# Uma Ferramenta de Formatação Programável Por Gramáticas

**Evandro Sperfeld Coan[1], Rafael de Santiago[1]**

[1]Centro Tecnológico, Departamento de Informática e Estatistica – Universidade Federal de Santa Carina (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 linguagem de programação por meio de uma nova definição gramáticas, 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 destacamento aos elementos mais importes do código-fonte; 2) fazer menor destacamento 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.

```java
public class Main
{
    public static void main(String[] args)
    {
        if(args.length > 0)
        {
            System.out.println("Bye World!");
        }
        else
        {
            System.out.println("Hello World!");
        }
    }
}
```

**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 estrura “if” com a seguinte forma “if(alguma coisa)” ou somente “if()”. Uma vez que esses “if”'s sejam

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

```
language_syntax: _NEWLINE? ( match_statement | scope_name_statement )+ _NEWLINE?

match_statement: "match" ": " /.+/ _NEWLINE
scope_name_statement: "scope" ": " /.+/ _NEWLINE

CR: "/r"
LF: "/n"

_NEWLINE: ( /\r?\n[\t ]*/ )+
SPACES: /[\t \f]+/

%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.

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

**Figura 3. Exemplo de gramática pelas regras da mínima metagramática**

Seguindo 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, substituíndo 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, atribuíndo 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.

```
class SingleSpaceFormatter(AbstractFormatter):

    def format_text(self, code_to_format, setting_value):
        code_to_format = code_to_format.strip( " " )

        if setting_value:
            return " " * setting_value + code_to_format + " " * setting_value
        else:
            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 utilizado uma pilha de múltiplos contextos, com já feito em editores de texto que serviram de inspiração a este trabalho.

```
public class Main
{
    public static void main(String[] args)
    {
        if(  args.length > 0  )
        {
            System.out.println("Bye World!");
        }
        else
        {
            System.out.println("Hello World!");
        }
    }
}
```

**Figura 7. Resultado do Formatador de Código para Java**

## 4. Conclusão

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

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

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

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

## References


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

AI HUA WU; PAQUET, J. The translator generation in the general intensional programming compilier. In: 8TH International Conference on Computer Supported Cooperative Work in Design. [S.l.: s.n.], maio 2004. 668–672 vol.2. DOI: 10.1109/CACWD.2004.1349274.

AHO, Alfred V.; LAM, Monica S. et al. Compilers: Principles, Techniques, and Tools (2Nd Edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

HOPCROFT, John E.; MOTWANI, Rajeev; ULLMAN, Jeffrey D. Introduction to Automata Theory, Languages, and Computation. In: New York, NY, USA: Pearson Education, 2006. ISBN 978-0321455369. DOI: 10.1145/568438.568455 . Disponível em: <http://doi.acm.org/10.1145/568438.568455>.

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

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

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

SKINNER, Jon. SUBLIME TEXT 3 DOCUMENTATION, Syntax Definitions. 2015a. Disponível em: <https://www.sublimetext.com/docs/3/syntax.html>. Acesso em: 1 mar. 2017.

SKINNER, Jon. SUBLIME TEXT 3 DOCUMENTATION, Scope Naming. 2015b. Disponível em: <http://www.sublimetext.com/docs/3/scope_naming.html>. Acesso em: 17 set. 2019.

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

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

SINGER, Adam B. The Command Line Interface. In: PRACTICAL C++ Design. 1. ed. Online: Apress, 2017. p. 97–113. ISBN 978-1-4842-3056-5. DOI: 10.1007/978-1-4842-3057-2_5.