INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Bahia

Campus
Irecê

Átila Gama Silva

Witch Cooking: Formatação de Código via Linguagem de Consulta do *Tree-Sitter*

Irecê

2023

Átila Gama Silva

Witch Cooking: Formatação de Código via Linguagem de Consulta do *Tree-Sitter*

Trabalho de Conclusão de Curso apresentado ao Curso Técnico em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia da Bahia – Campus Irecê, como requisito parcial para obtenção do diploma de Técnico em Análise e Desenvolvimento de Sistemas, mediante a orientação do professor Rafael Xavier.

Irecê

2023

Resumo

A formatação de código é fundamental no desenvolvimento de software, permitindo estabelecer aspectos desejados como a padronização e legibilidade do código, que impactam positivamente o ciclo de vida do software. Em ambientes de desenvolvimento modernos, é comum o uso de ferramentas para automatizar a formatação de código. No entanto, essas ferramentas apresentam limitações no número de linguagens suportadas, nas opções de configuração e não permitem a definição de regras de formatação personalizadas. Visando superar ou reduzir as limitações frequentemente encontradas nas ferramentas convencionais de formatação, neste trabalho é apresentado o *Witch Cooking*, um software protótipo que tem como objetivos: (I) abranger uma gama de linguagens de programação; além de (II) permitir que o usuário defina suas próprias regras de formatação.

Palavras-chave: formatação de código; *Tree-Sitter*; *prettyprint*; customização; *query*.

Sumário

1 Introdução

A flexibilidade presente na sintaxe de linguagens de programação permite que diferentes arranjos dum mesmo código compartilhem um valor sintático equivalente. Essa característica possibilita a formatação do código de acordo com aspectos desejados, como a legibilidade, que é fundamental no ciclo de vida do software (BUSE; WEIMER, 2009, p. 1; OLIVEIRA et al., 2020, p. 1). Consequentemente, a formatação é frequentemente utilizada para estabelecer um nível satisfatório de legibilidade em bases de código.

A formatação de código pode ser realizada manualmente pelo programador, embora esse processo possa ser demorado, especialmente em grandes bases de código, exigindo do programador um tempo que poderia ser empregado em outra tarefa. Além disso, a formatação manual também pode ser falha e inconsistente devido à suscetibilidade humana ao erro, podendo ser agravada quando há múltiplos programadores em uma base de código.

Para evitar os problemas inerentes da formatação manual, é comum a utilização de softwares que automatizam a formatação de forma determinística, tornando o código padronizado e consistente. No entanto, é comum que essas ferramentas de formatação sejam limitadas a uma linguagem ou a uma família de linguagens de programação. Além disso, algumas delas são *opinionated*, assim restringindo as possiblidades de customização do usuário. Finalmente, é também comum que essas ferramentas de formatação não permitam que o usuário defina regras de formatação personalizadas.

Em ambientes de desenvolvimento modernos, além da utilização de ferramentas que automatizam a formatação de código, é também comum a utilização do *Tree-Sitter* (TREE-SITTER..., 2023): um sistema de análise sintática de código aberto que foi disponibilizado ao público geral no GitHub primeiramente em 2019, tendo sido inicialmente desenvolvido por Max Brunsfeld. Desde seu lançamento, o *Tree-Sitter* tem ganhado popularidade na comunidade de desenvolvedores devido dentre outros motivos à: (I) sua capacidade de suportar várias linguagens de programação; além de (II) sua linguagem de consulta, a qual permite a realização de buscas complexas na árvore analisada de um código fonte.

Visando superar ou reduzir as limitações previamente mencionadas, quais são frequentemente presentes nas ferramentas convencionais de formatação, este trabalho tem como objetivo geral desenvolver um software protótipo para a formatação de código. Nomeada antecipadamente de *Witch Cooking* (SILVA, 2023) pelo autor, a ferramenta a ser desenvolvida tem como objetivos: (I) abranger uma gama de linguagens de programação; além de (II) permitir que o usuário defina suas próprias regras de formatação.

Para alcançar o objetivo geral proposto, este trabalho tem como objetivos específicos: (I) apresentar e contextualizar ferramentas de formatação de código conceituadas, além de abordar seus métodos de formatação; (II) conceituar, contextualizar e abordar as tecnologias do sistema *Tree-Sitter*; e, por fim, (III) utilizar o *Tree-Sitter* para desenvolver e atender às aspirações do *Witch Cooking*.

2 A Formatação de Código-Fonte

Desde os primórdios da computação, métodos foram desenvolvidos para garantir que a saída impressa fosse formatada de maneira esteticamente agradável (HARRIS, 1956 apud YELLAND, 2015b, p. 1). Esses métodos ganharam popularidade sob o termo *prettyprinting*, que se refere à formatação visual de diversos tipos de conteúdo. No desenvolvimento de software, o *prettyprinting* é conhecido como formatação de código: uma prática histórica e comum que envolve a adoção de convenções estilísticas para estruturar o código-fonte. Existem diversas terminologias utilizadas para se referir às ferramentas que realizam a formatação do código-fonte, incluindo: (I) formatadores de código, ou *code formatters* em inglês; (II) *prettyprinters*; ou ainda (III) *beautifiers*.

Durante as décadas de 60 e 70, a linguagem de programação LISP[1] proporcionou condições favoráveis para o avanço da formatação de código (YELLAND, 2015a, p. 2). (I) LISP apresentava uma sintaxe distinta e expressiva, baseada em listas e estruturas aninhadas delimitadas por parênteses[2], o que tornava o código-fonte praticamente ilegível e incompreensível caso não fosse devidamente organizado. Além disso, (II) LISP era homoicônico, ou seja, permitia a manipulação do código via dados da própria linguagem. Essas duas características intrínsecas do LISP fundamentaram o surgimento de: (I) *prettyprinters* para a linguagem, visando aperfeiçoar a legibilidade do código-fonte escrito ou emitido; além de (II) novas técnicas e abordagens para a formatação de código em geral. Esses avanços contribuíram significativamente para o conhecimento e o aprimoramento de práticas de formatação de código em trabalhos subsequentes.

Em 1967, Bill Gosper desenvolveu o *GRINDEF* (acrônimo para ***GRIND***[3] ***F**unction*), considerado o primeiro *prettyprinter* a mensurar o tamanho das linhas e ter ciência de sua localização no arquivo (GOSPER, 2023; GRIESEMER, 2022). Essa ferramenta implementava o algoritmo *recursive re-predictor*, que, como descrito por Goldstein (1973), percorria as árvores de nós representando as listas e imprimia os nós considerando seus tamanhos e a quantidade máxima de caracteres que ainda poderiam ser inseridos em uma linha.

Posteriormente, Hearn e Norman (1979) propuseram um algoritmo mais elaborado para *prettyprinting* que utilizava um par de corrotinas[4]: uma (I) responsável por produzir uma sequência de caracteres que representavam o programa sendo impresso; e outra (II) responsável

[1] O termo LISP (acrônimo para ***LIS**t **P**rocessing*) inicialmente se referia à linguagem de programação desenvolvida por McCarthy (1960), porém, com a disseminação de dialetos da linguagem, o termo também passou a ser utilizado para se referir à família de linguagens derivadas da original.

[2] Na notação original de McCarthy, eram utilizadas *M-expressions* entre colchetes para representar expressões. Essas *M-expressions* seriam posteriormente traduzidas em *S-expressions*. Por exemplo, a *M-expression* "`car[cons[A; B]]`" seria equivalente à *S-expression* "`(car (cons A B))`". No entanto, assim que a linguagem LISP foi implementada, os programadores prontamente adotaram o uso das *S-expressions* em vez das *M-expressions*, tornando as *S-expressions* a forma predominante de representar a estrutura do código na linguagem.

[3] O termo *grind* era utilizado em alguns círculos de LISP como sinônimo para *prettyprinters*.

[4] Na obra, os autores enfatizam que a implementação com o uso de corrotinas é tão simples que poderia ser simulada – sem grandes dificuldades – em linguagens que não possuíam suporte nativo para corrotinas (HEARN; NORMAN, 1979, p. 53).

por decidir como esses caracteres seriam exibidos. Essas corrotinas se comunicavam por meio de um buffer *FIFO*, permitindo que as decisões de formatação fossem adiadas até que houvesse informações suficientes disponíveis para tomá-las com confiabilidade.

No ano seguinte, Oppen (1980) apresentou em sua influente obra, intitulada *Prettyprinting*, um algoritmo inovador de formatação de código-fonte. Embora apresentasse semelhanças ao algoritmo proposto por Hearn e Norman (1979), destacava-se por sua capacidade de formatar código derivado de qualquer linguagem de programação. Para realizar essa tarefa, o algoritmo necessitava que o código-fonte fosse anotado com espaços em branco[5] e delimitadores especiais para marcar o início e fim de blocos logicamente contíguos. Assim, o código a ser fornecido ao algoritmo precisaria ser processado por uma ferramenta intermediária capaz de compreender a sintaxe da linguagem e fornecer um código anotado de forma adequada, permitindo que o algoritmo realizasse a formatação apropriada.

Desde então, uma ampla variedade de *prettyprinters* e trabalhos relacionados ao tema têm sido lançados e publicados. Mais recentemente, por exemplo, em *A New Approach to Optimal Code Formatting*, Yelland (2015a) descreveu os fundamentos do algoritmo (de formatação de código) utilizado no *rfmt* (YELLAND, 2015b), além das abstrações utilizadas para facilitar a implementação do algoritmo em uma variedade de linguagens e políticas de layout.

[5] Quebras de linha, avanços de formulário e alimentações de linha também eram tratados como espaços em branco pelo algoritmo.

3 O Sistema *Tree-Sitter*

O *Tree-Sitter* é um sistema multilingual de análise sintática para ferramentas de programação inicialmente desenvolvido como um projeto secundário por Max Brunsfeld. Como relatado pelo próprio autor (TREE-SITTER. . . , 2017), o *Tree-Sitter* surgiu como uma tentativa de solucionar problemas presentes nas ferramentas de análise sintática da época. Mais especificamente, o sistema tinha como objetivos: (I) ser utilizado no ambiente de desenvolvimento para produzir árvores de sintaxe a partir da análise de códigos escritos em várias linguagens; (II) implementar a análise incremental, permitindo a atualização da árvore de sintaxe em tempo real; (III) expor através da árvore de sintaxe os nós representando suas construções gramaticais no código (*e.g.*, classes, funções, declarações, etc.), diferentemente das ferramentas contemporâneas, que utilizavam uma abordagem simplística baseada em expressões regulares; e, por fim, (IV) ser livre de dependências, assim beneficiando sua adoção e aplicabilidade.

Além das funcionalidades previamente mencionadas presentes no *Tree-Sitter*, o sistema também conta uma ferramenta de linha de comando que pode ser utilizada para gerar *parsers* para uma linguagem a partir de sua gramática. A gramática é definida via a linguagem de programação JavaScript, a qual: (I) foi eleita 15 vezes seguidas pela *Stack Overflow Developer Survey* (STACK. . . , 2022) como a linguagem de programação mais comumente usada; além de (II) ser amplamente considerada pela comunidade de programadores como uma das linguagens mais fáceis de aprender e programar (11. . . , 2023; GOEL, 2023; JAVASCRIPT. . . , 2023). A ferramenta de geração de *parsers* também disponibiliza funções preestabelecidas para permitir a criação de gramáticas com diferentes níveis de complexidade. Não é surpreendente que, devido a essas características e facilidades presentes na criação de *parsers*, exista uma variedade de linguagens de programação e formatos de arquivos – variando de linguagens com sintaxes complexas, como C++ e Perl, a formatos de arquivos mais específicos, como *.vhs* e *.rasi* –, os quais têm parsers gerados pelo *Tree-Sitter* e são suportados pelo sistema.

Na análise de código, é comum realizar tarefas que envolvem a busca de padrões em árvores sintáticas. Para isso, o *Tree-Sitter* oferece uma pequena linguagem de consulta declarativa que é capaz de expressar esses padrões por meio de *S-expression* e buscar correspondências. A linguagem de consulta suporta operadores que permitem: (I) a captura de nós; (II) a quantificação de nós, análoga às expressões regulares; (III) o agrupamento de nós; (IV) as alternâncias de nós; (V) o uso de *wildcards*; e (VI) a ordenação de nós. Adicionalmente, é possível definir propriedades nos nós da árvore sintática usando a linguagem de consulta do *Tree-Sitter*. Também é permitido o uso de predicados – funções arbitrárias para filtrar nós ou realizar verificações mais complexas durante a busca de padrões – sejam eles *builtin* ou estendidos por meio de uma API.

Um exemplo da relevância e utilidade da linguagem de consulta é o plugin *nvim-treesitter* (NVIM-TREESITTER. . . , 2023), que é frequentemente utilizado no editor *Neovim* (NEOVIM. . . , 2023). Esse plugin utiliza a linguagem de consulta para definir diferentes recursos, tais como: (I) *code folding*, que permite ocultar blocos de código; (II) highlights, que realçam a sintaxe

do código; (III) indentações, que definem a estrutura do código; (IV) injeções, que permitem adicionar novas sintaxes a arquivos existentes; além de (V) captura de nós correspondentes a construções gramaticais (*e.g.*, funções, classes, métodos, etc.), os quais são frequentemente utilizados em rotinas de programação tais como a remoção e navegação do código.

Em suma, o sistema *Tree-Sitter* se mostrou uma solução inovadora e eficiente para a análise sintática de códigos em diversas linguagens de programação, com uma abordagem diferenciada e sofisticada que possibilita a atualização em tempo real das árvores sintáticas e a identificação precisa das construções gramaticais presentes no código. Além disso, a ferramenta de linha de comando disponível no sistema facilita a geração de parsers a partir de gramáticas definidas em JavaScript, o que torna o processo mais acessível e personalizável para os programadores. Finalmente, a linguagem de consulta declarativa oferecida pelo *Tree-Sitter* se mostrou uma importante *feature*, sendo utilizada em diversos plugins de editores de código para realizar tarefas variadas e sofisticadas, contribuindo significativamente no ambiente de desenvolvimento.

4 O *Witch Cooking*

Desenvolver um *prettyprinter* não é uma tarefa simples (HUGHES, 1995, p. 2). Durante o processo de formatação, é necessário (I) ter conhecimento das estruturas gramaticais da sintaxe específica em questão. Além disso, o objetivo do *prettyprinter* é (II) formatar o código de maneira otimizada e agradável para o leitor (HUGHES, 1995, p. 2), levando em consideração as melhores práticas de formatação. Em alguns casos, também é importante (III) permitir que o usuário possa optar por estilos de formatação personalizados, atendendo às suas preferências individuais. Adicionalmente, as sintaxes das linguagens de programação costumam ser flexíveis em relação à estruturação do código-fonte. Assim, para atender o item II, geralmente é necessário (IV) utilizar algoritmos que calculem o layout mais adequado de acordo com o contexto. Fatalmente, essas características essenciais de um *prettyprinter* normalmente o restringem a uma linguagem específica ou, em alguns casos, a uma família de linguagens que compartilham semelhanças sintáticas ou estruturais.

5 Conclusão

Referências

11 Most In-Demand Programming Languages. Berkeley Extension. Disponível em: <https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages>. Acesso em: 5 mai. 2023.

BUSE, Raymond P. L.; WEIMER, Westley R. Learning a Metric for Code Readability. **IEEE Transactions on Software Engineering**, IEEE, v. 36, n. 4, p. 546–558, 2009. DOI: 10.1109/TSE.2009.70.

TREE-SITTER: a new parsing system for programming tools - GitHub Universe 2017. 1 vídeo (43 min). GitHub. 2017. Disponível em: <https://youtu.be/a1rC79DHpmY>. Acesso em: 30 abr. 2023.

GOEL, Aman. **How to Learn JavaScript in 2023 | 8 Best Ways For Beginners**. Disponível em: <https://hackr.io/blog/how-to-learn-javascript>. Acesso em: 4 mai. 2023.

GOLDSTEIN, Ira. **Pretty printing**: Converting list to linear structure. [S.l.], fev. 1973.

GOSPER, Ralph William. **Twubblesome Twelve**. Disponível em: <http://gosper.org/bill.html>. Acesso em: 21 mai. 2023.

GRIESEMER, Robert. **The Cultural Evolution of gofmt**. Google Research. 2022. Disponível em: <https://go.dev/talks/2015/gofmt-en.slide>. Acesso em: 20 mai. 2023.

HARRIS, R. W. Keyboard Standardization. **Western Union Technical Review**, v. 10, n. 1, p. 37–42, 1956.

HEARN, Anthony C.; NORMAN, Arthur C. A one-pass prettyprinter. **ACM SIGPLAN Notices**, v. 14, n. 12, p. 50–58, 1979. DOI: 10.1145/954004.954005.

HUGHES, John. The Design of a Pretty-printing Library. In: **Advanced Functional Programming**: First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 1995, Tutorial Text. Edição: Johan Jeuring e Erik Meijer. Båstad, Sweden: Springer, 15 mai. 1995. v. 925, p. 53–96. (Lecture Notes in Computer Science). DOI: 10.1007/3-540-59451-5_3.

MCCARTHY, John. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. **Communications of the ACM**, v. 3, n. 4, p. 184–195, 1 abr. 1960. DOI: 10.1145/367177.367199.

NEOVIM: hyperextensible Vim-based text editor. Versão 0.9.0. Neovim. Disponível em: <https://neovim.io/>. Acesso em: 7 mai. 2023.

NVIM-TREESITTER: Nvim Treesitter configurations and abstraction layer. Versão 0.9.0. nvim-treesitter. Disponível em: <https://github.com/nvim-treesitter/nvim-treesitter>. Acesso em: 7 mai. 2023.

OLIVEIRA, Delano et al. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In: IEEE. 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). [S.l.: s.n.], 2020. DOI: 10.1109/icsme46990.2020.00041.

OPPEN, Derek C. Prettyprinting. **ACM Transactions on Programming Languages and Systems**, v. 2, n. 4, p. 465–483, out. 1980. DOI: 10.1145/357114.357115.

SILVA, Átila Gama. **Witch Cooking**: Experimental multilingual code formatter based on Tree-Sitter's query. Disponível em: <https://github.com/atchim/witch-cooking>. Acesso em: 27 abr. 2023.

STACK Overflow Developer Survey. Stack Overflow. 2022. Disponível em: <https://survey.stackoverflow.co/2022>. Acesso em: 1 mai. 2023.

TREE-SITTER: a parsing system for programming tools. Versão 0.20.8. Tree-Sitter. Disponível em: <https://tree-sitter.github.io/>. Acesso em: 6 abr. 2023.

JAVASCRIPT Tutorial. W3Schools. Disponível em: <https://www.w3schools.com/js>. Acesso em: 5 mai. 2023.

YELLAND, Phillip M. A New Approach to Optimal Code Formatting. **Google Research**, 2015. Disponível em: <https://research.google.com/pubs/archive/44667.pdf>. Acesso em: 7 abr. 2023.

_____. **rfmt**: R source code formatter. Versão 1.0. 5 ago. 2015. Disponível em: <https://github.com/google/rfmt>. Acesso em: 23 mai. 2023.

Glossário

B

Builtin No desenvolvimento de software, *builtin* é um termo utilizado para descrever funções ou comandos que são incorporados no próprio sistema ou ambiente de programação, em contraste com funções ou comandos definidos pelo usuário ou programador. Por exemplo, em Python, a função *print* é um exemplo de função *builtin*. Essas funções ou comandos *builtin* estão prontamente disponíveis, o que significa que não é necessário definir ou importá-los explicitamente por parte do usuário ou programador.

C

Code folding No desenvolvimento de software, *code folding* é uma *feature* que permite a ocultação seletiva de partes de um documento ou código-fonte, permitindo ao usuário ou programdor visualizar apenas as seções de interesse. Esse recurso é especialmente útil ao lidar com grandes volumes de informações, possibilitando uma melhor organização e foco em seções específicas. O *code folding* é comumente utilizado em ambientes de programação que possuem estruturas hierárquicas, como árvores ou blocos aninhados, facilitando a navegação e a compreensão do código.

F

Feature Na programação de computadores, uma *feature* é uma funcionalidade específica ou capacidade de um software que agrega valor ao sistema. Uma *feature* pode ser um componente, uma função, um conjunto de comandos ou qualquer outra característica que ofereça uma determinada funcionalidade aos usuários do software. Ela pode abranger desde recursos básicos até funcionalidades mais avançadas e personalizadas, e é projetada para atender às necessidades e demandas dos usuários, melhorar a usabilidade do software e fornecer uma experiência mais completa.

FIFO Na computação e na teoria de sistemas, *FIFO* é um acrônimo para *first in, first out* (o primeiro a entrar é o primeiro a sair), um método para organizar a manipulação de uma estrutura de dados (geralmente, especificamente um buffer de dados) onde a entrada mais antiga (primeira), ou "cabeça" da fila, é processada primeiro.

H

Homoicônico Na programação de computadores, a homoiconicidade (das palavras gregas *homo-*, que significa "o mesmo", e *icon*, que significa "representação") é uma propriedade de algumas linguagens de programação. Uma linguagem é homoicônica se

um programa escrito nela pode ser manipulado como dados utilizando a própria linguagem, permitindo inferir a representação interna do programa apenas lendo o próprio programa. Essa propriedade é frequentemente resumida dizendo que a linguagem trata o **código como dados**.

O

Opinionated No desenvolvimento de software, *opinionated* é um termo utilizado para se referir a um conjunto de práticas preestabelecidas sobre como abordar uma determinada tarefa, podendo não permitir customizações ou desvio das abordagens.

P

Parser Na programação de computadores, um *parser* (ou analisador) é um componente de software que recebe dados de entrada (frequentemente texto) e constrói uma estrutura de dados – frequentemente algum tipo de árvore de análise, árvore de sintaxe abstrata ou outra estrutura hierárquica – fornecendo uma representação estrutural da entrada enquanto verifica a sintaxe correta. A análise pode ser precedida ou seguida por outras etapas, ou essas etapas podem ser combinadas em uma única etapa. O *parser* é frequentemente precedido por um analisador léxico (*lexer*) separado, que cria tokens a partir da sequência de caracteres de entrada; alternativamente, esses elementos podem ser combinados na análise sem scanner. *Parsers* podem ser programados manualmente ou podem ser gerados automaticamente ou semi-automaticamente por um gerador de *parser*.

Prettyprinting (pp) Na programação de computadores, *prettyprinting* é a aplicação de diversas convenções estilísticas de formatação a arquivos de texto, como código-fonte, marcação e conteúdos similares. Essas convenções de formatação podem incluir o uso de estilos de indentação, cores e tipos de fonte diferentes para destacar elementos sintáticos do código-fonte, ou ajustes de tamanho, para tornar o conteúdo mais fácil de ser lido e compreendido por pessoas. *Prettyprinters* para código-fonte são às vezes chamados de formatadores de código ou *beautifiers*.

S

S-expression (sexp) Na programação de computadores, uma *S-expression* (ou expressão simbólica, abreviada como *sexpr* ou *sexp*) é uma expressão em uma notação de mesmo nome para dados em lista aninhada (estruturados em árvore). As *S-expressions* foram inventadas e popularizadas pela linguagem de programação LISP, que as utiliza tanto para código-fonte quanto para dados.

W

Wildcard No software, um *wildcard* é um tipo de espaço reservado geralmente representado por um único caractere – como um asterisco (‘*’) – ou símbolo especial, que pode representar: (I) qualquer outro caractere; (II) um conjunto de caracteres; (III) objetos, elementos, símbolos, etc.; ou até (IV) nada, ou seja, nenhum caractere ou objeto, elemento, símbolo, etc. Os usos mais comuns de *wildcards* são: (I) em expressões regulares (*regex*), onde um caractere especial é usado para representar um conjunto de caracteres desconhecidos ou variáveis em uma string; e (II) na interface de linha de comando, onde as *shells* normalmente disponiblizam o caractere asterisco (‘*’) para corresponder a nomes de arquivos.