



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

## BREVE SUMÁRIO ⇐ | ←

1	Capa . . . . .	1
2	Folha de Rosto . . . . .	3
3	Ficha Catalográfica . . . . .	3
4	Folha de Aprovação . . . . .	4
5	Resumo em Português . . . . .	5
6	English's Abstract . . . . .	6
7	Lista de Figuras . . . . .	7
8	Lista de Quadros . . . . .	8
10	Lista de Códigos . . . . .	9
11	Sumário . . . . .	11

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

**Notas legais:**

Não há garantia para qualquer parte do software documentado. Os autores tomaram cuidado na preparação desta tese, mas não fazem nenhuma garantia expressa ou implícita de qualquer tipo e não assumem qualquer responsabilidade por erros ou omissões. Não se assume qualquer responsabilidade por danos incidentais ou consequentes em conexão ou decorrentes do uso das informações ou programas aqui contidos.

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.

Arquivo compilado às 23:27h do dia 26 de julho de 2020.

Evandro Coan

Uma Ferramenta de Formatação Programável Por Gramáticas / Evandro Coan; Orientador, Rafael de Santiago, Dr. – Florianópolis, Santa Catarina – Brasil, 25 de Novembro de 2019.  
243 p.

Trabalho de Conclusão de Curso – Universidade Federal de Santa Catarina, INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico, Programa de Graduação em Ciências da Computação.

Inclui referências

1. Formatador do texto, 2. Embelezador de código-fonte, 3. Impressão-bonita, 4. Gramáticas Livre de Contexto, 5. Sintaxe de Linguagens de Programação, I. Rafael de Santiago, Dr. II. Programa de Graduação em Ciências da Computação III. Uma Ferramenta de Formatação Programável Por Gramáticas

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 . . . . .	20
Figura 2	–	Árvore do Programa “token sem nome” . . . . .	24
Figura 3	–	Árvore do Programa “token com nome” . . . . .	24
Figura 4	–	Árvore do Programa “token com outro nome” . . . . .	25
Figura 5	–	Árvore da palavra “12” . . . . .	26
Figura 6	–	Árvore da palavra “1” . . . . .	26
Figura 7	–	Árvore da palavra “2” . . . . .	27
Figura 8	–	Árvore da palavra “3” . . . . .	27
Figura 9	–	Hierarquia de Chomsky . . . . .	29
Figura 10	–	Gramáticas Determinísticas <i>versus</i> suas Linguagens . . . . .	32
Figura 11	–	Gramáticas Determinísticas <i>versus</i> suas Linguagens . . . . .	33
Figura 12	–	Árvore de computação com 4 passos de um Problema da Classe <i>NP</i> . . . . .	41
Figura 13	–	Árvore de computação com 15 passos (Determinísticos) . . . . .	43
Figura 14	–	Árvore de computação com 15 passos <sup>1</sup> utilizando Busca em Largura . . . . .	44
Figura 15	–	Exemplo da tela de configuração de UniversalIndentGUI . . . . .	49
Figura 16	–	Árvore Sintática do Código 10 criada pelo Código 11 . . . . .	64
Figura 17	–	Árvore Sintática Abstrata do Código 12 . . . . .	65
Figura 18	–	Fluxo de uso comum de um analisador . . . . .	68
Figura 19	–	Uso feito pela nova ferramenta de Formatação de Código . . . . .	69
Figura 20	–	Relacionamentos entre os diferentes públicos deste projeto . . . . .	71
Figura 21	–	Relação entre Metagramáticas, Metacompiladores e Metaprogramas . . . . .	72
Figura 22	–	Exemplo de classificação de código-fonte com múltiplos escopos . . . . .	75
Figura 23	–	Diagrama das principais classes . . . . .	79
Figura 24	–	Árvore Sintática “main_formatter_syntax_tree.png” . . . . .	163
Figura 25	–	Árvore Sintática Abstrata “main_formatter_abstract_syntax_tree.png” . . . . .	164
Figura 26	–	Árvore Sintática “main_highlighter_syntax_tree.png” . . . . .	168
Figura 27	–	Árvore Sintática Abstrata “main_highlighter_abstract_syntax_tree.png” . . . . .	169

## **LISTA DE QUADROS** ⇐ | ←

Quadro 1 – Exemplo de Ambiguidade Linguística . . . . .	37
Quadro 2 – Exemplo de Ofuscador de Código . . . . .	51

## LISTA DE CÓDIGOS ⇡ | ←

Código 1	– Exemplo de gramática utilizada pelo Analisador Lark . . . . .	22
Código 2	– Exemplo de gramática com uma Estrutura de Sintaxe . . . . .	25
Código 3	– Trecho do Arquivo de Configuração de Uncrustify . . . . .	46
Código 4	– Antes do ofuscamento . . . . .	51
Código 5	– Depois do ofuscamento . . . . .	51
Código 6	– Exemplo de um arquivo “.sublime-syntax” . . . . .	53
Código 7	– Exemplo de um arquivo “.sublime-color-scheme” . . . . .	54
Código 8	– Exemplo de um programa na linguagem Java . . . . .	61
Código 9	– Exemplo mínimo da metagramática . . . . .	62
Código 10	– Exemplo de gramática pelas regras da mínima metagramática .	62
Código 11	– Exemplo de uso da metagramática e uma gramática . . . . .	63
Código 12	– Pseudo-código do algoritmo Euclidiano (BRENT, 1976) . . . . .	64
Código 13	– Resultado do reconhecimento do programa Java pela gramática	66
Código 14	– Exemplo de configuração utilizada mínima do Formatador de Código . . . . .	66
Código 15	– Exemplo mínimo de Formatador de Código . . . . .	67
Código 16	– Resultado do Formatador de Código para Java . . . . .	67
Código 17	– Símbolo Inicial da Metagramática “ObjectBeauty” . . . . .	74
Código 18	– Exemplo de Gramática, Símbolo Inicial . . . . .	76
Código 19	– Exemplo de Gramática, Contextos . . . . .	76
Código 20	– Exemplo de Gramática, Grupos de Captura . . . . .	77
Código 21	– Exemplo de Gramática, Tipos numéricos . . . . .	78
Código 22	– Exemplo de Gramática, Reconhecimento de Erros . . . . .	78
Código 23	– Construtor do Analisador Semântico . . . . .	80
Código 24	– Construtor do Formatador . . . . .	82
Código 25	– Construtor de “AbstractFormatter” . . . . .	84
Código 26	– Arquivo “source/requirements.txt” . . . . .	157
Código 27	– Arquivo “source/main_formatter.py” . . . . .	157
Código 28	– Arquivo “source/main_highlighter.py” . . . . .	159
Código 29	– Arquivo “source/utilities.py” . . . . .	161
Código 30	– Arquivo HTML gerado pelo programa de exemplo “main_formatter.py” . . . . .	162
Código 31	– Resultado da execução do arquivo “source/main_formatter.py” .	162
Código 32	– Arquivo HTML gerado pelo programa de exemplo “main_highlighter.py” . . . . .	167
Código 33	– Resultado da execução do arquivo “source/main_highlighter.py”	167
Código 34	– Resultado da execução dos Testes de Unidade . . . . .	171
Código 35	– Arquivo “source/unit_tests.py” . . . . .	171

Código 36 – Arquivo “source/semantic_analyzer.py” . . . . .	196
Código 37 – Arquivo “source/code_highlighter.py” . . . . .	211
Código 38 – Arquivo “source/code_formatter.py” . . . . .	221
Código 39 – Arquivo “source/grammars_grammar.pushdown” . . . . .	231

## SUMÁRIO ⇐ | ←

Breve Sumário . . . . .	1
<b>I PESQUISA</b>	<b>13</b>
<b>1 INTRODUÇÃO</b>	<b>14</b>
1.1 CONTEXTUALIZAÇÃO . . . . .	14
1.2 MOTIVAÇÃO . . . . .	17
1.3 OBJETIVOS . . . . .	18
1.3.1 Objetivo Geral . . . . .	18
1.3.2 Objetivos Específicos . . . . .	18
1.4 MÉTODO DE PESQUISA . . . . .	18
1.5 ESTRUTURAÇÃO DO TEXTO . . . . .	19
<b>2 FUNDAMENTAÇÃO TEÓRICA</b>	<b>20</b>
2.1 COMPILADORES E TRADUTORES . . . . .	20
2.2 GRAMÁTICAS . . . . .	21
2.2.1 Hierarquia de Chomsky . . . . .	27
2.2.2 Gramáticas Regulares . . . . .	28
2.2.3 Gramáticas Livres de Contexto . . . . .	30
2.2.4 Gramáticas Sensíveis ao Contexto . . . . .	30
2.2.5 Gramáticas Irrestritas . . . . .	30
2.3 ANALISADORES SINTÁTICOS . . . . .	31
2.3.1 Gramáticas <i>versus</i> Linguagens . . . . .	31
2.3.2 Reduções e Derivações . . . . .	34
2.3.3 Analisadores LR(K) . . . . .	35
2.3.4 Análise Semântica . . . . .	36
2.3.5 Alterações nos Analisadores Sintáticos . . . . .	38
2.4 CLASSES DE COMPLEXIDADE . . . . .	39
2.4.1 Computadores Quânticos . . . . .	40
2.4.2 Complexidade Teórica <i>versus</i> Real . . . . .	41
2.4.3 Mecanismos Reconhecedores . . . . .	42
2.4.4 Busca em Largura e Profundidade . . . . .	43
<b>3 ESTADO DA ARTE</b>	<b>45</b>
3.1 PROGRAMAÇÃO AUXILIADA . . . . .	45
3.2 FORMATADORES DE CÓDIGO . . . . .	45
3.3 QUAL A UTILIDADE DE FORMATADORES? . . . . .	47
3.4 TRABALHOS RELACIONADOS . . . . .	48

3.5	OFUSCADORES . . . . .	50
3.6	ADIÇÃO DE CORES . . . . .	52
<b>3.6.1</b>	<b>Gramáticas . . . . .</b>	<b>53</b>
3.7	LISTA DE CURIOSIDADES . . . . .	55
<b>II</b>	<b>IMPLEMENTAÇÃO</b>	<b>59</b>
<b>4</b>	<b>FORMATADOR DESENVOLVIDO . . . . .</b>	<b>60</b>
4.1	VISÃO GERAL . . . . .	61
4.2	INTRODUÇÃO À METAGRAMÁTICA . . . . .	68
4.3	ESPECIFICAÇÃO DA METALINGUAGEM . . . . .	73
4.4	ANALISADOR SEMÂNTICO . . . . .	78
4.5	FORMATADOR DE CÓDIGO . . . . .	82
<b>5</b>	<b>CONCLUSÃO . . . . .</b>	<b>86</b>
5.1	COMPARAÇÃO COM OUTROS TRABALHOS . . . . .	87
5.2	TRABALHOS FUTUROS . . . . .	88
	<b>REFERÊNCIAS . . . . .</b>	<b>91</b>
	<b>APÊNDICE A – MANUAL DO FORMATADOR . . . . .</b>	<b>155</b>
	<b>APÊNDICE B – MAIN_FORMATTER.PY . . . . .</b>	<b>162</b>
	<b>APÊNDICE C – MAIN_HIGHLIGHTER.PY . . . . .</b>	<b>167</b>
	<b>APÊNDICE D – TESTES DE UNIDADE . . . . .</b>	<b>171</b>
	<b>APÊNDICE E – ANALISADOR SEMÂNTICO . . . . .</b>	<b>195</b>
	<b>APÊNDICE F – CÓDIGO DE ADIÇÃO DE CORES . . . . .</b>	<b>211</b>
	<b>APÊNDICE G – CÓDIGO DO FORMATADOR . . . . .</b>	<b>221</b>
	<b>APÊNDICE H – CÓDIGO DA METAGRAMÁTICA . . . . .</b>	<b>231</b>
	<b>APÊNDICE I – ARTIGO SOBRE O TCC . . . . .</b>	<b>234</b>

# **Parte I**

## **Pesquisa** |

## 1 INTRODUÇÃO ⇐ | ←

A [primeira parte](#) deste trabalho será baseado em artigos, livros, outras dissertações, sites confiáveis e por novas evidências demonstradas no desenvolvimento deste trabalho. E finalmente, na [segunda parte](#) deste trabalho, será apresentada a sugestão e implementação de uma ferramenta.

### 1.1 CONTEXTUALIZAÇÃO ⇐ | ←

Programação de computadores usualmente não é uma tarefa solitária. Muitos projetos podem sempre ser escritos pela mesma pessoa durante toda a vida desse projeto mas, o quanto grande pode ser um projeto que uma pessoa pode fazer sozinha? Poderia uma pessoa escrever sozinha todo o sistema de controle de tráfego aéreo dos Estados Unidos? Uma vez que precisa-se escrever sistemas computacionais maiores, tende-se a colocar vários programadores, engenheiros de software, gerentes de projeto, etc, para que se possa escrever o sistema computacional requerido dentro do prazo que dado projeto computacional possui.

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](#)).

Perguntas como “O quê são boas práticas de programação” não são fáceis de responder. Cada programador aprende a escrever seus códigos-fonte de uma determinada maneira, seja utilizando 2, 4, ou 8 espaços para criar blocos de indentação, ou sempre deixar uma linha em branco antes de estruturas de controle como if's, while's, etc ([ALLAMANIS; BARR; SUTTON, 2014](#)). Quando diferentes programadores cooperam em grandes projetos, como essas pessoas todas poderiam trabalhar juntas, mesmo elas não tendo as mesmas experiências? Uma barreira adicional para cada um desses programadores no 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; KALLEBERG \*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 do tipo “if”, “for”, etc. 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?

Existem algumas possibilidades, uma delas poderia ser que a metade do código-fonte escrita pelo “Programador A” estará de um jeito. Já a outra metade escrita pelo “Programador B” estará de outro. Então, que tipos de problema isso poderia causar? Uma vez que o código-fonte não está mais todo escrito sobre um mesmo padrão, certas heurísticas tem sua taxa de acerto reduzida ao fazer o reconhecimento dos padrões. Um exemplo de padrão que estaria prejudicado seria o de rapidamente identificar onde pode estar uma estrutura de controle, como “if” ou “for” 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 sendo facilmente reconhecida como um separador de blocos de código-fonte ([BAGGE; HASU, 2013](#)).

E por mais uma vez, estas heurísticas são fortemente dependentes do aprendizado de cada programador durante sua vida. Imagine um segundo programador que não está habituado com a heurística de outro, e toda vez que ele for ler o código-fonte deste terceiro programador, ela estará perdido para poder se localizar rapidamente dentro do código-fonte. E portanto, o que isso significa? Que ele terá que atentamente ler o código-fonte escrito por este terceiro programador, uma vez que a sua organização difere dos padrões no qual ele sabe como rapidamente se localizar ([MINELLI; MOCCI; LANZA, 2014](#)).

As diferenças entre os diversos estilos de programação entre os programadores deve-se à diversos fatores:

- 1 Alguns programadores simplesmente fazem de um determinado jeito porque foi assim que eles aprenderam de seus tutores ou possuem determinado estilo devido as suas experiências iniciais e nunca preocuparam-se em questionar se poderia ser feito de outra maneira. Um caso onde isso pode acontecer é devido ao cansaço que algumas linguagens de programação como  $\text{\LaTeX}$  causam por sua sintaxe incomum e extraordinária. Uma vez que você já está cansado de ver tantos erros, você simplesmente para de questionar como o texto poderia ser formado e aceita o que você encontra escrito e funcionando como parte da sintaxe da linguagem;
- 2 Outros podem ter determinadas características na formatação da escrita de seus códigos-fonte devido a algum aspecto emocional ou físico. Emocionalmente o programador pode “não gostar” de alguma característica porque sofreu algum trauma ou simplesmente pode estar muito acostumado a ter todos os elementos muito próximos um dos outros na tela. Assim, mesmo tal característica não sendo nada fácil para que outras pessoas entendam o código-fonte, ela pode ser muito fácil de ser compreendida pelo próprio programador que a escreveu, já que o mesmo vem a muitos anos habituado a realizar a escrita

de seus códigos-fonte desta determinada forma, portanto é muito hábil com a mesma e rapidamente consegue entender o que escreve;

- 3 Uma característica física que pode determinar a formatação de código-fonte de alguém pode ser um teclado defeituoso. Caso determinada tecla não funcione adequadamente e a pessoa não se dê ao trabalho de comprar um teclado novo, a pessoa pode então adquirir um novo hábito, e uma vez que ela aprenda este hábito, ela provavelmente continuará com ele mesmo depois de comprar um teclado novo. Por exemplo, suponha que a sua tecla “TAB” não funcione. Assim, ao invés de somente pressionar “TAB” uma vez para indentar seu código-fonte, você ganha hábito de pressionar a tecla de espaços 4 vezes. Portanto, nasce um motivo para alguém utilizar espaços ao invés “TAB’s” para realizar a indentação de seu código-fonte.

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 facilmente realizar o rastreamento das mudanças ([LASTER, 2016a](#)) e assim, permitir que entenda-se melhor o que cada programador está fazendo, todas as vezes que eles formalizam uma nova alteração no código-fonte, por exemplo, através de uma “*commit*” como conhecidas em sistemas *git* ([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)<sup>1</sup>.

Segue-se estas conclusões devido aos problemas gerados ao permitir que cada programador escreva como lhe agrada melhor. Caso cada um escreva como bem entende, irá existir ruídos de formatação em abundância espalhados por todo o código-fonte, o quê complicará a revisão de código-fonte, dificultando entender o que cada programador fez ([ARNDT; RADTKE, 2016](#)).

Portanto, caso todos os programadores decidam re-escrever o histórico, fazendo

<sup>1</sup> 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.

mudanças insignificantes como inserindo novas linhas antes de cada “if”:

- 1 Eles irão desperdiçar um tempo que eles poderiam estar utilizando para escrever código-fonte;
- 2 O histórico *git* irá conter ruídos desnecessários, pois o foco de um sistema de controle de versões são conter mudanças se sejam significantes, como criações de novas funções, correção de *bugs* ([BENDIK; BENES; CERNA, 2017; STEINERT et al., 2009](#)), etc.

Poderia-se também pensar em uma abordagem para criar um novo sistema de controle de versão, que foca apenas mudanças significativas no código-fonte enquanto faz-se a revisão das mudanças no código-fonte. No entanto, essa abordagem poderia não ser ideal, pois, por exemplo, permitiria que os programadores iniciassem guerras tediosas de ajustes de código-fonte improdutivos. Por exemplo, imagine como seria se todo os dias você tivesse que passar pelo código-fonte re-adicionando novas linhas antes de cada um dos seus amados if's, só porque algum programador do turno da noite tinha acabado de removê-los ([ATWOOD, 2009](#))?

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 sua frente por meio de layouts extravagantes ou incomuns ([ATKINS et al., 2002](#)).

## 1.2 MOTIVAÇÃO ⇐ | ←

Ter que aprender e configurar muitas ferramentas de formatação de código-fonte é um tarefa cansativa ([Seção 3.2: Formatadores de código](#)). 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.

### 1.3 OBJETIVOS ⇐ | ←

Devido ao trabalho tratar-se de uma tese de graduação, limita-se seu escopo de trabalho a criação de um núcleo básico de recursos. Enumerar e explicar as ferramentas de formatação de código-fonte e realizar o desenvolvimento de uma nova ferramenta de formatação de código-fonte que seja extensível por gramáticas fornecidas pelos usuários.

#### 1.3.1 Objetivo Geral ⇐ | ←

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

#### 1.3.2 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.

### 1.4 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](#)).

## 1.5 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.

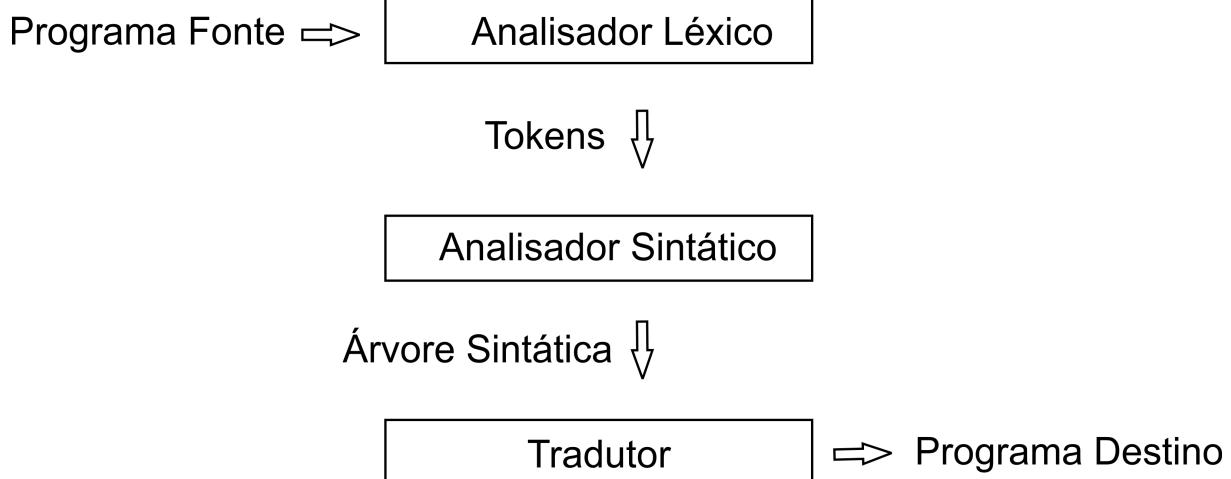
## 2 FUNDAMENTAÇÃO TEÓRICA ⇐ | ←

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](#)). Pelo outro lado, dado um programa de entrada em C++ e um programa de saída em *Assembly*, tem-se um processo de compilação ([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 Ascendentes<sup>1</sup> ou Descendentes<sup>2</sup>. 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 folhas<sup>3</sup>.

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 Ascendentes<sup>4</sup> (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)  $V = V_n \cup V_t$ .

<sup>1</sup> Um conjunto  $V_t$  de símbolos terminais, chamados algumas vezes de “tokens” devido a sua forte conexão. Cada terminal corresponde a um símbolo presente no alfabeto da 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,

<sup>2</sup> Do inglês, *Bottom-Up*.

<sup>3</sup> Do inglês, *Top-Down*.

<sup>3</sup> Como pode ser observados 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.

<sup>4</sup> Conceito abordado na Seção 2.3: Analisadores Sintáticos.

quando a gramática possui símbolos inúteis ([HOPCROFT; MOTWANI; ULLMAN, 2006](#)));

- 2 Um 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 Sintática<sup>5</sup>. 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:

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

- 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](#).

---

<sup>5</sup> Por exemplo, quando a gramática da linguagem não contém símbolos inúteis, i.e., todos os símbolos da gramática são férteis e permitem a geração de palavras além do conjunto vazio  $\emptyset$  ([HOPCROFT; MOTWANI; ULLMAN, 2006](#)).

### Código 1 – Exemplo de gramática utilizada pelo Analisador Lark

```

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" )

```

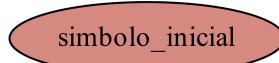
A sintaxe da gramática utilizada no [Código 1](#), é uma adaptação especial do padrão EBNF<sup>6</sup> ([PATTIS, 1994](#); [PARR, 2013](#); [SHINAN, 2019b; 2018](#)) que o Analisador Lark faz. Símbolos que possuem todas as letras em “MAIÚSCULA” definem os símbolos terminais da gramática que aparecerão na Árvore Sintática. Enquanto símbolos escritos todos em letras “minúsculas” definem os símbolos não-terminais da gramática. Para mais informações sobre a sintaxe de entrada de gramáticas do Analisador Lark, consulte sua documentação ([SHINAN, 2019b; 2018](#)).

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.

---

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

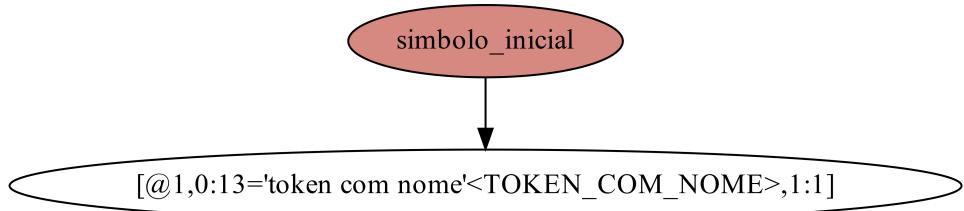
Figura 2 – Árvore do Programa “token sem nome”



Fonte: Própria

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.

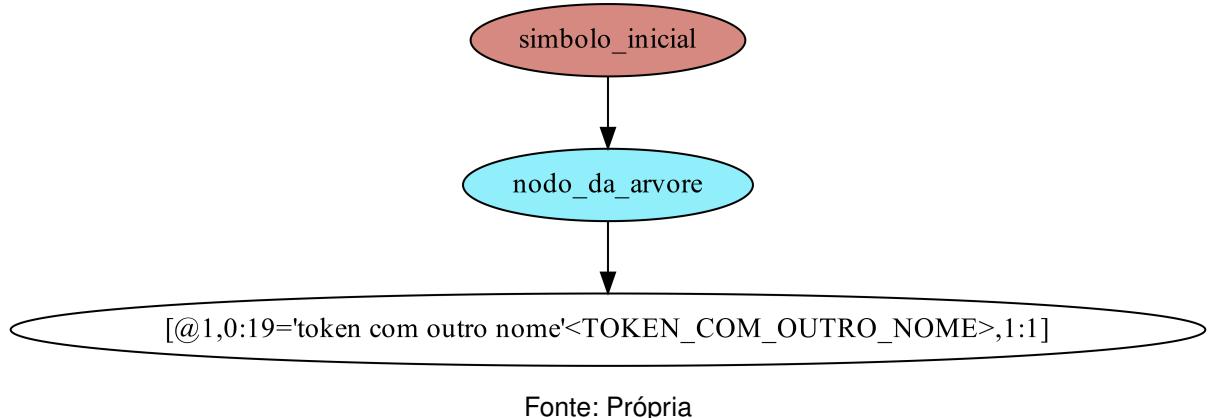
Figura 3 – Árvore do Programa “token com nome”



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: NUMERO01 NUMERO02 |  NUMERO03  | nao_terminal
6         nao_terminal: NUMERO01 | NUMERO02
7         NUMERO01: "1"
8         NUMERO02: "2"
9         NUMERO03: "3"
10        """
11        ,
12        start='simbolo_inicial',
13        parser="lalr",
14    )
15
16 def parseit(program, filename):
17     pushdown.tree.pydot__tree_to_png(
18         parser.parse( program ), filename, "TB", debug=1, dpi=600 )
19
20 parseit( '1', "palavra1.png" )
21 parseit( '2', "palavra2.png" )
  
```

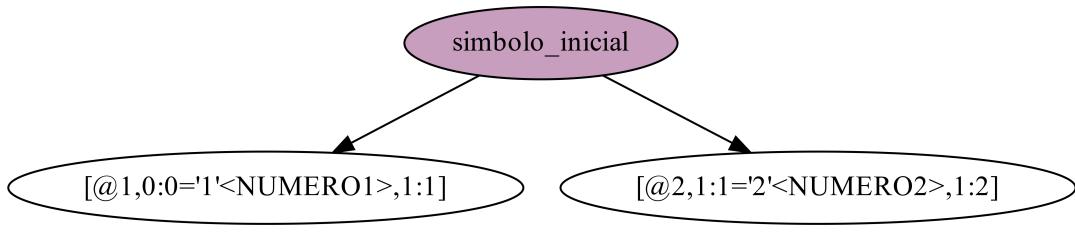
```

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.

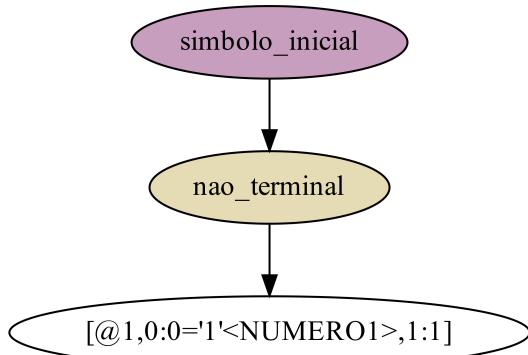
Figura 5 – Árvore da palavra “12”



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 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”

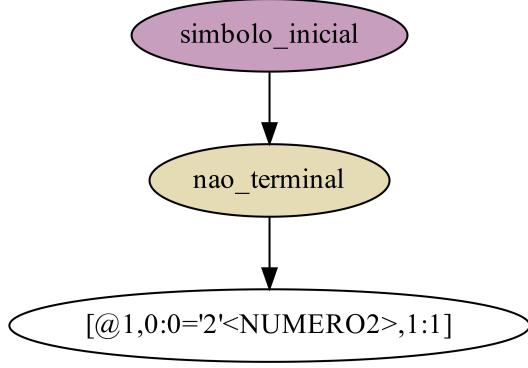


Fonte: Própria

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 uso de um único símbolo do

alfabeto, o símbolo “2” (implicitamente derivado a partir das produções da gramática pelo Analisador Lark).

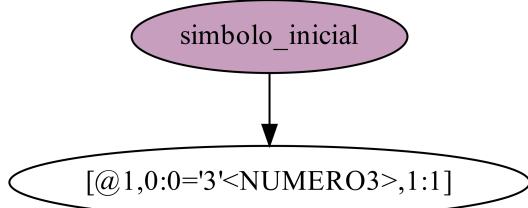
Figura 7 – Árvore da palavra “2”



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 mínimo<sup>7</sup> Gramáticas Tipo 0 (AHO; ULLMAN, 1972; CHOMSKY, 1956), também conhecidas como Gramáticas Irrestritas

<sup>7</sup> 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 tempo<sup>8</sup>, 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 realizar diversas classificações como 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 computational<sup>9</sup> 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 Bombeamento<sup>10</sup> 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 ⇐ | ←

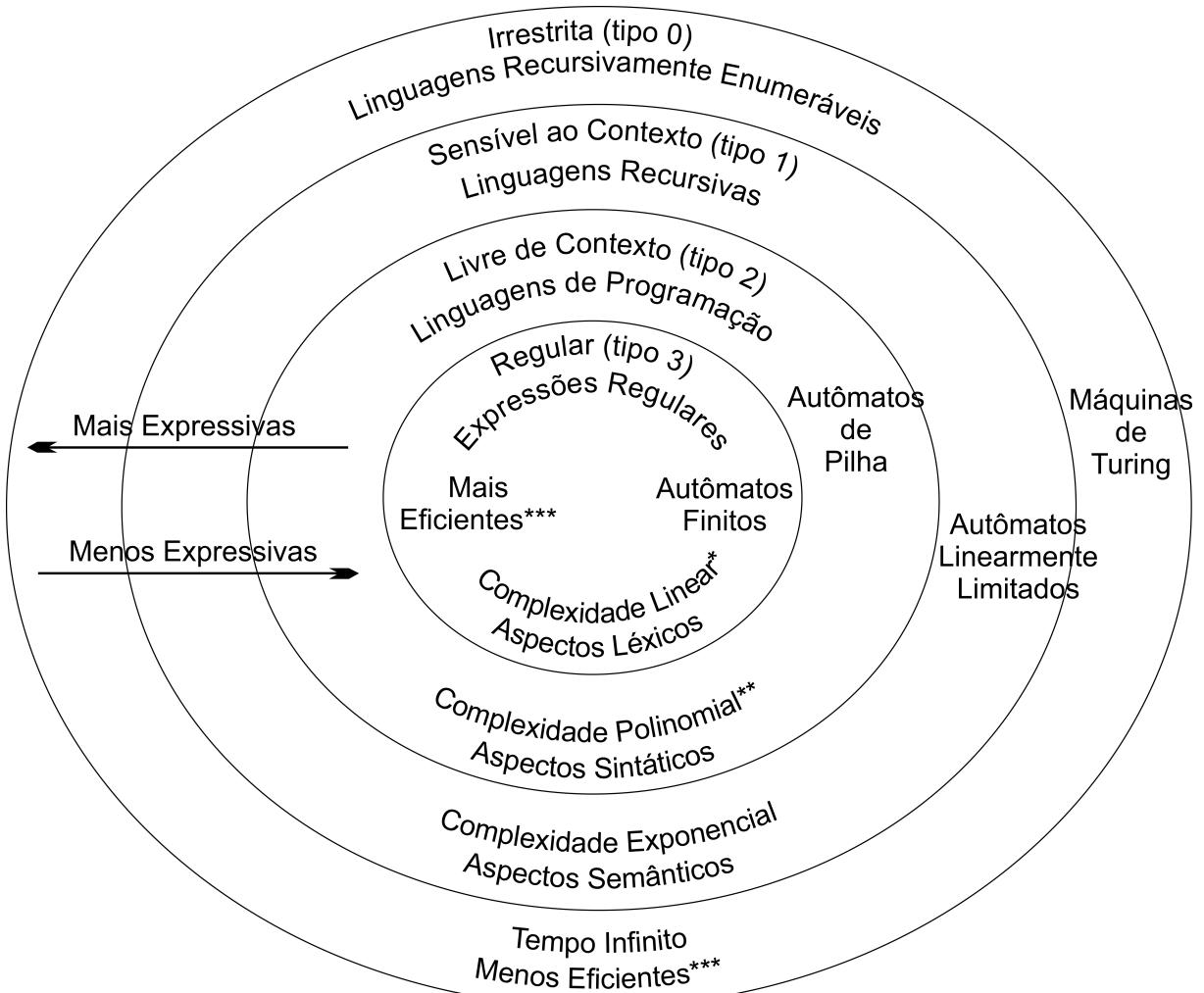
Gramáticas Regulares (também conhecidas como Tipo 3) são todas aquelas reconhecidas por Autômatos Finitos Determinísticos e/ou Não-Determinísticos. Autômatos reconhecem linguagens e gramáticas geram palavras de sua linguagem (Seção 2.3.2:

<sup>8</sup> 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(;;);”?

<sup>9</sup> Tempo computational refere-se ao número de passos ou instruções necessários para executar determinado procedimentos (Seção 2.4: Classes de Complexidade).

<sup>10</sup> 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: 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. Veja [Torán \(1991\)](#) e [Arora e Barak \(2009\)](#), para aprender mais sobre Classes de Complexidade.

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: Classes de Complexidade](#).

Nota: Aprenda o que são Gramáticas Determinísticas ou Não-Determinísticas lendo a [Seção 2.3.1: Gramáticas versus Linguagens](#).

[Reduções e Derivações](#)). Para toda linguagem reconhecida por Autômato Finito, existe uma Gramática Regular que gera palavras dessa linguagem. Gramáticas de Lingua-

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 $\Leftarrow | \Leftarrow$

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 $\Leftarrow | \Leftarrow$

Gramáticas Sensíveis ao Contexto (também conhecidas como Tipo 1) são todas aquelas reconhecidas por Autômatos Linearmente Limitados<sup>11</sup>, 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  $\alpha$  e  $\beta$ , respectivamente:

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

### 2.2.5 Gramáticas Irrestritas $\Leftarrow | \Leftarrow$

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 Turing<sup>12</sup>, e diferente das Gramáticas Sensíveis ao Contexto, a Máquina de Turing não possui parada garantida<sup>13</sup>.

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

<sup>11</sup> Do inglês, Linear Bounded Automata ([DAVIS; SIGAL; WEYUKER, 1994](#)).

<sup>12</sup> 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.

<sup>13</sup> Aprenda mais a respeito procurando sobre o problema da parada [Royer \(2015\)](#) e [Sipser \(2012\)](#).

implementação pode ser representada por algoritmos<sup>14</sup>, pois possuem parada garantida, apesar de terem em pior caso, tempo exponencial ao contrário de tempo infinito<sup>15</sup>, como nas Linguagens Irrestritas.

## 2.3 ANALISADORES SINTÁTICOS ⇐ | ←

Analisadores são equivalentes à Mecanismos Reconhecedores como Autômatos Finitos, Autômatos de Pilha ou Máquinas de Turing (NUNES, 2017; ROYER, 2015). 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<sup>16</sup> é um nome dado para analisadores que recebem como entrada uma Gramática Livre de Contexto que representa os aspectos estruturais de uma linguagem (NIEHUES, 2013), 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<sup>17</sup> 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 Ambíguas<sup>18</sup> 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: Classes de Complexidade).

A maior classe de Gramáticas Determinísticas suportadas por Analisadores Sin-

<sup>14</sup> Aprenda mais sobre decidibilidade e computabilidade com Royer (2015) e Sipser (2012).

<sup>15</sup> Porque podem em pior caso levar tempo infinito para responder, imagine o seguinte trecho de código-fonte na linguagem C “for(;;);”.

<sup>16</sup> Além de Analisadores Sintáticos (para Gramáticas Livre de Contexto), existem muitos outros como Analisadores Semânticos (Gramáticas Sensíveis ao Contexto) (WOODS, 1970).

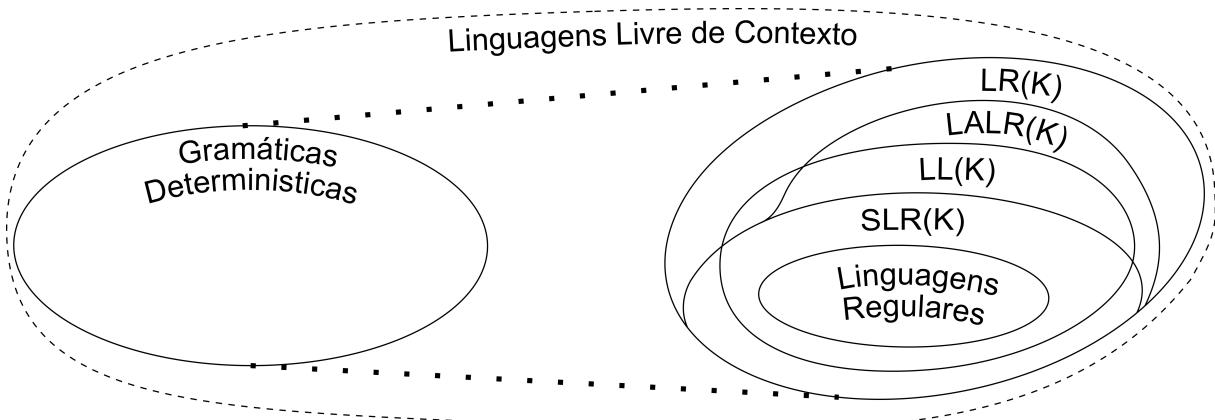
<sup>17</sup> Como visto no começo desde capítulo na Seção 2.1: Compiladores e Tradutores.

<sup>18</sup> Do inglês, *Inherently Ambiguous Languages*. Veja o que significa ambiguidade na Seção 2.3.4: Análise Semântica e Quadro 1.

táticos são as Gramáticas  $LR(K)$ <sup>19</sup>. Analisadores  $LR(K)$  (AHO; LAM *et al.*, 2006) são Ascendentes e reconhecem um subconjunto das Linguagens Livre de Contexto (Figura 10). Já os Analisadores  $LL(K)$ <sup>20</sup> são Descendentes (PARR, 2013; PARR; FISHER, 2011; PARR; HARWELL; FISHER, 2014) e reconhecem somente um subconjunto das Linguagens  $LR(K)$ <sup>21</sup>.

Na Figura 10, encontra-se um Diagrama de Venn (KESTLER *et al.*, 2004) com 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 Determinísticas<sup>22</sup>, 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 Análise<sup>23</sup>.

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.

A Figura 11, não é inteiramente um Diagrama de Venn (KESTLER *et al.*, 2004), inicialmente, nas camadas mais externas, ela apresenta uma relação abstrata entre Linguagens Ambíguas e Gramáticas Determinísticas. O Conjunto das Gramáticas Livre

<sup>19</sup> 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.

<sup>20</sup> 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.

<sup>21</sup> Diz-se que uma linguagem é  $LR(K)$  ou  $LL(K)$  quando ela é reconhecida por este analisador.

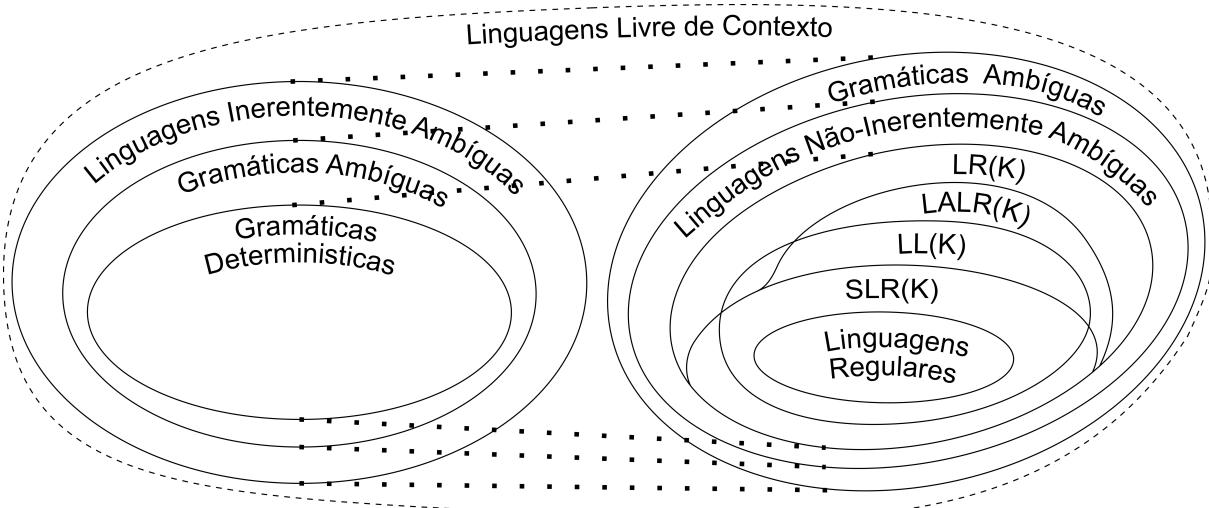
<sup>22</sup> As Gramáticas Determinísticas representam o conjunto de Linguagens que podem ser Analisadas Deterministicamente e tais Linguagens também podem ser conhecidas como  $LR(K)$ . Reveja os parágrafos após a Figura 9.

<sup>23</sup> Do inglês, *Parsing Table*, (AHO; LAM *et al.*, 2006).

de Contexto Determinísticas está contido dentro das Linguagens Livre de Contexto<sup>24</sup>.

O primeiro nível significa que todas as Linguagens Inerentemente Ambíguas<sup>25</sup> são representáveis somente por Gramáticas Ambíguas. O segundo nível significa que Linguagens Não-Inerentemente Ambíguas<sup>26</sup> podem ser representadas por Gramáticas Ambíguas e/ou Determinísticas.

Figura 11 – 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 programaticamente<sup>27</sup>, 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

<sup>24</sup> Também existem Gramáticas Sensíveis ao Contexto Determinísticas (WOODS, 1970), entretanto, algoritmos de análise possuem em pior caso, complexidade exponencial (POST, 2019).

<sup>25</sup> É comum confundir-se e chamar Gramáticas de Inerentemente Ambíguas, mas esse termo não existe para gramáticas. Ou elas são Ambíguas ou Não. Somente uma linguagem pode ser Inerentemente Ambíguia.

<sup>26</sup> Somente utilizado para enfatizar o conjunto de Linguagens na qual existem Gramáticas Ambíguas e Determinísticas (ou Não-Ambíguas).

<sup>27</sup> Neste caso, o analisador pode ser conhecido como Descendente Recursivo, (do inglês, Recursive Descent).

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 um programa pertence ou não a linguagem (análise linear ao tamanho do programa<sup>28</sup> 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: Classes de Complexidade), precisam ser utilizados para construção da Árvore Sintática. Como mostrado para Máquinas de Turing Não-Determinísticas na Seção 2.4.3: Mecanismos Reconhecedores, Analisadores com Backtracking também funcionam em pior caso, com tempo exponencial e podem escolher uma estratégia como Busca em Profundidade<sup>29</sup> (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

<sup>28</sup> 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).

<sup>29</sup> Veja a Seção 2.4.4: Busca em Largura e Profundidade para saber mais.

- “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”.

Para reduções, estas mesmas condições se aplicam, mas em ordem reversa, i.e.,  $\Leftarrow$ ,  $\Leftarrow^*$  e  $\Leftarrow^+$ , ao invés de  $\Rightarrow$ ,  $\Rightarrow^*$  e  $\Rightarrow^+$  (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 as 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) $\Leftarrow | \Leftarrow^*$

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)<sup>30</sup> e SLR(K)<sup>31</sup> com o 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 ⇐ | ←

Usualmente<sup>32</sup>, 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.

<sup>30</sup> Do inglês, *Look-Ahead LA(K) LR(0)*, onde LR(0) é um Analisador LR(K) com  $K = 0$ .

<sup>31</sup> Do inglês, *Simple LR(K) parser*.

<sup>32</sup> 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.

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 ambiguidades<sup>33</sup> ([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 ([WOODS, 1970](#))), pois essa característica da linguagem C é um aspecto Sensível ao Contexto. Portanto, para simplificar a implementação do Analisador Semântico, escolheu-se programaticamente<sup>34</sup> 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

<sup>33</sup> Conhecido também como Não-Determinismo.

<sup>34</sup> 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](#).

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 “ $x * y;$ ”. Portanto, o “Analizador Sintático” é capaz de consultar a Tabela de Símbolos e descobrir se dado *token* é um tipo ou uma variável numérica.

Entretanto, 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 na caso da implementação da linguagem C, 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 garantida pela sua Classe de Complexidade (Seção 2.4: Classes de Complexidade).

Sintaxe e Semântica de Linguagens são completamente ortogonais. Gramáticas

de Linguagens Irrestritas<sup>35</sup> podem ser Turing Completas<sup>36</sup> 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 Compleitude<sup>37</sup> (KIRPICHOV, 2014; FENNER, 2019).

## 2.4 CLASSES DE COMPLEXIDADE $\in | \leftarrow$

Como um todo, o conjunto de Linguagens Regulares pode ser considerado com complexidade linear<sup>38</sup> 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

<sup>35</sup> Não a linguagem que elas representam, mas a própria gramática em si (FENNER, 2019).

<sup>36</sup> A Turing Compleitude acontece quando uma dada linguagem pode simular o funcionamento completo de uma Máquina de Turing (MICHAELSON, 2016).

<sup>37</sup> Caso isso esteja confuso, reveja a Figura 9 e note que de todas as Linguagens, quem tem Turing Compleitude são as Linguagens Irrestritas, enquanto Autômatos Finitos são um subconjunto das Máquinas de Turing (FENNER, 2019).

<sup>38</sup> Complexidade linear é um caso particular de complexidade polinomial onde o grau do Polinômio é 1, i.e.,  $\Theta(n)$ . Aprenda mais sobre classes de complexidade com Cormen *et al.* (2009) e Arora e Barak (2009).

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 Derivação<sup>39</sup>.

### 2.4.1 Computadores Quânticos $\Leftarrow | \Leftarrow$

Com a exceção de alguns problemas específicos (KATZGRABER *et al.*, 2015), a execução probabilística de Computadores Quânticos (ABRAMS; LLOYD, 1998), baseados nas leis da Física Quântica (DICKE; WITTKE, 1963), podem cortar<sup>40</sup> caminho “pulando” Ramos de Computação Não-Determinísticos (da computação Clássica) com a superposição quântica. Assim, conseguindo resolver alguns problemas que são exponenciais, em tempo polinomial ao tamanho da entrada, utilizando algoritmos específicos para computadores quânticos (KANAMORI *et al.*, 2006; ROSA, 2019).

Esta é a gama de problemas nos quais Computadores Quânticos são úteis (RIEF-FEL; POLAK, 2000), não sendo assim, substitutos completos da Computação Tradicional (ou Clássica) (PUREWAL, 2006), somente otimizadores na resolução de alguns problemas que podem ser otimizados devido as propriedades específicas/probabilísticas das leis Física Quântica (DEUTSCH; PENROSE, 1985).

Pode-se confundir Computadores Quânticos como equivalentes a Analisadores Não-Determinísticos devido as nomenclaturas utilizadas. Enquanto analisadores são Não-Determinísticos devido à ambiguidades nas gramáticas de entrada, Computadores Quânticos são Não Determinísticos devido à serem baseado em modelos Probabilísticos, i.e., Computadores Quânticos não são equivalentes a Analisadores Não-Determinísticos devido a sua execução ser probabilística (SHOR, 1999; BRUKNER *et al.*, 2003; ROSA, 2019).

Diferente dos Computadores Tradicionais, Computadores Quânticos são construídos com base nas leis da Física Quântica, que são radicalmente diferentes das Leis da Física Tradicional ou Clássica, i.e., as Leis de Newton. As Leis da Física Clássica regem os elementos muitos grandes na escala galáxias, planetas, células e virus (HALLIDAY; RESNICK; WALKER, 2013). Já as Leis da Física Quântica regem as elementos muito pequenos na escala de átomos, elétrons, prótons, fôtons e quarks (DICKE; WITTKE, 1963).

---

<sup>39</sup> Como a gramática é Não-Determinística, existem muitas possíveis Árvores de Derivação, (devido à ambiguidade da gramática, Quadro 1).

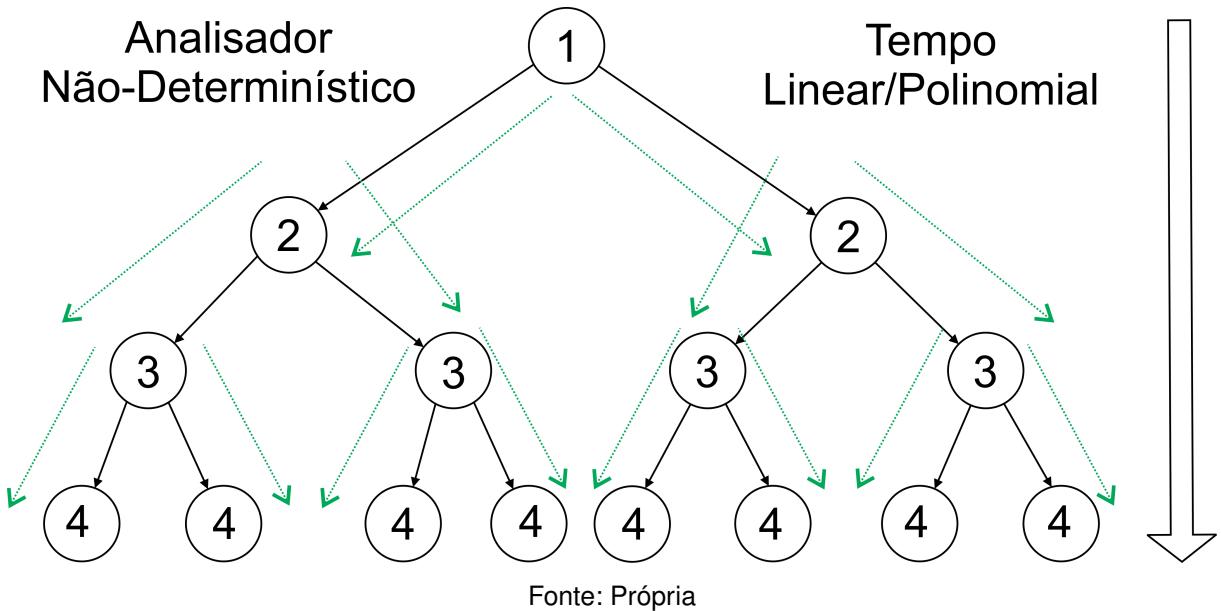
<sup>40</sup> Devido as probabilidades envolvidas, somente um ou alguns dos Ramos de Computação serão seguidos durante a execução do Algoritmo Quântico, pelo Computador Quântico.

### 2.4.2 Complexidade Teórica *versus* Real $\Leftarrow | \Leftarrow$

Na [Figura 12](#), encontra-se uma Árvore de Computação de um Analisador Não-Determinístico. Diz-se que o tempo de execução de um Analisador Não-Determinístico é Não-Determinístico Polinomial ( $NP^{41}$ ) ao tamanho da entrada, porque um Analisador Não-Determinístico executa simultaneamente todos os Ramos de Computação Não-Determinísticos ([HOPCROFT; MOTWANI; ULLMAN, 2006](#); [ROYER, 2015](#)).

Como mostrado na [Figura 12](#), após cada um dos passos de computação 1, 2, 3 e 4, todos os 15 Ramos de Computação foram concluídos. Cada um desses passos corresponde a um item a ser analisado na entrada do programa. E esta computação, acontece em tempo Não-Determinístico Polinomial, com expoente de  $n$  igual a 1, i.e.,  $\Theta(n^1)$ .

Figura 12 – Árvore de computação com 4 passos de um Problema da Classe  $NP$



O que torna a computação Não-Determinística<sup>42</sup> é o fato de cada um dos itens 1, 2, 3 e 4 da entrada, permitirem simultaneamente a escolha de mais de um caminho na escolha do próximo estado do analisador, i.e., mais de um Ramo de Computação, devido a ambiguidades da gramática de entrada ([VASUDEVAN; TRATT, 2013](#); [PARR, 2013](#)). Nesse contexto,  $P$  representa o conjunto de problemas resolvidos em tempo Determinístico Polinomial (por Máquinas de Turing Determinísticas), enquanto  $NP$ , o conjunto de Problemas resolvidos em tempo Não-Determinístico Polinomial (por

<sup>41</sup> Do inglês, *Non-Deterministic Polynomial Time*, comumente conhecida pela pergunta,  $P \stackrel{?}{=} NP$ , i.e., a classe de problemas com complexidade de tempo Determinístico Polinomial, está estritamente contida na classe de problemas  $NP$  (Não-Determinísticos Polinomiais) ([ARORA; BARAK, 2009](#))?

<sup>42</sup> E pertencente a classe dos problemas Não-Determinístico Polinomial.

Máquinas de Turing Não-Determinísticas). Portanto, um problema Não-Determinístico Polinomial, somente pode ser resolvido por uma Máquina de Turing Determinística em tempo exponencial. Veja as [Figuras 12 e 13](#) e as compare.

O tempo de execução será linear ao tamanho da entrada caso o Analisador Não-Determinístico seja de uma Linguagem Regular e implementado através de um Autômato Finito Não-Determinístico. O tempo de execução será polinomial ao tamanho da entrada caso o Analisador Não-Determinístico seja de uma Linguagem Livre de Contexto Ambígua (Gramática Não-Determinística) e implementado através de um Autômato de Pilha Não-Determinístico.

#### 2.4.3 Mecanismos Reconhecedores |

Uma vez que o conjunto de Gramáticas Determinísticas LR(K) (com tempo linear) está contido no conjunto das Gramáticas Livres de Contexto, não se considera tempos Análise Lineares ou Polinomiais de execução para Linguagens Sensíveis ao Contexto ou Irrestritas, porque tudo o que possui eficiência garantida pela sua classe de complexidade são as Gramáticas Regulares e Livre de Contexto Determinísticas. Pelo outro lado, Gramáticas Sensíveis ao Contexto terão em pior caso, complexidade exponencial ([BUNTRICK; LORYŚ, 1992](#)).

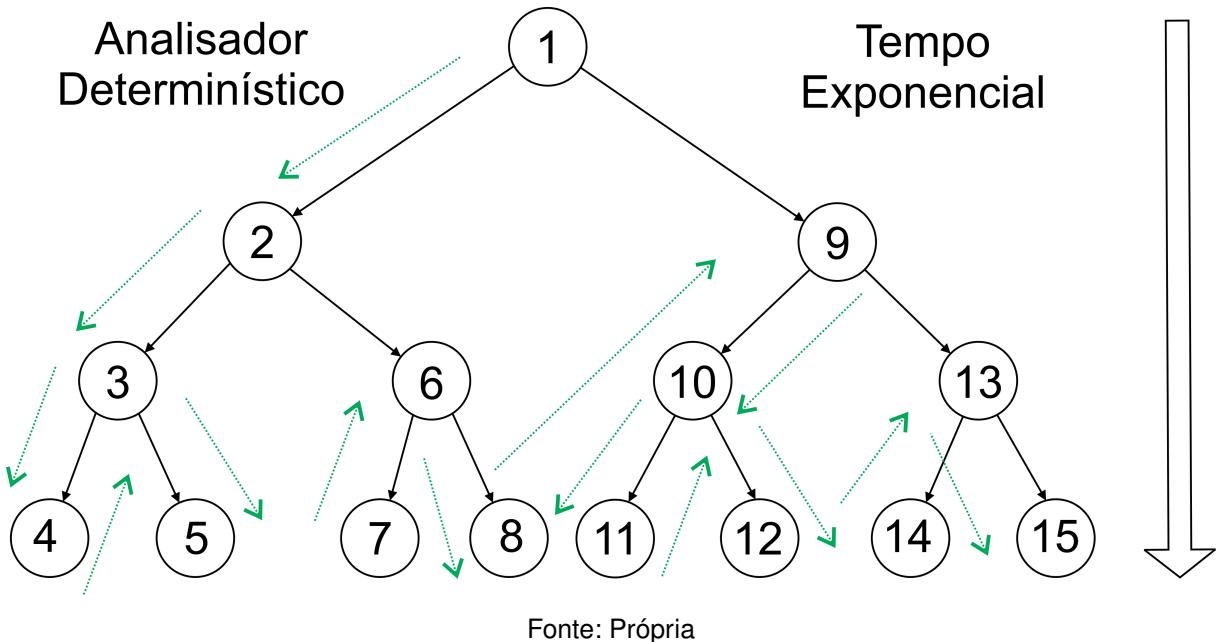
Máquinas de Turing Não-Determinísticas que resolvem os problemas da Classe  $NP$  em tempo Não-Determinístico Polinomial não existem fisicamente ([LANG, 1974](#)). Portanto sua complexidade de tempo reduzido (Não-Determinístico Polinomial) não pode ser alcançado e seu tempo de execução é Determinístico Exponential. Para simular o funcionamento de uma Máquina de Turing Não-Determinística, utiliza-se uma Máquina de Turing Determinística<sup>43</sup> ([SIPSER, 2012](#); [ROYER, 2015](#)).

Máquinas de Turing Determinísticas e Não-Determinísticas são equivalentes, pois sempre é possível simular o funcionamento de uma Máquina de Turing Não-Determinística, utilizando uma Máquina de Turing Determinística ([HOPCROFT; MOTWANI; ULLMAN, 2006](#)). Na [Figura 13](#), encontra-se a mesma Árvore de Computação apresentada na [Figura 12](#), mas com a diferença de que desta vez utiliza-se uma Máquina de Turing Determinística ao contrário de uma Máquina de Turing Não-Determinística. Com isso, ao invés de um tempo polinomial ao tamanho da entrada, tem-se um tempo exponencial ao tamanho da entrada.

Para que uma Máquina de Turing Determinística possa processar uma Gramática Não-Determinística, é necessário executar cada um dos Ramos de Computação.

<sup>43</sup> Máquinas de Turing Determinísticas seriam os equivalentes aos computadores de propósito geral.

Figura 13 – Árvore de computação com 15 passos (Determinísticos)



Já Máquinas de Turing Não-Determinísticas<sup>44</sup> executam simultaneamente todos os Ramos de Computação Não-Determinísticos (Figura 12), conseguindo assim, desempenho linear ou polinomial ao tamanho da entrada compondo os problemas da classe *NP* (com tempo Não-Determinístico Polinomial) (HOPCROFT; MOTWANI; ULLMAN, 2006).

#### 2.4.4 Busca em Largura e Profundidade ⇐ | ←

Quando uma Máquina de Turing Determinística é utilizada para simular o funcionamento de uma Máquina de Turing Não-Determinística, ela precisa decidir como escolher executar os Ramos de Computação Não-Determinísticos (SIPSER, 2012). Duas principais abordagens distintas e conhecidas<sup>45</sup> são a Busca em Largura<sup>46</sup> e Busca em Profundidade<sup>47</sup>. O funcionamento detalhado destes algoritmos de busca podem ser encontrados em Cormen et al. (2009).

Cada uma delas apresenta vantagens e desvantagens. Uma vantagem da Busca em Profundidade é possibilidade de “sorte”, caso o primeiro Ramo Não-Determinístico escolhido seja uma solução para o problema. Em outros tipos de problema que utilizam

<sup>44</sup> Máquinas de Turing da classe *NP*, somente existem teoricamente, com a exceção de alguns trabalhos como Lang (1974), que tenta utilizar computação paralela para representar o Não-Determinismo.

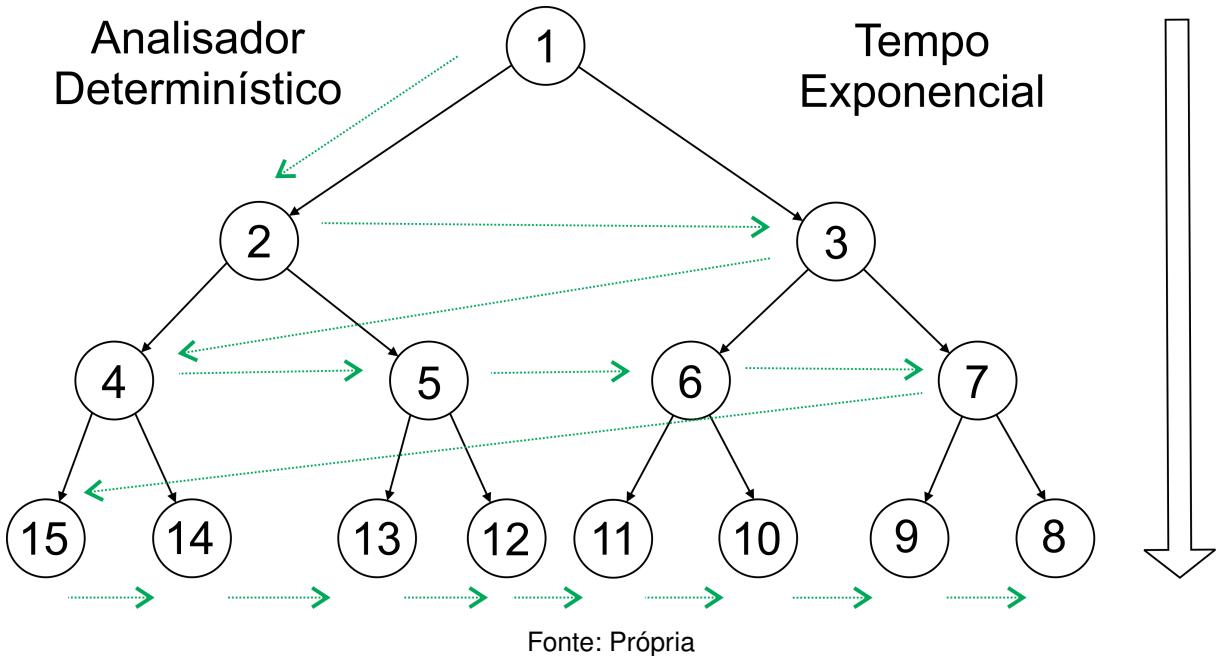
<sup>45</sup> Além dessas duas abordagens, existem muitas outras técnicas que podem ser criadas como misturas dessas duas estratégias extremas, como heurísticas e inteligências artificiais.

<sup>46</sup> Do inglês, *Breadth-First Search (BFS)*.

<sup>47</sup> Do inglês, *Depth-First Search (DFS)*.

a Busca em Profundidade, pode-se também ao contrário de “sorte”, ter o “azar” de que o primeiro Ramo de Computação seja infinito e que nunca leve a solução do problema.

Figura 14 – Árvore de computação com 15 passos<sup>48</sup> utilizando Busca em Largura



A Figura 13, mostrou um exemplo de uso do algoritmo de Busca em Profundidade. Já na Figura 14, encontra-se a variação de execução de um Analisador Determinístico que utilizou o algoritmo de Busca em Largura. Tanto o algoritmo de Busca em Largura quanto Busca em Profundidade, não precisam exatamente seguir resolvendo o problema pela esquerda ou direita. O que importa é a sua característica de avançar até o fim de algum dado Ramo de Computação, ou seguir executando todos os ramos que fazem parte de um mesmo nível de computação (CORMEN et al., 2009; SHANG; KITSUREGAWA, 2013).

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.

<sup>48</sup> Não-Determinísticos.

### 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 e gostar-se mais caso este “livro” tenha uma boa aparência. Existem livros muito horríveis de se 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, para aprender a programar pela primeira vez, 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 estejam 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 todo aglomerado ou desordenado em um único parágrafo, frase, capítulo, etc.

#### 3.1 PROGRAMAÇÃO AUXILIADA ⇐ | ←

Seu computador deveria ajudar você com aquelas imprevistas tarefas. Porque eu deveria gastar meu precioso tempo verificando se algo que estou copiando, está indentado por “TAB's” ou espaços? Talvez devêssemos sentar e chorar enquanto aguardamos que alguma força maior do Universo venha e nos salve. Ou talvez você deva parar de chorar e realmente fazer alguma coisa a respeito além de continuar esperando que sua mamãe venha e resgate você da escuridão crescendo pela suas costas levando você a infinitas noites acordado corrigindo seu código simplesmente porque tudo deu errado.

#### 3.2 FORMATADORES DE CÓDIGO ⇐ | ←

Conhecido como “*pretty-printing*” ou embelezadores<sup>1</sup> ([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

---

<sup>1</sup> Do inglês, *Source Code Beautifiers*.

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 configurações<sup>2</sup> 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 quanto difícil pode ser escrever e entender o que cada uma dessas incontáveis opções significam.

### 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

```

<sup>2</sup> Veja mais sobre configurações na [Seção 3.4: Trabalhos Relacionados](#).

```
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 3.4: 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.

Algumas opções de configuração do formatador de código-fonte são claras e fáceis de se entender. Já outras, não se consegue ter a mínima ideia do que elas estão fazendo ou qual será o seu resultado final. Por exemplo, é possível claramente entender o que a última configuração do [Código 3](#) faz?

### 3.3 QUAL A UTILIDADE DE FORMATADORES? ⇐ | ←

Um bom estilo de formatação é como uma boa pontuação, você consegue viver sem ([WICKHAM, 2017](#)):

mascomcertezornamascoisasmaisfaceisdeseler

Não pôde-se encontrar nenhuma forte evidência<sup>3</sup> ou relação sobre a compreensão de códigos-fonte e formatadores de código. Alguns estudos como [Scalabrin 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

<sup>3</sup> 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 sofre ganhos em tempo de desenvolvimento e qualidade de software”.

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 destroem 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 nossa<sup>4</sup>).

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.

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 nossa<sup>5</sup>).

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.

### 3.4 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 ferramen-

<sup>4</sup> 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.

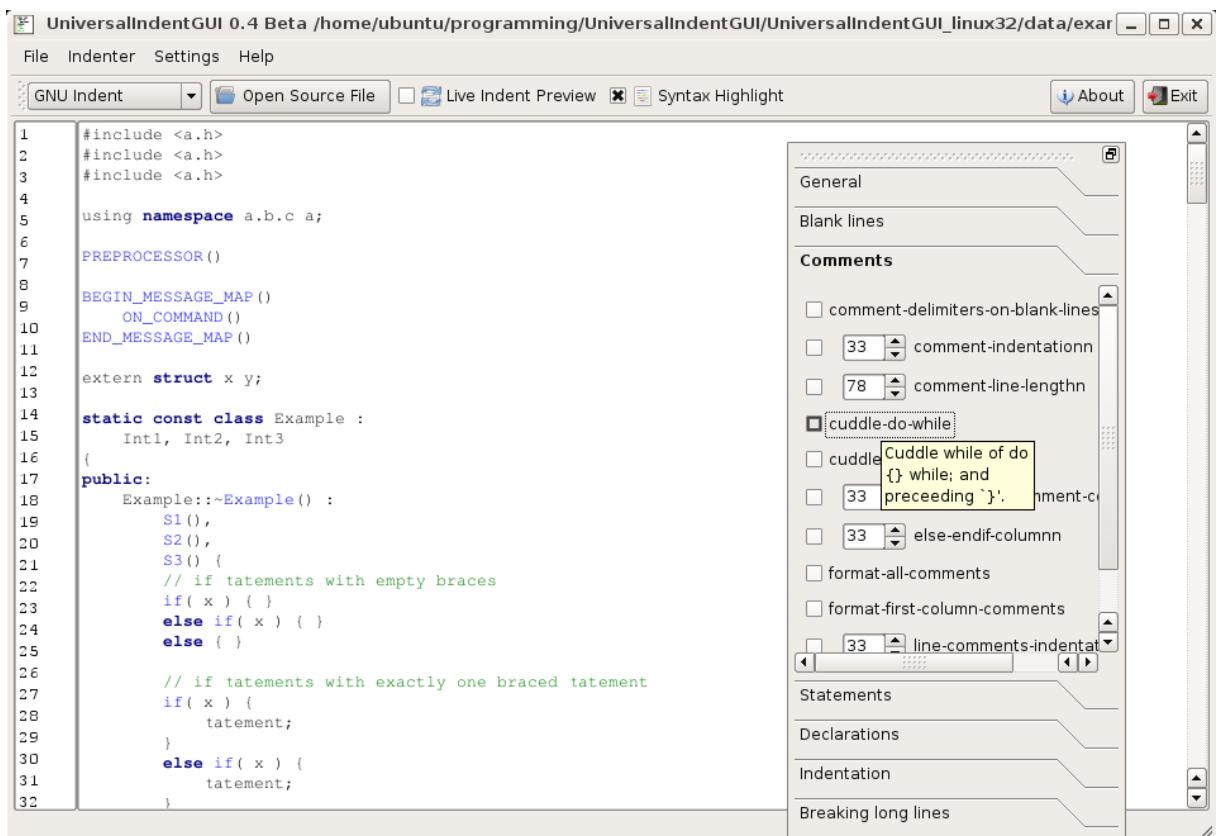
<sup>5</sup> So yes, absurd as it may sound, fighting over whitespace characters and other seemingly trivial issues of code layout is actually justified. Within reason of course – when done openly, in a fair and concensus building way, and without stabbing your teammates in the face along the way.

tas possui suas vantagens e desvantagens ([DINESH; USKUDARH, 1997; CAMERON, 1988; KIM et al., 2016](#)). Por isso, alguns desenvolvedores de software que são mais inquietos ou entusiastas por novas tecnologias 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, tema foco desta monografia. 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 15](#), pode-se ver como é a interface com o usuário que esta ferramenta proporciona.

Figura 15 – Exemplo da tela de configuração de UniversalIndentGUI



Fonte: [Schweitzer \(2007\)](#)

Nota: O nome da ferramenta é UniversalIndentGUI.

A interface da [Figura 15](#), 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 3.4: Trabalhos Relacionados](#), Uncrustify na sua versão “0.69.0” possui 671 configurações de formatação de código-fonte, mas a interface gráfica de [Schweitzer \(2006\)](#), não possui nem próximo de tantas 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 limitada interface gráfica apresenta.

Outra desvantagem da visualização das mudanças, como mostrado na [Figura 15](#), 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++”.

### 3.5 OFUSCADORES ⇐ | ←

Ofuscadores<sup>6</sup> 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. \(2008\)](#).

<sup>6</sup> Do inglês, *Obfuscator*, ([Obfuscation \(software\)](#)).

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 de documentação. Nesse arquivo, a transmissão dos comentários pode ser considerada inútil e consequentemente podem ser 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 YAML<sup>7</sup> ([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 obtusos<sup>8</sup> como “A88”.

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

<sup>8</sup> Sem significado de uso.

### 3.6 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 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 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 o-

riginais 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).

### 3.6.1 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 XML<sup>9</sup> (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 consegue especificar mais diferentes tipos de gramáticas (pelos novos operadores adicionados, como “set” e “branch” (SMITH, 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 (ODGAARD, 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

```

<sup>9</sup> Do inglês, *Extensible Markup Language*.

```

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 “*itálico*”.

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.

### 3.7 LISTA DE CURIOSIDADES ⇐ | ←

- 1 Tracking Changes, [Laster \(2016\)](#);
- 2 Aspect-oriented model-driven code generation: A systematic mapping study, [Mehmood; Jawawi \(2012\)](#);
- 3 An Annotation Assistant for Interactive Debugging of Programs with Common

- Synchronization Idioms, [Elmas et al. \(2009\)](#);
- 4 Style Analysis for Source Code Plagiarism Detection, [Mirza; Joy \(2015\)](#);
  - 5 Automated maintenance of a software portfolio, [Veerman \(2006\)](#);
  - 6 Towards automated modification of legacy assets, [Sellink; Verhoef \(2000\)](#);
  - 7 Automated Mass Maintenance of Software Assets, [Veerman \(2007\)](#);
  - 8 Pretty-printing for software reengineering, [Jonge \(2002\)](#);
  - 9 An architecture for context-sensitive formatting, [Brand; Kooiker; Vinju; Veerman \(2005\)](#);
  - 10 A Language Independent Framework for Context-sensitive Formatting, [Brand; Kooiker; Vinju \(2006\)](#);
  - 11 An industrial application of context-sensitive formatting, [Brand; Kooiker; Veerman et al. \(2005\)](#);
  - 12 Comparison of open source tools for project management, [Pereira et al. \(2013\)](#);
  - 13 Code Classification as a Learning and Assessment Exercise for Novice Programmers, [Thompson et al. \(2006\)](#);
  - 14 Code Scanning Patterns in Program Comprehension, [Aschwanden; Crosby \(2006\)](#);
  - 15 Debugging into Examples, [Steinert et al. \(2009\)](#);
  - 16 The Use of Reading Technique and Visualization for Program Understanding, [Porto; Mendonça; Fabbri \(2009\)](#);
  - 17 Documenting and Sharing Knowledge About Code, [Guzzi \(2012\)](#);
  - 18 Autofolding for Source Code Summarization, [Fowkes; Chanthirasegaran; Ranca \(2017\)](#);
  - 19 Development of a Learning Support System for Source Code Reading Comprehension, [Arai et al. \(2014\)](#);
  - 20 Syntax highlighting as an influencing factor when reading and comprehending source code, [Beelders; Plessis \(1\)](#);
  - 21 Improving code readability models with textual features, [Scalabrino et al. \(2016\)](#);
  - 22 How Novices Read Source Code in Introductory Courses on Programming: An Eye-Tracking Experiment, [Yenigalla et al. \(2016\)](#);

- 23 The Role of Method Chains and Comments in Software Readability and Comprehension; An Experiment, [Börstler; Paech \(2016\)](#);
- 24 An empirical study on code comprehension DCI compared to OO, [Valdecantos \(2016\)](#);
- 25 Enhancing legacy software system analysis by combining behavioural and semantic information sources, [Cutting \(2016\)](#);
- 26 Towards a Live, Moldable Code Editor, [Syrel \(2017\)](#);
- 27 A Comparison of Program Comprehension Strategies by Blind and Sighted Programmers, [Armaly; Rodeghero; McMillan \(2017\)](#);
- 28 2-Head Pushdown Automata, [Samson \(2015\)](#);
- 29 <https://github.com/r-lib/styler>;
- 30 <https://github.com/github/linguist>;
- 31 [https://github.com/wbond/package\\_control\\_channel/issues/4310](https://github.com/wbond/package_control_channel/issues/4310) Write a formatter utility;
- 32 <https://github.com/PyCQA/pydocstyle>;
- 33 <https://github.com/PyCQA/pycodestyle>;
- 34 <https://blog.codinghorror.com/check-in-early-check-in-often/>;
- 35 <https://www.youtube.com/watch?v=2GqpdfIAhz8>;
- 36 <http://langserver.org/>;
- 37 <https://github.com/Microsoft/language-server-protocol>;
- 38 <https://github.com/SublimeCodeIntel/SublimeCodeIntel>;
- 39 <https://code.visualstudio.com/blogs/2016/06/27/common-language-protocol>;
- 40 [https://www.eclipse.org/community/eclipse\\_newsletter/2017/may/article1.php](https://www.eclipse.org/community/eclipse_newsletter/2017/may/article1.php);
- 41 <https://github.com/Microsoft/language-server-protocol/wiki/Protocol-Implementations>.

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.

# **Parte II**

## **Implementação** ↪ | ←

## 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 metagramática<sup>1</sup> 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: 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*<sup>2</sup> (REN; ZHOU; KÄSTNER, 2018; BIAZZINI; BAUDRY, 2014; CELIŃSKA, 2018) do Analisador Lark, renomeado o Analisador Lark (SHINAN, 2014) para “pushdown”<sup>3</sup>. 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

<sup>1</sup> 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.

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

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

é encontrado o nome “pushdown” ao invés de “lark”. Já em outros, Lark continua sendo 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 (CLI<sup>4</sup>, [\(SINGER, 2017\)](#)). 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 “simples<sup>5</sup>”, capaz que pegar algum elemento com uma estrutura como “if(args.length > 0)”, de um programa de

<sup>4</sup> Do inglês, *Command Line Interface*.

<sup>5</sup> 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 Metalinguagem](#).

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. 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 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 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 Abstrata<sup>6</sup> ([Seções 4.4 e 4.5: Formatador de Código e Analisador Semântico](#)).

A Árvore Sintática Abstrata representa a estrutura do programa de uma linguagem 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 do programa e removendo-se elementos textuais como parenteses de procedência de operadores, que podem ser representados pela estrutura da árvore ([Código 12](#) e [Figura 17](#)). 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             ,
20             start='language_syntax',
21             parser="lalr",
22         )
23
24     def parseit(program, filename):
25         pushdown.tree.pydot__tree_to_png(

```

<sup>6</sup> Do inglês (AST), Abstract Syntax Tree.

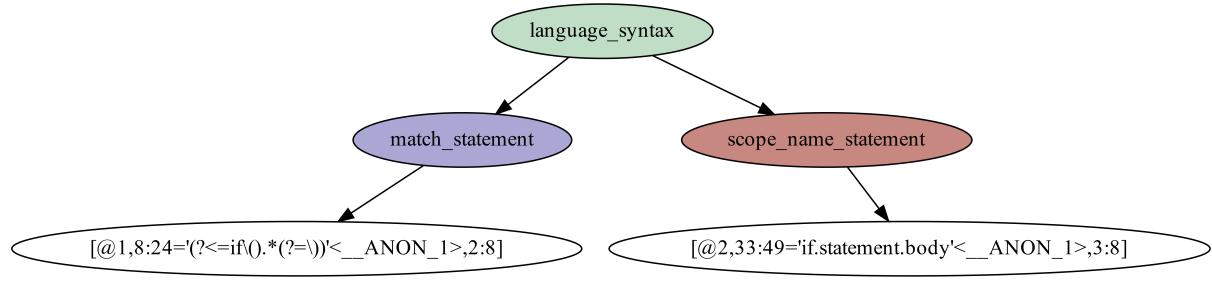
```

24         parser.parse( program ), filename, "TB", debug=1, dpi=600 )
25
26 parseit(
27     r"""
28         match: (?<=if\() .*(?=\\))
29         scope: if.statement.body
30     """
31     "arvore_da_gramatica.png"
32 )

```

Na [Figura 16](#) 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 16 – 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 3.6.1: 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.

Código 12 – Pseudo-código do algoritmo Euclidiano ([BRENT, 1976](#))

```

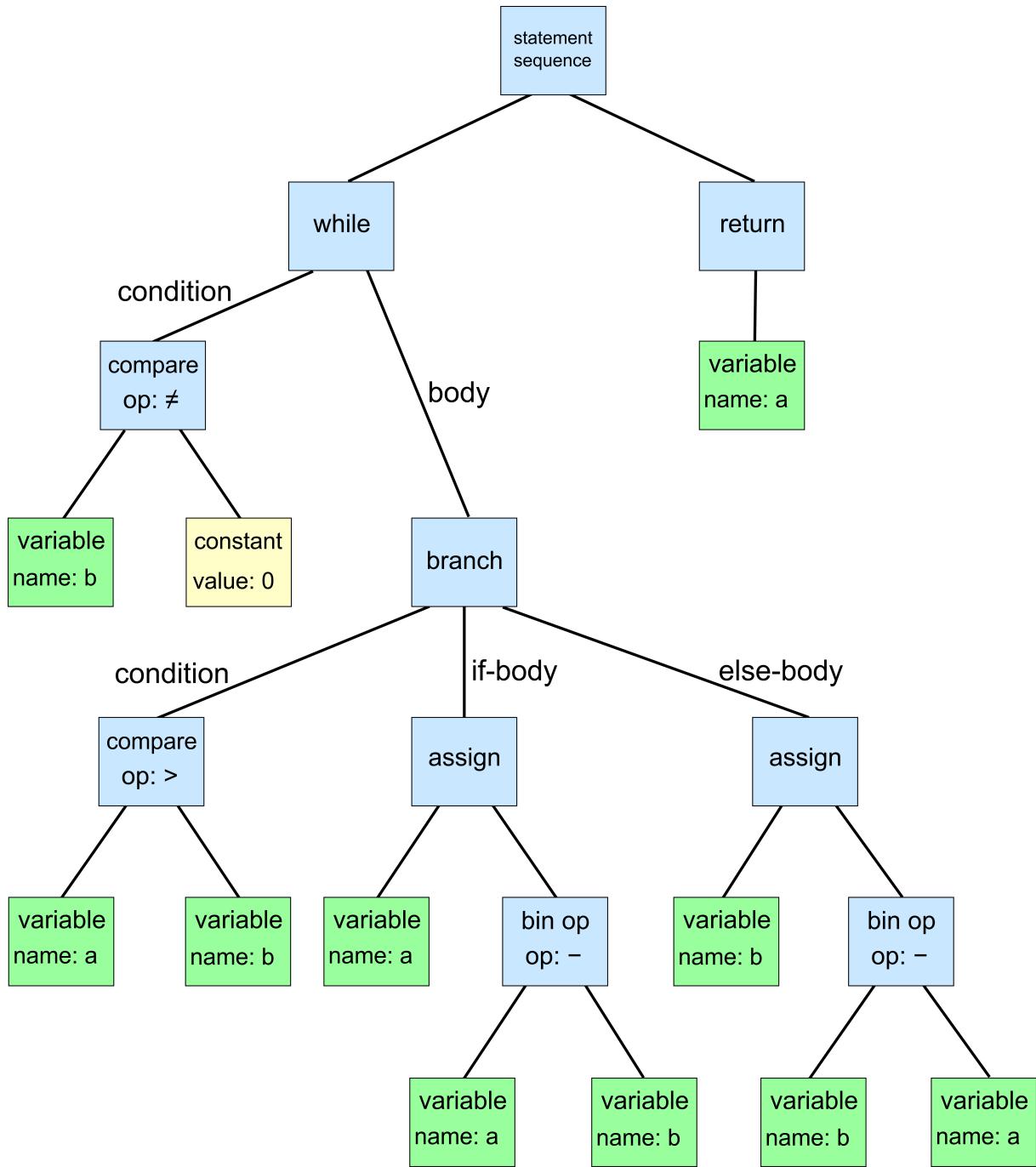
1  while b != 0
2      if a > b
3          a := a - b
4      else

```

```

5      b := b - a
6  return a

```

Figura 17 – Árvore Sintática Abstrata do [Código 12](#)

Fonte: Coetze (2011)

Ao fazer o processo de reconhecimento do programa de entrada, remove-se os caracteres reconhecidos, substituíndo eles por algum outro caractere de marcação (como “§”, símbolo de seção<sup>7</sup>) para que não altere-se o tamanho original do programa de entrada. No [Código 13](#) se pode ver como o programa original ([Código 8](#)) ficou ao

<sup>7</sup> Do inglês, section sign. Também conhecido sinal de seção ou sinal de corte.

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 13 – Resultado do reconhecimento do programa Java pela gramática

```

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 13](#)). 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 3.6.1: 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 14](#) 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 15](#).

#### Código 14 – Exemplo de configuração utilizada mínima do Formatador de Código

```

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

```
3 }
```

O [Código 15](#) 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 14](#).

#### Código 15 – 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 16](#) encontra-se o resultado da formatação 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 3.6: Adição de Cores](#)) que serviram de inspiração a este trabalho.

#### Código 16 – 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 }
```

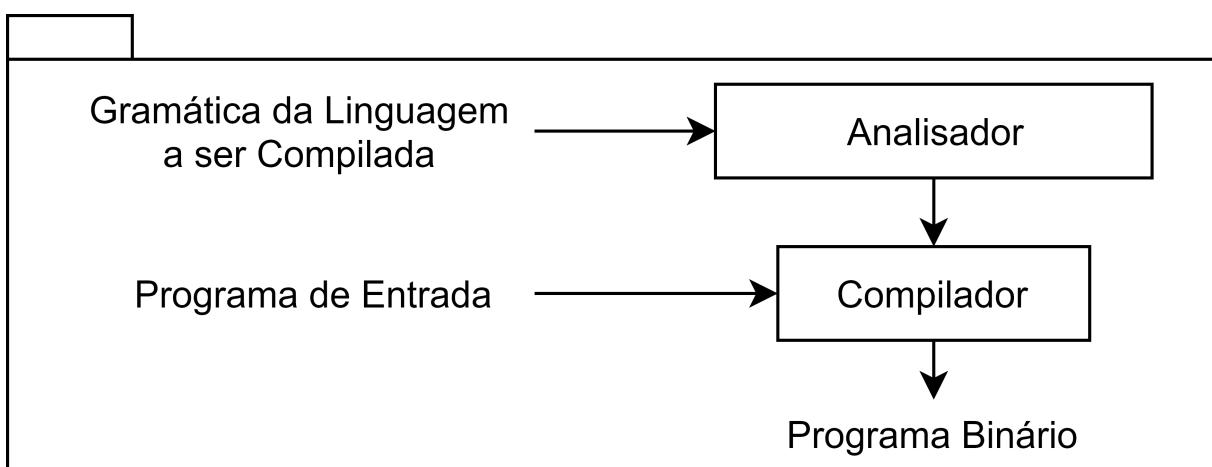
Nas próximas seções, será explicado em pouco mais de detalhes o mesmo funcionamento recém-apresentado brevemente. Para maiores detalhes, consulte a implementação e exemplos de uso destes algoritmos nos [Apêndices B e G: main\\_formatter.py e Código do Formatador](#).

## 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, sendo assim, um detalhe de implementação. Usualmente, analisadores seguem uma notação comum como EBNF ([PATTIS, 1994; PARR, 2013; SHINAN, 2019b; 2018](#)), que não difere muito de um analisador para outro, exceto por detalhes de implementação específicos de cada analisador.

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 18](#), é apresentado o fluxo de uso comum para um analisador. Neste processo, o desenvolvedor da linguagem escreve a gramática de especificação dessa linguagem que é entregue a algum analisador e gera-se um compilador para tal linguagem.

Figura 18 – Fluxo de uso comum de um analisador



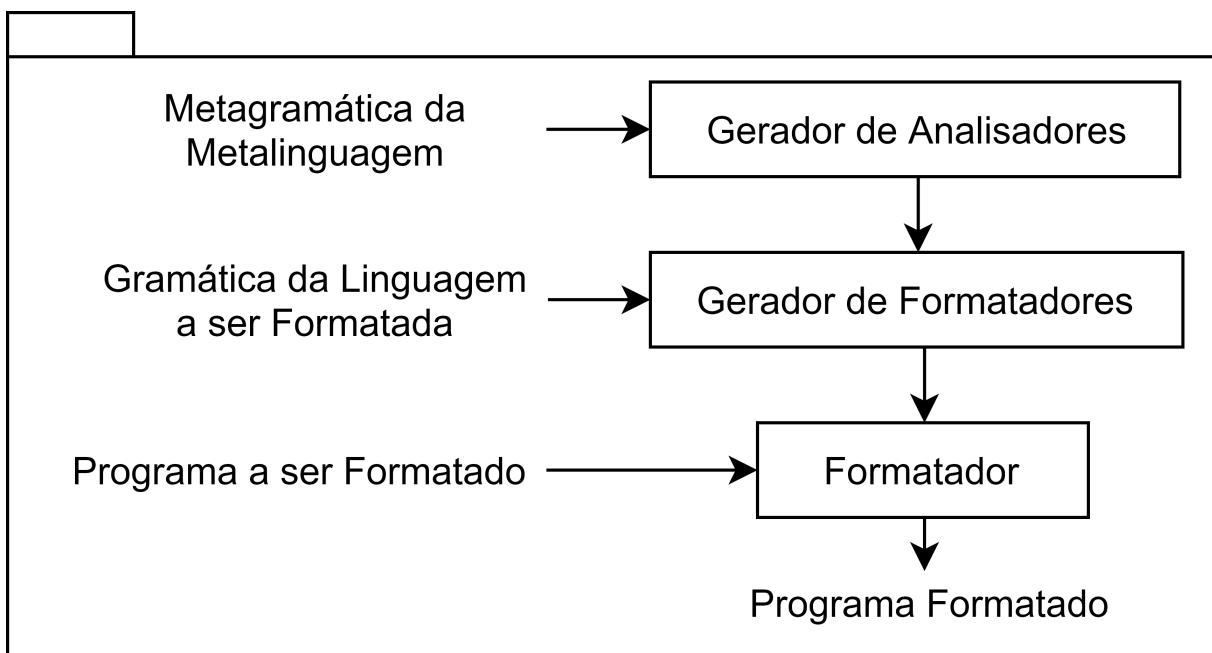
Fonte: Própria, traduzido e adaptado de [Coan \(2018\)](#)

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 18), 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). Já este trabalho faz um uso fora do comum. Como mostrado na Figura 19, 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 19 – Uso feito pela nova ferramenta de Formatação de Código



Fonte: Própria, traduzido e adaptado de [Coan \(2018\)](#)

Diferente da Figura 18, na Figura 19 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 Formadores”, que é a criação de um novo tipo de Analisador de propósito específico, para uma DSL<sup>8</sup> (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 gramáticas<sup>9</sup>, baseada nas gramáticas já utilizadas por editores de texto para adicionar cores aos códigos-fontes exibidos em sua interface gráfica ([Seção 3.6.1: Gramáticas](#)). Diferente das gramáticas 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-fonte<sup>10</sup> (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 3.6: Adição de Cores](#)).

Usualmente, o Analisador Lark é utilizado somente como um gerador de compiladores ([Figura 18](#)), entretanto, neste contexto Lark é utilizado como um “Compilador de Compiladores” ([Figura 19](#)). A vantagem de utilizar outro analisador (Lark) para criar outro analisador (Gerador de Formadores), é que a gramática de especificação do Gerador de Formadores 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 20](#), 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.

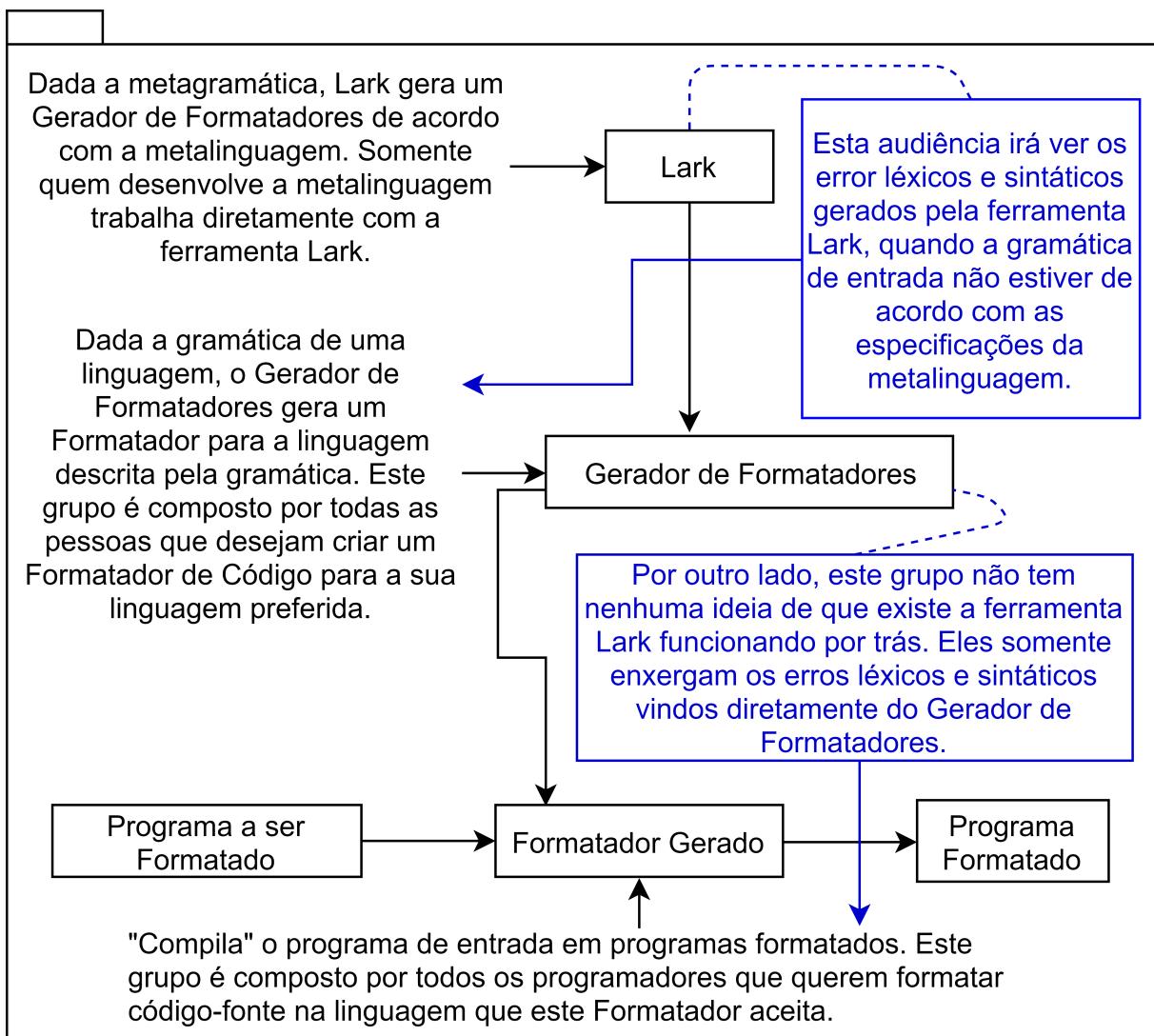
Até este ponto, já falou-se de metagramática e metalinguagem com a exceção dos metaprogramas ([VELDHUIZEN, 2006](#)). Nas [Figuras 19](#) e [20](#), por simplificação foram omitidos o relacionamento dos metaprogramas com a metagramática e metalinguagem. Metaprogramas fazem parte da entrada do metacompilador ([Figura 21](#))

<sup>8</sup> Do inglês, *Domain-Specific Language*.

<sup>9</sup> 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 ([KIRPICOV, 2014](#)).

<sup>10</sup> A final, 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 20 – Relacionamentos entre os diferentes públicos deste projeto



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

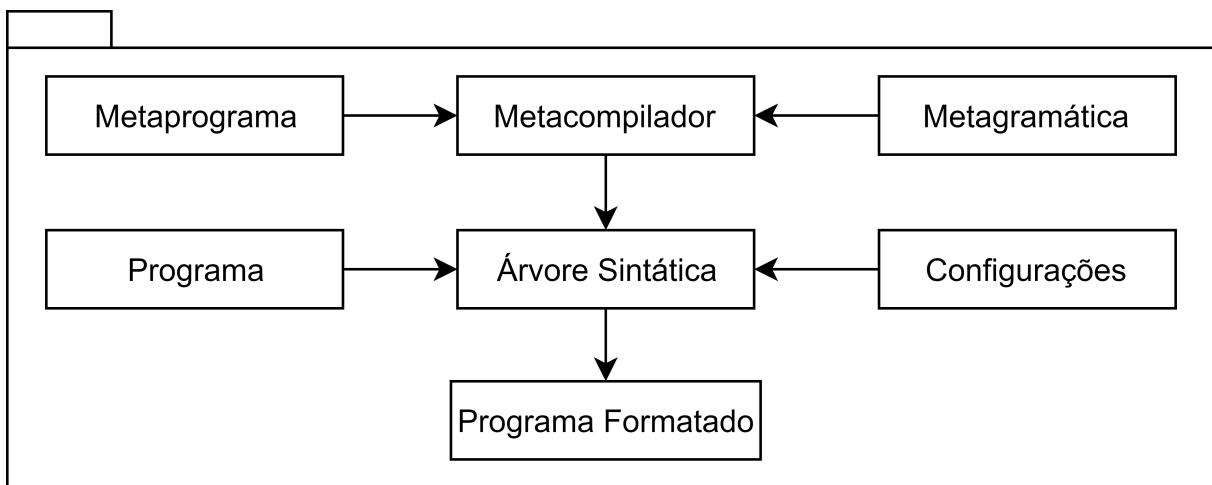
junto com a metagramática para gerar um novo compilador (ou Formatador de Código). Neste trabalho, os metaprogramas serão as gramáticas que serão utilizadas pelos formatadores de código-fonte.

Os metaprogramas (ou gramáticas) são entradas diretas do metacompilador, o Analisador Lark na Figura 19, utilizado para analisar uma gramática LALR(1) (Apêndice H: Código da Metagramática). Na Figura 20, 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 21, encontra-se uma extensão da Figura 19, e pode-se ver claramente as relações entre Metagramáticas, Metacompiladores e Metaprogramas. Por simplifi-

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.

Figura 21 – Relação entre Metagramáticas, Metacompiladores e Metaprogramas



Fonte: Própria, traduzido e adaptado de [Coan \(2018\)](#)

Esta não é a primeira vez que uma metagramática com simplificações foi escrita. Em trabalhos relacionados com esta implementação de metagramática ([Seção 3.6: 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](#)).

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 JSON<sup>11</sup> ([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

<sup>11</sup> Do inglês, *JavaScript Object Notation*.

características de outras linguagens como YAML, XML ou JSON.

Na [Figura 21](#), percebe-se que os Metaprogramas (ou gramáticas) são entradas diretas dos Metacompilador, e não do Formatador de Código como mostrado nas figuras anteriores ([Figuras 18 a 20](#)). Ambas as figuras estão corretas, entretanto, a [Figura 21](#) 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 21](#). 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 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 analisada<sup>12</sup>. 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 17](#), a metagramática define que o programa é constituído de três grandes áreas, que devem acontecer uma em sequência da outra:

<sup>12</sup> 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](#)).

- 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 contextos<sup>13</sup> 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 17 – Símbolo Inicial da Metagramática “ObjectBeauty” ([Apêndice H](#))

```

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

```

---

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

27 ...

Entre os [Códigos 18 a 21](#), encontram-se pequenos exemplos de gramáticas escritas na metalinguagem “ObjectBeauty” brevemente apresentada. No [Código 18](#), 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 3.6.1: Gramáticas](#), na [Figura 22](#) 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 22 – Exemplo de classificação de código-fonte com múltiplos escopos**

function	f1	()	{
		source.js	
		meta.function.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 22](#), 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 3.6.1: Gramáticas](#)). Para mais informações sobre a utilização de arquivos de temas em editores de texto veja [Skinner \(2015b\)](#) e [Dima \(2017\)](#).

O [Código 18](#), 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 18](#), 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 18 – 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 19](#), 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” atribui 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 19 – 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 20](#), 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 20 – 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 21](#), 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 21 – 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 22](#), 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 22](#), 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 22 – Exemplo de Gramática, Reconhecimento de Erros

```

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

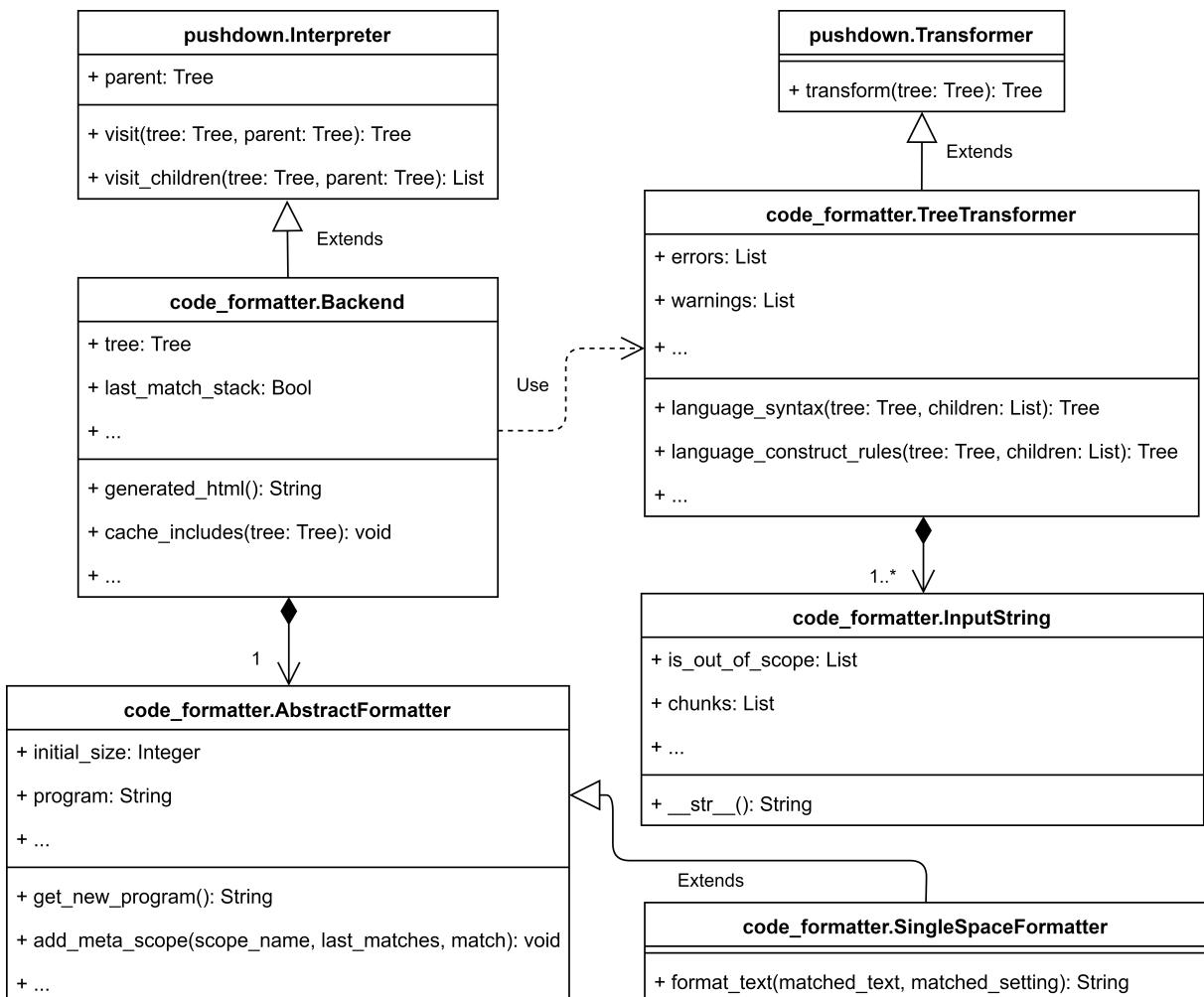
## 4.4 ANALISADOR SEMÂNTICO ⇐ | ←

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 des-

crita pelo metaprograma ([Figura 21](#)). 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 23](#). 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 20 e 21](#)).

Figura 23 – Diagrama das principais classes



Fonte: Própria

No [Código 23](#), 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 23 – Construtor do Analisador Semântico ([Apêndice E](#))

```
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 = {}
```

```

35      ## A list of regular expressions used on match statements,
36      ## for validation when the constants definitions are completely known
37      self.pending_match_statements = []
38
39      ## Responsible for calculating all open and close commands scoping
40      self.open_blocks = {}
41      self.indentation_level = 0
42      self.indentation_blocks = []

```

A visita dos nós da Árvore Sintática pela função “transform(tree)” acontece simplesmente chamando os métodos que a classe “TreeTransformer” ([Código 23](#)) 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 17](#), 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 Sémântica (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 3.4: 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. Imagine como seria escrever uma única gramática que represente todas as linguagens de programação?

No [Código 24](#), pode ser encontrado o construtor do formatador de código-fonte. Comparando-o com o construtor do Analisador Semântico ([Código 23](#)), 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 24 – 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]
26
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 23). 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 3.6.1: 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 “passear” pela Árvore Sintática Abstrata, a classe “AbstractFormatter” (Código 25) 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 25 – 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     self.new_program = []
14     self.cached_new_program = []
15     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 formatação<sup>14</sup>.

Durante o processo de consumo, o programa de entrada é modificado e os caracteres que foram consumidos pela gramática são substituídos por algum caractere qualquer<sup>15</sup> e fixo como “§” (*símbolo de seção*) para impedir que eles sejam reconsumidos. 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 metaprogramas, 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.

---

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

<sup>15</sup> 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.

## 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 3.4: 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 3.6: 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 prever 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 3.4: 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 parâme-

tro 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 3.4: 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 ⇐ | ←

O autor desse trabalho sugere alguns trabalhos futuros em [Coan \(2016c\)](#) e [Coan \(2016b\)](#). Antes que novos formatadores de código-fonte sejam implementados, estes pontos precisam ser revistos. Ao realizar estas mudanças, qualquer implementação já realizada para formatação será perdida devido ao número de alterações realizadas, como em um efeito borboleta ([TALIRONGAN; SISON; MEDINA, 2018](#)).

Caso exista, o melhor trabalho futuro que pode ser feito é uma reinvenção 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 15](#), 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 [S-kinner \(2015c\)](#) 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 3.6](#) 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.

Em relação aos formatadores gerados pelo gerador de formatadores, eles precisaram ser completamente reescritos, uma vez que as melhorias feitas no gerador de formatadores e metalinguagem linguagem forem concluídas. A implementação atual dos formatadores pode ser considerada nula, uma vez que toda sua lógica de funcionamento é construída como um simples iteração descendente pela Árvore Sintática Abstrata gerada pelo Analisador Semântico.

## REFERÊNCIAS ⇐ | ←

ABRAMS, Daniel S.; LLOYD, Seth. Nonlinear Quantum Mechanics Implies Polynomial-Time Solution for NP-Complete and #P Problems. **Phys. Rev. Lett.**, American Physical Society, v. 81, p. 3992–3995, nov. 1998. DOI: [10.1103/PhysRevLett.81.3992](https://doi.org/10.1103/PhysRevLett.81.3992). Disponível em: <<https://link.aps.org/doi/10.1103/PhysRevLett.81.3992>>. Citado uma vez na página 40.

**Resumo:** If quantum states exhibit small nonlinearities during time evolution, then quantum computers can be used to solve NP-complete and #P problems in polynomial time. We provide algorithms that solve NP-complete and #P oracle problems by exploiting nonlinear quantum logic gates. Using the Weinberg model as a simple example, the explicit construction of these gates is derived from the underlying physics. Nonlinear quantum algorithms are also presented using Polchinski type nonlinearities which do not allow for superluminal communication.

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 2 vezes nas páginas 53, 89.

**Resumo:** According to the documentation for TextMate scope selectors, VSCode supports the syntax for excluding matching scopes: entity.name.method - source.java matches all scopes that start with entity.name.method but not if a parent scope matches source.java. This functionality is used in at least one built-in theme that I could find: <https://github.com/Microsoft/vscode/blob/19ef61e5/extensions/theme-monokai/themes/monokai-color-theme.json#L316> But it seems that this syntax makes the selector with - in it invalid, and VSCode doesn't apply the given rule to anything at all.

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 15 vezes nas páginas 20, 21, 31–36, 38, 77.

**Resumo:** Compilers: Principles, Techniques and Tools, known to professors, students, and developers worldwide as the "Dragon Book," is available in a new edition. Every chapter has been completely revised to reflect developments in software engineering, programming languages, and computer architecture that have occurred since 1986, when the last edition published. The authors, recognizing that few readers will ever go on to construct a compiler, retain their focus on the broader set of problems faced in software design and software development.

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 uma vez nas páginas [27](#), [29](#).

**Resumo:** This book is intended for a one or two semester course in compiling theory at the senior or graduate level. It is a theoretically oriented treatment of a practical subject. Our motivation for making it so is threefold. (1) In an area as rapidly changing as Computer Science, sound pedagogy demands that courses emphasize ideas, rather than implementation details. It is our hope that the algorithms and concepts presented in this book will survive the next generation of computers and programming languages, and that at least some of them will be applicable to fields other than compiler writing. (2) Compiler writing has progressed to the point where many portions of a compiler can be isolated and subjected to design optimization. It is important that appropriate mathematical tools be available to the person attempting this optimization. (3) Some of the most useful and most efficient compiler algorithms, e.g. LR(k) parsing, require a good deal of mathematical background for full understanding. We expect, therefore, that a good theoretical background will become essential for the compiler designer. While we have not omitted difficult theorems that are relevant to compiling, we have tried to make the book as readable as possible. Numerous examples are given, each based on a small grammar, rather than on the large grammars encountered in practice. It is hoped that these examples are sufficient to illustrate the basic ideas, even in cases where the theoretical developments are difficult to follow in isolation.

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](https://doi.org/10.1109/CACWD.2004.1349274). Citado uma vez na página [20](#).

**Resumo:** General intensional programming compiler (GIPC) is one component of the general intensional programming system (GIPSY) that aims at the development of a programming system that would allow dynamic investigations on the possibilities of intensional programming and all its widely different flavors and domains of application. To cope with the constant evolution of intensional programming languages, we design the system in a very flexible infrastructure for the generation of compiler components upon the creation of a new version. This work focuses on one and most important component that is about the generation of the translator between generic and specific intensional programming languages.

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](https://doi.org/10.1145/2635868.2635883). Disponível

em: <[https://www.researchgate.net/publication/260250447\\_Learning\\_Natural\\_Coding\\_Conventions](https://www.researchgate.net/publication/260250447_Learning_Natural_Coding_Conventions)>. Acesso em: 27 out. 2017. Citado uma vez na página 14.

**Resumo:** Every programmer has a characteristic style, ranging from preferences about identifier naming to preferences about object relationships and design patterns. Coding conventions define a consistent syntactic style, fostering readability and hence maintainability. When collaborating, programmers strive to obey a project's coding conventions. However, one third of reviews of changes contain feedback about coding conventions, indicating that programmers do not always follow them and that project members care deeply about adherence. Unfortunately, programmers are often unaware of coding conventions because inferring them requires a global view, one that aggregates the many local decisions programmers make and identifies emergent consensus on style. We present NATURALIZE, a framework that learns the style of a codebase, and suggests revisions to improve stylistic consistency. NATURALIZE builds on recent work in applying statistical natural language processing to source code. We apply NATURALIZE to suggest natural identifier names and formatting conventions. We present four tools focused on ensuring natural code during development and release management, including code review. NATURALIZE achieves 94% accuracy in its top suggestions for identifier names. We used NATURALIZE to generate 18 patches for 5 open source projects: 14 were accepted.

ARAI, Tatsuya *et al.* Development of a Learning Support System for Source Code Reading Comprehension. In: **Human Interface and the Management of Information, 16th International Conference**. Edição: Sakae Yamamoto. Heraklion, Crete, Greece: Springer, jun. 2014. p. 12–19. ISBN 978-3-319-07863-2. DOI: [10.1007/978-3-319-07863-2\\_2](https://doi.org/10.1007/978-3-319-07863-2_2). Disponível em: <[https://www.researchgate.net/publication/295290682\\_Development\\_of\\_a\\_Learning\\_Support\\_System\\_for\\_Source\\_Code\\_Reading\\_Comprehension](https://www.researchgate.net/publication/295290682_Development_of_a_Learning_Support_System_for_Source_Code_Reading_Comprehension)>. Acesso em: 1 nov. 2017. Citado uma vez na página 56.

**Resumo:** In this paper, we describe the development of a support system that facilitates the process of learning computer programming through the reading of computer program source code. Reading code consists of two steps: reading comprehension and meaning deduction. In this study, we developed a tool that supports the comprehension of a program's reading. The tool is equipped with an error visualization function that illustrates a learner's mistakes and makes them aware of their errors. We conducted experiments using the learning support tool and confirmed that the system is effective.

ARMALY, A.; RODEGHERO, P.; MCMILLAN, C. A Comparison of Program Comprehension Strategies by Blind and Sighted Programmers. **IEEE Transactions on Software Engineering**, IEEE Computer Society, University of Notre Dame, Notre

Dame, Indiana, United States, PP, n. 99, p. 1–1, jul. 2017. ISSN 0098-5589. DOI: [10.1109/TSE.2017.2729548](https://doi.org/10.1109/TSE.2017.2729548). Disponível em: <[https://www.researchgate.net/publication/318576489\\_Blindness\\_and\\_Program\\_Comprehension](https://www.researchgate.net/publication/318576489_Blindness_and_Program_Comprehension)>. Acesso em: 2 nov. 2017. Citado uma vez na página 57.

**Resumo:** Programmers who are blind use a screen reader to speak source code one word at a time, as though the code were text. This process of reading is in stark contrast to sighted programmers, who skim source code rapidly with their eyes. At present, it is not known whether the difference in these processes has effects on the program comprehension gained from reading code. These effects are important because they could reduce both the usefulness of accessibility tools and the generalizability of software engineering studies to persons with low vision. In this paper, we present an empirical study comparing the program comprehension of blind and sighted programmers. We found that both blind and sighted programmers prioritize reading method signatures over other areas of code. Both groups obtained an equal and high degree of comprehension, despite the different reading processes.

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](https://doi.org/10.1145/2993318.2993349). Disponível em: <[https://www.researchgate.net/publication/309430151\\_Quit\\_Diff\\_Calculating\\_the\\_Delta\\_Between\\_RDF\\_Datasets\\_Under\\_Version\\_Control](https://www.researchgate.net/publication/309430151_Quit_Diff_Calculating_the_Delta_Between_RDF_Datasets_Under_Version_Control)>. Acesso em: 3 nov. 2017. Citado uma vez na página 16.

**Resumo:** Distributed actors working on a common RDF dataset regularly encounter the issue to compare the status of one graph with another or generally to synchronize copies of a dataset. A versioning system helps to synchronize the copies of a dataset, combined with a difference calculation system it is also possible to compare versions in a log and to determine, in which version a certain statement was introduced or removed. In this demo we present Quit Diff1, a tool to compare versions of a Git versioned quad store, while it is also applicable to simple unversioned RDF datasets. We are following an approach to abstract from differences on a syntactical level to differences on the level of the RDF data model, while we leave further semantic interpretation on the schema and instance level to specialized applications. Quit Diff can generate patches in various output formats and can be directly integrated in the distributed version control system Git which provides a foundation for a comprehensive co-evolution work flow on RDF datasets.

ARORA, Sanjeev; BARAK, Boaz. **Computational Complexity: A Modern Approach**.

1st. New York, NY, USA: Cambridge University Press, 2009. ISBN 0521424267, 9780521424264. Citado 2 vezes nas páginas [29](#), [39](#), [41](#).

**Resumo:** This beginning graduate textbook describes both recent achievements and classical results of computational complexity theory. Requiring essentially no background apart from mathematical maturity, the book can be used as a reference for self-study for anyone interested in complexity, including physicists, mathematicians, and other scientists, as well as a textbook for a variety of courses and seminars. More than 300 exercises are included with a selected hint set.

ASCHWANDEN, Christoph; CROSBY, Martha. Code Scanning Patterns in Program Comprehension. **Proceedings of the 39th Annual Hawaii International Conference on System Sciences**, Online, University of Hawaii at Manoa, jan. 2006. Disponível em: <[https://www.researchgate.net/publication/250718584\\_Code\\_Scanning\\_Patterns\\_in\\_Program\\_Comprehension](https://www.researchgate.net/publication/250718584_Code_Scanning_Patterns_in_Program_Comprehension)>. Acesso em: 31 out. 2017. Citado uma vez na página [56](#).

**Resumo:** Various publications have identified Beacons to play a key role in program comprehension. Beacons are code fragments that help developers comprehend programs. It has been shown that expert programmers pay more attention to Beacons than novices. Beacons are described as the link between source code and hypothesis verification. Beacons are sets of key features that typically indicate the presence of a particular data structure or operation in source code. However, only little research has been done trying to identify and explain them in greater detail. It has been demonstrated that good variable and procedure names help in program comprehension. Documentation is beneficial as well. The so-called swap operation for variables is a strong indicator for a sorting algorithm. We conducted an eye tracking study using the EventStream software framework as the instrument to investigate programmers' behavior during a code reading exercise. Preliminary results suggest Beacons to be present when the longest fixation duration is thousand milliseconds or higher. Comparing participants with correct understanding versus participants with wrong understanding showed differences in focus of attention. Based on the study conducted, we suggest to consider "int k=(a+b)/2" as Beacons during program comprehension as well as lines of code which exhibit very long fixations above 1000 milliseconds.

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](https://doi.org/10.1109/TSE.2002.1019478). Disponível em: <<http://ieeexplore.ieee.org/document/1019478/>>. Acesso em: 3 nov. 2017. Citado uma vez na página [17](#).

**Resumo:** Software tools can improve the quality and maintainability of software,

but are expensive to acquire, deploy, and maintain, especially in large organizations. We explore how to quantify the effects of a software tool once it has been deployed in a development environment. We present an effort-analysis method that derives tool usage statistics and developer actions from a project's change history (version control system) and uses a novel effort estimation algorithm to quantify the effort savings attributable to tool usage. We apply this method to assess the impact of a software tool called VE, a version-sensitive editor used in Bell Labs. VE aids software developers in coping with the rampant use of certain preprocessor directives (similar to #if/#endif in C source files). Our analysis found that developers were approximately 40 percent more productive when using VE than when using standard text editors.

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 2 vezes na página 48.

**Resumo:** Ah, spring. What a wonderful time of year. A time when young programmers' minds turn to thoughts of ... never ending last-man-standing filibuster arguments about code formatting. Naturally. And there is no argument more evergreen than the timeless debate between tabs and spaces.

ATWOOD, Jeff. **Who Wrote This Crap?** 2009. Disponível em: <<https://blog.codinghorror.com/who-wrote-this-crap/>>. Acesso em: 1 mar. 2017. Citado uma vez na página 17.

**Resumo:** I first read this in the original 1993 edition of Code Complete. It's quoted from a much earlier book, Stan Kelley-Bootle's The Devil's Dp Dictionary, which was published in 1981. It's still true, more than 25 years later. There's a knee-jerk predisposition to look at code you didn't write, and for various reasons large and small, proclaim it absolute crap. But figuring out who's actually responsible for that crappy code takes some detective work.

BAGGE, Anya Helee; HASU, Tero. A Pretty Good Formatting Pipeline. In: **Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings**. Edição: Martn Erwig, Richard F. Paige e Eric Van Wyk. Bergen Language Design Laboratory, Dept. of Informatics, University of Bergen, Norway: Springer, out. 2013. p. 177–196. ISBN 978-3-319-02654-1. DOI: [10.1007/978-3-319-02654-1\\_10](https://doi.org/10.1007/978-3-319-02654-1_10). Disponível em: <[https://www.researchgate.net/publication/300351526\\_A\\_Pretty\\_Good\\_Formatting\\_Pipeline](https://www.researchgate.net/publication/300351526_A_Pretty_Good_Formatting_Pipeline)>. Acesso em: 31 out. 2017. Citado uma vez na página 15.

**Resumo:** Proper formatting makes the structure of a program apparent and aids program comprehension. The need to format code arises in code generation and transformation, as well as in normal reading and editing situations. Commonly

used pretty-printing tools in transformation frameworks provide an easy way to produce indented code that is fairly readable for humans, without reaching the level of purpose-built reformatting tools, such as those built into IDEs. This paper presents a library of pluggable components, built to support style-based formatting and reformatting of code, and to enable further experimentation with code formatting.

BAGGE, O. S.; KALLEBERG, K. T. *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](https://doi.org/10.1109/SCAM.2003.1238032). Citado uma vez na página 14.

**Resumo:** The use of a high-level, abstract coding style can greatly increase developer productivity. For numerical software, this can result in drastically reduced run-time performance. High-level, domain-specific optimisations can eliminate much of the overhead caused by an abstract coding style, but current compilers have poor support for domain-specific optimisation. We present CodeBoost, a source-to-source transformation tool for domain-specific optimisation of C++ programs. CodeBoost performs parsing, semantic analysis and pretty-printing, and transformations can be implemented either in the Stratego program transformation language, or as user-defined rewrite rules embedded within the C++ program. CodeBoost has been used with great success to optimise numerical applications written in the Sophus high-level coding style. We discuss the overall design of the CodeBoost transformation framework, and take a closer look at two important features of CodeBoost: user-defined rules and totem annotations. We also show briefly how CodeBoost is used to optimise Sophus code, resulting in applications that run twice as fast, or more.

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 uma vez na página 37.

**Resumo:** LR parsers can't handle ambiguous grammar rules, by design. (Made the theory easier back in the 1970s when the ideas were being worked out).

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](https://doi.acm.org/10.1145/322344.322350). Disponível em: <<http://doi.acm.org/10.1145/322344.322350>>. Citado uma vez nas páginas 32, 33.

**Resumo:** It is shown that every p-reduced LL(1) grammar is LALR(1) and, as a corollary, that every A-free LL(1) grammar is SLR(1). A partial converse to this result is also demonstrated: If there is at most one marked rule m the basis of

every state set in the canonical collection of sets of LR(k) items for a grammar G in which  $S \Rightarrow^* S_y$  impossible, then G is LL(k).

BEELDERS, Tanya R.; PLESSIS, Jean-Pierre L. du. Syntax highlighting as an influencing factor when reading and comprehending source code. *Journal of Eye Movement Research*, University of the Free State, Bloemfontein, Free State, South Africa, v. 9, n. 1, 1. ISSN 1995-8692. Disponível em: <<https://bop.unibe.ch/index.php/JEMR/article/view/2429>>. Acesso em: 1 nov. 2017. Citado uma vez na página 56.

**Resumo:** Syntax highlighting or syntax colouring, plays a vital role in programming development environments by colour-coding various code elements differently. The supposition is that this syntax highlighting assists programmers when reading and analysing code. However, academic text books are largely only available in black-and-white which could influence the comprehension of novice and beginner programmers. This study investigated whether student programmers experience more difficulty in reading and comprehending source code when it is presented without syntax highlighting. Number of fixations, fixation durations and regressions were all higher for black-and-white code than for colour code but not significantly so. Subjectively students indicated that the colour code snippets were easier to read and more aesthetically pleasing. Based on the analysis it could be concluded that students do not experience significantly more difficulty when reading code in black-and-white as printed in text books.

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 3 vezes nas páginas 17, 60, 86.

**Resumo:** Version Control Systems (VCS) are frequently used to support development of large-scale software projects. A typical VCS repository of a large project can contain various intertwined branches consisting of a large number of commits. If some kind of unwanted behaviour (e.g. a bug in the code) is found in the project, it is desirable to find the commit that introduced it. Such commit is called a regression point. There are two main issues regarding the regression points. First, detecting whether the project after a certain commit is correct can be very expensive as it may include large-scale testing and/or some other forms of verification. It is thus desirable to minimise the number of such queries. Second, there can be several regression points preceding the actual commit; perhaps a bug was introduced in a certain commit, inadvertently fixed several commits later, and then reintroduced in a yet later commit. In order to fix the actual commit it is usually desirable to find the latest regression point. The currently used distributed

VCS contain methods for regression identification, see e.g. the git bisect tool. In this paper, we present a new regression identification algorithm that outperforms the current tools by decreasing the number of validity queries. At the same time, our algorithm tends to find the latest regression points which is a feature that is missing in the state-of-the-art algorithms. The paper provides an experimental evaluation of the proposed algorithm and compares it to the state-of-the-art tool git bisect on a real data set.

BERG, André; GARDNER, Ben; MAUREL, Guy. **Uncrustify**. 2005. Disponível em: <<http://uncrustify.sourceforge.net/>>. Acesso em: 26 jul. 2019. Citado uma vez na página 48.

**Resumo:** The goals of this project are simple: Create a highly configurable, easily modifiable source code beautifier. Features: 1. Indent code, aligning on parens, assignments, etc, 2. Align on '=' and variable definitions, 3. Align structure initializers, 4. Align #define stuff, 5. Align backslash-newline stuff, 6. Reformat comments (a little bit), 7. Fix inter-character spacing, 8. Add or remove parens on return statements, 9. Add or remove braces on single-statement if/do/while/for statements, 10. Supports embedded SQL 'EXEC SQL' stuff, 11. Highly configurable - 671 configurable options as of version 0.69.0.

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 3 vezes nas páginas 46, 48, 50.

**Resumo:** The goals of this project are simple: Create a highly configurable, easily modifiable source code beautifier. Features: 1. Indent code, aligning on parens, assignments, etc, 2. Align on '=' and variable definitions, 3. Align structure initializers, 4. Align #define stuff, 5. Align backslash-newline stuff, 6. Reformat comments (a little bit), 7. Fix inter-character spacing, 8. Add or remove parens on return statements, 9. Add or remove braces on single-statement if/do/while/for statements, 10. Supports embedded SQL 'EXEC SQL' stuff, 11. Highly configurable - 671 configurable options as of version 0.69.0.

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](https://doi.org/10.1145/2593868.2593875). Disponível em: <<http://doi.acm.org/10.1145/2593868.2593875>>. Citado uma vez na página 60.

**Resumo:** Multi-repository software projects are becoming more and more popular, thanks to web-based facilities such as Github. Code and process metrics generally assume a single repository must be analyzed, in order to measure the charac-

teristics of a codebase. Thus they are not apt to measure how much relevant information is hosted in multiple repositories contributing to the same codebase. Nor can they feature the characteristics of such a distributed development process. We present a set of novel metrics, based on an original classification of commits, conceived to capture some interesting aspects of a multi-repository development process. We also describe an efficient way to build a data structure that allows to compute these metrics on a set of Git repositories. Interesting outcomes, obtained by applying our metrics on a large sample of projects hosted on Github, show the usefulness of our contribution.

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](https://doi.org/10.1145/954344.954345). Disponível em: <<http://doi.acm.org/10.1145/954344.954345>>. Citado 3 vezes nas páginas 19, 68, 69.

**Resumo:** Cwic/360 (Compiler for Writing and Implementing Compilers) is a metacompiler system. It is composed of compilers for three special-purpose languages, each intended to permit the description of certain aspects of translation in a straightforward, natural manner. The Syntax language is used to describe the recognition of source text and the construction from it of an intermediate tree structure. The Generator language is used to describe the transformation of the tree into appropriate object language. The MOL/360 language is used to provide an interface with the machine and its operating system. This paper describes each of these languages, presents examples of their use, and discusses the philosophy underlying their design and implementation.

BÖRSTLER, J.; PAECH, B. The Role of Method Chains and Comments in Software Readability and Comprehension; An Experiment. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Department of Software Engineering, Blekinge Institute of Technology, Karlskrona, Sweden, v. 42, n. 9, p. 886–898, set. 2016. ISSN 0098-5589. DOI: [10.1109/TSE.2016.2527791](https://doi.org/10.1109/TSE.2016.2527791). Disponível em: <[https://www.researchgate.net/publication/294119566\\_The\\_Role\\_of\\_Method\\_Chains\\_and\\_Comments\\_in\\_Software\\_Readability\\_and\\_Comprehension\\_-\\_An\\_Experiment](https://www.researchgate.net/publication/294119566_The_Role_of_Method_Chains_and_Comments_in_Software_Readability_and_Comprehension_-_An_Experiment)>. Acesso em: 2 nov. 2017. Citado uma vez na página 57.

**Resumo:** Software readability and comprehension are important factors in software maintenance. There is a large body of research on software measurement, but the actual factors that make software easier to read or easier to comprehend are not well understood. In the present study, we investigate the role of method chains and code comments in software readability and comprehension. Our analysis comprises data from 104 students with varying programming experience.

Readability and comprehension were measured by perceived readability, reading time and performance on a simple cloze test. Regarding perceived readability, our results show statistically significant differences between comment variants, but not between method chain variants. Regarding comprehension, there are no significant differences between method chain or comment variants. Student groups with low and high experience, respectively, show significant differences in perceived readability and performance on the cloze tests. Our results do not show any significant relationships between perceived readability and the other measures taken in the present study. Perceived readability might therefore be insufficient as the sole measure of software readability or comprehension. We also did not find any statistically significant relationships between size and perceived readability, reading time and comprehension.

BRAND, M.G.J. Van den; KOOIKER, A.T.; VINJU, J.J. A Language Independent Framework for Context-sensitive Formatting. **CSMR '06 Proceedings of the Conference on Software Maintenance and Reengineering**, IEEE Computer Society, Bari, Italy, v. 26, p. 103–112, mar. 2006. DOI: [10.1109/CSMR.2006.4](https://doi.org/10.1109/CSMR.2006.4). Disponível em: <[https://www.researchgate.net/publication/4226896\\_A\\_language\\_independent\\_framework\\_for\\_context-sensitive\\_formatting](https://www.researchgate.net/publication/4226896_A_language_independent_framework_for_context-sensitive_formatting)>. Acesso em: 7 set. 2017. Citado uma vez na página 56.

**Resumo:** Automated formatting is an important technique for the software maintainer. It is either applied separately to improve the readability of source code, or as part of a source code transformation tool chain. In this paper we report on the application of generic tools for constructing formatters. In an industrial setting automated formatters need to be tailored to the requirements of the customer. The (legacy) programming language or dialect and the corporate formatting conventions are specific and non-negotiable. Can generic formatting tools deal with such unexpected requirements? Driven by an industrial case of nearly 80 thousand lines of Cobol code, several limitations in existing formatting technology have been addressed. We improved its flexibility by replacing a generative phase by a generic tool, and we added a little expressiveness to the formatting back end. Most importantly, we employed a multi-stage formatting framework that can cope with any kind of formatting convention using more computational power.

BRAND, M.G.J. van den; KOOIKER, A.T.; VEERMAN, N.P. *et al.* An industrial application of context-sensitive formatting. Online, 2005. Disponível em: <[https://www.researchgate.net/publication/228540036\\_An\\_industrial\\_application\\_of\\_context-sensitive\\_formatting](https://www.researchgate.net/publication/228540036_An_industrial_application_of_context-sensitive_formatting)>. Acesso em: 7 set. 2017. Citado uma vez na página 56.

**Resumo:** Automated formatting is an important technique for the software maintainer. It is either applied separately to improve the readability of source code,

or as part of a source code transformation tool chain. In this paper we report on the application of generic tools for constructing formatters. In an industrial setting automated formatters need to be tailored to the requirements of the customer. The (legacy) programming language or dialect and the corporate formatting conventions are specific and non-negotiable. Can generic formatting tools deal with such unexpected requirements? Driven by an industrial case of 78 thousand lines of Cobol code, several limitations in existing formatting technology have been addressed. We improved its flexibility by replacing a generative phase by a generic tool, and we added a little expressiveness to the formatting backend. Most importantly, we employed a multi-stage formatting architecture that can cope with any kind of formatting convention using more computational power.

BRAND, M.G.J. van den; KOOIKER, A.T.; VINJU, J.J.; VEERMAN, N.P. An architecture for context-sensitive formatting. **21st IEEE International Conference on Software Maintenance (ICSM'05)**, IEEE Computer Society, Budapest, Hungary, Hungary, set. 2005. DOI: [10.1109/ICSM.2005.17](https://doi.org/10.1109/ICSM.2005.17). Disponível em: <[https://www.researchgate.net/publication/4175894\\_An\\_architecture\\_for\\_context-sensitive\\_formatting](https://www.researchgate.net/publication/4175894_An_architecture_for_context-sensitive_formatting)>. Acesso em: 7 set. 2017. Citado uma vez na página 56.

**Resumo:** We developed an architecture for context-sensitive formatting of source code. The architecture was implemented and applied in an industrial formatting case.

BRENT, Richard P. Analysis of the Binary Euclidean Algorithm. **SIGSAM Bull.**, ACM, New York, NY, USA, v. 10, n. 2, p. 6–7, maio 1976. ISSN 0163-5824. DOI: [10.1145/1093397.1093399](https://doi.org/10.1145/1093397.1093399). Disponível em: <[http://doi.acm.org/10.1145/1093397.1093399](https://doi.acm.org/10.1145/1093397.1093399)>. Citado 3 vezes nas páginas 9, 64.

**Resumo:** The binary Euclidean algorithm finds the GCD of two integers  $u$  and  $v$  using subtraction, shifting and parity testing. Unlike the classical Euclidean algorithm, no divisions are required. We analyse a continuous model of the binary algorithm, and find the expected number of iterations.

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](https://doi.org/10.1145/3330431.3330441). Disponível em: <[http://doi.acm.org/10.1145/3330431.3330441](https://doi.acm.org/10.1145/3330431.3330441)>. Citado uma vez na página 86.

**Resumo:** Within this article creates a cross-platform compilation of the Golang programming language for raspberry pi. Golang, or Go supports type safety, the ability to dynamically enter data, and also contains a rich standard library of functions and built-in data types like arrays with dynamic size and associative arrays. With the help of multi-threading mechanisms, Go simplifies the distribution

of computations and network interactions, while modern data types open up to the programmer a world of flexible and modular code. The program quickly compiles, while there is a trash collector and reflection is maintained. Golang is a fast, statically typed, compiled language. Dealing with it you have the impression of using dynamically typed and interpreted language. A cross-platform compilation of the Golang programming language was created for raspberry pi - a single-board computer.

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 uma vez nas páginas [32, 33](#).

**Resumo:** People often say that LR( $k$ ) parsers are more powerful than LL( $k$ ) parsers. These statements are vague most of the time; in particular, should we compare the classes for a fixed  $k$  or the union over all  $k$ ? So how is the situation really? In particular, I am interested in how LL(\*) fits in.

BRUKNER, Časlav *et al.* Probabilistic instantaneous quantum computation. **Phys. Rev. A**, American Physical Society, v. 67, p. 034304, mar. 2003. DOI: [10.1103/PhysRevA.67.034304](https://doi.org/10.1103/PhysRevA.67.034304). Disponível em: <<https://link.aps.org/doi/10.1103/PhysRevA.67.034304>>. Citado uma vez na página [40](#).

**Resumo:** The principle of teleportation can be used to perform a quantum computation even before its quantum input is defined. The basic idea is to perform the quantum computation at some earlier time with qubits that are part of an entangled state. At a later time a generalized Bell-state measurement is performed jointly on the then defined actual input qubits and the rest of the entangled state. This projects the output state onto the correct one with a certain exponentially small probability. The sufficient conditions are found under which the scheme is of benefit.

BUNTRÖCK, Gerhard; LORYŚ, Krzysztof. On growing context-sensitive languages. *In: KUICH, W. (Ed.). Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992. p. 77–88. Citado uma vez na página [42](#).

**Resumo:** Growing context-sensitive grammars (GCSG) are investigated. The variable membership problem for GCSG is shown to be NP-complete. This solves a problem posed in [DW86]. It is also shown that the languages generated by GCSG form an abstract family of languages and several implications are presented.

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](https://doi.org/10.1109/52.10004). Citado uma vez na página [49](#).

**Resumo:** The author has distilled the basic operations of the pretty printer into

an abstract pretty printer that uses procedural parameters to perform low-level printing actions. By encapsulating the algorithm in one place, all the pretty-printing utilities will use the same algorithm, and the algorithm itself can be changed easily. The author describes how the abstract pretty printer can be used for basic design, printing to files and screens, setting the cursor, identifying a node, formatting text, and lexical changes.<>

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](https://doi.acm.org/10.1145/1456362.1456371). Disponível em: <[http://doi.acm.org/10.1145/1456362.1456371](https://doi.acm.org/10.1145/1456362.1456371)>. Citado uma vez na página 50.

**Resumo:** While many obfuscation schemes proposed, none of them satisfy any strong definition of obfuscation. Furthermore secure generalpurpose obfuscation algorithms have been proven to be impossible. Nevertheless, obfuscation schemes which in practice slow down malicious reverse-engineering by obstructing code comprehension for even short periods of time are considered a useful protection against malicious reverse engineering. In previous works, the difficulty of reverse engineering has been mainly estimated by means of code metrics, by the computational complexity of static analysis or by comparing the output of de-obfuscating tools. In this paper we take a different approach and assess the difficulty attackers have in understanding and modifying obfuscated code through controlled experiments involving human subjects.

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](https://doi.acm.org/10.1145/3217804.3217895). Disponível em: <[http://doi.acm.org/10.1145/3217804.3217895](https://doi.acm.org/10.1145/3217804.3217895)>. Citado uma vez na página 60.

**Resumo:** In this article we investigate developers involved in the creation of Open Source software to identify which characteristics favor innovation in the Open Source community. The results of the analysis show that higher reputation in the community improves the probability of gaining collaborators to a certain degree, but developers are also driven by reciprocity. This is consistent with the concept of gift economy. A significant network effect exists and emerges from standardization, showing that developers using the most popular programming languages in the service are likely to have more collaborators. Providing additional information (valid URL to developer's homepage) improves the chances of finding coworkers. The results can be generalized for the population of mature users of GitHub.

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://dl.acm.org/citation.cfm?id=563801.563804>>. Citado uma vez na página 21.

**Resumo:** When a compiler encounters a syntax error, it usually attempts to restart parsing to check the remainder of the input for any further errors. One common method of recovering from syntax errors is to repair the incorrect input string, allowing parsing to continue. This research presents a language independent method for repairing the input string to an LALR(1) parser. The method results in much faster repairs in general than an existing method, enabling some errors to be repaired that were previously too costly. Results are based on repairing syntax errors in Java programs from first year computer science students.

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 2 vezes nas páginas 27, 28.

**Resumo:** We investigate several conceptions of linguistic structure to determine whether or not they can provide simple and "revealing" grammars that generate all of the sentences of English and only these. We find that no finite-state Markov process that produces symbols with transition from state to state can serve as an English grammar. Furthermore, the particular subclass of such processes that produce n-order statistical approximations to English do not come closer, with increasing n, to matching the output of an English grammar. We formalize the notions of "phrase structure" and show that this gives us a method for describing language which is essentially more powerful, though still representable as a rather elementary type of finite-state process. Nevertheless, it is successful only when limited to a small subset of simple sentences. We study the formal properties of a set of grammatical transformations that carry sentences with phrase structure into new sentences with derived phrase structure, showing that transformational grammars are processes of the same elementary type as phrase-structure grammars; that the grammar of English is materially simplified if phrase structure description is limited to a kernel of simple sentences from which all other sentences are constructed by repeated transformations; and that this view of linguistic structure gives a certain insight into the use and understanding of language.

COAN, Evandro. **Alternate simplified logging support and general utilities functions**. 2016. Disponível em: <<https://github.com/evandrocoan/debugtools>>. Acesso em: 8 dez. 2019. Citado uma vez na página 156.

**Resumo:** Basic logger for Python logging module. If you are logging the debug output to a file and you would like to clean/erase all the log file contents, every time you re-create the logger, you need first to unlock the log file lock, otherwise,

the file contents are not going to be erased. To unlock your log file, you just need to call `log.setup("")`, with an empty string before creating a new logger.

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 uma vez nas páginas [68](#), [69](#), [71](#), [72](#).

**Resumo:** I see parsers like ANTLR offer some extension support for it: ANTLRErrorStrategy and Error Reporting and Recovery I looked into the `error_reporting_lalr.py`, and these examples, Lark just bail out parsing upon the first syntax error. It seem to correspond to the ANTLR BailErrorStrategy. I found this thesis from 2005, `lark.pdf`, there, they talk about error recovering. Is that thesis related with this Lark, or it is some other project with the same name?

COAN, Evandro. **Initial check list tasks to do.** 2016. Disponível em: <<https://github.com/evandrocoan/ObjectBeauty/issues/1>>. Acesso em: 1 dez. 2019. Citado uma vez na página [88](#).

**Resumo:** The languages still existing, however the formatting will not be based on languages but in the formatting based in visual formatting concepts. There will not be a setting to add spaces in paren for java, c++, etc. There will be a setting to add spaces in paren for specific regex and/or inclusion and exclusion laws formed on that particular setting file, together within its Unit Tests. Basically a setting file it will be very big and complex and only will treat a specific kind of visual formatting concept. Therefore to simply allow how to choose and setup a visual style for your own taste or needs, not all settings will be loaded to be processed and applied to parse the source code. Moreover not loading any settings file will imply on a zero-configuration, i.e., nothing will be changed on the source code after the parsing.

COAN, Evandro. **Options to control whitespace stripping.** 2016. Disponível em: <<https://github.com/uncrustify/uncrustify/issues/132#issuecomment-249205619>>. Acesso em: 21 jul. 2019. Citado uma vez na página [88](#).

**Resumo:** I think the best way to fix this is create a new Beautifier from 0. Why? Because I have seen the way Uncrustify is coded, and it is basically a structured programming for most things, and not fully documented. Therefore, it has a very high learning curve for new programmer get engaged/involved efficiently. Also is hard for new user to create a unified configuration file for all languages they use. For example, I to program on several languages, I need to create a big configuration file for each of them. In addition, when there is a change to my style, I need to go across several configuration files changing the same thing, if applicable on that target language. Then while building this new Beautifier take in account everybody in the world. Create a serious Object Oriented Program, fully

documented and aiming it be to easily extended program and configurable by the users.

COCKE, John. **Programming Languages and Their Compilers: Preliminary Notes**. New York, NY, USA: New York University, 1969. ISBN B0007F4UOA. Citado uma vez na página 29.

**Resumo:** Our aim in the present volume is to describe the inner working of a variety of programming languages, especially from the point of view of the compilers which translate these languages from their original "source" form into executable machine code. While this aim will of course make it necessary for us to describe in some detail the external form of each of the languages which we shall study, no more detail will be given than is strictly necessary in order to make it possible for the reader to gain a clear view of the machine code forms into which the language will be translated and of the problems that a compiler for the language must handle. However, internal description of the languages studied will be carried rather far. Thus the attentive reader of the present work should gain a rather good idea of the methods which can be employed to write a compiler for a given language. On the other hand, he cannot expect to find in this book the detailed account of the source conventions for any language which he would need to use the language.

COETZEE, Derrick. **Abstract syntax tree for Euclidean algorithm**. 2011. Disponível em: <[https://en.wikipedia.org/wiki/File:Abstract\\_syntax\\_tree\\_for\\_Euclidean\\_algorithm.svg](https://en.wikipedia.org/wiki/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg)>. Citado uma vez na página 65.

**Resumo:** The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches. This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees. Parse trees are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis.

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 uma vez na página 50.

**Resumo:** In software development, obfuscation is the deliberate act of creating source or machine code that is difficult for humans to understand. Like obfuscation in natural language, it may use needlessly roundabout expressions to compose

statements. Programmers may deliberately obfuscate code to conceal its purpose (security through obscurity) or its logic or implicit values embedded in it, primarily, in order to prevent tampering, deter reverse engineering, or even as a puzzle or recreational challenge for someone reading the source code. This can be done manually or by using an automated tool, the latter being the preferred technique in industry.

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](https://doi.acm.org/10.1145/1640089.1640133). Disponível em: <[http://doi.acm.org/10.1145/1640089.1640133](https://doi.acm.org/10.1145/1640089.1640133)>. Citado uma vez na página 171.

**Resumo:** In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming, which continues to this day. Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, abstract data types and objects. This essay attempts to explain the differences and also why the differences matter.

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](https://doi.acm.org/10.1145/586088.586092). Disponível em: <[https://www.researchgate.net/publication/220404285\\_Repairing\\_syntax\\_errors\\_in\\_LR\\_parsers](https://www.researchgate.net/publication/220404285_Repairing_syntax_errors_in_LR_parsers)>. Citado uma vez na página 21.

**Resumo:** This article reports on an error-repair algorithm for LR parsers. It locally inserts, deletes or shifts symbols at the positions where errors are detected, thus modifying the right context in order to resume parsing on a valid piece of input. This method improves on others in that it does not require the user to provide additional information about the repair process, it does not require precalculation of auxiliary tables, and it can be easily integrated into existing LR parser generators. A Yacc-based implementation is presented along with some experimental results and comparisons with other well-known methods.

CORMEN, Thomas H. *et al.* **Introduction to Algorithms, Third Edition**. 3rd. [S.I.]: The MIT Press, 2009. ISBN 0262033844, 9780262033848. Citado 4 vezes nas páginas 34, 39, 43, 44.

**Resumo:** If you had to buy just one text on algorithms, Introduction to Algorithms is a magnificent choice. The book begins by considering the mathematical foundations of the analysis of algorithms and maintains this mathematical rigor throughout the work. The tools developed in these opening sections are then applied to sorting, data structures, graphs, and a variety of selected algorithms

including computational geometry, string algorithms, parallel models of computation, fast Fourier transforms (FFTs), and more. This book's strength lies in its encyclopedic range, clear exposition, and powerful analysis. Pseudo-code explanation of the algorithms coupled with proof of their accuracy makes this book a great resource on the basic tools used to analyze the performance of algorithms.

CUTTING, David. **Enhancing legacy software system analysis by combining behavioural and semantic information sources**. Nov. 2016. Queen's University Belfast, Belfast, Northern Ireland. DOI: [10.13140/RG.2.2.21231.64160](https://doi.org/10.13140/RG.2.2.21231.64160). Disponível em: <[https://www.researchgate.net/publication/311289366\\_Enhancing\\_legacy\\_software\\_system\\_analysis\\_by\\_combining\\_behavioural\\_and\\_semantic\\_information\\_sources](https://www.researchgate.net/publication/311289366_Enhancing_legacy_software_system_analysis_by_combining_behavioural_and_semantic_information_sources)>. Acesso em: 2 nov. 2017. Citado uma vez na página 57.

**Resumo:** Computer software is, by its very nature highly complex and invisible yet subject to a near-continual pressure to change. Over time the development process has become more mature and less risky. This is in large part due to the concept of software traceability; the ability to relate software components back to their initial requirements and between each other. Such traceability aids tasks such as maintenance by facilitating the prediction of “ripple effects” that may result, and aiding comprehension of software structures in general. Many organisations, however, have large amounts of software for which little or no documentation exists; the original developers are no longer available and yet this software still underpins critical functions. Such “legacy software” can therefore represent a high risk when changes are required. Consequently, large amounts of effort go into attempting to comprehend and understand legacy software. The most common way to accomplish this, given that reading the code directly is hugely time consuming and near-impossible, is to reverse engineer the code, usually to a form of representative projection such as a UML class diagram. Although a wide number of tools and approaches exist, there is no empirical way to compare them or validate new developments. Consequently there was an identified need to define and create the Reverse Engineering to Design Benchmark (RED-BM). This was then applied to a number of industrial tools. The measured performance of these tools varies from 8.8% to 100%, demonstrating both the effectiveness of the benchmark and the questionable performance of several tools. In addition to the structural relationships detectable through static reverse engineering, other sources of information are available with the potential to reveal other types of relationships such as semantic links. One such source is the mining of source code repositories which can be analysed to find components within a software system that have, historically, commonly been changed together during the evolution of the system and from the strength of that infer a semantic link. An approach was implemented to mine such semantic relationships from repositories and relationships were found

beyond those expressed by static reverse engineering. These included groups of relationships potentially suitable for clustering. To allow for the general use of multiple information sources to build traceability links between software components a uniform approach was defined and illustrated. This includes rules and formulas to allow combination of sources. The uniform approach was implemented in the field of predictive change impact analysis using reverse engineering and repository mining as information sources. This implementation, the Java Code Relationship Analysis (jcRA) package, was then evaluated against an industry standard tool, JRipples. Depending on the target, the combined approach is able to outperform JRipples in detecting potential impacts with the risk of over-matching (a high number of false-positives and overall class coverage on some targets).

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 uma vez na página 21.

**Resumo:** The aim to improve error recovery in parsers generated by the LALR parser-generator Yacc. We describe an error recovery scheme which a new version of Yacc automatically builds into its parsers. The scheme uses state information to attempt to repair input which is syntactically incorrect. Repair by alteration of a single token is attempted first, followed by replacement of a phrase of the input. A parser for the C language is generated from existing specifications and tested on a collection of student programs. The quality of error recovery and diagnostic messages is found to be higher than that of the existing portable C compiler. The new version of Yacc may be used by any current user Yacc, with minor modifications to their existing specifications, to produce systems with enhanced syntax error recovery.

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 uma vez na página 30.

**Resumo:** Computability, Complexity, and Languages is an introductory text that covers the key areas of computer science, including recursive function theory, formal languages, and automata. It assumes a minimal background in formal mathematics. The book is divided into five parts: Computability, Grammars and Automata, Logic, Complexity, and Unsolvability. Computability theory is introduced in a manner that makes maximum use of previous programming experience, including a "universal" program that takes up less than a page. The number of exercises included has more than tripled. Automata theory, computational logic, and complexity theory are presented in a flexible manner, and can be covered in a variety of different arrangements.

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](https://doi.org/10.1109/ICSM.2002.1167816). Citado uma vez na página 45.

**Resumo:** Automatic software reengineering changes or repairs existing software systems. They are usually tailor-made for a specific customer and language dependent. Maintaining similar reengineering for multiple customers and different language dialects may, therefore, soon become problematic unless advanced language technology is used. Generic pretty-printing is part of such technology and is the subject of this paper. We discuss specific pretty-print aspects of software reengineering such as fulfilling customer-specific format conventions, preserving existing layout, and producing multiple output formats. In addition, we describe pretty-print techniques that help to reduce maintenance effort of tailor-made reengineering supporting multiple language dialects. Applications such as COBOL reengineering and SDL documentation generation show that our techniques, implemented in the generic pretty-printer GPP, are feasible.

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](https://doi.acm.org/10.1145/69622.357187). Disponível em: <[http://doi.acm.org/10.1145/69622.357187](https://doi.acm.org/10.1145/69622.357187)>. Citado 2 vezes nas páginas 36, 60.

**Resumo:** Two relations that capture the essential structure of the problem of computing LALR(1) look-ahead sets are defined, and an efficient algorithm is presented to compute the sets in time linear in the size of the relations. In particular, for a PASCAL grammar, the algorithm performs fewer than 15 percent of the set unions performed by the popular compiler-compiler YACC. When a grammar is not LALR(1), the relations, represented explicitly, provide for printing user-oriented error messages that specifically indicate how the look-ahead problem arose. In addition, certain loops in the digraphs induced by these relations indicate that the grammar is not LR( $k$ ) for any  $k$ . Finally, an oft-discovered and used but incorrect look-ahead set algorithm is similarly based on two other relations defined for the fwst time here. The formal presentation of this algorithm should help prevent its rediscovery.

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 uma vez na página 51.

**Resumo:** A source code obfuscator accepts a program source file, and generates another functionally equivalent source file which is much harder to understand or reverse-engineer. This is useful for technical protection of intellectual property

when: source code must be delivered for public execution purposes (with interpretive languages like ECMAScript in web pages)

DEUTSCH, David; PENROSE, Roger. Quantum theory, the Church-Turing principle and the universal quantum computer. **Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences**, v. 400, n. 1818, p. 97–117, 1985. DOI:

10.1098/rspa.1985.0070. eprint:

<https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1985.0070>. Disponível em: <<https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1985.0070>>.

Citado uma vez na página 40.

**Resumo:** It is argued that underlying the Church–Turing hypothesis there is an implicit physical assertion. Here, this assertion is presented explicitly as a physical principle: ‘every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means’. Classical physics and the universal Turing machine, because the former is continuous and the latter discrete, do not obey the principle, at least in the strong form above. A class of model computing machines that is the quantum generalization of the class of Turing machines is described, and it is shown that quantum theory and the ‘universal quantum computer’ are compatible with the principle. Computing machines resembling the universal quantum computer could, in principle, be built and would have many remarkable properties not reproducible by any Turing machine. These do not include the computation of non-recursive functions, but they do include ‘quantum parallelism’, a method by which certain probabilistic tasks can be performed faster by a universal quantum computer than by any classical restriction of it. The intuitive explanation of these properties places an intolerable strain on all interpretations of quantum theory other than Everett’s. Some of the numerous connections between the quantum theory of computation and the rest of physics are explored. Quantum complexity theory allows a physically more reasonable definition of the ‘complexity’ or ‘knowledge’ in a physical system than does classical complexity theory.

DICKE, R.H.; WITTKE, J.P. **Introduction to Quantum Mechanics**. [S.l.]: Addison-Wesley, 1963. (Addison-Wesley world student series). Disponível em: <<https://books.google.com.br/books?id=wIhLtAEACAAJ>>. Citado 2 vezes na página 40.

**Resumo:** At present, quantum mechanics provides us with the best model we have of the physical world and, in particular, of the submicroscopic world of the atom. This book is an introduction to the physical concepts and mathematical formulations of nonrelativistic quantum mechanics. To get the most profit from this book, a familiarity with basic undergraduatelevel physics, including atomic physics, electromagnetism, and classical mechanics, is required. A knowledge of

differential and integral calculus, and some familiarity with differential equations, is also needed.

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 14 vezes nas páginas 52, 53, 72, 75, 78, 85, 89.

**Resumo:** Visual Studio Code version 1.9 includes a cool performance improvement that we've been working on and I wanted to tell its story. TL;DR TextMate themes will look more like their authors intended in VS Code 1.9, while being rendered faster and with less memory consumption. Syntax Highlighting usually consists of two phases. Tokens are assigned to source code, and then they are targeted by a theme, assigned colors, and voilà, your source code is rendered with colors. It is the one feature that turns a text editor into a code editor. Tokenization in VS Code (and in the Monaco Editor) runs line-by-line, from top to bottom, in a single pass. A tokenizer can store some state at the end of a tokenized line, which will be passed back when tokenizing the next line. This is a technique used by many tokenization engines, including TextMate grammars, that allows an editor to retokenize only a small subset of the lines when the user makes edits.

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](https://doi.org/10.1109/VL.1997.626589). Citado uma vez na página 49.

**Resumo:** When input sentences are processed in some manner, say evaluated with some set of rules, the resulting sentence must be pretty-printed in order to be presented to the user. We introduce a technique called "Share-Where maintenance" which is used to preserve layout information by annotating abstract representations of visual sentences. The annotations in the abstract representation point to the presentation where sub-terms originated which were created either by the user (initial term) or by the language specifier (equations, as in introduced terms). Both kinds are visual presentations which are used for presenting the new term.

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](https://doi.acm.org/10.1145/362007.362035). Disponível em: <[http://doi.acm.org/10.1145/362007.362035](https://doi.acm.org/10.1145/362007.362035)>. Citado 2 vezes nas páginas 39, 60.

**Resumo:** Parsing algorithm which seems to be the most efficient general context-free algorithm known is described. It is similar to both Knuth's LR(k) algorithm and the familiar top-down algorithm. It has a time bound proportional to  $n^3$  (where  $n$  is

the length of the string being parsed) in general; it has an  $n^2$  bound for unambiguous grammars; and it runs in linear time on a large class of grammars, which seems to include most practical context-free programming language grammars. In an empirical comparison it appears to be superior to the top-down and bottom-up algorithms studied by Griffiths and Petrick.

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](https://doi.acm.org/10.1145/1639622.1639632). Disponível em: <[http://doi.acm.org/10.1145/1639622.1639632](https://doi.acm.org/10.1145/1639622.1639632)>. Acesso em: 30 out. 2017. Citado 2 vezes nas páginas [55](#), [63](#), [156](#).

**Resumo:** This paper explores an approach to improving the practical usability of static verification tools for debugging synchronization idioms. Synchronization idioms such as mutual exclusion and readers/writer locks are widely-used to ensure atomicity of critical regions. We present an annotation assistant that automatically generates program annotations. These annotations express noninterference between program statements, ensured by the synchronization idioms, and are used to identify atomic code regions. This allows the programmer to debug the use of the idioms in the program. We start by formalizing several well-known idioms by providing an abstract semantics for each idiom. For programs that use these idioms, we require the programmer to provide a few predicates linking the idiom with its realization in terms of program variables. From these, we automatically generate a proof script that is mechanically checked. These scripts include steps such as automatically generating assertions and annotating program actions with them, introducing auxiliary variables and invariants. We have successfully shown the applicability of this approach to several concurrent programs from the literature.

FENNER, Curtis. **Are Finite Automata Turing Complete?** Disponível em: <<https://cs.stackexchange.com/q/110998>>. Acesso em: 22 jun. 2019. Citado 3 vezes na página [39](#).

**Resumo:** Something is Turing Complete if it can be used to simulate any Turing Machine. So, can a Finite Automaton simulate a Turing Machine? On the question Can regular languages be Turing complete? they say a Regular Language can be Turing Complete, but it does not make sense to me. I am not talking about the Language being parsed, but the Finite Automaton itself.

FOWKES, Jaroslav; CHANTHIRASEGARAN, Pankajan; RANCA, Razvan. Autofolding for Source Code Summarization. **IEEE Transactions on Software Engineering**, IEEE Computer Society, School of Informatics, University of Edinburgh, Edinburgh, UK, PP, fev. 2017. ISSN 0098-5589. DOI: [10.1109/TSE.2017.2664836](https://doi.org/10.1109/TSE.2017.2664836). Disponível em:

<[https://www.researchgate.net/publication/260911164\\_Autofolding\\_for\\_Source\\_Code\\_Summarization](https://www.researchgate.net/publication/260911164_Autofolding_for_Source_Code_Summarization)>. Acesso em: 31 out. 2017. Citado uma vez na página 56.

**Resumo:** Developers spend much of their time reading and browsing source code, raising new opportunities for summarization methods. Indeed, modern code editors provide code folding, which allows one to selectively hide blocks of code. However this is impractical to use as folding decisions must be made manually or based on simple rules. We introduce the autofolding problem, which is to automatically create a code summary by folding less informative code regions. We present a novel solution by formulating the problem as a sequence of AST folding decisions, leveraging a scoped topic model for code tokens. On an annotated set of popular open source projects, we show that our summarizer outperforms simpler baselines, yielding a 28% error reduction. Furthermore, we find through a case study that our summarizer is strongly preferred by experienced developers. More broadly, we hope this work will aid program comprehension by turning code folding into a usable and valuable tool.

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 uma vez na página 16.

**Resumo:** We are considering to impose a single standard code format in our project (auto format with save actions in Eclipse). The reason is that currently there is a big difference in the code formats used by several (>10) developers which makes it harder for one developer to work on the code of another developer. The same Java file sometimes uses 3 different formats.

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 uma vez na página 21.

**Resumo:** Lark is a parser generator for LALR(2) and LR(1) grammars. With its backtracking facility it is even able to generate parsers for non-LR(k) languages. It is compatible with its predecessor, the parser generator Lalr [GrV]. The parser generator Lark offers the following features: generates highly efficient parsers provides automatic error reporting, error recovery, and error repair

GUZZI, Anja. Documenting and Sharing Knowledge About Code. In: PROCEEDINGS of the 34th International Conference on Software Engineering. Zurich, Switzerland: IEEE Press, 2012. (ICSE '12), p. 1535–1538. Disponível em:

<<http://dl.acm.org/citation.cfm?id=2337223.2337476>>. Acesso em: 31 out. 2017. Citado uma vez na página 56.

**Resumo:** Software engineers spend a considerable amount of time on program comprehension. Current research has primarily focused on assisting the developer trying to build up his understanding of the code. This knowledge remains only in the mind of the developer and, as time elapses, often “disappears”. In this research, we shift the focus to the developer who is using her Integrated Development Environment (IDE) for writing, modifying, or reading the code, and who actually understands the code she is working with. The objective of this PhD research is to seek ways to support this developer to document and share her knowledge with the rest of the team. In particular, we investigate the full potential of micro-blogging integrated into the IDE for addressing the program comprehension problem.

HALLIDAY, D.; RESNICK, R.; WALKER, J. **Fundamentals of Physics Extended, 10th Edition.** [S.I.]: John Wiley & Sons, Incorporated, 2013. ISBN 9781118473818. Disponível em: <<https://books.google.com.br/books?id=DTccAAAAQBAJ>>. Citado uma vez na página 40.

**Resumo:** The 10th edition of Halliday's Fundamentals of Physics, Extended building upon previous issues by offering several new features and additions. The new edition offers most accurate, extensive and varied set of assessment questions of any course management program in addition to all questions including some form of question assistance including answer specific feedback to facilitate success. The text also offers multimedia presentations (videos and animations) of much of the material that provide an alternative pathway through the material for those who struggle with reading scientific exposition. Furthermore, the book includes math review content in both a self-study module for more in-depth review and also in just-in-time math videos for a quick refresher on a specific topic. The Halliday content is widely accepted as clear, correct, and complete. The end-of-chapters problems are without peer. The new design, which was introduced in 9e continues with 10e, making this new edition of Halliday the most accessible and reader-friendly book on the market.

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 uma vez na página 34.

**Resumo:** Several notions of deterministic linear languages are considered and compared with respect to their complexities and to the families of formal languages they generate. We exhibit close relationships between simple linear languages and the deterministic linear languages both according to Nasu and Honda and to Ibarra, Jiang, and Ravikumar. Deterministic linear languages turn out to be

special cases of languages generated by linear grammars restricted to LL(1) conditions, which have a membership problem solvable in NC1. In contrast to that, deterministic linear languages defined via automata models turn out to have a DSPACE( $\log n$ )-complete membership problem. Moreover, they coincide with languages generated by linear grammars subject to LR(1) conditions.

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](https://doi.acm.org/10.1145/568438.568455). Disponível em: <<http://doi.acm.org/10.1145/568438.568455>>. Citado 13 vezes nas páginas 22, 24, 28, 30, 31, 39, 41–43, 60.

**Resumo:** This classic book on formal languages, automata theory, and computational complexity has been updated to present theoretical concepts in a concise and straightforward manner with the increase of hands-on, practical applications. This new edition comes with Gradiance, an online assessment tool developed for computer science. Gradiance is the most advanced online assessment tool developed for the computer science discipline. With its innovative underlying technology, Gradiance turns basic homework assignments and programming labs into an interactive learning

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 4 vezes nas páginas 52, 72.

**Resumo:** Syntect is a syntax highlighting library for Rust that uses Sublime Text syntax definitions. It aims to be a good solution for any Rust project that needs syntax highlighting, including deep integration with text editors written in Rust. It's used in production by at least two companies, and by many open source projects. If you are writing a text editor (or something else needing highlighting) in Rust and this library doesn't fit your needs, I consider that a bug and you should file an issue or email me. I consider this project mostly complete, I still maintain it and review PRs, but it's not under heavy development.

HUNNER, Trey; XU, Hong. **Editor Config**. 2012. Disponível em: <<https://github.com/editorconfig/editorconfig>>. Acesso em: 23 jul. 2019. Citado uma vez na página 49.

**Resumo:** EditorConfig helps maintain consistent coding styles for multiple developers working on the same project across various editors and IDEs. The EditorConfig project consists of a file format for defining coding styles and a collection of text editor plugins that enable editors to read the file format and adhere to defined styles. EditorConfig files are easily readable and they work nicely with version control systems.

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](https://doi.acm.org/10.1145/361227.361232). Disponível em: <[http://doi.acm.org/10.1145/361227.361232](https://doi.acm.org/10.1145/361227.361232)>. Citado uma vez na página 36.

**Resumo:** The problem of determining whether an arbitrary context-free grammar is a member of some easily parsed subclass of grammars such as the LR(k) grammars is considered. The time complexity of this problem is analyzed both when k is considered to be a fixed integer and when k is considered to be a parameter of the test. In the first case, it is shown that for every k there exists an  $O(nk+2)$  algorithm for testing the LR(k) property, where n is the size of the grammar in question. On the other hand, if both k and the subject grammar are problem parameters, then the complexity of the problem depends very strongly on the representation chosen for k. More specifically, it is shown that this problem is NP-complete when k is expressed in unary. When k is expressed in binary the problem is complete for nondeterministic exponential time. These results carry over to many other parameterized classes of grammars, such as the LL(k), strong LL(k), SLR(k), LC(k), and strong LC(k) grammars.

JBARA, A.; FEITELSON, D. G. How Programmers Read Regular Code: A Controlled 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](https://doi.org/10.1109/ICPC.2015.35). Disponível em: <[https://www.researchgate.net/publication/281579264\\_How\\_Programmers\\_Read-Regular\\_Code\\_A\\_Controlled\\_Experiment\\_Using\\_Eye\\_Tracking](https://www.researchgate.net/publication/281579264_How_Programmers_Read-Regular_Code_A_Controlled_Experiment_Using_Eye_Tracking)>. Acesso em: 31 out. 2017. Citado uma vez na página 14.

**Resumo:** Regular code, which includes repetitions of the same basic pattern, has been shown to have an effect on code comprehension: a regular function can be just as easy to comprehend as an irregular one with the same functionality, despite being longer and including more control constructs. It has been speculated that this effect is due to leveraging the understanding of the first instances to ease the understanding of repeated instances of the pattern. To verify and quantify this effect, we use eye tracking to measure the time and effort spent reading and understanding regular code. The results are that time and effort invested in the initial code segments are indeed much larger than those spent on the later ones, and the decay in effort can be modeled by an exponential or cubic model. This shows that syntactic code complexity metrics (such as LOC and MCC) need to be made context-sensitive, e.g. By giving reduced weight to repeated segments according to their place in the sequence.

JOHNSON, Maggie; ZELENSKI, Julie. **What about theses grammars and the minimal parser to recognize it?** 2009. Disponível em:

<<https://stackoverflow.com/questions/6379937>>. Acesso em: 20 jul. 2019. Citado uma vez nas páginas 32, 33.

**Resumo:** I'm trying to learn how to make a compiler. In order to do so, I read a lot about context-free language. But there are some things I cannot get by myself yet. Since it's my first compiler there are some practices that I'm not aware of. My questions are asked with in mind to build a parser generator, not a compiler neither a lexer. Some questions may be obvious. Among my reads are : Bottom-Up Parsing, Top-Down Parsing, Formal Grammars. The picture shown comes from : Miscellaneous Parsing. All coming from the Stanford CS143 class.

JONGE, Merijn De. Pretty-printing for software reengineering. **International Conference on Software Maintenance, 2002. Proceedings**, IEEE Computer Society, Montreal, Quebec, Canada, out. 2002. DOI: [10.1109/ICSM.2002.1167816](https://doi.org/10.1109/ICSM.2002.1167816). Disponível em: <[https://www.researchgate.net/publication/3998748\\_Pretty-Printing\\_for\\_Software\\_Reengineering](https://www.researchgate.net/publication/3998748_Pretty-Printing_for_Software_Reengineering)>. Acesso em: 7 set. 2017. Citado uma vez na página 56.

**Resumo:** Automatic software reengineering change or repair existing software systems. They are usually tailor-made for a specific customer and language dependent. Maintaining similar reengineering for multiple customers and different language dialects might therefore soon become problematic unless advanced language technology is being used. Generic pretty-printing is part of such technology and is the subject of this paper. We discuss specific pretty-print aspects of software reengineering such as fulfilling customer-specific format conventions, preserving existing layout, and producing multiple output formats. In addition, we describe pretty-print techniques that help to reduce maintenance effort of tailor-made reengineering supporting multiple language dialects. Applications, such as COBOL reengineering and SDL documentation generation show that our techniques, implemented in the generic pretty-printer GPP, are feasible.

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](https://doi.org/10.1145/3064848). Disponível em: <<https://hal.archives-ouvertes.fr/hal-01633123>>. Citado uma vez na página 37.

**Resumo:** The syntax of the C programming language is described in the C11 standard by an ambiguous context-free grammar, accompanied with English prose that describes the concept of " scope " and indicates how certain ambiguous code fragments should be interpreted. Based on these elements, the problem of implementing a compliant C11 parser is not entirely trivial. We review the main sources of difficulty and describe a relatively simple solution to the problem. Our solution employs the well-known technique of combining an LALR(1) parser with a " lexical feedback " mechanism. It draws on folklore knowledge and adds

several original aspects , including: a twist on lexical feedback that allows a smooth interaction with lookahead; a simplified and powerful treatment of scopes; and a few amendments in the grammar. Although not formally verified, our parser avoids several pitfalls that other implementations have fallen prey to. We believe that its simplicity, its mostly-declarative nature, and its high similarity with the C11 grammar are strong informal arguments in favor of its correctness. Our parser is accompanied with a small suite of " tricky " C11 programs. We hope that it may serve as a reference or a starting point in the implementation of compilers and analysis tools.

KANAMORI, Yoshito *et al.* A short survey on quantum computers. **International Journal of Computers and Applications**, ACTA Press, 2451 Dieppe Ave SW, Calgary, AB, Canada, v. 28, 2006. ISSN 1925-7074. DOI: [10.2316/Journal.202.2006.3.202-1700](https://doi.org/10.2316/Journal.202.2006.3.202-1700). Disponível em: <[https://www.researchgate.net/publication/228613051\\_A\\_short\\_survey\\_on\\_quantum\\_computers](https://www.researchgate.net/publication/228613051_A_short_survey_on_quantum_computers)>. Citado uma vez na página 40.

**Resumo:** Quantum computing is an emerging technology. The clock frequency of current computer processor systems may reach about 40 GHz within the next 10 years. By then, one atom may represent one bit. Electrons under such conditions are no longer described by classical physics, and a new model of the computer may be necessary by that time. The quantum computer is one proposal that may have merit in dealing with the problems presented. Currently, there exist some algorithms utilizing the advantage of quantum computers. For example, Shor's algorithm performs factoring of a large integer in polynomial time, whereas classical factoring algorithms can do it in exponential time. In this paper we briefly survey the current status of quantum computers, quantum computer systems, and quantum simulators.

KATZGRABER, Helmut G. *et al.* Seeking Quantum Speedup Through Spin Glasses: The Good, the Bad, and the Ugly. **Phys. Rev. X**, American Physical Society, v. 5, p. 031026, set. 2015. DOI: [10.1103/PhysRevX.5.031026](https://doi.org/10.1103/PhysRevX.5.031026). Disponível em: <<https://link.aps.org/doi/10.1103/PhysRevX.5.031026>>. Citado uma vez na página 40.

**Resumo:** There has been considerable progress in the design and construction of quantum annealing devices. However, a conclusive detection of quantum speedup over traditional silicon-based machines remains elusive, despite multiple careful studies. In this work we outline strategies to design hard tunable benchmark instances based on insights from the study of spin glasses—the archetypal random benchmark problem for novel algorithms and optimization devices. We propose to complement head-to-head scaling studies that compare quantum annealing machines to state-of-the-art classical codes with an approach that compares the

performance of different algorithms and/or computing architectures on different classes of computationally hard tunable spin-glass instances. The advantage of such an approach lies in having to compare only the performance hit felt by a given algorithm and/or architecture when the instance complexity is increased. Furthermore, we propose a methodology that might not directly translate into the detection of quantum speedup but might elucidate whether quantum annealing has a “quantum advantage” over corresponding classical algorithms, such as simulated annealing. Our results on a 496-qubit D-Wave Two quantum annealing device are compared to recently used state-of-the-art thermal simulated annealing codes.

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](https://doi.org/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 2 vezes na página 32.

**Resumo:** Motivation: Microarray experiments generate vast amounts of data. The unknown or only partially known functional context of differentially expressed genes may be assessed by querying the Gene Ontology database via GOMiner. Resulting tree representations are difficult to interpret and are not suited for visualization of this type of data. Methods are needed to effectively visualize these complex set relationships. Results: We present a visualization approach for set relationships based on Venn diagrams. The proposed extension enhances the usual notion of Venn diagrams by incorporating set size information. The cardinality of the sets and intersection sets is represented by their corresponding circle (polygon) sizes. To avoid local minima, solutions to this problem are sought by evolutionary optimization. This generalized Venn diagram approach has been implemented as an interactive Java application (VennMaster) specifically designed for use with GOMiner in the context of the Gene Ontology database. Availability: VennMaster is platform-independent (Java 1.4.2) and has been tested on Windows (XP, 2000), Mac OS X, and Linux. Supplementary information and the software (free for non-commercial use) are available at <http://www.informatik.uni-ulm.de/ni/mitarbeiter/HKestler/vennm> together with a user documentation. Contact: [hans.kestler@medizin.uni-ulm.de](mailto:hans.kestler@medizin.uni-ulm.de)

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](https://doi.org/10.1145/2884781.2884802). Citado 2 vezes nas páginas 49, 63.

**Resumo:** Refactoring engines are standard tools in today’s Integrated Development Environments (IDEs). They allow programmers to perform one refactoring at

a time, but programmers need more. Most design patterns in the Gang-of-Four text can be written as a refactoring script - a programmatic sequence of refactorings. In this paper, we present R3, a new Java refactoring engine that supports refactoring scripts. It builds a main-memory, non-persistent database to encode Java entity declarations (e.g., packages, classes, methods), their containment relationships, and language features such as inheritance and modifiers. Unlike classical refactoring engines that modify Abstract Syntax Trees (ASTs), R3 refactorings modify only the database; refactored code is produced only when pretty-printing ASTs that reference database changes. R3 performs comparable precondition checks to those of the Eclipse Java Development Tools (JDT) but R3's codebase is about half the size of the JDT refactoring engine and runs an order of magnitude faster. Further, a user study shows that R3 improved the success rate of retrofitting design patterns by 25% up to 50%.

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 uma vez na página 53.

**Resumo:** PackageDev provides syntax highlighting and other helpful utility for Sublime Text resource files. Resource files are ways of configuring the Sublime Text text editor to various extends, including but not limited to: custom syntax definitions, context menus (and the main menu), and key bindings. Thus, this package is ideal for package developers, but even normal users of Sublime Text who want to configure it to their liking should find it very useful.

KIRPICOV, Eugene. **Can regular languages be Turing complete?** 2014. Disponível em: <<https://cs.stackexchange.com/questions/33666>>. Acesso em: 13 jun. 2019. Citado 2 vezes nas páginas 39, 70.

**Resumo:** While the set of legal programs in Jot is regular, Jot itself is Turing-complete. That means that each computable function can be expressed in Jot. We can even come up with a language in which all binary strings are legal, but the language itself is Turing complete (exercise). You're confusing syntax and semantics.

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 2 vezes nas páginas 34, 36.

**Resumo:** There has been much recent interest in languages whose grammar is sufficiently simple that an efficient left-to-right parsing algorithm can be mechani-

cally produced from the grammar. In this paper, we define LR(k) grammars, which are perhaps the most general ones of this type, and they provide the basis for understanding all of the special tricks which have been used in the construction of parsing algorithms for languages with simple structure, e.g. algebraic languages. We give algorithms for deciding if a given grammar satisfies the LR(k) condition, for given k, and also give methods for generating recognizes for LR(k) grammars. It is shown that the problem of whether or not a grammar is LR(k) for some k is undecidable, and the paper concludes by establishing various connections between LR(k) grammars and deterministic languages. In particular, the LR(k) condition is a natural analogue, for grammars, of the deterministic condition, for languages.

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](https://doi.org/10.1007/3-540-06841-4_65). Disponível em: <<http://dl.acm.org/citation.cfm?id=646230.681872>>. Citado 2 vezes nas páginas 29, 42, 43.

**Resumo:** A general study of parallel non-deterministic parsing and translation à la Earley is developed formally, based on non-deterministic pushdown acceptor-transducers. Several results (complexity and efficiency) are established, some new and other previously proved only in special cases. As an application, we show that for every family of deterministic context-free pushdown parsers (e.g. precedence, LR(k), LL(k), ...) there is a family of general context-free parallel parsers that have the same efficiency in most practical cases (e.g. analysis of programming languages).

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](https://doi.org/10.1002/9781119285021.ch5). Disponível em: <[https://www.researchgate.net/publication/311666005\\_Getting\\_Productive](https://www.researchgate.net/publication/311666005_Getting_Productive)>. Acesso em: 2 nov. 2017. Citado uma vez na página 16.

**Resumo:** This chapter explains how people can work with help in Git, understand the multiple repositories model, and stage files. Git includes two different forms of help: an abbreviated version and a full version. Partial and interactive staging, committing files into the local repository, and writing good commit messages are also discussed. The chapter also elucidates concepts such as SHA1, options for staging files, and forming good commit messages. Although working with multiple repositories at the same time is common in Git, it is a different way of working for most people. Git commit includes a –verbose option. The first time this option is used on the command line, it results in the diff output between the staging area and the local repository being included. The chapter also talks about ways to

amend commits and use some advanced techniques such as commit message template files to improve commit messages.

LASTER, Brent. Tracking Changes. In: PROFESSIONAL Git. Cary, North Carolina, USA: John Wiley & Sons, Inc., nov. 2016. p. 105–125. ISBN 9781119285021. DOI: 10.1002/9781119285021.ch6. Disponível em: <[https://www.researchgate.net/publication/311666333\\_Tracking\\_Changes](https://www.researchgate.net/publication/311666333_Tracking_Changes)>. Acesso em: 6 nov. 2017. Citado uma vez na página 55.

**Resumo:** This chapter considers different versions of files at the different levels in Git to keep track of where everything is and how the versions at the different levels may differ from each other. It explains ways to keep track of all of the work that's in progress. Git has two commands that can help with this: status and diff. Using these two commands users can quickly understand the state of their changes in the local environment and ensure that the correct changes are tracked and stored in Git. For files that are in the working directory or staging area, the status command answers three questions: whether or not a file is tracked, what is in the staging area, and whether or not a file is modified. Git can report the status of untracked files in a couple of different ways, depending on whether or not something is staged.

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 uma vez na página 34.

**Resumo:** A new general context-free parsing algorithm is presented which runs in linear time and space on every LR(k) grammar without using any lookahead and without making use of the LR property. Most of the existing implementations of tabular parsing algorithms, including those using lookahead, can easily be adapted to this new algorithm without a noteworthy loss of efficiency. For some natural right recursive grammars both the time and space complexity will be improved from O(n<sup>2</sup>) to O(n). This makes this algorithm not only of theoretical but probably of practical interest as well.

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 uma vez na página 53.

**Resumo:** The common abstraction of XML Schema by unranked regular tree languages is not entirely accurate. To shed some light on the actual expressive power

of XML Schema, intuitive semantical characterizations of the Element Declarations Consistent (EDC) rule are provided. In particular, it is obtained that schemas satisfying EDC can only reason about regular properties of ancestors of nodes. Hence, with respect to expressive power, XML Schema is closer to DTDs than to tree automata. These theoretical results are complemented with an investigation of the XML Schema Definitions (XSDs) occurring in practice, revealing that the extra expressiveness of XSDs over DTDs is only used to a very limited extent. As this might be due to the complexity of the XML Schema specification and the difficulty of understanding the effect of constraints on typing and validation of schemas, a simpler formalism equivalent to XSDs is proposed. It is based on contextual patterns rather than on recursive types and it might serve as a light-weight front end for XML Schema. Next, the effect of EDC on the way XML documents can be typed is discussed. It is argued that a cleaner, more robust, larger but equally feasible class is obtained by replacing EDC with the notion of 1-pass preorder typing (1PPT): schemas that allow one to determine the type of an element of a streaming document when its opening tag is met. This notion can be defined in terms of grammars with restrained competition regular expressions and there is again an equivalent syntactical formalism based on contextual patterns. Finally, algorithms for recognition, simplification, and inclusion of schemas for the various classes are given.

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 uma vez na página 47.

**Resumo:** We are currently desperate, because we are not able to meet our code style by our current uncrustify configuration. We already spent plenty hours setup our configuration correctly, but there are still issues we are not able to fix. To show our issues I prepared a code example that shows our wanted code style where I marked the issues by a comment.

MEHMOOD, Abid; JAWAWI, Dayang Norhayati Abang. Aspect-oriented model-driven code generation: A systematic mapping study. **Information and Software Technology**, Elsevier North-Holland, Inc., Department of Software Engineering, Faculty of Computer Science e Information Systems, Universiti Teknologi Malaysia, 81310 Skudai, Johor, Malaysia, v. 55, p. 395–411, set. 2012. DOI: 10.1016/j.infsof.2012.09.003. Disponível em: <[https://www.researchgate.net/publication/257391227\\_Aspect-oriented\\_model-driven\\_code\\_generation\\_A\\_systematic\\_mapping\\_study](https://www.researchgate.net/publication/257391227_Aspect-oriented_model-driven_code_generation_A_systematic_mapping_study)>. Acesso em: 12 set. 2017. Citado uma vez na página 55.

**Resumo:** Context: Model-driven code generation is being increasingly applied to enhance software development from perspectives of maintainability, extensibility

and reusability. However, aspect-oriented code generation from models is an area that is currently underdeveloped. Objective: In this study we provide a survey of existing research on aspect-oriented modeling and code generation to discover current work and identify needs for future research. Method: A systematic mapping study was performed to find relevant studies. Classification schemes have been defined and the 65 selected primary studies have been classified on the basis of research focus, contribution type and research type. Results: The papers of solution proposal research type are in a majority. All together aspect-oriented modeling appears being the most focused area divided into modeling notations and process (36%) and model composition and interaction management (26%). The majority of contributions are methods. Conclusion: Aspect-oriented modeling and composition mechanisms have been significantly discussed in existing literature while more research is needed in the area of model-driven code generation. Furthermore, we have observed that previous research has frequently focused on proposing solutions and thus there is need for research that validates and evaluates the existing proposals in order to provide firm foundations for aspect-oriented model-driven code generation.

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](https://doi.org/10.1145/182.358437). Disponível em: <[https://www.researchgate.net/publication/234809222\\_Program\\_indentation\\_and\\_comprehensibility](https://www.researchgate.net/publication/234809222_Program_indentation_and_comprehensibility)>. Acesso em: 7 set. 2017.

Citado uma vez na página 48.

**Resumo:** The consensus in the programming community is that indentation aids program comprehension, although many studies do not back this up. We tested program comprehension on a Pascal program. Two styles of indentation were used – blocked and non-blocked – in addition to four possible levels of indentation (0, 2, 4, 6 spaces). Both experienced and novice subjects were used. Although the blocking style made no difference, the level of indentation had a significant effect on program comprehension. (2–4 spaces had the highest mean score for program comprehension.) We recommend that a moderate level of indentation be used to increase program comprehension and user satisfaction.

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](https://doi.org/10.1145/2889420.2892271). Disponível em: <[http://doi.acm.org/10.1145/2889420.2892271](https://doi.acm.org/10.1145/2889420.2892271)>. Citado 3 vezes nas páginas 39, 70.

**Resumo:** Turing complete languages can express unbounded computations over unbounded structures, either directly or by a suitable encoding. In contrast, Domain Specific Languages (DSLs) are intended to simplify the expression of

computations over structures in restricted contexts. However, such simplification often proves irksome, especially for constructing more elaborate programs where the domain, though central, is one of many considerations. Thus, it is often tempting to extend a DSL with more general abstractions, typically to encompass common programming tropes, typically from favourite languages. The question then arises: once a DSL becomes Turing complete, then in what sense is it still domain specific?

MINELLI, Roberto; MOCCI, Andrea; LANZA, Michele. Quantifying Program Comprehension with Interaction Data. **14th International Conference on Quality Software**, IEEE Computer Society, Dallas, TX, USA, out. 2014. DOI: [10.1109/QSIC.2014.11](https://doi.org/10.1109/QSIC.2014.11). Disponível em: <[https://www.researchgate.net/publication/286487172\\_Quantifying\\_Program\\_Comprehension\\_with\\_Interaction\\_Data](https://www.researchgate.net/publication/286487172_Quantifying_Program_Comprehension_with_Interaction_Data)>. Acesso em: 31 out. 2017. Citado uma vez na página 15.

**Resumo:** It is common knowledge that program comprehension takes up a substantial part of software development. This "urban legend" is based on work that dates back decades, which throws up the question whether the advances in software development tools, techniques, and methodologies that have emerged since then may invalidate or confirm the claim. We present an empirical investigation which goal is to confirm or reject the claim, based on interaction data which captures the user interface activities of developers. We use interaction data to empirically quantify the distribution of different developer activities during software development: In particular, we focus on estimating the role of program comprehension. In addition, we investigate if and how different developers and session types influence the duration of such activities. We analyze interaction data from two different contexts: One comes from the ECLIPSE IDE on Java source code development, while the other comes from the PHARO IDE on Smalltalk source code development. We found evidence that code navigation and editing occupies only a small fraction of the time of developers, while the vast majority of the time is spent on reading & understanding source code. In essence, the importance of program comprehension was significantly underestimated by previous research.

MIRZA, Olfat; JOY, Mike. Style Analysis for Source Code Plagiarism Detection. **International Conference Plagiarism across Europe and Beyond 2015**, Online, Brno, Czech Republic, p. 53–61, jun. 2015. Disponível em: <[https://www.researchgate.net/publication/303932091\\_Style\\_Analysis\\_for\\_Source\\_Code\\_Plagiarism\\_Detection](https://www.researchgate.net/publication/303932091_Style_Analysis_for_Source_Code_Plagiarism_Detection)>. Acesso em: 27 out. 2017. Citado uma vez na página 56.

**Resumo:** Plagiarism has become an increasing problem in higher education in recent years. A number of research papers have discussed the problem of

plagiarism in terms of text and source code and the techniques to detect it in various contexts. There is a variety of easy ways of copying others' work because the source code can be obtained from online source code banks and textbooks, which makes plagiarism easy for students. Source code plagiarism has a very specific definition, and Parker and Hamblen define plagiarism on software as "A program that has been produced from another program with a small number of routine transformations". The transformations can range from very simple changes to very difficult ones, which can be one of the six levels of program modifications that are given by Faidhi and Robinson. Coding style is a way to detect source code plagiarism because it relates to programmer personality without affecting the logic of a program, and can be used to differentiate between different code authors. This paper reviews a number of publications which report style comparison to detect source code plagiarism in order to determine research gaps and explore areas where this approach can be improved. A summary of the plagiarism techniques in which style analysis can help identify plagiarism is presented.

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](https://doi.org/10.1007/BF01934200). Disponível em: <<https://doi.org/10.1007/BF01934200>>. Citado uma vez na página 20.

**Resumo:** The paper presents a technique for generating translators for problem-oriented and other command- or data-oriented languages which can be interpretively executed. The system consists of: (a) a stored grammar or table, representing in a modified graph form the syntactically valid statements and the corresponding semantic actions; (b) a set of application procedures, written in a procedural language; (c) a universal translator, which performs reading, input conversion, matching input items against the grammar, and calling the procedures specified by the grammar. The generator consists of the translator and a specific grammar and set of procedures, which together convert the problem-oriented description of the source grammar into the table used by the translator for execution.

NIEHUES, Lucas Boppo. **Estudo e Criação de um Editor de Código Estruturado**. 2013. Graduation Thesis submitted to the Computer Science Department – Federal University of Santa Catarina, Florianópolis, Santa Catarina, Brazil. [Departamento de Informática e Estatística]. Disponível em: <<https://tcc.inf.ufsc.br>>. Acesso em: 1 mar. 2017. Citado uma vez na página 31.

**Resumo:** Virtually all of the code editors for business use in general-purpose languages are oriented to lines and characters. The alternative, editors working directly with the syntactic tree, has received most attention in this area, aiming to have important advantages, such as automatic control of the structure and greater efficiency in the script. The purpose of this paper is to study these alternating

active editors, called structured editors or syntax-driven editors, comparing them with traditional editors. Once the advantages and disadvantages have been raised, a structured editor will be developed that best fits in with this medium. This work first provides for an in-depth study of these structured editors, in the context of programming in general or non-general purpose languages. Previous efforts in this area will be sought, assessing their impact, lessons learned and limitations encountered. In a second moment these lessons will be used for the development of a proprietary, fully structured or hybrid editor, in order to increase the efficiency of the programmer. Preferably this editor will be prepared for a standard general purpose language and will be able to edit actual programs. This work aimed to better understand the principles behind structured editors, seeking information on their effectiveness, advantages, disadvantages and factors involved in their lack of adoption. It is also envisaged the construction of a functional structured editor, capable of being used on a daily basis, that makes better use of these advantages.

NUNES, Ghabriel Calsa. **Simulador de Automatos e Maquinas de Turing**. 2017. Graduation Thesis submitted to the Computer Science Department – Federal University of Santa Catarina, Florianópolis, Santa Catarina, Brazil. [Departamento de Informática e Estatística]. Disponível em: <<https://tcc.inf.ufsc.br>>. Acesso em: 7 dez. 2019. Citado uma vez na página 31.

**Resumo:** The recognition of sentences by recognizing mechanisms such as finite automata, pushdown automata and Turing machines is one of the most important subjects in disciplines such as Theory of Computation and Formal Languages. However, there's a lack of high quality systems to simulate such recognition, which frequently makes students have questions about these subjects, resulting in a more difficult learning process. We propose a new web application capable of fulfilling these needs, aiming to improve the student comprehension about these subjects. It will be possible to download the new tool to run it without needing an Internet connection, and the interface will be easy to use and compatible with mobile devices. The system will then be compared with other existing solutions using criteria and characteristics such as ease of use, completeness, correctness and number of provided functionalities.

ODGAARD, Allan. **TextMate - A graphical text editor for macOS**. 2004. Disponível em: <<https://github.com/textmate/textmate>>. Acesso em: 30 nov. 2019. Citado 7 vezes nas páginas 52, 53.

**Resumo:** TextMate 1.0 was released on 5 October 2004, after 5 months of development, followed by version 1.0.1 on 21 October 2004. The release focused on implementing a small feature set well, and did not have a preference window or a toolbar, didn't integrate FTP, and had no options for printing. At first only a

small number of programming languages were supported, as only a few “language bundles” had been created. Even so, some developers found this early and incomplete version of TextMate a welcome change to a market that was considered stagnated by the decade-long dominance of BBEdit.

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 uma vez na página 16.

**Resumo:** They are known by many names: verison control, source control, revision control. Regardless of how you call it, a version control tool is critical to the success of your software project. In its most basic form, a version control system is a tool that allows you to store a modification history of all your code. This concept is used in many other systems including Wikis and incremental backup solutions. Version control tools for software go far beyond a simple history of code modifications, however. They increase productivity, allow for concurrent development of large projects, maintain multiple versions (branches) of the same project, and much more.

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 2 vezes nas páginas 60, 156.

**Resumo:** Many instructors have already discovered the joy of teaching programming using the Python programming language. Now it's time to take Python to the next level. This workshop will introduce Django, an open source Python web framework that saves you time and makes web development fun. It's aimed at Computer Science instructors who want to teach how to build elegant web applications with minimal fuss. Django follows the Model-View-Controller (MVC) architectural pattern. Its goal is to ease the creation of complex, database-driven websites. Django emphasizes reusability and "pluggability" of components, rapid development, and the principle of DRY (Don't Repeat Yourself). Python is used throughout, even for settings, files, and data models. Topics that will be covered during the workshop include: setup and configuration, template language, and database integration through object-relational mapping. Participants should have some familiarity with Python, HTML and SQL. Laptop Required.

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 uma vez na página 31.

**Resumo:** In this report, certain properties of context-free (CF or type 2) grammars are investigated, like that of Chomsky. In particular, questions regarding structure, possible ambiguity and relationship to finite automata are considered. The following results are presented: The language generated by a context-free grammar is linear in a sense that is defined precisely. The requirement of unambiguity—that every sentence has a unique phrase structure—weakens the grammar in the sense that there exists a CF language that cannot be generated unambiguously by a CF grammar. The result that not every CF language is a finite automaton (FA) language is improved in the following way. There exists a CF language L such that for any L ⊂ L, if L is FA, an L ⊂ L can be found such that L is also FA, L ⊂ L and L contains infinitely many sentences not in L. A type of grammar is defined that is intermediate between type 1 and type 2 grammars. It is shown that this type of grammar is essentially stronger than type 2 grammars and has the advantage over type 1 grammars that the phrase structure of a grammatical sentence is unique, once the derivation is given.

PARR, Terence. **The Definitive ANTLR 4 Reference**. 2nd. [S.l.]: Pragmatic Bookshelf, 2013. ISBN 1934356999, 9781934356999. Citado 7 vezes nas páginas 23, 32, 41, 68, 73, 155.

**Resumo:** Programmers run into parsing problems all the time. Whether it's a data format like JSON, a network protocol like SMTP, a server configuration file for Apache, a PostScript/PDF file, or a simple spreadsheet macro language—ANTLR v4 and this book will demystify the process. ANTLR v4 has been rewritten from scratch to make it easier than ever to build parsers and the language applications built on top. This completely rewritten new edition of the bestselling Definitive ANTLR Reference shows you how to take advantage of these new features. Build your own languages with ANTLR v4, using ANTLR's new advanced parsing technology. In this book, you'll learn how ANTLR automatically builds a data structure representing the input (parse tree) and generates code that can walk the tree (visitor). You can use that combination to implement data readers, language interpreters, and translators. You'll start by learning how to identify grammar patterns in language reference manuals and then slowly start building increasingly complex grammars. Next, you'll build applications based upon those grammars by walking the automatically generated parse trees. Then you'll tackle some nasty language problems by parsing files containing more than one language (such as XML, Java, and Javadoc). You'll also see how to take absolute control over parsing by embedding Java actions into the grammar. You'll learn directly from well-known parsing expert Terence Parr, the ANTLR creator and project lead. You'll master ANTLR grammar construction and learn how to build language tools using the built-in

parse tree visitor mechanism. The book teaches using real-world examples and shows you how to use ANTLR to build such things as a data file reader, a JSON to XML translator, an R parser, and a Java class-interface extractor. This book is your ticket to becoming a parsing guru! What You Need: ANTLR 4.0 and above. Java development tools. Ant build system optional (needed for building ANTLR from source)

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](https://doi.acm.org/10.1145/1993498.1993548). Disponível em: <[http://doi.acm.org/10.1145/1993498.1993548](https://doi.acm.org/10.1145/1993498.1993548)>. Citado uma vez na página 32.

**Resumo:** Despite the power of Parser Expression Grammars (PEGs) and GLR, parsing is not a solved problem. Adding nondeterminism (parser speculation) to traditional LL and LR parsers can lead to unexpected parse-time behavior and introduces practical issues with error handling, single-step debugging, and side-effecting embedded grammar actions. This paper introduces the LL(\*) parsing strategy and an associated grammar analysis algorithm that constructs LL(\*) parsing decisions from ANTLR grammars. At parse-time, decisions gracefully throttle up from conventional fixed  $k \geq 1$  lookahead to arbitrary lookahead and, finally, fail over to backtracking depending on the complexity of the parsing decision and the input symbols. LL(\*) parsing strength reaches into the context-sensitive languages, in some cases beyond what GLR and PEGs can express. By statically removing as much speculation as possible, LL(\*) provides the expressivity of PEGs while retaining LL's good error handling and unrestricted grammar actions. Widespread use of ANTLR (over 70,000 downloads/year) shows that it is effective for a wide variety of applications.

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](https://doi.acm.org/10.1145/2660193.2660202). Disponível em: <[http://doi.acm.org/10.1145/2660193.2660202](https://doi.acm.org/10.1145/2660193.2660202)>. Citado uma vez na página 32.

**Resumo:** Despite the advances made by modern parsing strategies such as PEG, LL(\*), GLR, and GLL, parsing is not a solved problem. Existing approaches suffer from a number of weaknesses, including difficulties supporting side-effecting embedded actions, slow and/or unpredictable performance, and counter-intuitive matching strategies. This paper introduces the ALL(\*) parsing strategy that combines the simplicity, efficiency, and predictability of conventional top-down LL(k) parsers with the power of a GLR-like mechanism to make parsing decisions. The

critical innovation is to move grammar analysis to parse-time, which lets ALL(\*) handle any non-left-recursive context-free grammar. ALL(\*) is O(n<sup>4</sup>) in theory but consistently performs linearly on grammars used in practice, outperforming general strategies such as GLL and GLR by orders of magnitude. ANTLR 4 generates ALL(\*) parsers and supports direct left-recursion through grammar rewriting. Widespread ANTLR 4 use (5000 downloads/month in 2013) provides evidence that ALL(\*) is effective for a wide variety of applications.

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](https://doi.acm.org/10.1145/191033.191155). Disponível em: <[http://doi.acm.org/10.1145/191033.191155](https://doi.acm.org/10.1145/191033.191155)>. Citado 2 vezes nas páginas 23, 68.

**Resumo:** This paper is a guided tour through the first day of a CS 1 course. It discusses teaching Extended Backus-Naur Form (EBNF) as the first topic—not to facilitate presenting the syntax of a programming language, but because EBNF is a microcosm of programming. With no prerequisites, students are introduced to a variety of fundamental concepts in programming: formal systems, abstraction, control structures, equivalence of descriptions, the difference between syntax and semantics, and the relative power of recursion versus iteration. As a non-numeric formal system, EBNF provides a small but concrete context in which to study all these topics. EBNF descriptions include abstraction (named rules) and the four fundamental control structures (sequence, decision, repetition, and recursion). Because there are no data or parameters in EBNF, it is easy to sidestep tricky issues surrounding variables, scope, assignment statements, and parameter modes. Describing entities in EBNF is similar to describing computations in a programming language. Students learn to read a description and analyze whether it generates/matches candidate symbols; then they learn to synthesize descriptions from English specifications, augmented by legal and illegal exemplars of symbols. All these concepts can be covered in one lecture, establishing a high level of formality early in the course, while foreshadowing actual programming language features and techniques to be covered later. Of course, learning EBNF also facilitates presenting the syntax of a programming language concisely during the rest of the course.

PEREIRA, André Marques *et al.* Comparison of open source tools for project management. **International Journal of Software Engineering and Knowledge Engineering**, World Scientific Publishing, Federal University of Santa Catarina (UFSC), Florianópolis, Santa Catarina, Brazil, v. 23, p. 189–209, mar. 2013. DOI: [10.1142/S0218194013500046](https://doi.org/10.1142/S0218194013500046). Disponível em: <[https://www.researchgate.net/publication/273569026\\_Comparison\\_of\\_open\\_](https://www.researchgate.net/publication/273569026_Comparison_of_open_)>

[source\\_tools\\_for\\_project\\_management](#). Acesso em: 27 out. 2017. Citado uma vez na página 56.

**Resumo:** Software projects often fail, because they are not adequately managed. The establishment of effective and efficient project management practices still remains a key challenge to software organizations. Striving to address these needs, "best practice" models, such as, the Capability Maturity Model Integration (CMMI) or the Project Management Body of Knowledge (PMBOK), are being developed to assist organizations in improving project management. Although not required, software tools can help implement the project management process in practice. In order to provide comprehensive, low-cost tool support for project management, specifically, for small and medium enterprises (SMEs), in this paper we compare the most popular free/open-source web-based project management tools with respect to their compliance to PMBOK and CMMI for Development (CMMI-DEV). The results of this research can be used by organizations to make decisions on tool adoptions as well as a basis for evolving software tools in alignment with best practices models.

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](https://doi.org/10.1145/2872427.2883029). Disponível em: [<https://doi.org/10.1145/2872427.2883029>](https://doi.org/10.1145/2872427.2883029). Citado uma vez na página 72.

**Resumo:** JSON – the most popular data format for sending API requests and responses – is still lacking a standardized schema or meta-data definition that allows the developers to specify the structure of JSON documents. JSON Schema is an attempt to provide a general purpose schema language for JSON, but it is still work in progress, and the formal specification has not yet been agreed upon. Why this could be a problem becomes evident when examining the behaviour of numerous tools for validating JSON documents against this initial schema proposal: although they agree on most general cases, when presented with the greyer areas of the specification they tend to differ significantly. In this paper we provide the first formal definition of syntax and semantics for JSON Schema and use it to show that implementing this layer on top of JSON is feasible in practice. This is done both by analysing the theoretical aspects of the validation problem and by showing how to set up and validate a JSON Schema for Wikidata, the central storage for Wikimedia.

PORTE, Daniel; MENDONÇA, Manoel G.; FABBRI, Sandra Camargo Pinto Ferraz. The Use of Reading Technique and Visualization for Program Understanding. Online, Boston, Massachusetts, p. 386–391, jul. 2009. Disponível em: [https://www.researchgate.net/publication/221390090\\_The\\_Use\\_of\\_Reading\\_<](https://www.researchgate.net/publication/221390090_The_Use_of_Reading_<)

[Technique\\_and\\_Visualization\\_for\\_Program\\_Understanding](#). Acesso em: 31 out. 2017. Citado uma vez na página 56.

**Resumo:** Code comprehension is the basis for many other activities in software engineering. It is also time consuming and can greatly profit from tools that decrease the time that it usually consumes. This paper presents a tool named CRISTA that supports code comprehension through the application of Stepwise Abstraction. It uses a visual metaphor to represent the code and supports essential tasks for code reading, inspection and documentation. Three case studies were carried out to evaluate the tool with respect to usability and usefulness. In all of them the experiment participants considered that the tool facilitates code comprehension, inspection and documentation.

POST, Reinier. **Are Context Sensitive Grammar with Polynomial Complexity**

**Time?** Disponível em: <<https://cs.stackexchange.com/q/111573>>. Acesso em: 8 jul. 2019. Citado uma vez na página 33.

**Resumo:** Assuming a Deterministic Parser is the one which can parse unambiguous grammars in something like linear time complexity, while a NonDeterministic Parser can parse any grammar in some worst time complexity like  $O(n^5)$  or exponential. Parsing context-sensitive grammars is PSPACE-complete, so it takes exponential time in practice; it takes only polynomial time for growing context-sensitive grammars.

PUREWAL Jr., Tarsem S. Revisiting a Limit on Efficient Quantum Computation. *In:* PROCEEDINGS of the 44th Annual Southeast Regional Conference. Melbourne, Florida: ACM, 2006. (ACM-SE 44), p. 239–243. DOI: [10.1145/1185448.1185502](https://doi.acm.org/10.1145/1185448.1185502).

Disponível em: <[http://doi.acm.org/10.1145/1185448.1185502](https://doi.acm.org/10.1145/1185448.1185502)>. Citado uma vez na página 40.

**Resumo:** In this paper, we offer an exposition of a theorem originally due to Adleman, Demarrais and Huang that shows that the quantum complexity class BQP (Bounded-error Quantum Polynomial time) is contained in the classical counting class PP (Probabilistic Polynomial time). Our proof follows the one given by Fortnow and Rogers that relates quantum computing to counting complexity classes by way of GapP functions. The contribution of this paper is an exposition of an important result that assumes a minimal background in computational complexity theory and no knowledge of quantum mechanics.

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](https://doi.acm.org/10.1145/3183440.3195085). Disponível em: <[http://doi.acm.org/10.1145/3183440.3195085](https://doi.acm.org/10.1145/3183440.3195085)>. Citado uma vez na página 60.

**Resumo:** Fork-based development allows developers to start development from existing software repository by copying the code files. However, when the number of forks grows, contributions are not always visible to others, unless an explicit merge-back attempt is made. To solve this problem, we implemented Forks Insight ([www.forks-insight.com](http://www.forks-insight.com)) to help developers get an overview of forks on GitHub. The current release version focuses on simple analytics for the high level overview which is lightweight, scalable and practical. It has a user-friendly interactive web interface with features like searching and tagging.

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 uma vez nas páginas [32, 33](#).

**Resumo:** It is easy to know that there are LALR(K) grammars which are not LL(K) because any grammar with left recursion which is LALR(K), is not LL(K) because all LL(K) grammar must be left recursion free. And the opposite? Is there a LL(K) Grammar which is not LALR(K) Grammar? Do you have an example?

RIEFFEL, Eleanor; POLAK, Wolfgang. An Introduction to Quantum Computing for Non-physicists. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 32, n. 3, p. 300–335, set. 2000. ISSN 0360-0300. DOI: [10.1145/367701.367709](https://doi.acm.org/10.1145/367701.367709). Disponível em: <<https://doi.acm.org/10.1145/367701.367709>>. Citado uma vez na página [40](#).

**Resumo:** Richard Feynman's observation that certain quantum mechanical effects cannot be simulated efficiently on a computer led to speculation that computation in general could be done more efficiently if it used these quantum effects. This speculation proved justified when Peter Shor described a polynomial time quantum algorithm for factoring integers. In quantum systems, the computational space increases exponentially with the size of the system, which enables exponential parallelism. This parallelism could lead to exponentially faster quantum algorithms than possible classically. The catch is that accessing the results, which requires measurement, proves tricky and requires new nontraditional programming techniques. The aim of this paper is to guide computer scientists through the barriers that separate quantum computing from conventional computing. We introduce basic principles of quantum mechanics to explain where the power of quantum computers comes from and why it is difficult to harness. We describe quantum cryptography, teleportation, and dense coding. Various approaches to exploiting the power of quantum parallelism are explained. We conclude with a discussion of quantum error correction.

RONACHER, Armin. **Flask - The Python micro framework for building web applications.** Disponível em: <<https://github.com/pallets/flask>>. Acesso em: 29 nov. 2019. Citado uma vez na página [156](#).

**Resumo:** Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks. Flask offers suggestions, but doesn't enforce any dependencies or project layout. It is up to the developer to choose the tools and libraries they want to use. There are many extensions provided by the community that make adding new functionality easy.

ROSA, Evandro Chagas Ribeiro da. **QSystem: simulador quântico para Python.** 2019. Graduation Thesis submitted to the Computer Science Department – Federal University of Santa Catarina, Florianópolis, Santa Catarina, Brazil. [Departamento de Informática e Estatística]. Disponível em: <<https://tcc.inf.ufsc.br>>. Acesso em: 28 jun. 2019. Citado 2 vezes na página 40.

**Resumo:** Esse trabalho documenta a implementação de um simulador de computação quântica baseado no modelo de circuitos quânticos onde é possível executar uma computação quântica tanto em vetor de estado quanto em matriz densidade, possibilitando, assim, a simulação de erros quânticos. O simulador foi desenvolvido majoritariamente em C++ e entregue como um módulo de Python, denominado QSystem, desta forma, obtém uma boa performance ao mesmo tempo que é dinâmico para o uso. Toda a base teórica referente a computação quântica necessária para a implementação do simulador é apresentada nos primeiros capítulos.

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](https://doi.org/10.1145/3022671.2984018). Disponível em: <[https://www.researchgate.net/publication/311477527\\_Purposes\\_concepts\\_misfits\\_and\\_a\\_redesign\\_of\\_git](https://www.researchgate.net/publication/311477527_Purposes_concepts_misfits_and_a_redesign_of_git)>. Acesso em: 30 out. 2017. Citado uma vez na página 16.

**Resumo:** Git is a widely used version control system that is powerful but complicated. Its complexity may not be an inevitable consequence of its power but rather evidence of flaws in its design. To explore this hypothesis, we analyzed the design of Git using a theory that identifies concepts, purposes, and misfits. Some well-known difficulties with Git are described, and explained as misfits in which underlying concepts fail to meet their intended purpose. Based on this analysis, we designed a reworking of Git (called Gitless) that attempts to remedy these flaws. To correlate misfits with issues reported by users, we conducted a study of Stack Overflow questions. And to determine whether users experienced fewer complications using Gitless in place of Git, we conducted a small user study. Results suggest our approach can be profitable in identifying, analyzing, and fixing design problems.

ROSSUM, Guido van; WARSAW, Barry; COGHLAN, Nick. **Should a Line Break Before 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 uma vez na página 48.

**Resumo:** For decades the recommended style was to break after binary operators. But this can hurt readability in two ways: the operators tend to get scattered across different columns on the screen, and each operator is moved away from its operand and onto the previous line. Here, the eye has to do extra work to tell which items are added and which are subtracted.

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 6 vezes nas páginas 28, 30, 31, 41, 42.

**Resumo:** Normally, the concepts of time and space complexity for deterministic and nondeterministic machines are defined separately, although they share several similar theorems. Blum (1967, p. 324) define a “computational resource” using two axioms; this allows for an axiomatic treatment of the computational complexity, enabling us to provide a single proof for these analogous theorems. However, the Blum axioms are not defined in terms of deciders, but for integer function computers (that is, Turing machines that compute functions of the form  $f : N \rightarrow N$  (HOPCROFT; ULLMAN, 1979, p. 151)). Therefore, to define P and NP via Blum axioms, we need to see a decider as an integer function computer. The case where the machine is deterministic is easy, because it have a single sequence of computation. The nondeterministic case, which is more complex, is studied in this text. In this text, we define the concept of “nondeterministic function” in a way that the composition of nondeterministic functions correspond to the composition of the analogous deterministic functions.

SAMSON, Awe Ayodeji. 2-Head Pushdown Automata. **Procedia - Social and Behavioral Sciences**, Elsevier North-Holland, Inc., Department of Mathematics, Eastern Mediterranean University, Mersin 10 Turkey, Famagusta, North Cyprus, v. 195, p. 2037–2046, jun. 2015. DOI: [10.1016/j.sbspro.2015.06.225](https://doi.org/10.1016/j.sbspro.2015.06.225). Disponível em: <[https://www.researchgate.net/publication/282556609\\_2-Head\\_Pushdown\\_Automata](https://www.researchgate.net/publication/282556609_2-Head_Pushdown_Automata)>. Acesso em: 12 set. 2017. Citado uma vez na página 57.

**Resumo:** Finite state automata recognize regular languages which can be used in text processing, compilers, and hardware design. Two head finite automata accept linear context free languages. In addition, pushdown automata are able to recognize context free languages which can be used in programming languages and artificial intelligence. The finite automaton has deterministic and non-deterministic

version likewise the two head finite automata and the pushdown automata. The deterministic version of these machines is such that there is no choice of move in any situation while the non-deterministic version has a choice of move. In this research the 2-head pushdown automata are described which is more powerful than the pushdown automata and it is able to recognize some non-context free languages as well. During this work, the main task is to characterize these machines.

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](https://doi.org/10.1109/ICPC.2016.7503707). Disponível em: <[https://www.researchgate.net/publication/301685380\\_Improving\\_Code\\_Readability\\_Models\\_with\\_Textual\\_Features](https://www.researchgate.net/publication/301685380_Improving_Code_Readability_Models_with_Textual_Features)>. Acesso em: 1 nov. 2017. Citado uma vez nas páginas 47, 56.

**Resumo:** Code reading is one of the most frequent activities in software maintenance; before implementing changes, it is necessary to fully understand source code often written by other developers. Thus, readability is a crucial aspect of source code that may significantly influence program comprehension effort. In general, models used to estimate software readability take into account only structural aspects of source code, e.g., line length and a number of comments. However, source code is a particular form of text; therefore, a code readability model should not ignore the textual aspects of source code encapsulated in identifiers and comments. In this paper, we propose a set of textual features aimed at measuring code readability. We evaluated the proposed textual features on 600 code snippets manually evaluated (in terms of readability) by 5K+ people. The results demonstrate that the proposed features complement classic structural features when predicting code readability judgments. Consequently, a code readability model based on a richer set of features, including the ones proposed in this paper, achieves a significantly higher accuracy as compared to all of the state-of-the-art readability models.

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](https://doi.org/10.1145/158976.158989). Disponível em: <<http://doi.acm.org/10.1145/158976.158989>>. Citado 2 vezes nas páginas 19, 60.

**Resumo:** Rule-Based Systems in the legal domain are often obtained by formalizing legislation. We consider the addition of meta-knowledge in the form of meta-rules to such a system. Such an approach has many advantages both for control and for dealing with the intrinsic vagueness of legal rules. Legal computer

systems of different kinds have been proposed and built over the years. In this paper we shall present a legal reasoning system which uses concepts discussed in this paper. The system consists of a knowledge base, obtained by formalizing legislation, and uses a meta-rules mechanism for deduction and legal reasoning.

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 7 vezes nas páginas 49, 50, 87.

**Resumo:** Ever concerned about how your code looks like? Ever heard of different indenting styles, for example K&R? Ever received code from someone else who didn't care about code formatting? Ever tried to configure a code indenter to convert such code to your coding style? Ever got bored by that tedious "changing a parameter"- "call the indenter"- "try and error" procedure? Help is close to you. UniversalIndentGUI offers a live preview for setting the parameters of nearly any indenter. You change the value of a parameter and directly see how your reformatted code will look like. Save your beauty looking code or create an anywhere usable batch/shell script to reformat whole directories or just one file even out of the editor of your choice that supports external tool calls.

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 uma vez na página 49.

SELLINK, Alex; VERHOEF, Chris. Towards automated modification of legacy assets. **Annals of Software Engineering**, Kluwer Academic Publishers, Red Bank, NJ, USA, v. 9, p. 315–336, jan. 2000. DOI: [10.1023/A:1018941228255](https://doi.org/10.1023/A:1018941228255). Disponível em: <[https://www.researchgate.net/publication/2825635\\_Towards\\_Automated\\_Modification\\_of\\_Legacy\\_Assets](https://www.researchgate.net/publication/2825635_Towards_Automated_Modification_of_Legacy_Assets)>. Acesso em: 11 set. 2017. Citado uma vez na página 56.

**Resumo:** In this paper we argue that there is a necessity for automating modifications to legacy assets. We propose a five layered process for the introduction and employment of tool support that enables automated modification to entire legacy systems. Furthermore, we elaborately discuss each layer on a conceptual level, and we make appropriate references to sources where technical contributions supporting that particular layer can be found. We sketch the perspective that more and more people working in the software engineering area will be contributing to working on existing systems and/or tools to support such work.

SHANG, Haichuan; KITSUREGAWA, Masaru. Efficient Breadth-first Search on Large

Graphs with Skewed Degree Distributions. In: PROCEEDINGS of the 16th International Conference on Extending Database Technology. Genoa, Italy: ACM, 2013. (EDBT '13), p. 311–322. DOI: [10.1145/2452376.2452413](https://doi.org/10.1145/2452376.2452413). Disponível em: <<http://doi.acm.org/10.1145/2452376.2452413>>. Citado uma vez na página 44.

**Resumo:** Many recent large-scale data intensive applications are increasingly demanding efficient graph databases. Distributed graph algorithms, as a core part of practical graph databases, have a wide range of important applications, but have been rarely studied in sufficient detail. These problems are challenging as real graphs are usually extremely large and the intrinsic character of graph data, lacking locality, causes unbalanced computation and communication workloads. In this paper, we explore distributed breadth-first search algorithms with regards to large-scale applications. We propose DPC (Degree-based Partitioning and Communication), a scalable and efficient distributed BFS algorithm which achieves high scalability and performance through novel balancing techniques between computation and communication. In experimental study, we compare our algorithm with two state-of-the-art algorithms under the Graph500 benchmark with a variety of settings. The result shows our algorithm significantly outperforms the existing algorithms under all the settings.

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 uma vez na página 60.

**Resumo:** Parse any context-free grammar, FAST and EASY! Beginners: Lark is not just another parser. It can parse any grammar you throw at it, no matter how complicated or ambiguous, and do so efficiently. It also constructs a parse-tree for you, without additional code on your part. Experts: Lark implements both Earley(SPPF) and LALR(1), and several different lexers, so you can trade-off power and speed, according to your requirements. It also provides a variety of sophisticated features and utilities.

SHINAN, Erez. **Design a new Lark cheatsheet.** 2018. Disponível em: <<https://github.com/lark-parser/lark/issues/195>>. Acesso em: 30 set. 2019. Citado 3 vezes nas páginas 23, 68.

**Resumo:** The existing cheatsheet is here: [https://github.com/lark-parser/lark/blob/master/docs/lark\\_cheatsheet.pdf](https://github.com/lark-parser/lark/blob/master/docs/lark_cheatsheet.pdf). The design and format are somewhat lacking.

SHINAN, Erez. **Grammar Reference.** 2019. Disponível em: <<https://lark-parser.readthedocs.io/en/latest/grammar/>>. Acesso em: 30 set. 2019. Citado 3 vezes nas páginas 23, 68.

**Resumo:** In Lark, a terminal may be a string, a regular expression, or a concat-

tenation of these and other terminals. Each rule is a list of terminals and rules, whose location and nesting define the structure of the resulting parse-tree. A parsing algorithm is an algorithm that takes a grammar definition and a sequence of symbols (members of the alphabet), and matches the entirety of the sequence by searching for a structure that is allowed by the grammar.

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 4 vezes nas páginas 19, 39, 60.

**Resumo:** Lark extends the traditional YACC-based architecture with a contextual lexer, which automatically provides feedback from the parser to the lexer, making the LALR(1) algorithm stronger than ever. The contextual lexer communicates with the parser, and uses the parser's lookahead prediction to narrow its choice of tokens. So at each point, the lexer only matches the subgroup of terminals that are legal at that parser state, instead of all of the terminals. It's surprisingly effective at resolving common terminal collisions, and allows to parse languages that LALR(1) was previously incapable of parsing. This is an improvement to LALR(1) that is unique to Lark.

SHOR, P. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. **SIAM Review**, v. 41, n. 2, p. 303–332, 1999. DOI: 10.1137/S0036144598347011. eprint: <https://doi.org/10.1137/S0036144598347011>. Disponível em: <<https://doi.org/10.1137/S0036144598347011>>. Citado uma vez na página 40.

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](https://www.researchgate.net/publication/320120365_The_Command_Line_Interface)>. Acesso em: 10 out. 2017. Citado uma vez na página 61.

**Resumo:** This is a very exciting chapter. While command line interfaces (CLIs) may not have the cachet of modern graphical user interfaces (GUIs), especially those of phones or tablets, the CLI is still a remarkably useful and effective user interface. This chapter details the design and implementation of the command line interface for pdCalc. By the end of this chapter, we will, for the first time, have a functioning (albeit feature incomplete) calculator, which is a significant milestone in our development.

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](https://doi.org/10.1145/582153.582173). Disponível em: <<http://doi.acm.org/10.1145/582153.582173>>. Citado uma vez na página 21.

**Resumo:** An automatic syntax error handling technique applicable to LR parsing is presented and analyzed. The technique includes a "phrase-level" error recovery strategy augmented with certain additional features such as "local correction". Attention has also been paid to diagnostic aspects, i.e. the automatic generation of error message texts. The technique has been implemented in the compiler writing system HLP (Helsinki Language Processor), and some promising experimental results have been obtained by testing the technique with erroneous student-written Algol and Pascal programs.

SIPSER, Michael. Introduction to the Theory of Computation. In: New York, NY, USA: Pearson Education, 2012. ISBN 978-1133187790. DOI: [10.1145/230514.571645](https://doi.org/10.1145/230514.571645). Disponível em: <<http://doi.acm.org/10.1145/230514.571645>>. Citado 9 vezes nas páginas 28–31, 39, 42, 43.

**Resumo:** Gain a clear understanding of even the most complex, highly theoretical computational theory topics in the approachable presentation found only in the market-leading INTRODUCTION TO THE THEORY OF COMPUTATION, 3E. The number one choice for today's computational theory course, this revision continues the book's well-known, approachable style with timely revisions, additional practice, and more memorable examples in key areas. A new first-of-its-kind theoretical treatment of deterministic context-free languages is ideal for a better

SKINNER, Jon. **SUBLIME TEXT 3 DOCUMENTATION, Color Schemes**. 2015. Disponível em: <[https://www.sublimetext.com/docs/3/color\\_schemes.html](https://www.sublimetext.com/docs/3/color_schemes.html)>. Acesso em: 30 nov. 2019. Citado 3 vezes nas páginas 52, 55.

**Resumo:** The highlighting of source code and prose in Sublime Text is controlled by a color scheme. A color scheme assigns colors and font styles to scopes, which are assigned to the text by the syntax. The rest of the look of the user interface is controlled by the theme. The theme controls such elements as buttons select lists, the sidebar and tabs. Sublime Text color schemes are implemented using .sublime-color-scheme files, containing JSON. Sublime Text also supports a subset of features using the TextMate .tmTheme format.

SKINNER, Jon. **SUBLIME TEXT 3 DOCUMENTATION, Scope Naming**. 2015. Disponível em: <[https://www.sublimetext.com/docs/3/scope\\_naming.html](https://www.sublimetext.com/docs/3/scope_naming.html)>. Acesso em: 17 set. 2019. Citado 4 vezes nas páginas 52, 55, 75, 85.

**Resumo:** Syntax definitions and color schemes in Sublime Text interact through the use of scope names. Scopes are dotted strings, specified from least-to-most specific. For example, the if keyword in PHP could be specified via the scope name

keyword.control.php. Sublime Text supports TextMate language grammars, and inherited its default syntaxes from various open-source bundles. The TextMate language grammar documentation provided a base set of scope names that have been slowly expanded and changed by the community.

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 22 vezes nas páginas 52–54, 75, 88, 89.

**Resumo:** Sublime Text can use both .sublime-syntax and .tmLanguage files for syntax highlighting. This document describes .sublime-syntax files. Sublime Syntax files are YAML files with a small header, followed by a list of contexts. Each context has a list of patterns that describe how to highlight text in that context, and how to change the current text.

SMITH, Thomas. **branch temporarily highlights wrong 3042**. 2019. Disponível em: <[https://github.com/sublimehq/sublime\\_text/issues/3042](https://github.com/sublimehq/sublime_text/issues/3042)>. Acesso em: 30 nov. 2019. Citado uma vez na página 53.

**Resumo:** With certain inputs, branch rules can temporarily be in the wrong state. Open a new tab and use the above syntax. Then type exactly A, return, return, Z. Expected behavior: A should be green. Actual behavior: A is red. However, if you modify the view, A will be recolored green. Also, if you only type one newline, A will be green initially.

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](https://doi.acm.org/10.1145/1943371.1943391). Disponível em: <<http://doi.acm.org/10.1145/1943371.1943391>>. Citado uma vez na página 21.

**Resumo:** The LR(1) Parser Generation System generates full LR(1) parsers that are comparable in speed and size to those generated by LALR(1) parser generators, such as yacc [5]. In addition to the inherent advantages of full LR(1) parsing, it contains a number of novel features. This paper discusses three of them in detail: an LR(1) grammar specified automatic error recovery algorithm, oracles, and generic tokens. The error recovery algorithm depends on the fact that full LR(1) parse tables preserve context. Oracles are pieces of code that are defined in a grammar and that are executed between the scanner and parser. They are used to resolve token ambiguities, including semantic ones. Generic tokens are used to replace syntactically identical tokens with a single token, which is, in effect, a variable representing a set of tokens.

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](https://doi.org/10.1007/978-3-642-05031-2_18). Disponível em: <[https://www.researchgate.net/publication/221047094\\_Debugging\\_into\\_Examples](https://www.researchgate.net/publication/221047094_Debugging_into_Examples)>. Acesso em: 31 out. 2017. Citado 4 vezes nas páginas 17, 56, 60, 86, 156.

**Resumo:** Enhancing and maintaining a complex software system requires detailed understanding of the underlying source code. Gaining this understanding by reading source code is difficult. Since software systems are inherently dynamic, it is complex and time consuming to imagine, for example, the effects of a method's source code at run-time. The inspection of software systems during execution, as encouraged by debugging tools, contributes to source code comprehension. Leveraged by test cases as entry points, we want to make it easy for developers to experience selected execution paths in their code by debugging into examples. We show how links between test cases and application code can be established by means of dynamic analysis while executing regular tests.

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 uma vez na página 47.

**Resumo:** I've looked throughout Preferences -> Java -> Code Style -> Formatter and can't find any way to get eclipse to format my code breaking lines after a . ... apart from modifying the formatter code itself. (And that means you are not running the standard formatter anymore!)

SYREL, Aliaksei. Towards a Live, Moldable Code Editor. In: COMPANION to the First International Conference on the Art, Science and Engineering of Programming. Brussels, Belgium: ACM, abr. 2017. (Programming '17), 43:1–43:3. DOI: [10.1145/3079368.3079376](https://doi.org/10.1145/3079368.3079376). Disponível em: <[https://www.researchgate.net/publication/318873843\\_Towards\\_a\\_live\\_moldable\\_code\\_editor](https://www.researchgate.net/publication/318873843_Towards_a_live_moldable_code_editor)>. Acesso em: 2 nov. 2017. Citado uma vez na página 57.

**Resumo:** Creating and evolving object-oriented applications requires developers to reason about source code and run-time state. Integrated development environments (IDEs) are tools that support developers in this activity. Many mainstream IDEs, however, focus on code editors that promote reading of static text even in the debugger. This affects program comprehension, as developers have to manually link the static code with the run-time objects. In this work we explore how to address this problem through a moldable code editor that enables developers to select executable snippets of code and replace them with graphical views of the resulting objects. Each object can define multiple views that developers can select. This way objects and code coexist in the same editor.

TALIRONGAN, Hidear; SISON, Ariel M.; MEDINA, Ruji P. A New Advanced Encryption Standard-Butterfly Effect in Protecting Image of Copyright Piracy. *In: PROCEEDINGS of the 6th International Conference on Information Technology: IoT and Smart City.* Hong Kong, Hong Kong: ACM, 2018. (ICIT 2018), p. 214–218. DOI: [10.1145/3301551.3301603](https://doi.acm.org/10.1145/3301551.3301603). Disponível em: <[http://doi.acm.org/10.1145/3301551.3301603](https://doi.acm.org/10.1145/3301551.3301603)>. Citado uma vez na página 88.

**Resumo:** Cryptography is growing every day where secure communication on the internet is very much concerned encryption is a very hot topic when it comes to keeping the data security and integrity several photographic images are published to particular interests which makes it more prone to copyright piracy of images this paper presents a novelty on advanced encryption standard AES algorithm with butterfly effect be implementation which is imperative in protecting the copyright of the digital image the study is implemented and simulated using p h p this study is analyzed using the five metrics namely the visual analysis file size luminosity histogram comparison by pixel and hamming distance in the file sizes there are variations in the size of file where it shows that the average value of the percentage changes to 23 85 from the original to the encrypted image and 1 45 percentage value from the original to the decrypted image this proves that there is a change in the primary data encryption and decryption process however the zero results in the hamming distance indicate that there is no changes in the image from the original until it is being encrypted and decrypted in the histogram analysis it yields a match or the same result from the original encrypted and decrypted file which interprets that the image did not changed from these results AES be encryption techniques can be used for copyright protection of the image.

THOMPSON, Errol *et al.* Code Classification as a Learning and Assessment Exercise for Novice Programmers. **Proceedings of the 19th Annual Conference of the National Advisory Committee on Computing Qualifications**, Online, NACCQ, Wellington, New Zealand, p. 291–298, jul. 2006. Disponível em: <[https://www.researchgate.net/publication/255948009\\_Code\\_Classification\\_as\\_a\\_Learning\\_and\\_Assessment\\_Exercise\\_for\\_Novice\\_Programmers](https://www.researchgate.net/publication/255948009_Code_Classification_as_a_Learning_and_Assessment_Exercise_for_Novice_Programmers)>. Acesso em: 31 out. 2017. Citado uma vez na página 56.

**Resumo:** When students are given code that is very similar in structure or purpose, how well do they actually recognise the similarities and differences? As part of the BRACElet project, a multi-institutional investigation into reading and comprehension skills of novice programmers, students were asked to classify four code segments that found the minimum or maximum in an array of numbers. This paper reports on the analysis of responses to this question and draws conclusions about the students' ability to recognise the similarities and differences in example code. It then raises questions with respect to an approach to teaching that uses

variations in code examples. Received Citrus Award for Collaborative Research and highly recommended in the Best Paper awards

TORÁN, Jacobo. Complexity Classes Defined by Counting Quantifiers. **J. ACM**, ACM, New York, NY, USA, v. 38, n. 3, p. 752–773, dez. 1991. ISSN 0004-5411. DOI: 10.1145/116825.116858. Disponível em:

<<http://doi.acm.org/10.1145/116825.116858>>. Citado uma vez na página 29.

**Resumo:** The polynomial-time counting hierarchy, a hierarchy of complexity classes related to the notion of counting is studied. Some of their structural properties are investigated, settling many open questions dealing with oracle characterizations, closure under Boolean operations, and relations with other complexity classes. A new combinatorial technique to obtain relativized separations for some of the studied classes, which imply absolute separations for some logarithmic time bounded complexity classes, is developed.

VALDECANTOS, Héctor Adrián. **An empirical study on code comprehension DCI compared to OO**. Ago. 2016. National University of Tucuman, Tucumán Province, Argentina. DOI: 10.13140/RG.2.2.29331.48169. Disponível em:  
<[https://www.researchgate.net/publication/308314679\\_An\\_empirical\\_study\\_on\\_code\\_comprehension\\_DCI\\_compared\\_to\\_OO](https://www.researchgate.net/publication/308314679_An_empirical_study_on_code_comprehension_DCI_compared_to_OO)>. Acesso em: 2 nov. 2017. Citado uma vez na página 57.

**Resumo:** Comprehension of source code affects software development, especially its maintenance where reading code is the most time consuming performed activity. A programming paradigm imposes a style of arranging the source code that is aligned with a way of thinking toward a computable solution. Then, a programming paradigm with a programming language represents an important factor for source code comprehension. Object-Oriented (OO) is the dominant paradigm today. Although, it was criticized from its beginning and recently an alternative has been proposed. In an OO source code, system functions cannot escape outside the definition of classes and their descriptions live inside multiple class declarations. This results in an obfuscated code, a lost sense the run-time, and in a lack of global knowledge that weaken the understandability of the source code at system level. A new paradigm is emerging to address these and other OO issues, this is the Data Context Interaction (DCI) paradigm. We conducted the first human subject related controlled experiment to evaluate the effects of DCI on code comprehension compared to OO. We looked for correctness, time consumption, and focus of attention during comprehension tasks. We also present a novel approach using metrics from Social Network Analysis to analyze what we call the Cognitive Network of Language Elements (CNLE) that is built by programmers while comprehending a system. We consider this approach useful to understand source code properties uncovered from code reading cognitive tasks. The results

obtained are preliminary in nature but indicate that DCI approach produces more comprehensible source code and promotes a stronger focus the attention in important files when programmers are reading code during program comprehension. Regarding reading time spent on files, we were not able to indicate with statistical significance which approach allows programmers to consume less time.

VASUDEVAN, Naveneetha; TRATT, Laurence. Detecting Ambiguity in Programming Language Grammars. In: ERWIG, Martin; PAIGE, Richard F.; VAN WYK, Eric (Ed.).

**Software Language Engineering**. Cham: Springer International Publishing, 2013.

p. 157–176. Citado uma vez na página 41.

**Resumo:** Ambiguous Context Free Grammars (CFGs) are problematic for programming languages, as they allow inputs to be parsed in more than one way. In this paper, we introduce a simple non-deterministic search-based approach to ambiguity detection which non-exhaustively explores a grammar in breadth for ambiguity. We also introduce two new techniques for generating random grammars – Boltzmann sampling and grammar mutation – allowing us to test ambiguity detection tools on much larger corpuses than previously possible. Our experiments show that our breadth-based approach to ambiguity detection performs as well as, and generally better, than extant tools.

VEERMAN, Niels. Automated maintenance of a software portfolio. **Science of Computer Programming - Special issue on source code analysis and manipulation (SCAM 2005)**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, v. 62, p. 287–317, out. 2006. DOI: [10.1016/j.scico.2006.04.006](https://doi.org/10.1016/j.scico.2006.04.006).

Disponível em: <[https://www.researchgate.net/publication/222831264\\_Automated\\_maintenance\\_of\\_a\\_software\\_portfolio](https://www.researchgate.net/publication/222831264_Automated_maintenance_of_a_software_portfolio)>. Acesso em: 12 set. 2017. Citado uma vez na página 56.

**Resumo:** This is an experience report on automated mass maintenance of a large Cobol software portfolio. A company in the financial services and insurance industry upgraded their database system to a new version, affecting their entire software portfolio. The database system was accessed by the portfolio of 45 systems, totalling nearly 3000 programs and covering over 4 million lines of Cobol code. We upgraded the programs to the new database version using several automatic tools, and we performed an automated analysis supporting further manual modifications by the system experts. The automatic tools were built using a combination of lexical and syntactic technology, and they were deployed in a mass update factory to allow large-scale application to the software portfolio. The updated portfolio has been accepted and taken into production by the company, serving over 600 employees with the new database version. In this paper, we discuss the automated upgrade from problem statement to project costs.

VEERMAN, Niels. Automated Mass Maintenance of Software Assets. **11th European Conference on Software Maintenance and Reengineering (CSMR'07)**, IEEE Computer Society, Amsterdam, The Netherlands, mar. 2007. DOI: [10.1109/CSMR.2007.15](https://doi.org/10.1109/CSMR.2007.15). Disponível em: <[https://www.researchgate.net/publication/221569579\\_Automated\\_Mass\\_Maintenance\\_of\\_Software\\_Assets](https://www.researchgate.net/publication/221569579_Automated_Mass_Maintenance_of_Software_Assets)>. Acesso em: 11 set. 2017. Citado uma vez na página [56](#).

**Resumo:** This is a research summary of a PhD project in the area of massive software maintenance automation. We explain the context, approach, and contributions.

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](https://doi.org/10.1109/FIE.1999.841580). Citado uma vez na página [45](#).

**Resumo:** CS first-year students expect the user interface of programming environments to be similar to that of common PC applications. A natural evolution of educational programming environments consists in incorporating many of their user-friendly facilities. The authors concentrate in this paper on the facilities that WinHipe, an environment for functional programming, provides to students for customizing the visualization of expressions. Expressions can be either pretty-printed as text or displayed graphically, showing drawings of lists and binary trees. Besides, fonts, sizes, colors, lines and distances are parameters that can be customized for any visualization. Finally, the visualization of large expressions can be simplified to show only their most relevant parts. Students obtain several benefits from customization facilities. They feel more comfortable with the programming environment WinHipe, because they can develop more readable programs written "in their personal style". Students can also experiment at small effort with different formats, becoming proficient in style issues. Finally customization facilities allow making clear in a course on programming languages the relevant role of visualization in programming tools and their relative independence from language syntax.

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](https://doi.org/10.1145/1111542.1111569). Disponível em: <<http://doi.acm.org/10.1145/1111542.1111569>>. Citado uma vez na página [70](#).

**Resumo:** The design of metaprogramming languages requires appreciation of the tradeoffs that exist between important language characteristics such as safety

properties, expressive power, and succinctness. Unfortunately, such tradeoffs are little understood, a situation we try to correct by embarking on a study of metaprogramming language tradeoffs using tools from computability theory. Safety properties of metaprograms are in general undecidable; for example, the property that a metaprogram always halts and produces a type-correct instance is  $\Pi_2^0$ -complete. Although such safety properties are undecidable, they may sometimes be captured by a restricted language, a notion we adapt from complexity theory. We give some sufficient conditions and negative results on when languages capturing properties can exist: there can be no languages capturing total correctness for metaprograms, and no ‘functional’ safety properties above  $\Sigma_3^0$  can be captured. We prove that translating a metaprogram from a general-purpose to a restricted metaprogramming language capturing a property is tantamount to proving that property for the metaprogram.

WICKHAM, Hadley. **The tidyverse style guide**. 2017. Disponível em: <<https://style.tidyverse.org/>>. Acesso em: 9 jul. 2019. Citado uma vez na página 47.

**Resumo:** Good coding style is like correct punctuation: you can manage without it, but it sure makes things easier to read. Just as with punctuation, while there are many code styles to choose from, some are more reader-friendly than others. The style presented here, which is used throughout the tidyverse, is derived from Google’s R style guide and has evolved considerably over the years.

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](https://doi.acm.org/10.1145/362686.362695). Disponível em: <[http://doi.acm.org/10.1145/362686.362695](https://doi.acm.org/10.1145/362686.362695)>. Citado 4 vezes nas páginas 31, 33, 37, 38.

**Resumo:** This paper presents a canonical form for context-sensitive derivations and a parsing algorithm which finds each context-sensitive analysis once and only once. The amount of memory required by the algorithm is essentially no more than that required to store a single complete derivation. In addition, a modified version of the basic algorithm is presented which blocks infinite analyses for grammars which contain loops. The algorithm is also compared with several previous parsers for context-sensitive grammars and general rewriting systems, and the difference between the two types of analyses is discussed. The algorithm appears to be complementary to an algorithm by S. Kuno in several respects, including the space-time trade-off and the degree of context dependence involved.

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](https://doi.org/10.1145/3297280.3297433). Disponível em: <<http://doi.acm.org/10.1145/3297280.3297433>>. Citado uma vez na página 77.

**Resumo:** CPEG is an extended parsing expression grammar with regex-like capture annotation. Two annotations (capture and left-folding) allow a flexible construction of syntax trees from arbitrary parsing patterns. More importantly, CPEG is designed to guarantee structural constraints of syntax trees for any input strings. This reduces the amount of user code needed to check whether the intended elements exist. To represent the structural constraints, we focus on regular expression types, a variant formalism of tree automata, which have been intensively studied in the context of XML schemas. Regular expression type is inferred from a given CPEG by the type inference that is formally developed in this paper. We prove the soundness and the uniqueness of the type inference. The type inference enables a CPEG to serve both as a syntactic specification of the input and a schematic specification of the output.

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](https://doi.org/10.1002/9781119419310.ch9). Disponível em: <[https://www.researchgate.net/publication/316657029\\_Using\\_Source\\_Control](https://www.researchgate.net/publication/316657029_Using_Source_Control)>. Acesso em: 3 nov. 2017. Citado uma vez na página 19.

**Resumo:** This chapter focuses on continuous integration (CI) servers that act as the cement between all other processes and convert them into an automated life cycle. It explains how to download and install CI server. The chapter explores how to set up a build job from a Git repository, how to trigger a build cycle on every commit, and how to publish one's app automatically to Google Play. CI servers are very flexible and easy to integrate and can handle Android projects that use make files, Maven, or Gradle. One needs to choose one of those to fully integrate his/her project with a CI server. Version control systems are another crucial part of the CI process. Each code commit triggers a build process that results in compilation, running tests, and packaging the app on the CI server. The chapter also focuses on installing Jenkins.

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](https://doi.org/10.1002/9781119419310.ch9). Disponível em: <[https://www.researchgate.net/publication/316657029\\_Using\\_Source\\_Control](https://www.researchgate.net/publication/316657029_Using_Source_Control)>. Acesso em: 3 nov. 2017. Citado uma vez na página 16.

**Resumo:** Git is currently the most widely accepted source control system among Android developers, and has built-in support for Android Studio. This chapter covers how to create a Git repository, add files to it, and perform commits. Unlike other systems that just watch changes in the file system and commit changes to

the version control server, Git runs on a client computer and changes need to be committed locally first. This way, a developer can revert any change/branch or version locally through Git. The local Git can push the set of changes committed to network Git servers. Android Studio comes with Git support. However, one may still need to install Git to be able to use it through the command line. GitHub is a popular Git-based project-hosting site that offers free hosting for public repositories. One reason for GitHub's popularity is the available easy-to-use tools for Git.

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](https://doi.org/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](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 uma vez nas páginas 45, 56.

**Resumo:** We present an empirical study using eye tracking equipment to understand how novices read source code in the context of two introductory programming classes. Our main goal is to begin to understand how novices read source code and to determine if we see any improvement in program comprehension as the course progresses. The results indicate that novices put in more effort and had more difficulty reading source code as they progress through the course. However, they are able to partially comprehend code at a later point in the course. The relationship between fixation counts and durations is linear but shows more clusters later in the course, indicating groups of students that learned at the same pace. The results also show that we did not see any significant shift in learning (indicated by the eye tracking metrics) during the course, indicating that there might be more than one course that needs to be taken over the course of a few years to realize the significance of the shift. We call for more studies over a student's undergraduate years to further learn about this shift.

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](https://doi.org/10.1145/2491411.2491454). Disponível em: <[http://doi.acm.org/10.1145/2491411.2491454](https://doi.acm.org/10.1145/2491411.2491454)>. Citado 4 vezes nas páginas 39, 51, 53, 70.

**Resumo:** Model checking is an established method for verifying behavioral properties of system models. But model checkers tend to support low-level modeling

languages that require intricate models to represent even the simplest systems. Modeling complexity arises in part from the need to encode domain knowledge at relatively low levels of abstraction. In this paper, we demonstrate that formalized domain knowledge can be reused to raise the abstraction level of model and property specifications, and hence the effectiveness of model checking. We describe a novel method for domain-specific model checking called cascading verification that uses composite reasoning over high-level system specifications and formalized domain knowledge to synthesize both low-level system models and their behavioral properties for verification. In particular, model builders use a high-level domain-specific language (DSL) based on YAML to express system specifications that can be verified with probabilistic model checking. Domain knowledge is encoded in the Web Ontology Language (OWL), the Semantic Web Rule Language (SWRL) and Prolog, which are used in combination to overcome their individual limitations. A compiler then synthesizes models and properties for verification by the probabilistic model checker PRISM. We illustrate cascading verification for the domain of uninhabited aerial vehicles (UAVs), for which we have constructed a prototype implementation. An evaluation of this prototype reveals nontrivial reductions in the size and complexity of input specifications compared to the artifacts synthesized for PRISM.

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](https://doi.acm.org/10.1145/2555289.2555301). Disponível em:

<<http://doi.acm.org/10.1145/2555289.2555301>>. Citado uma vez na página 155.

**Resumo:** Parallelizing HTML parsing is challenging due to the complexities of HTML documents and the inherent dependencies in its parsing algorithm. As a result, despite numerous studies in parallel parsing, HTML parsing remains sequential today. It forms one of the final barriers for fully parallelizing browser operations to minimize the browser's response time—an important variable for user experiences, especially on portable devices. This article provides a comprehensive analysis on the special complexities of parallel HTML parsing and presents a systematic exploration in overcoming those difficulties through specially designed speculative parallelizations. This work develops, to the best of our knowledge, the first pipelining and data-level parallel HTML parsers. The data-level parallel parser, named HPar, achieves up to 2.4× speedup on quadcore devices. This work demonstrates the feasibility of efficient, parallel HTML parsing for the first time and offers a set of novel insights for parallel HTML parsing

# Apêndices

[⇐](#) |  [←](#)

## 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 39](#)). Uma utiliza a metalinguagem para adicionar cores ([Código 37](#)), como feito em editores de texto e a outra realiza a formatação de código-fonte ([Código 38](#)).

Nos [Códigos 27](#) a [29](#), 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 35](#)). Para manter a implementação simples, tanto o [Código 27](#) quanto o [Código 28](#) 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 31](#) e [33](#), 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 37](#), quanto para a escrita do [Código 38](#) por con-

sequência, 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 “&lt;”.

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. Portando, 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, 2016a](#)), “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 26](#))
- 3) python3 main\_formatter.py ([Código 27](#))
- 4) python3 main\_highlighter.py ([Código 28](#))

## Código 26 – Arquivo “source/requirements.txt”

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

## Código 27 – 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 example_settings = {
63     "if.statement.body" : 2,
64 }
65
66 backend = code_formatter.Backend( code_formatter.SingleSpaceFormatter,
67     ↳ abstract_syntax_tree, example_program, example_settings )
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/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 29 – 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 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 30](#), encontra-se o arquivo HTML gerado pelo programa de exemplo ([Código 27](#)). Nas [Figuras 24 e 25](#), encontram-se as imagens geradas pelo programa “`main_formatter.py`” ([Código 27](#)). No [Código 31](#), encontra-se a saída da linha de comando que se obtém ao realizar a execução do programa de exemplo ([Código 27](#)).

Ambas as [Figuras 24 e 25](#) quanto o [Código 31](#) representam os mesmos dados, mas mostrados de formas diferentes (*figura versus textual*) e com níveis de detalhe diferentes. No [Código 31](#), são mostrado os nós-folhas das árvores com maior nível de detalhe. Já nas [Figuras 24 e 25](#), os nós-folhas são mostrados com simplificações para que a figura possa ser vista em uma única tela.

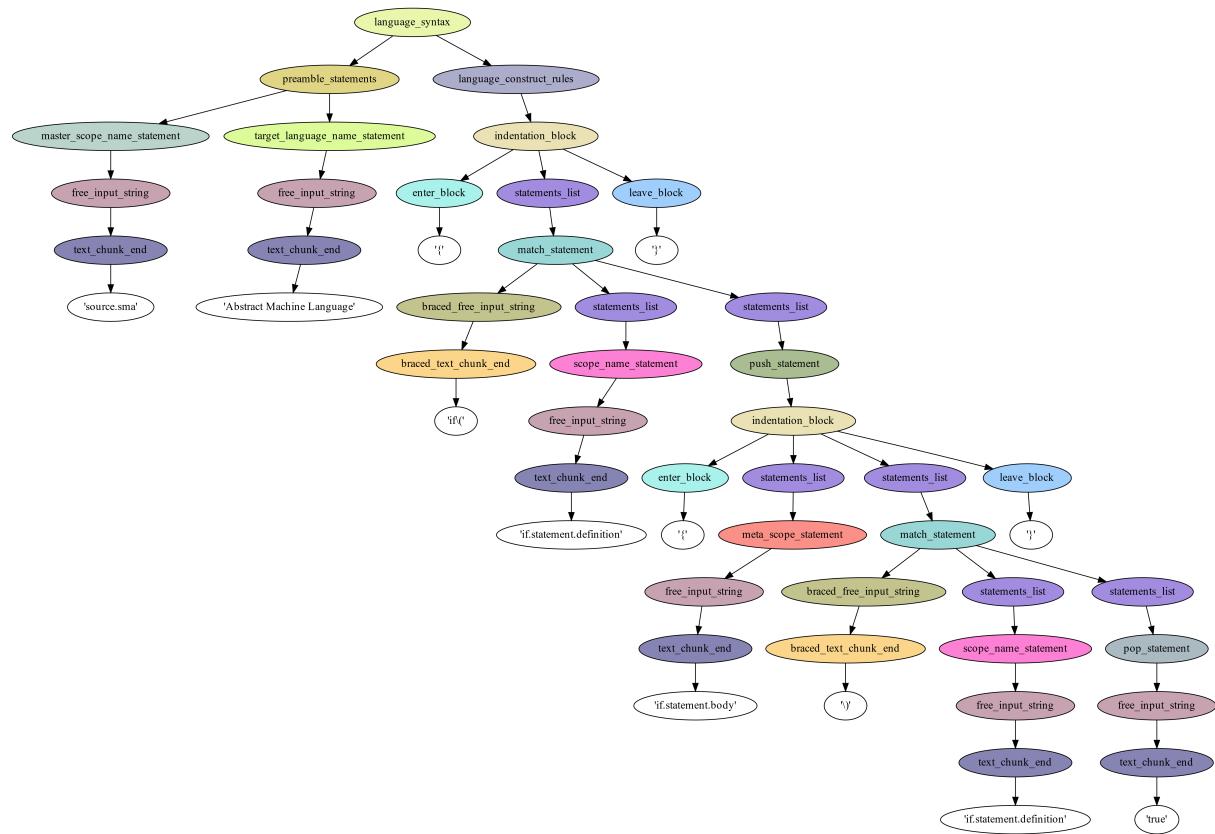
Código 30 – Arquivo HTML gerado pelo programa de exemplo “`main_formatter.py`”

```
1 <!DOCTYPE html><html><head><title>Abstract Machine Language -  
→ source.sma</title></head>  
2 <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>
```

Código 31 – 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
```

Figura 24 – Árvore Sintática “main\_formatter\_syntax\_tree.png”



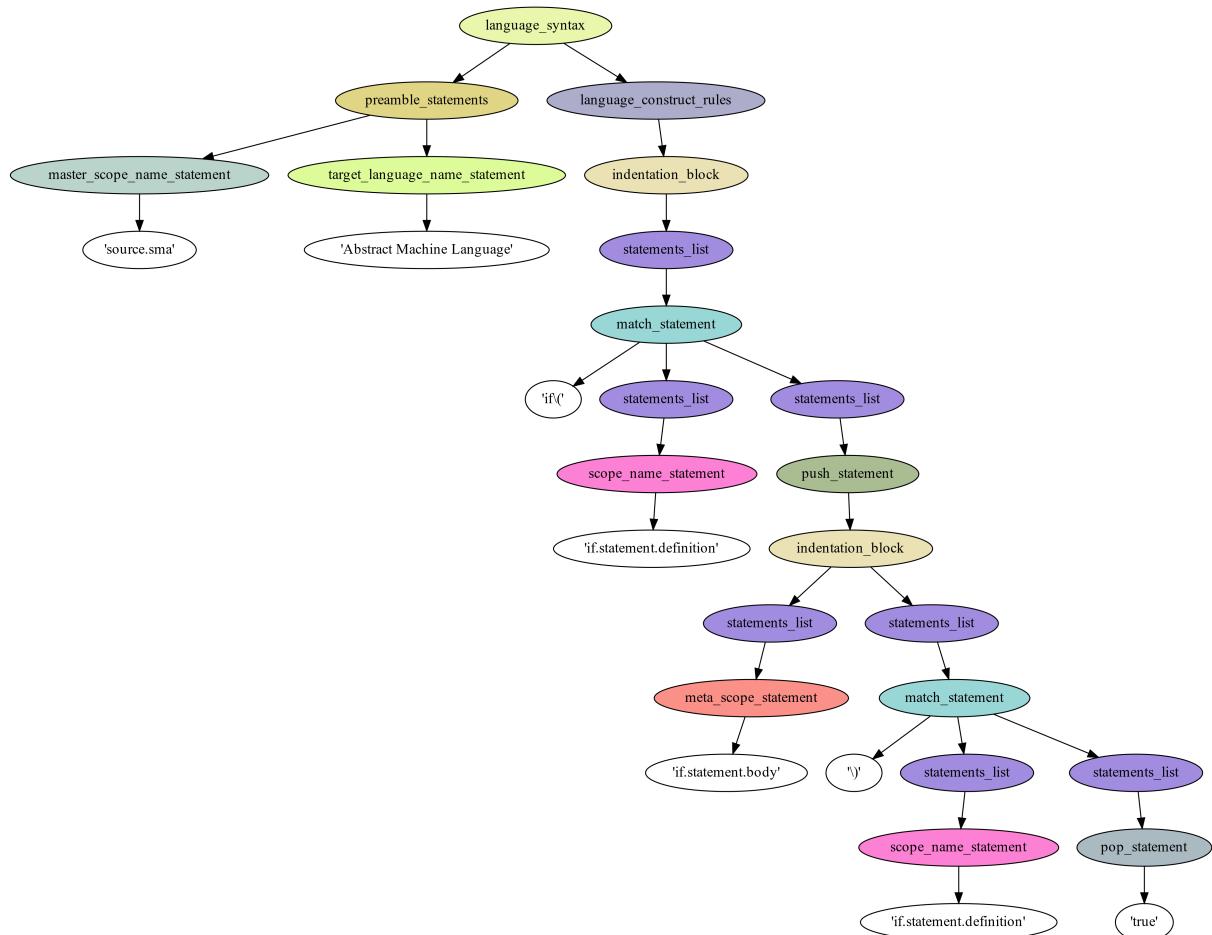
Fonte: Própria

```

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
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

```

Figura 25 – Árvore Sintática Abstrata “main\_formatter\_abstract\_syntax\_tree.png”



Fonte: Própria

```

29      text_chunk_end
30      →  [@7,159:175='if.statement.body'<TEXT_CHUNK_END_>,7:25]
31      statements_list
32      match_statement
33      braced_free_input_string
34      braced_text_chunk_end
35      →  [@8,196:197='\\)'<BRACED_TEXT_CHUNK_END_>,8:20]
36      statements_list
37      scope_name_statement
38      free_input_string
39      text_chunk_end      [@9,224:246='if.statement.definitio_]
40      →  n'<TEXT_CHUNK_END_>,9:24]
41      statements_list
42      pop_statement
43      free_input_string
44      text_chunk_end
45      →  [@10,269:272='true'<TEXT_CHUNK_END_>,10:22]

```

```
42         leave_block      '@11,296:296=' }'<CLOSE_BRACE>,12:9]
43     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:
        ↳ [source.sma], definitions: {}, errors: [], indentations: [<indent 2, open
        ↳ 133, close 296>, <indent 1, open 60, close 304>], is_resolved: False;
49     target_language_name_statement      InputString str: , is_out_of_scope: [],
        ↳ chunks: [Abstract Machine Language], definitions: {}, errors: [],
        ↳ indentations: [<indent 2, open 133, close 296>, <indent 1, open 60, close
        ↳ 304>], is_resolved: False;
50 language_construct_rules
51     indentation_block
52         statements_list
53             match_statement
54                 InputString str: if\(), is_out_of_scope: [], chunks: [if\()],
                    definitions: {},
                    errors: [],
                    indentations: [<indent 2, open 133, close 296>,
                    ↳ <indent 1, open 60, close 304>], is_resolved: True;
55         statements_list
56             scope_name_statement      InputString str: , is_out_of_scope: [],
                ↳ chunks: [if.statement.definition], definitions: {}, errors: [],
                ↳ indentations: [<indent 2, open 133, close 296>, <indent 1, open 60,
                ↳ close 304>], is_resolved: False;
57         statements_list
58             push_statement
59                 indentation_block
60                     statements_list
61                         meta_scope_statement  InputString str: , is_out_of_scope: [],
                            ↳ chunks: [if.statement.body], definitions: {}, errors: [],
                            ↳ indentations: [<indent 2, open 133, close 296>, <indent 1,
                            ↳ open 60, close 304>], is_resolved: False;
62         statements_list
63             match_statement
64                 InputString str: \(), is_out_of_scope: [], chunks: [\()),
                    ↳ definitions: {}, errors: [],
                    indentations: [<indent 2, open
                    ↳ 133, close 296>, <indent 1, open 60, close 304>],
                    ↳ is_resolved: True;
65         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: False;  
70             statements_list  
71             pop_statement      InputString str: , is_out_of_scope: [],  
72             ↳  chunks: [true], definitions: {}, errors: [],  
73             ↳  indentations: [<indent 2, open 133, close 296>, <indent  
74             ↳  1, open 60, close 304>], is_resolved: False;  
75  
76 Generating 'main_formatter.html'...  
77 Generating 'main_formatter_syntax_tree.png'...  
78 Generating 'main_formatter_abstract_syntax_tree.png'...
```

## APÊNDICE C – EXECUÇÃO DE “MAIN\_HIGHLIGHTER.PY” ⇐ | ←

O programa do [Código 28](#), 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 32](#), encontra-se o arquivo HTML gerado pelo programa exemplo ([Código 28](#)). Nas [Figuras 26 e 27](#), encontram-se as imagens geradas pelo programa “`main_highlighter.py`” ([Código 28](#)). No [Código 33](#), encontra-se a saída da linha de comando que se obtém ao realizar a execução do programa de exemplo ([Código 28](#)).

Ambas as [Figuras 26 e 27](#) quanto o [Código 33](#) representam os mesmos dados, mas mostrados de formas diferentes (figura *versus* textual) e com níveis de detalhe diferentes. No [Código 33](#) são mostrado os nós-folhas das árvores com maior nível de detalhe. Já nas [Figuras 26 e 27](#), os nós-folhas são mostrados com simplificações para que a figura possa ser vista em uma única tela.

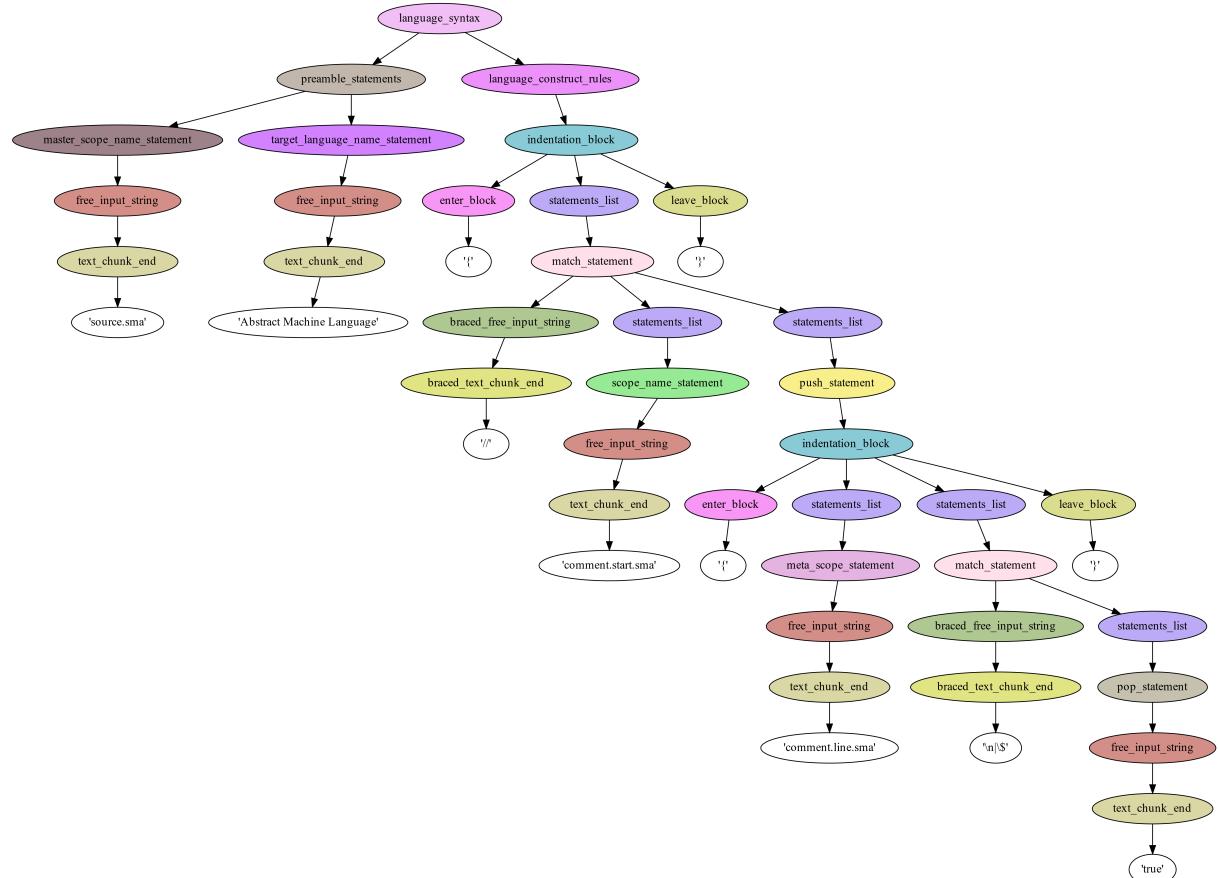
Código 32 – Arquivo HTML gerado pelo programa de exemplo “`main_highlighter.py`”

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

Código 33 – 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
```

Figura 26 – Árvore Sintática “main\_highlighter\_syntax\_tree.png”



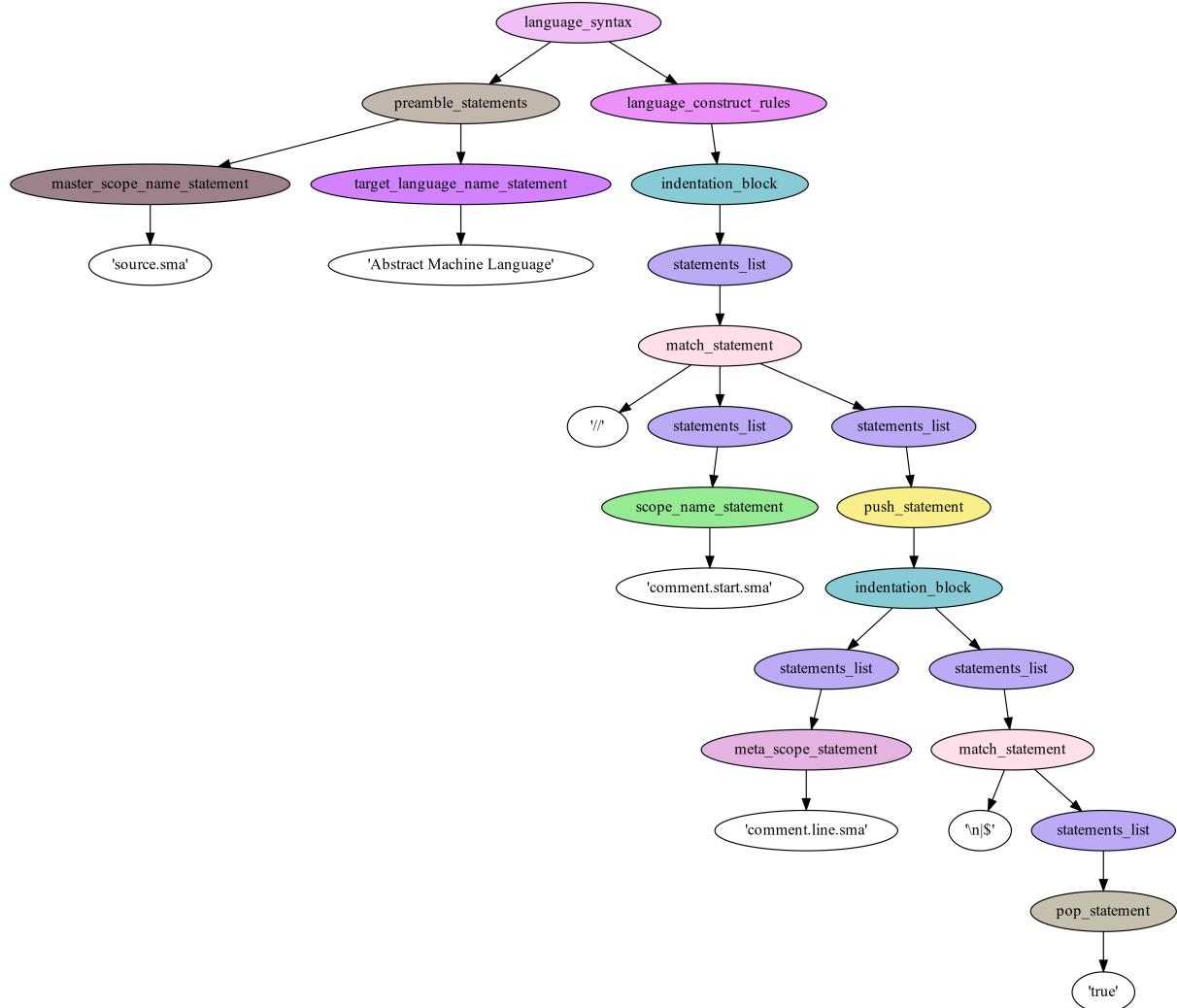
Fonte: Própria

```

17      braced_text_chunk_end      [@4,73:74='//<BRACED_TEXT_CHUNK_END>,4:12]
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
            braced_free_input_string

```

Figura 27 – Árvore Sintática Abstrata “main\_highlighter\_abstract\_syntax\_tree.png”



Fonte: Própria

```

33     braced_text_chunk_end
34         ↪  [ @8, 187:191='\\n|\\$' <BRACED_TEXT_CHUNK_END_>, 8:20 ]
35     statements_list
36     pop_statement
37     free_input_string
38     text_chunk_end
39         ↪  [ @9, 216:219='true' <TEXT_CHUNK_END_>, 9:22 ]
40     leave_block      [ @10, 243:243='}' <CLOSE_BRACE>, 11:9 ]
41     leave_block      [ @11, 251:251='}' <CLOSE_BRACE>, 13:1 ]
42
43 Abstract Syntax Tree
44 language_syntax
45 preamble_statements

```

```
44     master_scope_name_statement InputString str: , is_out_of_scope: [], chunks:
45         ↳ [source.sma], definitions: {}, errors: [], indentations: [<indent 2, open
46             ↳ 125, close 243>, <indent 1, open 60, close 251>], is_resolved: False;
47     target_language_name_statement      InputString str: , is_out_of_scope: [],
48         ↳ chunks: [Abstract Machine Language], definitions: {}, errors: [],
49             ↳ indentations: [<indent 2, open 125, close 243>, <indent 1, open 60, close
50                 ↳ 251>], is_resolved: False;
51     language_construct_rules
52         indentation_block
53             statements_list
54             match_statement
55                 InputString str: //, is_out_of_scope: [], chunks: [//], definitions: {},
56                     ↳ errors: [], indentations: [<indent 2, open 125, close 243>, <indent
57                         ↳ 1, open 60, close 251>], is_resolved: True;
58             statements_list
59                 scope_name_statement      InputString str: , is_out_of_scope: [],
60                     ↳ chunks: [comment.start.sma], definitions: {}, errors: [],
61                         ↳ indentations: [<indent 2, open 125, close 243>, <indent 1, open 60,
62                             ↳ close 251>], is_resolved: False;
63             statements_list
64                 push_statement
65                     indentation_block
66                         statements_list
67                             meta_scope_statement  InputString str: , is_out_of_scope: [],
68                                 ↳ chunks: [comment.line.sma], definitions: {}, errors: [],
69                                     ↳ indentations: [<indent 2, open 125, close 243>, <indent 1,
70                                         ↳ open 60, close 251>], is_resolved: False;
71             statements_list
72                 match_statement
73                     InputString str: \n|$, is_out_of_scope: [], chunks: [\n|\$],
74                         ↳ definitions: {}, errors: [], indentations: [<indent 2, open
75                             ↳ 125, close 243>, <indent 1, open 60, close 251>],
76                             ↳ is_resolved: True;
77             statements_list
78                 pop_statement      InputString str: , is_out_of_scope: [],
79                     ↳ chunks: [true], definitions: {}, errors: [],
80                         ↳ indentations: [<indent 2, open 125, close 243>, <indent
81                             ↳ 1, open 60, close 251>], is_resolved: False;
82
83
84 Generating 'main_highlighter.html'...
85 Generating 'main_highlighter_syntax_tree.png'...
86 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 35](#), 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 36](#)) na classe testes “TestSemanticRules”; ii) 5 testes de Adição de Cores ([Código 37](#)) na classe testes “TestCodeHighlighterBackEnd”; iii) 1 teste de Formatação de Código ([Código 38](#)) 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 36](#)). Para se executar os testes de unidade, basta executar o comando “python3 unit\_tests.py”. No [Código 34](#), pode-se encontrar um exemplo de execução dos testes de unidade apresentados no [Código 35](#).

Código 34 – 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 35 – 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
```

```
46     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         if return_tree:
78             new_tree = semantic_analyzer.TreeTransformer().transform( tree )
79
80             # log( 1, 'tree: \n%s', tree.pretty() )
81             # log( 1, 'tree: \n%s', new_tree.pretty() )
82             return new_tree
83
84
85     else:
86         with self.assertRaises( semantic_analyzer.SemanticErrors ) as error:
87             new_tree = semantic_analyzer.TreeTransformer().transform( tree )
88
89             return error
```



```
124         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         )
156         error = self._getError( example_grammar )
157
158         self.assertTextEqual(
159             r"""
160                 + 1. Missing include `missing_include` defined in your grammar on
161                 → [ @-1,140:154='missing_include'<TEXT_CHUNK_END_>,7:14]
162                 "", error.exception )
163
164     def test_invalidRegexInput(self):
165         example_grammar = wrap_text(
166             r"""
```

```
164     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         "", error.exception )
179
180     def test_duplicatedGlobalNames(self):
181         example_grammar = wrap_text(
182             r"""
183                 name: Abstract Machine Language
184                 name: Abstract Machine Language
185                 scope: source.sma
186                 scope: source.sma
187                 contexts: {
188                     match: (true|false) {
189                 }
190             }
191             error = self._getError( example_grammar )
192
193             self.assertTextEqual(
194                 r"""
195                 + 1. Duplicated target language name defined in your grammar on
196                 ↳ [-1,38:62='Abstract Machine Language'<TEXT CHUNK_END_>,2:7]
197                 + 2. Duplicated master scope name defined in your grammar on
198                 ↳ [-1,89:98='source.sma'<TEXT CHUNK_END_>,4:8]
199                 "", error.exception )
200
201     def test_missingScopeGlobalName(self):
202         example_grammar = wrap_text(
203             r"""
204                 name: Abstract Machine Language
```

```
203     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                 $constant: test
261                 contexts: {
262                     match: (true|false) {
263                         }
264                     }
265             """
266             error = self._getError( example_grammar )
267
268             self.assertTextEqual(
269                 r"""
270                     +  Warnings:
271                         + 1. Unused constant `$constant:` defined in your grammar on
272 →  [-1,50:59='$constant:'<CONSTANT_NAME_>,3:1]
273                     "", error.exception )
274
275     def test_constantUsage(self):
276         example_grammar = wrap_text(
277             r"""
278                 scope: source.sma
279                 name: Abstract Machine Language
280                 $constant:  test
281                 contexts: {
282                     match: (true$constant:$false)$constant: {
283                         }
284                     }
285             """
286             tree = self._getError( example_grammar, True )
```

```
285
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_: /(\\{|\\}|\\\\$|[^\n{}$])+/
311                 CONSTANT_USAGE_END_: /(?:\$[^\n\$:]+:)(?=:(?:\n|$))/
312                 TEXT_CHUNK_END_: /(\\{|\\}|\\\\$|[^\n{}$])+(?=:(?:\n|$))/
313             """
314             ,
315             start='free_input_string', parser='lalr', lexer='contextual' )
316             tree = my_parser.parse( r"true$constant:$|false" )
317
318             self.assertTextEqual(
319                 r"""
320                     + free_input_string
321                     + text_chunk  [@1,0:3='true'<TEXT_CHUNK_>,1:1]
322                     + constant_usage  [@2,4:13='$constant:<CONSTANT_USAGE_>,1:5]
323                     + text_chunk_end  [@3,14:21='\\$|false'<TEXT_CHUNK_END_>,1:15]
324                 """", tree.pretty(debug=True) )
325
326     def test_isolatedBracedEnd(self):
327         my_parser = pushdown.Lark(
```

```

326     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_redifinedConst(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

```

```
366     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
```

```
406         +     [ @-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                 """", error.exception )
436
437 class TestCodeHighlighterBackEnd(TestingGrammarUtilities):
438
439     def _getBackend(self, example_grammar, example_program, example_theme):
440         function_file = get_relative_path( "examples/%s.html" % getCallerName(),
441             __file__ )
442         # log( 1, "function_file: %s", function_file )
443
444         tree = self._getError( example_grammar, True )
445         backend = code_highlighter.Backend(tree, example_program, example_theme)
446         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"""
490                 
```

```
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|else|
654     ← 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_CRAZYNES 10000<br /></font><span
  ↳ grammar_scope="none" theme_scope="none">     </span><font """

```

717

```
# Break this unit test result into two parts otherwise Latex cannot
718   ↳ 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> 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,
731         tree, example_program, example_settings )
732     generated_html = backend.generated_html()
733
734     with open( function_file, 'w', newline='\n', encoding='utf-8' ) as
735         output_file:
736             output_file.write( generated_html )
737             output_file.write("\n")
738
739     return generated_html
740
741 def test_singleIfStatement(self):
742     example_grammar = wrap_text(
743         r"""
744             scope: source.sma
745             name: Abstract Machine Language
746             contexts: {
747                 match: if\(\ {
748                     scope: if.statement.definition
749                     push: {
750                         meta_scope: if.statement.body
751                         match: \) {
752                             scope: if.statement.definition
753                             pop: true
754                         }
755                     }
756                 }
757             """
758     example_program = wrap_text(
759         r"""if(something) bar""")
```

```
767     self.assertTextEqual(
768         r"""
769             + <!DOCTYPE html><html><head><title>Abstract Machine Language -
770             source.sma</title></head>
771                 + <body style="white-space: pre; font-family: monospace;"><span
772                 setting="unformatted" grammar_scope="if.statement.definition" setting_scope=""
773                 original_program="if(>if(</span><span setting="2"
774                 grammar_scope="if.statement.body" setting_scope="if.statement.body"
775                 original_program="something"> something </span><span setting="unformatted"
776                 grammar_scope="if.statement.definition" setting_scope=""
777                 original_program=")">)</span><span grammar_scope="none" setting_scope="none">
778                 bar</span></body></html>
779             """, generated_html )
780
781
782
783
784 if __name__ == "__main__":
785     main()
```

## APÊNDICE E – CÓDIGO DO ANALISADOR SEMÂNTICO ⇐ | ←

De todos os códigos-fonte criados neste trabalho, o Analisador Semântico ([Código 36](#)) 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 36](#)). A classe “`TestSemanticRules`” ([Código 35](#)), 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 35](#)).

**`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 36 – 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
→   always beforehand know.
119 """
120 unescape_control_characters = re.compile( r"\\\([${}]\)" )
121
122 def __init__(self, chunks, definitions, errors, indentations):
123     super(InputString, self).__init__()
124     self.is_out_of_scope = []
125
126     self.chunks = chunks
127     self.definitions = definitions
128     self.errors = errors
129     self.indentations = indentations
130     self.is_resolved = False
131
132 def __str__(self):
133 """
134     @param `definitions` a dictionary with all completely known
135     constants. For example { "$varrrr": " varrrr " }
136 """
137     if self.is_resolved: return self.str
138
139     scope_usage_error = ""
140     is_resolved = True
141     resolutions = []
142
143     # log( 1, 'self.chunks %s', self.chunks )
144     # log( 1, 'self.definitions %s', self.definitions )
145
146     for usage in self.chunks:
147         # log( 1, 'usage %s', usage )
148         # log( 1, 'usage %s', type(usage) )
149         # log( 1, 'usage.name %s', usage.name )
150         # log( 1, 'usage.name %s', type(usage.name) )
151         usage_name_in_self_definitions = usage.name in self.definitions
152
153         if usage.str or usage_name_in_self_definitions:
154             definition = usage
155
156             if usage_name_in_self_definitions:
157                 definition = self.definitions[usage.name]
158
```

```
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                     # possible global token
192                     if definition_block.indentation < 0:
193                         continue
194
195                     else:
196                         break
197
198             # log('resolution for definition_block', definition_block)
199             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     def __init__(self):
230         ## Saves all the semantic errors detected so far
231         self.errors = []
232
233         ## Saves all warnings noted so far
234         self.warnings = []
235
236         ## Whether the mandatory/obligatory global scope name statement was declared
237         self.is_master_scope_name_set = False
238
239         ## Whether the mandatory/obligatory global language name statement was
240             ↪   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         ↪ %" % ( include_name, first_token.pretty() ) )
296
297     if include_name == 'contexts':
298         self.errors.append( "Extra `contexts` rule defined in your grammar on
299         ↪ %" % 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%', 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 %" % ( 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 37](#)), que faz uso da metagramática no [Código 39](#). No [Código 28](#), pode-se encontrar um exemplo explícito de utilização do [Código 37](#).

Código 37 – 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)
```

```
74     assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
    ↵   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
```

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

```
178
179     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:
```

```
217
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
247         log( 4, "miscellaneous_language_rules: %s", miscellaneous_language_rules )
248
249         ## match_statement -> scope_name_statement -> push_statement ->
250         ##                                         match_statement ->
251         ##                                         pop_statement
252         ##                                         match_statement ->
253         ##                                         scope_name_statement -> pop_statement
254         ##                                         match_statement ->
255         ##                                         meta_scope_statement -> match_statement -> pop_statement
256
257     def match_statement(self, tree):
258         log( 4, "is_there_push_after_match: %s", self.is_there_push_after_match )
259
260         ## Discards the last_match_stack because we do not need the stack when
```

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

```
295     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 38](#)) que faz uso da metagramática no [Código 39](#). No [Código 27](#), pode-se encontrar um exemplo explícito de utilização do [Código 38](#).

Código 38 – 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 def get_html_header(title):
29     return debug_tools.utilities.wrap_text(
30         r"""
31             <!DOCTYPE html><html><head><title>%s</title></head>
32             <body style="white-space: pre; font-family: monospace;">
33             """ % ( title )
34     )
```

```
31
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         """
```

```
72     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
101
102     # At the end of the process, we must to preserve the program size on
103     # self.program
104
105     # for the correct merge with the text chunks processed
106     assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
107         self.initial_size, len(self.program) )
108     fixed_program = sorted( self.new_program, key=lambda item: item[0] )
109     fixed_program_len = len(fixed_program)
110
111     log( 4, "fixed_program:\n%s", pprint.pformat( [ ( item[0], item[1],
112         str(item[2]) ) for item in fixed_program ], indent=2, width=200 ) )
113     log( 4, "cached_new_program: %s", self.cached_new_program )
114
115     self.cached_new_program = [ item[2] for item in fixed_program ]
116
117     return self.cached_new_program
```

```
107
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`'", setting_value, scope_name, setting_scope )
119
120                     if setting_scope == scope_name:
121                         log( 8, "Selecting %s with %s", setting_scope, setting_value )
122                         return setting_value, setting_scope
123
124
125             matched_setting, setting_scope = select()
126
127             log( 8, "matched_setting: '%s' (on %s)", matched_setting, setting_scope )
128             return matched_setting, setting_scope
129
130     def add_match(self, scope_name, last_match_stack, match):
131         log( 4, "add_match: %s", match )
132         last_match_stack[-1].append(match)
133         match_start = match.start(0)
134         match_end = match.end(0)
135         match_start, match_end = match_start, match_end
136
137         self.program = self.program[:match_start] + "$" * ( match_end - match_start
138                                     ↪ ) + self.program[match_end:]
139         assert len( self.program ) == self.initial_size, "Expected %s got %s" % (
140                                     ↪ self.initial_size, len(self.program) )
141
142         self._generate_chunk_html( scope_name, match[0], match_start, match_end )
143
144     def add_meta_scope(self, scope_name, last_matches, match,
145                           ↪ ignore_last_match=False):
146         """ 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             match = tree.children[0]
279             self.match = re.compile( str(match) )
280
281             log( 4, "match: %s", self.match.pattern )
282             # log( 4, "tree: %s", tree )
283             self.visit_children( tree )
284
285     def push_statement(self, tree):
286         self.is_there_push_after_match = True
287         log( 4, "push_stack: %s", self.last_match_stack )
288
289         if not self.is_there_scope_after_match:
290             self.last_match_stack.append( [ match for match in self.match.finditer(
291                 → str(self.program) ) ] )
292             # log( 4, "tree: %s", tree )
293             self.visit_children( tree )
294
295     def meta_scope_statement(self, tree):
296         meta_scope = tree.children[0]
297         self.meta_scope = meta_scope
298         log( 4, "pattern: %s", self.match.pattern )
299         log( 4, "meta_scope: %s", meta_scope )
300
301     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 39](#)) utilizada pela adição de cores e formatação de código-fonte. Nos [Códigos 27](#) e [28](#), 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 39 – 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
25 OPEN_BRACE: "{"
26 CLOSE_BRACE: "}"
27
28 push_statement: "push" ":" " indentation_block
29 include_statement: "include" ":" " free_input_string
30 constant_definition: constant_name free_input_string
```

```
26 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
53             → statements_list )* _NEWLINE "}"
54 scope_name_statement: "scope" ":" free_input_string
55
56 capturing_block: "captures" ":" "{" ( _NEWLINE capturing_lines )+ _NEWLINE "}"
57 capturing_lines: INTEGER+ ":" free_input_string
58
59 pop_statement: "pop" ":" free_input_string
60 meta_scope_statement: "meta_scope" ":" free_input_string
61
62 // General terminal tokens
63 CR: "/r"
64 LF: "/n"
65 SPACES: /[\\t \\f]+/
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 Sperfeld Coan<sup>1</sup>, Rafael de Santiago<sup>1</sup>**

<sup>1</sup>Centro 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

**Resumo.** 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. 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.

**Abstract.** It is 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](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](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](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.