1–10. DOI: 10.1109/ICPC.2016.7503707 . Disponível em: <https://www.researchgate.net/publication/301685380_Improving_Code_Readability_Models_with_Textual_Features>.
- CECCATO, Mariano et al. Towards Experimental Evaluation of Code Obfuscation Techniques. In: PROCEEDINGS of the 4th ACM Workshop on Quality of Protection. Alexandria, Virginia, USA: ACM, 2008. (QoP '08), p. 39–46. DOI: 10.1145/1456362.1456371. Disponível em: <<http://doi.acm.org/10.1145/1456362.1456371>>.

- SKINNER, Jon. SUBLIME TEXT 3 DOCUMENTATION, Syntax Definitions. 2015a.
Disponível em: <<https://www.sublimetext.com/docs/3/syntax.html>>. Acesso em: 1 mar. 2017.
- SKINNER, Jon. SUBLIME TEXT 3 DOCUMENTATION, Scope Naming. 2015b.
Disponível em: <http://www.sublimetext.com/docs/3/scope_naming.html>. Acesso em: 17 set. 2019.
- SKINNER, Jon. SUBLIME TEXT 3 DOCUMENTATION, Color Schemes. 2015c.
Disponível em: <https://www.sublimetext.com/docs/3/color_schemes.html>. Acesso em: 30 nov. 2019.
- ORTIZ, Ariel. Web Development with Python and Django (Abstract Only). In: PROCEEDINGS of the 43rd ACM Technical Symposium on Computer Science Education. Raleigh, North Carolina, USA: ACM, 2012. (SIGCSE '12), p. 686–686.
DOI: 10.1145/2157136.2157353 . Disponível em: <<http://doi.acm.org/10.1145/2157136.2157353>>.
- SINGER, Adam B. The Command Line Interface. In: PRACTICAL C++ Design. 1. ed. Online: Apress, 2017. p. 97–113. ISBN 978-1-4842-3056-5. DOI: 10.1007/978-1-4842-3057-2_5.